"Complejidad Computacional"
(Máster M & C)

Luis M. Pardo

1 de febrero de 2016

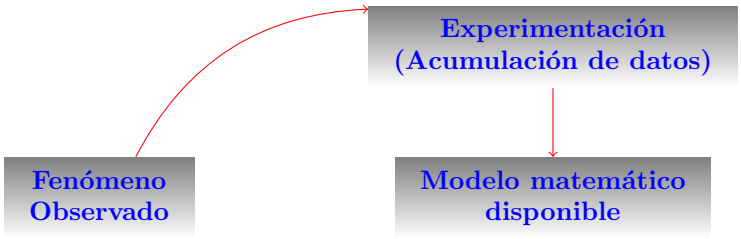
Prólogo

Lo que siguen son unas pocas notas para un curso de Máster, construidas con materiales usados en cursos de similares contenidos anteriormente impartidos. No están refinados como para ser presentados públiocamente pero sí pueden servir como material de apoyo al curso impartido en el año 2014/15.

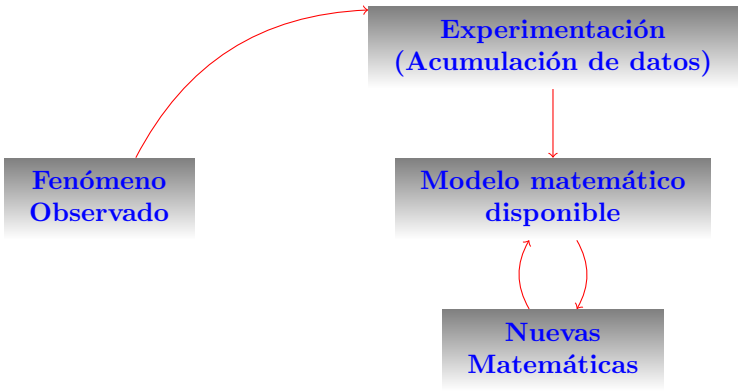
Como primera idea general, debe destacarse que la Informática es el suceso más destacado de la Ciencia y no por sus efectos sociales, sino por su estructura como Ciencia y sus orígenes. La Informática, como la lengua oral y escrita, es un fenómeno radicalmente humano, un ente que no hubiera existido de no existir los seres humanos y que, como la literatura y las matemáticas, no existen en la Naturaleza y sólo existen porque los seres humanos así lo han construido desde su abstracción. De hech, la Informática podría haber seguido existiendo sin ninguna realización (ordenador), aunque, en ese caso, su impacto social habría sido menor.

La Ciencia tradicionalmente parte del principio de la interpretación de la realidad. De entre todos los ámbitos de la Ciencia es la Física la que ha evolucionado más y ha alcanzado los niveles de conocimiento más profundos. La Física parte del principio de la interpretación de ciertos fenómenos que acontecen en la Naturaleza y que, con el paso del tiempo, han ido denominándose fenómenos físicos en contraposición a otros tipos de fenomenología de la Naturaleza. No es mi intención aquí delimitar esos fenómenos, pero sí tratar de exponer su procedimiento.

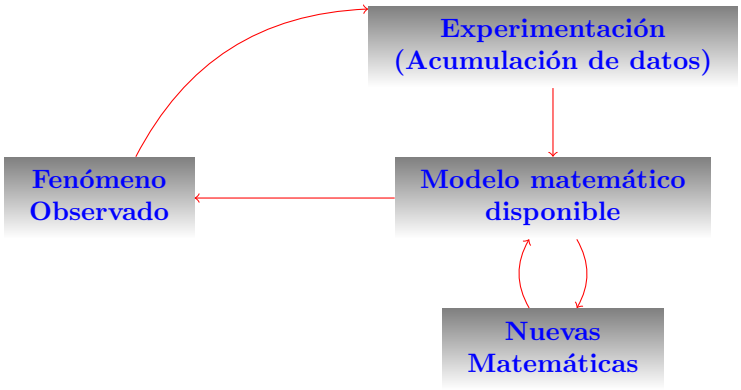
En primer lugar se observa un fenómeno (una piedra cae desde la Torre de Pisa), se toman datos de las observaciones y se tratan de usar las Matemáticas Disponibles para modelizar los resultados de la observación (a ésto se lo denominará *Teoría del Aprendizaje*, con el paso de los años). Galileo Galilei piensa en usar las pocas matemáticas que conoce y establece unas primeras leyes que relacionan velocidad, aceleración, distancia y tiempo. También toma cientos de datos de observaciones y comprueba que el único modelo matemático que se ajusta a sus datos es un modelo heliocéntrico. En esencia, Galileo es el primero en plantear el primer paso de la construcción de la física:



Cuando llega Newton, unos años más tarde, observa que puede interpretar esos mismos fenómenos de la mecánica mediante leyes aún más completas, pero de da cuenta de que necesita unas nuevas matemáticas para poder trabajar con ellas. Ese es el origen de la *Teor'ıa de Fluxiones* (hoy conocido como Análisis Matemático, aunque es un ámbito mucho más avanzado de lo propuesto por Newton). Por supuesto, las nuevas matemáticas inciden en el modelo que se va modificando conforme éstas evolucionan y producen sus consecuencias. Esto se refleja en el gráfico siguiente:



Con el modelo más evolucionado se vuelve a la exprimentación como elemento de contraste y se van confirmando las deducciones del modelo. Hasta que surge un nuevo *fenómeno observable*. En Física es el clásico experimento de Michelson y Morley sobre la velocidad constante de la luz. Este fenómeno obligará a modificar los principios de las Leyes de NEwton y darán lugar a la relatividad especial de Einstein y Poincaré y, con el tiempo, a la relatividad general de Einstein y Hilbert. El circuito queda, por tanto, establecido en un grafo como el que sigue:



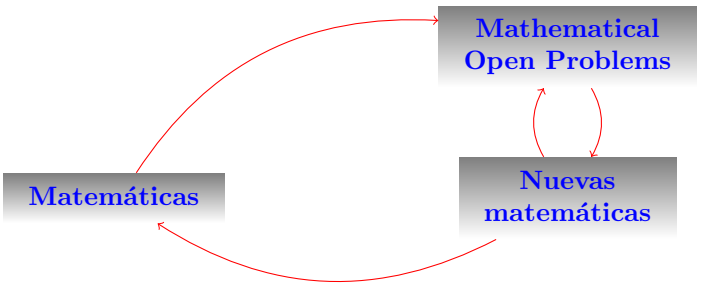
Este grafo de interacciones permite ir ajustando los modelos que describen la realidad física y surgen nuevas dificultades como la integración de las cuatro fuerzas de la naturaleza o la *Teoría del Todo*, tan querida por Einstein como difícil de alcanzar muchos años después.

La Informática corresponde a un modelo de Ciencia totalmente distinto, en la que las flechas aparecen orientadas en sentido contrario. La informática es excepcional con respecto a otros conocimienos científicos. Para explcarlo, analicemos su funcionamiento. Para empezar, el punto de partida no son los fenómenos observables sino las Matemáticas per se.

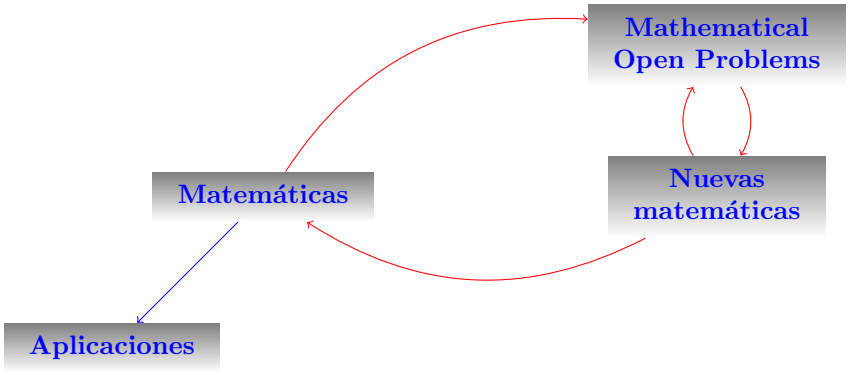
Las Matemáticas viven de sí mismas, de sus preguntas, de sus problemas y de las nuevas matemáticas que se desarrollan para resolver sus problemas. Ningún Matemático está (o debería estar) interesado en lo que sucede en otros ámbitos de la Ciencia. Cuando esto sucede algo extraño está sucediendo. Una manera de interpretar las grandes tendencias de la Matemática desde principios del siglo XX. Son las lista de problemas importantes a resolver. Así, la lista de *23 problemas propuestos por D. Hilbert en el International Congress of Mathematiians* del año 1900 en París, condicionaron mucha de la Matemática desarrollada a lo largo del siglo XX [1]. Lo

[1] Para más información véase http://en.wikipedia.org/wiki/Hilbert's_problems

mismo se puede decir para las Matemáticas del siglo XXI. Para ellas disponemos de dos grandes listas de problemas. De una parte, los *Millenum Problems*[2] propuestos por el Instituto Clay, de los cuales ólo hay uno resuelto a día de hoy. La otra gran lista son los *18 Problemas propuestos por el medalla Fields Stephen Smale para las Matemáticas del siglo XXI*[3], alguno de los cuales ya ha sido resuelto. Se pueden consultar esos Problemas en las páginas de Wikipedia indicadas y observar que casi ninguno tiene nada que ver con la Física o el reconocimiento y modelización de la realidad. Un grafo que puede representar las Matemáticas es el siguiente:

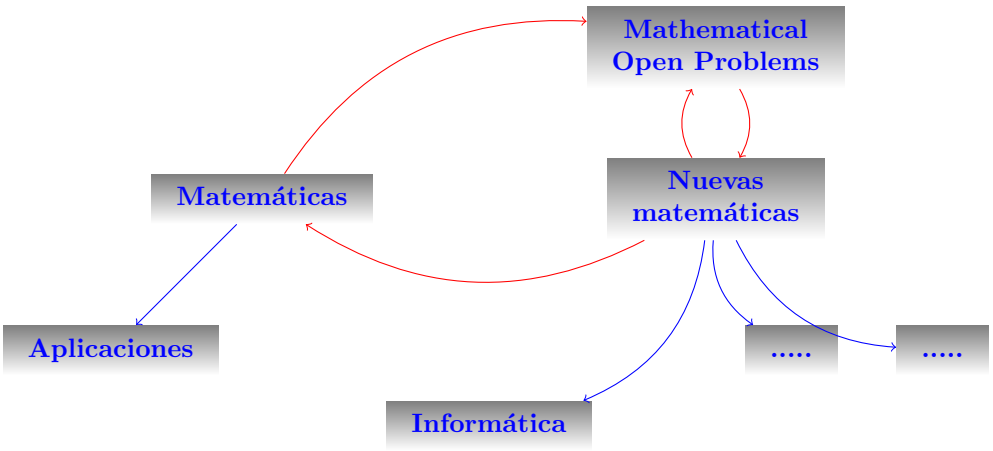


En ocasiones, las *Matemáticas dearrolladas tienen Aplicaciones* en ciertos ámbitos dstintos de ella misma. Esto se produce bien porque un Matemático modeliza un problema y lo resuelve o bien porque alguien de un ámbito distinto a la Matemática toma modelos de los conceptos Matemáticos para interpretarlos en su propio ámbito y usarlos para modelizar su fenómeno.

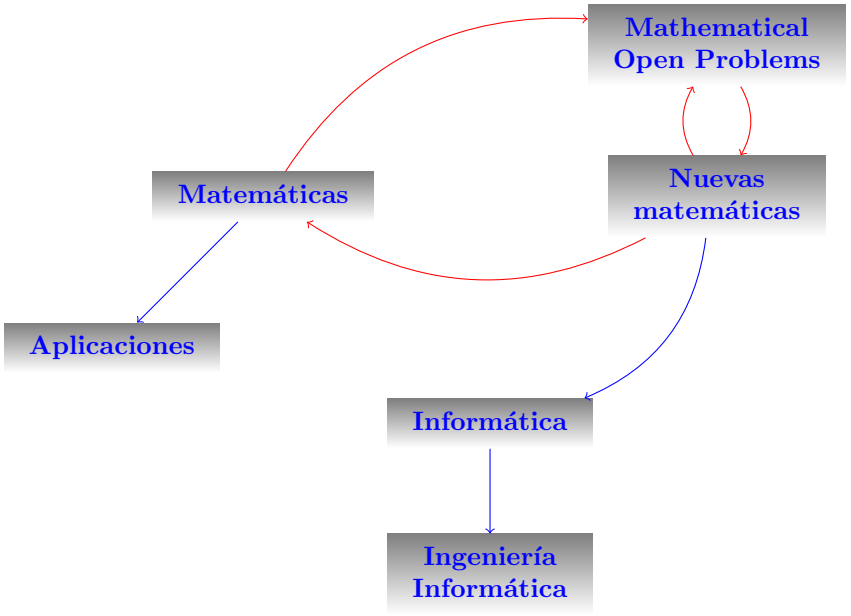


En la Informática se produce un fenómeno sorprendente para la Humanidad. En este caso, el desarrollo de Nuevas Matemáticas se transforma en la creación de un objeto inexistente en la Naturaleza hasta la fecha. Así las ideas abstractas de A. Turing dieron lugar, con la colaboración de un gran equipo de ingenieros, al primer ordenador físico (programable) conocido: *Colossus*. Del mismo modo, von Neumann comprende las ideas de A. Turing y genera el primer ordenador norteamericano conocido: *ENIAC*. Otros ámbitos podrían presentar nuevas generaciones de objetos inexistentes en la Naturaleza. Debe destacarse que es uno de los problemas prpuestos por Hilbert (el Problema X de Hilbert) el desencadenante de l defición de algoritmo y, con Turing, el modelo matemático transformable en realidad. **No es un modelo creado a partir de lo observado, sino una realidad creada a partir de un modelo**. Esto cambia completamente el papel de la Informática como ciencia.

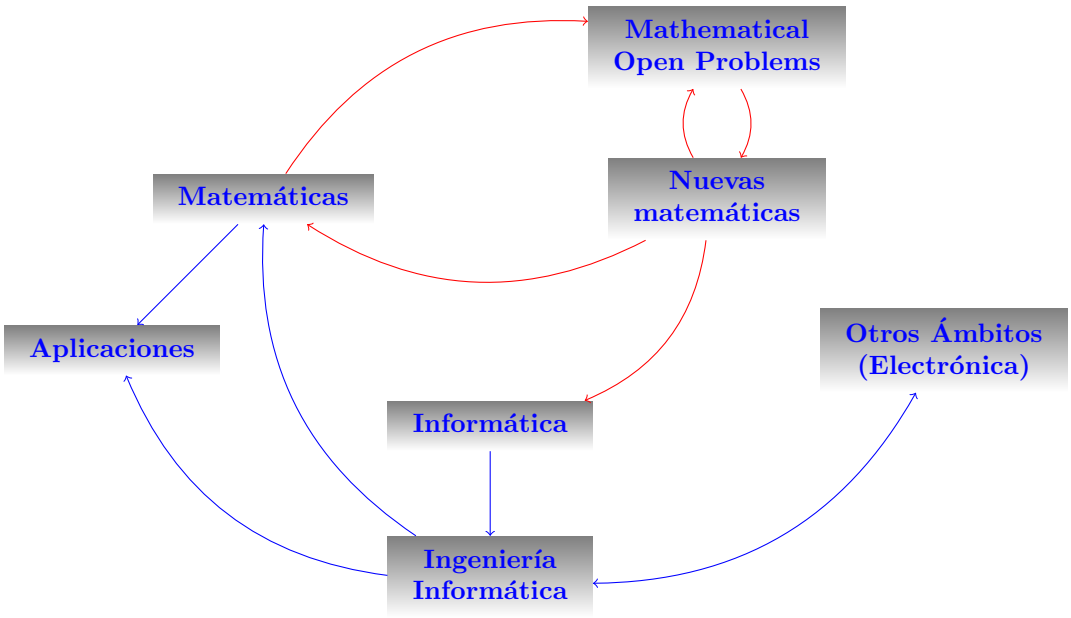
[2]Para más información véase http://en.wikipedia.org/wiki/Millennium_Prize_Problems
[3]Para más información véase http://en.wikipedia.org/wiki/Smale's_problems



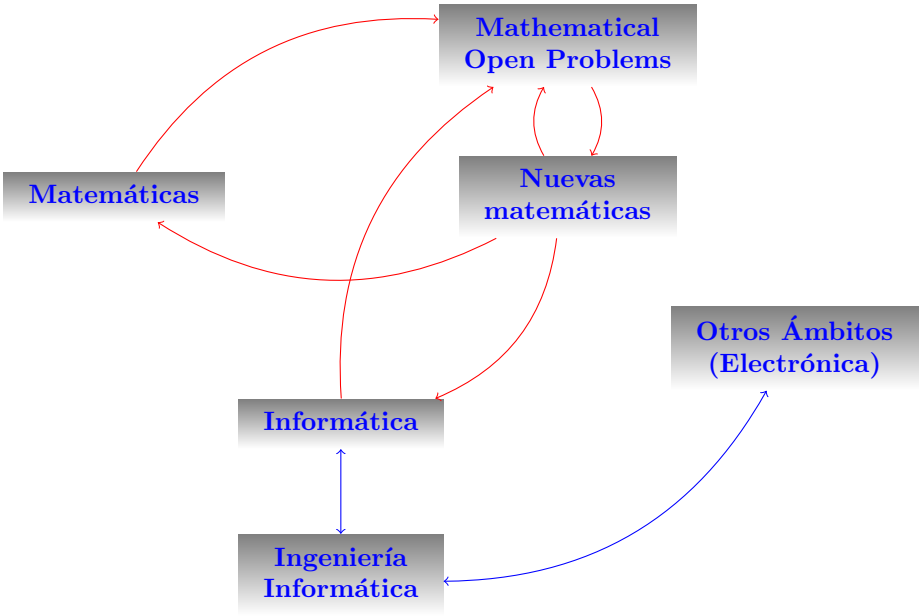
Suprimiremos los elementos de otras posibles creaturas matemáticas y nos quedamos sólo con la Informática. A partir de aquí la Informática, una vez que existe como realidad separada de la Matemática, genera la Ingeniería Informática.



A partir de este punto, se producen diversos tipos de interacciones. De una parte, la Ingeniría Informática propone problemas y retos a la electrónica, tanto en el desarrollo de arquitecturas de ordenador, como en el desarrollo de procesadores más eficientes. Esto genera una interacción bidireccional. De otro lado, la Ingeniería Informática, realiza abundantes aplicaciones a muchos otros ámbitos, muchas veces a través de la Matemática. Esta última interacción la hace muy propular y ha producido su difución en todos los ámbitos de la sociedad y la economía. Pero, como se observa, la popularidad social y económica de la Ingenier´ia Informática no es un elemento esencial, sino meramente accesorio, que tiene su ventaja en las salidas profesionales de quienes conocen estos elementos: los *ingenieros informáticos* que as´iobtienen empleos que antes no existían para ellos.



Pero, desde la perspectiva científica, no es el impacto social (es decir, las aplicaciones) lo que tiene carácter científico en Inform'atica. Así que, para dar más visibilidad a la parte científica, suprimo en la gráfica siguiente el asunto de las aplicaciones y nos centramos en la Ciencia y/o la Ingeniería Informática. Desde luego, la Interacción bidireccional entre Informática e Ingeniería es crucial, generando nuevos retos, preguntas y cuestiones. Pero, también la Informática genera nuevas preguntas fundamentales cuya respuesta viene de la generación de nuevos problemas, nuevos modelos matemáticos , nuevos resultados y, posiblemente, nuevos dearrollos informáticos. Así, por ejemplo, el problema de la factorización de números enteros (números en $\mathbb{Z}$) llevó a Schor a utilizar modelos matemáticos, basados en espacios de Hilbert, para la informática. Dieron en llamarlo Computación Cuántica porque eso es muy positivo en términos de publicidad. Sin embaro, el modelo se basa solamente en Álgebra Lineal elemental y manipulación de variables aleatorias (esto es, variables que llevan asignada una distribución de probablidad). Pero, desde entonces, no se ha podido construir ningún ordenador cuántico que desarrolle el formalismo abstracto de Shor. Lo mismo sucedión con el fracaso del paralelismo (los modelos de Nick Pippenger y Steve Cook nunca han podido ser realizados). Por otro lado, desarrrollos de modelos de algoritmos como los *evolutivos, genéticos o bio-inspirados, aprendizaje estadístico*, etc. se han alimentado de nociones matemáticas tan clśicas como los mínimos cuadrados (desarrollado por C.F. Gauss para predecir la óbita de Ceres). Así la Informática genera nuevos problemas abiertos para la Matemática que, a su vez, necesitan un tratamiento matemático e inundan este ámbito del conocimiento.



Lo que se pretende destacar en este último diagrama es que, a diferencia del ciclo de las Ciencias Experimentales, la Informática y las Matemáticas forman el centro (núcleo) del conocimiento n el ámbito de la Computación y, por tanto, son las técnicas de análisis, deducción y desarrollo de la Matemática quienes deben dar respuesta a las cuestiones propuestas por la Informática. Un poco como una madre y su hija interactúan, a pesar de los desarrollos un poco independientes de la "nieta" conocida como Ingeniería Informática.
En este curso las preguntas propuestas por la Informática a la Matemática hacen referencia a la Complejidad Computacional. Y, por tanto, se recomienda:

$$\acute{\alpha}\gamma\varepsilon\omega\mu\acute{\varepsilon}\tau\rho\eta\tau\omega\zeta\ \mu\eta\delta\varepsilon\acute{\iota}\zeta\ \varepsilon\iota\sigma\acute{\iota}\tau\omega$$
Ageōmétrētos mēdes eisítō.

O sea, hay que saber Matemáticas (o, al menos, adquirir información escrita en legunaje matemátio) antes de entrar en lo que sigue.

Índice general

1. Historia **15**
1.1. Introito . . . 15
1.2. La Matemática Griega y el Período Greco–Romano. . . . 16
1.2.1. De Atenas a Samarkanda. . . . 16
1.2.2. El Largo Período de Silencio de la Edad Media (cf. [Ca, 95]) . . . 17
1.2.3. La Matemática Musulmana . . . 17
1.2.3.0.1. Los traductores: Transmisión de estas ideas a Europa Occidental. . . . 17
1.3. Del Cinquecento a Descartes. . . . 18
1.4. El XVII y el XVIII. . . . 22
1.5. El siglo XIX . . . 24
1.6. La Aparición de la Noción de Algoritmo. . . . 27
1.6.1. Teorías Formales . . . 28
1.6.2. Turing, Gödel y Church . . . 29

I Los Algoritmos de los Lenguajes Formales **33**

2. Jerarquía de Chomsky **35**
2.1. Introducción . . . 35
2.2. Lenguajes Formales y Monoides . . . 38
2.2.1. Operaciones Básicas con palabras. . . . 39
2.2.2. Operaciones Elementales con Lenguajes . . . 39
2.2.3. Sistemas de Transición . . . 40
2.3. Gramáticas Formales . . . 41
2.3.1. Sistema de Transición Asociado a una Gramática. . . . 41
2.3.2. Otras Notaciones para las Producciones. . . . 42
2.3.2.1. Notación BNF. . . . 42
2.3.2.2. Notación EBNF. . . . 42
2.4. Jerarquía de Chomsky . . . 43
2.5. Disgresión: Problemas de Palabra . . . 44

3. Expresiones Regulares **47**
3.1. Las Nociones y Algoritmos Básicos . . . 47
3.1.1. Las Nociones . . . 47
3.1.2. La Semántica de las expresiones regulares. . . . 48
3.2. De RE's a RG's: Método de las Derivaciones . . . 49
3.2.1. Derivación de Expresiones Regulares . . . 49
3.2.2. Cómo no construir la Gramática . . . 51
3.2.3. Derivadas Sucesivas: el Método de las derivaciones . . . 52
3.3. De RG's a RE's: Uso del Lema de Arden . . . 54
3.3.1. Ecuaciones Lineales. Lema de Arden . . . 54
3.3.2. Sistema de Ecuaciones Lineales Asociado a una Gramática. . . . 55

3.4. Problemas y Cuestiones. 56
3.4.1. Cuestiones Relativas a Lenguajes y Gramáticas. 56
3.4.2. Cuestiones Relativas a Expresiones Regulares. 57
3.4.3. Problemas Relativos a Lenguajes Formales y Gramáticas 57
3.4.4. Problemas Relativos a Expresiones Regulares 59

4. Autómatas Finitos **61**
4.1. Introducción: Correctores Léxicos o Morfológicos 61
4.2. La Noción de Autómata 62
4.2.1. Sistema de Transición de un autómata: 63
4.2.1.1. Representación Gráfica de la Función de Transición. 64
4.2.1.2. Lenguaje Aceptado por un Autómata 65
4.3. Determinismo e Indeterminismo 66
4.3.1. El Autómata como Programa 66
4.3.2. Autómatas con/sin $\lambda-$Transiciones. 66
4.3.2.1. Grafo de $\lambda-$transiciones. 67
4.3.3. Determinismo e Indeterminismo en Autómatas 68
4.4. Lenguajes Regulares y Autómatas. 68
4.4.1. Teorema de Análisis de Kleene 68
4.4.2. Teorema de Síntesis de Kleene 69
4.5. Lenguajes que no son regulares 71
4.5.1. El Palíndromo no es un Lenguaje Regular. 74
4.6. Minimización de Autómatas Deterministas 75
4.6.1. Eliminación de Estados Inaccesibles. 75
4.6.2. Autómata Cociente 75
4.6.3. Algoritmo para el Cálculo de Autómatas Minimales. 76
4.7. Disgresión: Los autómatas finitos como cadenas de Markov 78
4.7.1. Markov Chains 78
4.7.2. Adjacency Matrix 78
4.7.3. Finite Automata as Markov Chains 78
4.7.4. Probabilistic Finite Automata 78
4.8. Cuestiones y Problemas. 78
4.8.1. Cuestiones. 78
4.8.2. Problemas 80

5. Libres de Contexto **83**
5.1. Introducción 83
5.2. Árboles de Derivación de una Gramática 85
5.2.1. Un algoritmo incremental para la vacuidad. 86
5.3. Formas Normales de Gramáticas. 87
5.3.1. Eliminación de Símbolos Inútiles o Inaccesibles 87
5.3.1.1. Eliminación de Símbolos Inaccesibles. 88
5.3.1.2. Eliminación de Símbolos Inútiles. 89
5.3.2. Transformación en Gramáticas Propias. 90
5.3.2.1. Eliminación de $\lambda-$producciones. 90
5.3.2.2. Eliminación de Producciones Simples o Unarias 91
5.3.2.3. Hacia las Gramáticas Propias. 92
5.3.3. El Problema de Palabra para Gramáticas Libres de Contexto es Decidible. 94
5.3.4. Transformación a Formal Normal de Chomsky. 94
5.3.5. Forma Normal de Greibach 96
5.4. Cuestiones y Problemas 96
5.4.1. Cuestiones 96
5.4.2. Problemas 96

6. Autómatas con Pila **99**
6.1. Noción de Autómatas con Pila. . . . 99
6.1.1. Las Pilas como Lenguaje (Stacks). . . . 99
6.2. Sistema de Transición Asociado a un Autómata con Pila. . . . 102
6.2.1. Modelo gráfico del sistema de transición. . . . 102
6.2.2. Transiciones: Formalismo. . . . 102
6.2.3. Codificación del Autómata con Pila. . . . 104
6.3. Lenguaje Aceptado por un Autómata con Pila. . . . 106
6.4. Equivalencia con Gramáticas Libres de Contexto. . . . 110
6.5. Propiedades Básicas 112
6.6. Problemas 114
6.6.1. Problemas 114

7. Introducción a Parsing **119**
7.1. Introducción 120
7.1.1. El problema de parsing: Enunciado 122
7.2. Compiladores, Traductores, Intérpretes 122
7.2.1. Traductores, Compiladores, Intérpretes 123
7.2.1.0.1. Ventajas del Intérprete. . . . 123
7.2.1.0.2. Inconvenientes de los Intérpretes. . . . 123
7.2.1.1. Compiladores Interpretados. . . . 123
7.2.2. Las etapas esenciales de la compilación. . . . 124
7.2.2.1. La Compilación y su entorno de la programación. . . . 124
7.2.2.2. Etapas del Proceso de Compilación. . . . 124
7.2.2.3. En lo que concierne a este Capítulo. . . . 124
7.3. Conceptos de Análisis Sintáctico 124
7.3.1. El problema de la Ambigüedad en CFG 125
7.3.2. Estrategias para el Análisis Sintáctico. . . . 126
7.4. Análisis CYK 128
7.4.1. La Tabla CYK y el Problema de Palabra. . . . 128
7.4.2. El Árbol de Derivación con las tablas CYK. . . . 130
7.4.3. El Algoritmo de Análisis Sintáctico CYK 130
7.5. Traductores Push–Down. . . . 131
7.5.0.1. Sistema de Transición asociado a un PDT. . . . 132
7.6. Gramáticas $LL(k)$: Análisis Sintáctico 133
7.6.1. FIRST & FOLLOW 133
7.6.2. Gramáticas $LL(k)$ 137
7.6.3. Tabla de Análisis Sintáctico para Gramáticas $LL(1)$ 139
7.6.4. Parsing Gramáticas $LL(1)$ 140
7.7. Cuestiones y Problemas 142
7.7.1. Cuestiones 142
7.7.2. Problemas 143

8. Gramáticas Sensibles al contexto **147**
8.1. Introducción 147
8.2. Lenguajes Sensibles al contexto 147
8.3. Autómatas Linealmente Acotados 147
8.3.1. Kuroda's Problems 148
8.3.2. El Problema de Palabra para Gramáticas Sensibles al contexto 149

II Las Ideas de Alan Turing 151

9. Máquinas de Turing 153
9.1. Introducción . . . 154
9.2. La Noción de Máquina de Turing . . . 155
9.2.1. Dinámica de una Máquina de Turing: El Modelo Gráfico y el Sistema de Transición. . . . 156
9.2.1.1. Modelo gráfico de una máquina de Turing. . . . 157
9.2.1.2. Un paso de cálculo: . . . 158
9.3. Algoritmos, funciones computables. Lenguajes Recursivamente enumerables . . . 160
9.4. Funciones y Clases de Complejidad . . . 161
9.5. Rudimentos con Máquinas de Turing . . . 162
9.5.1. La independencia del alfabeto. . . . 162
9.5.1.1. Primera Parte : Rechazar las Palabras que no están en la Imagen. 165
9.5.1.2. Segunda Parte : Simulación de Un Paso (SimUP). . . . 166
9.5.1.2.1. Definición del Proceso de Simulación de un Paso. . . . 167
9.5.2. Los conjuntos recursivamente enumerables como conjuntos imagen, indecidibilidad y otras interpretaciones . . . 172
9.5.3. Independencia del Número de Cintas . . . 176
9.6. La máquina Universal de A. Turing. . . . 177
9.6.1. El código de una máquina de Turing. . . . 177
9.6.1.1. El código de una máquina de Turing sobre el alfabeto Universal. 177
9.6.2. La máquina Universal : ejemplo de compilador e intérprete. . . . 179
9.6.2.1. El código de una configuración de una máquina de Turing dada por su código. . . . 179
9.6.3. El problema de la Parada y cuestiones conexas. . . . 185
9.7. El Problema de la Parada. . . . 185
9.8. El final del Problema X de Hilbert . . . 186
9.9. Disgresión: Problemas de la Palabra . . . 189
9.10. Números reales recursivamente enumerables. . . . 191
9.11. Tape Compression Lemma y Linear Speed-Up . . . 193
9.11.1. Tape Compression Lemma . . . 193
9.11.1.0.1. Intento pormenorizado de describir la máquina del Tape Compression Lemma. . . . 196
9.11.2. Linear Speed–Up. . . . 200

10. Euclides y Ecuaciones Diofánticas 203
10.1. Caso $n = 2$. . . . 205
10.2. Caso $n \geq 3$. . . . 212
10.2.0.1. Divisiones–Reducciones. . . . 212

III Algunos Rudimientos con la Complejidad Computacional 217

11. Clases de Complejidad y Primeras Relaciones 219
11.1. Terminología Básica. . . . 219
11.2. El indeterminismo en juego. . . . 220
11.2.1. Clases de complejidad funcionales . . . 223
11.3. Mezclando Clases. . . . 227
11.3.1. Teoremas de Jerarquía. . . . 227
11.3.2. Unas palabras sobre grafos orientados. . . . 232
11.3.2.1. Clausura Transitiva . . . 233
11.3.2.2. Alcanzable . . . 234
11.3.3. Una codificación más corta de las configuraciones . . . 234
11.3.4. Espacio indeterminista frente a tiempo determinista. . . . 235

11.3.5. Tiempo indeterminista frente a espacio determinista. 236
11.4. El Teorema de Savitch. 236
11.5. Un pequeño grafo con las relaciones conocidas. 237

12. Clases Centrales de Complejidad. **239**
12.1. Introducción 239
12.1.1. Tesis de Cobham-Edmonds: a partir de unos experimentos teóricos 239
12.1.2. Clases Centrales de Complejidad. 241
12.2. La clase NP. 241
12.2.1. Ejemplos naturales de problemas indeterminísticos. 243
12.2.2. Ejemplo: Primalidad y el Teorema de Pratt. 244
12.2.2.0.1. Criba de Eratóstenes (s. III a. de C.) : 244
12.2.3. El Teorema de Pratt :PRIMES $\in$ **NP** 244
12.2.4. Máquinas con Oráculos 250
12.3. El Cálculo Proposicional y su Cálculo de Predicados 251
12.3.0.1. El Cálculo Proposicional : Fórmulas Booleanas. 251
12.3.0.1.1. La Sintaxis. 251
12.3.0.1.2. Semántica. 254
12.4. **NP**−completitud : Teoremas de Cook y Karp. 258
12.4.1. Reducciones 259
12.4.1.1. Problemas de Búsqueda (Search Problem). 259
12.4.1.2. Clausura bajo reducciones 260
12.4.2. El Teorema de Cook: Problemas **NP**–completos. 261
12.4.3. El Teorema de Ladner 268
12.5. La clase **PSPACE** 269
12.5.1. Problemas **PSPACE**-completos 269
12.5.2. La Jerarquía Polinomial **PH** 271
12.6. Un grafo final 272

13. Algoritmos Probabilistas **275**
13.1. Clases de Algoritmos Aleatorios : **BPP**, **RP**, **ZPP** 275
13.2. La clase **P**/poly: circuitos booleanos 278
13.3. Una disgresión : el Sistema Criptográfico RSA. 279
13.3.1. Criptología 279
13.3.1.1. Contexto de la Teoría de la Información : 280
13.3.2. Criptografía y Criptoanálisis. 281
13.3.3. El Sistema Criptográfico RSA. 281
13.3.3.1. El sistema RSA : Definición de Clave Pública : 281
13.3.4. El sistema RSA : Codificación de un mensaje 281
13.3.5. El sistema RSA : Descodificación de un mensaje 281
13.4. Test de Primalidad de Miller-Rabin: COMPUESTO $\in$ **RP** 282
13.4.0.0.1. El grafo $K(2^\ell)$. 286
13.4.1. La búsqueda de primos de talla dada y otras cuestiones. 291
13.4.2. Comentarios sobre Criptoanálisis :Factorización. 293
13.4.2.1. Comentario Preliminar. 293
13.4.2.2. Algoritmos de Factorización basados en Fermat. 293
13.4.2.3. El método ρ de Pollard. 294
13.5. Reciprocidad Cuadrática: El algoritmo de Solovay-Strassen 295
13.5.0.1. Símbolo de Lengendre. Criterio de Euler 295
13.5.1. La demostración de Einsenstein 297
13.5.1.0.1. Una Construcción Geométrica 297
13.5.1.0.2. Una Reflexión actuando sobre el rectángulo R: 298
13.5.2. El Lema de Gauss y las raíces cuadradas modulares de 2 299
13.5.3. El símbolo de Jacobi 299
13.5.4. El Test de Solovay-Strassen 301

13.6. Tests de Nulidad para Polinomios. 303
13.6.1. El Test de Schwartz–Zippel. 303
13.6.2. Cuestores. 305
13.6.3. Witness Theorem. 306
13.6.4. Tests de Nulidad para Números Dados por Esquemas de Evaluación. . . . 307

IV Algunas ideas más sutiles **309**

A. Teoría Intuitiva de Conjuntos **311**
A.1. Introducción 311
A.2. Conjuntos. Pertenencia. 311
A.2.1. Algunas observaciones preliminares. 312
A.3. Inclusión de conjuntos. Subconjuntos, operaciones elementales. 312
A.3.1. El retículo $\mathcal{P}(X)$. 313
A.3.1.1. Propiedades de la Unión. 313
A.3.1.2. Propiedades de la Intersección. 314
A.3.1.3. Propiedades Distributivas. 314
A.3.2. Leyes de Morgan. 314
A.3.3. Generalizaciones de Unión e Intersección. 314
A.3.3.1. Un número finito de uniones e intersecciones. 314
A.3.3.2. Unión e Intersección de familias cualesquiera de subconjuntos. . 314
A.3.4. Conjuntos y Subconjuntos: Grafos No orientados. 315
A.4. Producto Cartesiano (`list`). Correspondencias y Relaciones. 315
A.4.1. Correspondencias. 316
A.4.2. Relaciones. 317
A.4.2.1. Relaciones de Orden. 318
A.4.2.2. Relaciones de Equivalencia. 319
A.4.3. Clasificando y Etiquetando elementos: Conjunto Cociente. 319
A.5. Aplicaciones. Cardinales. 320
A.5.1. Determinismo/Indeterminismo. 321
A.5.2. Aplicaciones Biyectivas. Cardinales. 323

Capítulo 1

Algunas Notas Históricas sobre Ecuaciones, Polinomios, Números y Algoritmos.

Contents

1.1. Introito **15**
1.2. La Matemática Griega y el Período Greco–Romano. **16**
1.2.1. De Atenas a Samarkanda. 16
1.2.2. El Largo Período de Silencio de la Edad Media (cf. [Ca, 95]) 17
1.2.3. La Matemática Musulmana 17
1.3. Del Cinquecento a Descartes. **18**
1.4. El XVII y el XVIII. **22**
1.5. El siglo XIX **24**
1.6. La Aparición de la Noción de Algoritmo. **27**
1.6.1. Teorías Formales 28
1.6.2. Turing, Gödel y Church 29

1.1. Introito

Las páginas que siguen son un esbozo de la extraordinaria ligazón que ha ido trabando históricamente el Algebra, la Teoría de Números y los Fundamentos de la Informática. Si bien muchos de estos campos aparecen desligados en la Ciencia hiper–especializada de nuestros días, creo que vale la pena insistir en esas coincidencias. En ellas verán el devenir histórico como un progreso y podrán percibir la esencia de una Matemática viva y en evolución. No es tan importante el texto particular, sino trasladar la impresión de ciencia viva y activa que no se consigue (o se consigue difícilmente) mostrando una estructura acabada y cerrada.
El interés de un Capítulo como éste no es sólo de carácter didáctico, sino que también tiene mucho que ver con mostrar los nombres de los hombres que iniciaron muchos de los asuntos que luego se tratarán a lo largo del curso. Si acaso, conocer las raíces, recordar, siempre es útil para entender hacia dónde se va.
Dividimos este resumen histórico en dos partes, fuertemente relacionadas: La primera considera la cuestión de la resolución de ecuaciones polinomiales. Tratando de mostrar cómo se las ingenieban para "resolver" mediante el desarrollo de cálculos cuyo sentido algebraico es indiscutible. Observamos cómo el cálculo literal (ya implícito en Bombelli y explícito en Descartes) es un prototipo del cálculo simbólico. En la segunda parte, retomamos el surgimiento de la noción de

algoritmo desde el problema X de Hilbert hasta la Tesis de Church. Módulo algunas disquisiciones menores, tratamos de recorrer los modelos introducidos por aquellos que se involucraron en su desarrollo en la década de los treinta.

1.2. La Matemática Griega y el Período Greco–Romano.

1.2.1. De Atenas a Samarkanda.

La historia de los Fundamentos de la Informática y del Algebra han corrido parejas a lo largo del tiempo. Si bien los matemáticos hindúes y chinos llevan la delantera en lo concerniente a la aritmética, no es hasta la matemática griega que se produce el efecto de formalización y corrección en los enunciados y demostraciones que sustentarán el formalismo matemático que conocemos. El primer griego cuyos conocimientos más influyen en la matemática posterior es Anaxágoras. Recordemos que Anaxágoras es un griego de Asia Menor, que se ha formado bajo la influencia de las matemáticas orientales y llega a Atenas en la madurez. La primera cosa que asombra a Anaxágoras a su llegada a Atenas es que los griegos desconocen el cero. Más exactamente, los griegos con incapaces de realizar pequeñas operaciones aritméticas como una sencilla multiplicación. A la sazón, hombres de la talla de Sócrates o Platón no sienten vergüenza en reconocer que no saben multiplicar; aunque son muy capaces de alcanzar altísimas cimas en ciencias especulativas como la filosofía. La razón hay que buscarla en la falta de un sistema de representación numérica eficiente, lo que lleva al subdesarrollo aritmético y calculatorio. Anaxágoras cambia esta situación formando a varios jóvenes. Sin embargo, no quedan textos escritos de Anaxágoras; aunque sí queda el texto de uno de sus alumnos : *Euclides*. Partamos, pues, de Euclides para comenzar esta historia. En los ELEMENTOS de Euclides ya encontramos una cierta profusión de algoritmos fuertemente entroncados con el Algebra Computacional :

- El algoritmo de Euclides para el cálculo del máximo común divisor,
- La criba de Eratóstenes para la determinación de números primos,
- Algoritmos de reducción en sistemas de ecuaciones lineales,
- Cálculos de aproximaciones de raíces cuadradas,
- Construcciones con regla y compás (dibujo de un hexágono regular o el método del pastor para el dibujo de elipses a partir de sus focos).

Uno de los resultados más notables del período helenístico es el descubrimiento de la existencia de números irracionales. La ecuela Pitagórica atribuye a Hyppasus (s. V a.C.) en descrubimiento de que $\sqrt{2} \notin \mathbb{Q}$ y una prueba del mismo puede reencontrarse en el libro X de Euclides, así como la presencia de otros números irracionales que Euclides atribuye a Teeteto ($\Theta\epsilon\alpha\acute{\iota}\tau\eta\tau o\zeta$ - Zeaitetós, 417–369 a.C.).
No es menos destacable en este período el llamado *Método de Herón de Alejandría* (10-70, d.de C.)(que, en realidad, ya era conocido por los Babilonios y que antecede a Newton en 1600 años). Dada la ecuación $X^2 - a = 0$, una aproximación se obtiene eligiendo un punto inicial x_0 "próximo" a la raíz y aplicando la iteración:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right).$$

Se trata de hallar la media aritmética entre x_n y a/x_n lo que, en realidad, es el método de Newton porque:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^2 - a}{2x_n}.$$

1.2.2. El Largo Período de Silencio de la Edad Media (cf. [Ca, 95])

Tras la caída del Imperio Romano (s. V), se produce una enorme pérdida del conocimiento grecoromano. Durante los siglos V y VI, este conocimiento se salvará en Irlanda. Desde la llegada, en 432 y por accidente, del monje que llevará el cristianismo a Irlanda (St. Patrick),éste se expande lentamente, sobre todo a través de eremitas y pequeños monasterios. Irlanda es un territorio aislado de la Europa continental (no en vano ha preservado sus orígenes de celtas españoles y su idoma gaélico) y permanece aislado durante los siglos V y VI de las sucesivas invasiones de tribus bárbas de origen germánico. En esa tierra, fructifica la idea de un difusor del cristianismo Columbanus (543-615) quien, durante su vida, estableción entre 60 y 100 monasterios a lo largo de la tierre irlandesa. En esos monasterios se guarda parte de la bibliografía clásica y se copian esos textos, haciéndolos resistentes al paso del tiempo. El cristianismo de la isla y su preservación de la documentación clásica, les permitirá desembarcar ya en el siglo IX como los filósofos irlandeses que llegan a la Europa continental, con sus textos clásicos bajo el brazo. Esto salvó parcialmente la obra de Platón y, sobre todo, Aristóteles. El salto es significativo si observamos que San Agustín (354-430) es el filósofo de la decadencia del Imperio Romano, conocedor de las obras clásicas y ligeramente tendente al platonismo, mientras que Santo Tomás de Aquino (1224-1274) es el fundado del escolasticismo, un filósofo más tendente a Aristóteles, y que ha recibido el beneficio de los textos salvados de la caída po acciones como la de los monjes irlandeses.
Sin embargo, la principal contribución al salvamente de la cultura grecorromana no viene de Irlanda sino del mundo musulmán y, muy especialmente, la matemática.

1.2.3. La Matemática Musulmana

Sin embargo, la matemática moderna y la computación, deben mucho a la matemática musulmana. La matemática árabe se encuentra a caballo entre la cultura helenística y las culturas matemáticas hindúes y chinas. Así, mientras Europa se hunde en el abandono cultural causado por el hundimiento del Imperio Romano, es en el mundo árabe donde sobrevive la obra de Euclides [1] .
En la primera mitad del siglo IX, el matemático uzbeko Muhammad ibn–Musa Al–Juaritzmi escribió su tratado *"Hisab al–jabr wa–al–muqabala"* (traducido libremente por Libro (o Tratado) sobre las operaciones "jabr" (restablecimiento) y "qabala" (reducción)). Los científicos europeos comenzaron a conocer el Algebra a principios del siglo XII justamente a partir de traducciones al latín de los trabajos de Al-Khwuarizmi. Como anécdota, nótese que aparecen conectados dos grupos de fonemas que hoy son de uso común "al–jabr" (álgebra) y "al–Juaritzmi" (algoritmo). Y a la exposición de métodos de manipulación de números y ecuaciones estaba dedicado este tratado. No se trata de una obra original, sino de un compendio del conocimiento combinado de las matemáticas helenísticas y la teoría de números conocida en la India. El libro está fundamentalmente dedicado a la resolución sistemática de ecuaciones de primer y segundo grado, ciencia que se considera independiente. Así son resueltas, por ejemplo, las siguientes clases de Ecuaciones

$$ax = b, \quad x^2 + bx = a; \quad ax^2 = b$$

$$x^2 + a = bx; \quad ax^2 = bx; \quad bx + a = x^2$$

Pensemos que aún no se usan los números enteros, que serán una aportación de las matemáticas del Renacimiento. Otra obra de Juaritzmi (traducido por guarismo esta vez) *"Sobre los números hindúes"*, transfiere a las matemáticas europeas la representación de los números enteros en base decimal.

1.2.3.0.1. Los traductores: Transmisión de estas ideas a Europa Occidental. La transferencia de este conocimiento se hará en España. Así, el británico, residente en Espña, Robert de Chester, traducirá en Segovia, en 1145, la obra de Al-Khwaurizmi como "Liber Algebra et

[1] Bajo la forma de documentos como la traducción al árabe de A.S. Saidan. *"The Arithmetic of al–Uqlîdisî"*. Dordrecht, D. Reidel (1975). Una edición comentada de la obra de Euclides.

Almucabola"[2]. En Toledo, Gerardo de Cremona (1114-1187), italiano esta vez, traducirá al latín los "Elementos" de Euclides, desde su correspondiente traducción árabe, y Johannes Hispaliensis (Juan de Sevilla) introducirá su "Liber Algorithmi Practica Arithmetica", que introducirá en Europa Occidental el uso del sistema de nuemración hindú (traducido de Abu ibn Aslam) en el entorno de 1550. Posteiorment, en 1202, Fibonacci (Leonardo Pisano, 1170-1250), escribirá su "Liber Abacii", que significará el establecimiento y la divulgación por toda Europa de la condificación de los enteros con digitos en $\{0,\ldots,9\}$, dados por su posición y de los números racionales en la forma $\frac{m}{n}$.
Destaquemos algunos de los métodos transmitidos por este escuela árabe de la aritmética :

- Obtención de hasta 17 cifras de π (Khan, s. XV). Obsérvese que Fran çois Viéte (Vieta una vez castellanizado) obtiene sólo 9 cifras a finales del XVI y principios del XVII.
- Cálculo de raíces por el método conocido hoy como Ruffini–Horner. El método se describe como sigue : Supongamos que queremos calcular las cifras de $\sqrt[r]{q} = a, b, c, ...$ Se observa que está relacionado con la búsqueda de la sucesión de diferencias :

$$q = a^n, q - \left(a + \frac{b}{10}\right)^n, q - \left(a + \frac{b}{10} + \frac{c}{100}\right)^n, ...$$

- Expresión del desarrollo binomial $(a+b)^n$, con la relación entre coeficientes binomiales

$$\binom{m}{n} = \binom{m}{n-1} + \binom{m-1}{n-1}$$

- Suma de progresiones geométricas del tipo $\sum_{i=1}^{n} a^k$. Recuérdese la anécdota del matemático árabe jugador de ajedrez, que consigue $2^{64} - 1$ granos de trigo.

Pero quizá lo más destacable sea su preservación de las matemáticas griegas y. muy especialmente, de los *"Elementos"* de Euclides, obra conocida y comentada por muchos de sus autores.

1.3. Del Cinquecento a Descartes.

El Algebra y los Fundamentos de la Informática continúan su camino juntos a lo largo del tortuoso devenir de la historia de la resolución de ecuaciones polinomiales. Señalemos algunos de los hombres clave en este devenir.
Es al Renacimiento italiano a quien hay que darle el mayor crédito en este período. En el Cinquecento italiano (s. XVI) se produce el raro fenómeno de los torneos públicos de resolución de ecuaciones polinomiales. En ellos destacó *Scipione del Ferro* (profesor de matemáticas de la Universidad de Bolonia (en Italia) entre los años 1496 y 1526) quien afirmaba tener un método para la búsqueda de una raíz positiva de las ecuaciones de la forma $x^3 + px = q$. Mantuvo en secreto su método. Al final de sus días comunicó su secreto a su pariente y heredero Annibal della Nave y a su alumno *Fiore*. Fiore era un petulante y dejó correr por Bolonia la voz de que sabía resolver ecuacione cúbicas. Estos rumores llegaron a Niccolo Fontana (1500-1557), lo que provocón un duelo público en Bolonia.
A comienzos del año 1535 debía celebrarse un duelo público entre Fiore y *Niccolo Fontana "Tartaglia"* (1500–1557). Estos duelos para la resolución de problemas matemáticos eran muy similares a los concursos de televisión con preguntas y respuestas, con premios subvencionados por algún "sponsor" en forma de mecenas (la república de Florencia, por ejemplo, era muy dada a financiar este tipo de competiciones). Trataglia era un matemático de origen humilde que se ganaba la vida dando clase de matemáticas y mecánica en las ciudades del Norte de Italia. Conociendo que Fiore poseía la fórmula de Ferro, Tartaglia trató de reconstruirla por sí mismo.

[2]De hecho, la aparición de la letra "a" en el título es una especia de error de transcripción. En el original árabe, debería haber sonado como "al-jbr", pero Robert de Chester añadirá una "a" para pronunciarlo como "aljbar", que permanecerá en el tiempo, hasta la actualidad.

Y lo consiguió, según citan ciertas crónicas, en la víspera de la disputa celebrada el 12 de febrero de 1535. La justa consistía en que cada concursante ofrecía al otro una lista de 30 problemas y luego se darían entre 40 y 50 días para resolverlos. Tartaglia le propuso a Fiore una lista de problemas con gran variedad y Fiore, que era un matemático mediocre, nu pudo resolverlos. Fiore, en cambio, propuso a Tartaglia una serie de ecuaciones cúbicas y Tartaglia las resolvió todas en 2 horas. Algunas fuentes, sin embargo, discuten si no robó él mismo la fórmula bien de la fuente de Annibal della Nave o del propio Fiore.
El método de Tartaglia se describe como sigue :
Supongamos que la ecuación $x^3+px=q$ tiene una solución del tipo $x=\sqrt[3]{u}-\sqrt[3]{v}$. Sustituyendo esta expresión en la ecuación, obtenemos $p=3\sqrt[3]{uv}$. Obtenemos así el sistema :

$$u-v=q, \quad p^3=\frac{uv}{27}$$

Tenemos así descirtos u y v como raíces de una ecuación de grado 2 que resolvemos por el método usual. Finalmente, podemos resolver la ecuación mediante :

$$u:=\sqrt{\left(\frac{q}{2}\right)^2+\left(\frac{p}{3}\right)^2}+\frac{q}{2}$$

$$v:=\sqrt{\left(\frac{q}{2}\right)^2+\left(\frac{p}{3}\right)^2}-\frac{q}{2}$$

Posteriormente, Tartaglia consiguió resolver la ecuación de la forma $x^3=px+q$ con la sustitución $x=\sqrt[3]{u}+\sqrt[3]{v}$ y las ecuaciones de la forma $x^3+q=px$ reduciéndolas al tipo anterior. Pero Tartaglia nunca escribió sus métodos, guardando el secreto para posteriores torneos.
La siguiente etapa en la resolución de ecuaciones polinomiales por métodos algorítmicos pasa a la figura de *Cardano* (1501–1539). Comienza a estudiar la resolución de las ecuaciones cúbicas en 1539. Su figura es una de esas que destacan en la historia de las matemáticas, sobre todo por la influencia que tuvo su libro "Artis Magnae sive de regulis algebraicis" *"El gran arte de las reglas del Algebra"* (1545). Cardano es el primero en utilizar la palabra Algebra para designar los algoritmos de resolución de ecuaciones polinomiales y sistemas de ecuaciones lineales. Cardano era un hombre rico, instruido y talentoso, aficionado a la física y la matemática, a la filosofía, la medicina o la astrología. Al oír hablar del descubrimiento de Tartaglia, trató de ponerse en contacto con él para conseguir que le comunicara el método e incluirlo en su libro. Al final lo consiguió. Cardano reescribió la suma de métodos de Trataglia, resolviendo la ecuación general de grado tres : x^3+ax^2+bx+c. La desaparición del término en x^2 se conseguía con el método elemental de sustituir $x:=x_1+h$ y determinar h para que la nueva ecuación de grado 3 en x_1 no contuviera término cuadrático. Obsérvese que se trata de resolver una simple ecuación lineal.
Cuando supo de la aparición del libro de Cardano, Tartaglia montó en cólera y se dedicó a lanzarle retos a Cardano y a su alumno Lodovico Ferrari (1522-1565). El debate inflamó a toda Italia y Ferrari retó a Tartaglia a un duelo público de resolución de ecuaciones cúbicas y cuárticas. Tartaglia, que nunca tuvo mucho dinero, se vió obligado a concurrir al duelo si quería obtener un puesto de trabajo en su tierra natal (Brescia). El 10 de agosto de 1548, el esperado debate tuvo lugar en la iglesia y los jardines de Frati Zoccolanti en Milan. Lo ganó Ferrari, más joven y más diestro en los métodos que habí avanzado con Cardano, pero basados en los métodos del propio Tartaglia. Fontana tuvo que retirarse vencido esta vez.
El libro de Cardano no hay que entenderlo como un libro actual, como una disquisición con el esquema de Definición, Teorema, Demostración, Corolario. Se trata de una serie de problemas, resueltos mediante métodos que se describen para cada ejemplo concreto, quedando en manos del lector el reutilizarlos para sus problemas particulares. Además de las ecuaciones cúbicas, Cardano incluyó un método de su alumno *L. Ferrari* (1522–1565) para la resolución de las ecuaciones de grado 4. A la sazón, las ecuaciones bicuadradas eran resolubles por métodos conocidos.
Tomemos del texto el problema propuesto por D. Colla :
Dividir el número 10 *en tres partes, tales que constituyan una progresión geométrica y el producto de sus dos primeras partes sea* 6

Tomemos por x tal punto y escribamos las condiciones

$$\frac{6}{x} : x = x : \frac{x^3}{6}, \quad \frac{6}{x} + x + \frac{x^3}{6} = 10$$

Esto se traduce en la ecuación :

$$x^4 + 6x^2 + 36 = 60x$$

Reescribamos el método en nuestro lenguaje actual , tratando de resolver la ecuación de grado 4 :

$$x^4 + ax^2 + b = cx$$

El primer paso consiste en completar ambas partes hasta lograr que la parte izquierda sea un cuadrado perfecto :

$$(x^2 + 6)^2 = 60x + 6x^2.$$

En nuestro lenguaje, sumemos a ambos lados de la ecuación una cantidad px^2 tal que exista d verificando :

$$(x^2 + d)^2 = x^4 + ax^2 + b + px^2.$$

Es fácil observar que esta identidad se reduce al sistema de ecuaciones de grado 2 :

$$\begin{aligned} 2d &= a + p \\ d^2 &= b \end{aligned}$$

Nos quedamos con la ecuación :

$$(x^2 + d)^2 = cx + px^2$$

En la etapa siguiente introduzcamos una nueva variable t y sumemos a ambos lados $2(x^2+6)t+t^2$. Obtenemos :

$$(x^2 + 6 + t)^2 = 60x + 6x^2 + 2(x^2 + 6)t + t^2$$

Hallemos t para la parte izquierda de la ecuación sea un cuadrado perfecto.
En nuestro lenguaje, sumemos $2(x^2 + d)t + t^2$, obteniendo :

$$(x^2 + d + t)^2 = cx + px^2 + 2(x^2 + d)t + t^2$$

Para que el término de la derecha sea un cuadrado perfecto es necesario y suficiente que el polinomio de grado 2 en x tenga discriminante nulo, esto es, que el determinante de la siguiente matriz sea 0 :

$$\begin{pmatrix} p+2 & 2(p+2) & 0 \\ c & c & 2(p+2) \\ 2dt & 0 & c \end{pmatrix}.$$

Este determinante es un polinomio de grado a los más 3 en la nueva variable t introducida. En concreto, Cardano escribe en su ejemplo : *Hay que resolver la ecuación :*

$$30^2 = (2t + 6)(t^2 + 12t)$$

Resolvamos esa ecuación cúbica y sustituyamos en nuestra expresión inicial. Obtendremos :

$$(x^2 + d + t) = (x + q)^2$$

Resolver ahora se descompone en una suma por diferencia y, por tanto, en resolver dos ecuaciones de grado 2.
El método para la ecuación general $x^4 + ax^3 + bx^2 + cx + d$ se reduce al caso en el que el término de grado 3 desaparezca haciendo un cambio $x = x_1 + h$. Para las ecuaciones que no contienen términos de primer grado a la izquierda, basta con hacer un cambio del tipo :$x = \frac{k}{y}$ y resolver en y.
Esta amalgama de métodos se completaron con la obra del ingeniero italiano *R. Bombelli* de Bolonia, quien escribió su texto *"Algebra"*(1572) introduciendo los números complejos apoyándose

en las reglas $i.i = -1, (-i)(-i) = -1$, $(-i).i = 1, i.(-i) = 1$. Así pudo dar por concluido el método de Tartaglia–Cardano. Bombelli discurre un método que modifica la ecuación de grado 3 original, pero que no produce efectos esenciales (cf. [Ri, 87], p. 131–32).
La siguiente figura significativa es el matemático francés *François Viète* (1540–1603). Además de jurista fue profesor de matemáticas, dedicando grandes esfuerzos al perfeccionamiento de la trigonometría, orientada a la explicación del sistema copernicano. Fue un consejero científico y un miembro de las cortes de los reyes Enrique III y IV de Francia. Su obra fundamental fue la *"Introducción al arte del análisis"* que es una obra enorme y desmesuradamente detallada. Se benefició muy ampliamente de la aparición del cálculo literal, es decir, el prototipo del cálculo simbólico, la manipulación de expresiones literales que es el fundamento de gran parte de la matemática moderna.
Uno de los inconvenientes de los métodos similares a los de Cardano, era que se mostraban como métodos específicos, útiles solamente para ciertos tipos de ecuaciones de grados 3 y 4. Crecía el número de ecuaciones y crecía el tipo de método, volviéndolo todo confuso e intratable.
La ventaja de Viéte con el uso sistemático del nuevo formalismo simbólico, le permite no sólo resumir las matemáticas del Renacimiento, sino modelizar la resolución de ecuaciones de grado 3 y 4 con estrategias bien descritas como las siguientes :

- Sustituir $x = y + k$ para eliminar el término que contiene la incógnita elevada al grado $n - 1$, cuando n es el grado de la ecuación.
- Sustituir $x = \frac{y}{k}$ para eliminar el término que contiene la incógnita elevada a grado 1.
- Sustituir $x = ky$ para eliminar denominadores, etc.

Así consigue reducir toda ecuación de grado 3 a una ecuación de la forma

$$x^3 + 3ax = b$$

Toma la sustitución $a = t^2 + tx$ para llegar a la ecuación :

$$x^3 + 3tx^2 + 3t^2x = b$$

O lo que es lo mismo, obtiene el sistema :

$$\begin{aligned} (x+t)^3 - t^3 &= b, \\ t^3(t+x)^3 &= a^3 \end{aligned}$$

Obtiene así una ecuación cuadrática con respecto a t^3 :

$$(t^3)^2 + bt^3 = a^3$$

Resolviendo esta ecuación de grado 2 y despejando

$$x = \frac{a}{t} - t$$

ha resuelto su ecuación.
Más adelante, *R. Descartes* (1586–1650) publica su *"Geometría"* (1637). El tratado está dividio en tres libros

- *"Sobre los problemas que pueden construirse utilizando sólo círculos y líneas rectas"*,
- *"Sobre la naturaleza de las líneas curvas"* (muy bien comentado por Baruch de Spinoza, cuya fama comenzó justamente con sus comentarios a este texto)
- *"Sobre la construcción de sólidos o más que sólidos"* (que, curiosamente, se corresponde a la construcción de una teoría general de la resolución de ecuaciones y la utilización, junto a los recursos algebraicos, de los lugares geométricos. En otras palabras, se funda la Geometría Algebraica)

La simbología algebraica de Descartes ya no se diferencia esencialmente de la actual. Descartes ya tiene en cuenta las raíces reales (positivas), falsas (negativas) y aquellas que pueden ser imaginadas (imaginarias complejas). La consideración de la divisiblidad del polinomio general de grado n, $P_n(x)$, por $x-a$ para hallar las soluciones ya aparece mencionada de manera explícita la relación entre la factorización y la resolución. Una cuestión relevante en su discurso es la consideración del problema de la irreducibilidad. La reducibilidad de una ecuación de grado 4 queda transformada en una pregunta sobre reducibilidad de la ecuación de grado 3. Su método pasaba por la siguiente concepción :
Dada la ecuación $x^4+px^2+qx+r=0$ *entonces, puede ser descrita en la forma :*

$$(x^2-yx+1/2y^2+1/2p+\frac{q}{2y})\times(x^2+yx-1/2y^2+1/2p-\frac{q}{2y})=0$$

donde la variable auxiliar y *se determina a partir de la ecuación cúbica en* y^2 *:*

$$y^6+2py^4+(p^2-4r)y^2-q^2=0$$

Intentemos acabar estos comentarios de la obra de R. Descartes con las siguientes frases de Hegel :
... El gran giro de las matemáticas fue la "variable" de Descartes. Gracias a esto se introdujo en las matemáticas el "movimiento" y con él la "dialéctica", merced a lo cual surgió la "inmediata necesidad del cálculo diferencial e integral que.. Newton y Leibnitz... perfeccionaron; pero no inventaron"....

1.4. El XVII y el XVIII.

En las *"Cartas Filosóficas"* de Voltaire podemos encontrar las siguientes frases :
"...Descartes ha hecho tanto camino desde el punto en que encontró la Geometría hasta el punto en que la llevó, como Newton ha hecho tras de él : es el primero que ha encontrado la manera de dar las ecuaciones algebraicas de las curvas. "
En *I. Newton* encontramos al gran matemático del XVII. Además de la "Teoría de las Fluxiones", que tuvo que reivindicar frente a Leibnitz durante largos años de disputas, se debe destacar una obra de juventud dedicada a la resolución de ecuaciones polinomiales. Alrededor del año 1676, I. Newton se encontraba más dedicado al Algebra, desarrollando un método para la descripción local de curvas algebraicas. El algoritmo se conoce como el algoritmo de Newton–Puisseux, reaparecerá en este curso bajo la forma de Lema de Hensel, es el algoritmo al que hace referencia Voltaire en el anterior texto. En este trabajo, Newton trata de resolver el problema de la descripción local de curvas planas alrededor de puntos lisos (es decir, ofrece una versión efectiva del Teorema de la Función Implícita). Hoy, tras la prueba de Cauchy, se ha perdido mucho del método constructivo de Newton al punto de que pocos son los alumnos que pueden relacionar ambos objetos.
Entre los años 1673–1683, Newton dictó conferencias sobre álgebra en la Universidad de Cambridge. Su sucesor en la cátedra editó en el año 1707 estas conferencias bajo el título *"Arithmetica Universalis"*. Son notorias como un resumen especial del conocimiento y el desarrollo del álgebra del siglo XVII. Newton llega a decir : *" ...Todas las operaciones de la aritmética son tan necesarias en el álgebra que ellas sólo conjuntamente formas una ciencia completa de cálculos y por esto expondré ambas conjuntamente".*
De esta época datan los estudios de Newton sobre las funciones simétricas hoy conocidas como sumas de Newton. Dado un polinomio univariado $f(x):=x^n+a_{n-1}x^{n-1}+\cdots+a_0$ y supongamos que tiene por soluciones $\alpha_1,\ldots,\alpha_n$ definamos la $k-$ésima suma de Newton mediante :

$$S_k:=\sum_{i=1}^{n}\alpha_i^k$$

Siendo S_1 la traza del polinomio. Newton consigue relacionar estas sumas con los coeficientes del polinomio f mediante un sistema de ecuaciones lineales. Resolviéndolo encuentra una primera aproximación a las raíces, pero no las describe propiamente.

También encontraremos en Newton el origen de los métodos numéricos para la resolución de ecuaciones polinomiales. En el verano de 1669, Newton ha terminado su obra *"De Analysis per AEquationes Numero Terminorum Infinitas"*. Este trabajo circuló entre matemáticos británicos e incluso entre matemáticos franceses, aunque su publicación definitiva está fechada en 1771. En este trabajo, I. Newton muestra un método para aproximar una raíz de la ecuación $f(X) := X^3 - 2X - 5$. Se trata del origen del operador de Newton de un polinomio univariado :

$$N_f(z) := z - \frac{f(z)}{f'(z)}.$$

Por su parte, podemos datar el origen del cálculo numérico a partir de un segundo algoritmo de Newton de gran importancia. Se trata de resolver el problema de la descripción local de curvas planas. Sea $f(x, y) = 0$ una ecuación polinomial que define una curva pasando por el origen (i.e. $f(0, 0) = 0$). Definamos las series de exponente fraccionario (la escuela francesa las denomina series de Puisseux) como series formales en la variable x de la forma :

$$\sigma = \sum_{k \geq 0} a_k x^{k/q}$$

donde $q \in \mathbb{N}$ es una constante llamada el índice de la serie σ. Supongamos que el polinomio $f(x, y)$ es mónico en la variable Y. Entonces, existen series de potencias fraccionarias $\sigma_1(x), \ldots, \sigma_r(x)$ y enteros positivos $m_1, \ldots, m_r \in \mathbb{N}$ tales que

$$f(X, Y) = g(X, Y) \prod_{i=1}^{r} (Y - \sigma_i)$$

donde $g(0, 0) \neq 0$.
El proceso introducido por Newton (y retomado por Puisseux años después) construye para un d dado, los coeficientes de las series σ_i hasta orden d, siempre que se sepan resolver ecuaciones polinomiales univariadas. El procedimiento pasa por la construcción de un polígono, definido como la envolvente convexa del conjunto formado por los exponentes de los monomios que aparecen en la descripción de f. Nótese en Newton el precursor de cálculos efectivos mediante el estudio de exponentes monomiales (lo que se reencontrarán en las bases estándar de Hornaka y Buchberger o en los estudios del polígono de Newton de Bernstein, Kuchnirenko y Sturmfels, pero dejemos estas disquisiciones para otros momentos). Para más datos precisos del algoritmo de Nweton Puisseux, véanse [Wal, 50], [Che, 74], [Du, 87], [**?**]).
Una variante del método (posiblemente su precursor) es el llamado método de Newton para la aproximación de soluciones de ecuaciones bien condicionadas. Supongamos que la ecuación anterior $f(x, y) = 0$ verifica, además, que

$$\frac{\partial f}{\partial y}(0, 0) \neq 0$$

y definamos el operador :

$$N_f := y - \left(\frac{\partial f}{\partial y}(x, y) \right)^{-1} f(x, y)$$

Entonces, Hay solamente una de tales series σ, es el g¡ermen de una función analítica y su desarrollo de Taylor se puede calcular, obervando que $N_f^k(x, 0)$ es una función racional que coincide con σ hasta orden 2^k (ver también el capítulo dedicado a los métodos de Hensel–Newton).
Si este procedimiento permitía dsecribir local (y globalmente) las curvas, tuvo una importante incidencia en su utilización univariada. Así supongamos que $f(y)$ es un polinomio univariado y redefinamos el operador de Newton mediante :

$$N_f(y) := y - \frac{f(y)}{f'(y)}$$

Para puntos bien elegidos $x_0 \in \mathbb{Q}$ la $k-$ésima iteración del operador N_f describe un número racional x_k que está muy próximo a una raíz de la ecuación $f(y) = 0$, es decir, existe α tal que $f(\alpha) = 0$ y

$$| \alpha - x_k | \leq \frac{c}{2^{2^k}}$$

donde c es una constante que depende de f y x_0. Newton exhibe el método y es Halley quien lo formaliza. Este resultado, también tendente a la resolución de ecuaciones univariadas, supone el surgimiento del cálculo numérico y, en cierta medida, una pequeña renuncia a la resolución por radicales de las ecuaciones univariadas.
Desde el punto de vista de Newton, los "métodos geométricos" son elementos auxiliares para la estimación aproximada de la magnitud de las raíces y no se renuncia, parece que tampoco el propio Newton, a obtener estrategias que acaben resolviendo ecuaciones de grado 5 y superior.
Sobre la resolución de las quínticas por radicales la historia continúa. Tschirnaus (publicando en 1683) afirmó haber hallado una solución, pero Leibnitz demostró que era una falacia. Euler fracasó también con la quíntica, aunque encontró nuevos métodos para las cuárticas. Muchos otros (L'Hôpital, Stirling, Brenouilli, Newton, Cramer,...) trataron de encontrar soluciones como puntos de intersección de la curva dada por la ecuación

$$y = a_n x^n + \cdots + a_1 X = 0$$

con la recta $y = -a_0$, sin lograr avances signficativos.
Lagrange dió en 1770 un gran paso cuando unificó los diferentes métodos existente para la ecuación de grado 4. Su obra *"Reflexiones sobre la resolución algebraica de las ecuaciones"* (1771–72) reconsidera críticamente todos los métodos usados hasta le época y añade el suyo : la *resolvente de Lagrange*. Se basó en el método erróneo de Tschirnaus, modificándolo de modo adecuado (ver [Ri, 87], p. 322–23 para una discusión pormenorizada o en [Va, 94]).
En esencia, el método de Lagrange consideraba para una ecuación con soluciones $x_1, \ldots, x_n$ una nueva variable :

$$t = x_1 + \alpha x_2 + \alpha^2 x_3 + \cdots + \alpha^{n-1} x_n$$

donde α es una raíz primitiva $n-$ésima de la unidad (t es llamada *resolvente de Lagrange*, cf. también [vdW, 49]). Consideraba la función $\theta = t^n$ para los distintos valores de t asociados a las distintas permutaciones de las raíces. Los valores de θ eran $k \leq n!$. Ahora desarrollamos un método que permita calcular la ecuación verificada por esos valores de θ. Halladas las soluciones, hallamos los valores de t tomando raíces $n-$ésimas y con esos valores $t_1, \ldots, t_k$ obtenidos, reescribimos

$$\begin{array}{rcl} t_1 & = & x_{\sigma_1(1)} + \alpha x_{\sigma_1(2)} + \cdots + \alpha^{n-1} x_{\sigma_1(n)} \\ \ldots & = & \ldots \\ t_k & = & x_{\sigma_k(1)} + \alpha x_{\sigma_k(2)} + \cdots + \alpha^{n-1} x_{\sigma_k(n)} \end{array}$$

y tratamos de despejar las raíces. Para hallar los coeficientes del polinomio de θ observemos simplemente que son funciones simétricas en las raíces, por lo tanto, expresables como polinomios en las funciones simétricas.

1.5. El siglo XIX

El cálculo de la resolvente de Lagrange conducía a la resolución de un polinomio de grado $k \leq n!$ a partir del cual se podrían obtener las raíces de la ecuación original. Aunque el asunto funcionaba para grado ≤ 4, sucedía que para grado mayor que 5 la resolvente tenía grado $k > n$. Esto obligó a Lagrange a dudar de que los métodos por él considerados fueran a resolver las ecuaciones de grado ≥ 5. No obstante, consideró que los grupos de sustituciones por él considerados eran el *"camino para la solución"* preludiando la siguiente etapa (y el final) de la resolución de ecuaciones por radicales.

En 1813 Ruffini creyó haber demostrado la imposibilidad de la resolución de la quíntica por radicales. Su trabajo apareció en una oscura revista con varios agujeros en su demostración. Fue el matemático noruego N.G. Abel (1802–1829) quien consiguió una demostración de la imposibilidad en 1824. El siguiente problema era caracterizar fielmente la condición de la resolubilidad por radicales de cualquier ecuación que nos dieran. Parece que estaba trabajando en esto a su muerte en 1829. Pero la parte final de la historia la escribió otro joven matemático, esta vez francés.

La vida de E. Galois (1811–1832) es uno de los dramas más significativos de la histria de las matemáticas. Un lugar accesible donde consultar algunos de los hechos esenciales de su vida es la introducción del libro de I. Stewart [St, 89]. No comentaré aquí muchos de esos sucesos, pero sí la aportación fundamental de su obra. Si bien Lagrange ya había preludiado la utilización del grupo de sustituciones, es Galois quien le da forma. En su testamento, dentro del barullo de sus notas apresuradas, se observa una demostración de la imposibilidad de la resolución de las ecuaciones de grado 5 por radicales. Sin embargo, su obra se perdió hasta que J. Liouville en 1843 contó a la academia de Ciencias de Paris los resultados de E. Galois. Había nacido un formalismo (la teoría de grupos) y una caracterización completa.

Tomamos del [St, 89] el siguiente ejemplo :

En función de los resultados de Galois, una ecuación $f(x) = 0$ es resoluble por radicales si y solamente si el grupo de Galois de la extensión sobre $\mathbb{Q}$ del cuerpo de escisión de f es un grupo resoluble.

Estos sólo significa que si G es el tal grupo de Galois, existe una cadena de subgrupos : $1 = G_0 \subseteq G_1 \subseteq \cdots \subseteq G_n = G$ donde cada uno es un subgrupo normal de su siguiente y el cociente G_{i+1}/G_i es un grupo abeliano.

Entre los grupos no resolubles destaquemos el grupo simétrico S_5 de orden 5!.

Dado el polinomio $f(x) = x^5 - 6x + 3$, el grupo de Galois de f sobre $\mathbb{Q}$ es justamente el grupo simétrico S_5 que no es resoluble y, por ende, tampoco es resoluble la ecuación de turno por radicales. (Para una didáctica exposición del asunto véase [St, 89], p. 134 y anteriores).

La imposibilidad de resolver todos los polinomios por radicales tansforma el álgebra en una ciencia del lenguaje formal sobre estructuras del tipo grupo, anillo, cuerpo, etc. Sin embargo, este curso trata el, problema de la resolución de ecuaciones polinomiales por métodos simbólicos. El significado de esto puede traducirse así : dada la ecuación $f(x) \in \mathbb{Q}[x]$ se trata de :

- factorizar f y resolver cada factor irreducible separadamente,
- para cada factor ireducible, determinar el cuerpo de escisión y el grupo de Galois,
- si el grupo es irresoluble, dejarlo como está (sic), en caso contrario tratar de resolverlo por radicales utilizando una serie de composición (una cadena del tipo $1 = G_0 \subseteq G_1 \subseteq \cdots \subseteq G_n = G$ anterior).

Véase el excelente trabajo de S. Landau y G. Miller [La, 1985] y [LaMi, 85].

Lamentablemente alcanzar una resolución de este tipo no es del todo satisfactoria por lo que, en la práctica, se recurre al uso de aproximaciones. Por ello el curso estrá orientado (en el caso univariado) solamente a la fase de factorización, saliéndose del marco previsto el cálculo del grupo de Galois. (véase [PoZas, 89] para más detalles sobre el asunto). Es mi opinión que este tipo de disquisiciones algorítmicas en torno a la teoría de Galois, debe pertenecer al curso de Teoría de Cuerpos, incluyendo algún algoritmo de factorización de polinomios (mejor L^3 si es posible, que lo es).

Como final, una pequeña observación que muestra un camino entre estos problemas de la resolución de ecuaciones univariadas y problemas con rango más analítico. Tomemos una de las dos situaciones siguientes :

- *Resolución de sistemas de ecuaciones diferenciales lineales homogéneos.* Sean $x_1, \ldots, x_n : \mathbb{C} \longrightarrow \mathbb{C}$ funciones que suponemos verifican la condición inicial $x_i(0) = a_i$. Sea $A \in \mathcal{M}_n(\mathbb{C})$ una matriz cuadrada. Escribamos X por el vector columna cuyas coordenadas son las

funciones $x_1, \ldots, x_n$. sea $\dot{X}$ el vector columna cuyas coordenadas son las derivadas de las funciones $x_1, \ldots, x_n$ con respecto al parámetro t Se trata de resolver la ecuación diferencial :

$$\dot{X} = AX$$

- *Resolución de una ecuación diferencial homogénea.* En este caso, tomemos una sola función $x : \mathbb{C} \longrightarrow \mathbb{C}$ y la ecuación diferencial :

$$\frac{d^n f}{dt^n} + a_{n-1}\frac{d^{n-1} f}{dt^{n-1}} + \cdots + a_1 \frac{df}{dt} + a_0 f = 0$$

La condición inicial impuesta que sea del tipo :

$$\frac{d^{n-1} f}{dt^{n-1}}(0) = a_n, \ldots, f(0) = a_1$$

También se trata de resolver.

Es conocido cómo el segundo caso se reduce al primero siendo la matriz A la matriz compañera del polinomio $p(x) := x^n + a_{n-1}x^{n-1} + \cdots + a_0$. En ambos casos, el camino de resolución pasa por el cálculo de la forma canónica de Jordan (ver, por ejemplo, el librito de Arnold sobre las ecuaciones diferenciales ordinarias, vol. 1). Veremos cómo el cálculo de la primera forma racional (o forma de Frobenius) o la segunda forma racional (con los divisores elementales) son resultados de un procedimiento algorítmico. Sin embargo, no existe procedimiento alguno que calcule las soluciones del polinomio mínimo de la matriz A de turno, luego no hay manera de resolver la ecuación diferencial. Obsérvese que esta es un afirmación demasiado corriente en Análisis como para tomarla a la ligera.

Sin embargo, hay que dar a E. Galois otro papel importante en la historia común de la Matemática y la Informática. *Es el primero que descubre la relevancia de la complejidad de los algoritmos y su papel en el análisis de la matemática.* Si bien se reconoce el resultado de G. Lamé (1844) como el primer resultado de complejidad teórica al estimar el número de divisones necesarias para ejecutar el algoritmo de Euclides sobre $\mathbb{Z}$, debe darse a E. Galois el mérito de haber descubierto la relevancia de la complejidad. En su célebre testamento, E. Galois dice que dispone de un método para decidir si una ecuación de grado 5 o mayor es resoluble por radicales. El insiste diciendo que si alguien le da la ecuación, él sabe bien cuáles son los cálculos que hay que hacer para decidir la resolubilidad por radicales; pero, resume desesperadamente, no puede realizarlos con lápiz y papel. La frase dramática, vistas las circunstancias, y premonitoria, vista la complejidad del cálculo con grupos de Galois, dice textualmente : *"En un mot, les calculs sont impracticables"*. Lo que Galois ha descubierto es un algoritmo de complejidad exponencial en tiempo de ejecución y, por tanto, cuando el grado es superior a 100 impracticable ni siquiera por los ordenadores actuales. Podemos imaginar el dramatismo de sus esfuerzos por ejecutar un algoritmo de tales características con lápiz y papel.

Hacia finales del siglo XIX y principios del XX, los matemáticos, y los algebristas en particular, continúan desarrollando algoritmos. Esencialmente algoritmos para la resolución de sistemas de ecuaciones polinomiales multivariadas. En este sentido cabe destacar la obra de *L. Kronecker* [3], la obra de Hilbert sobre el Nullstellensatz [4] y las continuaciones que se propagarán hasta principios del siglo XX en obras compilatorias como las de J. König [5] o la de F.S. Macaulay [6]. La historia reciente ya continúa con las diversas etapas que conducen hasta la Eliminación actual.

[3] L. Kronecker. "Grundzüge einer arithmetischen theorie de algebraischen grössen". *J. reine angew. Math.*, **92** (1882) 1–122.

[4] D. Hilbert. "Über der Theorie der Algebraischen Formen". *Math. Annalen* **36** (1890) 473–534.

[5] J. König. *"Einleitung in die allgemeine Theorie der algebraischen Grözen"*.Druck und Verlag von B.G. Teubner,Leipzig. (1903).

[6] F.S. Macaulay. *The Algebraic Theory of Modular Systems"*. Cambridge tracts in math. and Math. Physics, Cambridge University Press (1916).

1.6. La Aparición de la Noción de Algoritmo.

De lo expuesto en el apartado anterior, puede deducirse que los algebristas no estaban sólo interesados en disquisisiones teóricas sobre propiedades de estructuras comúnmente llamadas algebraicas, sino en la resolución de problemas bien concretos : ecuaciones polinomiales univariadas. Tampoco hay que olvidar cómo, desde el Renacimiento, los matemáticos tratan de construir máquinas que les resuelvan las tareas (véase la máquina aritmética de Pascal, por ejemplo).
Sin embargo, debemos señalar que *nadie sabía qué era un algoritmo*. Definiciones del tipo *algoritmo es una fórmula* o *una serie de cálculos finitarios*, o extravagancias del mismo pelaje, eran moneda de cambio entre matemáticos reputados, muy delicados en el manejo de definiciones altamente sofisticadas.
Por eso, cuando *D. Hilbert* propone el décimo de sus famosos 23 problemas en la conferencia inaugural del Congreso Internacional de matemáticos de París del año 1900 (véase [7]), D. Hilbert no esconde ninguna intención próxima a lo que sucedió. El famoso Décimo Problema de Hilbert se enuncia del modo siguiente :

Problema (**Problema X de Hilbert).** Dar un algoritmo que permita decidir para un polinomio dado $q(X_1, \ldots, X_n) \in \mathbb{Q}[X_1, \ldots, X_n]$ si existe un punto $(x_1, \ldots, x_n) \in \mathbb{Z}^n$ tal que :

$$q(x_1, \ldots, x_n) = 0.$$

Es decir, existencia de soluciones diofánticas (i.e. con coordenadas en $\mathbb{Z}$) en hipersuperficies diofánticas (i.e. dadas por un polinomio con coeficientes enteros)
El problema era fundamental pues enlazaba con problemas más amplios que interesan mucho a éste curso. Típicamente, su Nullstellensatz (que parece ser de Kronecker, luego anterior) es un buen instrumento para resolver el siguiente problema :

Problema (**Nullstellensatz de Hilbert).** Dados $f_1, \ldots, f_s, \in \mathbb{C}[X_1, \ldots, X_n]$, decidir si existe un punto $(x_1, \ldots, x_n) \in \mathbb{C}^n$ tal que :

$$f_1(x_1, \ldots, x_n) = \ldots = f_s(x_1, \ldots, x_n) = 0.$$

Una demostración a este resultado (i.e. un Nullstellensatz con cotas de grado) se puede encontrar en el trabajo de la alumna de D. Hilbert G. Hermann en [He, 26]. A la sazón, L. Kronecker había introducido muchos años antes un algoritmo para resolver tal problema en su trabajo de 1882.
De hecho, el problema atañe a una serie de preguntas más amplias. Supongamos dada una sucesión de polinomios $f_1, \ldots, f_s, g_1, \ldots, g_r$ y supongamos $K = \mathbb{Z}/p\mathbb{Z}, \mathbb{Q}, \mathbb{R}$ o $\mathbb{C}$, alguno de los cuerpos subyacentes, siendo $f_i, g_j \in K[X_1, \ldots, X_n]$. las preguntas que podemos hacernos son del tipo siguiente :

Problema 1.6.1. Dados $f_1, \ldots, f_s, g_1, \ldots, g_r \in K[X_1, \ldots, X_n]$, decidir si existe un punto $(x_1, \ldots, x_n) \in K^n$ tal que :

$$f_1(x_1, \ldots, x_n) = 0 \cdots f_s(x_1, \ldots, x_n) = 0; g_1(x_1, \ldots, x_n) \neq 0, \ldots, g_r(x_1, \ldots, x_n) \neq 0$$

La primera transformación obvia resume el problema en la forma siguiente :

$$f_1(x_1, \ldots, x_n) = 0 \cdots f_s(x_1, \ldots, x_n) = 0; g(x_1, \ldots, x_n) \neq 0$$

donde $g = \prod_{i=1}^{r} g_i$. Añadamos una variable más x_{n+1} y la pregunta anterior es equivalente a la siguiente :

Problema (Satisfabilidad)**.** Dados $f_1, \ldots, f_s, g_1, \ldots, g_r \in K[X_1, \ldots, X_n]$, decidir si existe un punto $(x_1, \ldots, x_n) \in K^n$ tal que :

$$f_1(x_1, \ldots, x_n) = 0 \cdots f_s(x_1, \ldots, x_n) = 0; x_{n+1} g(x_1, \ldots, x_n) - 1 = 0.$$

[7] D. Hilbert. "Mathematische Probleme". *Archiv für Mathematik und Physik* **1**(1901) 44–63 y 213–237. Véase también la versión inglesa en D. Hilbert "Mathematical Problems". *Bull. of the A.M.S.* **8** (1902) 437–479.

En el caso complejo ($K = \mathbb{C}$) y en el caso $K = \mathbb{Z}/p\mathbb{Z}$ el modelo de problema se quedaría en este nivel de enunciado. En el caso real y el racional, podemos simplificar aún un poco más. Tomemos el polinomio :

$$p(x_1, \ldots, x_n) := f_1^2 + \cdots + f_s^2 + (x_{n+1}g(x_1, \ldots, x_n) - 1)^2 = 0$$

y tenemos la decisión de fórmulas cuantificadas en la Teoría Elemental de Números.
Si, por el contrario, se preguntara sobre la existencia de raíces reales o racionales (i.e. soluciones en $\mathbb{R}^n$ o en $\mathbb{Q}^n$) , no se conocía ningún algoritmo en la época en que Hilbert establece su famoso problema. De hecho, en mi opinión, D. Hilbert creía en la existencia de tal algoritmo y trataba, simplemente, de provocar el estudio para encontrarlo.
Así el caso real se resolvió pronto. En 1931, el matemático polaco *A. Tarski* anuncia que tiene un algoritmo para dedicir si una o varias ecuacones polinomiales poseen solución real (en $\mathbb{R}^n$). Esto aparece publicado en su trabajo [8]. Las circunstancias del ascenso del nazismo en Alemania y la emigración de Tarski a los Estados Unidos, postpuso la publicación de este resultado hasta la aparición de una edición preparada por J.C.C. MacKinsey [9] . En la misma época, *A. Seidenberg* publica su propio algoritmo para resolver el caso real en su trabajo de 1956 [10] .
A pesar de todos estos avances, que siguen sin resolver el Problema X de Hilbert, pocos matemáticos han pensado el problema de modo alternativo. Si no se conoce la noción de algoritmo poco o nada se puede reflexionar sobre el problema propuesto por D. Hilbert. Por tanto, es sobre la noción de algoritmo sobre la que vuelcan sus esfuerzos algunos matemáticos. En 1916, el matemático noruego *A. Thue* introduce sus sistemas de reescritura que serán pronto descartados.

1.6.1. Teorías Formales

El caso de cuerpos finitos no ofrece mayor dificultad, el algoritmo es el obvio : probar con todos los elementos hasta qu eno quede nada con lo que probar.
Este problema era esencial en el desarrollo de toda la matemática posterior; pero reposaba en la indefinición de la noción de algoritmo. El asunto era aún peor. Lo que subyace a la pregunta del Problema X de Hilbert es una questión más profunda y esencial no sólo en la construcción de las matemáticas, sino en todo sistema de pensamiento hasta ahora propuesto por el ser humano.
Resumamos muy sucintamente el asunto : Lo poco que el ser humano ha aportado a la Naturaleza ha sido el pensamiento deductivo. Tanto en el caso del empirismo de Locke y Newton como en el racionalismo de Spinoza y Descartes, no queda mas remedio que reflejar en algun sistema gramatical afirmaciones (o interrogaciones) y, las afirmaciones son las verdades que uno puede exponer, demostrar o aceptar a priori. Lo que es inetivable es discernir entre ciertos, pocos elementos. Vamos a introducirlos :

Una Teoría Formal es :

- *Alfabeto y símbolos :* Un conjunto de símbolos expresables, sobre un alfabeto finito.
- *Expresiones :* Cualquier lista finita de símbolos.
- *Fórmula bien formadas :* De cualquiera de las expresiones, aquellas que una cierta Gramática considera como válidas (pensar en el castellano)
- *Axiomas :* Un conjunto de fbf's seleccionado como sistema de hipótesis[11].
- *Reglas de inferencia :* Son reglas que permiten "deducir" o "inferir" alguna fbf de unas fbf's dadas. La más usual es el "modus ponens" Si Sócrates es griego y los griegos son hombres, puedo inferir que Sócrates es hombre. ($A \to B, A$ permite deducir B).

[8] A. Tarski . " Sur les ensembles définissables de nombres réeles". *Fund. Math.* **17** (1931) 210–239.
[9] A. Tarski. *"A decision method for elementary algebra and geometry".* (Prepared for publication by J.C.C. Mac Kinsey, Berkely (1951).
[10] A. Seidenberg. "A new decision method for elementary algebra". *Ann. of Math.* **60** (1954) 365–374.
[11] En ciencia común y silvestre estos axiomas pueden ser tomados de la experiencia (Locke, Newton) o de la interiorizacion reflexiva (Descartes, Spinoza).

Una demostración en una teoría formal $\mathcal{T}$, es una lista de fbf's

$$A_1, \ldots, A_r$$

donde cada A_i es o bien un axioma o es deducible de $A_1, \ldots, A_{i-1}$ por el uso de las reglas de inferencia.
Un Teorema de una teoría formal $\mathcal{T}$ es toda fbf que posee demostración, es decir una fbf T que existe

$$A_1, \ldots, A_r$$

demostración, siendo $T = A_i$ para algún i. Los Teoremas son solamente LO DEDUCIBLE.
El problema surge de la relación entre lo deducible y lo que es verdad.
Para ello, debemos pasar al plano de la semántica, de los significados, de las interpretraciones de los objetos descritos por las fbf's.

- *Una interpretación de una teoría formal* es una asignación de verdad o falsedad a cada fbf de la teoría.
- *Un modelo de un conjunto S de fbf's* es una interpretación en la que todas las fbf's de S son ciertas. Un modelo para una teoría es una interpretación en la que todos los Teoremas son ciertos.

Ahora empiezan los problemas :

- *Una teoría es completa* si cada fbf que es cierta para cualquier interpretación es un Teorema (i.e. es deducible).
- *Una teoría es sólida* si cada Teorema (i.e. lo deducible) es cierto en cualquier interpretación.

La idea es no salirse de los modelos de la teoría (i.e. interpretaciones sólidas) y surgen preguntas obvias : Es toda teoría sólida, además completa?. Aún hay que sufrir más :

- *Una teoría es consistente* cuando la negación aparece entre sus símbolos y no es posible deducir una fbf y su negación.

Una teoría es inconsistente cuando hay cosas tales que ni ellas ni suys negaciones son deducibles.
Pero, existe algo así?.
El Problema X, añade un elemento más : la decidibilidad.

- *Una teoría es decidible* si existe un algoritmo tal que ante toda fbf puede decidir si la propiedad enunciada es demostrable o no.

Evidentemente, el Problema X interviene en la cuestión de la decidibilidad para una teoría bien sencilla : la Teoría Elemental de Números.

1.6.2. Turing, Gödel y Church

Volvemos pues a la pregunta sobre una definición de algoritmo. Aquí vamos a resumir muy deprisa su génesis y, más tarde, trataremos de ordenar un poco todo el asunto.
Hacia mediados de los años 30, dos figuras relevantes aparecen para fijar la noción de algoritmo : al austríaco *K. Gödel* y el británico *A. Turing*. Rodeados de las figuras de *A. Church* y su alumno *S.C. Kleene*.
Nos vamos directamente al Círculo de Viena, donde la participación de Hahn fue central (como fundador) y al que Gödel se incorpora . Este influyente conjunto de matemáticos y lógicos se ve roto por el nazismo en pedazos irrecomponibles, pero influyó muy notablemente la filosofía y la lógica de primeros de siglo (hasta finales de los 30). El Círculo de Viena se preocupa de aspectos filosóficos y metamatemáticos (sobre los fundamentos de la Matemática).

Es en este ambiente donde K. Gödel elabora su famosa tesis (23 páginas) en la que demuestra la Incompletitud de la Teoría Elemental de Números (véase la publicación del trabajo [12]). Aquí Gödel usa por vez primera algo parecido a las funciones computables (él las llamó "rekursiv"). Son las llamadas "funciones primitivas recursivas", i.e. las que permiten hallar $f(n+1)$ de la información de $f(n)$.
Durante los años 30, K. Gödel visita Princeton varias veces, hasta su traslado definitivo en 1940. En 1934, durante una de sus visitas, dió una charla cuyas notas circularon. Esats notas fueron tomadas por Kleene y Rosser, quienes a la sazón completaban sus estudios de doctorado bajo la dirección de A. Church. Las notas fueron eventualmente publicadas en 1965, como [Gödel, 65]. En esta conferencia, él hace surgir la noción de computabilidad efectiva. Notó que fromas más generales de recursión deberían ser admitidas antes de que sus funciones rekursiv pudieran cubrir todo lo computable. Así definió una clase que llamó "funciones generales recursivas" (al parecer esto fue sugerido por una carta de Herbrand).
Alonzo Church estaba en Princeton desde 1929 y con su estudiante de tesis Kleene habían desarrollado el concepto de función $\lambda-$definible. La cuestión de la relación entre $\lambda-$definibilidad y computabilidad efectiva fue estudiada por el propio Church en [Chu, 35] y [Chu, 36]. Estos trabajos no sólo contenían la propuesta que hoy lleva su nombre, sino los primeros ejemplos de problemas insolubles decisionales. El alumno de Church, S.C. Kleene, es quien demuestra en [Kl, 36] la equivalencia entre el concepto de funciones general recursivas y $\lambda-$ definibles.
Está naciendo la **Tesis de Church**. La tesis de Church no es propiamente una Tesis, ni un Teorema, sino una Definición : *Se llama algoritmo a toda función recursiva, todo procedimiento del $\lambda-$calculus y toda noción equivalente a ambas.*
Por su parte, A. Turing estaba ya interesado en los trabajos sobre computación y Algebra. En su trabajo de 1948 A. Turing [13] introducirá la noción de condicionamiento de los métodos del Algebra Lineal Numérica, convirtiéndose en el fundador del Algebra Lineal Numérica moderna.
A la sazón, A. Turing publicaba su modelo en su trabajo de 1936 [14] dedicado a caracterizar los números reales computables (recursivamente enumerables) y ya hacía referencia al trabajo de Church. Probó en un apéndice la equivalencia entre su modelo y la $\lambda-$definibilidad. De hecho, dos son las aportaciones fundamentales de Turing en este artículo. De una parte, la introducción de un nuevor modelo alternativo de algoritmo (máquinas de Turing) y el resultado de autoreducibilidad basado en la máquina Universal. De otro, el análisis de los números reales recursivamente enumerables y la demostración de que $\mathbb{R}_{re}$ no es un cuerpo computable.
Emil Post, que también introdujo su modelo de cálculo en 1936, y que resultó equivalente al de Turing (véase [15]), cuyo formalismo ha influenciado fuertemente los formalismos introducidos a posteriori, llegó a describir *la tesis de Church como una ley natural, "un descubrimiento fundamental" conceniente a "the mathematizing power of Homo Sapiens"*. Así, la Tesis de Church toma la forma siguiente :

Definición 1.6.1 (TESIS de CHURCH)**.** Llamaremos computable a toda función calculable por alguno de los siguientes métodos equivalentes de caracterización :

- Calculable por una máquina de Turing,
- es un función gerenal recursiva,
- es $\lambda-$definible,
- es Post–calculable,

o calculable por cualquier otro procedimiento equivalente a alguno de éstos.

[12] K. Gödel. "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I". *Monatsh. Math. Phys.* **38** (1931) 173–198.
[13] A. Turing. "Rounding-off errors in matrix processes". *Quart. J. Mech. Appl. Math.*, **1** (1948) 287–308.
[14] A. Turing. "On computable numbers, with an application to the Enscheidungspoblem". *Proc. London Math. Soc., Ser. 2,* **42** (1936) 230–265.
[15] E. Post. "Finite Combinatory processes–formulation I". *J. Symbolic Logic* **1**, (1936), 103–105.

El modelo de Turing es, con mucho, el de mayor sencillez definitoria, pero también el más relacionado con algo que no soñaban en aquélla época : los ordenadores. De hecho, existe una historia, cuyos datos han sido revelados muy recientemente, que indica bastante de la aplicación del modelo de Turing a la Computación.

Turing es un matemático brillante a finales de los años 30 cuando comienza la Segunda Guerra Mundial. En esa época, algunos matemáticos polacos han logrado descubrir el sistema de comunicación secreto del ejército alemán : la "famosa" máquina Enigma con tres rotores. A partir de 1939, de poco le sirve a Polonia esta información y es Churchill quien, personalmente, se hace cargo del rescate de algunos de estos matemáticos y los traslada a Inglaterra. Allí, manteniendo el máximo secreto, Churchill ha creado el proyecto "Ultra" en un lugar aislado de la campiña británica (conocido como Bletchley Park). Entre los miembros del proyecto "Ultra" se encuentra A. Turing quien acabará dirigiendo el equipo de descodificadores. Tras conseguir la información de que disponían los polacos e incorporarla al equipo, Churchill impone el más absoluto de los secretos. Se trata de poder descodificar los mensaje secretos alemanes; pero los alemanes no deben saber nunca que los ingleses conocen tal secreto.

La primitiva máquina Enigma es una máquina con tres rotores de giro independiente asociados a un teclado. Las diversas combinaciones de los rotores permiten biyecciones sofisticadas entre los conjuntos de letras del teclado. Así, poseyendo una clave, normalmente asociada al día, para ajustar los rotores, se puede transmitir información confidencial por medio de la radio. Los efectos del trabajo de Turing, sobre la base del trabajo preliminar, fueron esenciales en la Batalla de Inglaterra. Desarrolla una máquina, conocida como The Bombe, basada en los análisis de los criptógrafos polacos que permite hacer muchas cálculos rápidamente. Pero es aún una calculadora, no es programable. Churchill y la fuerza aérea británica, eran capaces de predecir los movimientos de los grupos de bombarderos de la Luftwafe, consiguiendo, en muchos casos, interceptarlos.

Estos éxitos iniciales, hicieron que Churchill aumentara las dotaciones del proyecto "Ultra", creando distintos departamentos en constante ampliación. Dos elementos eran cruciales : el secretismo de sus trabajos no debía llegar a manos alemanas; pero ni siquiera los aliados deberían saber que disponían de medios para descodificar las máquinas Enigma alemanas.

Sin sospechar que todas, o muchas, de sus conversaciones estaban siendo escuchadas y transcritas, el ejército y la armada alemanes aumentaron el número de rotores por razones puramente instintivas. A. Turing, también fue capaz de descodificar la nueva máquina Enigma usando simplemente lápiz y papel.

A mediados de 1942, los alemanes introducen una sofisticación adicional a sus comunicaciones por radio. Se trata del codificador de Lorentz de 12 rotores. Ahora, el número de posibles biyecciones entre teclados ha aumentado considerablemente. La nueva máquina, basada en el mismo principio, se incorpora en los submarinos alemanes.

Los británicos descubren bien pronto que los alemanes han cambiado su sistema criptográfico y es entonces cuando, por vez primera, Turing observa la imposiblidad de seguir descodificando a mano. Apoyado por un alto prespuesto, por la voluntad explícita de Churchill que considera su proyecto "Ultra" como la clave de la guerra, un grupo de ingenieros se pone a disposición de A. Turing para construir el primer ordenador electrónico de la historia. Se trata del ordenador *Colossus* y su hermano mayor *Colossus 2* que entraron en servicio en 1943 y estuvieron trabajando hasta el final de la Segunda Guerra Mundial. Ambos ordenadores eran capaces de procesar las combinaciones de la máquina de Lorentz y descodificar los mensajes por radio alemanes.

Cuando los norteamericanos entran en la Segunda Guerra Mundial, Churchill mantiene el secreto de su conocimiento del sistema criptográfico alemán. Sólo tras la Cumbre de Yalta, Churchill contará a Roosevelt que conoce el secreto; le transmitirá información ya descodificada; pero no le mostrará la existencia de las máquinas Colossus. En cuanto a Stalin, Churchill le transmitirá información; pero nunca llegará a informarle ni de la existencia del proyecto "Ultra" ni, mucho menos, de su funcionamiento. Sorprendentemente, Stalin ha conseguido infiltrar un hombre de su confianza en los barracones de Bletchley Park. Así Stalin conocerá todo el funcionamiento y evolución del proyecto "Ultra" sabiendo, al mismo tiempo, que sus aliados le mantienen apartado del secreto.

Con el final de la Segunda Guerra Mundial, y el advenimiento de la Guerra Fría, Churchill da órdenes de desmantelar el proyecto "Ultra", destruir los ordenadores Colossus y dispersar a los miembros de los equipos con la orden de guardar el secreto más absoluto. Así desaparecieron los primeros ordenadores electrónicos y su existencia no ha sido conocida hasta pasados los cincuenta años preceptivos de los Secretos Británicos.
De vuelta a sus actividades académicas, no siempre muy satisfactorias por la falta de comprensión de sus colegas, A. Turing participará en la creación de los ordenadores británicos Mark I y Mark II, ya metidos en la década de los cincuenta. Pensar en A. Turing reconstruyendo su modelo, casi diez años después de hebrlo construido una vez, sólo por razones políticas, ayuda a entender muchas de sus crisis personales y profesionales.
En los Estados Unidos, *J. von Neumann*, que ha dedicado bastante tiempo a la Teoría de Autómatas y, por ende, conce bien la obra de Turing, es nombrado el consejero matemático en la construcción de los primeros ENIAC estadounidenses.
Desde entonces, hasta nuestros días, todos los ordenadores han mantenido las pautas de la máquina abstracta de Turing. En ocasiones, el modelo es modificado ligeramente para crear nuevas "Arquitecturas de Ordenadores"; pero manteniendo siempre el concepto inicial de Turing.
Se produce un hecho extraordinario en la Historia de la Matemática : *Por vez primera un modelo teórico antecede al modelo físico, por vez primera en la Hisoria no hay que crear un modelo matemático de la realidad : es la Realidad la que imita al modelo matemático*. Las consecuencias de tal fenómeno son, obviamente, extraordinarias para la posición de un matemático.
Sobre la resolución del Problema X de Hilbert, véase la Sección 9.8

Parte I

Los Algoritmos de los Lenguajes Formales

Capítulo 2

Jerarquía de Chomsky

Contents

2.1. Introducción 35
2.2. Lenguajes Formales y Monoides 38
2.2.1. Operaciones Básicas con palabras. 39
2.2.2. Operaciones Elementales con Lenguajes 39
2.2.3. Sistemas de Transición 40
2.3. Gramáticas Formales 41
2.3.1. Sistema de Transición Asociado a una Gramática. 41
2.3.2. Otras Notaciones para las Producciones. 42
2.3.2.1. Notación BNF. 42
2.3.2.2. Notación EBNF. 42
2.4. Jerarquía de Chomsky 43
2.5. Disgresión: Problemas de Palabra 44

2.1. Introducción

La primera disquisición importante al fijar un modelo de cálculo hace referencia a los fundamentos de la comunicación y el lenguaje. Para ser precisos todo cálculo algorítmico consiste fundamentalmente en un proceso de comunicación: algo es emitido (input), manipulado (transmisión) y respondido (output). Es una comunicación entre hombre y máquina o una comunicación entre seres humanos. Pero el principio de esta discusión debe orientarse hacia lo que es susceptible de ser comunicado (tanto el input como el output son objetos comunicables).
Nos vamos directamente al Círculo de Viena, con precursores como K. Popper, y con actores activos como R. Carnap, H. Hahn y O. Neurath. En el ámbito de la lógica matemática son relevantes la pertenencia de miembros de la talla de K. Gödel o A. Tarski. Este influyente conjunto de filósofos, matemáticos y lógicos se ve roto por el nazismo en pedazos incomponibles, pero influyó muy notablemente la filosofía y la lógica de primeros de siglo (hasta finales de los 30).
Apliquemos el empirismo lógico como ideología provisional. Tomemos la disquisición inicial: ¿qué es susceptible de ser comunicado?.
Una respuesta razonable (empírica en nuestra aproximación) es que es comunicable todo aquello expresable en un alfabeto finito. Nuestra aproximación empírica se basa en la experiencia: no conozco ningún caso de información emitida o recibida por alguien, con contenido semántico no ambiguo, que no haya sido expresada sobre un alfabeto finito.
A partir de esta idea consideraremos como Σ un conjunto finito que denominaremos *alfabeto* y por Σ^* el conjunto de todas las *palabras* expresables sobre este alfabeto finito.

Dos precisiones importantes: Lo comunicado (el significante) es una palabra sobre un alfabeto finito, pero el significado (la componente semántica de la comunicación) no es tan claramente finito. Tomemos un ejemplo de las matemáticas. Sea D^1 el conjunto de números reales dado por:

$$\{(x, y) \in \mathbb{R}^2 \ : \ x^2 + y^2 - 1 \leq 0\}$$

El tal conjunto no es finito, ni contable. Podría quizá discutirse su existencia (uno de los problemas más difíciles de la filosofía de las matemáticas es determinar el significado de existencia: existe lo que es expresable –esto es seguro–, pero, ¿existe lo que no puedo expresar? [1]). Suponiendo que $\mathbb{R}$ exista, yo puedo expresar un conjunto cuyo cardinal no es numerable mediante una expresión sobre un alfabeto finito. Por lo tanto, los significantes caminan sobre una digitalización finita, sobre un alfabeto finito, no así los significados. No olvidemos, finalmente, que la modelización continua de la semántica es una de las corrientes de la moda última; pero tampoco olvidemos que la semántica (y la Semiótica) cuentan con los elementos adicionales de la subjetividad que son bastante "difusos".
La segunda consideración es que nosotros usaremos el lenguaje de la Teoría de la Recursividad y no el de la Lingística. Para referencias al asunto véase, por ejemplo, [Marc, 67]. En este caso, la terminología se modifica del modo siguiente: el alfabeto se denomina vocabulario, las palabras son lo mismo, y el lenguaje es una cantidad, posiblemente infinita, de palabras sobre el vocabulario. Pero vayamos a nuestra definición:

Definición 2.1.1 (Alfabeto)**.** Sea Σ un conjunto finito que llamaremos *alfabeto*.

- Una palabra sobre Σ es una lista finita de símbolos de Σ. Podemos formalmente identificar las listas $x = x_1 \cdots x_n$ de símbolos ($x_i \in \Sigma$) con los elementos del producto cartesiano Σ^n. Denotaremos por $\mid x \mid = n$ la longitud de la palabra $x_1 \cdots x_n$.
- El conjunto de todas las palabras sobre el alfabeto Σ se denotará mediante Σ^* y podemos identificarlo con la unión disjunta
$$\Sigma^* = \dot{\bigcup_{n \in \mathbb{N}}} \Sigma^n$$
- Los subconjuntos L de Σ^* se denominan lenguajes.

Insistamos en la notación $x_1 \cdots x_n$ para expresar la palabra $(x_1, \ldots, x_n) \in \Sigma^n$. Los ")" y las ", " pudieran ser (y suelen ser) elementos del alfabeto.
Nótese por ejemplo la identificación obvia, cuando $\Sigma = \{a\}$ es un alfabeto de una sola palabra, entre Σ^* y $\mathbb{N}$.

Observación 2.1.1. La única observación relevante es que si Σ es un conjunto finito Σ^* es un conjunto numerable. No podemos expresar mucho más allá que una cantidad numerable de significantes (a lo sumo). La verdad, no es gran cosa: una sonata de Mozart no es sino una triste palabra de unas pocas numerables posibles.
El considerar alfabetos numerables no cambiaría gran cosa, lo que se puede expresar sobre un alfabeto numerable es expresable sobre un alfabeto finito (por razones obvias). El punto duro comienza cuando uno se pregunta si es posible la comunicación (hombre–máquina u hombre–hombre) a través de lenguajes sobre alfabetos no numerables.

Otros ejemplos, "El Quijote", entendido como el libro completo, es, para nuestro contexto, una palabra sobre el alfabeto del castellano, i.e. $\{a, b, \ldots, z\}$, $\{A, B, \ldots, Z\}$, {?,Á, !, ",", "."$, \dot{:}$, "$\sharp$", .}, donde las "," y "." son los obvios, $\dot{:}$ es el "punto-y-aparte" y $\sharp$ son los espacios entre palabras.
Uno podría muy bien argumentar porqué el "Otello" de Shakespeare no es una palabra del castellano y la respuesta es la obvia: es una palabra sobre el alfabeto castellano; pero el castellano no es solamente un alfabeto, sino un lenguaje $\mathcal{C} \subseteq \Sigma^*$ en el que se incluyen solamente las palabras formadas por sucesiones de símbolos del Diccionario de la Real Academia de la Lengua

[1] Esto será lo máximo que nos introduciremos por los oscuros caminos de la filosofía. Paul Gordan, gran matemático del siglo XIX, amonestó a David Hilbert con su famosa frase "Das ist keine Matematik, Das ist Theologie" por "demostrar" la existencia de objetos, sin "mostrar" esos objetos.

(ver autómatas finitos para más disquisiciones). El "Otello" pertenece al lenguaje inglés $\mathcal{I} \subseteq \Sigma^*$. Módulo traducciones (que no juegan por ahora) una versión original de la Ilíada[2], el Corán, El Idiota o los Vedas no pertenecerían a Σ^* por usar diferentes alfabetos (el griego, el árabe, el cirílico o el sánscrito). Por lo tanto, variarán tanto los alfabetos como los conjuntos llamados lenguajes, teniendo la comunicación occidental en común el alfabeto. Sin embargo, las traducciones muestran que no hay mucho que añadir aunque se cambie el alfabeto.
Esto muestra que alfabetos complicados o sencillos no tienen relación con la simplicidad del lenguaje. Aprovecharemos este momento para introducir un lenguaje que volverá a aparecer más adelante.

Ejemplo 2.1.1. Sea $\Sigma = \{ A, C, G, T \}$, representando las cuatro bases que conforman el ADN, a saber: Adenina, Guanina, Citosina y Timina. Las cadenas de ADN forman un lenguaje que contiene la información genética y esta empaquetada de esta forma para su posible transmisión hereditaria. Curiosamente, este lenguaje es casi universal y se aplica a todos los seres vivos menos en excepciones contadas dentro de casos contados.

Cada cadena de ADN guarda la codificación de varias proteínas. Dentro del ADN, cada secuencia de tres bases de las mencionadas corresponden a un aminoacido concreto. Esto se conoce como el código genético. Por ejemplo, la combinación ATG representa el inicio de la secuencia y el final puede ser representada por TGA, TAG, TAA. [3]

Primeras concreciones sobre lo computable:
Un algoritmo es usado para resolver un problema. La entrada del algoritmo son los "datos del problema" que, convenientemente manipulados, aportan una "solución". Por lo tanto, todo algoritmo evalúa una correspondencia

$$f : D \longrightarrow S$$

donde D son los datos y S las soluciones. Como ya queda claro de lo discutido arriba (o eso esperamos), éste que escribe sólo puede discernir "datos" y "soluciones" como significantes de algo (el "algo" ya no corresponde a la disquisición). Luego

Definición 2.1.2. Se definen:

- Un problema es una correspondencia $f : D \longrightarrow S$ entre dos conjuntos. Resolver un problema es evaluar f.
- Un problema $f : D \longrightarrow S$ es susceptible de ser resuelto algorítmicamente si y solamente si D y S son lenguajes expresables sobre un alfabeto finito. Uniendo alfabetos, uno podría suponer que son lenguajes sobre un alfabeto común Σ.
- Un problema es, por tanto, evaluar una correspondencia $f : \Sigma^* \longrightarrow \Sigma^*$. Los elementos del dominio (los datos) se suelen llamar inputs (también son susceptibles de ser llamados inputs aquellos $x \in \Sigma^*$ tales que no existe $f(x)$). Los elementos del rango de f son las soluciones y se denominan outputs.

Entre los muchos problemas distinguimos una subclase de gran importancia: los PROBLEMAS DECISIONALES. Se trata de evaluar funciones parcialmente definidas $f : \Sigma^* \longrightarrow \{0, 1\}$. Claramente si $D(f)$ es el dominio de definición de f y definimos $L := f^{-1}(\{1\})$, la función f es del tipo restricción al dominio de f de la función característica de L (ver Ejemplo A.5.1 en el Apéndice A). Los tales lenguajes L se denominarán lenguajes recursivamente enumerables cuando su función característica sea parcialmente computable, i.e. cuando $f : \Sigma^* \longrightarrow \{0, 1\}$ sea computable y:

- $L \subseteq D(f)$,

[2] Aunque la tradición mantiene la ceguera de Homero y, por tanto, la transmisión oral de sus versos, aceptamos como "original" cualquier versión escrita durante el período helenístico.

[3] El ADN siempre se ha considerado el "lenguaje de la vida" y parece que se cumple la máxima de Galileo: "La Naturaleza es un libro escrito con el lenguaje de las matemáticas".

- $\chi_L \mid_{D(f)} = f$.

Para perfilar la noción de función computable y problema resoluble por un algoritmo debemos avanzar aún un largo trecho, que supera los estrechos márgenes de este curso. En todo caso, comencemos tratando de precisar cómo han de entenderse las manipulaciones de objetos de Σ^* que sirven para evaluar correspondencias f.

2.2. Lenguajes Formales y Monoides

La operación esencial sobre Σ^* es la concatenación (también llamada adjunción) de palabras:

$$\cdot : \Sigma^* \times \Sigma^* \longrightarrow \Sigma^*$$
$$(x, y) \longmapsto x \cdot y$$

es decir, si $x = x_1 \cdots x_n$ e $y = y_1 \cdots y_m$, entonces

$$x \cdot y = x_1 \cdots x_n y_1 \cdots y_m.$$

Denotemos por $\lambda \in \Sigma^*$ la palabra vacía (para distinguirla del lenguaje vacío $\emptyset$, usando la notación estándar de Teoría de Conjuntos (cf. Apéndice A para más detalles).

Lema 2.2.1. *$(\Sigma^*, \cdot)$ es un monoide[4], donde λ es el elemento neutro. La longitud define un morfismo de monoides[5] entre Σ^* y el conjunto de los número naturales. El monoide Σ^* es abeliano[6] si y solamente si el cardinal de Σ es uno.*

Demostración. Ejercicio obvio. □

Lema 2.2.2. *Si Σ es un alfabeto finito, el conjunto Σ^* es numerable, esto es, es biyectable con el conjunto $\mathbb{N}$ de los números naturales.*

Demostración. Para dar una prueba de este enunciado basta con fijar un buen orden en Σ^*. Un ejercicio razonable consiste en definir el buen orden basado en "lexicográfico + longitud" que define la biyección. Recuérdese que el orden lexicográfico es el orden usual del "diccionario", i.e. basado en establecer un orden en el alfabeto Σ (en ocasiones lo llamarán alfabético). Es decir, sea $\ll$ un orden total en Σ (que existe por ser Σ finito). Supongamos que los elementos de Σ quedan ordenados mediante:

$$\Sigma := \{\alpha_1 \ll \alpha_2 \ll \cdots \ll \alpha_r\}.$$

Definimos para $x = x_1 \ldots x_n, y = y_1 \ldots y_m \in \Sigma^*$ la relación de orden siguiente:

$$x \leq y \Longleftrightarrow \begin{cases} \text{o bien} & n = |x| < |y| = m, \\ \text{o bien} & |x| = |y| = n = m, \quad \begin{cases} \exists k \leq n = m, \forall i \leq k-1, \\ x_i = y_i, \\ x_k \ll y_k \end{cases} \\ \text{o bien} & x = y. \end{cases}$$

Esta relación de orden es un buen orden en Σ^* y permite una identificación (biyección) entre Σ^* y $\mathbb{N}$, asociando a cada elemento de Σ^*, el número de elementos menores que el:

$$\begin{array}{lcl} \lambda & \longmapsto & 0 \\ \alpha_1 & \longmapsto & 1 \\ \alpha_2 & \longmapsto & 2 \\ & \cdots & \\ \alpha_1\alpha_1 & \longmapsto & r+1 \\ \alpha_1\alpha_2 & \longmapsto & r+2 \\ & \cdots & \end{array}$$

[4] Recordemos que un monoide es un conjunto X con una operacón $* : X \times X \longrightarrow X$ que verifica la propiedad asociativa y de tal modo que X contiene un elementos neutro.

[5] Una transformación $f : (G, *) \longrightarrow (T, \perp)$, que verifica $f(\lambda) = \lambda$ y $f(x * y) = f(x) \perp f(y)$, es decir, respeta el elemento neutro y la operación entre dos elementos del primer monoide se transforma en la operación entre las imágenes.

[6] Todos sus elementos conmutan al operarlos.

□

Nótese que como consecuencia (Corolario) se tienen las propiedades siguientes:

Corollario 2.2.3. *Sea Σ un alfabeto finito y $L \subseteq \Sigma^*$ un lenguaje. Entonces, L es un conjunto contable (i.e. es finito o numerable).*
Más aún, el cardinal de los posibles lenguajes $L \subseteq \Sigma^*$ coincide con el número de subconjuntos de $\mathbb{N}$ y, por tanto, verifica:

$$\sharp(\mathcal{P}(\Sigma^*)) = \sharp(\mathcal{P}(\mathbb{N})) = \sharp(\mathbb{R}) = 2^{\aleph_0}.$$

En particular, hay una cantidad infinita no numerable de lenguajes sobre un alfabeto finito (cf. el ejemplo A.5.4).

2.2.1. Operaciones Básicas con palabras.

Además de la concatenación de palabras, podemos destacar las siguientes:

- *Potencia de Palabras.* Se define recursivamente a partir de la concatenación. Así, dada una palabra $\omega \in \Sigma^*$ y un número natural $n \in \mathbb{N}$, definimos la potencia ω^n, mediante:
  - Definimos $\omega^0 = \lambda$,
  - Para $n \geq 1$, definimos
  $$\omega^n := \omega \cdot \omega^{n-1}.$$
- *Reverso de una Palabra:* Se trata de una biyección
  $$^R : \Sigma^* \longrightarrow \Sigma^*,$$
  dada mediante:
  - Si $\omega = \lambda$, $\lambda^R = \lambda$,
  - Si $\omega = x_1 \cdots x_n \in \Sigma^*$, con $x_i \in \Sigma$, definimos
  $$\omega^R := x_n x_{n-1} \cdots x_1 \in \Sigma^*.$$

Un lenguaje que tendrá cierta relevancia en nuestras discusiones posteriores es el *Palíndromo* $\mathcal{P} \subseteq \{0,1\}^*$ y que viene dado por la siguiente igualdad:

$$\mathcal{P} := \{\omega \in \{0,1\}^* \ : \ \omega^R = \omega\}.$$

2.2.2. Operaciones Elementales con Lenguajes

Vamos a considerar las siguientes operaciones básicas con lenguajes formales. Tendremos fijado un alfabeto Σ,

- *Unión de Lenguajes:* De la manera obvia como subconjuntos. Dados $L_1, L_2 \subseteq \Sigma^*$, definimos:
  $$L_1 \cup L_2 := \{\omega \in \Sigma^* \ : \ [\omega \in L_1] \vee [\omega \in L_2]\}.$$
- *Concatenación de Lenguajes:* Dados $L_1, L_2 \subseteq \Sigma^*$, definimos su concatenación:
  $$L_1 \cdot L_2 := \{\omega_1 \cdot \omega_2 \in \Sigma^* \ : \ \omega_1 \in L_1, \ \omega_2 \in L_2\}.$$
- *Potencia de Lenguajes:* Se define recursivamente.
  - Si $n = 0$, $L^0 = \{\lambda\}$.
  - Si $n \geq 1$, $L^n := L \cdot (L^{n-1})$.

Observación 2.2.1. Obsérvese que $L_1 \cdot L_2$ no es, en general, igual a $L_2 \cdot L_1$. Tampoco es cierto que si $L_1 \cdot L_2 = L_2 \cdot L_1$ entonces se tiene $L_1 = L_2$. El ejemplo más sencillo de esto es $\Sigma = \{a\},\ L_1 = \{a\},\ L_2 = \{aa\}$.

Proposición 2.2.4 (Distributivas)**.** *Con las anteriores notaciones, se tienen las siguientes propiedades para lenguajes L_1, L_2 y L_3 contenidos en Σ^*:*

$$L_1 \cdot (L_2 \cup L_3) = L_1 \cdot L_2 \cup L_1 \cdot L_3.$$

$$(L_1 \cup L_2) \cdot L_3 = L_1 \cdot L_3 \cup L_2 \cdot L_3.$$

Otras operaciones importantes entre lenguajes, hacen referencia al cálculo de la clausura transitiva por la operación de adjunción :

- *Clausura transitiva o monoide generado por un lenguaje:* Dado un lenguaje $L \subseteq \Sigma^*$ definimos el monoide L^* que genera mediante la igualdad siguiente:

$$L^* := \bigcup_{n \in \mathbb{N}} L^n.$$

- *Clausura positiva de un lenguaje:* Dado un lenguaje $L \subseteq \Sigma^*$ definimos la clausura positiva L^+ que genera mediante la igualdad siguiente:

$$L^+ := \bigcup_{n \geq 1} L^n.$$

Observación 2.2.2. Es obvio que L^* es la unión (disjunta si $\lambda \notin L$) entre L^+ y $\{\lambda\}$. En otro caso (i.e. si $\lambda \in L$), ambos lenguajes coinciden.

2.2.3. Sistemas de Transición

Una de las ideas esenciales en un proceso algorítmico es que "se van dando pasos hasta obtener un resultado". Lo del número finito de pasos lo dejamos para un poco después. Surge así la noción de
Sistema de Transición, Sistema Deductivo, Sistema de Producciones, Sistema de semi–Thue etc.

Definición 2.2.1. Llamaremos sistema de transición a todo par $(S, \to)$, donde S es un conjunto (que se denomina espacio de configuraciones) y $\to \subseteq S \times S$ es una relación.

Una *sucesión de computación* en el sistema de transición $(S, \to)$ es simplemente una sucesión finita de elementos de S:

$$s_1, \ldots, s_n$$

donde $(s_i, s_{i+1}) \in \to$ (se dice que la configuración s_{i+1} es deducible de s_i en un sólo paso deductivo). Normalmente, uno prefiere escribir $s_i \to s_{i+1}$ en lugar de $(s_i, s_{i+1}) \in \to$. Con ello, una computación en el sistema de transición $(S, \to)$ es simplemente:

$$s_1 \to \cdots \to s_n$$

Se dice que el sistema de transición es determinístico si cada $s \in S$ tiene un sólo sucesor a lo sumo y es indeterminista en caso contrario.

Definición 2.2.2. Dada una configuración $s \in S$, diremos que una configuración $s' \in S$ es deducible de s y lo denotaremos por $s \vdash s'$, si existe una sucesión de computación

$$s = s_1 \to \cdots \to s_n = s'$$

La relación que debe existir entre los datos de un problema y su resolución es de ser deducible para algún sistema de transición. En cada caso clarificaremos los sistemas de transición del modelo de cálculo introducido (es decir, la acción dinámica del modelo definido).

Observación 2.2.3. Nótese la obvia analogía entre sistemas de transición y grafos (potencialmente con un número infinito de nodos). De hecho, un grafo orientado es simplemente un sistema de transición con un conjunto de configuraciones finito.

La siguiente sección introducirá el concepto de gramáticas formales.

2.3. Gramáticas Formales

A. Thue[7] fue un matemático noruego que en 1914 introdujo la noción de sistema de reescritura. El interés de Thue era el análisis de los problemas de palabra para grupos y semi-grupos. Habrá que esperar a los trabajos de Noam Chomsky a finales de los años 50 para tener una estructuración de los sistemas de transición en el formato de gramáticas formales que, inicialmente, intentaba utilizar para modelizar los lenguajes naturales.

Definición 2.3.1 (Gramáticas Formales)**.** Una gramática formal es una cuaterna $G = (V, \Sigma, Q_0, P)$, donde:

- V es un conjunto finito llamado alfabeto de símbolos no terminales o, simplemente, alfabeto de variables.
- Σ es otro conjunto finito, que verifica $V \cap \Sigma = \emptyset$ y se suele denominar alfabeto de símbolos terminales.
- $Q_0 \in V$ es una "variable" distinguida que se denomina *símbolo inicial*.
- $P \subseteq (V \cup \Sigma)^* \times (V \cup \Sigma)^*$ es un conjunto finito llamado conjunto de producciones (o, simplemente, sistema de reescritura).

2.3.1. Sistema de Transición Asociado a una Gramática.

Para poder definir la dinámica asociada a una gramática, necesitamos asociarle un sistema de transición.

Definición 2.3.2. Sea $G = (V, \Sigma, Q_0, P)$ una gramática, definiremos el sistema de transición asociado $(S_G, \to_G)$ dado por las propiedades siguientes:

- El espacio de configuraciones será dado por:
$$S_G := (V \cup \Sigma)^* .$$
- Dadas dos configuraciones $s_1, s_2 \in S_G$, decimos que $s_1 \to_G s_2$ si se verifica la siguiente propiedad:
$$\exists x, y, \alpha, \beta \in S_G = (V \cup \Sigma)^*, \text{ tales que}$$
$$s_1 := x \cdot \alpha \cdot y, \quad s_2 := x \cdot \beta \cdot y, \quad (\alpha, \beta) \in P.$$

Ejemplo 2.3.1. Consideremos la gramática: $G = (V, \Sigma, Q_0, P)$, donde

$$V := \{Q_0\}, \quad \Sigma := \{a, b\}, \quad , P := \{(Q_0, aQ_0), (Q_0, \lambda)\}.$$

El sistema de transición tiene por configuraciones $S := \{Q_0, a, b\}^*$ y un ejemplo de una computación sería:

$$aaQ_0bb \to aaaQ_0bb \to aaaaQ_0bb \to aaaa\lambda bb = aaaabb.$$

Nótese que las dos primeras veces hemos usado la regla de reescritura (Q_0, aQ_0) y la última vez hemos usado (Q_0, λ).

Notación 2.3.1. Por analogía con el sistema de transición, se suelen usar la notación $A \mapsto B$ en lugar de $(A, B) \in P$, para indicar una producción. Y, en el caso de tener más de una producción que comience en el mismo objeto, se suele usar $A \mapsto B \mid C$, en lugar de escribir $A \mapsto B$, $A \mapsto C$.

[7] A. Thue. Probleme über Veränderungen von Zichereihen nach gegebenen reglen. *Regeln. Skr. Videnk. Selks. Kristiania I, Mat. Nat. Kl. :* **10** (1914).

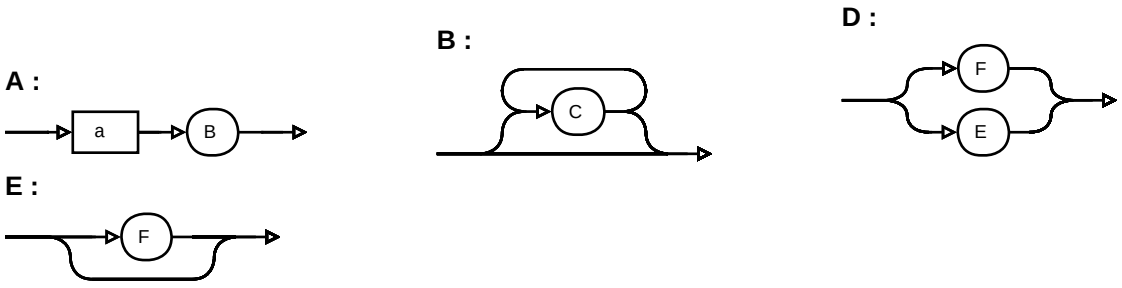


Figura 2.1: Representación EBNF A:"a"B, B:C*, D:F | E, E:F?

2.3.2. Otras Notaciones para las Producciones.

2.3.2.1. Notación BNF.

La notación de Backus-Naur, también conocida como **BNF** (de Backus–Naur Form), es una notación alternativa para las gramáticas y que remonta su origen a la descripción que, del sánscrito, hizo el gramático hindú Panini. No es una notación estandarizada, aunque está bien establecida. Entre otroas cosas porque los primeros usuarios de esta notación inisistieron en diversas notaciones para el símbolo $\mapsto$. Aquí usaremos el `‘‘estándar Wiki’’` por llamarlo de algún modo. Se trata de hacer los siguientes cambios

- Las variables $X \in V$ se representan mediante $\langle X \rangle$.
- Los símbolos terminales (del alfabeto Σ) se presentan entre comillas (“a”, “b”, “c”, ...)
- El símbolo asociado a las producciones $\mapsto$ es reemplazado por ::=.

Así, la gramática descrita en el Ejemplo 2.3.1 anterior vendría dada por:

$$V = \{\langle Q \rangle\}, \quad \Sigma = \{\text{“a”}, \text{“b”}\},$$

y las producciones estarían dadas por las reglas:

$$\langle Q \rangle = \text{“a”} \langle Q \rangle \mid \lambda$$

2.3.2.2. Notación EBNF.

Esta notación es extensión de la notación BNF. Es un estándar ISO-1497 y es utilizada (con algunas modificaciones) en los generadores de compiladores, como ANTLR.

Básicamente, añade funcionalidad a la notación BNF, permitiendo repeticiones o diferentes opciones. Varios ejemplos están dados en la figura encabezando la página (nótese la diferencia para los símbolos terminales y no terminales). Los siguientes son las principales modificaciones con respecto a la notación BNF,

- Las variables $X \in V$ no son modificadas.
- Los símbolos terminales (del alfabeto Σ) se representan entre comillas simples.
- El símbolo asociado a las producciones $\mapsto$ es reemplazado por :.
- Se introducen nuevos símbolos para representar repeticiones $*$ (ninguna, una o mas repeticiones) $+$ (una repetición al menos).
- ? indica que la expresión puede ocurrir o no.

Se deja como ejercicio al alumno hallar la expresión de la gramática

$$\begin{aligned}\langle Q\rangle = &\quad a\langle Q\rangle\\ \langle Q\rangle = &\quad \lambda.\end{aligned}$$

con notación EBNF.
Independiente de las notaciones, el elemento clave es la noción de lenguaje generado por una gramática. En lo que respecta a este manuscrito, utilizaremos la notación usada en páginas anteriores (equivalente a BNF) y evitaremos (por excesiva e innecesaria para nuestros propósitos) la notación EBNF.

Definición 2.3.3 (Lenguaje Generado por una gramática)**.** Sea G una gramática definida como $G := (V, \Sigma, Q_0, P)$. Llamaremos lenguaje generado por la gramática G al lenguaje $L(G) \subseteq \Sigma^*$ dado por:

$$L(G) := \{x \in \Sigma^* \ : \ Q_0 \vdash_G x\},$$

es decir, a las palabras sobre el alfabeto de símbolos terminales alcanzables (dentro del sistema de transición asociado) desde el símbolo inicial de la gramática.

2.4. Jerarquía de Chomsky

Chomsky pretende la modelización de los lenguajes (formales y naturales) mediante gramáticas en su trabajo [Ch, 57]. El uso de máquinas con un número finito de estados (autómatas) ya aparece en [ChMi, 57].
Es en sus trabajos del año 59 (ca.[Ch, 59a] y [Ch, 59b]) donde aparece la clasificación que discutiremos en las páginas que siguen.

Definición 2.4.1 (Gramáticas Regulares o de Tipo 3)**.** Definiremos las gramáticas con producciones lineales del modo siguiente:

- Llamaremos gramática lineal por la izquierda a toda $G := (V, \Sigma, Q_0, P)$ gramática tal que todas las producciones de P son de uno de los dos tipos siguientes:
  - $A \mapsto a$, donde $A \in V$ y $a \in \Sigma \cup \{\lambda\}$.
  - $A \mapsto aB$, donde $A, B \in V$ y $a \in \Sigma \cup \{\lambda\}$.
- Llamaremos gramática lineal por la derecha a toda gramática $G = (V, \Sigma, Q_0, P)$ tal que todas las producciones de P son de uno de los dos tipos siguientes:
  - $A \mapsto a$, donde $A \in V$ y $a \in \Sigma \cup \{\lambda\}$.
  - $A \mapsto Ba$, donde $A, B \in V$ y $a \in \Sigma \cup \{\lambda\}$.
- Llamaremos **gramáticas regulares** a las gramáticas lineales por la izquierda o lineales por la derecha.

La dualidad (y simetría) entre las garmáticas lineales a izquierda o lineales a derecha es obvia y nos quedaremos solamente con las gramáticas lineales a izquierda.

Definición 2.4.2 (Lenguajes Regulares)**.** Un lenguaje $L \subseteq \Sigma^*$ se denomina un lenguaje regular si existe una gramática regular $G = (V, \Sigma, Q_0, P)$ que lo genera.

Por definición una producción puede ser una transformación del tipo $\alpha A\beta \mapsto \omega$, donde $\alpha, \beta \in (\Sigma \cup V)^*$, $A \in V$. A las palabras α y β se las denomina *contexto* de la producción (o contexto de la variable A en esa producción). Así, una producción libre de contexto es una producción en la que ninguna variables tiene contexto, esto es, de la forma $A \mapsto \omega$, con $A \in V$.

Definición 2.4.3 (Gramáticas libres de contexto o de Tipo 2)**.** Llamaremos gramática libre de contexto a toda $G = (V, \Sigma, Q_0, P)$ gramática tal que todas las producciones de P son del tipo siguiente:

$$A \mapsto \omega, \text{ donde } A \in V \text{ y } \omega \in (\Sigma \cup V)^*.$$

Un lenguaje **libre de contexto** es un lenguaje generado por una gramática libre de contexto.

Definición 2.4.4 (Gramáticas sensibles al contexto o de Tipo 1)**.** Llamaremos gramática sensible al contexto a toda gramática $G = (V, \Sigma, Q_0, P)$ tal que todas las producciones de P son del tipo siguiente:

$$\alpha A \beta \mapsto \alpha \gamma \beta, \text{ donde } A \in V \text{ y } \alpha, \beta \in (\Sigma \cup V)^*,\ \gamma \in (\Sigma \cup V)^*, \gamma \neq \lambda$$

es decir, en todas las producciones hay al menos una variable en la parte "izquierda" de la producción y no se reemplaza por la palabra vacía.
Un lenguaje **sensible al contexto** es un lenguaje generado por una gramática sensible al contexto.

Definición 2.4.5 (Gramáticas formales, sistemas de semi–Thue o de Tipo 0)**.** Llamaremos gramática formal (o sistema de semi–Thue o sistema de reescritura finitamente generado y finitamente presentado) a toda gramática $G = (V, \Sigma, Q_0, P)$ que admite todo tipo de producciones, esto es, sus producciones son de la forma

$$\alpha \mapsto \omega, \text{ donde } \alpha, \omega \in (\Sigma \cup V)^*, \alpha \neq \lambda.$$

En las gramáticas de tipo 0 (las más generales) admitimos que haya producciones sin ninguna variable en el lado izquierdo de la producción.

2.5. Sistemas de Thue: Problemas de Palabra

Las gramáticas de tipo 0 son también *Sistemas de Semi–Thue* (véase, por ejemplo, la referencia en [DaWe, 94]) en honor del matemático que las introdujo. Hblaremos de sistemas de Semi–Thue finitamente generados y finitamente presentados cuando el alfabeto subyacente sea finito y las reglas de reescritura sean dadas en número finito. El objetivo de Thue era analizar el siguiente tipo de problemas.

Problema (Problema de Palabra para Sistemas de Semi–Thue)**.** Dado un sistema de semi–Thue (Σ, R) y dados $x, y \in \Sigma^*$, decidir si $x \vdash_R y$.

Problema (Problema de Palabra en Semigrupos)**.** Dado R un sistema de semi–Thue sobre un alfabeto finito Σ, consideramos la estructura de semigrupo con unidad de Σ^* (monoide). Dos palabras $x, y \in \Sigma^*$ se dicen relacionadas mediante R, si $x \vdash_R y$ en el sistema de transición asociado (i.e. si y es deducible de x).

Un sistema de Thue es un sistema de semi–Thue en el que R verifica la siguiente propiedad adicional :

$$\forall x, y \in \Sigma^*,\ \ (x, y) \in R \Leftrightarrow (y, x) \in R$$

Entonces, R define una relación de equivalencia $\vdash_R$ en Σ^* y podemos considerar el conjunto cociente :

$$S(\Sigma, R) := \Sigma^* / \vdash_R$$

Claramente se tiene que $S(\Sigma, R)$ es un semigrupo, cuyos elementos son las clases $[x]$ definidas por elementos $x \in \Sigma^*$.
El problema de la palabra para semigrupos se define mediante :
Dados un sistema de Thue (Σ, R) *y dados* $x, y \in \Sigma^*$, *decidir si* $[x] = [y]$

Observación 2.5.1. Esta versión del problema de la palabra está relacionada directamente con un hábito muy común en matemáticas. Supongamos que quiero trabajar con un semigrupo S, no necesariamente conmutativo. Para describirlo, todos pondríamos un conjunto de generadores (digamos $\{\gamma_1, \ldots, \gamma_n\}$). Sabidos los generadores, sabemos que los elementos son todos de la forma :

$$\gamma_{s(1)} \cdots \gamma_{s(m)}$$

donde $s : \{1, \ldots, m\} \longrightarrow \{1, \ldots, n\}$ es una aplicación, con $m \in \mathbb{N}$. El problema de una representación –tal cual ésta– es que uno no puede hacer cosas tan elementales como comparar dos elementos dados (obsérvese que nadie dijo que las cosas conmuten ni que la representación sea única). Por lo tanto, uno debería dar, al menos, las relaciones entre los generadores (que son inevitables). Estas relaciones tienen la pinta

$$\gamma_{r_i(1)} \cdots \gamma_{r_i(m_i)} = \gamma_{s_i(1)} \cdots \gamma_{s_i(k_i)}$$

para $1 \leq i \leq N$, siendo r_i y k_i aplicaciones con rango $\{1, \ldots, n\}$. Nos conformamos con que sólo haya un número finito de relaciones entre los generadores. Claramente, tenemos un sistema de reescritura sobre el alfabeto $\Sigma = \{1, \ldots, n\}$, siendo

$$R := \{(r_i(1) \cdots r_i(m_i), s_i(1) \cdots s_i(k_i)) \ : \ 1 \leq i \leq N\}$$

Es obvio que nuestro semigrupo S inicial es justamente $S(\Sigma, R)$. Luego el problema de las palabras viene a decir si somos capaces de identificar o distinguir dos elementos de un semigrupo dado por sus generadores y sus relaciones. La respuesta , dada por E. Post[8] en 1947 es que el problema de la palabra para semi–grupos finitamente presentados es indecidible (luego, insoluble).

Teorema 2.5.1 (Post[9], 47)**.** *Los problemas de palabras para sistemas de semi–Thue, y semigrupos son insolubles algorítmicamente.*

El problema de palabra en grupos El problema anterior se sofistica un poco más, si en lugar de semigrupo hablamos de grupos. Un grupo finitamente generado (no necesariamente abeliano) no es sino un semigrupo asociado a un sistema de Thue (Σ, R) que, además verifica la propiedad siguiente :
existe una aplicación $\sigma : \Sigma \longrightarrow \Sigma$ *tal que :*

$$\forall a \in \Sigma, (a\sigma(a), \lambda) \in R)$$

donde λ *es la palabra vacía.* Escribamos $G(\Sigma, R)$ por el grupo cociente Σ^*/R
El problema de la palabra es también :
`Dado un sistema de grupo` (Σ, R) `y dadas` $x, y \in \Sigma^*$`, decidir si` $[x] = [y]$ `en` $G(\Sigma, R)$`.`
Tras mucho esfuerzo P. Novikov[10] (en 1955) y W.W. Boone[11] (con una demostración mucho más simple, en 1958) lograron demostrar que el enunciado siguente:

Teorema 2.5.2 (Novikov–Boone)**.** *El problema de palabra para grupos finitamente presentados y finitamente generados es insoluble algorítmicamente.*

Como aún no sabemos lo que es un algoritmo, dejemos la demostración para alguna referencia bibliográfica (cf. [Weh, 97]). Nótese que los problemas de palabras de los sistemas de producciones también pueden interpretarse como una primera aproximación al problema de lo deducible en una teoría formal. Pero eso es otro asunto.
Problema de correspondencia de Post. Se trata de otro problema basado en los sistemas de reescritura y que resulta, también insoluble algorítmicamente (cf. E. Post[12] en 1946).

[8] E. Post. "Recursive unsolvability of a Problem of Thue". *J. of Symb. Logic* **12** (1947) 1–11.
[9] E. Post. "Recursive unsolvability of a Problem of Thue". *J. of Symb. Logic* **12** (1947) 1–11.
[10] P.S. Novikov. "On the algorithmic unsolvability of the word problem in group theory". *Proceedings of the Steklov Institute of Mathematics* **44** (1995), 1-143.
[11] William W. Boone. "The word problem". *Proceedings of the National Academy of Sciences* **44** (1958) 1061-1065.
[12] E. Post . "A variant of a recursively unsolvable problem." *Bull. A.M.S.* **52** (1946) 264–268.

Problema (Post Correspondence)**.** Consideremos un sistema de semi–Thue (Σ, R) y sus elementos como piezas de dominó :

$$R := \{(x_1, y_1), \ldots, (x_n, y_n)\}$$

y las piezas

$$\mathcal{D}_i :=| \frac{x_i}{y_i} |$$

Decidir si existe una secuencia de fichas

$$\mathcal{D}_{s(1)} \cdots \mathcal{D}_{s(n)}$$

tal que lo que aparece escrito en las partes superiores de los dominós coincide con lo escrito debajo.

Por ejemplo, sea R (Post prefiere Pairing Lists i.e. PL)

$$R := \{(a, aa), (bb, b), (a, bb)\}$$

para el alfabeto $\Sigma := \{a, b\}$. La siguiente es una solución :

$$| \frac{a}{aa} || \frac{a}{bb} || \frac{bb}{b} || \frac{bb}{b} |$$

Teorema 2.5.3 (Post, 46)**.** *El problema de la correspondencia es insoluble por métodos algorítmicos. En otras palabras, no existe (ni se puede encontrar) un algoritmo que resuelva el problema de correspondencia de Post.*

La prueba de la Indecidibilidad de este Problema puede verse en el [Weh, 97] o en el [DaWe, 94] , entre otros.

Capítulo 3

Expresiones Regulares

Contents

3.1. Las Nociones y Algoritmos Básicos 47
3.1.1. Las Nociones 47
3.1.2. La Semántica de las expresiones regulares. 48
3.2. De RE's a RG's: Método de las Derivaciones 49
3.2.1. Derivación de Expresiones Regulares 49
3.2.2. Cómo no construir la Gramática 51
3.2.3. Derivadas Sucesivas: el Método de las derivaciones 52
3.3. De RG's a RE's: Uso del Lema de Arden 54
3.3.1. Ecuaciones Lineales. Lema de Arden 54
3.3.2. Sistema de Ecuaciones Lineales Asociado a una Gramática. 55
3.4. Problemas y Cuestiones. 56
3.4.1. Cuestiones Relativas a Lenguajes y Gramáticas. 56
3.4.2. Cuestiones Relativas a Expresiones Regulares. 57
3.4.3. Problemas Relativos a Lenguajes Formales y Gramáticas 57
3.4.4. Problemas Relativos a Expresiones Regulares 59

El problema que se pretende resolver mediante la introducción de las expresiones regulares es el de obtener algún tipo de descriptores para los lenguajes generados por las gramáticas regulares (las gramáticas de Tipo 3 o Regulares en la jerarquía de Chomsky), además de utilizarlos en la notación EBNF.

3.1. Las Nociones y Algoritmos Básicos

3.1.1. Las Nociones

Siendo éste un curso de lenguajes formales, utilizaremos la metodología propia del área. Empezaremos definiendo las reglas de formación (la gramática) de las expresiones regulares. A continuación las dotaremos de significado (la semántica) y veremos los recursos que nos ofrece esta nueva herramienta.

Definición 3.1.1. Sea Σ un alfabeto finito. Llamaremos expresión regular sobre el alfabeto Σ a toda palabra sobre el alfabeto Σ_1 definido por la siguiente igualdad:

$$\Sigma_1 := \{'\emptyset', '\lambda', +, \cdot, (,), ^* \} \cup \Sigma,$$

conforme a las reglas siguientes:

- Las siguientes son expresiones regulares:

  - El símbolo $'\emptyset'$ es una expresión regular,
  - el símbolo $'\lambda'$ es una expresión regular,
  - y el símbolo a es una expresión regular, para cualquier símbolo a en el alfabeto Σ,

- Si α y β son expresiones regulares, también lo son las construidas mediante las reglas siguientes:
  - $(\alpha + \beta)$ es una expresión regular,
  - $(\alpha \cdot \beta)$ es una expresión regular,
  - $(\alpha)^*$ es una expresión regular,

Observación 3.1.1. Por comodidad de la escritura (y sólo en el caso de que no haya ninguna posibilidad de ambigüedades) se suprimen los paréntesis y los símbolos de producto $(\cdot)$.

Observación 3.1.2. También por simplificación de la escritura escribiremos simplemente $\emptyset$ y λ en lugar de $'\emptyset'$ y $'\lambda'$ y siempre que no haya confusión entre su sentido como expresión regular y su uso habitual como conjunto vacío o como palabra vacía.

Observación 3.1.3. La anterior definición no es sino la definición de un lenguaje sobre el alfabeto Σ_1: el lenguaje formado por las expresiones regulares. Dicha gramática se pude representar mediante una única variable $\langle RE \rangle$, el alfabeto Σ_1 y las producciones siguientes:

$$\langle RE \rangle \mapsto' \emptyset' \mid' \lambda' \mid a, \quad \forall a \in \Sigma.$$

$$\langle RE \rangle \mapsto (\langle RE \rangle + \langle RE \rangle) \mid (\langle RE \rangle \cdot \langle RE \rangle) \mid (\langle RE \rangle)^*.$$

Ejemplo 3.1.1. Tomemos el alfabeto $\Sigma := \{a, b\}$. Son expresiones regulares las secuencias de símbolos (palabras) siguientes:

$$a \cdot a + b^* a, ab^* ba, \ldots$$

No serán expresiones regulares cosas del tipo:

$$(+b^* \emptyset)^*, \ldots$$

3.1.2. La Semántica de las expresiones regulares.

A cada objeto sintáctico, como lo es una expresión regular, conviene añadirle el mecanismo de asignación de significado (semántica). En el caso de expresiones regulares asignaremos *un único significado* a cada expresión como *el lenguaje formal que describe*.

Definición 3.1.2. Sea Σ un alfabeto finito. A cada expresión regular sobre el alfabeto α le asignaremos un lenguaje formal $L(\alpha) \subseteq \Sigma^*$ conforme a las siguientes reglas:

- En el caso de que α sea una palabra de longitud 1, seguiremos las reglas siguientes:
  - Si $\alpha = \emptyset$, entonces $L(\alpha) = \emptyset$,
  - Si $\alpha = \lambda$, entonces $L(\alpha) = \{\lambda\}$,
  - Si $\alpha = a \in \Sigma$, entonces $L(\alpha) = \{a\}$,

- Aplicando las reglas recursivas, si α y β son dos expresiones regulares sobre el alfabeto Σ usaremos las reglas siguientes:
  - $L(\alpha + \beta) = L(\alpha) \cup L(\beta)$,
  - $L(\alpha \cdot \beta) = L(\alpha) \cdot L(\beta)$,
  - $L(\alpha^*) = L(\alpha)^*$.

Ejemplo 3.1.2. A modo de ejemplo, sea $\alpha := 0^*10^*$ la expresión regular sobre el alfabeto $\Sigma := \{0, 1\}$. Entonces,

$$L(0^*10^*) = L(0)^* \cdot L(1) \cdot L(0)^* = \{0^m 10^n \ : \ n, m \in \mathbb{N}\}.$$

Definición 3.1.3. Diremos que dos expresiones regulares α y β son tautológicamente equivalentes (o, simplemente, equivalentes) si se verifica:

$$L(\alpha) = L(\beta).$$

Escribamos $\alpha \equiv \beta$ para indicar equivalencia tautológica.

Algunas de las propiedades básicas de la asignación semántica de lenguajes a expresiones regulares se resumen en la siguiente Proposición, cuya demostración es completamente obvia.

Proposición 3.1.1 (Propiedades Básicas)**.** *Sea Σ un alfabeto finito, se verifican las siguientes propiedades para expresiones regulares α, β, γ sobre α:*

i) Asociativas.
$$\alpha \cdot (\beta \cdot \gamma) \equiv (\alpha \cdot \beta) \cdot \gamma, \quad \alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma.$$

ii) Conmutativa (sólo para $+$)[1]:
$$\alpha + \beta \equiv \beta + \alpha.$$

iii) Elementos Neutros:
$$\alpha + \emptyset \equiv \alpha, \quad \alpha \cdot \lambda \equiv \alpha, \quad \alpha \cdot \emptyset \equiv \emptyset.$$

iv) Idempotencia:
$$\alpha + \alpha \equiv \alpha.$$

v) Distributivas:
$$\alpha \cdot (\beta + \gamma) \equiv \alpha \cdot \beta + \alpha \cdot \gamma.$$
$$(\alpha + \beta) \cdot \gamma \equiv \alpha \cdot \gamma + \beta \cdot \gamma.$$

vi) Invariantes para *:
$$\lambda^* \equiv \lambda, \quad \emptyset^* \equiv \emptyset.$$

vii) La notación α^+:
$$\alpha^* \cdot \alpha \equiv \alpha \cdot \alpha^* \equiv \alpha^+.$$

viii) $\alpha^* = \lambda + \alpha^+$

ix) Relación de * con la suma:
$$(\alpha + \beta)^* \equiv (\alpha^* \beta^*)^*.$$

3.2. De RE's a RG's: Método de las Derivaciones

3.2.1. Derivación de Expresiones Regulares

En esta Sección dedicaremos algún tiempo a fijar una de las operaciones básicas en el tratamiento de expresiones regulares: la derivación.

Definición 3.2.1. Sea Σ un alfabeto finito, $a \in \Sigma$ un símbolo del alfabeto, y α una expresión regular sobre el alfabeto Σ. Llamaremos derivada de α con respecto al símbolo α a la expresión regular $\frac{\partial \alpha}{\partial a}$ definida mediante la regla recursiva siguiente:

[1] Aunque la insistencia sea innecesaria, es común olvidar que $2 \cdot 3$ no es igual que $3 \cdot 2$. Cosas de malos hábitos.

- Para expresiones regulares de longitud 1, tenemos las definiciones siguientes:

$$\frac{\partial \emptyset}{\partial a} = \emptyset, \quad \frac{\partial \lambda}{\partial a} = \emptyset, \quad \frac{\partial b}{\partial a} = \emptyset, \quad \forall b \in \Sigma,\ b \neq a.$$

$$\frac{\partial a}{\partial a} = \lambda.$$

- Si α y β son dos expresiones regulares sobre Σ, definiremos:

$$\frac{\partial(\alpha)^*}{\partial a} = \frac{\partial(\alpha)}{\partial a} \cdot \alpha^*,$$

$$\frac{\partial(\alpha+\beta)}{\partial a} = \frac{\partial \alpha}{\partial a} + \frac{\partial \beta}{\partial a},$$

$$\frac{\partial(\alpha \cdot \beta)}{\partial a} = \frac{\partial \alpha}{\partial a} \cdot \beta + t(\alpha)\frac{\partial \beta}{\partial a},$$

donde $t(\alpha)$ es la función dada por la identidad siguiente:

$$t(\alpha) := \begin{cases} \lambda & \text{si } \lambda \in L(\alpha), \\ \emptyset & \text{en caso contrario.} \end{cases}$$

Observación 3.2.1. La derivada de una expresión regular con respecto a un símbolo de un alfabeto finito es, claramente, una derivada parcial y, por tanto, está perfectamente descrita mediante el símbolo $\frac{\partial \alpha}{\partial a}$. Sin embargo, el símbolo ∂ parece poner nerviosos a ciertos autores, por lo que también es costumbre (solamente entre los nerviosos) usar el símbolo menos correcto (pero menos enervante) $D_a(\alpha)$. Dejaremos que los alumnos reescriban la definición anterior con esta nueva notación. De ahora en adelante usaremos $D_a(\alpha)$.

La propiedad fundamental por la cual derivar es una acción útil, viene dada por la siguiente Proposición (cuya prueba omitiremos por obvia).

Proposición 3.2.1. *Con las notaciones anteriores, para cada expresión regular α sobre un alfabeto Σ, la derivada $D_a(\alpha)$ es una expresión regular que verifica la siguiente propiedad:*

$$L(D_a(\alpha)) = \{\omega \in \Sigma^* \ :\ a\omega \in L(\alpha)\}.$$

Demostración. Como pista para la demostración, digamos que sale de manera inmediata a partir de la definición recursiva de expresiones regulares. □

Una identificación más clara de la relación de una palabra con sus derivadas viene dada por la siguiente Proposición (que resume la regla de Leibnitz para polinomios homogéneos multivariados).

Proposición 3.2.2 (Regla de Leibnitz para Expresiones Regulares)**.** *Dada una expresión regular α sobre un alfabeto finito Σ, supongamos que $\Sigma = \{a_1, \ldots, a_n\}$. Entonces,*

$$\alpha \equiv a_1 D_{a_1}(\alpha) + \cdots + a_n D_{a_n}(\alpha) + t(\alpha),$$

donde $t(\alpha)$ es la función definida anteriormente.

Demostración. Mediante la proposición anterior, basta con verificar a que las palabras en $L(\alpha)$ son de los tipos (obvios) siguientes: o empiezan por algún símbolo de $a_1 \in \Sigma$ (y, por tanto, están en $a_1 D_{a_1}(\alpha)$) o es la palabra vacía (y queda sumida en la expresión $t(\alpha)$). El caso restante es que no haya ninguna palabra en $L(\alpha)$ lo cual también queda expresado por la identidad y por $t(\alpha)$. □

3.2.2. Cómo no construir la Gramática

En esta Sección demostraremos que el lenguaje descrito por una expresión regular es un lenguaje regular, es decir, que existe una gramática regular que lo genera. Más aún, daremos un algoritmo que transforma expresiones regulares en gramáticas regulares respetando los lenguajes que describen/generan: es el *Método de las Derivaciones.*

Lema 3.2.3. *Sea L_1 y L_2 dos lenguajes (regulares) sobre el alfabeto Σ generados respectivamente por gramáticas $G_1 = (V_1, \Sigma, Q_1, P_1)$ y $G_2 = (V_2, \Sigma, Q_2, P_2)$, entonces $L_1 \cup L_2$ es también un lenguaje (regular) generado por una gramática. La gramática que genera la unión es una nueva gramática $G = (V, \Sigma, Q_0, P)$ dada por las reglas siguientes:*

i) Al precio de renombrar las variables, podemos suponer que $V_1 \cap V_2 = \emptyset$ (es decir, G_1, G_2 no poseen símbolos no terminales comunes) y $P_1 \cap P_2 = \emptyset$.

ii) Introducimos una nueva variable $Q_0 \notin V_1 \cup V_2$.

iii) Finalmente, definimos $V := V_1 \cup V_2 \cup \{Q_0\}$.

iv) Y definimos $P := P_1 \cup P_2 \cup \{Q_0 \mapsto Q_1 \mid Q_2\}$.

Demostración. Con esta definición de la nueva gramática G es un mero ejercicio de demostración por inducción en el número de pasos de cálculo. □

Lema 3.2.4. *En el caso de unión finita $L = L_1 \cup \cdots \cup L_m$, el Lema anterior se puede extender de la forma obvia. Por tanto, la unión finita de lenguajes generados por gramáticas (resp. regulares) es un lenguaje generado por una gramática (resp. regulares).*

Lema 3.2.5. *Sea $L \subseteq \Sigma^*$ un lenguaje sobre el alfabeto Σ generado por una gramática (regular) $G := (V, \Sigma, q_0, P)$. Sea $a \in \Sigma$ un símbolo del alfabeto. Entonces, la siguiente gramática $G_a = (V_a, \Sigma, Q_a, P_a)$ genera el lenguaje $a \cdot L$:*

- *Sea Q_a una nueva variable (no presente en V) y definamos $V_a := V \cup \{Q_a\}$.*
- *Definamos $P_a := P \cup \{Q_a \mapsto aQ_0\}$.*

Demostración. De nuevo un mero ejercicio de demostración por inducción. Es importante señalar que si la gramática G es regular, la nueva gramática también es regular. □

Combinando la Proposición 3.2.2 con los lemas 3.2.4 y 3.2.5, uno pensaría en un argumento inductivo para generar un lenguaje dado por una expresión regular α a partir de sus derivadas. La idea, grosso modo, sería la siguiente:

Sea $L(\alpha)$ un lenguaje dado por una expresión regular α sobre un alfabeto Σ, supongamos que $\Sigma = \{a_1, \ldots, a_n\}$. entonces, la Regla de Leibnitz para expresiones regulares nos da la siguiente identidad:

$$L(\alpha) = a_1 \cdot L(D_{a_1}(\alpha)) \cup \cdots \cup a_n \cdot L(D_{a_n}(\alpha)) \cup t(\alpha).$$

A partir de esta identidad, uno pretende generar un árbol entre expresiones regulares y podría tratar de argumentar como sigue:

- Supongamos dadas gramáticas $G_1, \ldots, G_n$ que generan (respectivamente) los lenguajes $L(D_{a_1}(\alpha)), \ldots, L(D_{a_n}(\alpha))$.
- Utilizado el Lema 3.2.5, uno podría construir gramáticas $G'_1, \ldots, G'_n$ de tal modo que G'_i es la gramática que genera el lenguaje $a_i L(D_{a_i}(\alpha))$.
- Finalmente, utilizando el Lema 3.2.4 uno concluiría exhibiendo la gramática que genera el lenguaje $L(\alpha)$ a través de la identidad dada por la Regla de Leibnitz (Proposición 3.2.2).

El problema en esta forma de pensamiento es la "gradación" de las gramáticas. En esta propuesta hay implícitamente una suposición de que las expresiones regulares asociadas a las derivadas son "más pequeñas" que la expresión regular original. El concepto de "más pequeño" es inevitable para poder dar un argumento recursivo con esta construcción. Sin embargo, la intuición sobre las propiedades de las derivadas no debe confundirnos. La derivada de una expresión regular puede ser "más grande" (o de mayor "grado") que la expresión original, debido justamente al papel del operador *. Veamos algunos ejemplos:

Ejemplo 3.2.1. Sea $\Sigma = \{a, b\}$ y consideremos la expresión regular $a^* \subseteq \Sigma^*$. Consideramos las derivadas $D_a(a^*) = a^*$, $D_b(a^*) = \emptyset$. Tendremos, por Leibnitz,

$$\{a\}^* = L(a^*) = a \cdot L(a^*) + \emptyset + \{\lambda\}.$$

Claramente, la inducción pretendida nos dice que para hallar la gramática asociada a la expresión a^* **necesitamos calcular previamente la gramática asociada a la expresión** a^***!**. La respuesta a este dilema en este caso, sería la gramática siguiente:

- Dado que $\lambda \in L(a^*)$ escribamos la producción $q \mapsto \lambda$,
- Dado que $D_a(a^*) \neq \lambda, \emptyset$, escribamos la producción $q \mapsto aq$.

Nótese que, en este ejemplo, hemos identificado la variable q con la expresión regular a^* y, hemos escrito la producción $q \mapsto aq$ porque $D_a(a^*) = a^*$.

Ejemplo 3.2.2. En el anterior ejemplo, la expresión regular obtenida tras derivar no "crece" con respecto a la expresión regular original (en todo caso, se estabiliza). Pero es posible que se produzca un crecimiento (al menos en la longitud como palabra) y eso se muestra a través del ejemplo $(abc)^*$ de una expresión regular sobre el alfabeto $\Sigma = \{a, b, c\}$. Al derivar observamos:

$$D_a((abc)^*) = bc(abc)^*,$$

cuya longitud es mayor que la longitud de la expresión regular original.

3.2.3. Derivadas Sucesivas: el Método de las derivaciones

Para resolver este problema acudiremos al análisis de las derivadas sucesivas de una expresión regular.

Definición 3.2.2 (Derivadas sucesivas (de una RE))**.** Sea $\Sigma = \{a_1, \ldots, a_n\}$ un alfabeto finito, $\omega \in \Sigma^*$ una palabra sobre el alfabeto y α una expresión regular. Definiremos la derivada $D_\omega(\alpha)$ mediante el proceso siguiente:

- Si $\omega = \lambda$ es la palabra vacía, $D_\lambda(\alpha) = \alpha$.
- Si $|\omega| = 1$ (es una palabra de longitud 1) y, por tanto, $\omega = a_i \in \Sigma$, definimos $D_\omega(\alpha) = D_{a_i}(\alpha)$, conforme a la definición de derivada anterior.
- Si $|\omega| = n \geq 2$ (es una palabra de longitud n) y, por tanto, existe $a_i \in \Sigma$ y existe $\omega_1 \in \Sigma^*$, con $|\omega_1| = n - 1$, tal que
$$\omega = a_i\omega_1,$$
definimos
$$D_\omega(\alpha) = D_{a_i}(D_{\omega_1}(\alpha)),$$
conforme a la definición recursiva para palabras de longitud $n - 1$.

Observación 3.2.2. De nuevo la intuición puede hacer estragos, nótese que no hay conmutatividad de las derivadas (como sí ocurría en el caso de las derivadas parciales habituales). Es decir, $D_{ab} \neq D_{ba}$. Por poner un ejemplo, consideremos la expresión $\alpha = aa^*bb^*$. Tendremos,

$$D_a(\alpha) = a^*bb^*, D_b(\alpha) = \emptyset.$$

Por tanto,

$$D_{ba}(\alpha) = D_b(D_a(\alpha)) = D_b(a^*bb^*) = b^*,$$

mientras que

$$D_{ab}(\alpha) = D_a(D_b(\alpha)) = D_a(\emptyset) = \emptyset.$$

El resultado crucial es el siguiente:

Proposición 3.2.6. *Sea α una expresión regular sobre un alfabeto finito Σ y sea $Der(\alpha)$ el conjunto de todas las derivadas sucesivas de α con respecto a palabras en Σ^*. Esto es,*

$$Der(\alpha) := \{\beta \ : \ \exists \omega \in \Sigma^*, \ \beta = D_\omega(\alpha)\}.$$

Entonces, $Der(\alpha)$ es un conjunto finito.

Demostración. Se demostraría por inducción en la definición recursiva de la expresión regular. □
Nuestro propósito es construir un grafo con pesos asociado al conjunto de todas las derivadas de la expresión regular. Esto va a constituir la gramática buscada.

Proposición 3.2.7. *El algoritmo siguiente transforma toda expresión regular α en una gramática finita G que genera el lenguaje $L(\alpha)$ descrito por la expresión. En particular, los lenguajes descritos por expresiones regulares son lenguajes regulares.*

Demostración. La idea principal para realizar este algoritmo es la Regla de Leibnitz, combinando las gramáticas con los Lemas 3.2.4 y 3.2.5. □
Consideremos el siguiente algoritmo:

---

begin
INPUT: *Una expresión regular α sobre un alfabeto finito Σ*

- Hallar todos los elementos del conjunto $Der(\alpha) := \{D_\omega(\alpha) \ : \ \omega \in \Sigma^*\}$.
- Definir un conjunto finito V de variables, biyetable al conjunto $Der(\alpha)$. Sea $Q_0 \in V$ un elemento de ese conjunto de variables.
- Definir una biyección $E : Der(\alpha) \longrightarrow V$, tal que $E(\alpha) = Q_0$.
- Definir $P_1 := 1$ y

$$P_2 := \begin{cases} \{Q_0 \mapsto \lambda\}, & \text{si } \lambda \in L(\alpha) \\ \emptyset, & \text{en caso contrario} \end{cases}$$

while $P_2 \neq P_1$ **do**
$P_1 := P_2$
Para cada $\beta \in Der(\alpha)$ **do**
Para cada $a \in \Sigma$ **do**
Hallar $\gamma := D_a(\beta)$, $Q_1 := E(\gamma)$ y $Q_2 := E(\beta)$ en V.
Si $\lambda \in L(\gamma)$, hacer $P_2 := P_2 \cup \{Q_2 \mapsto a\}$.
Si $\gamma \neq \emptyset, \lambda$, hacer $P_2 := P_2 \cup \{Q_2 \mapsto aQ_1\}$.
next a
od
next β
od
od

- OUTPUT: La lista $[V, \Sigma, Q_0, P_2]$.

end

3.3. De RG's a RE's: Uso del Lema de Arden

3.3.1. Ecuaciones Lineales. Lema de Arden

La ecuaciones lineales en los lenguajes regulares juegan un papel muy importante. Estás nos posibilitarán probar que las palabras generadas por una gramática regular forman un lenguaje dado por una expresión regular. Empecemos con la definición.

Definición 3.3.1. Llamaremos sistema de ecuaciones lineales en expresiones regulares a toda ecuación del tipo siguiente:

$$\begin{pmatrix} X_1 \\ \vdots \\ X_n \end{pmatrix} = \begin{pmatrix} \alpha_{1,1} & \cdots & \alpha_{1,n} \\ \vdots & \ddots & \vdots \\ \alpha_{n,1} & \cdots & \alpha_{n,n} \end{pmatrix} \begin{pmatrix} X_1 \\ \vdots \\ X_n \end{pmatrix} + \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_n \end{pmatrix}, \tag{3.3.1}$$

donde los $\alpha_{i,j}$ y los β_k son expresiones regulares sobre un alfabeto Σ.

Una solución de uno de tales sistemas de ecuaciones es una lista $(\omega_1, \ldots, \omega_n)$ de expresiones regulares sobre el mismo alfabeto, tales que

$$\omega_i \equiv \alpha_{i,1} \cdot \omega_1 + \cdots + \alpha_{i,n} \cdot \omega_n + \beta_i,$$

donde $\equiv$ es la igualdad entre los lenguajes que describen (i.e. la igualdad tautológica de las expresiones regulares).
El objetivo de esta Subsección es la discusión del método obvio de resolución de este tipo de ecuaciones lineales. La clave para poder establecer lo obvio es un clásico resultado de Arden:

Definición 3.3.2. Se denomina ecuación lineal fundamental en expresiones regulares a la ecuación lineal en una variable X siguiente:

$$X = \alpha X + \beta,$$

donde α y β son expresiones regulares sobre un alfabeto finito Σ.

Lema 3.3.1 (Lema de Arden)**.** *Dada la ecuación fundamental siguiente:*

$$X = \alpha X + \beta,$$

donde α, β son expresiones regulares sobre un alfabeto Σ. Se verifican las propiedades siguientes:

i) La ecuación fundamental anterior posee una solución única si y solamente si $\lambda \notin L(\alpha)$.

ii) La expresión regular $\alpha^* \cdot \beta$ es siempre solución de la ecuación fundamental anterior.

iii) Si $\lambda \in L(\alpha)$, para cualquier expresión regular γ, la expresión $\alpha^* \cdot (\beta + \gamma)$ es una solución de la ecuación fundamental

Demostración. Aunque no se pretende dar una demostración completa del Lema, al menos señalaremos los hechos fundamentales.
El alumno puede ver fácilmente que cualquier expresión regular que sea solución debe contener al lenguaje $L(\alpha^*\beta)$. Otra observación trivial es que cualquier palabra del lenguaje generado por una solución debe de estar en el lenguaje generado por β o es la concatenación de dos palabras, la primera en el lenguaje generado por α y la segunda en el lenguaje generado por X.
También nótese que si α es una expresión regular, se tiene que

$$L(\alpha \cdot \alpha^*) = L(\alpha)^+.$$

Es decir, $\alpha \cdot \alpha^* \equiv \alpha^+$. Ahora bien, nótese que $L(\alpha^*) = L(\alpha^+)$ si y solamente si $\lambda \in L(\alpha)$.
Del mismo modo, consideremos una expresión regular γ cualquiera, tendremos:

$$\alpha \cdot \alpha^* \cdot (\beta + \gamma) + \beta \equiv \alpha^+ \cdot \beta + \beta + \alpha^+ \cdot \gamma \equiv (\alpha^+ + \lambda) \cdot \beta + \alpha^+ \cdot \gamma \equiv \alpha^* \cdot \beta + \alpha^+ \cdot \gamma.$$

Por su parte, $\alpha^* \cdot (\beta + \gamma) = \alpha^* \cdot \beta + \alpha^* \cdot \gamma$. Esto nos da inmediatamente que si $\alpha^* \equiv \alpha^+$ o si $\gamma = \emptyset$ tenemos la equivalencia. □

Este simple Lema es la base para el algoritmo obvio de sustitución, es decir, eligiendo una variable y sustituyéndola en las demás ecuaciones. Formalmente, esto se expresa de la siguiente manera.

Proposición 3.3.2. *Toda ecuación lineal en expresiones regulares del tipo de la Definición 3.3.1 posee solución, que es una lista de expresiones regulares sobre el mismo alfabeto.*

Demostración. El algoritmo se divide en las dos fases obvias: triangulación/reducción (a través del Lema de Arden) y levantamiento (invirtiendo las expresiones ya despejadas). Los detalles del algoritmo se dejan como ejercicio al alumno.

- **Triangulación:** Seguiremos la notación 3.3.1 y procederemos utilizando inducción. El caso $n = 1$ se resuelve mediante el Lema de Arden. Para el caso $n > 1$, usaremos un doble paso:
  - *Despejar.* Podemos despejar X_n en la última ecuación, mediante la expresión siguiente:

$$X_n := \alpha_{n,n}^* R_n, \tag{3.3.2}$$

    donde $R_n := \sum_{j=1}^{n-1} \alpha_{n,j} X_j + \beta_n$.
  - *Sustituir.* Podemos sustituir la expresión anterior en el resto de las ecuaciones obteniendo un nuevo sistema de $(n-1)$ ecuaciones en $(n-1)$ variables. Este sistema viene dado, obviamente, por las expresiones siguientes para $1 \leq i \leq n-1$:

$$X_i := \left(\sum_{j=1}^{n-1} \left(\alpha_{i,j} + \alpha_{i,n} \alpha_{n,n}^* \alpha_{n,j} \right) X_j \right) + \left(\beta_i + \alpha_{n,n}^* \beta_n \right).$$

- **Levantamiento.** Una vez llegados al caso $n = 1$, se obtiene una expresión regular válida para X_1 y se procede a levantar el resto de las variables usando las expresiones obtenidas en la fase de despejado (expresiones (3.3.2)).

□

3.3.2. Sistema de Ecuaciones Lineales Asociado a una Gramática.

Comenzaremos asociando a cada gramática regular $G = (V, \Sigma, Q_0, P)$ un sistema de ecuaciones lineales en expresiones regulares mediante la regla siguiente:

- Supongamos $V = \{Q_0, \ldots, Q_n\}$ es el conjunto de los símbolos no terminales, que supondremos de cardinal $n+1$. Definamos un conjunto de variables $\{X_0, \ldots, X_n\}$ con el mismo cardinal y con la asignación $q_i \longmapsto X_i$ como biyección.

- Definamos para cada i, $0 \leq i \leq n$, la expresión regular β_i mediante la construcción siguiente. Consideremos todas las producciones que comienzan en la variable q_i y terminan en elementos de $\Sigma \cup \{\lambda\}$. Supongamos que tales producciones sean

$$Q_i \mapsto a_1 \mid \ldots \mid a_r.$$

  Definimos[2]

$$\beta_i := a_1 + \ldots + a_r.$$

  Si no hubiera ninguna producción del tipo $Q_i \mapsto a \in \Sigma \cup \{\lambda\}$, definiremos $\beta_i := \emptyset$.

[2]Note el lector que algún a_i puede ser λ.

- Para cada i y para cada j, definiremos el coeficiente $\alpha_{i,j}$ del modo siguiente. Consideremos todas las producciones que comienzan en el símbolo no terminal Q_i e involucran al símbolo no terminal Q_j. Supongamos que tales producciones sean:

$$Q_i \mapsto a_1 Q_j \mid \cdots \mid a_r Q_j,$$

con $a_k \in \Sigma \cup \{\lambda\}$. Entonces definiremos

$$\alpha_{i,j} := a_1 + \cdots + a_r.$$

Si no hubiera ninguna de tales producciones, definiremos $\alpha_{i,j} := \emptyset$.

Definición 3.3.3 (Sistema asociado a una gramática)**.** Dada una gramática $G = (V, \Sigma, Q_0, P)$ llamaremos sistema asociado a G y lo denotaremos por $\mathcal{S}(G)$ al sistema:

$$\mathcal{S}(G) := \left\{ \begin{pmatrix} X_0 \\ \vdots \\ X_n \end{pmatrix} = \begin{pmatrix} \alpha_{0,1} & \cdots & \alpha_{0,n} \\ \vdots & \ddots & \vdots \\ \alpha_{n,0} & \cdots & \alpha_{n,n} \end{pmatrix} \begin{pmatrix} X_0 \\ \vdots \\ X_n \end{pmatrix} + \begin{pmatrix} \beta_0 \\ \vdots \\ \beta_n \end{pmatrix} \right\},$$

dado por las anteriores reglas de construcción.

Proposición 3.3.3. *Con las anteriores notaciones, sea $(\alpha_0, \ldots, \alpha_n)$ una solución del sistema $S(G)$ asociado a una gramática G. Entonces, $L(\alpha_0)$ es el lenguaje generado por la gramática G.*

Demostración. La idea de la demostración es que estamos asociando una expresión regular a cada variable. La variable X_i es la expresión regular de las palabras que se pueden generar a través de derivaciones empezando por la variable Q_i. Por esa razón la solución de nuestro problema es encontrar X_0. A partir de esta idea, la demostración se realiza por inducción. □

Teorema 3.3.4. *Los lenguajes regulares son los descritos por las expresiones regulares. Es decir, todo lenguaje descrito por una expresión regular es el lenguaje generado por alguna gramática regular y, recíprocamente, todo lenguaje generado por alguna gramática regular puede ser descrito por alguna expresión regular. Además, existen algoritmos que transforman RE's en RG's y recíprocamente.*

Demostración. Basta con combinar los algoritmos descritos en las Proposiciones 3.3.3 y 3.2.7. □

3.4. Problemas y Cuestiones.

3.4.1. Cuestiones Relativas a Lenguajes y Gramáticas.

Cuestión 3.4.1. Se considera una gramática sobre el alfabeto $\Sigma := \{a, b\}$, cuyas producciones vienen dadas por

$$Q_0 \longrightarrow \lambda \mid aQ_0a \mid bQ_0b.$$

Decidir si el lenguaje generado por esa gramática es el conjunto de los palíndromos sobre Σ.

Cuestión 3.4.2. Demostrar la falsedad de las afirmaciones dando el código java sobre las siguientes afirmaciones (se deja a un lado la funcionalidad del programa, por ahora sólo se requiere si el compilador devolverá un error al tratar de compilarlo):

- Cambiar una orden por otra correcta no provoca errores de compilación.
- Trabajando con paréntesis () y corchetes [], no hay que tener más cuidado que cuando abramos alguno, se cierre en la misma orden.

Cuestión 3.4.3. Si el sistema de producciones de una gramática no posee ninguna transformación del tipo $A \longrightarrow a$, ¿ podemos asegurar que no es una gramática regular?.

Cuestión 3.4.4. El lenguaje sobre el alfabeto $\{0,\ 1\}$ de las palabras que no contienen a 00 como subpalabra, ¿es un lenguaje regular?.

Cuestión 3.4.5. Dados dos lenguajes L_1 y L_2 sobre el alfabeto $\{a, b\}$, ¿ podemos asegurar que se verifica la siguiente igualdad

$$(L_1 \cdot L_2)^R = L_1^R \cdot L_2^R?$$

Cuestión 3.4.6. Dar una definición inductiva (recursiva) de la transformación $w \longmapsto w^R$ que revierte las palabras.

3.4.2. Cuestiones Relativas a Expresiones Regulares.

Cuestión 3.4.7. Se dice que una expresión regular α está en forma normal disyuntiva si

$$\alpha = \alpha_1 + \cdots + \alpha_n,$$

donde las expresiones regulares $\alpha_1, \ldots,\ \alpha_n$ no involucran el operador $+$. Decidir si la siguiente expresión regular esta en forma disyuntiva ó encontrar una forma de ponerla en forma disyuntiva:

$$(0 + 00 + 10)^*,$$

con $\Sigma = \{0,\ 1\}$.

Cuestión 3.4.8. Decidir si es verdadera la siguiente igualdad de expresiones regulares:

$$(a + b)^* = (a^* + b^*)^*.$$

Cuestión 3.4.9. ¿Pertenece la palabra *acdcdb* al lenguaje descrito por la expresión regular siguiente:

$$\alpha = \left(b^* a^* \left(cd\right)^* b\right)^* + \left(cd\right)^*?.$$

Cuestión 3.4.10. Sea L el lenguaje sobre el alfabeto $\{a, b\}$ formado por todas las palabras que contienen al menos una aparición de la palabra b. ¿Es L el lenguaje descrito por la expresión regular siguiente

$$\alpha := a^* \left(ba^*\right)^* bb^* \left(b^* a^*\right)^*?.$$

Cuestión 3.4.11. Dada cualquier expresión regular α, ¿Se cumple $\alpha^* \alpha = \alpha^*$?.

Cuestión 3.4.12. Dadas tres expresiones regulares α, β, γ, ¿Es cierto que $\alpha + (\beta \cdot \gamma) = (\alpha + \beta) \cdot (\alpha + \gamma)$?.

Cuestión 3.4.13. ¿Es siempre la derivada de una expresión regular otra expresión regular?.

3.4.3. Problemas Relativos a Lenguajes Formales y Gramáticas

Problema 3.4.1. Sea $L := \{\lambda, a\}$ un lenguaje sobre el alfabeto $\Sigma := \{a, b\}$. Hallar L^n para los valores $n = 0, 1, 2, 3, 4$. ¿Cuántos elementos tiene L^n?.

Problema 3.4.2. Dados los lenguajes $L_1 := \{a\}$ y $L_2 := \{b\}$ sobre el mismo alfabeto anterior, describir $(L_1 \cdot L_2)^*$ y $(L_1 \cdot L_2)^+$. Buscar coincidencias.

Problema 3.4.3. Probar que la concatenación de los lenguajes no es distributiva con respecto a la intersección de lenguajes.

Problema 3.4.4. Probar que la longitud $|\cdot| : \Sigma^* \longrightarrow \mathbb{N}$ es un morfismo de monoides suprayectivo, pero no es un isomorfismo excepto si $\sharp\,(\Sigma) = 1$.

Problema 3.4.5. Dado el alfabeto $\Sigma = \{0, 1\}$, se consideran los siguientes dos lenguajes:

$$L_1 := \{\omega \in \Sigma^* \ : \ \sharp \text{ (ceros en } \omega) \ \in 2\mathbb{Z}\}.$$
$$L_2 := \{\omega \in \Sigma^* \ : \ \exists\, n \in \mathbb{N},\ \omega = 01^n\}.$$

Demostrar que $L_1 \cdot L_2$ es el lenguaje L_3 siguiente:

$$L_3 := \{\omega \in \Sigma^* \ : \ \sharp \text{ (ceros en } \omega) \ \in 2\mathbb{Z} + 1\}.$$

Problema 3.4.6. Sea $G = (\{Q_0\}, \{a, b\}, Q_0, P)$ una gramática libre de contexto dada por las producciones:

$$Q_0 \longrightarrow aQ_0b \ \mid \ \lambda.$$

Probar que $L(G)$ es el lenguaje definido por

$$L := \{a^n b^n \ : \ n \in \mathbb{N}\}.$$

Problema 3.4.7. Sea $L := \{a^n b^n c^n \ : \ n \in \mathbb{N}\}$ un lenguaje sobre el alfabeto $\Sigma = \{a, b, c\}$. Hallar una gramática G tal que $L(G) = L$. Clasificar G dentro de la jerarquía de Chomsky.

Problema 3.4.8. Hallar una gramática libre de contexto (no regular) y otra equivalente regular para cada uno de los dos lenguajes siguientes:

$$L_1 := \{ab^n a \ : \ n \in \mathbb{N}\},$$

$$L_2 := \{0^n 1 \ : \ n \in \mathbb{N}\}.$$

Problema 3.4.9. Hallar gramáticas que generen los siguientes lenguajes:

$$L_1 := \{0^m 1^n \ : \ [m, n \in \mathbb{N}] \wedge [m \geq n]\},$$

$$L_2 := \{0^k 1^m 2^n \ : \ [n, k, m \in \mathbb{N}] \wedge [n = k + m]\}.$$

Problema 3.4.10. Dado el lenguaje $L := \{z \in \{a, b\}^* \ : \ \exists w \in \{a, b\}^*, \text{ con } z = ww\}$, hallar una gramática que lo genere.

Problema 3.4.11. Clasificar las siguientes gramáticas en términos de la jerarquía de Chomsky. Tratar de analizar los lenguajes generados por ellas y definirlos por compresión.

i) $P := \{Q_0 \to \lambda \ \mid \ A, A \to c \ \mid \ AA\}$, $V := \{Q_0, A\}$, $\Sigma := \{c\}$.

ii) $P := \{Q_0 \to \lambda \ \mid \ A, A \to Ad \ \mid \ cA \ \mid \ c \ \mid \ d\}$, $V := \{Q_0, A\}$, $\Sigma := \{c, d\}$.

iii) $P := \{Q_0 \to c \ \mid \ Q_0cQ_0\}$, $V := \{Q_0\}$, $\Sigma := \{c\}$.

iv) $P := \{Q_0 \to c \ \mid \ AcA, A \to cc \ \mid cAc, cA \to cQ_0\}$, $V := \{Q_0, A\}$, $\Sigma := \{c\}$.

v) $P := \{Q_0 \to AcA, A \to 0, Ac \to AAcA \ \mid \ ABc \ \mid \ AcB, B \to B \ \mid \ AB\}$, $V := \{Q_0, A, B\}$, $\Sigma := \{0, c\}$.

Problema 3.4.12. Sea G la gramática dada por las siguientes producciones:

$$Q_0 \to 0B \ \mid \ 1A,$$

$$A \to 0 \ \mid \ 0Q_0 \ \mid \ 1AA,$$

$$B \to 1 \ \mid \ 1Q_0 \ \mid \ 0BB.$$

Siendo $V := \{Q_0, A, B\}$ y $\Sigma := \{0, 1\}$, probar que

$$L(G) := \{\omega \in \{0, 1\}^* \ : \ \sharp\,(\text{ceros en } \omega) = \sharp\,(\text{unos en } \omega) \ \wedge \ |\omega| \geq 0\}.$$

Problema 3.4.13. Probar que si L es el lenguaje dado por la siguiente igualdad:

$$L := \{\omega \in \{0,1\}^* \ : \ \sharp\,(\text{ceros en } \omega) \neq \sharp\,(\text{unos en } \omega)\},$$

entonces $L^* = \{0,1\}^*$.

Problema 3.4.14. Sea $L \subseteq \{a,b\}^*$ el lenguaje dado por la siguiente definición:

- $\lambda \in L$,
- Si $\omega \in L$, entonces $a\omega b \in L$ y $b\omega a \in L$,
- Si $x, y \in L$, entonces $xy \in L$.

Describir el lenguaje y definirlo por comprensión.

Problema 3.4.15. Probar que si L es generado por una gramática regular a izquierda, entonces L^R es generado por una gramática regular a derecha.

3.4.4. Problemas Relativos a Expresiones Regulares

Problema 3.4.16. Dadas α, β dos expresiones regulares. Probar que si $L(\alpha) \subseteq L(\beta)$, entonces $\alpha + \beta \equiv \beta$.

Problema 3.4.17. Dada la expresión regular $\alpha = a + bc + b^3a$, ¿Cuál es el lenguaje regular $L(\alpha)$ descrito por ella?. ¿Cuál es la expresión regular que define el lenguaje $\{a,b,c\}^*$?

Problema 3.4.18. Simplificar la expresión regular $\alpha = a + a(b + aa)(b^*(aa)^*)^*b^* + a(aa + b)^*$, usando las equivalencias (semánticas) vistas en clase.

Problema 3.4.19. Calcular la derivada $D_{ab}(\alpha) = D_a(D_b(\alpha))$, siendo $\alpha := a^*ab$.

Problema 3.4.20. Comprobar que $x := \alpha^*\beta$ es una solución para la ecuación fundamental $\alpha \equiv \alpha x + \beta$.

Problema 3.4.21. Simplificar la expresión regular $\alpha := 1^*01^*\left(01^*01^*0 + 1\right)^* 01^* + 1^*$.

Problema 3.4.22. Hallar la expresión regular α asociada a la siguiente gramática (por el método de las ecuaciones lineales):

$$Q_0 \to aA \ \mid \ cA \ \mid \ a \ \mid \ c,$$

$$A \to bQ_0.$$

Aplicar el método de las derivadas a α y comparar los resultados.

Problema 3.4.23. Hallar la gramática que genera el lenguaje descrito por la siguiente expresión regular:

$$\alpha := (b + ab^+a)^*ab^*.$$

Problema 3.4.24. Comprobar la equivalencia tautológica

$$(b + ab^*a)^*ab^* \equiv b^*a(b + ab^*a)^*.$$

Problema 3.4.25. Dada la expresión regular $\alpha := (ab + aba)^*$, hallar una gramática que genere el lenguaje $L(\alpha)$.

Problema 3.4.26. Dada la expresión regular $\alpha := a(bc)^*(b + bc) + a$, hallar una gramática G que genere el lenguaje $L(\alpha)$. Construir el sistema $S(G)$ asociado a la gramática calculada, resolverlo y comparar los resultados.

Problema 3.4.27. Hallar la expresión regular α asociada a la siguiente gramática:

$$Q_0 \to bA \mid \lambda,$$

$$A \to bB \mid \lambda,$$

$$B \to aA.$$

Aplicar el método de las derivadas a α y comparar los resultados.

Problema 3.4.28. Idem con la gramática:

$$Q_0 \to 0A \mid 1B \mid \lambda,$$

$$A \to 1A \mid 0B,$$

$$B \to 1A \mid 0B \mid \lambda.$$

Problema 3.4.29. Probar que si α es una expresión regular tal que $\alpha^2 \equiv \alpha$, entonces $\alpha^* = \alpha + \lambda$.

Problema 3.4.30. Probar que si α es una expresión regular se cumple $\alpha^* \equiv \alpha^* \alpha + \lambda$.

Problema 3.4.31. Hallar dos expresiones regulares distintas que sean solución de la siguiente ecuación lineal

$$(a + \lambda)X = X.$$

Problema 3.4.32. Las *Expresiones Regulares Avanzadas* son expresiones regulares añadiendo diferentes operadores. Se utilizan en lenguajes de programación como Perl para búsquedas dentro de texto. Los operadores añadidos son los siguientes:

- operador de rango: Para las letras [a..z], significa que cualquier letra del rango es correcta.
- operador ?: Este operador aplicado a una expresión regular entre paréntesis indica que necesariamente encaja en este esquema.
- operador /i: Este operador, indica el símbolo en la posición i de la palabra, por ejemplo /1 indica el primer símbolo de la palabra.

Mostrar como transformar las expresiones regulares avanzadas a expresiones regulares. Aplicarlo al siguiente caso,

$$[a..c]([C..E]^*)?/1.$$

Capítulo 4

Autómatas Finitos

Contents

4.1. Introducción: Correctores Léxicos o Morfológicos 61
4.2. La Noción de Autómata 62
4.2.1. Sistema de Transición de un autómata: 63
4.2.1.1. Representación Gráfica de la Función de Transición. 64
4.2.1.2. Lenguaje Aceptado por un Autómata 65
4.3. Determinismo e Indeterminismo 66
4.3.1. El Autómata como Programa 66
4.3.2. Autómatas con/sin $\lambda-$Transiciones. 66
4.3.2.1. Grafo de $\lambda-$transiciones. 67
4.3.3. Determinismo e Indeterminismo en Autómatas 68
4.4. Lenguajes Regulares y Autómatas. 68
4.4.1. Teorema de Análisis de Kleene 68
4.4.2. Teorema de Síntesis de Kleene 69
4.5. Lenguajes que no son regulares 71
4.5.1. El Palíndromo no es un Lenguaje Regular. 74
4.6. Minimización de Autómatas Deterministas 75
4.6.1. Eliminación de Estados Inaccesibles. 75
4.6.2. Autómata Cociente 75
4.6.3. Algoritmo para el Cálculo de Autómatas Minimales. 76
4.7. Disgresión: Los autómatas finitos como cadenas de Markov 78
4.7.1. Markov Chains 78
4.7.2. Adjacency Matrix 78
4.7.3. Finite Automata as Markov Chains 78
4.7.4. Probabilistic Finite Automata 78
4.8. Cuestiones y Problemas. 78
4.8.1. Cuestiones. 78
4.8.2. Problemas 80

4.1. Introducción: Correctores Léxicos o Morfológicos

La siguiente etapa, que constituye un buen entrenamiento para las máquinas de Turing, son los autómatas finitos. Los autómatas finitos corresponden a correctores ortográficos. Se trata de la vieja tarea del maestro de primaria, corrigiendo los dictados, esto es, evaluando la presencia de errores ortográficos. El maestro no se ocupa de la corrección sintáctica del dictado (es él quien ha dictado las palabras y su secuencia, incluyendo signos ortográficos) sino solamente de los errores

de transcripción y, por tanto, errores en la escritura morfológica o léxica. El otro ejemplo son los populares `Spell Checkers`, sobre todo si no tienen en cuenta elementos sintácticos del texto que corrigen (concordancias de género, número, subordinadas...).
En términos informáticos, los autómatas finitos se usan para corregir (o señalar) los lugares en los que la morfología de un lenguaje de programación no ha sido respetada. Si, por ejemplo, alguien elabora un pequeño programa en C, Maple, Matlab o, simplemente, un documento Textures, como parte del proceso de compilación existe un autómata finito que detecta la presencia de errores y, si encuentra alguno, salta mostrando dónde aparece.
El gran impulsor de la Teoría de Autómatas fue J. von Neumann. Este matemático, gastó buena parte de los últimos años de su vida en el desarrollo de la teoría de autómatas y, durante la Segunda Guerra Mundial, en el desarrollo de los computadores electrónicos de gran tamaño que fructificó en la aparición del ENIAC (un ordenador para calcular rápidamente trayectorias balísticas que fue financiado por el ejército de los Estados Unidos y finalizado en 1948 [1]).

4.2. La Noción de Autómata

Formalmente, se definen como sigue:

Definición 4.2.1. Llamaremos autómata finito indeterminístico a todo quíntuplo $A := (Q, \Sigma, q_0, F, \delta)$ donde:

- Σ es un conjunto finito (alfabeto),
- Q es un conjunto finito cuyos elementos se llaman estados y que suele denominarse espacio de estados,
- q_0 es un elemento de Q que se denomina estado inicial,
- F es un subconjunto de Q, cuyos elementos se denominan estados finales aceptadores,
- $\delta : Q \times (\Sigma \cup \{\lambda\}) \longrightarrow Q$ es una correspondencia que se denomina función de transición.

Si δ es aplicación, el autómata se denomina determinístico.

Observación 4.2.1. En el caso indeterminístico, uno podría considerar la transición δ no como una correspondencia

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \longrightarrow Q$$

sino como una aplicación

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \longrightarrow \mathcal{P}(Q),$$

donde $\mathcal{P}(Q)$ es el conjunto de todos los subconjuntos del espacio de estados. Así, por ejemplo, si (q, a) no está en correspondencia (vía δ) con ningún estado, podríamos haber escrito $\delta(q, a) = \emptyset$. Sin embargo, mantendremos la notación (incorrecta, pero cómoda) $\delta(q, a) = p$ para indicar que el estado $p \in Q$ están en correspondencia con el par (q, a) a través de la correspondencia δ. Así, por ejemplo, escribiremos $\neg\exists p \in Q,\ \delta(q, a) = p$ (o, simplemente, $\neg\exists\delta(q, a)$) para denotar que no hay ningún estado de Q en correspondencia con (q, a) a través de δ. Del mismo modo, $\exists p \in Q,\ \delta(q, a) = p$ (o, simplemente, $\exists\delta(q, a)$) en el caso contrario.

Para ver la acción dinámica asociada a un autómata, definamos su sistema de transición.

[1] La historia del diseño y puesta en marcha del ENIAC y las personas involucradas puede seguirse en la página web *http://ftp.arl.mil/ ~mike/comphist/eniac-story.html* .

4.2.1. Sistema de Transición de un autómata:

Sea dado el autómata $A := (Q, \Sigma, q_0, F, \delta)$

- $S := Q \times \Sigma^*$ es el espacio de configuraciones,
- La transición $\rightarrow_A \subseteq S \times S$ se define por las reglas siguientes:

$$(q, x) \rightarrow_A (q', x') \Leftrightarrow \exists \alpha \in \Sigma \cup \{\lambda\},\ \ x = \alpha x',\ \ q' = \delta(q, x)$$

Para interpretar mejor el proceso, hagamos nuestra primera descripción gráfica. Las palabras del alfabeto Σ^* se pueden imaginar como escritas en una cinta infinita, dividida en celdas en cada una de las cuales puedo escribir un símbolo de Σ.

$$\overline{\underline{\mid x_1 \mid x_2 \mid x_3 \mid \cdots}}$$

Hay una unidad de control que ocupa diferentes posiciones sobre la cinta y que dispone de una cantidad finita de memoria en la que puede recoger un estado de Q:

$$\begin{array}{c} \overline{\underline{\mid x_1 \mid x_2 \mid x_3 \mid \cdots}} \\ \uparrow \\ \overline{\underline{\mid q \mid}} \end{array}$$

Las configuraciones de S sólo representan el momento instantáneo (*snapshot*) de cálculo correspondiente. Así, dada una palabra $x = x_1 \cdots x_n \in \Sigma^*$ el autómata A computa sobre esta palabra de la manera siguiente:

- Inicializa $(q_0, x) \in S$, es decir

$$\begin{array}{c} \overline{\underline{\mid x_1 \mid x_2 \mid x_3 \mid \cdots}} \\ \uparrow \\ \overline{\underline{\mid q_0 \mid}} \end{array}$$

- $q_1 := \delta(q_0, x_1)$, $x^{(1)} := x_2 \cdots x_n \in \Sigma^*$,

$$(q_0, x) \rightarrow_A (q_1, x^{(1)})$$

  Gráficamente, borramos el contenido de la primera celda, cambiamos el estado en la unidad de control de q_0 (estado inicial) a q_1 y movemos la unidad de control un paso a la derecha:

$$\begin{array}{c} \overline{\underline{\mid \quad \mid x_2 \mid x_3 \mid \cdots}} \\ \uparrow \\ \overline{\underline{\mid q_1 \mid}} \end{array}$$

- El proceso continúa hasta que nos quedamos sin palabra, i.e. llegamos a la configuración $(q_{n-1}, x^{(n)}) \in S$, donde $x^{(n)} := x_n$ es una palabra de longitud 1. Sea $q_n := \delta(q_{n-1}, x_n)$ y λ la palabra vacía y tenemos la sucesión de computación:

$$(q_0, x) \rightarrow_A (q_1, x^{(1)}) \rightarrow_A \cdots \rightarrow_A (q_{n-1}, x^{(n)}) \rightarrow_A (q_n, \lambda)$$

$$\begin{array}{c} \overline{\underline{\mid \quad \mid \quad \mid \quad \mid \cdots \mid \quad \mid \cdots\cdots}} \\ \uparrow \\ \overline{\underline{\mid q_n \mid}} \end{array}$$

Ejemplo 4.2.1. Consideremos el siguiente autómata $A = (Q, \Sigma, q_0, F, \delta)$. Donde,

- $\Sigma = \{a, b\}$.
- $Q := \{q_0, q_1, q_2, q_3\}$.
- $F := \{q_2\}$.

Para la función de transición δ elegiremos una representación a través de una tabla:

δ	**a**	**b**
q_0	q_1	q_3
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_3	q_3

Esta tabla debe interpretarse como $\delta(q_i, x)$ es el estado que aparece en la fila q_i y columna x.
Revisemos la computación del autómata A sobre un par de entradas:
Sea $x = aabbb \in \Sigma^*$ y veamos cómo funciona nuestro autómata:

$$(q_0, aabbb) \to_A (q_1, abbb) \to_A (q_1, bbb) \to_A$$

$$\to_A (q_2, bb) \to_A (q_2, b) \to_A (q_2, \lambda)$$

Y la palabra *aabbb* es aceptada.
Tomemos la palabra $y = baba \in \Sigma^*$ y tratemos de seguir los cálculos de nuestro autómata:

$$(Q_0, baba) \to_A (Q_3, aba) \to_A (Q_3, ba) \to_A (Q_3, a) \to_A (Q_3, \lambda)$$

y la palabra *baba* no es aceptada por nuestro autómata.

El autómata ya presenta una primera aproximación a las máquinas de Turing. Procesa listas y va corrigiendo la palabra dada. Si, al final del proceso, el autómata alcanza una configuración final aceptadora, es porque la palabra dada en la configuración inicial era correcta. En caso contrario rechaza. Así surge la primera noción de problema susceptible de ser tratado computacionalmente.

Definición 4.2.2. Dado un autómata A en las notaciones anteriores y una palabra $\omega \in \Sigma^*$, definimos la correspondencia

$$\delta^* : Q \times \Sigma^* \longrightarrow \mathcal{P}(Q)$$

dada por:

- $\delta^*(q, a) = \delta(q, a)$ si $a \in \Sigma \cup \{\lambda\}$.
- $\delta^*(q, aw) = \{\delta^*(q', \omega) \quad \forall q' \in \delta(q, a)\ \}$.

4.2.1.1. Representación Gráfica de la Función de Transición.

Una forma estética, pero no siempre conveniente a la hora de manipular autómatas relativamente grandes, es la representación de sistemas de transición mediante grafos con aristas etiquetadas (pesos), un ejemplo se puede ver en la Figura 4.1. Las reglas son las siguientes:

- Los nodos del grafo están dados por los estados del grafo. Cada nodo está rodeado de, al menos, una circunferencia.
- Los nodos finales aceptadores del grafo son aquellos que están rodeados por dos circunferencias, el resto de los nodos aparecen rodeados de una sola circunferencia.

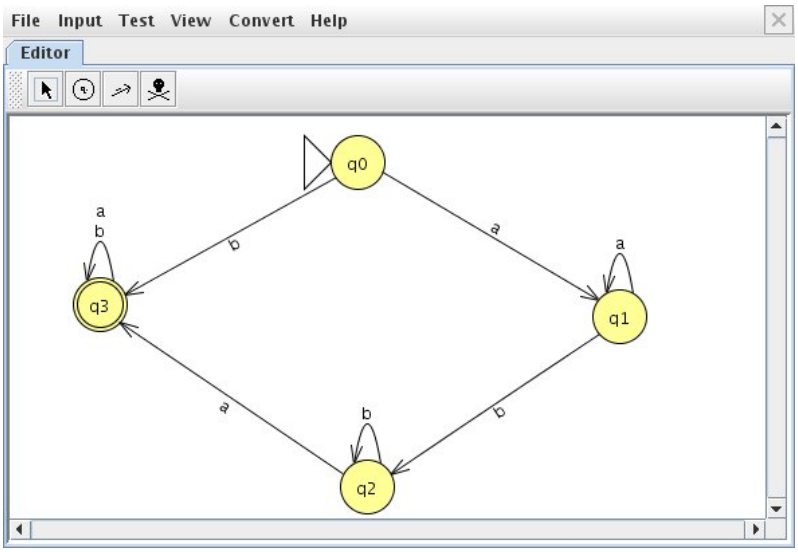


Figura 4.1: Representación gráfica del autómata.

- Dada una transición $\delta(q, z) = p$, asignaremos la arista del grafo (q, p) con etiqueta z.
- Hay una arista sin entrada, cuya salida es el nodo asociado al estado inicial.

Usaremos más habitualmente la representación de las funciones de transición bien mediante listas o bien mediante tablas.

4.2.1.2. Lenguaje Aceptado por un Autómata

Definición 4.2.3. Llamaremos lenguaje aceptado por un autómata A al conjunto de palabras $\omega \in \Sigma^*$ tales que $\delta(q_0, \omega) \in F$, es decir al conjunto de palabras tales que se alcanza alguna configuración final aceptadora.

En términos de la Definición 4.2.2, podremos también escribir:

$$L(A) := \{x \in \Sigma^* \ : \ \delta^*(q, x) \cap F \neq \emptyset\}.$$

Podemos interpretar un autómata como un evaluador de la función característica de un subconjunto de $L \subseteq \Sigma^*$:

$$\chi_L : \Sigma^* \longrightarrow \{0, 1\}$$

Los autómatas deterministas directamente sirven para evaluar χ_L y la interpretación es la obvia en términos de pregunta respuesta:

INPUT: Una palabra $\omega \in \Sigma^*$
OUTPUT:

- 1 si el autómata llega a una configuración final aceptadora (i.e., $\delta(q_0, \omega) \in F$).
- 0 si el autómata llega a una configuración final no aceptadora (i.e., $\delta(q_0, \omega) \in Q \setminus F$).

Una buena referencia sobre autómatas es el texto [DaWe, 94], donde también se pueden encontrar ejemplos sencillos que ayuden al alumno a asimilar la noción.

4.3. Determinismo e Indeterminismo

4.3.1. El Autómata como Programa

Una manera bastante natural de interpretar el autómata finito es usar un pseudo–código para expresar un autómata como un programa con un **while** basado en el sistema de transición anterior. Informalmente, sea $A := (Q, \Sigma, q_0, F, \delta)$ un autómata. El programa (algoritmo) que define es el dado por la siguiente descripción:

INPUT: $x \in \Sigma^*$ (una palabra sobre el alfabeto).

Initialize: $I := (q_0, x)$ (la configuración inicial sobre x)

while $I \notin F \times \{\lambda\}$ **do**
  if $I = (q, x_1x')$, $x_1 \in \Sigma \cup \{\lambda\}$, $x_1x' \neq \lambda$, **then** $I := (\delta(q, x_1), x')$
    else OUPUT **NO**
  fi
od
OUTPUT **YES**

Nótese que hemos introducido deliberadamente un pseudo–código que no necesariamente termina en todos los inputs. Esto es por analogía con las máquinas de Turing y el estudio de los lenguajes recursivamente enumerables y recursivos. Aquí, el pseudo–código tiene una interpretación directa y natural en el caso determinístico y genera una forma imprecisa en el caso indeterminístico. Esta interpretación como programa (determinístico) de este pseudo–código depende esencialmente de la ausencia de dos obstrucciones:

- La presencia de $\lambda-$transiciones, esto es, de transiciones de la forma $\delta(q, \lambda)$ que pueden hacer que caigamos en un ciclo infinito.
- La indefinición de $I = (\delta(q, x_1), x')$ por no estar definido $\delta(q, x_1)$ o por tener más de un valor asociado.

Ambas obstrucciones se resuelven con los algoritmos que se describen a continuación.

4.3.2. Autómatas con/sin $\lambda-$Transiciones.

Se denominan $\lambda-$transiciones a las transiciones de una autómata $A := (Q, \Sigma, q_0, F, \delta)$ de la forma:

$$\delta(q, \lambda) = p,$$

done hemos mantenido la notación válida para el caso determinístico e indeterminístico, a pesar de la notación incorrecta del segundo caso. Un autómata se dice *libre de $\lambda-$transiciones* si no hay ninguna de tales transiciones.
En un sentido menos preciso, las $\lambda-$transiciones son meras transformaciones de los estados conforme a reglas que no dependen del contenido de la cinta.
En términos del sistema de transición, para cada una configuración (q, x) en el sistema de transición asociado al autómata y supuesto que existe una $\lambda-$transición $\delta(q, \lambda) = p$, entonces la transición será de la forma $(q, x) \to (p, x)$, donde x no es modificado y sólo hemos modificado el estado.
En términos de operaciones de lecto–escritura, nuestra $\lambda-$transición realiza las siguientes tareas:

- *NO lee* el contenido de la cinta.
- *Modifica* el estado en la unidad de control.
- *NO borra* el contenido de la celda señalada por la unidad de control.
- *NO se mueve* a la izquierda.

4.3.2.1. Grafo de $\lambda-$transiciones.

A partir de las $\lambda-$transiciones de un autómata podemos construir un grafo. Dado un autómata $A := (Q, \Sigma, q_0, F, \delta)$, definimos el grafo de las $\lambda-$transiciones de A mediante $G := (V, E)$, donde las reglas son:

- $V = Q$.
- Dados $p, q \in V$, decimos que $(p, q) \in E$ si $q \in \delta(p, \lambda)$, i.e.

$$E := \{(p, q) \ : \ q \in \delta(p, \lambda)\}.$$

Si miramos el grafo asociado al autómata (cf. 4.2.1.1), podemos extraer el grafo de $\lambda-$transiciones, dejando los mismos nodos (o vértices) y suprimiendo todas las aristas que estén etiquetadas con algún símbolo del alfabeto (y dejando solamente las que están etiquetadas con λ).
A partir del grafo de las $\lambda-$transiciones podemos considerar la *clausura transitiva* de un nodo (estado), definiéndola del modo siguiente:

$$\lambda - cl(p) := \{q \in V \ : \ (p, \lambda) \vdash (q, \lambda)\}.$$

Obsérvese que la $\lambda-$clausura de un nodo p está determinada por las configuraciones (con palabra vacía λ) alcanzables desde la configuración (p, λ) dentro del sistema de transición asociado al autómata.
Obsérvese también que la palabra vacía λ está en el lenguaje aceptado $L(A)$ si y solamente si la clausura $\lambda - cl(q_0)$ del estado inicial contiene algún estado final aceptado (i.e. $\lambda - cl(q_0) \cap F \neq \emptyset$).
Del mismo modo, dados $p \in Q$ y $a \in \Sigma$, definiremos la $\lambda-$clausura de p y a mediante:

$$\lambda - cl(p, a) := \{q \in V \ : \ (p, \lambda) \vdash (q, \lambda), \ \exists \delta(q, a)\}.$$

Nuestro objetivo es probar el siguiente enunciado:

Proposición 4.3.1. *Dado cualquier lenguaje L que sea aceptado por un autómata con $\lambda-$transiciones, entonces existe un autómata libre de $\lambda-$transiciones que acepta el mismo lenguaje. Más aún, la transformación de un autómata a otra se puede realizar algorítmicamente.*

Demostración. Como en el resto de los casos, nos basta con tomar como dado de entrada un autómata $A := (Q, \Sigma, q_0, F, \delta)$ y definir un nuevo autómata que elimina las $\lambda-$transiciones. El nuevo autómata no ha de ser determinista, pero éso es irrelevante como veremos en la Proposición 4.3.2.
Construiremos un nuevo autómata $\bar{A} := (\bar{Q}, \Sigma, \overline{q_0}, \bar{F}, \bar{\delta})$ definido conforme al algoritmo siguiente:

INPUT: Autómata $A := (Q, \Sigma, q_0, F, \delta)$.

Initialize: $\bar{Q} := Q$ y $\overline{q_0} := q_0$.

for each $p \in Q$ **do find** $\lambda - cl(p)$ **od**

$\bar{F} := F \cup \{p \ : \ \lambda - cl(p) \cap F \neq \emptyset\}$.

for each $p \in Q$ **do**

if $\lambda - cl(p, a) \neq \emptyset$, **then** $\bar{\delta}(p, a) := \bigcup_{q \in \lambda - cl(p)} \lambda - cl(\delta(q, a))$.

fi

od

OUTPUT $\bar{A} := (\bar{Q}, \Sigma, \overline{q_0}, \bar{F}, \bar{\delta})$

Nótese que $\bar{\delta}(p, \lambda)$ no está definida para ningún $p \in Q$. Dejamos como ejercicio la comprobación de que el autómata $\bar{A}$ acepta L. □

Observación 4.3.1. Obsérvese que el resultado de eliminar $\lambda-$transiciones puede ser un autómata indeterminista.

Observación 4.3.2. Nótese que en el caso en que $\lambda \in L(A)$ (i.e. $\lambda - cl(q_0) \cap F \neq \emptyset$), el estado inicial pasa a ser también estado final aceptador.

4.3.3. Determinismo e Indeterminismo en Autómatas

Una primera preocupación técnica podría ser el papel que juega el indeterminismo en la clase de lenguajes aceptados por autómatas. Los siguientes resultados tranquilizan mostrando que el indeterminismo es irrelevante en cuanto a la clase de lenguajes aceptados.

Proposición 4.3.2. *Si un lenguaje $L \subseteq \Sigma^*$ es aceptado por un autómata finito indeterminista, entonces, existe un autómata finito determinista que lo acepta.* [2]

Demostración. La idea es simple, sea $A = (Q, \Sigma, q_0, F, \delta)$ un autómata indeterminista sin $\lambda-$transiciones que acepta un lenguaje $L \subseteq \Sigma^*$. Definamos el siguiente autómata determinista $\overline{A}$ dado por:

- $\overline{Q} := \mathcal{P}(Q)$ (el espacio de estados es el conjunto de las partes de Q).
- $\overline{F} := \{X \in \overline{Q} \ : \ X \cap F \neq \emptyset\}$ (las configuraciones finales aceptadoras son aquellas que contienen algún estado del espacio F de estados finales aceptadores).
- $\overline{q_0} := \{q_0\}$ (el conjunto formado por la antigua configuración inicial).
- La función de transición
$$\overline{\delta} : \overline{Q} \times \Sigma \longrightarrow \overline{Q}$$
definida mediante:
$$\overline{\delta}(X, a) := \{q \in Q \ : \ \exists q' \in X, \ q = \delta(q', a)\}.$$

Dejamos el asunto de la comprobación como ejercicio. □

Observación 4.3.3. A partir de ahora usaremos autómatas deterministas e indeterministas sin la preocupación sobre el indeterminismo, dado que podemos reemplazar unos por otros sin mayores problemas.

4.4. Lenguajes Regulares y Autómatas.

Como indica el título, el objetivo de esta sección es mostrar que los lenguajes aceptados por los autómatas son los lenguajes regulares. Para ello, mostraremos dos procedimientos de paso conocidos como Teorema de Análisis y Teorema de Síntesis de Kleene (cf. [Kl, 56]).

4.4.1. Teorema de Análisis de Kleene

Nuestra primera duda que cualquier lenguaje aceptado por un autómata finito esta generado por una expresión regular. El teorema siguiente afirma eso y además da un algoritmo para calcularlo. Se deja al alumno el ejercicio de demostrar la complejidad del algoritmo.

Teorema 4.4.1. *Sea $L \subseteq \Sigma^*$ un lenguaje aceptado por un autómata finito determinista. Entonces, existe una expresión regular α sobre el alfabeto Σ tal que $L = L(\alpha)$. Más aún, mostraremos que existe un procedimiento tratable que permite calcular la expresión regular asociada al lenguaje aceptado por un autómata.*

Demostración. Nos limitaremos con mostrar el procedimiento, que casi viene prefigurado por las definiciones.
Para ello construiremos un sistema de ecuaciones lineales en expresiones regulares con las reglas siguientes:

- Supongamos que $Q := \{q_0, \ldots, q_n\}$. Introducimos un conjunto de variables biyectable con Q dado por $\{X_0, \ldots, X_n\}$. La biyección será dada por $q_i \longmapsto X_i$.

[2] Una característica del indeterminismo es que no modifica la clase de lenguajes aceptados; aunque sí podría modificar los tiempos de cálculo. Esto no afecta a los autómatas finitos, según se prueba en este enunciado, pero sí está detrás de la Conjetura de Cokk $\mathbf{P} = \mathbf{NP}$?.

- Definimos un sistema de ecuaciones lineales en expresiones regulares:

$$\begin{pmatrix} X_0 \\ \vdots \\ X_n \end{pmatrix} = \begin{pmatrix} \alpha_{0,0} & \cdots & \alpha_{0,n} \\ \vdots & \ddots & \vdots \\ \alpha_{n,0} & \cdots & \alpha_{n,n} \end{pmatrix} \begin{pmatrix} X_0 \\ \vdots \\ X_n \end{pmatrix} + \begin{pmatrix} \beta_0 \\ \vdots \\ \beta_n \end{pmatrix},$$

  Conforme a las reglas siguientes:

  - Para cada i, $0 \leq i \leq n$, definamos $\beta_i = \lambda$ si $q_i \in F$ y $\beta_i = \emptyset$ si $q_i \notin F$.
  - Para cada i, j, $0 \leq i, j \leq n$, definamos $A_{i,j}$ mediante:

$$A_{i,j} := \{z \in \Sigma \ : \ \delta(q_i, z) = q_j\}.$$

    Definiremos

$$\alpha_{i,j} := \sum_{z \in A_{i,j}} z,$$

    notando que si $A_{i,j} = \emptyset$, entonces, $\alpha_{i,j} = \emptyset$.

Entonces, si $(\alpha_0, \ldots, \alpha_n)$ es una solución del anterior sistema lineal, $L(\alpha_0)$ es el lenguaje aceptado por el autómata. La idea de la demostración es la siguiente: Empecemos por calcular el lenguaje de las palabras que empezando en q_0 son aceptadas por el autómata y llamemos a este lenguaje X_0. De la misma forma, para cada uno de los estados ponemos un lenguaje X_1, $X_2 \ldots$ Hay una clara relación entre estos lenguajes, que esta dada por las ecuaciones lineales dadas más arriba. El lenguaje X_0 está claramente formado por la unión de los lenguajes X_i correspondientes, con prefijo dado por el símbolo de la transición. Además, si el estado es final hay que añadir la palabra λ. □

Definición 4.4.1 (Sistema Característico de un Autómata)**.** Se denomina sistema de ecuaciones característico de un autómata al sistema de ecuaciones lineales en expresiones regulares obtenido conforme a las reglas descritas en la demostración del Teorema anterior.

Observación 4.4.1. Nótese que, a partir del Sistema característico de un autómata A uno podría reconstruir una gramática regular G que genera el mismo lenguaje $L(G)$ que el aceptado por A, i.e. $L(G) = L(A)$.

4.4.2. Teorema de Síntesis de Kleene

En esta segunda parte, vamos a mostrar el recíproco. Esto es, que para cualquier lenguaje descrito por una expresión regular se puede encontrar un autómata determinista que lo acepta. Para ello haremos como en el caso del paso de expresiones regulares a gramáticas: usaremos el árbol de formación de la expresión regular. Comenzaremos por un sencillo Lema.

Lema 4.4.2. *Dado un lenguaje L aceptado por un autómata, existe un autómata $A := (Q, \Sigma, q_0, F, \delta)$ que acepta L y que verifica las siguientes propiedades:*

i) $\sharp(F) = 1$*, es decir, sólo hay una configuración final aceptadora. Supondremos $F := \{f\}$.*

ii) $\delta(q, x)$ *está definida para todo $q \in Q$ y todo $x \in \Sigma$.*

iii) Las únicas $\lambda-$transiciones entran en f. Es decir,

$$\text{Si } \ \delta(p, \lambda) = q \Leftrightarrow q = f.$$

Demostración. Dado el autómata $A := (Q, \Sigma, q_0, F, \delta)$, que podemos suponer determinista, definamos el nuevo autómata $\bar{A} := (\bar{Q}, \Sigma, q_0, \bar{F}, \bar{\delta})$ conforme a las reglas siguientes:

- Sea $f, ERROR$ dos nuevos estados tal que $f, ERROR \notin Q$. Definamos $\bar{Q} := Q \cup \{f\} \cup \{ERROR\}$.

- Definamos $\bar{F} := \{f\}$.
- Para cada $p \in Q$ y para cada $a \in \Sigma$, definamos para los nuevos estados

$$\bar{\delta}(ERROR, a) := ERROR, \qquad \bar{\delta}(f, a) = ERROR.$$

  y extendamos la función de transición para los antiguos estados si $a \in \Sigma$

$$\bar{\delta}(p, a) := \begin{cases} \delta(p, a), \text{ si } \delta(p, a) \text{ está definida,} \\ ERROR, \text{ en otro caso.} \end{cases}$$

- Para cada $p \in F$, definamos $\bar{\delta}(p, \lambda) := f$.

Es claro que $\bar{A}$ acepta el mismo lenguaje que aceptaba A. La razón es simple: la única manera de alcanzar el nuevo estado f es llegar a un estado final con la cinta vacía. □

Teorema 4.4.3. *Sea Σ un alfabeto finito y α una expresión regular sobre Σ. Entonces, existe un autómata finito A que reconoce el lenguaje $L(\alpha)$ descrito por α. Más aún, el proceso de obtención del autómata a partir de la expresión regular se puede lograr de manera algorítmica.*

Demostración. De nuevo nos limitaremos a describir un proceso algorítmico que transforma expresiones regulares en autómatas, usando los operadores de definición de la expresión (i.e., el procedimiento es recursivo en la construcción de la expresión regular).

- **El caso de los símbolos primarios:**
  - *El caso $\emptyset$:* Bastará un autómata con $Q := \{q_0, f\}$, $F := \{f\}$ tal que la función de transición no esté definida en ningún caso.
  - *El caso λ:* De nuevo usaremos $Q := \{q_0, f\}$, $F := \{f\}$, pero la función de transición está definida solamente para $\delta(q_0, \lambda) = f$ y no definida en el resto de los casos.
  - *El caso constante $a \in \Sigma$:* Igual que en el caso anterior, usaremos $Q := \{q_0, f\}$, $F := \{f\}$, pero la función de transición está definida solamente para $\delta(q_0, a) = f$ y no definida en el resto de los casos.
- **Siguiendo los operadores:**
  - *El autómata de la unión ($\alpha + \beta$):* Si tenemos $A_1 := (Q_1, \Sigma, q_1, F_1, \delta_1)$ un autómata determinista que acepta $L(\alpha) \subseteq \Sigma^*$ y un segundo autómata también deterministas $A_2 := (Q_2, \Sigma, q_2, F_2, \delta_2)$ un autómata que acepta $L(\beta) \subseteq \Sigma^*$, definimos un nuevo autómata[3] $A := (Q, \Sigma, q_0, F, \delta)$ que acepta $L_1 \cup L_2$ y viene dado por las reglas siguientes:
    - $Q := Q_1 \times Q_2$,
    - $F := (F_1 \times Q_2) \cup (Q_1 \times F_2)$
    - $Q_0 := (q_1, q_2)$
    - $\delta((p, q), z) = (\delta_1(p, z), \delta_2(q, z))$, $\forall p \in Q_1, q \in Q_2$ y $\forall z \in \Sigma \cup \{\lambda\}$.
  - *El autómata de la concatenación ($\alpha \cdot \beta$):* Supongamos $A_1 := (Q_1, \Sigma, q_1, F_1, \delta_1)$ un autómata que acepta $L(\alpha) \subseteq \Sigma^*$ y un segundo autómata $A_2 := (Q_2, \Sigma, q_2, F_2, \delta_2)$ un autómata que acepta $L(\beta) \subseteq \Sigma^*$. Supongamos que A_1 verifica las condiciones descritas en el Lema 4.4.2 y sea $F_1 := \{f\}$.

    Definimos un nuevo autómata $A := (Q, \Sigma, q_0, F, \delta)$ que acepta $L(\alpha\beta)$ y viene dado por las reglas siguientes:
    - $Q := (Q_1 \times \{1\}) \cup (Q_2 \times \{2\})$.
    - $F := F_2 \times \{2\}$.

[3] Esta construcción se la conoce como *Autómata Producto*.

- $q_0 := (q_1, 1)$
- La función de transición $\delta : Q \times (\Sigma \cup \{\lambda\}) \longrightarrow Q$, viene dada por:

$$\delta((q,i),z) := \begin{cases} (\delta_1(q,z),1)\,, & \text{si } q \in Q_1,\ i=1 \\ (Q_2,2)\,, & \text{si } q = f \in F_1,\ i=1,\ z=\lambda \\ (\delta_2(q,z),2)\,, & \text{si } q \in Q_2,\ i=2 \end{cases} \tag{4.4.1}$$

- *El autómata del monoide generado (α^*):* De nuevo suponemos que tenemos un autómata $A := (Q, \Sigma, q_0, F, \delta)$ que acepta el lenguaje $L(\alpha)$. Podemos suponer que dicho autómata verifica las condiciones del Lema 4.4.2 anterior. Supongamos $F = \{f\}$. Definamos un nuevo autómata $A^* := (Q, \Sigma, q_0, F, \bar{\delta})$ conforme a las reglas siguientes:
  - Para cada $q \in Q \setminus F$ y para cada $z \in \Sigma \cup \{\lambda\}$, definamos $\bar{\delta}(q,z) := \delta(q,z)$.
  - Finalmente, definamos:

$$\bar{\delta}(f,\lambda) := q_0.$$

y

$$\bar{\delta}(q_0,\lambda) := f.$$

  Es claro que este autómata acepta el lenguaje previsto.

Con esto acabamos la demostración, ya que cualquier expresión regular esta formada por concatenación, suma de expresiones regulares o es estrella de una expresión regular. □

4.5. Lenguajes que no son regulares

La tradición usa el Lema de Bombeo para mostrar las limitaciones de los lenguajes regulares. El resultado es debido a Y. Bar-Hillel, M. Perles, E. Shamir[4]. Este Lema se enuncia del modo siguiente:

Teorema 4.5.1 (Pumping Lemma)**.** *Sea L un lenguaje regular. Entonces, existe un número entero positivo $p \in \mathbb{N}$ ($p \geq 1$) tal que para cada palabra $\omega \in L$, con $|\omega| \geq p$ existen $x, y, z \in \Sigma^*$ verificando los siguientes propiedades:*

- $|y| \geq 1$ *(i.e. $y \neq \lambda$),*
- $|xy| \leq p$,
- $\omega = xyz$,
- *Para todo $\ell \in \mathbb{N}$, las palabras $xy^\ell z \in L$*

El Lema de Bombeo simplemente dice que hay prefijos y una lista finita de palabras tal que, bombeando esas palabras, permaneceremos en el mismo lenguaje regular.

Observación 4.5.1. Hay varias razones por las que éste es un enunciado insuficiente. La primera es estética: un exceso de fórmulas cuantificadas hace desagradable su lectura.
Adicionalmente, debemos señalar que el Lema de Bombeo da una condición necesaria de los lenguajes regulares, pero no es una condición suficiente. Es decir, hay ejemplos de lenguajes que no son regulares (ver Corolario 4.5.3) pero que sí satisfacen el Lema de Bombeo (ver ejemplo 4.5.1 más abajo)

Ejemplo 4.5.1. Los lenguajes regulares satisfacen el Lema de Bomeo y también lo satisface el siguiente lenguaje:

$$L := \{a^i b^j c^k \ : \ [i=0] \vee [j=k]\} \subseteq \{a,b,c\}^*.$$

Veamos que satisface el Teorema 4.5.1 anterior con $p = 1$. Para ello, sea $\omega \in \Sigma^*$ y tendremos tres casos:

[4] Y. Bar-Hillel, M. Perles, E. Shamir. "On formal properties of simple phrase structure grammars". *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung* **14** (1961) 143–172.

- *Caso 1:* $i = 0$ con $j = 0$ o, lo que es lo mismo, $\omega = c^k$, con $k \geq 1$. En ese caso, tomando $x = \lambda$, $y = c$, $z = c^{k-1}$, tenemos que $xy^\ell z \in L$, $\forall \ell \in \mathbb{N}$.
- *Caso 2:* $i = 0$ con $j \geq 1$ o, lo que es lo mismo, $\omega = b^j c^k$, $j \geq 1$. En ese caso, tomando $x = \lambda$, $y = b$, $z = b^{j-1}c^k$, tenemos que $xy^\ell z \in L$, $\forall \ell \in \mathbb{N}$.
- *Caso 3:* $i \geq 1$ o, lo que es lo mismo, $\omega = a^i b^j c^j$. En ese caso, tomando $x = \lambda$, $y = a$, $z = a^{i-1}b^j c^j$, tenemos que $xy^\ell z \in L$, $\forall \ell \in \mathbb{N}$.

Veremos más adelante (Corolario 4.5.3) que este lenguaje no es regular.

Definición 4.5.1 (Prefijos)**.** Sea Σ un alfabeto finito y sea $L \subseteq \Sigma^*$ un lenguaje cualquiera. Definimos la siguiente relación de equivalencia sobre Σ^*:
dados $x, y \in \Sigma^*$, $x \sim_L y$ si y solamente si:

$$\forall w \in \Sigma^*, xw \in L \Leftrightarrow yw \in L.$$

Verificar que estamos ante una relación de equivalencia es un mero ejercicio. Lo que pretendemos es caracterizar los lenguajes aceptados por un autómata mediante una caracterización del conjunto cociente: $\Sigma^*/\sim_L$.

Teorema 4.5.2 (Myhill–Nerode)**.** [5] *Si $L \subseteq \Sigma^*$ es un lenguaje, entonces L es regular si y solamente si $\Sigma^*/\sim_L$ es finito.*

Demostración. Comencemos con una de las implicaciones. Supongamos que L es el lenguaje aceptado por un autómata determinista $A := (\Sigma, Q, q_0, F, \delta)$. Consideremos el conjunto de los estados alcanzables por alguna computación de A:

$$\overline{Q} := \{q \in Q \ : \ \exists y \in \Sigma^*, \ (q_0, y) \vdash (q, \lambda)\}$$

es claro que $\overline{Q} \subseteq Q$ es un conjunto finito. Para cada $q \in \overline{Q}$, sea $y_q \in \Sigma^*$ un elemento cualquiera tal que $(q_0, y_q) \vdash (q, \lambda)$. Sea

$$S := \{y_q \ : \ q \in \overline{Q}\}$$

Claramente S es un conjunto finito y vamos a probar que

$$\Sigma^*/\sim_L = \{[y_q] \ : \ y_q \in S\},$$

donde $[y_q]$ es la clase de equivalencia definida por y_q y tendremos la afirmación. Ahora, tomemos $x \in \Sigma^*$ y sea $(q_0, x) \vdash (q, \lambda)$, $q \in \overline{Q}$. Para cualquier $w \in \Sigma^*$, el sistema de transición asociado al autómata A, trabajando sobre xw realiza algo como lo siguiente:

$$(q_0, xw) \to_A \cdots \to_A (q, w)$$

mientras vamos borrando la x. Ahora bien, si tomamos $y_q w \in \Sigma^*$, el cálculo hará también el camino:

$$(q_0, y_q w) \to_A \cdots \to_A (q, w)$$

Lo que pase a partir de (q, w) es independiente de por dónde hayamos empezado, luego xw es aceptado por A si y solamente si $y_q w$ es aceptado por A. Con esto hemos demostrado una de las direcciones del enunciado, esto es, si el lenguaje es regular y, por ende, aceptado por un autómata finito, entonces, el conjunto cociente $\Sigma^*/\sim_L$ es finito.
Para el recíproco, supongamos que $\Sigma^*/\sim_L$ es finito y supongamos:

$$\Sigma^*/\sim_L = \{[y_1], \ldots, [y_m]\},$$

donde $y_i \in \Sigma^*$. Podemos suponer que $y_1 = \lambda$ (la palabra vacía estará en alguna clase de equivalencia). Además, observemos que la clase de equivalencia $[y_1] = [\lambda]$ está formada por los elementos de L. Ahora definamos un autómata $A = (Q, \Sigma, q_0, F, \delta)$ con las reglas siguientes:

[5] Rabin, M. and Scott, D.."Finite automata and their decision problems". *IBM Journal of Research & Development* **3** (1959), 114-125.

- Los estados están definidos mediante:

$$Q := \{[y_1], \ldots, [y_m]\}.$$

- El estado inicial es dado por $q_0 = [y_1] = [\lambda]$.
- El espacio de estados finales aceptadores es $F := \{[\lambda]\}$.
- La función de transición es dada para cada $x \in \Sigma \cup \{\lambda\}$, mediante:

$$\delta([y], x) = [yx].$$

Veamos que esta autómata realiza la tarea indicada. La configuraciíon inicial es $([\lambda], \omega)$ para cualquier palabra $\omega \in \Sigma^*$. Conforme va avanzando a partir de esta configuración, el autómata alcanza $([\omega], \lambda)$ y, por tanto, acepta si y solamente si $[\omega] = [\lambda]$, lo cual es equivalente a $\omega \in L$. □

Corollario 4.5.3. *El lenguaje L descrito en el Ejemplo 4.5.1 no es un lenguaje regular.*

Demostración. Basta con verificar que no satisface las propiedades descritas en el Teorema de Myhill–Nerode. Para ello supongamos que el conjunto cociente $\Sigma^*/\sim_L$ es finito, es decir,

$$\Sigma^*/\sim_L := \{[y_1], \ldots, [y_r]\}.$$

Consideremos la sucesión infinita $x_n := ab^n$. Como sólo hay un número finito de clases de equivalencia, hay una clase de equivalencia (digamos $[y_1]$) que contiene una infinidad de términos de esa sucesión. En otras palabras, existe una sucesión infinita y creciente:

$$1 < n_1 < n_1 < \cdots < n_k < n_{k+1} < \cdots, \tag{4.5.1}$$

de tal modo que

$$\{x_{n_i} \ : \ i \in \mathbb{N}, i \geq 1\} \subseteq [y_1].$$

En este caso, se ha de tener, además, la siguiente propiedad:

$$\forall \omega \in \Sigma^*, \ y_1\omega \notin L. \tag{4.5.2}$$

Para probarlo, nótese que si existiera naluna palabra $\omega \in \Sigma^*$ tal que $y_1\omega \in L$, entonces podemos suponer que esa palabra es de longitud finita. Supongamos $p := |\omega| \in \mathbb{N}$ esa longitud. Como la sucesión de los n_i's es inifniota y creciente, entonces, ha de existir algún n_t tal que $n_t > p + 3$. Pero, además, $ab^{n_t} \sim_L y_1$, luego, como $y_1\omega \in L$, entonces también se ha de tener

$$ab^{n_t}\omega \in L.$$

Por la definición de L tendremos, entonces que

$$ab^{n_t}\omega = ab^j c^j,$$

para algún j. Obviamente ésto significa que ω es de la forma $\omega = b^r c^j$ y, necesariemente, $j = n_t + r \geq n_t$. Por lo tanto, $p + 3 = |\omega| + 3 \geq j + 3 = n_t + r + 3 > n_t$, contraviniendo nuestra elección de $n_t \geq p + 3$.
Con esto hemos probado la veracidad de la afirmación (4.5.2) anterior. Pero, de otro lado, $ab_{n_1}c_{n_1} \in L$ y $ab^{n_1} \sim_L y_1$ luego $y_1c^{n_1} \in L$, lo que contradice justamente la afirmación probada. La hipótesis que no se sostiene es que el conjunto cociente $\Sigma^*/\sim_L$ sea finito y, por tanto, L no es un lenguaje regular. □

4.5.1. El Palíndromo no es un Lenguaje Regular.

Se trata del ejemplo clásico y común que "deben" contemplar todos los cursos de Introducción a los lenguajes regulares: *el Palíndromo* o, en buen catalán, el problema de la detección de los *"cap–i–cua"*, del que veremos que no es un lenguaje regular, como consecuencia del resultado de Myhill y Nerode anterior.
Comencemos recordando la definición del Palíndromo ya presentado en Secciones anteriores. Dado un alfabeto finito Σ, y una palabra $\omega = x_1 \cdots x_n \in \Sigma^*$, denominamos el reverso de ω, ω^R a la palabra: $\omega^R = x_n \cdots x_1$.
El lenguaje del Palíndromo es dado por las palabras que coinciden con su reverso, esto es,

$$\mathcal{P} := \{x \in \Sigma^* \ : \ x^R = x\}.$$

Daremos una demostración del resultado siguiente usando la finitud de lso prefijos.

Corollario 4.5.4. *El Palíndromo no es un lenguaje regular si el alfabeto tiene al menos dos dígitos distintos.*

Demostración. Por simplicidad supongamos $\Sigma = \{0, 1\}$. Para cada número natural $n \in \mathbb{N}$, consideremos la palabra de longitud $n + 2$ siguiente:

$$x_n := 0^n 10.$$

Supongamos que el palíndromo es un lenguaje regular y será finito el conjunto cociente siguiente:

$$\Sigma^* / \sim_{\mathcal{P}} = \{[y_1], \ldots, [y_m]\}.$$

De otro lado, consideremos las clases definidas por los elementos de la sucesión anterior:

$$S := \{[x_n] \ : \ n \in \mathbb{N}\}.$$

Como el conjunto cociente es finito, el anterior conjunto S es finito y, por tanto, habrá alguna clase $[y]$ en la que estará una infinitud de elementos de la sucesión $\{x_n \ : \ n \in \mathbb{N}\}$. Es decir, que existe una sucesión infinita creciente de índices:

$$n_1 < n_2 < n_3 < \cdots < n_k < \cdots$$

de tal modo que $x_{n_j} \in [y]$. Supongamos n_j suficientemente grande (por ejemplo, $n_j \geq 2|y| + 3$). Ahora obsérvese que $x_{n_j} x_{n_j}^R \in \mathcal{P}$ es un palíndromo. Como $x_{n_j} \sim_{\mathcal{P}} y$ (están en la misma clase de equivalencia), tendremos que $y x_{n_j}^R \in \mathcal{P}$. Por tanto,

$$y x_{n_j}^R = x_{n_j} y^R. \tag{4.5.3}$$

Como la longitud de x_{n_j} es mayor que la de y, tendremos que y debe coincidir con los primeros $\ell = |y|$ dígitos de x_{n_j}. Por tanto, $y = 0^\ell$.
Ahora bien, el único dígito 1 de la palabra $y x_{n_j}^R$ en la identidad (4.5.3) ocupa el lugar $\ell + 2$, mientras que el único dígito 1 de la palabra $x_{n_j} y^R$ ocupa el lugar $n_j + 1$, como $n_j \geq 2\ell + 3$ no es posible que ambas palabra sean iguales, contradiciendo la igualdad (4.5.3) y llegando a contradicción. Por tanto, el palídromo no puede ser un lenguaje regular. □

Ejemplo 4.5.2. Los siguientes son también ejemplos de lenguajes no regulares:

- $\Sigma = \{0, 1\}$ y el lenguaje L dado por la condición el número de 1's es mayor que el número de 0's.
- Para el mismo alfabeto el lenguaje:

$$L := \{0^m 1^m \ : \ m \in \mathbb{N}\}$$

- Para el alfabeto $\Sigma = \{0, 1, \ldots, 9\}$ sea $\pi \subseteq \Sigma^*$ el lenguaje formado por las palabras que son prefijos de la expansión decimal de $\pi \in \mathbb{R}$, es decir:

$$L := \{3, 31, 314, 3141, 31415, \ldots\}$$

4.6. Minimización de Autómatas Deterministas

En ocasiones uno puede observar que el autómata que ha diseñado (usando algunas de las propiedades o métodos ya descritos) es un autómata con demasiados estados (y, por tanto, el código del programa es excesivo para el programador). Para resolver esta situación se utiliza un proceso de minimización de los autómatas que pasaremos a describir a continuación.
Comenzaremos observando que las computaciones que realizan varios estados pueden ser esencialmente las mismas, que los efectos que producen ciertos estados podrían ser los mismos. Esto se caracteriza mediante la relación de equivalencia siguiente:

4.6.1. Eliminación de Estados Inaccesibles.

En ocasiones se presentan autómatas en los que se han incluido estados innaccesibles, es decir, estados a los que no se puede llegar de ningún modo desde el estado inicial. Para describir esta noción, definiremos la siguiente estructura de grafo asociada a un autómata.
Sea $A := (Q, \Sigma, q_0, F, \delta)$ un autómata determinista. Consideremos el grafo de estados siguiente: $\mathcal{G}_A := (V, E)$, donde

- El conjunto de vértices o nodos V es el conjunto de estados Q (i.e. $V = Q$).
- Las aristas del grafo (que será orientado) son los pares (p, q) tales que existe $x \in \Sigma$ verificando $\delta(p, x) = q$.

Nótese que el grafo coincide con el grafo subyacente a la descripción del autómata como grafo con pesos.

Definición 4.6.1. Dado un autómata $A := (Q, \Sigma, q_0, F, \delta)$, un estado $q \in Q$ se denomina accesible si está en la clausura transitiva (componente conexa) del estado inicial q_0. Se llaman estados inaccesibles aquellos estados que no son accesibles.

Proposición 4.6.1. *Dado un autómata $A := (Q, \Sigma, q_0, F, \delta)$, existe un autómata A' que acepta el mismo lenguaje y que no contiene estados inaccesibles.*

Demostración. Para definir A' basta con eliminar los estados inaccesibles del autómata A, es decir, definimos $A' := (Q', \Sigma, q_0{}', F', \delta')$ mediante

- $Q' := \{q \in Q \ : \ q \text{ es accesible desde } q_0 \text{ en } \mathcal{G}_A\}$.
- $q'_0 = q_0$.
- $F' := F \cap Q$.
- La función de transición δ' es la restricción a Q' de δ:

$$\delta' := \delta \mid_{Q' \times \Sigma} .$$

□

4.6.2. Autómata Cociente

Sea $A := (Q, \Sigma, q_0, F, \delta)$ un autómata determinista. Supongamos que todos los estados son accesibles. Dos estados $p, q \in Q$ se dicen equivalentes si se verifica la siguiente propiedad:

$$\forall z \in \Sigma^*, \ \text{ Si } (((p, z) \vdash (p', \lambda)) \wedge ((q, z) \vdash (q', \lambda))) \Longrightarrow ((p' \in F) \Leftrightarrow (q' \in F)) .$$

En otras palabras, dos estados son equivalentes si para cualquier palabra el efecto de la computación que generan es el mismo (en términos de alcanzar o no un estado final aceptador).
Denotaremos por $p \sim_A q$ en el caso de que p y q sean equivalentes. Para cada estado $q \in Q$, denotaremos por $[q]_A$ la clase de equivalencia definida por q y denotaremos por $Q/\sim_A$ al conjunto cociente. Definiremos autómata minimal al autómata que tiene el menor número de estados y que acepta un lenguaje.

Teorema 4.6.2 (Autómata Cociente)**.** *Sea L un lenguaje aceptado por un autómata determinista A sin estados inaccesibles. Entonces, existe un autómata minimal que lo acepta. Dicho autómata $(\tilde{Q}, \Sigma, \widetilde{Q_0}, \tilde{F}, \tilde{\delta})$ viene dado en los términos siguientes:*

- $\tilde{Q} := Q/\sim_A$,
- $\tilde{F} := \{[q]_A \ : \ q \in F\}$.
- $\widetilde{q_0} := [q_0]_A$.
- $\tilde{\delta}([q]_A, z) := [\delta(q, a)]$.

Demostración. Lo dejamos para la reflexión de los alumnos. □

4.6.3. Algoritmo para el Cálculo de Autómatas Minimales.

De la definición del autómata cociente, concluimos la dificultad (aparente) del cálculo de las clases de equivalencia no puede hacerse de manera simple (porque habríamos de verificar todas las palabras $z \in \Sigma^*$). Por eso se plantean algoritmos alternativos como el que se describe a continuación (tomado de [Eil, 74]).
Para construir nuestro autómata cociente, tomaremos una cadena de relaciones de equivalencia. Las definiremos recursivamente del modo siguiente:
Sea $A := (Q, \Sigma, q_0, F, \delta)$ un autómata. Definamos las siguientes relaciones:

- *La relación E_0:* Dados $p, q \in Q$, diremos que pE_0q (p y q están relacionados al nivel 0) si se verifica:
$$p \in F \Leftrightarrow q \in F.$$
Es claramente una relación de equivalencia. El conjunto cociente está formado por dos clases:
$$Q/E_0 := \{F, Q \setminus F\}.$$
Definamos $e_0 := \sharp\,(Q/E_0) = 2$.

- *La relación E_1:* Dados $p, q \in Q$, diremos que pE_1q (p y q están relacionados al nivel 1) si se verifica:
$$pE_1q \Leftrightarrow \begin{cases} pE_0q, \\ \wedge \\ \delta(p,z)E_0\delta(q,z), \quad \forall z \in \Sigma \cup \{\lambda\} \end{cases}$$
Es, de nuevo, una relación de equivalencia. El conjunto cociente ya no es tan obvio, y definimos:
$$e_1 := \sharp\,(Q/E_1)\,.$$

- *La relación E_n:* Para $n \geq 2$, definimos la relación del modo siguiente: Dados $p, q \in Q$, diremos que pE_nq (p y q están relacionados al nivel n) si se verifica:
$$pE_nq \Leftrightarrow \begin{cases} pE_{n-1}q, \\ \wedge \\ \delta(p,z)E_{n-1}\delta(q,z), \quad \forall z \in \Sigma \cup \{\lambda\} \end{cases}$$
Es, de nuevo, una relación de equivalencia. El conjunto cociente ya no es tan obvio, y definimos:
$$e_n := \sharp\,(Q/E_n)\,.$$

Lema 4.6.3. *Sea $A := (Q, \Sigma, q_0, F, \delta)$ un autómata y sean $\{E_n \ : \ n \in \mathbb{N}\}$ la cadena de relaciones de equivalencia definidas conforme a la regla anterior. Se tiene:*

i) Para cada $n \in \mathbb{N}$, $e_n \leq e_{n+1}$.

ii) Si existe $n \in \mathbb{N}$, tal que $e_n = e_{n+1}$, entonces

$$e_m = e_n,\ \forall m \geq n.$$

Demostración. Es claro que si dos estados están relacionados a nivel n entonces, están relacionados a nivel $n-1$. Esto es así por pura construcción (por definición). Por tanto, la relación E_{n+1} lo más que puede hacer es partir en más de una clase de equivalencia alguna de las clases de equivalencia del conjunto cociente anterior. Por tanto,

$$e_n = \sharp(Q/E_n) \leq \sharp(Q/E_{n+1}) = e_{n+1}.$$

Como, además, la relación E_{n+1} se define inductivamente a partir de la relación E_n, si $e_n = e_{n+1}$, entonces, las clases a nivel n siguen siendo las clases a nivel $n+1$. En otras palabras, si $e_n = e_{n+1}$, entonces para todo par $p, q \in Q$, pE_nq si y solamente si $pE_{n+1}q$. En particular, $E_n = E_{n+1}$ y ambas relaciones de equivalencia son la misma. Inductivamente, para $n+2$ se tendrá

$$pE_{n+2}q \Leftrightarrow \left\{ \begin{array}{c} pE_{n+1}q, \\ \\ \delta(p,z)E_{n+1}\delta(q,z), \quad \forall z \in \Sigma \cup \{\lambda\} \end{array} \right\} \Leftrightarrow$$

$$\Leftrightarrow \left\{ \begin{array}{c} pE_nq, \\ \\ \delta(p,z)E_n\delta(q,z), \quad \forall z \in \Sigma \cup \{\lambda\} \end{array} \right\} \Leftrightarrow pE_{n+1}q \Leftrightarrow pE_nq.$$

Por tanto $E_{n+2} = E_{n+1} = E_n$ y, en consecuencia, $e_{n+2} = e_{n+1} = e_n$.
Para cualquier $m \geq n+3$, aplique inducción para concluir $E_m = E_{n+1} = E_n$ y, además, $e_m = e_n$. □

Proposición 4.6.4. *Con las notaciones del Lema anterior, para cada autómata A existe $n \in \mathbb{N}$, con $n \leq \sharp(Q) - 2$, tal que para todo $m \geq n$ se verifica:*

i) $pE_mq \Leftrightarrow pE_nq$, $\forall p, q \in Q$.

ii) $e_m = e_n$.

Demostración. Por el Lema anterior, concluimos:

$$2 = e_0 \leq e_1 \leq e_2 \leq \cdots \leq e_n \leq \cdots$$

Ahora consideremos $n = \sharp(Q) - 2$. Pueden ocurrir dos cosas:

- *Caso I:* Que $e_i \neq e_{i+1}$ para todo $i \leq n$. Es decir, que tengamos (con $n = \sharp(Q) - 2$):

  $$2 = e_0 < e_1 < e_2 < \cdots < e_n.$$

  En este caso, tendríamos

  $$e_1 \geq e_0 + 1 = 3,$$
  $$e_2 \geq e_1 + 1 \geq 4,$$
  $$\vdots$$
  $$e_{n-1} \geq e_{n-2} + 1 \geq \cdots \geq e_1 + (n-2) \geq 3 + (n-1) = n + 2 = \sharp(Q).$$
  $$e_n \geq e_{n-1} + 1 \geq n + 3 = \sharp(Q).$$

  Recordemos que $e_n = \sharp(Q/E_n)$ y el número de clases de equivalencia no puede ser mayor que el número de elementos de Q, es decir, habremos logrado que $n+3 \leq e_n \leq \sharp(Q) = n+2$ y eso es imposible. Así que este caso no se puede dar.

- *Caso II:* La negación del caso anterior. Es decir, existe un i, con $0 \leq i \leq n$ (y $n = \sharp(Q) - 2$) tal que $e_i = e_i + 1$. Entonces, por el Lema anterior, se tendrá:

$$2 = e_0 < e_1 < e_2 < \cdots < e_i = e_{i+1} = \cdots = e_m = \cdots$$

A partir de que sólo se puede dar el caso II, es obvio que se tienen las propiedades del enunciado. □

Teorema 4.6.5. *Sea $A := (Q, \Sigma, q_0, F, \delta)$ un autómata sin $\lambda-$transiciones y sean p, q dos estados. Entonces, tomando $n = \sharp(Q) - 2$, se tendrá que*

$$p \sim_A q \Leftrightarrow pE_n q.$$

Demostración. Lo dejamos como ejercicio para el alumno. □
En particular, el algoritmo que calcula el autómata minimal funciona como sigue:

- *Hallar el conjunto cociente (Q/E_0) y su cardinal e_0.*
- *(Siguiendo los E_i's) Mientras el conjunto cociente "nuevo" sea alterado con respecto al anterior, hallar el conjunto cociente siguiente.*
- *Parar cuando el cardinal del "nuevo" conjunto cociente coincida con el último calculado.*

4.7. Disgresión: Los autómatas finitos como cadenas de Markov

4.7.1. Markov Chains

4.7.2. Adjacency Matrix

4.7.3. Finite Automata as Markov Chains

4.7.4. Probabilistic Finite Automata

4.8. Cuestiones y Problemas.

4.8.1. Cuestiones.

Cuestión 4.8.1. Sea $A := (Q, \Sigma, q_0, F, \delta)$ un autómata indeterminista que verifica la siguiente propiedad: Para todo estado q y para todo símbolo $z \in \Sigma \cup \{\lambda\}$,

$$\sharp\left(\{p \ : \ \delta(q, z) = p\}\right) \leq 1,$$

donde $\sharp$ significa cardinal. Dar un procedimiento inmediato para hallar uno equivalente que sea determinista.

Cuestión 4.8.2. Hallar una expresión regular α sobre el alfabeto $\{a, b\}$ que describa el lenguaje aceptado por el autómata siguiente. Sea $Q := \{q_0, q_1\}$ y $F = \{q_1\}$. Siendo la función de transición dada por la tabla:

δ	**a**	**b**	λ
q_0	q_0	q_1	**N.D.**
q_1	**N.D.**	**N.D.**	**N.D.**

Donde **N.D.** significa "No Definido".

Cuestión 4.8.3. Considerar el autómata $A := (Q, \Sigma, q_0, F, \delta)$, donde

- $\Sigma := \{a\}$,
- $Q := \{q_0, q_1, q_2\}$,
- $F := \{q_2\}$.

Y la función de transición es dada por la Tabla siguiente:

δ	**a**	λ
q_0	q_1	**N.D.**
q_1	q_2	**N.D.**
q_2	q_0	**N.D.**

Probar que $L(A) = \{(aaa)^n aa \ : \ n \in \mathbb{N}\}$.

Cuestión 4.8.4. Describir un autómata que acepta el siguiente lenguaje:

$$L(A) := \{\omega \in \{a,b\}^* \ : \ \sharp(\text{apariciones de } b \text{ en } \omega) \in 2\mathbb{N}\}.$$

Cuestión 4.8.5. Considérese el autómata siguiente:

- $\Sigma := \{0, 1\}$,
- $Q := \{q_0, q_1, q_2, q_3, q_4, q_5\}$,
- $F := \{q_2, q_3, q_4\}$.

Cuya función de transición es dada por la tabla siguiente:

δ	**0**	**1**	λ
q_0	q_1	q_2	**N.D.**
q_1	q_0	q_3	**N.D.**
q_2	q_4	q_5	**N.D.**
q_3	q_4	q_4	**N.D.**
q_4	q_4	q_5	**N.D.**
q_5	q_5	q_5	**N.D.**

i) Dibuja el grafo que describe al autómata.

ii) Probar que q_0 y q_1 son equivalentes.

iii) Probar que q_2, q_3, q_4 son equivalentes.

iv) Hallar el autómata mínimo correspondiente.

Cuestión 4.8.6. Sea G una gramática sobre el alfabeto $\{a, b\}$ cuyas reglas de producción son las siguientes:

$$\begin{array}{lcl} Q_0 & \mapsto & bA \mid \lambda \\ A & \mapsto & bB \mid \lambda \\ B & \mapsto & aA \end{array}$$

Hallar un autómata que acepte el lenguaje generado por esa gramática. Hallar el autómata mínimo que acepte ese lenguaje. Hallar una expresión regular que describa ese lenguaje.

Cuestión 4.8.7. Dado un autómata A que acepta el lenguaje L, ¿hay un autómata que acepta el lenguaje L^R?, ¿cómo le describirías?.

Cuestión 4.8.8. Dado un autómata A que acepta el lenguaje descrito por una expresión regular α y dado un símbolo a del alfabeto, ¿Cómo sería el autómata finito que acepta el lenguaje $L(D_a(\alpha))$?

4.8.2. Problemas

Problema 4.8.1. Construir autómatas que acepten los siguientes lenguajes:

i) $L_1 := \{\omega \in \{a,b\}^* \ : \ abab \text{ es una subcadena de } \omega\}$.

ii) $L_2 := \{\omega \in \{a,b\}^* \ : \ \text{ni } aa \text{ ni } bb \text{ son subcadenas de } \omega\}$.

iii) $L_3 := \{\omega \in \{a,b\}^* \ : \ ab \text{ y } ba \text{ son subcadenas de } \omega\}$.

iv) $L_4 := \{\omega \in \{a,b\}^* \ : \ bbb \text{ no es subcadena de } \omega\}$.

Problema 4.8.2. Hallar un autómata determinista equivalente al autómata indeterminista $A := (Q, \{0,1\}, q_0, F, \delta)$, donde

- $\Sigma := \{0,1\}$,
- $Q := \{q_0, q_1, q_2, q_3, q_4\}$,
- $F := \{q_4\}$.

Y δ es dado por la tabla siguiente:

δ	**a**	**b**	λ
q_0	**N.D.**	q_2	q_1
q_1	q_0, q_4	**N.D.**	q_2, q_3
q_2	**N.D.**	q_4	**N.D.**
q_3	q_4	**N.D.**	**N.D.**
q_4	**N.D.**	**N.D.**	q_3

Problema 4.8.3. Minimizar el autómata sobre el alfabeto $\{0,1\}$ dado por las propiedades siguientes:

- $\Sigma := \{0,1\}$,
- $Q := \{q_0, q_1, q_2, q_3, q_4, q_5\}$,
- $F := \{q_3, q_4\}$.

δ	**0**	**1**	λ
q_0	q_1	q_2	**N.D.**
q_1	q_2	q_3	**N.D.**
q_2	q_2	q_4	**N.D.**
q_3	q_3	q_3	**N.D.**
q_4	q_4	q_4	**N.D.**
q_5	q_5	q_4	**N.D.**

Hallar su grafo, una gramática que genere el mismo lenguaje y una expresión regular que lo describa.

Problema 4.8.4. Construir una expresión regular y un autómata finito asociados al lenguaje siguiente:

$$L := \{\omega \in \{a,b\}^* \ : \ \exists z \in \{a,b\}^*, \ \omega = azb\}.$$

Problema 4.8.5. Hallar una expresión regular y una gramática asociadas al lenguaje aceptado por el autómata $A := (Q, \Sigma, q_0, F, \delta)$, dado por las propiedades siguientes

- $\Sigma := \{a,b\}$,
- $Q := \{q_0, q_1, q_2, q_3, q_4\}$,

- $F := \{q_3, q_4\}$.

Y δ es dado por la tabla siguiente:

δ	**a**	**b**	λ
q_0	q_1	**N.D.**	**N.D.**
q_1	q_2	q_4	**N.D.**
q_2	q_3	q_4	**N.D.**
q_3	q_3	q_4	**N.D.**
q_4	**N.D.**	q_4	**N.D.**

Problema 4.8.6. Hallar un autómata determinista que acepta el lenguaje descrito por la siguiente expresión regular:

$$a(bc)^*(b + bc) + a.$$

Minimiza el resultado obtenido.

Problema 4.8.7. Haz lo mismo que en el problema anterior para la expresión regular:

$$\left(a\,(ab)^*\right)^* da\,(ab)^*\,.$$

Problema 4.8.8. Haz lo mismo que en el problema anterior para la expresión regular:

$$0(011)^*0 + 10^*\left(1\,(11)^*\,0 + \lambda\right) + \lambda.$$

Problema 4.8.9. Calcula un autómata finito determinista minimal, una gramática regular y una expresión regular para el lenguaje siguiente:

$$L := \{\omega \in \{0,1\}^* \ : \ [\sharp\,(0\text{'s en } \omega) \in 2\mathbb{N}] \vee [\sharp\,(1\text{'s en } \omega) \in 3\mathbb{N}]\}.$$

Problema 4.8.10. Obtener una expresión regular para el autómata descrito por las siguientes propiedades: $A := (Q, \Sigma, q_0, F, \delta)$, y

- $\Sigma := \{a, b\}$,
- $Q := \{q_0, q_1, q_2\}$,
- $F := \{q_2\}$.

Y δ es dado por la tabla siguiente:

δ	**a**	**b**	λ
q_0	q_0, q_2	**N.D.**	q_1
q_1	q_2	q_1	**N.D.**
q_2	**N.D.**	**N.D.**	q_1

Problema 4.8.11. Dada la expresión regular $(ab)^*(ba)^* + aa^*$, hallar:

i) El autómata determinista minimal que acepta el lenguaje que describe esa expresión regular.

ii) Una gramática regular que genere dicho lenguaje.

Problema 4.8.12. Considera un tipo de datos real de algún lenguaje de programación. Halla una expresión regular que describa este lenguaje. Halla un autómata que los reconozca y una gramática que los genere.

Problema 4.8.13. Se considera el autómata descrito por la información siguiente $A := (Q, \Sigma, q_0, F, \delta)$, y

- $\Sigma := \{0, 1\}$,
- $Q := \{q_0, q_1, q_2, q_3, q_4, q_5\}$,
- $F := \{q_1, q_3, q_5\}$.

Y δ es dado por la tabla siguiente:

δ	**0**	**1**	λ
q_0	**N.D.**	**N.D.**	q_1
q_1	q_2	**N.D.**	**N.D.**
q_2	**N.D.**	q_3	**N.D.**
q_3	q_4	**N.D.**	q_1
q_4	q_5	**N.D.**	**N.D.**
q_5	**N.D.**	**N.D.**	q_3, q_1

Se pide:

i) Dibujar el grafo de transición del autómata.

ii) Decidir si 0101 es aceptado por al autómata y describir la computación sobre esta palabra.

iii) Hallar un autómata determinista que acepte el mismo lenguaje.

iv) Minimizar el autómata determinista hallado.

v) Hallar una expresión regular que describa el lenguaje aceptado por ese autómata.

vi) Hallar una gramática que genere dicho lenguaje.

Problema 4.8.14. Hallar un autómata que acepte las expresiones matemáticas con sumas y restas y sin paréntesis. Añadir a este una cinta donde escribir la traducción al español acabando en punto. Ejemplo:

$$4 + 3 - 5 \mapsto \textit{ cuatro mas tres menos cinco.}$$

Problema 4.8.15. Suponer que al autómata anterior se le quisiera añadir expresiones con paréntesis. Para hacer esto toma la expresión regular del autómata anterior α y se añade considera la siguiente expresión regular $(^*\alpha)^*$. Demostrar que el autómata no comprueba que todos los paréntesis que se abren son cerrados.

Problema 4.8.16. Otro de los problemas de los autómatas finitos es que no tienen en cuenta el orden entre los distintos elementos. Utilicemos una expresión regular α mencionada en el ejercicio anterior. Hallar el autómata que acepte el lenguaje generado por la siguiente expresión regular

$$('('+'\{')^* alpha(')'+'\}')^*.$$

Demostrar que el autómata no tiene en cuenta el orden de aparición de las llaves y los paréntesis.

Problema 4.8.17. Este ejercicio demuestra el problema de traducción para las estructuras condicionales. Suponemos que EXPRESION es conocido y los bucles condicionales están dados por la siguiente expresión regular:
((if EXPRESION then BUCLE) (else if EXPRESION then BUCLE))* Hallar un autómata finito que acepte el lenguaje dado por la expresión regular y discutir como añadir una cinta de traducción.

Capítulo 5

Gramáticas Libres de Contexto

Contents

5.1. Introducción 83
5.2. Árboles de Derivación de una Gramática 85
5.2.1. Un algoritmo incremental para la vacuidad. 86
5.3. Formas Normales de Gramáticas. 87
5.3.1. Eliminación de Símbolos Inútiles o Inaccesibles 87
5.3.1.1. Eliminación de Símbolos Inaccesibles. 88
5.3.1.2. Eliminación de Símbolos Inútiles. 89
5.3.2. Transformación en Gramáticas Propias. 90
5.3.2.1. Eliminación de $\lambda-$producciones. 90
5.3.2.2. Eliminación de Producciones Simples o Unarias 91
5.3.2.3. Hacia las Gramáticas Propias. 92
5.3.3. El Problema de Palabra para Gramáticas Libres de Contexto es Decidible. 94
5.3.4. Transformación a Formal Normal de Chomsky. 94
5.3.5. Forma Normal de Greibach 96
5.4. Cuestiones y Problemas 96
5.4.1. Cuestiones 96
5.4.2. Problemas 96

5.1. Introducción

El proceso de compilación es el proceso que justifica la presencia de un estudio teórico de lenguajes y autómatas como el que se desarrolla en esta asignatura. Sin embargo, el objetivo de un curso como éste no es, ni debe ser, el del diseño de un compilador, ni siquiera el del análisis de todos los procesos que intervienen en la compilación. Para culminar este proceso existe una asignatura dedicada a "Compiladores" (la asignatura llamada Procesadores de Lenguajes) en el curso cuarto de la titulación dentro del plan de estudios vigente.
Sin embargo, la pérdida de la motivación puede suponer la pérdida del interés (y del potencial atractivo) de una materia teórica y densa como la presente. En compensación, podemos recuperar el esquema básico de un clásico como los dos volúmenes (que han influido intensamente en el diseño del presente manuscrito) como son [AhUl, 72a] y [AhUl, 72b]. Por tanto, dedicaremos una parte del curso al *parsing* (*Análisis Sintático*) sin más pretensiones que las de ubicar este proceso dentro del contexto de la compilación. En términos bibliográficos, alcanzaremos los tópicos del volumen I ([AhUl, 72a]) dejando el resto de los temas de [AhUl, 72b] para la asignatura correspondiente.

Antes de comenzar señalemos que los lenguajes de programación modernos son típicamente lenguajes dados por gramáticas libres de contexto. El uso de gramáticas en niveles más elevados de la jerarquía de Chomsky hace que el problema de palabra se vuelva un problema indecidible:

- En la Sección 9.9 ya hemos observado que el Problema de Palabra para Gramáticas Formales cualesquiera es indecidible, esto es, no puede existir ningún algoritmo que lo resuelva.
- Para gramáticas sensibles al contexto, el problema de decidir si el lenguaje que genera es vacío o no también es un problema indecidible.
- En cuanto al Problema de Palabra para gramáticas sensibles al contexto, el problema es **PSPACE**–completo para ciertas subclases, con lo que se hace impracticable para su uso en compilación.

De ahí la restricción a Gramáticas Libres de Contexto para los lenguajes de programación. Veremos en este Capítulo (véase la Subsección 5.3.3) que el problema de palabra para gramáticas libres de contexto es decidible. Esto significa que es posible diseñar un algoritmo/programa que realice la tarea siguiente:

Problema 5.1.1 (Detección de errores sintácticos CFG.)**.** Dado un lenguaje de programacion $L \subseteq \Sigma^*$, mediante una gramática libre de contexto G que lo genera, y dada una palabra $\omega \in \Sigma^*$ (un fichero) decidir si ω es un programa sintácticamente válido (i.e. una palabra aceptada) para ese lenguaje de programación.

Recordemos que una Gramática Libre de Contexto (CFG) o de Tipo 2 es dada por la siguiente definición.

Definición 5.1.1 (Gramáticas libres de contexto o de Tipo 2)**.** Llamaremos gramática libre de contexto a toda gramática $G = (V, \Sigma, q, P)$ tal que todas las producciones de P son del tipo siguiente:

$$A \mapsto \omega, \text{ donde } A \in V \text{ y } \omega \in (\Sigma \cup V)^*.$$

Un lenguaje **libre de contexto** es un lenguaje generado por una gramática libre de contexto.

El sistema de transición asociado a una gramática libre de contexto es el mismo que asociamos a una gramática cualquiera. Usaremos el símbolo $C \vdash_G C'$ para indicar que la configuración C' es deducible de la configuración C mediante computaciones de G.
Esta sería la primera acción asociada a un compilador: resolver el problema de palabra para un lenguaje de programación fijado a priori. En otras palabras, ser capaz de decidir si un fichero es un programa o devolver al programador un mensaje de error.
El problema de palabra en general admite también como input la gramática que lo genera. Sin embargo, la situación usual es que nuestro lenguaje de programación está fijado. Por tanto, el problema a resolver no tiene a la gramática como input sino, simplemente, la palabra. Esto es,

Problema 5.1.2 (Errores sintácticos con lenguaje pre–fijado.)**.** Fijado un lenguaje de programación L decidir si una palabra $\omega \in \Sigma^*$ (un fichero) es un programa sintácticamente válido (i.e. una palabra aceptada) para ese lenguaje de programación.

En este caso de lenguaje fijado no se pide generar "el programa que decide" como parte del problema. Al contrario, se pre-supone que se dispone de ese programa (parser) y se desea que sea eficiente. El modelo de algoritmo natural que aparece en este caso es el *Autómata con Pila* (Pushdown Automata, PDA) que discutiremos en el Capítulo siguiente.
En el presente Capítulo nos ocupamos de resolver el Problema 5.1.1 y mostraremos cómo reducir a formas normales las Gramáticas Libres de Contexto, lo que simplificará el análisis de la equivalencia con los PDA's.
En el Capítulo próximo mostraremos los aspectos relativos a la equivalencia entre ambas concepciones: la gramática libre de contexto (como *generador del lenguaje*) y los PDA's (como *reconocedor/algoritmo de decisión*), resolviendo de paso el Problema 5.1.2.
Dejaremos para el Capítulo último el problema de la traducción que acompaña el proceso de compilación de manera esencial.

5.2. Árboles de Derivación de una Gramática

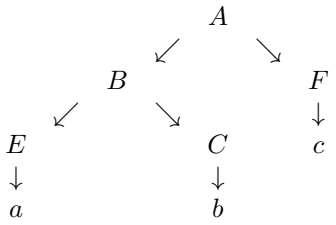
Definición 5.2.1 (Formas Sentenciales y Formas Terminales)**.** Llamamos formas sentenciales a todos los elementos ω de $(V \cup \Sigma)^*$. Llamaremos formas terminales a las formas sentenciales que sólo tienen símbolos en el alfabeto de símbolos terminales, es decir, a los elementos de Σ^*.

Definición 5.2.2 (Árbol de Derivación)**.** Sea $G := (V, \Sigma, Q_0, P)$ una gramática libre de contexto, sea $A \in V$ una variable. Diremos que un árbol $\mathcal{T}_A := (V, E)$ etiquetado es un árbol de derivación asociado a G si verifica las propiedades siguientes:

- La raíz del árbol es un símbolo no terminal (i.e. una variable).
- Las etiquetas de los nodos del árbol son símbolos en $V \cup \Sigma \cup \{\lambda\}$.
- Cada nodo interior está etiquetado con un símbolo en V (i.e. un símbolo no terminal).
- Cada hoja está etiquetada con una variable, un símbolo terminal o λ
- Si un nodo está etiquetado con una variable X y sus descendientes (leídos de izquierda a derecha) en el árbol son $X_1, \ldots, X_k$ entonces, hay una producción $X \mapsto X_1 \cdots X_k$ en G.

Ejemplo 5.2.1. Hallar árboles de derivación para las gramáticas siguientes:

- $A \mapsto BF, B \mapsto EC, E \mapsto a, C \mapsto b, F \mapsto c$, de tal manera que la raíz sea A y las hojas estén etiquetadas con a, b, c en este orden.



(5.2.1)

- $Q_0 \mapsto zABz, B \mapsto CD, C \mapsto c, D \mapsto d, A \mapsto a$, de tal manera que la raíz sea Q_0 y las hojas estén etiquetadas (de izquierda a derecha) mediante z, a, c, d, z.
- $Q_0 \mapsto aQ_0bQ_0 \mid bQ_0aQ_0 \mid \lambda$. Escribe árboles de derivación cuyas hojas tengan la lectura siguiente:
  - Una sóla hoja con λ.
  - Un árbol con varias hojas, tales que leyendo la palabra se obtenga *abab*.
  - Un árbol distinto, con varias hojas tales que leyendo la palabra se obtenga *abab*.

Proposición 5.2.1. *Sea $G := (V, \Sigma, Q_0, P)$ una gramática libre de contexto, sea $A \in V$ una variable. Sea $\mathcal{T}_A$ un árbol asociado a la gramática con raíz A. Sea $\omega \in (V \cup \Sigma)^*$ la forma sentencial obtenida leyendo de izquierda a derecha los símbolos de las hojas de $\mathcal{T}_A$. Entonces, $A \vdash_G \omega$. En particular, las formas sentenciales alcanzables desde el símbolo inicial Q_0 está representados por los árboles de derivación de Q_0.*

Demostración. Obvio a partir de la definición. □

Corollario 5.2.2. *Las palabras sobre el alfabeto Σ están en el lenguaje $L(G)$ generado por una gramática G si y solamente si existe un árbol de derivación cuyas hojas (leídas de izquierda a derecha) expresan la palabra ω.*

Demostración. Obvio. □

5.2.1. Un algoritmo incremental para la vacuidad.

Comenzaremos con un ejemplo de un algoritmo que será reutilizado, por analogía en los demás algoritmos de esta subsección. Su objeitvo consiste en mostrar que el problema de decidir si es o no vacío el lenguaje generado por una gramática libre de contexto.

Teorema 5.2.3 (Vacuidad de un lenguaje libre de contexto)**.** *El problema de la vacuidad de los lenguajes generados por gramáticas libres de contexto es decidible. Es decir, existe un algoritmo que toma como input una gramática libre de contexto G y devuelve una respuesta afirmativa si $L(G) \neq \emptyset$ y negativa en caso contrario.*

Demostración. Definimos el algoritmo siguiente:

INPUT: Una gramática libre de contexto $G = (V, \Sigma, Q_0, P)$.
$M := \emptyset$
$N := \{A \in V \ : \ (A \mapsto a) \in P, \ a \in \Sigma^*\}$
while $N \neq M$ **do**
$M := N$
$N := \{A \in V \ : \ (A \mapsto a) \in P, \ a \in (N \cup \Sigma)^*\} \cup N.$
endwhile
if $Q_0 \in N$, **then** OUPUT SI
else OUTPUT NO
fi

Obsérvese que este algoritmo tiene la propiedad de que los sucesivos conjuntos M y N construidos en su recorrido son siempre subcojuntos del conjunto de símbolos no terminales V. Por tanto, es un algoritmo que acaba sus cálculos en todos los datos de entrada.
Veamos que este algoritmo realiza las tareas prescritas. Para ello, consideremos una cadena de subconjuntos N_i de V que reflejan los sucesivos pasos por el ciclo while.
Escribamos $N_0 = \emptyset$ y denotemos por N_i al conjunto obtenido en el $i-$ésimo paso por el ciclo while, sin considerar la condición de parada. Esto es,

$$N_i := \{A \in V \ : \ (A \mapsto a) \in P, \ a \in (N_{i-1} \cup \Sigma)^*\} \cup N_{i-1}.$$

Está claro que tenemos una cadena ascendente

$$N_0 \subseteq N_1 \subseteq N_2 \subseteq \cdots \subseteq N_n \subseteq \cdots$$

Por construción observamos, además, que si existe un paso i tal que $N_i = N_{i+1}$, entonces, $N_i = N_m$ para todo $m \geq i + 1$.
Analicemos qué propiedades han de verificar las variables en N_i. Por inducción se probará lo siguiente:

Una variable $X \in V$ verifica que $X \in N_i$ si y solamente si existe un árbol de derivación de G de altura[1] *$i + 1$ que tiene X como raíz y cuyas hojas están etiquetadas con símbolos en $\Sigma \cup \{\lambda\}$.*

Una vez probada esta propiedad, es claro que se tiene:

- Sea i tal que nuestro algoritmo detiene sus cálculos tras i pasos por el ciclo while. Sea N el conjunto de variables calculado en ese paso. Entonces, $N = N_m$, $\forall m \geq i + 1$.
- Si $Q_0 \in N$, entonces, $N = N_{i+1}$ y existe un árbol de derivación de la gramática de altura $i + 2$ cuyas hojas son todo símbolos en $\Sigma \cup \{\lambda\}$ y cuya raíz es Q_0. Sea $\omega \in \Sigma^*$, la palabra descrita mediante la lectura (de izquierda a derecha) de las hojas del árbol. Entonces, $Q_0 \vdash_G \omega \in \Sigma^*$, luego $\omega \in L(G) \neq \emptyset$.

[1] Medimos altura por el número de nodos atravesados en el camino más largo.

- Por otro lado, si $\omega \in L(G) \neq \emptyset$ habrá un árbol de derivación de G cuya raíz es Q_0 y cuyas hojas tienen sus etiquetas en $\Sigma \cup \{\lambda\}$ producen, leyendo de izquierda a derecha, ω. Sea m la altura de tal árbol ($m \geq 1$, obviamente) y, por tanto, $Q_0 \in N_{m-1} \subseteq N_{i+1} = N$ para cualquier m.

□

5.3. Formas Normales de Gramáticas.

El objetivo de esta Sección es la de realizar una serie de reducciones algorítmicas (transformaciones de gramáticas) hasta reducir una gramática libre de contexto a una gramática en Forma Normal de Chomsky. Las diferentes subsecciones están basadas en las progresivas reducciones y simplificaciones.

Definición 5.3.1. Dos gramáticas libres de contexto G y G' se dicen equivalentes, si generan el mismo lenguajes, esto es, si $L(G) = L(G')$.

De ahora en adelante, todas las transformaciones de gramáticas serán transformaciones que preserven la considión de "ser equivalentes".

5.3.1. Eliminación de Símbolos Inútiles o Inaccesibles

Definición 5.3.2 (Símbolos Inútiles)**.** Sea $G := (V, \Sigma, Q_0, P)$ una gramática libre de contexto. Llamamos símbolos útiles de G a todos los símbolos (terminales o no) $X \in V \cup \Sigma$ tales que existen $\alpha, \gamma \in (V \cup \Sigma)^*$ y $\omega \in \Sigma^*$ de tal modo que:

$$Q_0 \vdash_G \alpha X \gamma, \quad \text{y} \quad \alpha X \gamma \vdash_G \omega.$$

Los símbolos inútiles son los que no son útiles.

Ejemplo 5.3.1. Consideremos la gramática $G := (\{Q_0, A, B\}, \{a, b\}, Q_0, P)$, donde las producciones de P son dadas por:

$$P := \{Q_0 \mapsto a \mid A, A \mapsto AB, B \mapsto b\}.$$

Obsérvese que A, B, b son símbolos inútiles en esta gramática. La razón es que el lenguaje aceptado es $\{a\}$. Si, por el contrario, añadiéramos la producción $A \mapsto a$, entonces, todos ellos serían símbolos útiles.

Definición 5.3.3. Sea $G := (V, \Sigma, Q_0, P)$ una gramática libre de contexto.

- Llamamos símbolos productivos (o fecundos) de G a todos los símbolos no terminales $X \in V$ tales que existe $\omega \in \Sigma^*$ tal que $X \vdash_G \omega$. Son improductivos (o infecundos) a los que no satisfacen esta propiedad.
- Llamamos símbolos accesibles de G a todos los símbolos (terminales o no) $X \in V \cup \Sigma$ tales que existen $\alpha, \gamma \in (V \cup \Sigma)^*$ de tal modo que:

  $$Q_0 \vdash_G \alpha X \gamma.$$

  Se llaman inaccesibles a los que no son accesibles.

Ejemplo 5.3.2. Nótese que si X es un símbolo útil, se han de producir dos propiedades. De una parte, la propiedad $Q_0 \vdash_G \alpha X \gamma$ que nos dice que X es accesible. De otra parte, por estar en una gramática libre de contexto, ha de existir $\beta \in \Sigma^*$ tal que $X \vdash_G \beta$. Esto es necesario porque, al ser libre de contexto, todas las producciones se basan en reemplazar una variales por formas sentenciales. Si la variable X no alcanzara nunca una forma terminal en el sistema de transición, entonces, $\alpha X \beta$ tampoco alcanzaría una forma terminal contradiciendo el hecho de ser X útil. La

existencia de $\beta \in \Sigma^*$ tal que $X \vdash_G \beta$ nos dice que X es un símbolo fecundo o productivo. En el siguiente ejemplo:

$$P := \{Q_0 \mapsto AB \mid CD, A \mapsto AQ_0, B \mapsto b, C \mapsto Cc \mid \lambda, D \mapsto d\}.$$

El símbolo B es fecundo y accesible, pero es un símbolo inútil.

Proposición 5.3.1. *Si $G := (V, \Sigma, Q_0, P)$ es una gramática libre de contexto, entonces los símbolos útiles son productivos y accesibles. El recíproco no es cierto.*

Demostración. Es obvia la implicación enunciada. En cuanto a ejemplos que muestran que el recíproco no es en general cierto, baste con ver las gramáticas expuestas en los Ejemplos 5.3.1 y 5.3.2 anteriores. □

Proposición 5.3.2. *Si $G := (V, \Sigma, Q_0, P)$ es una gramática libre de contexto, y libre de símbolos infecundos, entonces todo símbolo es útil si y solamente si es accesible.*

Demostración. En ausencia de símbolos infecundos accesibilidad es sinónimo de utilidad. La prueba es la obvia. □

Proposición 5.3.3 (Eliminación de símbolos infecundos)**.** *Toda gramática libre de contexto es equivalente a una gramática libre de contexto sin símbolos infecundos. Además, dicha equivalencia puede hacerse de manera algorítmica.*

Demostración. El algoritmo es esencialmente el propuesto en el Teorema 5.2.3 anterior:

INPUT: Una gramática libre de contexto $G = (V, \Sigma, Q_0, P)$.
$M := \emptyset$
$N := \{A \in V \ : \ (A \mapsto a) \in P, \ a \in \Sigma^*\}$
while $N \neq M$ **do**
$M := N$
$N := \{A \in V \ : \ (A \mapsto a) \in P, \ a \in (N \cup \Sigma)^*\} \cup N.$
endwhile
if $Q_0 \notin M$, **then** OUTPUT $(\{Q_0\}, \Sigma, Q_0, \emptyset)$
else OUTPUT $\bar{G} := (V \cap N, \Sigma, Q_0, \bar{P})$, donde $\bar{P}$ son las producciones de P que involucran solamente símbolos en $(V \cap N) \cup \Sigma \cup \{\lambda\}$
fi

Por la prueba del Teorema 5.2.3, sabemos que N es justamente el conjunto de variables productivas y el algoritmo realiza la tarea pretendida. □

5.3.1.1. Eliminación de Símbolos Inaccesibles.

Teorema 5.3.4. *[Eliminación de Símbolos Inaccesibles] Toda gramática libre de contexto es equivalente a una gramática libre de contexto sin símbolos inaccesibles. Además, dicha equivalencia puede hacerse de manera algorítmica.*

Demostración. El siguiente algoritmo elimina símbolos inaccesibles de una gramática libre de contexto. La demostración de que es un algoritmo y de que realiza la tarea prevista es análoga a la demostración del Teorema 5.2.3 anterior. Nótese que, de facto, el algoritmo calcula los símbolos que sí son accesibles.

INPUT: Una gramática libre de contexto $G = (V, \Sigma, Q_0, P)$.
$M := \{Q_0\}$
$N := \{X \in V \cup \Sigma \ : \ \exists A \in M, \ \exists \alpha, \beta \in (V \cup \Sigma)^*, \text{ con } A \mapsto \alpha X \beta \text{ en } P\}.$

while $N \neq M$ **do**

$M := N$
$N := \{X \in V \cup \Sigma \ : \ \exists A \in M,\ \exists \alpha, \beta \in (V \cup \Sigma)^*\ ,\ \text{con } A \mapsto \alpha X \beta \ \text{ en } P\}.$
endwhile

OUTPUT: La gramática $\bar{G} = (\bar{V}, \bar{\Sigma}, Q_0, \bar{P})$, con
$\bar{V} := N \cap V,\ \ \bar{\Sigma} := N \cap \Sigma,$
$\bar{P} := \{$Las producciones de P que sólo contienen los elementos de $\bar{V} \cup \bar{\Sigma}\}$.

□

5.3.1.2. Eliminación de Símbolos Inútiles.

Teorema 5.3.5. *[Eliminación de Símbolos Inútiles] Toda gramática libre de contexto es equivalente a una gramática sin símbolos inútiles. Además, esta equivalencia es calculable algorítmicamene.*

Demostración. Utilizaremos un algoritmo que combina los dos algoritmos descritos anteriormente y el enunciado de la Proposición 5.3.2. Primero eliminamos los símbolos infecundos y luego los inaccesibles.

INPUT: Una gramática libre de contexto $G = (V, \Sigma, Q_0, P)$.

Eliminar Símbolos Infecundos

$M := \emptyset$
$N := \{A \in V \ : \ (A \mapsto a) \in P,\ a \in \Sigma^*\}$

while $N \neq M$ **do**
$M := N$
$N := \{A \in V \ : \ (A \mapsto a) \in P,\ a \in (N \cup \Sigma)^*\} \cup N.$
endwhile

$G_1 := (V_1, \Sigma, Q_0, P_1)$, donde
$V_1 := V \cap N,$
$P_1 := \{$Las producciones en P que no involucran símbolos fuera de $V_1 \cup \Sigma\}$.

Eliminar Símbolos Inaccesibles de G_1

$M := \{Q_0\}$
$N := \{X \in V_1 \cup \Sigma \ : \ \exists A \in M,\ \exists \alpha, \beta \in (V \cup \Sigma)^*\ ,\ \text{con } A \mapsto \alpha X \beta \ \text{ en } P\}.$

while $N \neq M$ **do**
$M := N$
$N := \{X \in V_1 \cup \Sigma \ : \ \exists A \in M,\ \exists \alpha, \beta \in (V \cup \Sigma)^*\ ,\ \text{con } A \mapsto \alpha X \beta \ \text{ en } P\}.$

endwhile

OUTPUT: La gramática $\bar{G} = (\overline{V_1}, \overline{\Sigma}, Q_0, \overline{P_1})$, con
$\overline{V_1} := N \cap V_1$, $\overline{\Sigma} := N \cap \Sigma$,
$\overline{P} := \{$Las producciones de P que sólo contienen los elementos de $\overline{V_1} \cup \overline{\Sigma}\}$.

□

5.3.2. Transformación en Gramáticas Propias.

En nuestro camino hasta la forma normal de Chomsky, continuaremos con transformaciones de las gramáticas libes de contexto hasta obtener gramáticas propias.

5.3.2.1. Eliminación de $\lambda-$producciones.

Definición 5.3.4. Sea $G = (V, \Sigma, Q_0, P)$ una gramática libre de contexto.

i) Llamaremos $\lambda-$producciones en G a todas las producciones de la forma $X \mapsto \lambda$, donde $X \in V$ es un símbolo no terminal.

ii) Diremos que la gramática G es $\lambda-$libre si verifica una de las dos propiedades siguientes:

- O bien no posee $\lambda-$producciones,
- o bien la única $\lambda-$producción es de la forma $Q_0 \mapsto \lambda$ y Q_0 no aparece en el lado derecho de ninguna otra producción de P (es decir, no existe ninguna producción de la forma $X \mapsto \alpha Q_0 \beta$, con $\alpha, \beta \in (V \cup \Sigma)^*$).

Ejemplo 5.3.3. Consideremos la gramática cuyas producciones son:

$$Q_0 \mapsto aQ_0bQ_0 \mid bQ_0aQ_0 \mid \lambda.$$

No es una gramática $\lambda-$libre.

Teorema 5.3.6 (Transformación a Gramática $\lambda-$libre)**.** *Toda gramática libre de contexto es equivalente a una gramática $\lambda-$libre. Además, dicha equivalencia es calculable algorítmicamente.*

Demostración. El algoritmo comienza con una tarea que repite esencialmente lo hecho en algoritmos anteriores. Se trata de hacer desaparecer las variables que van a parar a la palabra vacía λ; pero de manera selectiva. No las eliminamos completamente porque podrían ir a parar a constantes o formas terminales no vacías.
Hallar $V_\lambda := \{A \in V \ : \ A \vdash_G \lambda\}$.
A partir del cálulo de V_λ procedemos de la forma siguiente:

i) Calculamos el nuevo sistema de producciones $\overline{P}$ del modo siguiente:

- Consideremos todas las producciones de la forma siguiente:

  $$A \mapsto \alpha_0 B_1 \alpha_1 \cdots B_k \alpha_k,$$

  donde $\alpha_i \notin V_\lambda^*$, $B_i \in V_\lambda$ y no todos los α_i son iguales a λ. Definamos

  $$\overline{P} := \overline{P} \cup \{A \mapsto \alpha_0 X_1 \alpha_1 \cdots X_k \alpha_k \ : \ X_i \in \{B_i, \lambda\}\}.$$

- Consideremos todas las producciones de la forma siguiente:

  $$A \mapsto B_1 \cdots B_k,$$

  donde $B_i \in V_\lambda$. Definamos:

  $$\overline{P} := \overline{P} \cup (\{A \mapsto X_1 \cdots X_k \ : \ X_i \in \{B_i, \lambda\}\} \setminus \{A \mapsto \lambda\}).$$

ii) Eliminamos todas las $\lambda-$producciones restantes.

iii) Finalmente, si $Q_0 \in V_\lambda$ sea $\overline{V} := V \cup \{Q_0'\}$, con $Q_0' \notin V$. Y añadamos

$$\overline{P} = \overline{P} \cup \{Q_0' \mapsto Q_0 \mid \lambda\}.$$

En otro caso, $\overline{V} = V$.

El output será la gramática $\overline{G} := (\overline{V}, \Sigma, Q_0', \overline{P})$ y satisface las propiedades pretendidas. □

Observación 5.3.1. La eliminación de $\lambda-$producciones puede tener un coste exponencial en el máximo de las longitudes de las formas sentenciales (en $(\Sigma \cup V)^*$) que aparecen a la derecha de las producciones de la gramática dada.

5.3.2.2. Eliminación de Producciones Simples o Unarias

Definición 5.3.5 (Producciones Simples o Unarias)**.** Se llaman producciones simples (o unarias) a las producciones de una gramática libre de contexto de la forma $A \mapsto B$, donde A y B son símbolos no terminales.

Teorema 5.3.7. *[Eliminación de Producciones Simples] Toda gramática $\lambda-$libre es equivalente a una gramática $\lambda-$libre y sin producciones simples. Esta equivalencia es calculable algorítmicamente.*

Demostración. El algoritmo tiene dos partes. La primera parte sigue el mismo esquema algorítmico usado en resultados anteriores. La segunda parte se dedica a eliminar todas las producciones simples.

- *Clausura Transitiva de símbolos no terminales.* Se trata de calcular, para cada $A \in V$, el conjunto siguiente:

$$W_A := \{B \in V \ : \ A \vdash B\} \cup \{A\}.$$

Nótese que se trata de la clausura transitiva en el grafo $(V, \to)$, inducido por el sistema de transición sobre el conjunto de variables. El algoritmo obvio funciona del modo siguiente:

INPUT: Una gramática libre de contexto $G := (V, \Sigma, Q_0, P)$ y $\lambda-$libre.

Para cada $A \in V$ calcular

$M_A := \emptyset$

$N_A := \{A\}$

while $N_A \neq M$ **do**

$M := N_A$

$N_A := \{C \in V \ : \ B \mapsto C \text{ está en P, y } B \in N_A\} \cup N_A$

endwhile

OUTPUT: Para cada $A \in V$, N_A.

- También podemos definir el conjunto de los antepasados de una variable

$$V_A := \{B \in V \ : \ B \vdash A\} := \{B \in V \ : \ A \in W_B\}.$$

Es calculable por el algoritmo anterior del modo obvio.

- *Eliminar las producciones simples.* Para cada variable B tal que existe una producción simple $A \mapsto B$ en P, procederemos como sigue:
  - Hallar todos los X's tales que $B \in W_X$ (o, equivalentemente, los X's en V_B) .
  - Para cada producción $B \mapsto \alpha$ que no sea producción simple, añadir a P la producción $X \mapsto \alpha$.
  - Eliminar toda producción del tipo $X \mapsto B$.

Nótese que cada iteración de la parte segunda del proceso añade producciones no simples y elimina al menos una producción simple. Con ello se alcanza el objetivo buscado. □

5.3.2.3. Hacia las Gramáticas Propias.

Definición 5.3.6. Diremos que una gramática libre de contexto $G := (Q, \Sigma, Q_0, F, \delta)$ es acíclica (o libre de ciclos) si no existe ningún símbolo no terminal $A \in V$ tal que existe una computación no trivial (en el sistema de transición asociado):

$$A \to \omega_1 \to \cdots \to \omega_k = A.$$

Definición 5.3.7 (Gramáticas Propias)**.** Diremos que una gramática libre de contexto $G := (Q, \Sigma, Q_0, F, \delta)$ es propia si verifica las siguientes propiedades:

- G es acíclica,
- G es $\lambda-$libre,
- G es libre de símbolos inútiles.

Lema 5.3.8. *[Interacción entre los algoritmos expuestos] Se dan las siguientes propiedades:*

i) Si G es una gramática libre de contexto que es $\lambda-$libre y está libre de producciones simples, entonces G es acíclica.

ii) Sea G es una gramática libre de contexto, $\lambda-$libre. Sea $\bar{G}$ la gramática obtenida después de aplicar a G el algoritmo de eliminación de producciones simples descrito en la demostración del Teorema 5.3.7. Entonces, $\bar{G}$ sigue siendo $\lambda-$libre.

iii) Sea G es una gramática libre de contexto, libre de producciones simples y $\lambda-$libre. Sea $\bar{G}$ la gramática obtenida después de aplicar a G el algoritmo de eliminación de símbolos inútiles descrito en la demostración del Teorema 5.3.5. Entonces, $\bar{G}$ sigue siendo libre de producciones simples y $\lambda-$libre.

Demostración.

i) Supongamos que la gramática fuera $\lambda-$libre y libre de producciones simples, pero hubiera un estado que generara un ciclo. Es decir, supongamos que existe:

$$A \to \omega_1 \to \cdots \to \omega_k \to A,$$

con $k \geq 1$. Entonces, puedo suponer que

$$\omega_k := \alpha_0 X_1 \alpha_1 \cdots \alpha_{n-1} X_n \alpha_n,$$

donde $\alpha_i \in \Sigma^*$, $X_i \in V$. En primer lugar, obsérvese que, como estamos hablando de gramáticas libres de contexto, las únicas producciones que se aplican son de la forma $B \to \gamma$. Si alguno de los α_i fuera distinto de la palabra vacía λ, no habría forma de "borrarlos" usando las producciones libre de contexto (para quedarnos solamente con un símbolo no terminal como A). Por tanto, $\alpha_i = \lambda$ para cada i, $0 \leq i \leq n$. En conclusión, sólo tenemos (como última acción):

$$\omega_k = X_1 \cdots X_n \to A.$$

Si, además, $n \geq 2$, con una única producción libre de contexto, no podríamos eliminar nada más que un símbolo no terminal. Con lo cual no podríamos obtener A. Por tanto, tenemos, en realidad, $n \leq 2$. Tenemos dos casos $n = 1$ o $n = 2$. Si $n = 1$ tendremos:

$$\omega_k = X_1 \to A.$$

En el este caso debería existir la producción simple $X_1 \mapsto A$, pero hemos dicho que nuestra gramática es libre de producciones simples. Por tanto, el segundo caso ($n = 1$) no puede darse. Nos queda un único caso $n = 2$ y tendremos:

$$\omega_k = X_1 X_2 \to A.$$

En este caso hay dos opciones simétricas, con lo que discutiremos sólo una de ellas: $X_1 = A$ y $X_2 \mapsto \lambda$ es una producción. Pero nuestra gramática es $\lambda-$libre. Si hay una producción $X_2 \mapsto \lambda$ es sólo porque X_2 es el símbolo inicial $X_2 = Q_0$. Por tanto, tenemos una computación:

$$A \to \omega_1 \to \cdots \to \omega_{k-1} \to AQ_0 \to A,$$

Pero, para llegar a AQ_0 desde A tiene que ocurrir que Q_0 esté en la parte derecha de alguna producción (lo cual es imposible por la propia defínción de $\lambda-$libre. Luego no uede haber ciclos.

ii) Retomemos el algoritmo descrito en la prueba del Teorema 5.3.7. Una vez hemos calculado V_A para cada A, y tratamos las producciones unarias del tipo $A \mapsto B$ del modo siguiente:

- Hallar todos los X's tales que $B \in W_X$.
- Para cada producción $B \mapsto \alpha$ que no sea producción simple, añadir a P la producción $X \mapsto \alpha$.
- Eliminar la producción $A \mapsto B$.

Ahora bien, si G es una gramática $\lambda-$libre, y si Q_0 es el estado inicial, Q_0 no puede estar en la derecha de ninguna producción. En particular, no existen producciones simples de la forma $A \mapsto Q_0$. Por tanto, toda producción simple $A \mapsto B$ de P verifica que $B \neq Q_0$. De otro lado, como G es $\lambda-$ libre, para todo $B \neq Q_0$, las producciones de la forma $B \mapsto \alpha$ verifican que $\alpha \neq \lambda$. Por tanto, ninguna de las producciones que añadimos en este proceso es una $\lambda-$producción. Y, por tanto, $\bar{G}$ sigue siendo $\lambda-$libre.

iii) Basta con observar que el algoritmo de eliminación de símbolos inútiles solamente elimina símbolos y producciones que los involucran. Pero no añade producciones. Por ello, si no había en G ninguna producción simple, tampoco la habrá en $\bar{G}$. Si G era $\lambda-$libre, también lo será $\bar{G}$ puesto que no añadimos $\lambda-$ producciones.

□

Teorema 5.3.9. *Toda gramática libre de contexto es equivalente a una gramática propia. Dicha equivalencia es calculable algorítmicamente.*

Demostración. Basta con unir los algoritmos antes expuestos en el orden adecuado que indica el Lema 5.3.8 anterior. El proceso será el siguiente:

INPUT: Una gramática G libre de contexto.

- *Hallar* G_1 la gramática $\lambda-$libre obtenida aplicando a G, mediante el algoritmo del Teorema 5.3.6.
- *Hallar* G_2 la gramática obtenida de aplicar a G_1 el algoritmo del Teorema 5.3.7.
- *Hallar* G_3 la gramática obtenida de aplicar a G_2 el algoritmo del Teorema 5.3.5.

OUTPUT: G_3

El algoritmo anterior realiza la tarea prescrita. La gramática G_1 es claramente $\lambda-$libre como consecuencia del Teorema 5.3.6. Como consecuencia del apartado 2 del Lema 5.3.8, como la gramática G_1 es $\lambda-$libre, también es $\lambda-$libre la gramática G_2. De otro lado, el Teorema 5.3.7 nos garantiza que G_2 es libre de producciones simples.
Como consecuencia del apartado 3 del Lema 5.3.8, la gramática G_3 sigue siendo una gramática $\lambda-$libre y libre de producciones simples. Asimismo, el Teorema 5.3.5 nos garantiza que G_3 es libre de símbolos inútiles.
Finalmente, el apartado 1 del Lema 5.3.8, nos garantiza que G_3 es acíclica. Por tanto, verifica todas las propiedades para ser una gramática propia. □

5.3.3. El Problema de Palabra para Gramáticas Libres de Contexto es Decidible.

En la Sección 9.9 hemos descrito algunos resultados relativos a la indecidibilidad del Problema de palabra de sistemas de semi–Thue (o gramáticas de Tipo 0). En esta subsección mostaremos que este problema es decidible para gramáticas libres de contexto. Es decir, se trata de mostrar un algoritmo que resuelve el problema siguiente:

Problema de Palabra para Gramáticas libres de Contexto. Dada una gramática libre de contexto $G = (V, \Sigma, Q_0, P)$ y dada una palabra $\omega \in \Sigma^*$, decidir si $\omega \in L(G)$.

Lema 5.3.10. *Sea $G = (V, \Sigma, Q_0, P)$ una gramática libre de contexto y $\lambda-$libre. Sea $\omega \in L(G)$ una palabra aceptada por la gramática y sea:*

$$Q_0 \to \alpha_1 \to \alpha_2 \to \cdots \to \alpha_n = \omega,$$

una derivación aceptadora de ω, donde $\alpha_i \in (V \cup \Sigma)^*$ son formas sentenciales de la gramática. Entonces, la longitud de cada una de estas formas sentenciales verifica:

$$|\alpha_i| \leq |\omega|, \ \forall i.$$

Demostración. De hecho, basta con observar que si tenemos dos configuraciones (i.e. dos formas sentenciales) $c \to_G c'$ y si la gramática es $\lambda-$libre, entonces o bien $c' = \lambda$ (en cuyo caso $c = Q_0$ forzosamente) o bien $|c| \leq |c'|$ (dado que no suprimimos ninguna variable, al reemplazarla nos sale, al menos, un símbolo y la longitud no puede disminuir). □

Teorema 5.3.11. *El problema de palabra es decidible para gramáticas libres de contexto.*

Demostración. Basta con usar el Lema anterior. El procedimiento es el siguiente: En primer lugar, trasnformamos nuestra gramática original en una gramática $\lambda-$libre. posteriormente, dado ω, construiremos un grafo $G_\omega := (N_\omega, E_\omega)$ con ls reglas siguientes:

- Los vértices del grafo N_ω son todas las palabras de $(V \cup \Sigma)^*$ de longitud menor o igual que la longitud de ω.
- Las aristas del grafo E_ω son los pares $(c, c') \in N_\omega$ tales que $c \to_G c'$.

A partir de del grafo G_ω, calculamos la clausura transitiva de Q_0: $CT_G(Q_0)$. Entonces, usando el Lema anterior, ω está en $L(G)$ si y solamente si está en la clausura transitiva de Q_0. □

Observación 5.3.2. Decidibilidad no significa eficiencia. Es decir, el hecho de la existencia de un algoritmo para el problema de palabra no singifica de modo inmediato que se pueda usar ese algoritmo para la detección de errores. De hecho, debe haber un pre–proceamiento para el cálculo del autómata (con pila, como veremos) que decide el problema de palabra y luego sólo debe usarse el autómata para cada palabra concreta. En otro caso estaríamos consumiendo una enormidad de tiempo de cálculo para cada verificación de correctitud de una palabra.

5.3.4. Transformación a Formal Normal de Chomsky.

Definición 5.3.8 (Forma Normal de Chomsky)**.** Una gramática libre de contexto $G := (Q, \Sigma, Q_0, F, \delta)$ se dice que está en forma normal de Chomsky si es $\lambda-$libre y las únicas producciones (exceptuando, eventualmente, la única $\lambda-$ producción $Q_0 \mapsto \lambda$), son exclusivamente de uno de los dos tipos siguientes.

- $A \mapsto b$, con $A \in V$ y $b \in \Sigma$,

- $A \mapsto CD$, con $A, C, D \in V$.

Nótese que es acíclica porque carece de producciones unarias. En la definición, bien podrímos haber supuesto que es propia sin cambiar la esencia de la definición. Hemos dejado la habitual. De otro lado, debe señalarse que el modelo se corresponde al modelo de codificación de polinomios (en este caso sobre anillos no conmutativos) llamado *"straight–line program"*.

Teorema 5.3.12 (Transformación a forma normal de Chomsky)**.** *Toda gramática libre de contexto es equivalente a una gramática libre de contexto en forma normal de Chomsky. Además, esta equivalencia es algorítmicamente computable.*

Demostración. Bastará con que demos un algoritmo que transforma gramáticas propias en gramáticas en forma normal de Chomsky. Así, supongamos que tenemos una gramática $G = (V, \Sigma, Q_0, P)$ propia. Procederemos del modo siguiente:
Definamos un par de clases $\bar{V}$ y $\bar{P}$ de símbolos no terminales y producciones, conforme a las reglas siguientes:

- Inicializar con $\bar{V} := V$, $\bar{P} = \emptyset$.
- Si $Q_0 \mapsto \lambda$ está en P, añadir $Q_0 \mapsto \lambda$ a $\bar{P}$ sin modificar $\bar{V}$.
- Si en P hay una producción del tipo $A \mapsto a \in \Sigma$ entonces, añadir $A \mapsto a$ a $\bar{P}$ sin modificar $\bar{V}$.
- Si en P hay una producción del tipo $A \mapsto CD$, con $C, D \in V$, está en P, entonces, añadir $A \mapsto CD$ a $\bar{P}$ sin modificar $\bar{V}$.
- Finalmente, Para cada producción en P del tipo

  $$A \mapsto X_1 \cdots X_k,$$

  con $X_i \in V \cup \Sigma$ que no sea de ninguno de los tres tipos anteriores[2] realizar las tareas siguientes:

  - Para cada i tal que $X_i \in V$, no modificar $\bar{V}$
  - Para cada i tal que $X_i \in \Sigma$, añadir a $\bar{V}$ una nueva variable $\bar{X}_i$, distinta a todas las que ya estuvieran en $\bar{V}$. Añadir a $\bar{P}$ la producción $\bar{X}_i \mapsto X_i$ en este caso. (Obs'ervese que, en este caso, aparece una producción del Tipo 1).
  - Añadir $\bar{P}$ la producción $A \mapsto X_1' \cdots X_k'$, donde X_i' viene dada por:

    $$X_i' := \begin{cases} X_i, & \text{si } X_i \in V \\ \bar{X}_i, & \text{en otro caso} \end{cases}$$

  - Si $k = 2$, no modificar.
  - Si $k > 2$, reemplazar en $\bar{P}$, la producción $A \mapsto X_1' \cdots X_k'$ por una cadena de producciones:

    $$A \mapsto X_1'Y_2, Y_2 \mapsto X_2'Y_3, \cdots, Y_{k-1} \mapsto X_{k-1}'X_k',$$

    añadiendo a $\bar{V}$ las variables $\{Y_2, \ldots, Y_{k-1}\}$.
- OUTPUT: $(\bar{V}, \Sigma, Q_0, \bar{P})$.

Es claro que el algoritmo descrito verifica las propiedades deseadas. □

Observación 5.3.3. Obsérvese que los árboles de derivación asociados a gramáticas en forma normal de Chomsky son árboles binarios cuyas hojas vienen de nodos con salida unaria.

[2]Obsérvese que $k \geq 2$ puesto que no hay producciones simples. Si $k = 2$, al no ser de ninguno de los tipos anteriores, son o bien dos símbolos en Σ o bien uno es un símbolos en Σ y el otro está en V. En cualquier caso se aplica el mismo método.

5.3.5. Forma Normal de Greibach

Es otra "normalización" de las gramáticas libres de contexto que hace referencia al trabajo de Sheila A. Greibach en Teoría de Autómatas. Si la Forma Normal de Chomsky se corresponde con la presentación de polinomios como straight–line programs, la forma de Greibach se corresponde a la codificación mediante monomios.

Definición 5.3.9 (Producciones Monomiales)**.** Una gramática $G := (V, \Sigma, Q_0, P)$ se dice en forma normal de Greibach si es $\lambda-$libre y las únicas producciones (exceptuando, eventualmente, la única $\lambda-$ producción $Q_0 \mapsto \lambda$) pertenecen al tipo siguiente:

$$A \mapsto \alpha X_1 \cdots X_k,$$

donde $A \in V$ es una variable, $\alpha \in \Sigma$ es un símbolo terminal (posiblemente λ) y $X_1, \ldots, X_k \in V^*$ es una lista (posiblemente vacía) de símbolos no terminales (variables en V) entre los cuales no está el símbolo inicial Q_0.

Obviamente toda gramática libre de contexto es equivalente a una gramática en Forma Normal de Greibach.

5.4. Cuestiones y Problemas

5.4.1. Cuestiones

Cuestión 5.4.1. Comprobar, utilizando las siguientes producciones de una gramática G, que al convertir una gramatica a $\lambda-$libre, puede quedar con símbolos inútiles:

$$S \mapsto a \mid aA, A \mapsto bB, B \mapsto \lambda.$$

Cuestión 5.4.2. Decidir si existe un algoritmo que realice la tarea siguiente:
Dada una gramática libre de contexto G y dadas dos formas sentenciales de la gramática c y c', el algoritmo decide si $c \vdash_G c'$.

Cuestión 5.4.3. Sean L_1 y L_2 dos lenguajes libres de contexto. Decidir si es libre de contexto el lenguaje siguiente:

$$L := \bigcup_{n \geq 1} (L_1)^n (L_2)^n.$$

Cuestión 5.4.4. Hallar una estimación del número de pasos necesarios para generar una palabra de un lenguaje libre de contexto, en el caso en que la gramática que lo genera esté en forma normal de Chomsky.

Cuestión 5.4.5. Discutir si alguno de los siguientes lenguajes es un lenguaje incontextual:

i) $\{\omega \in \{a,b\}^* \ : \ \omega = \omega^R, \ \forall x, y \in \{a,b\}^*, \omega \neq xabay\}$.

ii) $\{a^i b^j c^k \ : \ i = j \vee j = k\}$.

iii) $\{a^i b^j c^k d^l \ : \ (i = j \wedge k = l) \vee (i = l \wedge j = k)\}$.

iv) $\{xcy \ : \ x, y \in \{a,b\}^* \wedge \sharp_a(x) + \sharp_b(y) \in 2\mathbb{Z} \wedge |x| = |y|\}$.

5.4.2. Problemas

Problema 5.4.1. Dada una gramática libre de contexto G, con las siguientes producciones:

$$Q_0 \mapsto AB \mid 0Q_01 \mid A \mid C, A \mapsto 0AB \mid \lambda, B \mapsto B1 \mid \lambda.$$

Se pide:

- Eliminar los símbolos inútiles
- Convertirla en $\lambda-$libre
- Eliminar las producciones unitarias

Problema 5.4.2. Eliminar las variables improductivas en la gramática G con las siguientes producciones:

$$Q_0 \mapsto A \mid AA \mid AAA,\ A \mapsto ABa \mid ACa \mid a, B \mapsto ABa \mid Ab \mid \lambda,$$

$$C \mapsto Cab \mid CC, D \mapsto CD \mid Cd \mid CEa, E \mapsto b.$$

Eliminar los símbolos inaccesibles en la gramática resultante.

Problema 5.4.3. Hallar una gramática $\lambda-$libre equivalente a la siguiente:

$$Q_0 \mapsto aQ_0a \mid bQ_0b \mid aAb \mid bAa, A \mapsto aA \mid bA \mid \lambda.$$

¿Es una gramática propia?.

Problema 5.4.4. Hallar una gramática propia equivalente a la siguiente:

$$Q_0 \mapsto XY, X \mapsto aXb \mid \lambda, Y \mapsto bYc \mid \lambda.$$

Problema 5.4.5. Sea $G = (V, \Sigma, Q_0, P)$ la gramática libre de contexto dada por las propiedades siguientes:

- $V := \{Q_0, X, Y, Z, T\}$,
- $\Sigma := \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, \times, (,)\}$,
- Las producciones en P están dadas por:

$$Q_0 \mapsto X \mid X + Q_0, X \mapsto T \mid Y \times Z,$$
$$Y \mapsto T \mid (X + Q_0), Z \mapsto Y \mid Y \times Z,$$
$$T \mapsto 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9.$$

Se pide:

i) Hallar la clase de formas terminales de esta gramática.

ii) Hallar el lenguaje generado por esta gramática.

iii) Eliminar producciones unarias.

iv) Eliminar producciones inútiles.

v) Convertirla en una gramática propia.

vi) Transformarla en forma Normal de Chomsky.

Problema 5.4.6. Hacer los mismos pasos 3 a 6 del problema anterior con la gramática siguiente:

$$Q_0 \mapsto A \mid B, A \mapsto aA \mid aAb \mid a, B \mapsto Bd \mid ABa \mid b.$$

Problema 5.4.7. Eliminar $\lambda-$producciones y hacer los mismos pasos del problema anterior con la gramática siguiente:

$$Q_0 \mapsto ABQ_0 \mid BAQ_0 \mid \lambda, A \mapsto bAA \mid a, B \mapsto aBB \mid b.$$

Problema 5.4.8. Dar un algoritmo que decida si el lenguaje generado por una gramática libre de contexto es finito o infinito.

Capítulo 6

Autómatas con Pila.

Contents

6.1. Noción de Autómatas con Pila. 99
6.1.1. Las Pilas como Lenguaje (Stacks). 99
6.2. Sistema de Transición Asociado a un Autómata con Pila. 102
6.2.1. Modelo gráfico del sistema de transición. 102
6.2.2. Transiciones: Formalismo. 102
6.2.3. Codificación del Autómata con Pila. 104
6.3. Lenguaje Aceptado por un Autómata con Pila. 106
6.4. Equivalencia con Gramáticas Libres de Contexto. 110
6.5. Propiedades Básicas 112
6.6. Problemas 114
6.6.1. Problemas 114

6.1. Noción de Autómatas con Pila.

Antes de pasar a definir un autómata con pila (o *Pushdown Automata*, PDA) recordemos (superficialmente) la estructura de datos pila (stack) vista como lenguaje formal y las funciones y relaciones que la caracterizan.

6.1.1. Las Pilas como Lenguaje (Stacks).

Podemos identificar las pilas con ciertos lenguajes formales sobre un nuevo alfabeto Γ. Comenzaremos añadiendo un nuevo símbolo Z_0 que no está en Γ. Las pilas (stacks) son elementos del lenguaje: $Z_0 \cdot \Gamma^*$. El símbolo Z_0 se identificará con el significado *Fondo de la Pila*[1]
Tendremos unas ciertas funciones sobre pilas:

- `empty`: Definimos la aplicación

$$\texttt{empty} : Z_0 \cdot \Gamma^* \longrightarrow \{0, 1\},$$

dada mediante:

$$\texttt{empty}(Z_0 Z) := \begin{cases} 1, & \text{si } Z = \lambda \\ 0, & \text{en otro caso} \end{cases}$$

[1] El ímbolo de 'fondo de la pila' no está suficientemente estandarizado. Diversos autores usan diveras variantes de símbolos como $\S, \sharp, \$, []$ y otros. Usaremos Z_0 como la simplificación menos molesta de todas esas variaciones.

- `top`: Definimos la aplicación

$$\texttt{top} : Z_0 \cdot \Gamma^* \longrightarrow \Gamma \cup \{\lambda\},$$

mediante la regla siguiente:
Dada una pila $Z_0 \cdot Z \in Z_0 \cdot \Gamma^*$ (con $Z = z_1 \cdots z_n \in \Gamma^*$),

$$\texttt{top}(Z_0 Z) := \begin{cases} z_n \in \Gamma, & \text{si } Z = z_1 \cdots z_n \in \Gamma^*,\ Z \neq \lambda \\ Z_0, & \text{en caso contrario} \end{cases}$$

Obsérvese que hemos elegido leer Z_0 cuando la pila está vacía[2]

- `push`: Apilar (empujar) una pila encima de otra. Definimos la aplicación

$$\texttt{push} : Z_0 \cdot \Gamma^* \times \Gamma^* \longrightarrow Z_0 \cdot \Gamma^*,$$

mediante la regla siguiente:
Dada una pila $Z_0 \cdot Z \in Z_0 \cdot \Gamma^*$ (con $Z = z_1 \cdots z_n \in \Gamma^*$), y una palabra $x \in \Gamma^*$, dada mediante: $x := x_1 \cdots x_r$, definimos

$$\texttt{push}(Z_0 Z, x) := Z_0 z_1 \cdots z_n x_1 \cdots x_r \in Z_0 \cdot \Gamma^*.$$

- `pop` *(Pull Out the toP):* Definimos la aplicación

$$\texttt{pop} : Z_0 \cdot \Gamma^* \longrightarrow Z_0 \cdot \Gamma^*,$$

mediante la regla siguiente:
Dada una pila $Z_0 \cdot Z \in Z_0 \cdot \Gamma^*$, definimos $\texttt{pop}(Z_0 Z)$ como el resultado de eliminar $\texttt{top}(Z_0 Z)$, esto es

$$\texttt{pop}(Z_0 Z) := \begin{cases} Z_0 z_1 \cdots z_{n-1} \in Z_0 \cdot \Gamma^*, & \text{si } Z = z_1 \cdots z_n \in \Gamma^*,\ Z \neq \lambda \\ Z_0, & \text{en caso contrario} \end{cases}$$

Obsérvese que el símbolo "fondo de pila" no se borra al hacer la operación `pop`.

Observación 6.1.1. Una de las propiedades básicas de las operaciones es, obviamente, la siguiente:

$$\texttt{push}(\texttt{pop}(Z_0 Z), \texttt{top}(Z_0 Z)) = Z_0 Z.$$

Definición 6.1.1 (Non–Deterministic Pushdown Automata)**.** Un autómata con pila (o Pushdown Automata) indeterminista es una lista $A := (Q, \Sigma, \Gamma, q_0, F, Z_0, \delta)$ donde:

- Q es un conjunto finito cuyos elementos se llaman estados y que suele denominarse espacio de estados,
- Σ es un conjunto finito (alfabeto),
- Γ es un conjunto finito llamado "alfabeto de la pila".
- q_0 es un elemento de Q que se denomina estado inicial,
- F es un subconjunto de Q, cuyos elementos se denominan estados finales aceptadores,
- Z_0 es un símbolo que no está en el alfabeto Γ y que se denomina "fondo de la pila".

[2] Podríamos también haber definido $\texttt{top}(Z_0) = \lambda$. Esta corrección subjetiva la hemos hecho para enfatizar el hecho de que la pila está vacía. En caso de hacer la elección $\texttt{top}(Z_0) = \lambda$, deberían modificarse todas las notaciones que siguen de modo conforme.

- δ es una correspondencia:

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \times (\Gamma \cup \{Z_0\}) \longrightarrow Q \times \Gamma^*,$$

que se denomina función de transición y que ha de verificar la propiedad siguiente[3] para cada lista $(q, x, A) \in Q \times (\Sigma \cup \{\lambda\}) \times (\Gamma \cup \{Z_0\})$:

$$\sharp\left(\{(q,\omega) \in Q \times \Gamma^* \ : \ \delta(q,x,A) = (q,\omega)\}\right) < \infty.$$

Es decir, sólo un número finito de elementos de $Q \times \Gamma^*$ estarán relacionados con cada elemento (q, x, A) mediante la función de transición.

Además, impondremos la condición siguiente[4]:

- $\delta(q, x, Z_0) \neq (q', \lambda)$ para cualesquiera $q, q' \in Q$ y $x \in \Sigma \cup \{\lambda\}$ (i.e. no se "borra" el 'fondo de la pila').

Observación 6.1.2. De nuevo, como en el caso de Autómatas Finitos indeterministas, hemos preferido usar una notación funcional menos correcta del tipo

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \times (\Gamma \cup \{Z_0\}) \longrightarrow Q \times \Gamma^*,$$

para representar correspondencias, que la notación como aplicación más correcta:

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \times (\Gamma \cup \{Z_0\}) \longrightarrow \mathcal{P}\left(Q \times \Gamma^*\right).$$

La notación elegida pretende, sobre todo, enfatizar la diferencia entre el caso determinístico (δ es aplicación) frente al indeterminístico (δ es correspondencia).

Observación 6.1.3. Nótese que hemos supuesto que la función de transición δ tiene su rango (el conjunto hacia el que va a parar) en $Q \times \Gamma^*$. Esta condición nos dirá (más adelante) que no podemos escribir en la pila el símbolo de "fondo de pila" nada más que cuando se escriba en la configuración inicial. Podremos, sin embargo, leerlo. No estará, en ningún caso, "en medio" de la pila.

El determinismo en autómatas con pila difiere del caso de autómatas finitos. No vamos a exigir que δ sea aplicación sino algo más delicado.

Definición 6.1.2 (Autómata con Pila Determinista)**.** Un autómata con pila indeterminista $A := (Q, \Sigma, \Gamma, q_0, Z_0, \delta)$ se denomina determinista si verifica las siguientes dos propiedades:

- La imagen de cualquier lectura contiene a lo sumo 1 elemento. Es decir, para cualesquiera $(q, x, A) \in Q \times (\Sigma \cup \{\lambda\}) \times (\Gamma \cup \{Z_0\})$, el conjunto de los elementos relacionados con él a través de δ tiene, a lo sumo, 1 elemento:

$$\sharp\left(\{(p,\omega) \in Q \times \Gamma^* \ : \ \delta(q,x,A) = (p,\omega)\}\right) \leq 1.$$

- Si dados $q \in Q$ y $A \in \Gamma$, existieran $(p, \omega) \in Q \times \Gamma^*$ tales que $\delta(q, \lambda, A) = (q, \omega)$, entonces, ninguno de los elementos de $Q \times \Sigma \times (\Gamma \cup \{Z_0\})$ tiene imagen por δ. Es decir, si

$$\sharp\left(\{(p,\omega) \in Q \times \Gamma^* \ : \ \delta(q,\lambda,A) = (p,\omega)\}\right) = 1,$$

entonces

$$\sharp\left(\bigcup_{x \in \Sigma} \{(p,\omega) \in Q \times \Gamma^* \ : \ \delta(q,x,A) = (p,\omega)\}\right) = 0.$$

[3]Recuérdese que si X es un conjunto finito, denotamos por $\sharp(X)$ su cardinal (i.e. el número de sus elementos).

[4]La primera de ellas indica que no podemos "borrar" el símbolo de "fondo de pila", aunque sí podemos leerlo. La pila "vacía es una pila que comienza en el símbolo Z_0.

Observación 6.1.4. No es cierto, en el caso de autómatas con pila, que todo autómata con pila indeterminista sea equivalente a un autómata con pila determinista. Así, el siguiente lenguaje es aceptado por un autómata con pila indeterminista, pero no puede ser aceptado por un atómata con pila determinista:

$$L := \{a^n b^m c^n \ : \ n, m \geq 1\} \bigcup \{a^n b^m c^m \ : \ n, m \geq 1\} \subseteq \{a, b, c\}^*.$$

6.2. Sistema de Transición Asociado a un Autómata con Pila.

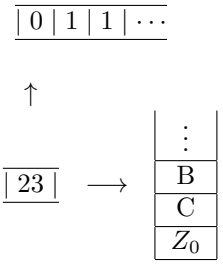
Sea dado un autómata $A := (Q, V, \Sigma, q_0, F, Z_0, \delta)$. El espacio de configuraciones es el producto $\mathcal{S}_A := Q \times \Sigma^* \times Z_0 \cdot V^*$. Dada una palabra $\omega \in \Sigma^*$, la configuración inicial vendrá dada por:

$$I_A(\omega) := (q_0, \omega, Z_0),$$

esto es, escribimos ω en la cinta de trabajo , escribimos el estado inicial en la unidad de control y escribimos Z_0 en la pila como tareas de inicialización.

6.2.1. Modelo gráfico del sistema de transición.

Gráficamente podemos dibujar el autómata con pila mediante los siguientes elementos. El alfabeto de la "cinta" será $\Sigma = \{0, 1\}$. El alfabeto de la pila será $\Gamma = \{A, B, C\}$. El espacio de estados serán los primeros cien números naturales, i.e. $Q = \{0, 1, 2, 3, \ldots, 98, 99, 100\}$. Una representación gráfica de la configuración será:



En este dibujo, el estado es 23 y la unidad de control "lee" el primer símbolo de la cinta de entrada 0 y también "lee" el símbolo B en la pila. Supondremos que, en la pila, "lee" siempre el símbolo que se encuentra más arriba en la pila (el `top`), lo que supone que, en el dibujo, por encima de B no hay nada en la pila.

6.2.2. Transiciones: Formalismo.

Las transiciones son de los tipos siguientes:

- **Transiciones Read/Push.** Son transiciones entre dos configuraciones:

  $$(q, x, Z_0 Z) \to_A (q', x', Z_0 Z'),$$

  donde $q \in Q$ y

  $$x := x_1 \cdots x_r, \quad Z_0 Z = Z_0 z_1 \cdots z_n.$$

  Realizamos las siguientes operaciones:

  - Read Leemos la información (q, x_1, z_n) (es decir, el estado, el primer símbolo de la palabra y el `top` de la pila). Supondremos $x_1 \neq \lambda$.
  - Transition Aplicamos la función de transición obteniendo $\delta(q, x_1, z_n) = (q', Y)$, con $Y \in \Gamma^*$, $Y \neq \lambda$.

- Push and Move Entonces,
  - $q' = q$ (cambia el estado de la transición).
  - $x' = x_2 \cdots x_r$ ("borramos" un símbolo de la palabra a analizar (move).
  - $Z' := Z_0 z_1 \cdots z_{n-1} Y$, es decir, $Z_0 Z' := \texttt{push}(\texttt{pop}(Z_0 Z), Y)$.

- **Transiciones Read/Pop:** Son transiciones entre dos configuraciones:

$$(q, x, Z_0 Z) \to_A (q', x', Z_0 Z'),$$

donde $q \in Q$ y

$$x := x_1 \cdots x_r, \quad Z_0 Z = Z_0 z_1 \cdots z_n.$$

Realizamos las siguientes operaciones:

- Read Leemos la información (q, x_1, z_n) (es decir, el estado, el primer símbolo de la palabra y el `top` de la pila). Supondremos $x_1 \neq \lambda$.
- Transition Aplicamos la función de transición obteniendo $\delta(q, x_1, z_n) = (q', \lambda)$ (ésta es la caracterización de las acciones Read/Pop). Indica que debemos hacer `pop`.
- Pop and Move Entonces,
  - $q' = q$ (cambia el estado de la transición).
  - $x' = x_2 \cdots x_r$ ("borramos" el primer símbolo de la palabra a analizar (move).
  - $Z_0 Z' := Z_0 z_1 \cdots z_{n-1} := \texttt{pop}(Z_0 Z) = \texttt{push}(\texttt{pop}(Z_0 Z), \lambda)$.

- **Transiciones Lambda/Push.** Son transiciones entre dos configuraciones:

$$(q, x, Z_0 Z) \to_A (q', x', Z_0 Z'),$$

donde $q \in Q$ y

$$x := x_1 \cdots x_r, \quad Z_0 Z = Z_0 z_1 \cdots z_n.$$

Realizamos las siguientes operaciones:

- Read Lambda En este caso, no se lee la cinta a unque sí se lee la pila. Leeremos (q, λ, z_n) (es decir, el estado, la palabra vacía y el `top` de la pila).
- Transition Aplicamos la función de transición obteniendo $\delta(q, \lambda, z_n) = (q', Y)$, con $Y \in \Gamma^*$, $Y \neq \lambda$.
- Push Entonces,
  - $q' = q$ (cambia el estado de la transición).
  - $x' = x$ ("No borramos" un símbolo de la palabra a analizar).
  - $Z_0 Z' := Z_0 z_1 \cdots z_{n-1} Y := \texttt{push}(\texttt{pop}(Z_0 Z), Y)$.

- **Transiciones Lambda/Pop:** Son transiciones entre dos configuraciones:

$$(q, x, Z_0 Z) \to_A (q', x', Z_0 Z'),$$

donde $q \in Q$ y

$$x := x_1 \cdots x_r, \quad Z_0 Z = Z_0 y z_1 \cdots z_n, \text{con } z_n \neq Z_0.$$

Realizamos las siguientes operaciones:

- Read Lambda De nuevo, no se lee la cinta aunque sí se lee la pila. Tendremos (q, λ, z_n) (es decir, el estado, la palabra vacía y el `top` de la pila).
- Transition Aplicamos la función de transición obteniendo $\delta(q, \lambda, z_n) = (q', \lambda)$. Obsérvese que, en este caso, se ha obtenido λ como símbolo de la pila, ésto indica que debemos hacer `pop`.

- Pop and Move Entonces,
  - $q' = q$ (cambia el estado de la transición).
  - $x' = x$ ("No borramos" un símbolo de la palabra a analizar).
  - $Z_0Z' := Z_0z_1 \cdots z_{n-1} := \texttt{pop}(Z_0Z) = \texttt{push}(\texttt{pop}(Z_0Z), \lambda)$.

Observación 6.2.1. Es importante señalar que la diferencia entre "instrucciones" **/Push** y **/Pop** es "artificial". La mantenemos por razones didácticas. Nótese que

$$\texttt{pop}(Z_0Z) = \texttt{push}(\texttt{pop}(Z_0Z), \lambda)$$

y

$$\texttt{push}(Z_0Z, Y) = \texttt{push}(\texttt{pop}(Z_0Z), Y),$$

con $\omega \in \Gamma^*$ no tienen diferencias semánticas significativas porque, obviamente, $\lambda \in \Gamma^*$. Las distinguimos para que el lector pueda ver la operación "borrar el último dígito de la pila" como una operación distinguida.

Observación 6.2.2. Los autómatas finitos del Capítulo anterior se pueden releer como autómatas con pila del modo siguiente: Suponemos que la función de transición δ verifica que

$$\delta(q, x, z) = (q', \lambda), \forall (q, x, z) \in Q \times (\Sigma \cup \{\lambda\}) \times \Gamma^*.$$

En este caso, todas las instrucciones pasan por hacer $\texttt{push}(\texttt{pop}(Z_0), \lambda)$ que no cambia el contenido de la pila desde la configuración inicial.

Proposición 6.2.1. *Si A es un autómata con pila determinista, su sistema de transición $(S_A, \to_A)$ es determinista. Es decir, dada una configuración $c \in S_A$, existirá a lo sumo una única configuración $c' \in S_A$ tal que $c \to_A c'$.*

Demostración. Siguiendo las dos hipótesis de la Definición de Autómata Determinista: Dada una configuración, uno pude hacer una transición **Read/...** o una **Lambda/...**. Si cabe la posibilidad de hacer una transición **Lambda/...**, no habrá ninguna transición que permita hacer lectura (por la segunda de las condiciones impuestas). Si, por el contrario, no hay ninguna transición **Lambda/...**, entonces es forzoso hacer una transición de lectura y ésta es, a lo sumo, única. □

Observación 6.2.3. Esta propiedad nos garantiza que la ejecución de un autómata con pila determinista es posible dado que a cada configuración le sigue o bien una única configuración siguiente o bien una salida (hacia estado final aceptador o de rechazo) por no tener ninguna opción de continuar.

6.2.3. Codificación del Autómata con Pila.

La introducción que hemos hecho de la noción de Autómata con Pila se basa en varios principios básicos. El primero es que los Autómatas con Pila son "expresables" sobre un alfabeto finito, del mismo modo que lo fueron los autómatas. Una vez visto que expresable sobre un alfabeto finito y visto el sistema de transición podemos admitir (espero) que se trata de un programa que es ejecutable en algún tipo de intérprete que sea capaz de seguir las "instrucciones" asociadas a la transición. Es, obviamente, simple diseñar un programa (en Java, C++, C o cualquier lenguaje) asociado a un autómata (con la salvedad de los problemas de determinismo/indeterminismo: no tiene sentido "programar" autómatas no deterministas porque su sistema de transición no es determinista).
Sin embargo, podemos utilizar diversas representaciones del autómata. La más simple es la de tabla. Para ello, podemos usar dos entradas. De una parte, el producto cartesiano $E := Q \times (\Sigma \cup \{\lambda\}) \times (\Gamma \cup \{Z_0\})$ que será el conjunto de las entradas de la función de transición. De otro lado tenemos un conjunto infinito de las "salidas" $O := Q \times \Gamma^*$, pero nuestra hipótsis nos dice que sólo un número finito de elementos de $Q \times \Gamma^*$ van a entrar en la relación. Así tendremos una tabla como la siguiente:

Ejemplo 6.2.1. El PDA viene dado por:

- $Q := \{q_0, q_1, r, s\}$,
- $\Sigma := \{0, 1\}$,
- $\Gamma := \{A\}$,
- Estado Inicial q_0.
- Símbolo de fondo de pila Z_0.
- $F := \{r\}$.

Tabla de transición:

E	**O**
$(q_0, 0, Z_0)$	(q_0, A)
$(q_0, 0, A)$	(q_0, AA)
$(q_0, 1, A)$	(q_1, λ)
$(q_1, 1, A)$	(q_1, λ)
(q_1, λ, Z_0)	(r, λ)
(q_1, λ, A)	(s, λ)
$(q_1, 1, Z_0)$	(s, λ)

Tomemos como entrada la palabra $0^3 1^3 \in \Sigma^*$.
Inicializamos
$I := (q_0, 000111, Z_0)$
Ahora las sucesivas computaciones serán dadas por la secuencia siguiente:

- *Read/Push* $I \to_A c_1 = (q_0, 00111, Z_0 A)$
- *Read/Push* $c_1 \to_A c_2 = (q_0, 0111, Z_0 AA)$
- *Read/Push* $c_2 \to_A c_3 = (q_0, 111, Z_0 AAA)$
- *Read/Pop* $c_3 \to_A c_4 = (q_1, 11, Z_0 AA)$
- *Read/Pop* $c_4 \to_A c_5 = (q_1, 1, Z_0 A)$
- *Read/Pop* $c_5 \to_A c_6 = (q_1, \lambda, Z_0)$
- *Lambda/Pop* $c_6 \to_A c_6 = (r, \lambda, Z_0)$

Si, por el contrario, escojo la palabra $0^3 1^2 \in \Sigma^*$ se produce el efecto siguiente:
Inicializamos
$I := (q_0, 00011, Z_0)$

- *Read/Push* $I \to_A c_1 = (q_0, 0011, Z_0 A)$
- *Read/Push* $c_1 \to_A c_2 = (q_0, 011, Z_0 AA)$
- *Read/Push* $c_2 \to_A c_3 = (q_0, 11, Z_0 AAA)$
- *Read/Pop* $c_3 \to_A c_4 = (q_1, 1, Z_0 AA)$
- *Lambda/Pop* $c_4 \to_A c_5 = (q_1, \lambda, Z_0 A)$
- *Lambda/Pop* $c_5 \to_A c_6 = (s, \lambda, Z_0)$

Finalmente, escojo la palabra $0^2 1^3 \in \Sigma^*$ se produce el efecto siguiente:
Inicializamos
$I := (q_0, 00111, Z_0)$

- *Read/Push* $I \to_A c_1 = (q_0, 0111, Z_0 A)$
- *Read/Push* $c_1 \to_A c_2 = (q_0, 111, Z_0 AA)$
- *Read/Pop* $c_2 \to_A c_3 = (q_1, 11, Z_0 A)$
- *Read/Pop* $c_3 \to_A c_4 = (q_1, 1, Z_0)$
- *Lambda/Pop* $c_4 \to_A c_5 = (s, \lambda, Z_0)$

Obsérvese que:

- Si tengo más ceros que unos, llegaré a leer (q_1, λ, A) con lo que acabaré en el estado s
- Si tengo más unos que ceros, llegaré a leer $(q_0, 1, z_0 y)$ con lo que acabaré en s.
- La única forma de llegar a r sería tener el mismo número de ceros que de unos. Más aún, las únicas palabras que llegan al estado r son las dadas por

$$L := \{0^n 1^n \ : \ n \in \mathbb{N}\}.$$

6.3. Lenguaje Aceptado por un Autómata con Pila.

Hay dos maneras de interpretar el lenguaje aceptado por un autómata con pila: por estado final aceptador y por pila y cinta vacías. Veremos además que ambas nociones de lenguajes son equivalentes, aunque, posiblemente, con diferentes autómatas. La razón de usar ambas nociones se justifica por la orientación de cada una. Así, el lenguaje aceptado por estado final aceptador extiende la noción de Autómata Finito y es extensible a nociones más abstractas de máquinas como los autómatas bi-direccionales o las máquinas de Turing. Por su parte, ver los lenguajes como lenguajes aceptados pr cinta y pila vacías nos simplificarán (en la Sección **??**) probar la relación con los lenguajes libres de contexto y, como veremos más adelantem estarán mejor adaptados al diseño de procesos de Análisis Sintáctico (porque simplifica el formalismo).
Sea $A := (Q, \Sigma, \Gamma, q_0, Z_0, F)$ un autómata con pila y sea $\mathcal{S}_A$ el sistema de transición asociado. Escribiremos $c \vdash_A c'$ cuando la configuración c' es alcanzable desde c en el sistema de transición $\mathcal{S}_A$.

Definición 6.3.1 (Lenguaje aceptado mediante estado final aceptador)**.**
Sea $A := (Q, \Sigma, \Gamma, q_0, Z_0, F, \delta)$ un autómata con pila y sea $\mathcal{S}_A$ el sistema de transición asociado. Para cada palabra $\omega \in \Sigma^*$, definimos la configuración inicial en ω a la configuración:

$$I_A(\omega) := (q_0, \omega, Z_0) \in \mathcal{S}_A.$$

Llamaremos lenguaje aceptado (mediante estado final final aceptador) por el autómata A (y lo denotaremos por $L_f(A)$) al conjunto siguiente:

$$L_f(A) := \{\omega \in \Sigma^* \ : \ I_A(\omega) \vdash_A (f, \lambda, z) \in \mathcal{S}_A, \ f \in F\}.$$

Este concepto nos lleva a un modelo de programa en el que la condición de parada viene dada por los estados finales aceptadores y por tener la cinta vacía. Es decir, un "programa" con la estructura siguiente:

INPUT: $\omega \in \Sigma^*$
Initialize: $I := (q_0, \omega, Z_0)$.

while $I \notin F \times \{\lambda\} \times Z_0\Gamma^*$ **do**
Hallar $c' \in S_A$ tal que $I \to_A c'$
Com.: Realiza un paso en el sistema de transición.
$I := c'$
od
OUTPUT: ACEPTAR
end

Definición 6.3.2 (Lenguaje aceptado mediante pila y cinta vacías)**.** Sea $A := (Q, \Sigma, \Gamma, q_0, Z_0, F)$ un autómata con pila y sea $\mathcal{S}_A$ el sistema de transición asociado. Llamaremos lenguaje aceptado (mediante pila y cinta vacías) por el autómata A (y lo denotaremos por $L_\emptyset(A)$) al conjunto siguiente:

$$L_\emptyset(A) := \{\omega \in \Sigma^* \ : \ I_A(\omega) \vdash_A (f, \lambda, Z_0) \in \mathcal{S}_A, f \in Q \setminus \{q_0\}^5\}.$$

La diferencia entre aceptar por pila vacía o por estado final es que en el caso de aceptar por estado final, la pila puede estar o no vacía mientras que en el caso de aceptación por pila vacía, la pila queda vacía, pero no nos preocupamos de cuál es el estado alcanzado.
En este segundo caso el "programa" tendrá la pinta siguiente:

INPUT: $\omega \in \Sigma^*$
Initialize: $I := (q_0, \omega, z_0 y)$.
while $I \notin Q \times \{\lambda\} \times \{Z_0\}$ **do**
Hallar $c' \in S_A$ tal que $I \to_A c'$
Com.: Realiza un paso en el sistema de transición.
$I := c'$
od
OUTPUT: ACEPTAR
end

Observación 6.3.1. Nótese que las palabras no aceptadas (i.e. OUTPUT: RECHAZAR) no se han incluido en una salida del bucle sino que se admite que puedan continuar indefinidamente dentro del bucle. Trataremos de clarificar el significado de este proceso.
Nótese también que los "programas" anteriores no tienen el sentido usual cuando el autómata es indeterminista. Lo cual es, parcialmente, causa de este formalismo admitiendo bucles infinitos.

La siguiente Proposición muestra la equivalencia entre ambas formas de aceptación, aunque será más cómodo utilizar el caso de pila vacía.

Proposición 6.3.1. *Un lenguaje es aceptado por algún autómata con pila mediante pila y cinta vacías si y solamente si es aceptado por algún autómata con pila (posiblemente otro distinto) mediante estado final aceptador. Es decir, Sea A un autómata con pila solbre un alfabeto Σ y sean $L_1 := L_f(A) \subseteq \Sigma^*$ y $L_2 := L_\emptyset(A) \subseteq \Sigma^*$, respectivamente los lenguajes aceptados por A mediante cinta y pila vacías o mediante estado final aceptador. Entonces, se tiene:*

i) Existe un autómata con pila B_1 tal que $L_1 = L_\emptyset(B_1)$,

ii) Existe un autómata con pila B_2 tal que $L_2 = L_f(B_2)$.

Demostración. Mostraremos un mecanismo de paso, construyendo para cada lenguaje $L_f(A)$ aceptado por un autómata con pila A mediante estado final aceptador un autómata con pila B_1 que acepta el mismo lenguaje mediante pila y cintas vacías y lo mismo para la segunda de las afiramciones Esto es, $L_f(A) = L_\emptyset(B_1)$.

[5]Si admitimos q_0 entonces, todo lenguaje aceptado por pila vacía debería contener la palabra vacía.

- Dado un autómata con pila $A := (Q, \Sigma, \Gamma, q_0, Z_0, F, \delta)$ que acepta el lenguaje $L_f(A)$ mediante estado final aceptador, construyamos el nuevo autómata que aceptará el mismo lenguaje mediante pila vacía $B_1 := (\bar{Q}, \Sigma, \bar{\Gamma}, \bar{q_0}, \bar{Z}_0, \bar{F}, \bar{\delta})$ del modo siguiente:
  - Sea $p_0, p_f \notin Q$ dos nuevos estados y definamos $\bar{Q} := Q \cup \{p_0, p_f\}$.
  - $\bar{q_0} := p_0, \bar{Z}_0 := Z_0$.
  - La idea clave consiste en introducir un nuevo símbolo en el alfabeto de la pila X_0 que "protegerá" el símbolo de fondo de la pila. Así, elegiremos $X_0 \notin \Gamma$ y $\bar{\Gamma} := \Gamma \cup \{X_0\}$.
  - $\bar{F} := F$, dejamos el mismo conjunto de estados finales aceptadores[6].
  - Definamos
$$\bar{\delta} : \bar{Q} \times \left(\bar{\Sigma} \cup \{\lambda\}\right) \times \left(\bar{\Gamma} \cup \{z_0 y\}\right) \to \bar{Q} \times \bar{\Gamma}^*,$$
    mediante:
    - $\bar{\delta}(p_0, w, Z_0) = (q_0, Z_0 X_0)$. Es decir, inicializamos "protegiendo" Z_0 con una variable X_0. La transformación será:
$$I_{B_1}(\omega) \to_A (q_0, \omega, Z_0 X_0).$$
    - Mientras "vivamos" en el "viejo" autómata no cambiamos la función de transición, es decir:
$$\bar{\delta} \mid_{Q \times (\Sigma \cup \{\lambda\}) \times \Gamma} = \delta,$$
      Aquí nos garantizamos que la variable protectora X_0 no será nunca añadida después de haberla usado por vez primera.
    - Para una transición $\bar{\delta}(q, w, X_0)$ hacemos lo que hubiera hecho el viejo autómata si estuviera leyendo la pila vacía. Así, si $\delta(q, w, Z_0) = (p, z)$ y $z \neq \lambda$ haremos:
$$\bar{\delta}(q, w, X_0) := \delta(q, w, Z_0) = (p, z).$$
      Varios elementos importantes a describir aquí:
      - Por nuestra definición de la funci'on `pop`, Z_0 no puede borrarse, por eso, si la segunda coordenada de $\delta(q, w, Z_0) = (q, \lambda)$, entonces definiremos:
$$\bar{\delta}(q, w, X_0) := (p, X_0).$$
      - La idea de esta transformación es que, durante los cálculos del "viejo autómata", pudiera ser que, en una etapa intermedia, se vaciase la pila sin haber acabado con la palabra. En ese caso, seguiríamos apilando información que podría ser útil en el resto de la computación.
    - Si $q \in F$, definimos $\bar{\delta}(q, \lambda, z) := (p_f, \lambda)$, para $z \in \Gamma$, $z \neq Z_0, X_0$.
    - Si $q \in F$, definimos $\bar{\delta}(q, \lambda, X_0) := (p_f, X_0)$.
    - Finalmente, definimos para cada $z \neq Z_0$, $\bar{\delta}(p_f, \lambda, z) = (p_f, \lambda)$.

  Para alcanzar el estado p_f, debemos alcanzar un estado final aceptador de F. Además, las configuraciones de B_1 tienen la forma siguiente para $q \neq p_0, p_f$:
$$(q, x, Z_0 X_0 Z), \text{ con } x \in \Sigma^*, Z \in \Gamma^*,$$
  y están identificadas con las configuraciones de A dadas mediante:
$$(q, x, Z_0 Z), \text{ con } x \in \Sigma^*, Z \in \Gamma^*,$$
  Las demás configuraciones son o bien la configuración inicial (q_0, x, Z_0) o configuraciones cuyo estado es p_f. Para una palabra $x \in \Sigma^*$, tendremos:
$$I_A(x) \vdash_A (q, x, Z_0 Z).$$

[6] Aunque bien podríamos haber añadido p_f a F y tendríamos el todo.

Si $q \in F$, el autómata B_1 habría calculado también:

$$I_{B_1}(x) \vdash_{\bar{A}} (q, \lambda, Z_0 X_0 Z).$$

Y, en la siguiente fase, procedería a vaciar la pila, usando $\bar{\delta}(q, \lambda, X_0) := (p_f, X_0)$ y $\bar{\delta}(p_f, \lambda, z) = (p_f, \lambda)$, para todo $z \neq Z_0$. Esto nos da $L_f(A) \subseteq L_\emptyset(B_1)$.

De otro lado, dada $x \in \Sigma^*$ si $x \in L_\emptyset(B_1)$, entonces, habremos realizado una computación que produce el efecto siguiente:

$$I_{B_1}(x) \vdash_{\bar{A}} (p_f, \lambda, Z_0).$$

Ahora bien, p_f sólo se alcanza tras una configuración final aceptadora de A, por lo que deberíamos haber calculado:

$$I_{B_1}(x) \vdash_{B_1} (q, x', Z_0 X_0 Z) \vdash_{B_1} (p_f, \lambda, Z_0),$$

con $q \in F$ en algún momento intermedio. Entonces, la acción del autómata B_1 nos permite garantizar que se tiene:

$$I_{B_1}(x) \vdash_{B_1} (q, x', Z_0 X_0 Z) \vdash_{B_1} (q, x', Z_0 X_0) \vdash_{B_1} (p_f, \lambda, Z_0),$$

Si $x' \neq \lambda$, las transiciones asociadas al borrado de la pila ($\bar{\delta}(q, \lambda, \alpha) := (p_f, \lambda)$ $\bar{\delta}(q, \lambda, X_0) := (p_f, X_0)$ y $\bar{\delta}(p_f, \lambda, z) = (p_f, Z_0)$) no nos permiten borrar contenido en la cinta. Por tanto, la única configuración final alcanzable sería:

$$I_{B_1}(x) \vdash_{B_1} (q, x', Z_0 X_0 Z) \vdash_{B_1} (q, x', Z_0 X_0) \vdash_{B_1} (p_f, x', Z_0).$$

Por tanto, sólo cabe la posibilidad de que $x' = \lambda$ con lo cual habremos hecho la computación mediante:

$$I_{B_1}(x) \vdash_{B_1} (q, \lambda, Z_0 X_0 Z) \vdash_{B_1} (p_f, \lambda, Z_0).$$

Y el autómata A habría seguido la computación:

$$I_A(x) \vdash_A (q, \lambda, Z_0 Z),$$

con lo que $L_\emptyset(B_1) \subseteq L(A)$.

- Recíprocamente, dado un autómata con pila $A := (Q, \Sigma, \Gamma, q_0, Z_0, F, \delta)$ que acepta el lenguaje $L_\emptyset(A)$ mediante pila y cinta vacías, construyamos el nuevo autómata que aceptará el mismo lenguaje mediante estados finales aceptadores $B_2 := (\bar{Q}, \bar{\Sigma}, \bar{\Gamma}, \bar{q_0}, \bar{Z_0}, \bar{F}, \bar{\delta})$ del modo siguiente. Introduciremos un estado final aceptador nuevo p_f y definimos $\bar{F} := \{p_f\}, \bar{Q} := Q \cup \{p_f\}$. Introducimos un nuevo símbolo inicial para la pila $\bar{Z_0} := X_0$ y definimos $\bar{\Gamma} := \Gamma \cup \{Z_0\}$. Ahora introducimos una nueva función de transición $\bar{\delta}$ definida del modo siguiente:

$$\bar{\delta} \mid_{Q \times (\Sigma \cup \{\lambda\}) \times \bar{\Gamma}} = \delta.$$

$$\bar{\delta}(q, \lambda, Z_0) := (p_f, \lambda).$$

$$\bar{\delta}(\bar{q_0}, \lambda, X_0) := (q_0, Z_0).$$

Es clara la identificación entre las configuraciones de A y las configuraciones de B_2 que poseen un estado de A:

$$(q, x, \alpha) \leftrightarrow (q, x, X_0 \alpha).$$

Ahora cosideramos $L_f(B_2)$ el lenguaje aceptado mediante estado final aceptador por B_2. Una palabra $x \in \Sigma^*$ es aceptada si se ha producido una computación cuyos extremos son:

$$I_{B_2}(x) \vdash_{B_2} (p_f, \lambda, Z_0 Z).$$

Ahora observamos que el estado p_f sólo se alcanza mediante $\lambda-$transiciones que leen el símbolo Z_0 en la pila (i.e. $\bar{\delta}(q, \lambda, Z_0) := (p_f, \lambda)$). Pero el símbolo Z_0 sólo se lee cuando la pila "original" está vacía (i.e. $Z_0 Z = Z_0$). Así, nuestra computación debe tener la forma:

$$I_{B_2}(x) \vdash_{B_2} (q, x', X_0 Z_0) \vdash_{B_2} (p_f, \lambda, X_0).$$

De otro lado, estas $\lambda-$transiciones no borran información en la cinta. Por tanto, con los mismos argumentos que en el apartado anterior, necesariamente ha de darse $x' = \lambda$ y existirán:

$$I_{B_2}(x) \vdash_{B_2} (q, \lambda, X_0 Z_0) \vdash_{B_2} (p_f, \lambda, X_0 Z_0),$$

que se corresponde a la computación en A

$$I_A(x) \vdash_A (q, \lambda, Z_0).$$

Con ello concluimos que $x \in L_\emptyset(A)$, es aceptado por A mediante pila y cinta vacías, y $L_f(B_2) \subseteq L_\emptyset(A)$.
De otro lado, supongamos que x es aceptado por A mediante pila y cinta vacías. En ese caso, tendremos una computación en A de la forma:

$$I_A(x) \vdash_A (q, \lambda, Z_0).$$

Esto se transforma en una computación en B_2 de la forma:

$$I_{B_2}(x) \vdash_{B_2} (q, \lambda, X_0 Z_0).$$

Aplicando la transición $\delta(q, \lambda, Z_0) := (p_f, \lambda)$ obtendremos:

$$I_{B_2}(x) \vdash_{B_2} (q, \lambda, X_0 Z_0) \vdash_{B_2} (p_f, \lambda, X_0)$$

y habremos probado que $L_\emptyset(A) \subseteq L_f(B_2)$ como pretendíamos.

□

6.4. Equivalencia con Gramáticas Libres de Contexto.

Teorema 6.4.1. *Los lenguajes libres de contexto son exactamente los lenguajes aceptados por los autómatas con pila mediante cinta y pila vacías. Es decir, se verifican las siguiente dos propiedades:*

i) Para cada gramática libre de contexto G sobre un alfabeto Σ de símbolos terminales, existe un autómata con pila A tal que $L(G) = L_\emptyset(A)$.

ii) Para cada autómata A con alfabeto de cinta Σ existe una gramática libre de contexto G tal que el lenguaje generado por G coincide con $L_\emptyset(A)$.

Más aún, daremos procedimientos de construcción en ambos sentidos.

Demostración. Dividiremos la prueba en las dos afirmaciones.

i) Bastará con lo probemos para gramáticas en forma normal de Chomsky. El resto se obtiene en las progresivas transformaciones de gramáticas. Así, supongamos que G es dada mediante $G := (V, \Sigma, q_0, P)$, donde q_0 es el símbolo inicial. Defniremos un autómata con pila $A := (Q, \Sigma, \Gamma, q_0, Z_0, F, \delta)$ de la forma siguiente:

- $Q := \{q_0\}$ posee un único estado (que es también el estado inicial).
- El símbolo de fondo de la pila es un símbolo auxiliar.
- El alfabeto de la pila reúne a todos los símbolos (terminales o no) de la gramática $\Gamma := V \cup \Sigma$.

- La función de transición δ estará dada del modo siguiente:
  - $\delta(q_0, \lambda, Z_0) := (q_0, Q_0)$ (al comenzar pongamos Q_0 justo encima del fondo de la pila).
  - Si la gramática tiene una producción del tipo $A \mapsto a \in \Sigma \cup \{\lambda\}$, escribamos[7]:

$$\delta(q_0, \lambda, A) := (q_0, a).$$

  - Si la gramática tiene una producción del tipo $A \mapsto CD$, con $C, D \in V$, pongamos:

$$\delta(q_0, \lambda, A) := (q_0, DC).$$

  - Finalmente, para cada $a \in \Sigma$, pongamos:

$$\delta(q_0, a, a) := (q_0, \lambda).$$

Para ver la demostración de la igualdad bastará con observar que la pila ejecuta un árbol de derivación de la gramática. Por tanto, basta con seguir (borrando) las hojas para ir borrando en la cinta. El vaciado de la cinta y de la pila se produce conforme vamos verificando las hojas.

ii) Para la segunda de las afirmaciones, consideremos dado un autómata con pila $A := (Q, \Sigma, \Gamma, q_0, Z_0, \delta)$ que acepta un lenguaje $L_\emptyset(A)$. Construyamos la gramática $G := (V, \Sigma, Q_0, P)$ mediante las definiciones siguientes:

- $V := Q \times (\Gamma \cup \{Z_0\}) \times Q \cup \{Q_0\}$. Utilizaremos la notación $\langle qAp \rangle$ para representar el símbolo no terminal $(q, A, p) \in V$[8].
- El símbolo inicial Q_0 lleva acompañada unas producciones del tipo siguiente:

$$Q_0 \mapsto \langle q_0 Z_0 p \rangle,$$

  para cada $p \in Q$.
- Si la función de transición δ satisface $\delta(p, a, A) = (q, \lambda)$ con $a \in \Sigma \cup \{\lambda\}$ y $A \in \Gamma \cup \{Z_0\}$, escribiremos la producción:

$$\langle pAq \rangle \mapsto a.$$

- Si la función de transición δ satisface $\delta(p, a, A) = (q, B_1 \cdots B_n)$ con $a \in \Sigma \cup \{\lambda\}$ y $B_1, \ldots, B_n \in \Gamma \cup \{Z_0\}$, escribiremos las producciones siguientes:

$$\langle pAq \rangle \mapsto \langle pB_n s_1 \rangle \langle s_1 B_{n-1} s_2 \rangle \langle s_2 B_{n-2} s_3 \rangle \cdots \langle s_{n-1} B_1 q \rangle a,$$

  para todos los estados $(s_1, \ldots, s_{n-1}) \in Q^{n-1}$.

□

Observación 6.4.1. Nótese que la construcción de la gramática asociada a un autómata con pila introduce un número exponencial (en el número de estados) de producciones por lo que es poco aconsejable utilizar esa construcción. Nos conformaremos con saber de su existencia.
Pero, obsérvese también, hemos probado que se puede suponer que los autómatas con pila indeterministas posee un sólo estado y que, en el contexto de un sólo estado, el paso de autómatas a gramáticas se puede realizar en tiempo polinomial.

[7] Nótese que para las producciones $A \mapsto \lambda$ borramos A.
[8] Bien podríamos haber usado el convenio $Q_0 := \langle q_0 \rangle$, pero lo dejamos por comodidad como si fuera una nueva variable.

6.5. Algunas Propiedades de la clase de lenguajes libres de contexto

Definición 6.5.1. Llamamos lenguajes libres de contexto a los lenguajes generados por una gramática incontextual (o, equivalentemente, los reconocidos por un autómata con pila indeterminista mediante pila y cinta vacías).

En esta Sección nos ocuparemos de enunciar unas pocas propiedades de la clase de lenguajes libres de contexto.

Teorema 6.5.1 (Intersección con Lenguajes Regulares)**.** *La clase de lenguajes libres de contexto está cerrada mediante intersección con lenguajes regulares. Es decir, si $L \subseteq \Sigma^*$ es un lenguaje libre de contexto y si $M \subseteq \Sigma^*$ es un lenguaje regular, entonces, $L \cap M$ es un lenguaje libre de contexto.*

Observación 6.5.1. Veremos que la intersección de dos lenguajes libres de contexto puede no ser un lenguaje libre de contexto. Para ello, consideremos los dos lenguajes siguientes:

$$L := a^* \cdot \{b^n c^n \ : \ n \in \mathbb{N}\} \subseteq \{a, b, c\}^*.$$

$$M := \{a^n b^n \ : \ n \in \mathbb{N}\} \cdot c^* \subseteq \{a, b, c\}^*.$$

La intersección es el lenguaje:

$$L \cap M := \{a^n b^n c^n \ : \ n \in \mathbb{N}\} \subseteq \{a, b, c\}^*.$$

Veremos más adelante que $L \cap M$ no es un lenguaje libre de contexto.

Definición 6.5.2 (Morfismo de monoides)**.** Dados dos monoides $(M, *)$ y $(N, \perp)$ llamaremos morfismo de monoides a toda aplicación $f : M \longrightarrow N$ que verifica las propiedades siguientes:

i) $f(\lambda_M) = \lambda_N$, donde λ_M y λ_N son los respectivos elementos neutros de los monoides M y N.

ii) $f(x * y) = f(x) \perp f(y)$ para todo $x, y \in M$.

Teorema 6.5.2 (Imágenes inversas por morfismos)**.** *La clase de lenguajes libres de contexto es cerrada por imágenes inversas por morfismos de monoides. Esto es, dados dos alfabetos Σ_1 y Σ_2 y dado un morfismo de monoides $f : \Sigma_1^* \longrightarrow \Sigma_2^*$, para cada lenguaje libre de contexto $L \subseteq \Sigma_2^*$, el siguiente también es un lenguaje libre de contexto:*

$$f^{-1}(L) := \{x \in \Sigma_1^* \ : \ f(x) \in L\}.$$

Teorema 6.5.3 (Complementario y Determinismo)**.** *La clase de los lenguajes aceptados por un autómata con pila determinista es cerrada por complementación. Es decir, si $L \subseteq \Sigma^*$ es un lenguaje aceptado por un autómata con pila determinista, su complementario $L^c := \Sigma^* \setminus L$ es también un lenguaje libre de contexto.*

Observación 6.5.2. Como ya habíamos señalado en la Observación 6.1.4 los lenguajes aceptados por autómatas determinísticos definen una clase particular `DCGL` dentro de la clase de lenguajes libres de contexto: *lenguajes libres de contexto "deterministas"*. Son los lenguajes generados por una gramática libre de contexto determinística y ejemplos tan simples como el palíndromo no admiten autómtas con pila determinísticos que los decidan. No entremos en esa discusión hasta más adelante.

El siguiente es un importante resultado de caracterización de lenguajes libres de contexto debido a W.F. Ogden[9].

[9] W.F. Ogden. "A Helpful Result for Proving Inherent Ambiguity". *Mathematical Systems Theory* **2** (1968) 191–194.

Definición 6.5.3 (Marcador de una palabra)**.** Llamamos marcador de una palabra $x \in \Sigma^*$ a una lista $\varepsilon := (\varepsilon_1, \ldots, \varepsilon_n) \in \{0,1\}^n$, donde $n = |x|$.

Nótese que un marcador consiste en señalar ciertos símbolos de una palabra y no otros. Obviamente, el número de marcadores de una palabra x es igual a $2^{|x|}$. Una manera de interpretar un marcador es el de subrayar algunos símbolos de la palabra y no otros, conforme a la regla obvia: subraya el símbolos $i-$ésimo si $\varepsilon_i = 1$ y no lo subrayes en el caso contrario. A modo de ejemplo, tomemos la palabra $x = abbbbabaa$ de longitud 9 y elijamos dos marcados $\varepsilon = (0,1,1,0,0,0,1,0,1)$ y $\varepsilon' = (1,0,0,0,1,0,0,0,1)$. Estos dos marcadores señalan símbolos de la palabra conforma a las siguientes reglas:

$$marcado(x, \varepsilon) := a\underline{bb}bba\underline{b}a\underline{a}.$$
$$marcado(x, \varepsilon') := \underline{a}bbb\underline{b}aba\underline{a}.$$

Llamamos número de posiciones distinguidas de un marcador al número de 1's. Así, el número de posiciones disntinguidas de ε es 4 y el de ε' es 3.

Teorema 6.5.4 (Lema de Ogden)**.** *Sea $G := (V, \Sigma, Q_0, P)$ una gramática libre de contexto. Existe un número natural $N \geq 1$ tal que para toda palabra $z \in L(G)$ y para todo marcador de z con un número de posiciones distinguidas mayor o igual a N, existe una factorización:*

$$z = uvwxy,$$

verificando:

i) La subpalabra w contiene, al menos, una posición distinguida.

ii) Las subpalabras v y x contienen, al menos, una posición disntinguida entre las dos.

iii) La subpalabra vwx tiene, a lo sumo, N posiciones distinguidas.

iv) Existe un símbolo no terminal $A \in V$ tal que se verifica $Q_0 \vdash_G uAy, A \vdash_G vAx$ $A \vdash_G w$. En particular, tenemos una propiedad de bombeo, puesto que para todo $i \geq 0$, $uv^iwx^*iy \in L(G)$.

Aunque el enunciado tiene un aspecto complejo, es un instrumento más que útil para mostrar lemguajes que no son libres de contexto. Una de las conclusiones de este Lema de Ogden es el siguiente resultado conocido como Lema de Bar–Hillel[10] aunque es debido a Bar–Hillel, Perles y Shamir.

Corollario 6.5.5 (Lema de Bar–Hillel)**.** *Si L es un lenguaje que satisface la siguiente propiedad:*

Para cada número natural N, $N \geq 1$, existe una palabra $z \in L$ en el lenguaje de longitud mayor que N verificando la siguiente propiedad:
Para cualesquiera palabras $\forall u, v, w, x, y \in \Sigma^*$, verificando

$$[z = uvwxy, |vwx| \leq N, |w| \geq 1, |vx| \geq 1] \Rightarrow \exists i \geq 0,\ uv^iwx^iy \notin L.$$

Entonces L no es un lenguaje libre de contexto.

Demostración. La prueba es obvia a partir del Lema de Ogden. Nótese que las condiciones de longitud pueden reescribirse en términos de marcadores. □

Ejemplo 6.5.1. El lenguaje $\{a^nb^nc^n \ : \ n \in \mathbb{N}\}$ no es un lenguaje libre de contexto. Nótese que para cada $N \in \mathbb{N}$ la palabra $a^Nb^Nc^N$ verifica que para toda factorización

$$a^Nb^Nc^N = uvwxy,$$

con las propiedades prescritas: $|vwx| \leq N, |w| \geq 1, |vx| \geq 1$ significa que en vwx no pueden estar más de dos símbolos (o bien $\{a,b\}$ o bien $\{b,c\}$). Por tanto, bombeando uv^iwx^iy en algún momento desequilibramos el número de símbolos. Así, por ejemplo, si vwx sólo contiene símbolos en $\{a,b\}$, bombeando uv^iwx^iy aumentamos el número de a's y de b's, pero no aumentamos el número de c's, con lo que, en algún momento, $uv^iwx^iy \notin L$.

[10] Y. Bar–Hillel, M. Perles, E. Shamir. "Onformal properties of simple phase–structure grammars".*Zeitschrift für Phonetik, Sparchwissenschaft und Kommunikationsforschung* **14**(1961) 143–172.

6.6. Problemas

6.6.1. Problemas

Problema 6.6.1. Sea A un Autómata con pila con lenguage por pila y lista vacías $L = L_{\emptyset}(A)$, y suponed que la palabra vacía no está en el lenguage. Describid cómo modificar el autómata para que accepte también la palabra vacía mediante pila y lista vacías.

Problema 6.6.2. Hallar Autómatas con Pila asociados a todas y cada una de las gramáticas libres de contexto descritas en los Problemas del Capítulo anterior.

Problema 6.6.3. Hallar un autómata con pila para el lenguaje

$$L := \{0^n 1^n \ : \ n \in \mathbb{N}\}.$$

Problema 6.6.4. Hallar un autómata con pila para el palíndromo.

Problema 6.6.5. Construir autómatas con pila que acepten los complementarios (en $\{a, b\}^*$) de los siguiente lenguajes:

i) $\{a^n b^n c^n \ : \ n \in \mathbb{N}\}$.

ii) $\{\omega\omega^R \ : \ \omega \in \{a, b\}\}$.

iii) $\{a^m b^n a^m b^n \ : \ m, n \geq 1\}$.

Problema 6.6.6. [11] Probar que el conjunto de palabras que pueden aparecer en una pila de un autómata con pila es un lenguaje regular.

Problema 6.6.7. Describir una gramática sensible al contexto para el lenguaje (que no es libre de contexto) siguiente:

$$L := \{a^n b^n c^n \ : \ n \in \mathbb{N}\} \subseteq \{a, b, c\}^*.$$

Problema 6.6.8. Probar que si un autómata con pila verifica que existe una constante $k \in \mathbb{N}$ tal que la pila nunca contiene palabras de longitud mayor que k, entonces, el lenguaje aceptado por ese autómata es un lenguaje regular.

Problema 6.6.9. Diseñar un automata con pila para cada uno de los siguientes lenguajes

i)

$$\{a^i b^j c^k | i \neq j or j \neq k\}$$

ii) El conjunto de todos las cadenas de símbolos que no son de la forma ww, esto es que no son igual a ningún "string" repetido.

Problema 6.6.10. Sea el siguiente PDA definido por

$$P = (\{q_0, q_1, q_2, q_3, f), \{a, b\}, \{Z_0, A, B\}, \delta, q_0, Z_0, \{f\})$$

Donde δ está dada por las siguientes reglas:

$$\begin{array}{lll} \delta(q_0, a, Z_0) = (q_1, AAZ_0) & \delta(q_0, b, Z_0) = (q_2, BZ_0) & \delta(q_0, \lambda, Z_0) = (f, \lambda) \\ \delta(q_1, a, A) = (q_1, AAA) & \delta(q_1, b, A) = (q_1, \lambda) & \delta(q_1, \lambda, Z_0) = (q_0, Z_0) \\ \delta(q_2, a, B) = (q_3, \lambda) & \delta(q_2, b, B) = (q_2, BB) & \delta(q_2, \lambda, Z_0) = (q_0, Z_0) \\ \delta(q_3, \lambda, B) = (q_3, \lambda) & \delta(q_3, \lambda, Z_0) = (q_1, AZ_1) & \end{array}$$

i) Demostrar que la cadena de caracteres *bab* está en el lenguaje.

ii) Demotrar que la cadena *abb* está en el lenguaje.

[11] Problema Difícil.

iii) Dar los contenidos de la pila después de que el autómata ha leído b^7a^4.

iv) Describir el lenguaje generado.

Problema 6.6.11. Un autómata con pila se llama restringido si en cada transición la pila solamente aumenta en, a lo sumo, un síimbolo. Demostrar que todo lenguaje dado por un autómata se puede definir mediante autómatas restringidos.

Problema 6.6.12. Convertir la gramática

$$Q_0 \mapsto 0Q_01 \mid A, A \mapsto 0A1 \mid Q_0 \mid \lambda,$$

a un autómata con pila que acepte el lenguaje por pila vacía o por estados finales aceptadores.

Problema 6.6.13. Convertir la gramática con producciones

$$Q_0 \mapsto aAA, A \mapsto aQ_0 \mid bQ_0 \mid a,$$

a un autómata con pila que acepte el lenguaje por pila vacía o por estado finales.

Problema 6.6.14. Diseñar una gramática y un autómata con pila que acepte el siguiente lenguaje:

$$\{0^n1^m \mid n \leq m \leq 2m.\}$$

Problema 6.6.15. Consideremos la siguiente gramática libre de contexto. :

$$\begin{aligned}
< q_0 > &\mapsto while\ < EXPRESION >\ \{< PROCEDIMIENTO >\} \\
< q_0 > &\mapsto if\ < EXPRESION >\ then\ \{< PROCEDIMIENTO >\} \\
< PROCEDIMIENTO > &\mapsto < q_0 > \\
< PROCEDIMIENTO > &\mapsto t++; cout << endl << t << endl; \\
< EXPRESION > &\mapsto < NUMERO >\ less\ than\ < VARIABLE > \\
< EXPRESION > &\mapsto < VARIABLE >\ less\ than\ < NUMERO > \\
< EXPRESION > &\mapsto < NUMERO >\ less\ than\ < FORMULA > \\
< FORMULA > &\mapsto < FORMULA > + < VARIABLE > \\
< FORMULA > &\mapsto (< FORMULA >) \\
< NUMERO > &\mapsto 1 < NUMERO > \\
< NUMERO > &\mapsto 0 \\
< VARIABLE > &\mapsto t \\
< VARIABLE > &\mapsto < FORMULA > +t \\
< VARIABLE > &\mapsto < FORMULA > +1
\end{aligned}$$

Denotaremos a las variables utilizando $< x >$, donde la variable de inicio sera denotada con $< q_0 >$. El alfabeto esta formado por los siguientes símbolos:

$$\Sigma := \{while, if, \{, \}, then, t, +, ;, cout, endl, <<, less\ than, 1, 0, (,)\}.$$

Se pide lo siguiente:

i) Eliminar las producciones unarias de la gramática.

ii) Eliminar los símbolos inútiles de la gramática.

iii) Construir la tabla de analísis sintáctico de la gramática.

iv) Obtener la siguiente palabra mediante derivaciones a la izquierda:

$$while\ t\ less\ than\ 110\ \{if\ t\ less\ than\ 10\ then\ \{t++; cout << endl << t << endl;\}\}.$$

Problema 6.6.16. Una gramática se dice ambigua si se puede derivar la misma palabra mediante dos derivaciones diferentes a la izquierda.

Demostrar que la siguiente gramática es ambigua:

$$
\begin{aligned}
< q_0 > &\mapsto if\ < EXPRESION >\ then\ < PROCEDIMIENTO > \\
< Q_0 > &\mapsto if\ < EXPRESION >\ then\ < PROCEDIMIENTO > \\
&\quad else < PROCEDIMIENTO > \\
< PROCEDIMIENTO > &\mapsto if\ < EXPRESION >\ then\ < PROCEDIMIENTO > \\
< PROCEDIMIENTO > &\mapsto if\ < EXPRESION >\ then\ < PROCEDIMIENTO > \\
&\quad else < PROCEDIMIENTO > \\
< PROCEDIMIENTO > &\mapsto System.out.println(1); \\
< PROCEDIMIENTO > &\mapsto System.out.println(0); \\
< EXPRESION > &\mapsto < NUMERO >\ less\ than\ < VARIABLE > \\
< EXPRESION > &\mapsto < VARIABLE >\ less\ than\ < NUMERO > \\
< NUMERO > &\mapsto 1 < NUMERO > \\
< NUMERO > &\mapsto 0 \\
< VARIABLE > &\mapsto t
\end{aligned}
$$

Demostrar que esta gramática genera el mismo lenguaje:

$$
\begin{aligned}
< Q_0 > &\mapsto if\ < EXPRESION >\ then\ < PROCEDIMIENTO > \\
< Q_0 > &\mapsto if\ < EXPRESION >\ then\ < PROCED-ELSE > \\
&\quad else < PROCEDIMIENTO > \\
< PROCEDIMIENTO > &\mapsto if\ < EXPRESION >\ then\ < PROCEDIMIENTO > \\
< PROCED-ELSE > &\mapsto if\ < EXPRESION >\ then\ < PROCED-ELSE > \\
&\quad else < PROCED-ELSE > \\
< PROCEDIMIENTO > &\mapsto System.out.println(1); \\
< PROCEDIMIENTO > &\mapsto System.out.println(0); \\
< PROCED-ELSE > &\mapsto System.out.println(1); \\
< PROCED-ELSE > &\mapsto System.out.println(0); \\
< EXPRESION > &\mapsto < NUMERO >\ less\ than\ < VARIABLE > \\
< EXPRESION > &\mapsto < VARIABLE >\ less\ than\ < NUMERO > \\
< NUMERO > &\mapsto 1 < NUMERO > \\
< NUMERO > &\mapsto 0 \\
< VARIABLE > &\mapsto t
\end{aligned}
$$

y ademas demostrar que no es ambigua.

Problema 6.6.17. Construir una derivacion a la izquierda de la siguiente palabra $10+11*1111$

utilizando la siguiente gramática:

$$\begin{aligned}
< Q_0 > &\mapsto < EXPRESION > * < EXPRESION > \\
< Q_0 > &\mapsto < EXPRESION > + < EXPRESION > \\
< EXPRESION > &\mapsto < NUMERO > + < EXPRESION > \\
< EXPRESION > &\mapsto < NUMERO > - < EXPRESION > \\
< NUMERO > &\mapsto 1 < NUMERO > \\
< NUMERO > &\mapsto 0 < NUMERO > \\
< NUMERO > &\mapsto 0|1.
\end{aligned}$$

Decidir si la gramática es ambigua y que pasa si se elimina la segunda producción.

Problema 6.6.18. Supongamos que el siguiente lenguaje de programación dado por la siguiente gramática:

$$\begin{aligned}
< Q_0 > &\mapsto < DECLARACION >< Q_0 >< ORDEN > \\
< DECLARACION > &\mapsto < TIPO >< NOMBRE >=< VALOR >; \\
< TIPO > &\mapsto REAL|INTEGER \\
< NOMBRE > &\mapsto a|b \\
< ORDEN > &\mapsto < NOMBRE >=< ORDEN > + < ORDEN >; \\
< ORDEN > &\mapsto < VALOR > \\
< ORDEN > &\mapsto < NOMBRE > \\
< VALOR > &\mapsto 1 < VALOR > \\
< VALOR > &\mapsto 0
\end{aligned}$$

Hallar una derivación a la izquierda para esta palabra (los espacios y retornos de carro se eliminaran):

$$REAL\ a = 10; REAL\ b = 110; a = a + b; b = a + a + a;$$

Demostrar que el lenguaje asi definido admite como correcta la siguiente expresión:

$$REAL\ a = 10; a = a = a + a; +110s;$$

Comprobar que se acepta esta expresion y modificar la gramática para que solo se acepten expresiones correctas de sumas.

Capítulo 7

Una Sucinta Introducción a Parsing

Contents

7.1. Introducción 120
7.1.1. El problema de parsing: Enunciado 122
7.2. Compiladores, Traductores, Intérpretes 122
7.2.1. Traductores, Compiladores, Intérpretes 123
7.2.1.1. Compiladores Interpretados. 123
7.2.2. Las etapas esenciales de la compilación. 124
7.2.2.1. La Compilación y su entorno de la programación. 124
7.2.2.2. Etapas del Proceso de Compilación. 124
7.2.2.3. En lo que concierne a este Capítulo. 124
7.3. Conceptos de Análisis Sintáctico 124
7.3.1. El problema de la Ambigüedad en CFG 125
7.3.2. Estrategias para el Análisis Sintáctico. 126
7.4. Análisis CYK 128
7.4.1. La Tabla CYK y el Problema de Palabra. 128
7.4.2. El Árbol de Derivación con las tablas CYK. 130
7.4.3. El Algoritmo de Análisis Sintáctico CYK 130
7.5. Traductores Push–Down. 131
7.5.0.1. Sistema de Transición asociado a un PDT. 132
7.6. Gramáticas $LL(k)$: Análisis Sintáctico 133
7.6.1. FIRST & FOLLOW 133
7.6.2. Gramáticas $LL(k)$ 137
7.6.3. Tabla de Análisis Sintáctico para Gramáticas $LL(1)$ 139
7.6.4. Parsing Gramáticas $LL(1)$ 140
7.7. Cuestiones y Problemas 142
7.7.1. Cuestiones 142
7.7.2. Problemas 143

En el proceso de compilación, debemos considerar diversos pasos.
En un primer paso (Análisis Léxico) el compilador decide si todas las partes del fichero (trozos en los que está dividido) responden a los patrones de tipos de datos, identificadores o palabras reservadas (en forma de expresiones regulares) que hemos seleccionado. Para ello, aplica los diversos autómatas finitos asociados a las diversas expresiones regulares distinguidas.
Una vez realizado el análisis léxico (omitiendo la gestión de errores en este curso) se procede al proceso de traducción del código fuente a código objetivo. En este proceso utilizaremos un

modelo clásico basado en Sistemas de Traducción basados en Sintaxis Directa (SDTS). Con ellos, el proceso de traducción se reduce a un proceso de detección de pertenencia al lenguaje generado por la gramática que define nuestro lenguaje fuente (Problema de Palabra para gramáticas libres de contexto, con gramática fijada "a priori") y a un proceso de descripción de un árbol de derivación de la palabra dada (en el caso de ser aceptada). Los algoritmos que realizan esta tarea son los algoritmos de *parsing* (o Análisis Sintáctico) y la descripción de algunos de ellos es el objetivo de este Capítulo.

7.1. Introducción

El problema al que nos enfrentamos es el siguiente:

Problema (Problema de Traducción.)**.** *Fijados dos lenguajes de programación* $L_1 \subseteq \Sigma_1^*, L_2 \subseteq \Sigma_2^*$*, y dada una palabra* ω *realizar la tarea siguiente:*

- *Decidir si* $\omega \in L_1$*,*
- *En caso de respuesta afirmativa, traducir* ω *a una palabra* ω_2 *en el lenguaje de programación* L_2 *(i.e.* $\omega_2 \in L_2$*).*

La traducción en lenguajes naturales sigue siendo un proceso complejo e incompleto (*"traduttore, traditore"*). En lenguajes formales, si ambos son dados por gramáticas libres de contexto, esta tarea es factible y computable. De esto trata el presente Capítulo.
Sin embargo, el proceso de traducción no es evidente y acudiremos a sistemas de traducción basados en la existencia de una relación directa entre las sintaxis de ambos lenguajes (STDS's).

Definición 7.1.1 (Syntax–Directed Translation Scheme)**.** Un esquema de traducción sintáctica directa (SDTS) es un quíntuplo $(V, \Sigma, \Delta, Q_0, P)$ donde:

- V es un conjunto finito de símbolos llamados variables o símbolos no terminales.
- Σ es un alfabeto finito, llamado alfabeto de input.
- Δ es un segundo alfabeto finito llamado alfabeto de output.
- $Q_0 \in V$ es un símbolo distinguido llamado símbolo inicial del $SDTS$.
- $P \subseteq V \times (V \cup \Sigma)^* \times (V \cup \Delta)^*$ es un conjunto finito conocido como conjunto de pares de producciones.

Escribiremos las producciones mediante: $A \mapsto (\alpha, \beta)$ para denotar la terna $(A, \alpha, \beta) \in P$.
Un SDTS es un mecanismo de traducción basado en los componentes sintácticos de dos gramáticas.

Definición 7.1.2 (Input and Output Grammars)**.** Dado un SDTS $T := (V, \Sigma, \Delta, Q_0, P)$ disponemos de dos gramáticas libres de contexto asociadas:

- La gramática de input (o fuente) $T_{input} := (V, \Sigma, Q_0, P_{input})$ donde:

$$P_{input} := \{A \mapsto \alpha \ : \ \exists \beta \in (V \cup \Delta)^*, \ A \mapsto (\alpha, \beta) \in P\}.$$

- La gramática de output (u objetivo) $T_{output} := (V, \Delta, Q_0, P_{output})$ donde:

$$P_{output} := \{A \mapsto \beta \ : \ \exists \alpha \in (V \cup \Sigma)^*, \ A \mapsto (\alpha, \beta) \in P\}.$$

Nótese que ambas gramáticas son gramáticas libres de contexto. Puede observarse que un sistema de traducción por sintaxis directa $SDTS := (V, \Sigma, \Delta, Q_0, P)$ genera un proceso de traducción entre dos lenguajes. Una traducción es una aplicación:

$$\tau : L_1 \subseteq \Sigma^* \longmapsto L_2 \subseteq \Delta^*,$$

que transforma palabras (programas, ficheros, códigos) en el primer lenguaje en palabras (programas, ficheros, códigos) en el segundo lenguaje. La "traducción" asociada a nuestro sistema $SDTS$ anterior funciona del modo siguiente:

- El lenguaje del cual tomamos las entradas es L_1 y es el lenguaje generado por la gramática T_{input}, i.e. $L_1 := L(T_{input})$.
- El lenguaje al cual deben pertenecer las salidas es L_2 y es el lenguaje generado por la gramática T_{output}, i.e. $L_2 := L(T_{output})$.

La traducción funciona del modo siguiente: dado $\omega \in L_1$ sea

$$Q_0 \to \omega_1 \to \omega_2 \to \cdots \to \omega_n = \omega,$$

una cadena de computaciones o derivaciones en P_{input} (i.e. las producciones de la gramática T_{input}) . Por construcción, tendremos unas producciones en P que se corresponden con los pasos realizados.
Así, el paso de ω_i a ω_{i-1} se habrá realizado porque existen $\gamma, \rho \in \Sigma^*$ tales que $\omega_{i-1} = \gamma A_i \rho$, con $A_i \in V$ y existe una producción $A_i \mapsto \alpha_i$ en P_{input} tal que $\omega_i = \gamma\alpha_i\rho$.
Pero la producción en P_{input} debe proceder de una producción en P que ha de tener la forma:

$$A_i \mapsto (\alpha_i, \beta_i),$$

Ahora comienza a actuar el SDTS del modo siguiente:

- Comenzamos con $Q_0 \to \eta_1 = \beta_1$. Es decir, realizamos la producción emparejada a la producción de P_{input} usada inicialmente.
- Recursivamente, si hemos calculado $Q_0 \to \eta_1 \to \ldots \to \eta_i$, y si la variable A_i está en η_i, recodamos el par de producciones $A_i \mapsto (\alpha_i, \beta_i)$ en P. Si $\eta_i = \gamma_1 A_i \gamma_2$, entonces definimos $\eta_{i+1} = \gamma_1\beta_i\gamma_2$, en caso contrario devolvemos "error".

Ahora tendremos una cadena de derivaciones basadas en P_{output} de la forma:

$$Q_0 \to \eta_1 \to \eta_2 \to \cdots \to \eta_n = \omega'.$$

Las palabras $\eta_i \in \Delta^*$ se obtienen inductivamente aplicando las producciones $A_i \mapsto \beta_i$ de P_{output}.
Si tras el proceso anterior hallamos una forma terminal ω', tendremos que $\omega' \in L_2 = L(T_{output})$ y la podemos denotar mediante $\omega' = \tau(\omega)$ y la llamaremos traducción de ω mediante el SDTS P.
Nótese que ω' no es necesariamente única y ni siquiera tiene por qué existir. En primer lugar, porque no hemos indicado cuál de las posibles apariciones de la variable A_i es la que hay que reemplazar por β_i. También carece del rasgo determinista porque podría haber más de una producción en el SDTS cuya primera producción fuera $A_i \mapsto \alpha_i$. Finalmente podría incluso que no no hubiera ninguna variable A_i en η_{i-1} para reemplazar, con lo que $\eta_{i-1} = \eta_i$. La conclusión es que, en general, τ es correspondencia y no necesariamente aplicación.
Por todo ello, nos conformaremos con disponer de un SDTS de tipo simple.

Definición 7.1.3 (SDTS simple)**.** Un sistema de traducción por sintaxis directa $T := (V, \Sigma, \Delta, s_0, P)$ se llama simple, si para cada producción

$$A \mapsto (\alpha, \beta),$$

las variables que aparecen en α son las mismas, repetidas con la misma multiplicidad[1] que las que aparecen en β.

[1]Se podría incluso suponer que la variables aparecen en el mismo orden, éso conduciría a traducciones aún más inmediatas.

Por simplicidad de nuestro análisis podemos incluso suponer que el orden en que se presentan las variables en cada una de las producciones (orden de "lectura", de izqda. a derecha, de derecha a iazquierda o con otras ordenaciones) de T es el mismo.
En todo caso, una traducción basada en SDTS trata fielmente de seguir el proceso sintáctico de generación de ω en la primera gramática, para reproducirla en la segunda y así producir la traducción.
Nótese que el proceso de traducción natural consistirá en disponer "a priori" de dos gramáticas libres de contexto G_1 y G_2 y generar, a partir de ellas, un SDTS T tal que $L(G_1) = L(T_{input})$ y $L(G_2) = L(T_{output})$. Por simplicidad de la discusión admitiremos que disponemos de un $STDS$ a priori entre dos lenguajes de programación L_1 y L_2.
A partir de él nos planteamos el problema siguiente:

7.1.1. El problema de parsing: Enunciado

Fijado un lenguaje libre de contexto $L \subseteq \Sigma^*$, dada una palabra $\omega \in \Sigma^*$, entonces resolver:

- *Decidir si $\omega \in L$.*
- *En caso de respuesta afirmativa, dar un árbol de derivación (o una derivación) que produzca ω.*

El objetivo general de este Capítulo es resolver el problema de parsing siguiendo varias estrategias.
Obsérvese que disponiendo de un algoritmo que resuelva el problema de parsing para la gramática P_{input} y disponiendo de un SDTS, disponemos de un compilador (aunque sea indeterminista) que traduzca los "programas" de un lenguaje de programación al otro.
Antes de comenzar nuestro análisis del problema de parsing, dedicaremos un rato a recordar algunos términos informales sobre compiladores.

7.2. Palabras Técnicas de uso Común sobre Compiladores, Traductores, Intérpretes

Presentamos algunas ideas de las palabras de uso común en el mundo de los compiladores. Dejamos el formalismo para el curso próximo.

Definición 7.2.1 (Compilador)**.** Un compilador es un programa (o conjunto de programas) que transforma:

- un texto escrito en un lenguaje de programación (lenguaje fuente)
- en un texto escrito en otro lenguaje de programación (lenguaje objetivo)

El texto en lenguaje fuente se llama código fuente. La salida del compilador se llama código objetivo.

Notación 7.2.1 (Terminología Básica)**.** El uso más común de los términos suele ser:

- **Compilador:** suele usarse este término cuando el lenguaje objetivo es de nivel más bajo que el lenguaje fuente.
- **Fuente a Fuente:** transforma código entre lenguajes de alto nivel. Se suele usar el término reescritura cuando el lenguaje fuente es exactamente el lenguaje objetivo.
- **Decompilador:** el lenguaje objetivo es de mayor nivel que el lenguaje fuente. Su uso es harto infrecuente.

7.2.1. Traductores, Compiladores, Intérpretes

El resultado de un Traductor–Compilador es esencialmente un ejecutable. Sin embargo, presenta un inconveniente, sobre todo en el manejo de códigos fuente de gran tamaño:
El código debe ser primero compilado y sólo tras finalizar la compilación puede ser ejecutado.
La alternativa son los **Intérpretes**. Un Intérprete toma como entrada un lenguaje en código fuente y procede el modo siguiente con cada instrucción (o parte) del código fuente realiza las siguientes tareas:

- carga la instrucción,
- analiza la instrucción,
- ejecuta la parte del código fuente.

7.2.1.0.1. Ventajas del Intérprete. Tiene las siguientes ventajas:

- El programador trabaja en forma interactiva y va viendo los resultados de cada instrucción antes de pasar a la siguiente.
- El programa se utiliza sólo una vez y no importa tanto la velocidad (después se compila y se usa solamente el ejecutable).
- Se espera que cada instrucción se ejecute una sola vez.
- Las instrucciones tienen formas simples y son más fáciles de analizar.

7.2.1.0.2. Inconvenientes de los Intérpretes. Tienen los siguientes inconvenientes:

- La velocidad puede ser del orden de 100 veces más lento que la de un ejecutable.
- No sirve cuando se espera que las instrucciones se ejecuten frecuentemente.
- Tampoco es interesante cuando las instrucciones sean complicadas de analizar.

Ejemplo 7.2.1. Entre los intérpretes más habituales:

- shell El intérprete de comandos de Unix. Es una instrucción para el sistema operativo Unix. Se introduce dando el comando de forma textual. Actúa como se ha indicado: comando a comando. El usuario puede ver la acción de cada comando.
- LISP Es un lenguaje de programación de procesado de Listas. Common LISP.
- Un Intérprete de SQL. Ver curso previo de Bases de datos.

7.2.1.1. Compiladores Interpretados.

Combinan las cualidades de Compiladores e intérpretes.

- Transforma el código fuente en un lenguaje intermedio.
- Sus instrucciones tienen un formato simple y fácil de analizar.
- La traducción desde el lenguaje fuente al lenguaje intermedio es fácil y rápida.

Ejemplo 7.2.2 (Java)**.** Tiene las propiedades siguientes:

- El código JVM (Java Virtual Machine) es un lenguaje intermedio entre Java, de una parte, y Ensamblador y código máquina.
- La interpretación de JVM es más rápida que si hubiera un intérprete directo de Java.
- JVM es un traductor más un intérprete:
  - Traduce código Java a código JVM.
  - Interpreta código JVM.

7.2.2. Las etapas esenciales de la compilación.

7.2.2.1. La Compilación y su entorno de la programación.

Primero ubicaremos el proceso de compilación dentro del proceso completo:

- Recibimos el código fuente y realizamos un *pre–procesamiento* para eliminar extensiones.
- El Compilador traduce el código fuente (limpio de extensiones) en código objetivo (en lenguaje ensamblador).
- El ensamblador asociado a cada máquina concreta transforma (e interpreta) cada ejecutable en ensamblador en código máquina (el código traducido a la máquina concreta en la que será ejecutado).

7.2.2.2. Etapas del Proceso de Compilación.

Las etapas fundamentales de la compilación son las siguientes:

i) **Análisis Léxico:** Utilizando expresiones regulares y/o autómatas finitos, se verifica si el código fuente sigue las reglas básicas de la gramática (regular) en este caso que define el lenguaje. A modo de ejemplo, definimos los tipos admisibles de datos (pongamos `real`) mediante una expresión regular, después usamos el autómata correspondiente para reconocer una entrada del lenguaje fuente que pertenece al lenguaje definido por esa expresión regular. En caso de encontrar un error, devuelve un mensaje de error.

ii) **Análisis Sintáctico.** De nuevo usaremos el Problema de Palabra como referente y como sustrato las gramáticas libres de contexto y los autómatas con pila del Capítulo anterior.

iii) **Análisis Semántico.** Revisiones para asegurar que los componentes de un programa se ajustan desde el plano semántico.

iv) **Generador de Código Intermedio:** Un código intermedio que sea fácil de producir y fácil de traducir a código objetivo.

v) **Optimización del Código.** Trata de mejorar el código intermedio[2].

vi) **Generador de Código Final:** Etapa final de la compilación. Genera un código en ensamblador.

7.2.2.3. En lo que concierne a este Capítulo.

En lo que concierne a este curso nos interesaremos solamente por las etapas de **Análisis Léxico** y **Análisis Sintáctico (o parsing)**.
Como los analizadores léxicos contienen una menor dificultad, nos centraremos en el Análisis Sintáctico o parsing.

7.3. Conceptos de Análisis Sintáctico

El problema central del Análisis Sintáctico es el

Problema 7.3.1 (Parsing o Análisis Sintáctico)**.** Dada una forma terminal $\omega \in \Sigma^*$ sobre una gramática $G := (V, \Sigma, Q_0, P)$, se pide producir:

- Un árbol sintáctico con el que continuará la siguiente etapa de la compilación, si la palabra $\omega \in L(G)$.

[2]La generación de código óptimo es un problema **NP**–completo.

- Un informe con la lista de errores detectados, si la cadena contiene errores sintácticos. Este informe deberá ser lo más claro y exacto posible.

Omitiremos las distintas estrategias de Manejo de Errores, para concentrarnos en los otros dos aspectos.

7.3.1. El problema de la Ambigüedad en CFG

Definición 7.3.1 (Derivaciones Leftmost y Rightmost)**.** Sea $G = (V, \Sigma, Q_0, P)$ una gramática libre de contexto. Sean $c, c' \in (V \cup \Sigma)^*$ dos formas sentenciales.

i) Diremos que c' se obtiene mediante derivación "más a la izquierda" (o leftmost) de c, si existen $\omega \in \Sigma^*, A \in V, \alpha \in (V \cup \Sigma)^*$, y existe una producción $A \mapsto \beta$, con $\beta \in (V \cup \Sigma)^*$ tales que
$$c = \omega A \alpha, \;\; c' = \omega \beta \alpha.$$
Denotaremos mediante $c \to^G_{lm} c'$.

ii) Diremos que c' se obtiene mediante derivación "más a la derecha" (o rightmost) de c, si existen $\omega \in (V \cup \Sigma)^*, A \in V, \alpha \in \Sigma^*$, y existe una producción $A \mapsto \beta$, con $\beta \in (V \cup \Sigma)^*$ tales que
$$c = \omega A \alpha, \;\; c' = \omega \beta \alpha.$$
Denotaremos mediante $c \to^G_{rm} c'$.

Usualmente, y si no hay confusión, omitiremos el super-índice ^{G}.

Nótese que hay dos elementos indeterministas en las traducciones así planteadas. De una parte tenemos la elección de sobre qué variable actuamos, de otro lado qué pareja de producciones elegimos. La idea de introducir derivaciones "más a la izquierda" o "más a la derecha" consiste en tratar de reducir el ingrediente indeterminístico a la hora de seleccionar qué variable es la variable sobre la que vamos a actuar con nuestras producciones. En el proceso leftmost, seleccionamos la variable que se encuentra más a la izquierda y actuamos sobre ella mediante alguna de las producciones aplicables. En el rightmost hacemos lo propio con la variable que se encuentra más a la derecha. Nótese que esto no significa que el proceso de derivación sea determinístico: aunque seleccionemos sobre cuál de las variables actuamos primero, es claro que puede haber más de una producción que actúa sobre esa variable y mantiene sus rasgos indeterministas.

Definición 7.3.2 (Cadenas de Derivaciones Leftmost y Rightmost)**.** Con las mismas notaciones de la Definición anterior,

i) Diremos que c' es deducible de c mediante derivaciones más a la izquierda (y lo denotaremos mediante $c \vdash^G_{lm} c'$) si existe una cadena finita de derivaciones más a la izquierda que va de c a c'. Esto es, si existen:
$$c = c_0 \to^G_{lm} c_1 \to^G_{lm} \cdots c_{k-1} \to^G_{lm} c_k = c'.$$

ii) Diremos que c' es deducible de c mediante derivaciones más a la derecha (y lo denotaremos mediante $c \vdash^G_{rm} c'$) si existe una cadena finita de derivaciones más a la derecha que va de c a c'. Esto es, si existen:
$$c = c_0 \to^G_{rm} c_1 \to^G_{rm} \cdots c_{k-1} \to^G_{rm} c_k = c'.$$

De nuevo, omitiremos el super-índice ^{G} siempre que su omisión no induzca confusión alguna.

La idea es una cadena de pasos más a la izquierda, esto es, elegimos la variable más a la izquierda (leyendo de izquierda a derecha) de c. Aplicamos una producción de la gramática sobre esa variable y c' es simplemente el resultado de aplicar esa producción.

Ejemplo 7.3.1. Tomemos la gramática cuyas producciones son:

$$P := \{Q_0 \mapsto AB \mid CA \mid AQ_0 \mid 0, A \mapsto BA \mid 0A0 \mid 1, B \mapsto Q_0A, C \mapsto 1\}.$$

Una cadena de derivaciones leftmost (más la izquierda) sería la siguiente:

$$Q_0 \to AB \to CAB \to 1AB \to 11B \to 11Q_0A \to 110A \to 1101.$$

Una cadena de derivaciones rightmost (más a la derecha) sería la siguiente:

$$Q_0 \to AB \to AQ_0A \to AQ_01 \to A01 \to 0A001 \to 01001.$$

Definición 7.3.3 (Gramáticas Ambiguas)**.** Una gramática se dice ambigua si existe una forma sentencial $\omega \in (V \cup \Sigma)^*$ alcanzable desde el símbolo inicial (i.e $Q_0 \vdash_G \omega$) tal que existen al menos dos computaciones (derivaciones) más a la izquierda (o más a la derecha) distintas que permiten generar ω.

Ejemplo 7.3.2. Tomemos la gramática $P := \{E \mapsto E + E \mid E * E \mid a\}$. Ahora disponemos de dos cadenas de derivación para $a + a * a$ distintas:

$$E \to_{lm} E + E \to_{lm} a + E \to_{lm} a + E * E \to_{lm} a + a * E \to_{lm} a + a * a.$$

Y también

$$E \to_{lm} E * E \to_{lm} E + E * E \to_{lm} a + E * E \to_{lm} a + a * E \to_{lm} a + a * a.$$

Por lo que la anterior gramática es ambigua.

El problema con la ambigüedad de las gramáticas se enfrenta a la siguiente dificultad:

Teorema 7.3.1. *Decidir si una gramática libre de contexto es ambigua es indecidible (i.e. no existe algoritmo que permita decidir la cualidad de ser ambigua).*

La razón es la Indecidibilidad del Post Correspondence Problem. En algunos casos se pueden dar estrategias que eliminen la condición de ambigüedad de algunas gramáticas, pero el anterior Teorema nos garantiza que ésto no es posible en general.
En todo caso, los lenguajes de programación se diseñan generados mediante gramáticas no ambiguas.

7.3.2. Estrategias para el Análisis Sintáctico.

En este Capítulo nos dedicaremos a mostrar algunos ejemplos de análisis sintáctico, basados en distintos ejemplos de estrategias.

i) Estrategias de carácter general aplicables a cualquier gramática libre de contexto, incluyendo el caso indeterminista (en el Autómata con Pila) pero realizando un análisis determinístico. Hemos seleccionado la estrategia de `Parsing CYK` de Cocke[3], Younger[4] y Kasami[5]. Otros procesos generales, como la estrategia de Earley[6] , pueden seguirse en [AhUl, 72a]. Este tipo de análisis generalista, que admite lenguajes libres de contexto indeterminísticos, tendrá un coste en complejidad superior a los otros. De hecho, la complejidad

[3]Hays se lo atribuye a Cocke en D.G. Hays. *Introduction to Computational Linguistics*. Elsevier, New York, 1967. Véase también [Cocke-Schwartz, 70].

[4]D.H. Cocke. " Recognition and parsing of context–free languages in time n^3. *Information and Control* **10** (1967) 189–208.

[5]Véase el informe técnico de 1965, AFCRL–65–758, escrito por Kasami y el paper T. Kasami, K. Torii."A syntax analysis procedure for unambiguous context–free grammars". *J. of the ACM* **16** (1969) 423-431.

[6]J. Earley. *An Efficient context–free parsing algorithm*, Ph.D. Thesis, carnegie–Mellon, 1968.Y el trabajo J. Earley. "An Efficient context–free parsing algorithm". *Commun. ACM* **13** (1970) 94–102.

del algoritmo que presentamos es $O(n^3)$. Entre los algoritmos de Análisis Sintáctico generalista, el más eficiente es la variante de CYK introducida por Leslie G. Valiant[7] quien, en 1975, adapta estrategias de multiplicación rápida de matrices a la estrategia CYK para obtener un algoritmo de parsing generalista en tiempo $O(n^{2,38})$. Dado que estas estrategias algorítmicas se escapan a los contenidos usuales del curso las omitiremos,pero recomendamos al lector acudir a las fuentes de autores cono V. Strassen, Sh. Winograd, V. Pan o, más recientemente, A. Storjohann o G. Villard y sus referencias.

ii) **Análisis Descendente (Top–Down):LL(k).** Es un modelo de análisis sintáctico adaptado a *lenguajes libres de contexto determinísticos* (es decir, que son aceptados por un autómata con pila determinístico). Por tanto, no son analizadores generaliastas como los anteriores y no sirven para tratar todos los lenguajes libres de contexto. El modelo de análisis recorre un árbol de derivación desde la raíz (con el símbolo inicial Q_0, hasta las hojas. Basa su construcción en el uso de derivaciones "más a la izquierda". El proceso resulta determinista en el caso de que la gramática involucrada sea de un tipo específico: gramáticas $LL(k)$ (de " from left to right with left canonical derivation and a look-ahead of k symbols deterministic recognizable"). Haremos la exposición para gramáticas $LL(1)$. Estas estretagias más intuitivas (para un Europeo) de parsing fueron introducidas por N. Wirth en [Wi, 96] y adaptadas a lenguajes como `Pascal`. Un ejemplo de generador de parsers cuya ideologí se apoya fuertemente en lo que podemos llamar *LL–ismo* es ANTLR[8]. Hasta mediados de los 90, los analizadores para gramáticas en $LL(k)$ fueron descartados y considerados como impracticables porque tenían una complejidad exponencial en k (de hecho, la tabla de parsing $LL(k)$ crece de manera exponencial en k). La aparición de ANTLR (y su antecesor PCCTS) en 1992, revitalizó este tipo de analizadores, mostrando que la complejidad del caso peor es raramente alcanzada en la práctica.

iii) **Análisis Ascendente (Bottom–Up):LR(k).** De nuevo es un modelo de análisis sintáctico válido solamente para *lenguajes libres de contexto deterministas*. Fueron introducidos por D.E. Knuth en [**?**]. Knuth demostraba que todos los lenguajes libres de contexto deterministas admiten algún analizador sintáctico $LR(k)$ y que todo lenguaje libre de contexto determinista admite una gramática $LR(1)$. También demostraba que la complejidad del análisis $LR(k)$ era lineal en el tamao de la entrada. El proceso recorre un árbol de derivación desde las hojas hasta la raíz (Bottom–Up). Basa su análisis en derivaciones más a la derecha (right-most). Por ello el output de este tipo de parsers es el reverso de un árbol de derivación de una palabra aceptada. El término LR viene de " Left-to-right scan, Rightmost derivation. Durante años ha sido el analizador sintáctico preferido por los programadores. Mencionaremos como ejemplos:

- `Yacc`: *Yet Another Compiler Compiler* desarrollado por los laboratorios AT & T. Genera código en lenguaje C y es mantenido actualmente por la compañía SUN
- `Bison`: Es el compilador de compiladores del proyecto GNU. Genera código en C++.
- `SableCC`: Creado por Étienne Gagnon, este programa contiene los últimos avances en materia de compiladores.

En la actualidad los analizadores dominantes son los basados en estretagias `LALR` que combinan la estrategia look ahead del diseño descedente, con la eficiencia de los analizadores LR.

A modo de comentario adicional, el análisis descendente es más intuitivo y permite gestionar mejor los errores en el código fuente. Cuando oímos trozos de una frase, estamos analizando smiultáneamente (esta es la estretgia *look ahead*. No necesitamos oír la frase entera para ir deduciendo que quién será el sujeto o quién el predicado. Es más, notamos, sin tener que acabar la frase podemos deducir que no es una frase correcta en español.

[7] L.G. Valiant. " General Context–Free Recognition in less than cubic time". *J. of Comput. and Syst. Science* **10** (1975) 308–314.

[8] Consultar en `http://www.antlr.org/`, por ejemplo

7.4. Análisis CYK

Es el modelo de análisis debido a Cocke, Younger y Kasami. Es un modelo aplicable a cualquier gramática en forma normal de Chomsky (CNF) y $\lambda-$libre.

7.4.1. La Tabla CYK y el Problema de Palabra.

El input del algoritmo está formado por una gramática libre de contexto $G = (V, \Sigma, q_0, P)$, y por una forma terminal, es decir, $\omega := a_1 a_2 \cdots a_{n-1} a_n \in \Sigma^*$.
El output del algoritmo es una tabla triangular. Para interpretarlo, podemos entender la tabla de output como una aplicación:

$$t : \{(i,j) \ : 1 \leq i \leq n, 1 \leq j \leq n-i+1\} \longrightarrow \mathcal{P}(V),$$

donde V son los símbolos no terminales de la gramática y $\mathcal{P}(V)$ son los subconjuntos de V. A la imagen $t(i,j) \in \mathcal{P}(V)$ la denotaremos con sub-índices. Es decir, escribiremos $t_{i,j} \in \mathcal{P}(V)$.
En el conjunto $t_{i,j}$ escribiremos todas las variables $A \in V$ tales que

$$A \vdash^G a_i a_{i+1} \cdots a_{i+j-1}.$$

Nótese que, en realidad, t depende de G y de ω.

$t_{i,j}$	1	2	3	4	5
1	A	Q_0	A	A	Q_0
2	Q_0	A	Q_0	A	
3	A	Q_0	Q_0		
4	A	Q_0,A			
5	Q_0				

Cuadro 7.1: Los diferentes $t_{i,j}$ donde i es el número de fila y j es la columna

---

INPUT: Dos elementos:

- Una gramática $G := (V, \Sigma, Q_0, P)$ libre de contexto, en forma normal de Chomsky y $\lambda-$libre.
- Una palabra $\omega = a_1 a_2 \cdots a_n \in \Sigma^*$ de longitud n.

OUTPUT: La tabla

$$t : \{(i,j) \ : 1 \leq i \leq n, 1 \leq j \leq n-i+1\} \longrightarrow \mathcal{P}(V),$$

según las propiedades anteriores.
Inicializar: Hallar para cada i, $1 \leq i \leq n$ los conjuntos siguientes:

$$t_{i,1} := \{A \ : \ A \mapsto a_i \in P\}.$$

$j := 1$
while $j \leq n$ **do**
  for $i = 1$ **to** $n - j + 1$ **do**
    $t_{i,j} := \{A \ : \ \exists k,\ 1 \leq k < j,\ B \in t_{i,k},\ C \in t_{i+k,j-k},\ \text{ and } A \mapsto BC \in P\}$.
    next i
  od
  next j
od
end

Teorema 7.4.1. *El algoritmo calcula la Tabla t del análisis CYK. El tiempo de ejecución del algoritmo es $O(n^3)$ y la memoria usada es $O(n^2)$, donde n* [9].

Demostración. Es una mera comprobación. □

Ejemplo 7.4.1. Hallar la tabla para la gramática G cuyas producciones son:

$$Q_0 \mapsto AA \mid AQ_0 \mid b,\ A \mapsto Q_0 A \mid AQ_0 \mid a.$$

y la palabra $\omega = abaab$. La solución se puede encontrar en la tabla 7.1

Teorema 7.4.2. *Con las notaciones anteriores, $\omega \in L(G)$ si y solamente si $Q_0 \in t_{1,n}$. En particular, el cálculo de la tabla por el método CYK resuelve el problema de palabra para gramáticas en forma normal de Chomsky y $\lambda-$libres en tiempo $O(n^3)$ y espacio $O(n^2)$.*

Demostración. Por definición de la propia tabla, $Q_0 \in t_{1,n}$ si y solamente si $Q_0 \vdash \omega$. □

[9] Aunque se debe tener en cuenta que, en el tamaño de la tabla también juega su papel $\sharp(V)$, conb lo que la complejidad depende también del tamaño de la gramática. Sin embargo, la gramática permanece "fija" en el proceso de análisis sintáctico correspondiente, con lo que ese tamaño de la gramática se "oculta" dentro de la constante de $O()$.

7.4.2. El Árbol de Derivación con las tablas CYK.

A partir de la construcción de la tabla CYK, podemos desarrollar un algoritmo que no sólo resuelva el problema de palabra para gramáticas libres de contexto, sino que, además, genera un árbol de derivación para las palabras aceptadas.
Lo que haremos será definir una serie de aplicaciones: $gen(i, j, -)$ definidas en los subconjuntos $t_{i,j}$ de la tabla construida a partir del algoritmo CYK antes definido.
Comenzaremos introduciendo una enumeración en al conjunto de las producciones. Elegimos números enteros positivos $\{1, \ldots, N\}$ que usaremos para asignar un número a cada producción de P. La manera de enumerar es libre pudiendo elegir la que más nos convenga o guste. Esta es una cualidad significativa del análisis CYK: no importa la preferencia de nuestra enumeración, podríamos recuperar todas las posibles opciones de árbol de derivación. Así, buscando localmente, podemos encontrar una de las soluciones "globales" (con lo que admite una ideología "greedy" (voraz)). De hecho, esto es lo que permite enfrentar clases de lenguajes libres de contextos cualesquiera (sean o no determinísticos y forma parte intr´inseca de su naturaleza generalista.
La idea de base es la siguiente:

- Si $A \notin t_{i,j}$, la imagen de $gen(i, j, A)$ no está definida. En ese caso devolveremos **Error**.
- Si $A \in t_{i,1}$, es porque la producción (m) es de la forma $A \mapsto a_i$. Definamos $gen(i, 1A) := [m]$
- Si $A \in t_{i,j}$, entonces existe un k, $1 \leq k < j$ y existe una producción con número (m) de la forma $A \mapsto BC$ con $B \in t_{i,k}$, $C \in t_{i+k,j-k}$. Entonces,

$$gen(i, j, A) = [m, gen(i, k, B), gen(i + k, j - k, C)].$$

Esta definición adolecede de una dificultad fundamental:
No es aplicación puesto que podría haber más de una producción con las propiedades prescritas. Por ejemplo, podríamos tener:

- Una producción con número (m) de la forma $A \mapsto BC$, $B \in t_{i,k}$, $C \in t_{i+k,j-k}$ y
- otra producción con número (r) de la forma $A \mapsto XY$, $X \in t_{i,k}$, $Y \in t_{i+k,j-k}$.

Para corregir este "indeterminismo" y aprovechando su naturaleza de matroide, modificamos la definición de *gen* y lo transformamos en una determinística mediante, por ejemplo, eligiendo la producción de menor número entre las posibles. Es decir, el último item de la definición anterior queda:

- Si $A \in t_{i,j}$, consideremos todas las producciones de P tales que existe un k, $1 \leq k < j$ y la producción $A \mapsto BC$ con $B \in t_{i,k}$, $C \in t_{i+k,j-k}$. *Sea m el mínimo de las enumeraciones de tales producciones.* Entonces, definiremos

$$gen(i, j, A) = [m, gen(i, k, B), gen(i + k, j - k, C)]$$

7.4.3. El Algoritmo de Análisis Sintáctico CYK

Supondremos fijada una gramática libre de contexto $G := (V, \Sigma, Q_0, P)$ en CNF, $\lambda-$libre, en la que las producciones de P están enumeradas de 1 a N.

INPUT: Una forma terminal $x = x_1 x_2 \cdots x_n \in \Sigma^*$.
OUTPUT: Si $x \in L(G)$, devuelve un árbol de derivación de x en G, en caso contrario devuelve **error**.

Calcular la tabla $\{t_{i,j}\}$ del algoritmo CYK anterior (obsérvese que depende de x y, obviamente, de G).

if $Q_0 \notin t_{1,n}$ OUTPUT **error**
else do
eval $gen(1, n, Q_0)$
fi
OUTPUT $gen(1, n, Q_0)$.
end

Teorema 7.4.3. *El anterior algoritmo da como output un árbol de derivación de ω si $\omega \in L(G)$ y devuelve* **error** *en caso contrario. El tiempo de ejecución es del orden $O(n^3)$ y el espacio consumido es de orden $O(n^2)$.*

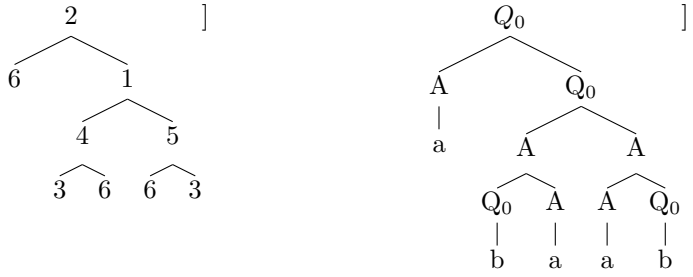
Ahora ya estamos listos para hallar una derivación de la palabra $x = abaab$ Empezaremos etiquetando las producciones:

(1)	$Q_0 \mapsto AA$	(4)	$A \mapsto Q_0A$
(2)	$Q_0 \mapsto AQ_0$	(5)	$A \mapsto AQ_0$
(3)	$Q_0 \mapsto b$	(6)	$A \mapsto a$

y ahora volvamos a la tabla 7.1. Por la definición sabemos que

$$\begin{aligned} gen(1,5,Q_0) &= [2, gen(1,1,A), gen(2,4,Q_0)] \\ &= [2,[6],[1, gen(2,2,A), gen(4,2,Q_0)]] \\ &= [2,[6],[1,[4, gen(2,1,Q_0), gen(3,1,A)],[5, gen(4,1,A), gen(5,1,Q_0)]]] \\ &= [2,[6],[1,[4,[3],[6]],[5,[6],[3]]]] \end{aligned}$$

Podemos asociar esto a la representación de un árbol utilizando pre orden.



Al lado, hemos dibujado el árbol de derivacion de la palabra.

7.5. Traductores Push–Down.

Los procesos de traducción con gramáticas libres de contexto son gestionados por un modelo de máquina basado en los autómatas con pila descritos en el Capítulo precedente: *Los traductores push–down* o traductores con pila.
Informalmente, un traductor con pila es un objeto compuesto de los siguientes elementos:

- **Un autómata con pila.** Esto es, disponemos de una cinta de entrada (IT), una unidad de control con una cantidad finita de memoria, y una pila.
- **Una cinta de output.** En la que el autómata simplemente puede escribir, no puede leer sus contenidos, y puede avanzar un paso a la derecha siempre que la celda anterior no esté vacía.

Las operaciones de un traductor con pila son sucesiones de operaciones del tipo siguiente:

i) **Read.** Lee una celda en la cinta de entrada y el top de la pila. Eventualmente puede hacer operaciones de lectura λ en la cinta de entrada. por supuesto, lee también el estado actual en la unidad de control.

ii) **Transition.** De acuerdo con una función de transición (o de una tabla como la que usaremos en las secciones siguientes) el autómata indica tres operaciones básicas a realizar en cada uno de los cuatro elementos.

iii) **Write and Move.** Escribirá en cinta, pila y/o cinta de outpt en función de las reglas naturales:

- En la cinta de Input: Si la lectura es una **Lambda**-lectura, no hace nada en la cinta de input. En caso de tratarse de una lectura propiamente(**Read**) borrará un símbolo de de la cinta y avanzará un paso hacia la derecha.
- En la unidad de control, modifica el estado conforme se indica en la Transición.
- En la pila realiza la operación $\texttt{push}(\texttt{pop}(\text{Pila}), z)$ donde z es el símbolo indicado por la transición. Será una operación `push` si $z \neq \lambda$ y una operación `pop` si $z = \lambda$, como ya se indicó en los Autómatas con Pila.
- En la cinta de Output escribe lo que se le indique. Puede ser que no escriba nada, en cuyo caso no se mueve, o que escriba un símbolo en cuyo caso se mueve un paso a la derecha hasta la siguiente celda vacía.

La computación se termina con *pila y cinta vacías*. Es decir, el autómata funciona con un gran ciclo **while** cuya condición de parada es que la cinta y la pila están vacías.
En ese caso, se dice que el input es aceptado (lo que significará que el input está en el lenguaje generado por la gramática de input). El output, es el contenido de la cinta de output y es el árbol sintáctico de derivación de la palabra escrita en la cinta de input (en un cierto sentido, la "traducción" de la palabra escrita en la cinta de input).
Esta definición informal va acompañada de una definición formal que escribimos a continuación:

Definición 7.5.1 (PDT)**.** Un traductor con pila (push–down transducer o PDT), es una lista $T := (Q, \Sigma, \Gamma, \Delta, Q_0, Z_0, F, \delta)$ donde:

i) Q es un conjunto finito (espacio de estados)

ii) Σ es un alfabeto finito, llamado alfabeto del input.

iii) Γ es un alfabeto finito, llamado alfabeto de la pila.

iv) Δ es un alfabeto finito, llamado alfabeto del output.

v) $q_0 \in Q$ es el estado inicial.

vi) $F \subseteq Q$ son los estados finales aceptadores.

vii) Z_0 es un símbolo especial, llamado fondo de la pila.

viii) δ es una correspondencia llamada función de transición:

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \times (\Gamma \cup \{Z_0\}) \to Q \times \Gamma^* \times \Delta^*.$$

7.5.0.1. Sistema de Transición asociado a un PDT.

Denominaremos configuraciones de un traductor push-down a los elementos del conjunto

$$\mathcal{S} := Q \times \Sigma^* \times Z_0 \cdot \Gamma^* \times \Delta^*.$$

Una configuración $c = (q, x, Z_0 z, y)$ está describiendo q como el estado de la unidad de control, x es el contenido de la cinta de input, $Z_0 z$ es el contenido de la pila, y es el contenido de la cinta de output.
De la manera obvia se describen las transiciones $c \to c'$ entre dos transiciones del sistema. Lo dejamos como ejercicio para los alumnos.

La configuración inicial en una palabra $\omega \in \Sigma^*$ será dada por:

$$I(\omega) := (q_0, \omega, Z_0, \lambda),$$

es decir, ω está en la cinta de input, q_0 en la unidad de control, la pila está vacía y la cinta de output también.
Una configuración final aceptadora es una configuración con pila y cinta vacías, esto es, una configuración de la forma (q, λ, Z_0, y) con $y \in \Delta^*$.
Una palabra $\omega \in \Sigma^*$ es aceptada si alcanza una configuración final aceptadora dentro del sistema de transición. Esto es, si ocurre que:

$$I(\omega) = (q_0, \omega, Z_0, \lambda) \vdash (q, \lambda, Z_0, y).$$

la palabra y es el resultado de la traducción de ω en el caso de que omega sea aceptada por el PDT (i.e. $y = \tau(\omega)$).

Observación 7.5.1. En lo que sigue, la traducción se hará a través de parsing y SDTS. Por tanto, usaremos PDT's del modo siguiente: Si L_1 es el lenguaje del input y T_{input} es la gramática del input asociada a nuestro sistema SDTS, procederemos enumerado los producciones de T_{input}. Definiremos el lenguaje de output Δ como $\Delta := \{1, \ldots, N\}$, donde N es el número de las producciones de T_{input}. Así, el output de un analizador sintáctico (descrito también mediante un PDT) es una lista $i_1 i_2 \cdots i_r \in \Delta^*$, que indican (en modelo directo o en reverso) las producciones que se aplican.

Observación 7.5.2. En los casos que siguen, el PDT y sus estados se describirán mediante diversos tipos de tablas. En cada caso iremos mostrando cómo se construyen esas tablas.

7.6. Gramáticas $LL(k)$: Análisis Sintáctico

Se trata de un modelo de análisis sintáctico descendente (top–down) basado en left–parsing (o sea, buscando árboles de derivación más a la izquierda) y es determinista para ciertas clases de gramáticas: las gramáticas $LL(k)$.
El ejemplo clásico de lenguaje de programación que admiten parsing LL(1) es `Pascal`. Véase la obra de Niklaus Wirth y la tradición "europea" de análisis sintático (cf. [Wi, 96]).

7.6.1. FIRST & FOLLOW

Definición 7.6.1 (Frist)**.** Sea $G := (V, \Sigma, Q_0, P)$ una gramática libre de contexto. Para cada forma sentencial $\alpha \in (V \cup \Sigma)^*$ y para cada $k \in \mathbb{N}$ definiremos la función

$$FIRST_k^G(\alpha) := \{x \in \Sigma^* \ : \ \begin{cases} |x| = k & \exists \beta \in \Sigma^*,\ \alpha \vdash_{lm}^G x\beta \\ |x| < k & \alpha \vdash_{lm}^G x \end{cases} \}.$$

Omitiremos el superíndice ^{G} siempre que su presencia sea innecesaria por el contexto.

En otras palabras, el operador $FIRST_k$ asocia a cada forma sentencial los primeros k símbolos de cualquier forma terminal alcanzable desde α mediante derivaciones "más a la izquierda". Si α alcanza una forma terminal con menos de k símbolos $x \in \Sigma^*$, con derivaciones "más a la izquierda", entonces también x está en $FIRST_k(\alpha)$.
Si $\alpha \in \Sigma^*$, $FIRST_k(\alpha)$ son los primeros k símbolos de α. Más específicamente, si $\alpha := x_1 \cdots x_k x_{k+1} \cdots x_n \in \Sigma^*$, entonces

$$FIRST_k(\alpha) = \{x_1 \cdots x_k\},$$

y si $|\alpha| \leq k$, $FIRST_k(\alpha) = \{\alpha\}$.
Nos ocuparemos, sobre todo, del operador $FIRST(\alpha) := FIRST_1(\alpha)$. Para dar un algoritmo que los calcule comenzaremos con algunas propiedades básicas del Operador FIRST. Más aún, comenzaremos con una construcción de un operador entre lenguajes:

Definición 7.6.2. Sean $L_1, \ldots, L_n \subseteq (V \cup \Sigma)^*$ lenguajes no vacíos. Definiremos el lenguaje $L_1 \oplus_1 \cdots \oplus_1 L_n \subseteq (V \cup \Sigma)^*$ mediante la siguiente igualdad: Sea $j \in \{1, \ldots, n\}$ tal que $\lambda \in L_i$ para $1 \leq i \leq j-1$ y $\lambda \notin L_j$. Entonces,

$$L_1 \oplus_1 \cdots \oplus_1 L_n := \bigcup_{i=1}^{j} L_i.$$

Supongamos dada una aplicación

$$F : (V \cup \Sigma) \longrightarrow \mathcal{P}((V \cup \Sigma))^*,$$

escribiremos $\oplus_1^F \alpha$ para cada forma sentencial α queriendo denotar

$$\oplus_1^F \alpha := F(X_1) \oplus_1 \cdots \oplus_1 F(X_n),$$

cuando $\alpha = X_1 \cdots X_n$.

Ejemplo 7.6.1. Dados $L_1 = \{\lambda, abb\}$ y $L_2 = \{b, bab\}$, entonces

$$L_1 \oplus L_2 = \{abb, b, bab, \lambda\} = L_1 \cup L_2, \quad L_2 \oplus L_1 = \{b, bab\} = L_2. \tag{7.6.1}$$

Lema 7.6.1. *Con las anteriores notaciones, se tienen las siguientes propiedades.*

i) Si $X = \lambda$, $FIRST(\lambda) = \{\lambda\}$.

ii) Si $X = a \in \Sigma$, $FIRST(X) = \{a\}$.

iii) Si $\alpha := X_1 \cdots X_n$ *donde* $X_i \in (V \cup \Sigma)^*$, *entonces*

$$FIRST(\alpha) = \oplus_1^{FIRST} \alpha = FIRST(X_1) \oplus_1 \cdots \oplus_1 FIRST(X_n).$$

iv) Si V_λ *son los símbolos no terminales que alcanzan la palabra vacía, entonces* $\lambda \in FIRST(X)$ *si y solamente si* $X \in V_\lambda$.

A partir de Lema anterior, el cálculo de $FIRST$ de una forma sentencial cualquiera puede reducirse al cálculo de los $FIRST$'s de sus símbolos. Definimos el siguiente algoritmo incremental:

INPUT: una gramática libre de contexto $G := (V, \Sigma, Q_0, P)$.

Hallar $V_\lambda := \{A \in V \ : \ A \vdash \lambda\}$.
if $A \in \Sigma$, **then** $F(A) := \{A\}$[10]
else do
$G(A) := \emptyset$
$F(A) := \begin{cases} \{A\} & \text{si } A \notin V_\lambda \\ \{A, \lambda\} & \text{si } A \in V_\lambda \end{cases}$
while $F(A) \neq G(A)$ para algún $A \in V$ **do** [11]
$G(A) := F(A)$
$F(A) := \{\oplus_1^F \alpha \ : \ X \mapsto \alpha, \ X \in F(A)\} \cup \{F(A)\}$
od

OUTPUT: $F(A) \cap (\Sigma \cup \{\lambda\})$, para cada $A \in V \cup \Sigma$.

---

[10] Aceptaremos que $A \mapsto_{lm} A$.
[11] Nótese que la frase debe escribirse como **while** $\exists A \in V, F(A) \neq G(A)$**do**

Proposición 7.6.2 (Evaluación de FIRST)**.** *El anterior algoritmo evalúa la función* $FIRST(X)$ *para cada* $X \in V \cup \Sigma$.

Definición 7.6.3 (FOLLOW)**.** Con las mismas notaciones anteriores, para cada forma sentencial $\alpha \in (V \cup \Sigma)^*$ definiremos la función $FOLLOW_k^G(\alpha)$ del modo siguiente.

- Si existe una forma sentencial $\omega\alpha$ (i.e. si $Q_0 \vdash_G \omega\alpha$), con $\omega \in (V \cup \Sigma)^*$, entonces $\lambda \in FOLLOW_k^G(\alpha)$.
- Adicionalmente, definamos

$$FOLLOW_k^G(\alpha) := \{x \in \Sigma^* \ : \ Q_0 \vdash \omega\alpha\gamma,\ \omega, \gamma \in (V \cup \Sigma)^*,\ x \in FIRST_k^G(\gamma)\}.$$

  De nuevo, omitiremos el super–índice ^{G} cuando no genere confusión.

De nuevo nos ocuparemos solamente de $FOLLOW := FOLLOW_1$. Obsérvese que $FOLLOW_k(\alpha) \subseteq \Sigma^*$ y que para cada $x \in FOLLOW_k(\alpha)$, $|x| \leq k$.
Obsérvese que para cada variable $A \in V$, $FOLLOW(A)$ son todos los símbolos terminales que pueden aparecer a la derecha de A en alguna forma sentencial de la gramática. Si A estuviera al final de alguna forma sentencial, la palabra vacía también se añade.

INPUT: Una gramática libre de contexto $G := (V, \Sigma, Q_0, P)$ que supondremos libre de símbolos inútiles[12].

Hallar $FIRST(X)$, para cada $X \in (V \cup \Sigma)$.
$G(X) := \emptyset$, para cada $X \in V$
$F(Q_0) := \{\lambda\}$
$F(A) := \emptyset$, para cada $A \neq Q_0$.
while $F(A) \neq G(A)$ para algún $A \in V$, **do**
$G(A) = F(A)$ para cada $A \in V$

$$F(A) := \left[\bigcup_{B \mapsto \omega A \omega'} (FIRST(\omega') \setminus \{\lambda\}) \right] \bigcup$$

$$\bigcup \left[\bigcup_{B \mapsto \omega A \omega',\ \lambda \in FIRST(\omega')} F(B) \right] \bigcup F(A)$$

od
OUTPUT: $F(A) \cap (\Sigma \cup \{\lambda\})$ para cada $A \in V$.

Ejemplo 7.6.2. Consideremos la gramática con $V := \{Q_0, E', X, T, T'\}$, $\Sigma := \{id, (,), +, *\}$. Las producciones son:

$$P := \{Q_0 \mapsto TE', E' \mapsto +TE' \mid \lambda, T \mapsto XT', T' \mapsto *XT' \mid \lambda, X \mapsto (Q_0) \mid id\}.$$

Tendremos

$$FIRST(Q_0) = \{(, id\}, FIRST(E') := \{+, \lambda\},$$
$$FIRST(X) = \{(, id\}, FIRST(T) = \{(, id\}, FIRST(T') = \{*, \lambda\}.$$

Calculemos todos los $FOLLOW$'s de las variables:

- Inicializar:
$$G(Q_0) = \emptyset, G(E') = \emptyset, G(X) = \emptyset, G(T) = \emptyset, G(T') = \emptyset,$$
$$F(Q_0) = \{\lambda\}, F(E') = \emptyset, F(X) = \emptyset, F(T) = \emptyset, F(T') = \emptyset.$$
- Primer **while**:
  - Variable Q_0:
    - Producción $F \mapsto (Q_0)$: Añadir $FIRST())$ a $F(Q_0)$:
    $$F(Q_0) := F(Q_0) \cup FIRST()) = \{\lambda,)\}.$$
  - Variable T:
    - Producción $Q_0 \mapsto TE'$: Añadir $FIRST(E') \setminus \{\lambda\}$ a $F(T)$:
    - Producción $Q_0 \mapsto TE'$, nótese que $\lambda \in FIRST(E')$: Añadir $F(Q_0)$ a $F(T)$.
    - Producción $E' \mapsto +TE'$: Añadir $FIRST(E') \setminus \{\lambda\}$ a $F(T)$.
    - Producción $E' \mapsto +TE'$, nótese que $\lambda \in FIRST(E')$: Añadir $F(E')$ a $F(T)$.
    $$F(T) = (FIRST(E') \setminus \{\lambda\}) \cup F(Q_0) \cup (FIRST(E') \setminus \{\lambda\}) \cup F(E'),$$
    $$F(T) = \{+, \lambda\} \cup \{)\} \cup \{+\} \cup \emptyset = \{+, \lambda,)\}.$$

[12] De hecho, nos interesa que no haya símbolos inaccesibles.

- Variable E':
  - Producción $Q_0 \mapsto TE'$, $\gamma = \lambda$, $\lambda \in FIRST(\gamma)$: Añadir $F(Q_0)$ a $F(E')$.
  - Producción $E' \mapsto +TE'$:No añade nada nuevo.

$$F(E') = F(Q_0) = \{\lambda,)\}.$$

- Variable X:
  - Producción $T \mapsto XT'$: Añadir $FIRST(T') \setminus \{\lambda\}$ a $F(X)$.
  - Producción $T' \mapsto *XT'$: Idem.
  - Producción $T \mapsto XT'$, como $\lambda \in FIRST(T')$: Añadir $F(T)$ a $F(X)$.

$$F(X) = (FIRST(T') \setminus \{\lambda\}) \cup F(T) = \{*, \lambda, +,)\}.$$

- Variable T':
  - Producción $T \mapsto XT'$: Añadir $F(T)$ a $F(T')$
  - Producción $T' \mapsto *XT'$: idem.

$$F(T') = \{+, \lambda,)\}.$$

- Segundo **while**: Todos coinciden.
- OUTPUT:

$$F(Q_0) = \{\lambda,)\}, F(E') = \{\lambda,)\}, F(X) = \{*, \lambda,), +\}, F(T) = \{+, \lambda,)\}, F(T') = \{+, \lambda,)\}.$$

7.6.2. Gramáticas $LL(k)$

Definición 7.6.4 (Gramáticas LL(k))**.** Una gramática libre de contexto $G = (V, \Sigma, Q_0, P)$ se dice de clase $LL(k)$ si verifica la siguiente propiedad: Dadas dos derivaciones, donde $\omega \in \Sigma^*, A \in V, \alpha, \beta, \gamma \in (V \cup \Sigma)^*$, del tipo siguiente:

- $Q_0 \vdash_{lm} \omega A\gamma \vdash_{lm} \omega\alpha\gamma \vdash \omega x \in \Sigma^*$,
- $Q_0 \vdash_{lm} \omega A\gamma \vdash_{lm} \omega\beta\gamma \vdash \omega y \in \Sigma^*$,

Si $FIRST_k(x) = FIRST_k(y)$, entonces $\alpha = \beta$.

La idea es que si hacemos dos derivaciones a izquierda desde una variable de nuestra gramática, y si llegamos a dos formas terminales en las que los primeros k símbolos a partir de A de una forma terminal coinciden, entonces es que hemos tenido que hacer la misma derivación desde A. La expresión formal es delicadamente retorcida, pero su sentido no se verá hasta que no procedamos a la construcción de la tabla de predicción y análisis sintáctico. Por ahora veamos unos poco ejemplos.

Ejemplo 7.6.3. Un ejemplo de gramática $LL(1)$ es la dada mediante: $Q_0 \mapsto aAQ_0 \mid b, A \mapsto a \mid bQ_0A$

Ejemplo 7.6.4. La gramática $\{Q_0 \mapsto \lambda \mid abA, A \mapsto Q_0aa \mid b\}$ es una gramática $LL(2)$

Ejemplo 7.6.5. La gramática $G_3 = (\{Q_0, A, B\}, \{0, 1, a, b\}, P_3, Q_0)$, donde

$$P_3 := \{Q_0 \mapsto A \mid B, A \mapsto aAb \mid 0, B \mapsto aBbb \mid 1\},$$

no es una gramática $LL(k)$ para cualquier k. El lenguaje generado $L(G_3)$ es el lenguaje dado por

$$L(G_3) := \{a^n 0 b^n \ : \ n \geq 0\} \cup \{a^n 1 b^{2n} \ : \ n \geq 0\}.$$

Proposición 7.6.3. *Una gramática $G = (V, \Sigma, Q_0, P)$ es $LL(k)$ si y solamente si se verifica la siguiente propiedad:*
Dadas dos producciones $A \mapsto \beta$ y $A \mapsto \gamma$ tales que A es accesible y se tiene $Q_0 \vdash_{lm} \omega A \alpha$, con $\omega \in \Sigma^*$ y $\alpha \in (V \cup \Sigma)^*$, entonces

$$FIRST_k(\beta\alpha) \cap FIRST_k(\gamma\alpha) = \emptyset.$$

Demostración. Siguiendo la propia definición. □

Ejemplo 7.6.6. La gramática $\{Q_0 \mapsto aQ_0 \mid a\}$ no puede ser $LL(1)$ porque $FIRST_1(aQ_0) = FIRST_1(a) = a$.

Como nos dicta la intuición, en las gramáticas $LL(k)$ tendremos que calcular $FIRST_k(\alpha)$, donde α es una forma no terminal. Estudiemos primero algunas propiedades que generalizan el caso $k = 1$.

Definición 7.6.5. Sea $L_1, L_2 \in \Sigma^*$, dos lenguajes definimos:

$$L_1 \oplus_k L_2 = \left\{ \omega : \exists x \in \text{Ł}_1,\ \exists y \in \text{Ł}_2 \begin{cases} |xy| \leq k \text{ y } \omega = xy, \text{ o} \\ w = FIRST_k(xy). \end{cases} \right\}$$

Nótese que esta definición puede ser extendida a de una forma recursiva a $L_1 \oplus_k \dots L_n$ simplemente calculando primero $M_1 = L_1 \oplus_k L_2$ y repetir el argumento con $M_2 = M_1 \oplus_k L_3$ sucesivamente.

Lema 7.6.4. *Dada una gramática libre de contexto G y una forma sentencial $\alpha\beta$ se tiene que*

$$FIRST_k(\alpha\beta) = FIRST_k(\alpha) \oplus_k FIRST_k(\beta).$$

Demostración. Las palabras que se pueden derivar a partir de α forman un lenguaje, que podemos llamar L_1, lo mismo las palabras que se pueden derivar de β. Tenemos que si

$$\alpha\beta \vdash_{lm} xy \text{ entonces } \alpha \vdash_{lm} x, \qquad \beta \vdash_{lm} y.$$

Por lo tanto, si $\omega \in FIRST_k(L_1 L_2) \iff \omega \in L_1 \oplus_k L_2$. □

Para calcular el $FIRST_k(\alpha)$ utilizaremos una generalización de que aparece en [AhUl, 72a].

INPUT: una gramática libre de contexto $G := (V, \Sigma, Q_0, P)$.

Definir $F_i(a) = a$ para todo símbolo del alfabeto y para todo $0 \leq i \leq k$.
Definir $F_0(A) = \{x \in \Sigma^k : A \mapsto x\alpha\}$ para todo símbolo del alfabeto y para todo $0 \leq i \leq k$.
Para $1 \leq i \leq k$ **y mientras** $F_{i-1}(A) \neq F_i(A)$ **para alguna variable** A **hacer**
    Para cada variable A hacer
        $F_i(A) = \{x \in \Sigma^k : A \mapsto Y_1 \dots Y_n \text{ y } x \in F_{i-1}(Y_1) \oplus_k \dots \oplus_k F_{i-1}(Y_n)\}$.
        fin hacer
    fin hacer
OUTPUT: $F(A) \cap \Sigma \cup \{\lambda\}$, para cada $A \in V \cup \Sigma$.

7.6.3. Tabla de Análisis Sintáctico para Gramáticas $LL(1)$

Antes de comenzar, enumeraremos nuestras producciones, asignándole un número natural a cada una de ellas. Además, introduciremos un nuevo símbolo § que hará las funciones de fondo de la pila.
Construiremos una tabla

$$M : (V \cup \Sigma \cup \{\S\}) \times (\Sigma \cup \{\lambda\}) \longrightarrow \mathcal{P}(P) \cup \{\mathbf{pop}, \mathbf{accept}, \mathbf{error}\},$$

donde $\mathcal{P}(P)$ es el conjunto de todos los subconjuntos del conjunto de las producciones.

INPUT: Una gramática libre de contexto $G = (V, \Sigma, Q_0, P)$.
Establecemos una tabla M cuyas filas están indicadas por los elementos de $V \cup \Sigma \cup \{\S\}$ y cuyas columnas están indicadas por los elementos de $\Sigma \cup \{\lambda\}$.
Definiremos M del modo siguiente:

- Dada una producción (i) $A \mapsto \alpha$
  - Para cada $a \in FIRST(\alpha),\ a \neq \lambda$, añadir i a la casilla $M(A, a)$.
  - Si $\lambda \in FIRST(\alpha)$ añadir i en todas las casillas $M(A, b)$ para cada $b \in FOLLOW(A)$.
- $M(a, a) =$**pop** para cada $a \in \Sigma$.
- $M(\S, \lambda) =$**accept**.
- En todos los demás casos escribir $M(X, i) =$**error**.

Observación 7.6.1. Si bien es verdad que para simplificar la escritura de una tabla, conviene enumerar las producciones, se hubiera podido hacer de la misma manera incluyendo la producción en cada casilla. Como se ha comentado antes, se usará esta enumeración tanto para definir la tabla de análisis sintáctico (parsing) como los procesos que sigan en la Subsección siguiente.
Como ejemplo, consideremos la gramática $G = (V, \Sigma, Q_0, P)$, donde las producciones son:

$$P := \{Q_0 \mapsto aAQ_0 \mid b, A \mapsto a \mid bQ_0A\}.$$

Enumeramos estas producciones del modo siguiente:

$$\begin{array}{ll} (1) & Q_0 \mapsto aAQ_0 \\ (2) & Q_0 \mapsto b \\ (3) & A \mapsto a \\ (4) & A \mapsto bQ_0A \end{array}$$

Ejemplo 7.6.7. Veamos un ejemplo basado en la gramática descrita en el ejemplo anterior. La tabla de análisis sintáctico correspondiente será la siguiente:

	a	b	λ
Q_0	1	2	**error**
A	3	4	**error**
a	**pop**	**error**	**error**
b	**error**	**pop**	**error**
§	**error**	**error**	**accept**

Obsérvese que la gramática es $LL(1)$ porque la tabla tiene siempre una entrada (y no más).

Proposición 7.6.5. *Dada una gramática libre de contexto G, y dada $T(G)$ la tabla construida por el algoritmo anterior, entonces G es $LL(1)$ si y solamente si todos las casillas de la tabla $T(G)$ contienen exactamente una producción o una de las palabras seleccionadas (***pop, accept, error***).*

7.6.4. Parsing Gramáticas $LL(1)$

Vamos a construir un traductor con pila (PDT) asociado a cada gramática $LL(1)$. Además, ese traductor será determinista cuando la gramática sea $LL(1)$. El PDT se define con las reglas siguientes:

- El espacio de estados estará formado por un estado más relevante M que hace referencia a la tabla de análisis sintáctico tal y como se ha descrito en la Subsección anterior, un estado inicial q_0 que sólo aparece al inicializar el proceso, un segundo estado **error** que indica que se ha producido un error en el análisis sintáctico y un último estado **accept** que indica que ha terminado la computación y hemos aceptado el input. Por tanto, $F = \{\textbf{accept}\}$, $Q := \{q_0, M, \textbf{error}, \textbf{accept}\}$.
- El alfabeto de la cinta de input es el alfabeto Σ de la gramática dada.
- El alfabeto Δ de la cinta de output son los números naturales $\{1, \ldots, N\}$ de una enumeración de las producciones de la gramática original.
- El alfabeto de la pila Γ es la unión de los alfabetos V (conjunto de variables de la gramática), Σ (el alfabeto de la cinta de input) y el símbolo § que jugará el papel de fondo de la pila.

$$\Gamma = V \cup \Sigma \cup \{\S\}.$$

- La función de transición δ vendrá dada por

$$\delta : \{q_0, M\} \times (\Sigma \cup \{\lambda\}) \times (\Gamma \cup \{\S\}) \longrightarrow \{M\} \times \Gamma^* \times \Delta.$$

  Las transiciones quedarán definidas por las reglas siguientes:

  i) Inicializar $\delta(q_0, \lambda, \S) = (M, Q_0, \lambda)$, donde Q_0 es la variable inicial de la gramática a analizar. Significa que comenzamos con una $\lambda-$transición para "cargar" la variable inicial en la pila, sin borrar ningún dígito de la cinta de input y sin añadir ninguna producción en la cinta de output.

  ii) Dados $X \in \Gamma$, $u \in \Sigma \cup \{\lambda\}$, supongamos que $M(X, u) \neq \textbf{error}, \textbf{pop}, \textbf{accept}$. [13] Entonces, existe una producción (i) tal que $i \in M(X, u)$. Supongamos que esa producción es de la forma $X \mapsto \beta$. La transición es, definida mediante:

$$\delta(M, u, X) = (M, \beta^R, M(X, u)) = (M, \beta^R, i)$$

  , donde i es el número de la producción correspondiente y $\beta^R \in \Gamma^*$ es el reverso de la lista de símbolos que aparecen a la derecha en esa producción.

  Significa que hacemos $\texttt{push}(\texttt{pop}('lista'), \beta^R)$ en la pila, y que añadimos i a la cinta de output, pasando a la siguiente celda vacía. No nos movemos en la cinta de entrada, entendiñdola como una Lambda-transición.

  iii) Dados $X \in \Gamma$, $u \in \Sigma$, supongamos que $M(X, u) =$**pop**, definimos $\delta(M, u, X) = (M, \lambda, \lambda)$ (indicando que hacemos **pop** en la pila, borramos una celda en la cinta de input y no escribimos nada en la cinta de output).

  iv) Dados $X \in \Gamma$, $u \in \Sigma \cup \{\lambda\}$, supongamos que $M(X, u) =$**error**, entonces, el proceso de análisis sintáctico cambia a estado **error** y se detienen las computaciones, aunque no se aceptan.

  v) Por último se define $\delta(M/\textbf{accept}, \lambda, \S) = (M/\textbf{accept}, \S, \lambda)$ (indicando que ha acabado la computación y aceptamos).

Teorema 7.6.6. *Existe un algoritmo que, en tiempo lineal $O(n)$ en el tamaño de la entrada, realiza el análisis sintáctico de los lenguajes dados por gramáticas $LL(1)$.*

[13] Necesariamente X ha de ser una variable por la definición de la tabla M

Demostración. Es el algoritmo dado por el anterior traductor con pila. □

Ejemplo 7.6.8. Retomemos el ejemplo de Subsecciones anteriores. Es la gramática $G = (V, \Sigma, Q_0, P)$, donde $V = \{Q_0, A\}$, $\Sigma = \{a, b\}$, y las producciones son:

$$P := \{Q_0 \mapsto aAQ_0 \mid b, A \mapsto a \mid bQ_0A\}.$$

Enumeramos estas producciones del modo siguiente:

$$\begin{array}{ll} (1) & Q_0 \mapsto aAQ_0 \\ (2) & Q_0 \mapsto b \\ (3) & A \mapsto a \\ (4) & A \mapsto bQ_0A \end{array}$$

Tomamos la tabla M de análisis sintáctico:

M	a	b	λ
Q_0	1	2	**error**
A	3	4	**error**
a	**pop**	**error**	**error**
b	**error**	**pop**	**error**
§	**error**	**error**	**accept**

Ahora tenemos el autómata correspondiente que, esencialmente, está descrito en esa tabla. Así, podemos evaluar:

$\delta(M, -, -)$	a	b	λ
Q_0	$(Q_0Aa, 1)$	$(b, 2)$	**error**
A	$(a, 3)$	$(AQ_0b, 4)$	**error**
a	**pop**	**error**	**error**
b	**error**	**pop**	**error**
§	**error**	**error**	**accept**

Ejemplo 7.6.9 (Sistema de transición asociado al ejemplo)**.** A modo de ejemplo, podemos considerar el sistema de transición asociado a este ejemplo y al PDT definido en el ejemplo anterior.
Tomemos una palabra de input $\omega = abbab$ y construimos la configuración inicial en la palabra ω

$$I(\omega) := (M, abbab, \S, \lambda).$$

Las computaciones del autómata irán como sigue:

- Inicializamos $I(\omega) \to (M, abbab, Q_0\S, \lambda)$.
- Leemos $M(a, Q_0) = 1$, luego $\delta(M, a, Q_0) = (M, aAQ_0, 1)$, con lo que tenemos la transición:

$$(M, abbab, Q_0\S, \lambda) \to (M, abbab, aAQ_0\S, 1).$$

- Leemos (M, a, a) y $M(a, a) =$**pop**, así que borramos

$$(M, abbab, aAQ_0\S, 1) \to (M, bbab, AQ_0\S, 1).$$

- Leemos (M, b, A) y $M(b, A) = 4$, luego aplicamos la transición $\delta(M, b, A) = (M, bQ_0A, 4)$ y tenemos la transición:

$$(M, bbab, AQ_0\S, 1) \to (M, bbab, bQ_0AQ_0\S, 14).$$

- Acudiendo a la tabla, iremos obteniendo las transiciones siguientes:

$$(M, bbab, bQ_0AQ_0\S, 14) \to (M, bab, Q_0AQ_0\S, 14) \to (M, bab, bAQ_0\S, 142) \to$$
$$\to (M, ab, AQ_0\S, 142) \to (M, ab, aQ_0\S, 1423) \to (M, b, Q_0\S, 1423) \to$$
$$\to (M, b, b\S, 14232) \to (M, \lambda, \S, 14232) \to (\textbf{accept}, \lambda, \S, 14232).$$

Dejamos que el alumno verifique que las transiciones se comportan como se ha indicado.

Nótese cómo en la cinta de output se han ido escribiendo los números de las producciones que, aplicados con estrategia leftmost, componen en árbol de derivación de la palabra aceptada.

7.7. Cuestiones y Problemas

7.7.1. Cuestiones

Cuestión 7.7.1. Compara la siguiente versión iterativa del algoritmo propuesto para el cálculo de FIRST:

Definiremos los conjuntos $F_i(X)$ para valores crecientes de i del modo siguiente:

- **if** $X \in \Sigma$ **then** $F_i(X) = \{X\}$, para todo i.
- $F_0(X) := \{x \in \Sigma \cup \{\lambda\} \ : \ \exists X \mapsto x\alpha \in P\}$.
- $F_i(X) := \{x \ : \ \exists X \mapsto Y_1 \cdots Y_n \in P \text{ y } x \in F_{i-1}(Y_1) \oplus F_{i-1}(Y_2) \oplus \cdots \oplus F_{i-1}(Y_n)\} \cup F_{i-1}(X)$.

if $F_i(X) = F_{i+1}(X)$, para todo X **then** OUTPUT

$$\{F_i(X) \ : \ X \in V \cup \Sigma\}.$$

Cuestión 7.7.2. Consideremos el siguiente SDTS. Denotaremos a las variables utilizando $< x >$:

$$\begin{array}{rcl}
< exp > & \mapsto & sums < exp >^1 \ \ with \ \ < var > \mapsto < exp >^2 \ to < exp >^2 \ do < statement >, \\
 & & begin \ \ < var > \mapsto < exp > \\
 & & if < var > \leq < exp >^1 \ then; \\
 & & begin < statement > \\
 & & < var > \mapsto < var > +1; \\
 & & endend \\
< var > & \mapsto & < id >, < id > \\
< exp > & \mapsto & < id >, < id > \\
< id > & \mapsto & a < id >, a < id > \\
< id > & \mapsto & b < id >, b < id > \\
< id > & \mapsto & a, a \\
< id > & \mapsto & b, b
\end{array}$$

Razonar porque no es un sistema de traducción directa. ¿ Cual deberia ser la traducción para la siguiente palabra:

$$for \ a \mapsto b \ to \ aa \ do \ baa \mapsto bba.$$

Cuestión 7.7.3. Deducir como se escriben las transiciones en un traductor con pila.

Cuestión 7.7.4. En el algoritmo CYK, para construir los posibles árboles de derivación, hay que calcular varios valores $t_{i,j}$. Discutir que representan estos valores y por qué hay que calcular los valores $t_{1,n}$, donde n es la longitud de la palabra.

Cuestión 7.7.5. Dar un ejemplo de una gramática libre de contexto ambigua, construir un traductor con pila y explicar porque no es aconsejable su utilización en lenguajes de programación.

Cuestión 7.7.6. Suponed que en una gramática libre de contexto se tiene la siguiente producción $A \mapsto AA$. Discutir si la gramática es ambigua.

Cuestión 7.7.7. Dada la siguiente gramática:

$$Q_0 \mapsto 0A0|1B1B,\ A \mapsto 0BQ_0|1|\lambda,\ B \mapsto 0|A|\lambda.$$

Calcular $FIRST(AB),\ FIRST(AA),\ FOLLOW(1B)$.

Cuestión 7.7.8. Demostrar que la siguiente gramática es LL(1):

$$Q_0 \mapsto aAQ_0|b,\ A \mapsto a|bQ_0A.$$

7.7.2. Problemas

Problema 7.7.1. Consideremos el siguiente SDTS. Denotaremos a las variables utilizando $< x >$:

$$\begin{array}{rcl}
< exp > & \mapsto & sums < exp >^1\ \ with\ \ < var > \mapsto < exp >^2\ to < exp >^3, \\
 & & begin\ local\ t; \\
 & & t = 0; \\
 & & for < var > \mapsto < exp >^2\ to < exp >^3\ do: \\
 & & t \mapsto t + < exp >^1;\ result\ t; \\
 & & end \\
< var > & \mapsto & < id >, < id > \\
< exp > & \mapsto & < id >, < id > \\
< id > & \mapsto & a < id >, a < id > \\
< id > & \mapsto & b < id >, b < id > \\
< id > & \mapsto & a, a \\
< id > & \mapsto & b, b
\end{array}$$

Dar la traducción para las siguientes palabras:

i) *sum aa witha* $\mapsto$ *b to bb.*

ii) *sum sum a withaa* $\mapsto$ *aaa toaaaawithb* $\mapsto$ *bb tobbb.*

Problema 7.7.2. Sea $L_1,\ L_2$ lenguajes libres de contexto con intersección vacía. Construir un SDTS tal que traduzca

$$\{(x, y)|si\ x \in\ L_1\ entonces\ y = 0\ y\ si\ x \in L_2\ entonces\ y = 1\}$$

Problema 7.7.3. Sea el siguiente traductor con pila,

$$(\{q, q_1\}, \{a, +, *\}, \{*, +, E\}, \{a, *, +\}, \delta, q, E)$$

donde δ está definido por las siguientes relaciones:

$$\begin{aligned}
\delta(q,a,E) &= \{(q,e,a)\} \\
\delta(q,e,E) &= \{(q_1,e,e)\} \\
\delta(q,+,E) &= \{(q,EE+,e)\} \\
\delta(q,*,E) &= \{(q,EE*,e)\} \\
\delta(q,e,*) &= \{(q,e,*),(q_1,e,*)\} \\
\delta(q,e,+) &= \{(q_1,e,+),(q,e,+)\} \\
\delta(q_1,e,*) &= \{(q_1,e,e)\} \\
\delta(q_1,e,E) &= \{(q_1,e,e)\} \\
\delta(q_1,e,+) &= \{q_1,e,e)\}
\end{aligned}$$

Traducir la siguiente palabra $w = +*aaa$.
Definir un autómata para el lenguaje de partida.

Problema 7.7.4. Construir un SDTS tal que

$$\{(x,y)|x \in \{a,b\}^* \ y \ y = c^i\}$$

donde i es el valor absoluto del numero de a menos el numero de b de x.

Problema 7.7.5. Diseñar un traductor con cola que realice la siguiente función. Dado el lenguaje sobe el alfabeto $\Sigma = \{a,+,-\}$
definido por la siguiente expresión regular

$$(+^* -^* a)^*$$

elimine todos los operadores innecesarios. Como ejemplo, la siguiente palabra $w = ++a+++-a--a$ debería ser traducida a $w' = a-a+a$.

Problema 7.7.6. Sea R un lenguaje regular, construir un traductor con pila M, tal que para un lenguaje libre de contexto L, la traducción
de L sea L-R.

Problema 7.7.7. Dada la siguiente gramática,

$$Q_0 \mapsto 0Q_0Q_00|A, \ A \mapsto \lambda|1Q_01|,$$

Aplicar el algoritmo CYK para la palabra $w = 0110$.

Problema 7.7.8. Deducir el número de operaciones del algoritmo CYK y la capacidad de memoria necesaria utilizada.

Problema 7.7.9. Construir la tabla de análisis sintáctico y el traductor con pila para la siguiente gramática LL(1):

$$Q_0 \mapsto AB|BB, \ A \mapsto 0A|1, \ B \mapsto 2B12|3.$$

Problema 7.7.10. Construir la tabla de análisis sintáctico y el traductor con pila para la siguiente gramática LL(1):

$$Q_0 \mapsto BAB|CBC, \ A \mapsto 0B|1C, \ B \ mapsto1|0BB, \ C \mapsto 0C|1Q_0.$$

Problema 7.7.11. Demostrar que la siguiente gramática es LL(1),

$$Q_0 \mapsto TE', \ E' \mapsto +TE'|\lambda, \ T \mapsto FT', \ T' \mapsto *FT'|T'\lambda, \ F \mapsto (Q_0)|a$$

Calcular el traductor con pila y construir el árbol de derivación de la palabra w=a+a+a*a.

Problema 7.7.12. Construir la tabla CYK y la tabla LL(1) para la gramática siguiente (escrita en BNF y con las numeraciones indicadas para las producciones):

$$\begin{array}{rcrr}
\langle exp\rangle & := & \langle term\rangle\langle termTail\rangle & (1)\\
\langle termTail\rangle & := & \langle addop\rangle\langle term\rangle\langle termTail\rangle \mid \lambda & (2 \mid 3)\\
\langle term\rangle & := & \langle factor\rangle\langle factorTail\rangle & (4)\\
\langle facorTail\rangle & := & \langle multop\rangle\langle factor\rangle\langle factorTail\rangle \mid \lambda & (5 \mid 6)\\
\langle factor\rangle & := & (\langle exp\rangle) \mid NUM \mid ID & (7 \mid 8 \mid 9)\\
\langle addop\rangle & := & + \mid - & (10 \mid 11)\\
\langle multop\rangle & := & * \mid / & (12 \mid 13)
\end{array}$$

Problema 7.7.13. Consideremos la siguiente gramática libre de contexto. Denotaremos a las variables utilizando $< x >$ y $\Sigma = \{sums, with, \mapsto, to, do, a, b\}$

$$\begin{array}{rcl}
< exp > & := & sums < exp > \ with \ < var > \mapsto < exp > to < exp >\\
< exp > & := & sums\ b\ with \ < var > \mapsto < exp > to < exp >\\
< exp > & := & < id >\\
< var > & := & < id >\\
< id > & := & a < id >\\
< id > & := & b < id >\\
< id > & := & a\\
< id > & := & b
\end{array}$$

Hallar una derivación mas a la izquierda y otra más a la derecha de la siguiente palabra utilizando el algoritmo CYK:

$$sums\ b\ with\ a \mapsto b\ to\ aa$$

Discutir si la gramática es ambigua. Eliminar la segunda producción y construir la tabla de análisis sintáctico. Discutir si la gramática es LL(1).

Problema 7.7.14. Dada la siguiente gramática:

$$\begin{array}{rcl}
< Orden > & := & < IntroducirElemento >< Orden >\\
< Orden > & := & < EliminarElemento >< Orden >\\
< Orden > & := & \lambda\\
< IntroducirElemento > & := & push < id >;\\
< EliminarElemento > & := & remove < id >;\\
< id > & := & b < id >\\
< id > & := & a\\
< id > & := & b\\
< id > & := & \lambda
\end{array}$$

Discutir si es una gramática propia. Aplicar el algoritmo CYK a la palabra

$$push\ baa; push\ baaa; remove\ baaa; push\ b; remove\ baa;$$

Dar una derivación a la izquierda.

Problema 7.7.15. Hallar $FIRST(< id >< id >< Orden >)$ y $FOLLOW(< id >< id ><$

$EliminarElemento >$):

$$
\begin{aligned}
< Orden > &:= < IntroducirElemento >< Orden > \\
< Orden > &:= < EliminarElemento >< Orden > \\
< Orden > &:= \lambda \\
< IntroducirElemento > &:= push < id >; \\
< EliminarElemento > &:= remove < id >; \\
< id > &:= b < id > \\
< id > &:= a \\
< id > &:= b \\
< id > &:= \lambda
\end{aligned}
$$

Capítulo 8

Gramáticas Senibles al Contexto

Contents

8.1. Introducción 147
8.2. Lenguajes Sensibles al contexto 147
8.3. Autómatas Linealmente Acotados 147
8.3.1. Kuroda's Problems 148
8.3.2. El Problema de Palabra para Gramáticas Sensibles al contexto 149

8.1. Introducción

En este Capítulo nos limitaremos a establecer las nociones básicas relativas a las gramáticas y lenguajes sensibles al contexto así como el modelo de máquina (i.e. algoritmo) que decide la pertenencia a uno de esos lenguajes.
La gramáticas sensibles al contexto fueron introducidas por N. Chomsky, y su análisis a través de Autómatas linealmente acotados es debido a J. Myhill en su trabajo de 1960 (cf. [My, 60]). Habrá que esperar a los trabajos de P.S. Lanweber (cf. [Law, 63]) y, finalmente, S.Y. Kuroda (cf. [Ku, 64]) para tener una caracterización completa de los lenguajes sensibles al contexto mediante Autómatas Linealmente Acotados.
Las ideas básicas de esta equivalencia son as que siguen.

8.2. Lenguajes Sensibles al contexto

Definición 8.2.1. Una gramática sensible al contexto es una gramática $G := (V, \Sigma, Q_0, P)$ en la que todas las producciones son del tipo siguente:

$$\alpha A \beta \mapsto \alpha \gamma \beta,$$

donde $A \in V$ es una variable y $\alpha, \beta, \gamma \in (V \cup \Sigma)^*$, $\gamma \neq \lambda$. Los elementos α, β se denominan contexto de la variable A en esa producción.

Definición 8.2.2. Un lenguaje $L \subseteq \Sigma^*$ es un lenguaje sensible al contexto si es el lenguaje generado por una gramática sensible al contexto.

8.3. Autómatas Linealmente Acotados

Las ideas de Myhill, Landweber y Kuroda se establecen a través de la noción de Autómata Linealmente Acotado (Linear-Bounded Automata, LBA). Escribamos su definición. Nótese que esun tipo particular de Turing machine por lo que habría que acudir al Capítulo 9 para tener

una noción más completa. Nos conformamos con introducirlo aquí y dejamos al lector la revisión de la misma idea.

Definición 8.3.1 (**Linear-Bounded Automata**). Un autómata Linealmente acotado es una máquina de Turing $A := (Q, \Sigma, q_0, F, \Delta)$ que satisface las siguiente propiedades adicionales:

i) El alfabeto Σ se descompone como la unión de dos conjuntos disjuntos, uno de los cuales contiene dos símbolos distinguidos $\{\triangleright, \sharp\}$, el primero de los cuales ($\triangleright$) se considerará como cursor (o "inicio de palabra") y el segundo de los cuales ($\sharp$) actuará como "fin de palabra".

$$\Sigma := \Sigma_1 \cup \{\triangleright, \sharp\}.$$

ii) La máquina contiene un autómata finito determinista que acepta el lenguaje:

$$L_0 := \triangleright\Sigma_1^*\sharp := \{\triangleright x_1 \dots x_n \sharp \ : \ x_i \in \Sigma_1^*\}.$$

iii) No hay posibilidad de escribir más allá de los márgenes de las palabras marcados por $\triangleright$ y $\sharp$. Esto es, la función de ransición satisface:

- Si $\delta(q, \triangleright) = (q', x, \varepsilon)$, entonces, $x = \triangleright$ y $\varepsilon \in \{0, 1\}$. Es decir, no se modifica el márgen izquierdo y, llegando a él, sólo podemos permanecer o movernos hacia su derecha.
- Si $\delta(q, \sharp) = (q', y, \varepsilon)$, entonces, $x = \sharp$ y $\varepsilon \in \{-1, 0\}$. Es decir, no se modifica el márgen derecho y, llegando a él, sólo podemos permanecer o movernos hacia su izquierda.

Elsiguiente enuncaido caracteriza las gram'aticas sensibles al contexto en términos de un autómata que decide si una palabra está o no en el lenguaje. Es debido la serie de trabajos [My, 60], [Law, 63] y [Ku, 64].

Teorema 8.3.1 (**de Myhill-Landweber-Kuroda**). *Los lenguajes aceptados por autómatas linealmente acotados son exactamente los lenguajes sensibles al contexto.*

No incluiremos la demostración (por ahora) y dejamos al lector seguir las referencias citadas, que son legibles, en especial el artículo de Kuroda.

Observación 8.3.1. Una caracterización alternativa de los autómatas linealmente acotados pude otenerse en términos de clases de complejidad: Se trata de la clase de lenguajes aceptados por una máquina de Turing indeterminista usando espacio a lo sumo lineal en el tamao de la entrada. Es decir, los lenguajes sensibles al contexto son exactamente la clase:

$$CSL := \mathbf{NLINSPACE} := \mathbf{NSPACE}(n).$$

El lector puede seguir las notaciones del Capítulo 11 siguiente pare encontrar la defínción de estas notaciones.

8.3.1. Kuroda's Problems

En relación con las gramáticas sensibles al contexto, existen una serie de problemas conocidos como los problemas de Kuroda. Así, por ejemplo, sabemos que **NLINSPACE**=co-**NLINSPACE** (es decir, ambas clases de comlejidad coinciden, lo que significa que los complementarios de lenguajes sensibles al conteto son sensibles al conteto). Pero se desconoce la relación entre determinismo e indeterminismo para espacio linela (ver la Sección relativa al Teorema de Savitch en 11.4).

Problema Abierto 1. *Decidir si el siguiente contenido es una desigualdad:*

$$\mathbf{DLINSPACE} \subseteq \mathbf{NLINSPACE}.$$

8.3.2. El Problema de Palabra para Gramáticas Sensibles al contexto

Teorema 8.3.2. *El problema de palabra para gramáticas sensibles al contexto determinísticas es* **PSPACE***-completo.*

Demostración. Es una combinación del principal resultado de [Ku, 64] con el Teorema de Savitch (11.4.1). □

En cambio, el problea de a Vacuidad (que era decidible en el caso de Gramáticas libres de contexto, ver **??**) es indecidible en el caso de sensibles al contexto.

Parte II

Las Ideas de Alan Turing

Capítulo 9

Máquinas de Turing y Computabilidad.

Contents

9.1. Introducción 154
9.2. La Noción de Máquina de Turing 155
9.2.1. Dinámica de una Máquina de Turing: El Modelo Gráfico y el Sistema de Transición. 156
9.2.1.1. Modelo gráfico de una máquina de Turing. 157
9.2.1.2. Un paso de cálculo: 158
9.3. Algoritmos, funciones computables. Lenguajes Recursivamente enumerables 160
9.4. Funciones y Clases de Complejidad 161
9.5. Rudimentos con Máquinas de Turing 162
9.5.1. La independencia del alfabeto. 162
9.5.1.1. Primera Parte : Rechazar las Palabras que no están en la Imagen. 165
9.5.1.2. Segunda Parte : Simulación de Un Paso (SimUP). 166
9.5.2. Los conjuntos recursivamente enumerables como conjuntos imagen, indecidibilidad y otras interpretaciones 172
9.5.3. Independencia del Número de Cintas 176
9.6. La máquina Universal de A. Turing. 177
9.6.1. El código de una máquina de Turing. 177
9.6.1.1. El código de una máquina de Turing sobre el alfabeto Universal. 177
9.6.2. La máquina Universal : ejemplo de compilador e intérprete. 179
9.6.2.1. El código de una configuración de una máquina de Turing dada por su código. 179
9.6.3. El problema de la Parada y cuestiones conexas. 185
9.7. El Problema de la Parada. 185
9.8. El final del Problema X de Hilbert 186
9.9. Disgresión: Problemas de la Palabra 189
9.10. Números reales recursivamente enumerables. 191
9.11. Tape Compression Lemma y Linear Speed-Up 193
9.11.1. Tape Compression Lemma 193
9.11.2. Linear Speed–Up. 200

9.1. Introducción

La historia del origen de la Informática está fuertemente ligada al Problema X de Hilbert, al surgimiento de las nociones de algoritmo e, incluso, a la Criptografía y a la Segunda Guerra Mundial (con la aparición de los primero ordenadores *Colossus I y II*, creados en Bletchley Park y destruidos al final de la contienda). Siendo una historia entretenida, ha dejado mucho rastro escrito, por lo dejaremos para otro texto su desarrollo, pero conservaremos un hecho sustancial. Los primeros ordenadores se crean con la intenación de acelerar la capacidad de cálculo del ser humano y, para ello, se toman como modelos los modelos de máquina que, a la sazón, Allan Turing ha introducido tras el éxito espectacular de los Teoremas de Incompletitud e Indecidibilidad de K. Gödel. La realidad imita a la abstracción por vez primera de la historia de la Humanidad. Desde entonces hasta la fecha, las arquitecturas y estructuras internas de los ordenadores han preservado el mismo camino: *desde la asbtracción a la realidad, suando sea posible*. Cada vez son más potentes, más rápidos, más capaces de almacenar información...pero mantienen como estructura nuclear la misma ya establecida en la definición de Turing. Incluso cuando, en tiempos más recientes, se habla de la posibilidad de diseñar ordenadores cuánticos, el modelo matemático antecede a la realidad e, incluso, los análisis de complejidad de algorítmos potencialmente implementables en ordenadores cuánticos (la factorización de primos en tiempo polinomial anuniciada por Shor es un ejemplo) anteceden la mera existencia de tales objetos. Mientras no aparezcan nuevas arquitecturas reales que modifiquen en esencia el modelo de A. Turing, *la máquina de Turing seguirá siendo el modelo por antonomasia, el patrón y la unidad de medida de la computación*. Ésta es la razón por la cual el papel de la Matemática se ve reforzado: sólo desde la Matemática se pueden analizar los modelos abstractos y, por tanto, una realidad que imita intencionalmente a esa abstracción. Es un caso inaudito en la historia de la modelización matemática de la realidad.

Para enfrentar la resolución del Problema X de Hilbert, unos pocos matemáticos han decidido tratar el problema en el sentido opuesto: si no se conoce la noción de algoritmo poco o nada se puede reflexionar sobre el Problema X. Por tanto, es sobre la noción de algoritmo sobre la que vuelcan sus esfuerzos. En 1916, el matemático noruego *A. Thue* introduce sus sistemas de reescritura que serán pronto vistos como insuficientes para modelizar el concepto de algoritmo, aunque serán recuperados por Chomsky para su clasificación de los lenguajes formales (gramáticas formales).

Es hacia mediados de los años 30, cuando dos figuras relevantes aparecen para fijar la noción de algoritmo : al austríaco *K. Gödel* y el británico *A. Turing*. Rodeados de las figuras de *A. Church* y su alumno *S.C. Kleene*.

K. Gödel elabora su famosa tesis (23 páginas) en la que demuestra la Incompletitud de la Teoría Elemental de Números (véase la publicación del trabajo [1]). Aquí Gödel usa por vez una noción que tiende a las funciones computables (él las llamó "rekursiv"). Hoy son llamadas "funciones primitivas recursivas", i.e. sin el operador de minimización.

Durante los años 30, K. Gödel visita Princeton varias veces, hasta su traslado definitivo en 1940. En 1934, durante una de sus visitas, dió una charla cuyas notas circularon. Estas notas fueron tomadas por S.C. Kleene y Rosser, quienes a la sazón completaban sus estudios de doctorado bajo la dirección de A. Church. Las notas fueron publicadas como [2]. En esta conferencia, Gödel introduce la noción de computabilidad efectiva. Notó que formas más generales de recursión deberían ser admitidas antes de que sus funciones "rekursiv" pudieran cubrir todo lo computable. Así definió una clase que llamó "funciones generales recursivas" (al parecer, sugerido por una carta de Herbrand). Había nacido la noción de algoritmo.

A. Church dirige la tesis de S.C. Kleene sobre una noción de recursividad alternativa a la noción introducida por Gödel. Se trata de la noción de $\lambda-$*calculus*. En los trabajos de Kleene [3] y Church

[1] K. Gödel. "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I". *Monatsh. Math. Phys.* **38** (1931) 173–198.

[2] K. Gödel. "On undecidable propositions of formal mathematical systems". In *The undecidable, Basic Papers on Undecidable Propositions, Unsolvable Problemas and Computable Functions* , Raven Press, Hewlett, NY (1965) 41–71.

[3] S.C. Kleene. "$\lambda-$definability and recursiveness". *Duke Math. J.* **2** (1936) 340–353.

[4] demuestran que su noción de algoritmo y la noción propuesta por K. Gödel son exactamente la misma. Está naciendo la **Tesis de Church**.
Por su parte, A. Turing basó gran parte de su investigación en la interacción entre Matemáticas y la naciente Informática. Por ejemplo, en su trabajo de 1948 A. Turing [5] introducirá la *noción de condicionamiento del Algebra Lineal Numérica*, convirtiéndose en el fundador del Algebra Lineal Numérica moderna (en los textos al uso se concede la prioridad a J. von Neumann y colaboradores o a autores como Wilkinson, olvidando la contribución esencial de Turing).
A. Turing publicaba su modelo en su trabajo de 1936 [6] orientado a los números reales "definibles" (recursivamente enumerables) y, como Gödel, fue invitado a visitar Princeton entre 1936 y 1938, donde defenderá su tesis introduciendo las máquinas de Turing con oráculos. Probó en un apéndice la equivalencia entre su modelo y la $\lambda-$definibilidad. De hecho, dos son las aportaciones fundamentales de Turing en este trabajo del 36. De una parte, la introducción de un nuevor modelo alternativo de algoritmo (máquinas de Turing) y de otro el resultado de autoreducibilidad basado en la máquina Universal.
Emil Post también introdujo su modelo de cálculo en 1936, que resultó equivalente al de Turing (véase [7]), cuyo formalismo ha influenciado fuertemente los formalismos introducidos a posteriori. Post llegó a describir *la tesis de Church como una ley natural, "un descubrimiento fundamental" concerniente a "the mathematizing power of Homo Sapiens"*. Así, la Tesis de Church toma la forma siguiente :

Definición 9.1.1 (TESIS DE CHURCH :)**.** Llamaremos computable a toda función calculable por alguno de los siguientes métodos equivalentes de caracetrización :

- Calculable por una máquina de Turing,
- es un función gerenal recursiva,
- es $\lambda-$definible,
- es Post–calculable,

o calculable por cualquier otro procedimiento equivalente a alguno de éstos.

Observación 9.1.1 (Estructura del Capítulo)**.**

9.2. La Noción de Máquina de Turing

De las distintas nociones que subyacen a la Tesis de Church, la máquina de Turing tiene la mayor simplicidad definitoria y refleja fielmente los ordenadores físicos, se convierte así en el Patrón, en la **unidad de medida** de todos los fenómenos observables físcamente en un ordenador. En este sentido, la máquina de Turing es el entorno matemático natural matemático para el diseño y análisis de los algoritmos.
Comenzaremos introduciendo dos símbolos auxiliares distinguidos $\{\triangleright, \square\}$ que se incorporarán a los alfabetos finitos Σ de manera un poco artificial.

Definición 9.2.1. Llamaremos máquina de Turing indeterminista (con una sola cinta de Input (IT.) en la que autorizamos solamente lectura, k cintas de trabajo (WT.)) a todo quíntuplo $M := (\Sigma, Q, q_0, F, \delta)$ donde

i) Σ es un conjunto finito (alfabeto),

ii) Q es un conjunto finito (espacio de estados),

[4] A. Church. "An unsolvable problem of elementary number theory". *Am. J. Math.* **58** (1936) 345–363.
[5] A. Turing. "Rounding-off errors in matrix processes". *Quart. J. Mech. Appl. Math.*, **1** (1948) 287–308.
[6] A. Turing. "On computable numbers, with an application to the Enscheidungspoblem". *Proc. London Math. Soc., Ser. 2,* **42** (1936) 230–265.
[7] E. Post. "Finite Combinatory processes–formulation I". *J. Symbolic Logic* **1**, (1936), 103–105.

iii) $q_0 \in Q$ es el estado inicial,

iv) $F \subseteq Q$ son los estados finales aceptadores, $F_1 \subseteq Q$ son los estados finales de rechazo.

v) Una correspondencia (llamada función de transición)

$$\delta : Q \times (\Sigma \cup \{\triangleright, \square\})^{k+1} \longrightarrow Q \times (\Sigma \cup \{\square\})^k \times \{-1, 0, 1\}^{k+1}.$$

Si δ es un aplicación, la máquina de Turing M se denomina determinista.

Observación 9.2.1. En la definición anterior hemos introducido la "función de transición" como una "correspondencia":

$$\delta : Q \times (\Sigma \cup \{\triangleright, \square\})^{k+1} \longrightarrow Q \times (\Sigma \cup \{\square\})^k \times \{-1, 0, 1\}^{k+1}.$$

Una formulación equivalente sería la de definir la función de transición como una verdadera aplicación de la forma siguiente:

$$\delta : Q \times (\Sigma \cup \{\triangleright, \square\})^{k+1} \longrightarrow \mathcal{P}(Q \times (\Sigma \cup \{\square\})^k \times \{-1, 0, 1\}^{k+1}),$$

donde $\mathcal{P}(X)$ representa, para un conjunto X, el conjunto de todos sus subconjuntos, i.e.

$$\mathcal{P}(X) := \{S \ : \ S \subseteq X\}.$$

En este caso, el determinismo vendría determinado por el hecho de que a partir de δ se pudiera definir una aplicación parcial, es decir, que verificase:

$$\forall (q, x) \in Q \times (\Sigma \cap \{\triangleright, \square\})^{k+1}, \ \sharp\,(\delta(q, x)) \leq 1.$$

Dejamos al lector la elección de cuál de las dos versiones equivalentes le parece más atractiva.

Obviamente, no se puede entender el funcionamiento de una máquina de Turing sin entender su sistema dinámico de transición que presentaremos apoyándonos en su modelo gráfico.

9.2.1. Dinámica de una Máquina de Turing: El Modelo Gráfico y el Sistema de Transición.

Introduciremos para cada máquina de Turing $M = (\Sigma, Q, q_0, F, \delta)$ un un grafo orientado infinito $(S_M, \rightarrow_M)$, que denominaremos *sistema de transición* y que representa la acción (dinámica) de una máquina de Turing. Los elementos de S_M se denominan *configuraciones* (o snapshots) de la máquina M y representan la imagen de la máquina en un instante determinado. Las configuraciones vienen dadas por la siguiente definición:

$$S_M \subseteq Q \times (\Sigma^*)^{k+2} \times (\mathbb{N})^{k+2}$$

$$C := (q, x, y_1, \ldots, y_k, y_{k+1}, n_0, n_1, \ldots, n_k) \in Q \times (\Sigma^*)^{k+2} \times (\mathbb{N})^{k+2},$$

diremos que $C \in S_M$ si y solamente si se verifican las propiedades siguientes :

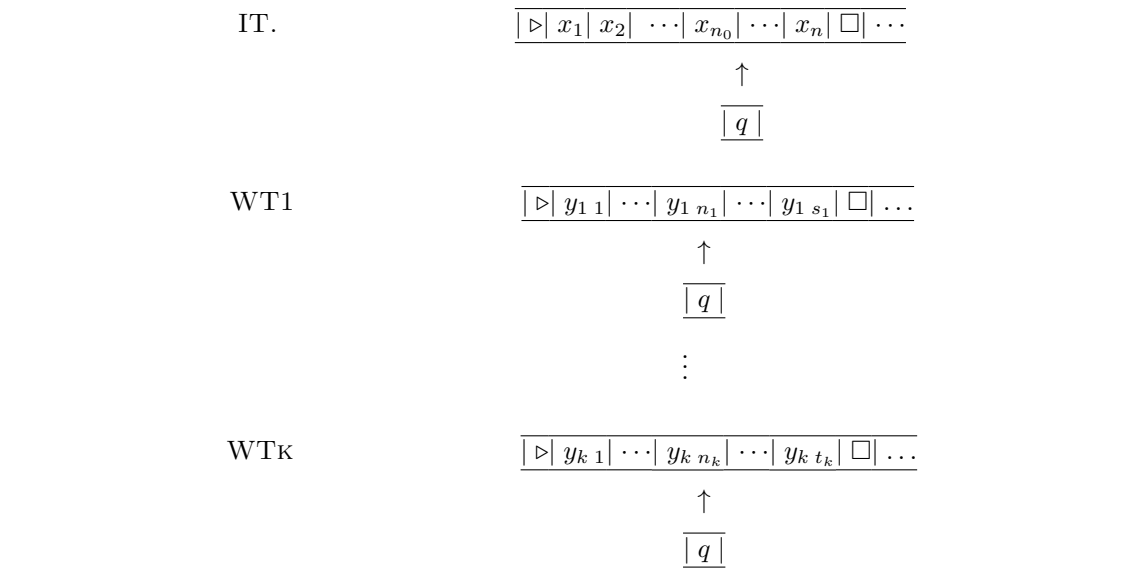
- $q \in Q$ es un estado (el estado de la configuración)
- $x := x_1 \ldots, x_n \in \Sigma^*$
- Para cada i, $1 \leq i \leq k$, se tiene

$$y_i := y_{i,1} \ldots y_{i,s_i} \in \Sigma^*$$

- $n_0, n_1, \ldots, n_k \in \mathbb{N}$ son las posiciones de las unidades de control, $0 \leq n_i \leq s_i + 1$, $1 \leq i \leq k$ y $0 \leq n_0 \leq n + 1$.

9.2.1.1. Modelo gráfico de una máquina de Turing.

La *interpretación de una configuración* debe hacerse del modo siguiente. Utilizaremos un modelo gráfico que representa la dinámica de la máquina de Turing. Para ello, observaremos la siguiente descripción de una máquina de Turing:
Dispondremos de una cinta de input y k cintas de trabajo. Cada cinta está dividida en unidades de memoria (o celdas) que son capaces de contener un símbolo del alfabeto Σ (o uno de los símbolos auxiliares $\triangleright$, $\square$). Cada cinta tiene adosada una unidad de control, con una memoria finita. En esa unidad de control podemos guardar un estado (la idea es que las unidades de control tienen la capacidad de almacenar una cantidad finita de información). La configuración C anterior queda descrita mediante la figura siguiente :



Para poder interpretar la figura, debemos hacer las siguientes consideraciones :

- El estado q es el indicador de la fase de cálculo en la que nos encontramos. El estado q está guardado en las unidades de control (todas con el mismo estado, aunque en diferentes posiciones).
- Cada unidad de control está apuntando una celda de cada cinta. El número n_i representa el lugar al que está apuntando la unidad i. En realidad, estamos indicando el número del registro de memoria que debemos atender en la fase de LECTURA.
- Cada cinta actúa como un disco duro (o, si se prefiere, una partición del disco duro en $k+1$ trozos). La información completa contenida en el disco duro no será utilizada simultáneamente. La cantidad de información utilizable es marcada por las unidades de control.
- El símbolo $\triangleright$ es el *cursor* : va a ser el representante del principio de cinta y sirve para prohibir (en la fase de Movimientos) el ir un paso a la izquierda del cursor. Lo que cuenta es la palabra que está descrita justo después. Nótese que admitimos su lectura, aunque no permitimos que sea escrito.
- El símbolo $\square$ denota que la celda está vacía. No debe confundirse con la palabra vacía λ.

Formalmente, estamos identificando los contenidos de las cintas (elementos de Σ^*) con las sucesiones

$$T \in (\Sigma \cup \{\triangleright, \square\})^{\mathbb{N}},$$

es decir, con aplicaciones $T : \mathbb{N} \longrightarrow \Sigma \cup \{\triangleright, \square\}$ que verifican las propiedades siguientes:

- $T(0) = \triangleright$,
- Existe $n \in \mathbb{N}$, tal que $T(i) \in \Sigma$, para $0 < i \leq n$, $T(k) = \square$, para $k \geq n+1$.

Así tenemos una identificación entre Σ^* y las sucesiones de este tipo, que asocia del modo siguiente:

$$\lambda \in \Sigma^* \longmapsto \triangleright\square\square\square\cdots ...$$

$$x_1 \cdots x_n \in \Sigma^* \in \Sigma^* \longmapsto \triangleright x_1 \cdots x_n \square\square\square\cdots ...$$

En realidad, los símbolos auxiliares $\triangleright, \square$ son útiles para interpretar la función de transición y se podría haber seguido otra vía. Aquí hemos elegido la filosofía propuesta por [ArBa, 09].

9.2.1.2. Un paso de cálculo:

Cuando una máquina de Turing se encuentra frente a una configuración como la descrita por la representación gráfica anterior, actúa del modo siguiente. Hay que dividir su acción en cinco etapas (el conjunto de todas ellas configura un *paso de cálculo* o, en términos técnicos, una *operación bit*).

i) Parada.

ii) Lectura

iii) Transición

iv) Escritura

v) Movimientos

`Parada.` Se trata de verificar la *Condición de Parada :* que viene marcada por los estados finales del modo siguiente :

---

while el estado q no es un estado final aceptador (i.e. $q \notin F$) **do**
  Lectura
  Transición
  Escritura
  Movimientos
od
Output el contenido de la cinta de trabajo $k-$ésima y termina la computación.
fi

Lectura. La fase de Lectura consiste en recuperar los contenidos de las celdas de las cintas señalados por las unidades de control. Así, tras verificar la condición de parada, procedemos a "leer" los contenidos de las distintas cintas de trabajo y de la cinta de input :

Lectura $:= (q, x_{n_0}, y_{1,n_1}, \ldots, y_{k,n_k}) \in Q \times (\Sigma \cup \{\triangleright, \square\}))^{k+1}$.
  go to Transición

Transición. La máquina acude con el resultado de Lectura a la función de transición δ (o, para ser más precisos, al grafo de la función de transición). Se obtiene el resultado de la transición mediante :

Transición:= δ(Lectura) $= (q', w_1, \ldots, w_k, \varepsilon_0, \ldots, \varepsilon_k) \in Q \times \Sigma^k \times \{-1, 0, 1\}^{k+1}$.
  go to Escritura

Escritura. El proceso de escritura contiene dos etapas. La primera es cambiar el contenido de las unidades de control (esto es cambiar el estado q que estaba en las unidades de control por el nuevo estado q'). La segunda es reemplazad el contenido de cada celda de cada cinta de trabajo (en el lugar donde estaba señalado) escribiendo (sustityendo por) el símbolo w_i. Esto es expresa diciendo :

Nuevo Estado : $q := q'$
for $i = 1$ to k **do**
    $y_{i,n_i} := w_i$
od
go to Movimientos

Movimientos. Ahora se trata de mover las unidades de control conforme a las reglas indicadas en la lista de movimientos

$$(\varepsilon_0, \ldots, \varepsilon_k) \in \{-1, 0, 1\}^{k+1},$$

conforme a las reglas siguientes :

$$\begin{array}{rcl} \varepsilon = -1 & \equiv & \text{Un paso a la izquierda} \\ \varepsilon = 0 & \equiv & \text{No te muevas} \\ \varepsilon = 1 & \equiv & \text{Un paso a la derecha} \end{array}$$

Así, las posiciones de las unidades de control se modifican mediante el siguiente proceso :

---

for $i = 0$ **to** k **do**
    $n_i := \text{máx}\{0, n_i + \varepsilon_i\}$
od
go to Parada

Observación 9.2.2. Nótese que hemos impuesto la condición de que la unidad de control no vaya más a la izquierda del cursor (posición 0) mediante $\text{máx}\{0, n_i + \varepsilon_i\}$. Es posible definir las llamadas máquinas de Turing bidireccionales, que son máquinas de Turing cuyas unidades de control pueden tomar posiciones positivas o negativas. Un sencillo ejercicio (identificando $\mathbb{Z}$ con $\mathbb{N}$, mediante una cualquiera de las biyecciones al uso) permite transformar máquinas de Turing con posiciones negativas con máquinas de Turing que solamente permiten ocupar posiciones positivas, sin cambiar esencialmente las propiedades de la teoría. Omitimos esta discusión por ser de escasa aportación de fondo.

El resultado de un paso (esto es, de las cuatro etapas descritas) es la configuración

$$C' := (q', x, y'_1, \ldots, y'_k; n'_0, \ldots, n'_k),$$

donde

- q' es el nuevo estado,
- el input x no ha sido modificado,
- y'_i es como y_i salvo en el lugar n_i donde y_{i,n_i} ha sido reemplazado por w_i
- las nuevas posiciones han sido cambiadas de acuerdo a los movimientos, esto es,

$$n'_i := n_i + \varepsilon_i.$$

Observación 9.2.3. Para garantizar que los contenidos de las cintas son siempre palabras con la codificación elegida, supondremos que no se puede escribir dejando una celda vacía entre dos secuencias de símbolos. Es decir, por comodidad de la fromalización con símbolos distinguidos $\{\triangleright, \square\}$, supondremos que

- No se puede escribir un símbolo ▷ más allá de la celda que ocupa el lugar 0. Es decir, no se admite que el contenido de una cinta pueda ser algo como:

$$\triangleright 01001 \triangleright \triangleright 11000\square\square\square\cdots,$$

con alfabeto $\{0,1\}$. Esto se puede hacer suponiendo que los símbolos auxiliares ▷ ya están escritos en cada cinta en la configuració inicial, pero que no se pueden escribir a los largo de la ejecución.

- No se pueden dejar celdas vacías entre símbolos de Σ. Es decir, no se admite que el contenido de una cinta pueda ser algo como:

$$\triangleright 01001\square\square\square\square 11000\square\square\square\cdots.$$

Esto se puede modelizar suponiendo que la función de transición no permite leer □ en una cinta y da como movimiento $+1$ sin haber escrito nada en la celda que acaba de leer.

Ejemplo 9.2.1. Además de los ejemplos de autómatas y autómatas con pila ya mostrados, el lector puede intentar diseñar una máquina de Turing que reconozca los palíndromos o, por ejemplo, puede intentar diseñar una máquina de Turing que realice las operaciones elementales entre naturales en base 2 (i.e. sobre el alfabeto $\{0,1\}$).

9.3. Algoritmos, funciones computables. Lenguajes Recursivamente enumerables

Dadas dos configuraciones C y C' de una máquina de Turing M, escribiremos $C \to_M C'$ para denotar que C' se obtiene desde C en *un paso de cálculo* (un paso de deducción o una operación bit). Escribiremos $C \vdash_M C'$ para denotar que C' se puede obtener de la configuración C en un número finito de pasos de cálculo de la máquina de Turing M (es decir,*alcanzable por un camino finito dentro del grafo del sistema de transición asociado*).

Definición 9.3.1 (Terminología Básica)**.** Dada una palabra $x \in \Sigma^*$ llamaremos **configuración inicial sobre** x como la configuración

$$I(x) := (q_0, \triangleright x, \triangleright, \ldots, \triangleright, 0, \ldots, 0) \in S_M,$$

esto es, todas las cintas de trabajo están vacías salvo la cinta de Input donde aparece la palabra x. Las unidades de control están sobre el cursor para empezar a trabajar.
Una palabra $x \in \Sigma^*$ se dice **aceptada** por una máquina de Turing M si a partir de la configuración inicial $I(x)$ se alcanza una configuración C (i.e. $I(x) \vdash_M C$ en la que el estado es un estado final aceptador (i.e. el estado de C está en F). El conjunto de las palabras aceptadas por M se llama **lenguaje aceptado por** M y es un subconjunto de Σ^* que se denota por $L(M)$.
Si $x \in L(M)$, existe C configuración final aceptadora tal que $I(x) \vdash_M C$, En ese caso, el output es el contenido de la $k-$ésima cinta de trabajo y se dice que

$$Res_M(x) := y_k \in \Sigma^*,$$

es el **resultado (output)** de la máquina de Turing sobre el input aceptado x.

Definición 9.3.2. Un lenguaje $L \subseteq \Sigma^*$ se llama *recursivamente enumerable* si es el lenguaje aceptado por alguna máquina de Turing (i.e. $L = L(M)$ para alguna máquina M). Se llama *recursivo* si tanto L como su complementario $\Sigma^* \setminus L$ son recursivamente enumerables.

Definición 9.3.3 (Funciones Computables)**.** Una función computable es una función

$$f : D(f) \subseteq \Sigma^* \longrightarrow \Sigma^*,$$

tal que existe una máquina de Turing M sobre Σ tal que :

i) $L(M) = D(f)$,

ii) $Res_M = f$.

9.4. Funciones y Clases de Complejidad

La utilización de las máquinas de Turing para el análisis de la compejidad de algoritmos se remonta a los años 60. Entre los trabajos iniciales para modelizar el fenómeno de la complejidad computacional caben destacar los trabajos de M. Blum ([Bl, 67]), J. Hartmanis y R Stearns ([HaSt, 65]), M.O. Rabin ([Ra, 60, Ra, 66]). Son Hartmanis y Stearns quienes inician la idelogía de las funciones de tiempo y espacio como funciones del tamaño de la entrada en su trabajo de 1965. Casi en la misma época, se establece la **Tesis de Cobham–Edmonds** (cf. [Co, 65], [**?**]) sobre los problemas *Tratables* informáticamente. Salvo la potencial aparición de la computación cuántica, el modelo de máquina de Turing como modelo de complejidad permanece. Las nociones básicas son las siguientes :

Definición 9.4.1 (Funciones de Tiempo y Espacio)**.** Sea M una máquina de Turing sobre el alfabeto Σ y sea $L(M)$ el lenguaje aceptado por M. Sea $x \in L(M)$.

i) Una *computación aceptadora sobre x de longitud t* en el sistema de configuración asociado a M es una sucesión finita:

$$I_M(x) = c_0 \mapsto_M c_1 \mapsto_M \cdots c_{t-1} \mapsto_M c_t,$$

donde c_t es una configuración final aceptadora de M.

ii) Llamaremos *tiempo de M sobre x* (y lo denotaremos mediante $T_M(x)$ al mínimo de las longitudes de las computaciones aceptadoras de M sobre x.

iii) Sea $c = (q; x, y_1, \ldots, y_k : n_0, \ldots, n_k) \in S_M$ una configuración de la máquina M con x en la cinta de input. Llamaremos *espacio de trabajo de c* a***

iv) Para una palabra $x \in \Sigma^*$ se denomina talla de x (y se denota mediante $|x| \in \mathbb{N}$) al número de símbolos de Σ que contiene la palabra x.

v) Dada una configuración C de M, se llama talla de la configuración (y se denota por $|C|$) al número total de bits ocupados en las distintas cintas de trabajo. En otras palabras, es el número de celdas ocupadas por símbolos de Σ en la configuración C.

vi) Dada una palabra $x \in L(M) \subseteq \Sigma^*$, llamaremos espacio–memoria consumido por M sobre x al máximo de las tallas de las configuraciones intermedias que surgen en el cálculo de M que comienza en la configuración incial en x ($I(x)$) y termina en alguna configuración final aceptadora C. Se denota esta cantidad mediante $S_M(x) \in \mathbb{N}$.

Observación 9.4.1. Es conveniente insistir en las frases "alguna configuración final aceptadora" usada en las defincciones de T_M y S_M. Dada una palabra $x \in \Sigma^*$ y dada la configuración inicial $I_M(x)$ podemos construir un grafo (potencialmente infinito) que tiene a $I_M(x)$ como raíz: se trata del subgrafo del sistema de transición asociado a M formado por todas las posibles configuraciones alcanzables desde $I_M(x)$. Este grafo es notablemente distinto en el caso determinista e indeterminista. En el primer caso, si $x \in L(M)$ el grafo sólo tiene un camino que comienza en $I_M(x)$ y termina en una (y sólo una) configuración final aceptadora posible (a partir de x). En el caso indeterminista puede haber varios posible sucesores de cada nodo y por tanto un número potencialemente alto (posiblemente infinito) de caminos que comienzan en $I_M(x)$. En el caso indeterminista, para $x \in L(M)$, hay algún camino finito que alcanza alguna configuración final. Para contabilizar el tiempo y/o el espacio buscaremos el mejor camino posible entre aquellos que terminan en una configuración final aceptadora.

A partir de las funciones T_M y S_M se definen dos tipos de funciones de complejidad.

Definición 9.4.2 (Complejidad del Caso Peor)**.** Sea M una máquina de Turing sobre el alfabeto Σ. Definimos las funciones siguientes :

i) Función de Tiempo : $T_M : \mathbb{N} \longrightarrow \mathbb{R}_+$ dada mediante :

$$t_M(n) := max\{T_M(x) \ : \ x \in L(M), \ |x| \leq n\},$$

es el "peor" de los tiempos de todos los inputs representables con, a lo sumo, n bits.

ii) Función de Espacio : $s_M : \mathbb{N} \longrightarrow \mathbb{R}_+$ dada mediante :

$$s_M(n) := max\{S_M(x) \ : \ x \in L(M), \ |x| \leq n\},$$

es el "peor" de los espacios de todos los inputs representables con, a lo sumo, n bits.

Observación 9.4.2. En ocasiones se utilizan las funciones de complejidad en promedio para el análisis del comportamiento de algoritmos; pero no incluiremos aquí esa discusión.

A partir de estas funciones se clasifican los problemas computables en clases de complejidad. Obsérvese que comenzamos estudiando las clases de complejidad de los lenguajes sobre un alfabeto, es decir, clasificamos complejidades de problemas decisionales.

Definición 9.4.3. Sea Σ un alfabeto de cardinal al menos 2 Sea $f : \mathbb{N} \longrightarrow \mathbb{R}_+$ una función monótona creciente. definiremos las clases :

i) $DTIME_\Sigma(f) := \{L \subseteq \Sigma^* \ : \ \exists$ una máquina de Turing determinista M
verificando $L = L(M), t_M(n) \leq f(n), \ \forall n \in \mathbb{N}\}$

ii) $DSPACE_\Sigma(f) := \{L \subseteq \Sigma^* \ : \ \exists$ una máquina de Turing determinista M
verificando $L = L(M), s_M(n) \leq f(n), \ \forall n \in \mathbb{N}\}$

iii) $NTIME_\Sigma(f) := \{L \subseteq \Sigma^* \ : \ \exists$ una máquina de Turing (determinista o no) M
verificando $L = L(M), t_M(n) \leq f(n), \ \forall n \in \mathbb{N}\}$

iv) $NSPACE_\Sigma(f) := \{L \subseteq \Sigma^* \ : \ \exists$ una máquina de Turing (determinista o no) M
verificando $L = L(M), s_M(n) \leq f(n), \ \forall n \in \mathbb{N}\}$

9.5. Rudimentos con Máquinas de Turing

9.5.1. La independencia del alfabeto.

En esta Subsección vamos a discutir el comportamiento de las clases de lenguajes recursivamente enumerables y recursivos, cuando se efectúa una simple modificación del alfabeto sobre el que están definidos.

Definición 9.5.1. Sean Σ, τ dos alfabetos finitos. Llamaremos sistema de codificación de caracteres (o, a veces, sistema de codificación de caracteres de símbolos) de longitud (r, s) entre los alfabetos Σ y τ a toda aplicación inyectiva

$$\sigma : \Sigma^r \longrightarrow \tau^s.$$

Observación 9.5.1. i) Dados cualesquiera dos lenguajes Σ y τ, tales que $\sharp(\tau) \geq 2$, y dado $r \in \mathbb{N}$ siempre existe al menos un sistema de codificación de ceracteres de longitud (r, s) que relaciona las palabras de longitud r de Σ con palabras de longitud s de τ, par cualquier $s \in \mathbb{N}$ satisfaciendo

$$s \geq rmax\{log_2\sharp(\Sigma), 1\}.$$

La razón es simple : dado que $\sharp(\tau) \geq 2$, entonces,

$$\sharp(\tau)^s \geq 2^s \geq 2^{rlog_2\sharp(\Sigma)} = \sharp(\Sigma^r).$$

Entonces, por la defincón de cardinal, el cardinal del conjunto τ^s es mayor que el cardinal de Σ^r si sy solamente si existe una aplicación inyectiva $\sigma : \Sigma^r \longrightarrow \tau^s$.

ii) Obsérvese que la afirmación anterior no es cierta cuando τ es un alfabeto unario. Por eso trataremos de trabajar siempre con alfabetos cuyo acrdinal sea, al menos, 2.

iii) Uno hubiera estado tentado de definir un sistema de codificación de caracteres no con exacta medida de la imagen sino con imagen acotada, esto es, como aplicaciones inyetivas :

$$\sigma \ : \Sigma^r \longrightarrow \bigcup_{i=1}^{s} \tau^i.$$

Sin embargo, esta noción no es del todo conveniente para lo que sigue.

iv) Obsérvese que si Σ es un alfabeto finito, podemos considerar el alfabeto Σ^r y considerar el conjunto de todas las palabras sobre este segundo alfabeto, esto es,

$$(\Sigma^r)^* .$$

Ahora bien, este conjunto de palabras también puede identificarse con un lenguaje sobre el alfabeto Σ del modo obvio siguiente :

$$(\Sigma^r)^* = \{x \in \Sigma^* \ : \ |x| \in r\mathbb{N}\} \subseteq \Sigma^*.$$

En particular, $(\Sigma^r)^*$ es, con la operación de adjunción un submonoide de Σ^*.

v) El sistema de codificación de caracteres más conocido es el sistema de codificación de caracteres asociado al uso de los "bytes" como nunidad de la informática. Un byte es una lista de 8 bits, es decir, 8 símbolos sobre el alfabeto $\{0, 1\}$. La razón del uso de 8 bits, tiene su origen en la introducción del códugo ASCII (American Standard Code for Information Interchange). Su origen se remonta a 1960, cuando se diseña un sistema de codificación de caracteres, basado en el alfabeto inglés (de ahí ciertas ficultades con la "ñ" o la "ç", por ejemplo) que codifica 128 caracteres (los números 0 a 9, las letras a-z y sus mayúsculas, algunos símbolos de puntuación y algunos símbolos originarios de los teletipos) mediante listas de 7 bits (i.e. un sistema de codificación de caracteres con símbolos en $\{0, 1\}^7$). Casi simultáneamente, IBM desarrolló su variante Extended Binary Coded Decimal Interchange Code (EBCDIC). Aunque sustancialmente diferentes, la preeminencia de las máquinas de IBM hizo que el código basado en 8 bits quedara como un estándar, equiparable a ASCII. La variante de ASCII de 8 dígitos, conocido como UFT-8, se convierte en estándar a partir del año 2007.

Proposición 9.5.1. *Sean Σ, τ dos alfabetos finitos y sea $\sigma \ : \Sigma^r \longrightarrow \tau^s$ un sistema de codificación de caracteres de longitud (r, s). Entonces, existe un único monomorfismo de monoides $\tilde{\sigma} \ : (\Sigma^r)^* \longrightarrow \tau^*$ tal que :*

i) $\tilde{\sigma} \mid_{\Sigma^r} = \sigma$,

ii) para cada $x \in (\Sigma^r)^*$, $|\tilde{\sigma}(x)| = \frac{s}{r}|x|$, donde $|x|$ es la longitud de x como palabra sobre el alfabeto Σ.

Dado cualquier lenguaje $L \subseteq (\Sigma^r)^*$, llamaremos traducción de L mediante el sistema de codificación de caracteres σ al lenguaje $\tilde{\sigma}(L) \subseteq \tau^*$.

Demostración. Bastará con que definamos la aplicación :

$$\tilde{\sigma} \ : (\Sigma^r)^* \subseteq \Sigma^* \longrightarrow \tau^*,$$

mediante : Sea $x = x_1 \cdots x_n \in (\Sigma^r)^*$, donde $x_i \in \Sigma^r$. Definamos

$$\tilde{\sigma}(x) := \sigma(x_1) \cdots \sigma(x_n)$$

y $\tilde{\sigma}(\lambda) = \lambda \in \tau^*$. Obsérvese que esta aplicación verifica las propiedades deseadas. Además

$$|\tilde{\sigma}(x)| = sn = \frac{s}{r}|x|,$$

puesto que la talla de x como palabra sobre el alfabeto Σ es, obviamente, rn. La unicidad se sigue del hecho de ser morfismo de monoides. □

Observación 9.5.2. Obsérvese que las aplicaciones inyectivas $\sigma : \Sigma^r \longrightarrow \bigcup_{i=1}^{s} \tau^i$ no necesariamente han de definir un monomorfismo de monoides como el anterior. Un contraejemplo puede venir dado por el siguiente :

$$\sigma : \{0,1\} \longrightarrow \{a,b\}^2 \cup \{a,b\},$$

dada mediante :

- $\sigma(0) = a$,
- $\sigma(1) = aa$

Claramente no hay ningn monomorfismo de monoides $\tilde{\sigma} : \{0,1\}^* \longrightarrow \{a,b\}^*$, tal que $\tilde{\sigma}(0) = a$ y $\tilde{\sigma}(1) = aa$. Si hubiera algún morfismo de monoides con tal propiedad, tendramos

$$\tilde{\sigma}(00) = \tilde{\sigma}(0)\tilde{\sigma}(0) = aa = \tilde{\sigma}(1).$$

En tal caso $\tilde{\sigma}(00) = \tilde{\sigma}(1)$ y $\tilde{\sigma}$ no sera una aplicación inyectiva. El resultado anterior, también tiene su recíproco.

Proposición 9.5.2. *Sean Σ y τ dos alfabetos finitos. Supongamos $\varphi : (\Sigma^r)^* \longrightarrow \tau^*$ un monomorfismo de monoides tal que $|\varphi(x)| = \frac{s}{r}|x|$, para toda palabra $x \in (\Sigma^r)^*$, donde $|x|$ es la longitud de x como palabra sobre el alfabeto Σ. Entonces, existe un único sistema de codificación de caracteres*

$$\sigma : \Sigma^r \longrightarrow \tau^s,$$

tal que $\tilde{\sigma} = \varphi$.

Demostración. En virtud de los propuesto, se tendrá que

$$\varphi(\Sigma^r) \subseteq \tau^s.$$

Entonces, definiendo

$$\sigma := \varphi\,|_{\Sigma^r} : \Sigma^r \longrightarrow \tau^s,$$

tendremos el resultado apetecido. □

A partir de ahora nos interesa la relación existente entre los lenguajes recursivamente enumerables y sus traducciones. El primero de los resultados es un proceso de "simulación" de máquinas de Turing que expondremos en la siguiente Proposición.

Proposición 9.5.3. *Sean Σ y τ dos alfabetos y sea $\sigma : \Sigma \longrightarrow \tau^s$ un sistema de codificación de caracteres de longitud $(1,s)$. Sea $M := (\Sigma, Q, q_0, F, \delta)$ una máquina de Turing sobre el alfabeto τ. Sean $L(M) \subseteq \Sigma^*$ el lenguaje aceptado por M y sea*

$$Res_M : L(M) \subseteq \Sigma^* \longrightarrow \Sigma^*$$

la función recursiva evaluada por M. Entonces, existe una máquina de Turing $M_1 := (\tau, Q', q_0', F', \delta')$ sobre el alfabeto τ que verifica las siguientes propiedades :

i) $L(M_1) := \tilde{\sigma}(L(M))$,

ii) $Res_{M_1}(x) := \tilde{\sigma}(Res_M(\tilde{\sigma}^{-1}(x))$, *para todo* $x \in L(M_1)$.

iii)

$$t_{M_1}(n) \leq O(n + st_M(\lfloor \frac{n}{s} \rfloor)).$$

$$s_{M_1}(n) \leq O(ss_M(\lfloor \frac{n}{s} \rfloor)).$$

Demostración. Lo que pretendemos hacer es simplemente la simulación del comportamiento de la máquina M sobre el alfabeto τ como si estuviera trabajando con las imágenes de los símbolos de Σ. Para ello, la máquina M_1 actuará como sigue :

INPUT $x \in \tau^*$,
Si $x \notin \tilde{\sigma}(\Sigma)$, RECHAZAR x
else SIMULA M sobre $\tilde{\sigma}^{-1}(x)$,
endif
OUTPUT $\tilde{\sigma}(Res_M(\tilde{\sigma}^{-1}(x))$.
END

9.5.1.1. Primera Parte : Rechazar las Palabras que no están en la Imagen.

La primera parte del proceso es sencilla de modelizar. La máquina va leyendo el input $x \in \tau^*$ y avnazando hasta que encuentra el símbolo λ de celda vacía. Si en su avance no ha encontrado ninguna sucesión de s símbolos de τ que no correspondan a imágenes de símbolos de Σ, acepta, volviendo la unidad de control a la posición inicial y procediendo a la Simulación. Si alguna vez encuentra una sucesión de s símbolos de τ que no están en $\sigma(\Sigma)$ entonces, rechaza el input.
Esta primera fase se puede modelizar como sigue: Definamos un pimer espacio de estados

$$Q_1' := \left[q_0 \times \left(\bigcup_{i=0}^{s} \{j\} \times \tau^j \right) \bigcup \{\mathbf{R}, \mathbf{Error}\} \right].$$

El estado inicial de la máquina M_1 ser el siguiente:

$$q_0' := (q_0(0, \lambda)) \in Q_1'.$$

La máquina M_1 en esta primera fase procederá acumulando en la unidad de control (es decir, en el estado) una cantidad finita de información. Esta cantidad finita de información es el contenido de s celdas sucesivas de la cinta de input. Esas s celdas sucesivas, serán la clave para aceptar o no la palabra dada.
El comportamiento de la función de transición en esta primera fase será como sigue. Vamos a indicar con $\trianglerighteq := (\triangleright, \ldots, \triangleright) \in \{\triangleright\}^k$. Así, tratamos de definir el valor :

$$\delta'(q'; \alpha, w)$$

del modo siguiente :

- Si $w \neq \trianglerighteq$, rechazar pasando al caso de **Error**.
- En otro caso, si $q' = (q_0, (0, \lambda))$, hacer lo siguiente :

  $$\delta'((q_0, (0, \lambda)); \triangleright, \trianglerighteq) = ((q_0, (0, \lambda)); \trianglerighteq, +1, 0, \ldots, 0)$$

  (esto es, en estado inicial, si leemos $\triangleright$ avanza la cinta de input, no muevas las demás y no cambies ningún símbolo).
- Si, $q' = (q_0, (0, \lambda))$ y $\alpha \in \tau$, hacer lo siguiente :

  $$\delta'(q'; \alpha, \trianglerighteq) = ((q_0, (1, \alpha)); \trianglerighteq, +1, 0, \ldots, 0),$$

  esto es, acumula el símbolo α con su longitud en la unidad de control y avanza la cinta de input, dejando las demás como están.

- Si $q' = (q_0, (i, \theta))$, con $\theta \in \tau^i$ y $1 \leq i \leq s-1$, y si $\alpha \neq \lambda$, con $\alpha \in \tau$, definimos

$$\delta'(q'; \alpha, \trianglerighteq) = (q_0, (i+1, \theta\alpha)); \trianglerighteq; +1, 0, \ldots, 0),$$

esto es, guarda la palabra $\theta\alpha \in \tau^{i+1}$, avanza un paso la cinta de input y no hagas nada con las demás.

- Si $q' = (q_0, (i, \theta))$, con $\theta \in \tau^i$ y $1 \leq i \leq s-1$, y si $\alpha = \lambda$, con $\alpha \in \tau$, definimos

$$\delta'(q'; \alpha, \trianglerighteq) = (\mathbf{Error}; \trianglerighteq; 0, 0, \ldots, 0),$$

Si aparece un símbolo λ, esto quiere decir que la longitud de x no puede ser un múltiplo de s, luego x no puede estar en $\tilde{\sigma}(\Sigma)$ y, por tanto, rechazamos.

- Si $q' = (q_0, (s, \theta))$, con $\theta \in \tau^s$ y si $\theta \in \sigma(\Sigma)$, y $\alpha = \lambda$ definimos

$$\delta'(q'; \alpha, \trianglerighteq) = (\mathbf{R}; \trianglerighteq; -1, 0, \ldots, 0),$$

esto es, la palabra leída x pertence efectivamente a $\tilde{\sigma}(\Sigma)$ y pasamos a la fase determinada por el estado $\mathbf{R}$ que será una fase de **Retroceso** hasta el inicio de la cinta.

- Si $q' = (q_0, (s, \theta))$, con $\theta \in \tau^s$ y si $\theta \in \sigma(\Sigma)$, y $\alpha \neq \lambda$ definimos

$$\delta'(q'; \alpha, \trianglerighteq) = ((q_0, (1, \alpha); \trianglerighteq; +1, 0, \ldots, 0),$$

esto es, los últimos s símbolos de τ leidos están en $\sigma(\Sigma)$, luego hasta aquí vamos bien sigamos con los s símbolos siguientes.

- Si $q' = (q_0, (s, \theta))$, con $\theta \in \tau^s$ y si $\theta \notin \sigma(\Sigma)$,

$$\delta'(q'; \alpha, \trianglerighteq) = (\mathbf{Error}; \trianglerighteq; +1, 0, \ldots, 0),$$

esto es, los últimos s símbolos leidos no están en $\sigma(\Sigma)$, luego no podemos aceptar esta palabra.

9.5.1.2. Segunda Parte : Simulación de Un Paso (SimUP).

Terminada esta primera fase, pasamos a la fase de simulación propiamente dicha. Para entenderla, supongamos que ya hemos definido un conjunto de estados Q_2' que corresponden al proceso de simulación. Entonces, trataremos de identificar las configuraciones de la máquina M con las configuraciones de la máquina M_1 que estamos tratando de construir. Comencemos una definición.

Definición 9.5.2. Para ello, sea $C \in S_M$ una configuración de la máquina M sobre el alfabeto Σ. Tendremos

$$C := (q; \triangleright x, \triangleright w_1, \ldots, \triangleright w_{k+1}; n_0, \ldots, n_{k+1}) \in S_M,$$

donde $x, w_1, \ldots, w_{k+1} \in \Sigma^*$. Llamaremos configuración traducida de la configuración C sobre el alfabeto τ a través del sistema de codificación de caracteres $\sigma : \Sigma \longrightarrow \tau^s$ a la configuración siguiente :

$$\tilde{\sigma}(C) = C' := (q'; \triangleright x', \triangleright w_1', \ldots, \triangleright w_{k+1}'; n_0' \ldots, n_{k+1}') \in S_{M_1},$$

tal que se verifican las propiedades siguientes :

- $q' = q \in Q_2'$,
- $x' := \tilde{\sigma}(x) \in \tau^*$,
- para cada i, $1 \leq i \leq k+1$, $w_i' := \tilde{\sigma}(w_i) \in \tau^*$.

- Para cada i, $0 \leq i \leq k+1$, si $n_i = 0$, entonces $n'_i = 0$ (esto es, si se encuentran mirando el cursor $\triangleright$ deben estar haciéndolo tanto las unidades de control de la máquina original M como las unidades de control de la máquina nueva M_1).

- Para cada i, $1 \leq i \leq k+1$, si $n_i = |w_i| + 2$, entonces, $n'_i = 2 + |w'_i|$ (esto, es si la unidad de control de una cinta de trabajo de la máquina M se encuentra en la celda vacía inmediatamente siguiente de palabra correspondiente, la misma posición debe tener la unidad de control de la máquina M_1 correspondiente.

- Si $n_0 = |x| + 2$, entonces, $n'_0 = 2 + |x'|$ (mismo comentario, pero con respecto a la cinta de input)

- Para cada i, $1 \leq i \leq k+1$ tal que $2 \leq n_i \leq |w_i| + 1$, entonces $s(n_i - 1) + 2 = n'_i$.

- Mismas consideraciones para las posiciones de las unidades de control sobre la cinta de input, esto es, si $2 \leq n_0 \leq |x| + 1$, entonces, $s(n_0 - 1) + 2 = n'_0$.

La máquina M_1 ha de ser una máquina tal que las traducciones de las configuraciones de M sean configuraciones de M_1. En este sentido, queda claro que el espacio de estados de M_1 debe contener al espacio de estdos de M.
En esencia el proceso de simulación de un paso debe funcionar del modo siguiente :

- Debemos hacer que las cintas de trabajo y la cinta de input de la máquina M_1 tengan sus objetos divididos en *bloques de talla s* (esto es, bloques que contienen s símbolos sobre el alfabato τ).

- Cada bloque de s símbolos de la máquina M_1 debe ser la traducción de un símbolo de Σ a través de σ.

- Para simular un paso de la máquina M mediante la máquina M_1 haremos lo que sigue. En primer lugar, comenzaremos por la traducción $\tilde{\sigma}(C)$ de una configuración C de la máquina M (esto es $C \in S_M$). Depes haremos cálculos sobre $\tilde{\sigma}(C)$ hasta obtener una nueva configuración $C'' := \mathbf{SimUP}(\tilde{\sigma}(C)$ sobre la máquina M_1. Esta nueva configuración C'' debe verificar
$$C'' = \tilde{\sigma}(C_1),$$
donde $C \to_M C_1$.

Lo que pretendemos es definir ese proceso **SimUP** que simula en M_1 un paso de cálculo de la máquina M. Lo haremos del modo siguiente :

9.5.1.2.1. Definición del Proceso de Simulación de un Paso. Para hacerlo, dividiremos el proceso de **SimUP** (Simulación de un paso) en cuatro fases.

i) **Lectura** La máquina M_1 se encuentra leyendo el primero de los símbolos de un bloque de s símbolos. En tal caso, lee de izquierda a derecha los s símbolos siguientes acumulando esa información en la correspondiente unidad de control.

ii) **Transición**. Con la información contenida en la Unidad de Control, puede, realizar el proceso de transición que hubiera hecho M, guardando el resultado de esa transición, de nuevo, en la Unidad de Control.

iii) **Escritura.** Ahora en s pasos, copia los símbolos pertinentes, contenidos en la unidad de control, en las s celdas a la izquierda de la posición que ocupaba cuando terminó de leer y terminó la transición. Obsérvese que, al mismo tiempo que escribe hacia la izquierda, retrocede $s - 1$ pasos hasta volver a su posición inicial. Se puede obligar (y de hecho se obliga) a la Unidad de Control de la cinta de Input a retroceder a lo largo del bloque de s símbolos, hasta la posición que tenía al principio de la simulación.

iv) **Movimientos**. Una vez realizada la escritura, ahora debe hacer los movimientos, esto es un paso a la izquierda de M son s pasos a la izquierda de M_1, un paso a la derecha de M son s pasos a la derecha de M_1.

Para modelizar estas cuatro etapas de un proceso de simulación, construiremos cuatro conjuntos que forman el espacio de estados, cada uno de la fase correspondiente :

$$Q_2' := Q_{2,L} \bigcup Q_{2,T} \bigcup Q_{2,E} \bigcup Q_{2,M}.$$

Para simplificar la descripción de la acción de la función de transición δ' sobre estos espacios de estados, supondremos que nuestra máquina M tiene una única cinta de trabajo. El caso con k cintas de trabajos es análogo.

Los estados en la fase de Lectura . Definiremos

$$Q_{2,L} := Q \bigcup Q \times (Q \times (j \times \tau^2))$$

La acción de la función de transición δ' en esa fase de Lectura vendrá dada del modo siguiente :

$$\delta'(q'; \alpha, \beta),$$

es dada mediante :

- Si $q' \in Q$, nos preguntamos si $q' \in F$, en ese caso termina la computación. En caso de que $q' \in Q \setminus F$, se procede del modo siguiente :

  $$\delta'(q'; \alpha, \beta) = ((q', (1, (\alpha, \beta))); \beta; +1, +1, 0)$$

  (esto es, acumulamos los símbolos en la unidad de control y comenzamos a avanzar a la derecha.

- Si $q' := (q, (i, \theta, \theta')))$ con $1 \leq i \leq s-1$, $\theta, \theta' \in \tau^i$, $\alpha, \beta \in \tau$.

  $$\delta'(q'; \alpha, \beta) = ((q, (i+1, (\theta\alpha, \theta'\beta)), \beta, +1, +1, 0),$$

  donde $\theta\alpha, \theta'\beta \in \tau^{i+1}$ (esto es, si aún no hemos dado s pasos a la derecha, acumulemos en la unidad de control, la información descrita en las celdas que estamos leyendo, y sigamos leyendo hacia la derecha).

- Si $q' := (q, (s, \theta, \theta')))$ con $\theta, \theta' \in \tau^s$, terminamos la fase de lectura y pasamos a la fase de transición, esto es,

  $$\delta'(q'; \alpha, \beta) = ((\mathbf{T}, (q, (\theta\alpha, \theta'\beta)), \beta, 0, 0, 0),$$

Los estados en la fase de Transición . Definiremos

$$Q_{2,T} := \left[\mathbf{T} \times \left(Q \times (\tau^s)^2\right) \bigcup \left(\mathbf{E} \times Q \times (\tau^s)^2 \times \{-1, 0, +1\}^3\right)\right].$$

La fase de transición tiene una acción simple ; a través de los contenidos de la unidad de control, ejecuta la función de transición ya pasa a la fase de ESCRITURA, esto es, si $q' := (\mathbf{T}, q, (\theta_1, \theta_2))$, con $\theta_i \in \tau^2$,

$$\delta'(q', \alpha, \beta) = (q'', \beta, 0, 0, 0),$$

donde

$$q'' := (\mathbf{E}, q_1, (\omega_1, \omega_2), (\varepsilon_0, \varepsilon_1, \varepsilon_2)),$$

de tal modo que :

$$\delta(q, \sigma^{-1}(\theta_1), \sigma^{-1}(\theta_2)) = (q_1; \sigma^{-1}(\omega_1), \sigma^{-1}(\omega_2); \varepsilon_0, \varepsilon_1, \varepsilon_2)$$

Los estados en la fase de Escritura . Definiremos

$$Q_{2,E} := (\mathbf{E} \times Q) \times \left[(\tau^s)^2 \cup \left(\bigcup_{j=1}^{s} \{j\} \times \left(\tau^j\right)^2 \right) \right] \times \{-1,0,+1\}^3 \bigcup \left[\mathbf{M} \times Q \times \{-1,0,+1\}^3 \right].$$

En esta fase, con movimientos hacia la izquierda, irá trasladando los contenidos de las unidades de control a las cintas de trabajo y a la cinta de output. Así pues, tratamos de definir

$$\delta'(q', \alpha, \beta),$$

cuando $q' \in Q_{2,E}$. Lo haremos del modo siguiente :

- Supongamos $q' := (\mathbf{E}, q, (\omega_1, \omega_2), \underline{\varepsilon})$ donde $\omega_1, \omega_2 \in \tau^s$, y $\underline{\varepsilon} := (\varepsilon_0, \varepsilon_1, \varepsilon_2 \in \{-1,0,+1\}^3$. Entonces,
$$\delta'(q', \alpha, \beta) = (q'', \beta; 0, 0, 0),$$
donde q'' indica que empieza la fase de escritura y es dado por :
$$q'' := (q, (s, (\omega_1, \omega_2)), \underline{\varepsilon}).$$

- Supongamos $q' := (q, (i, (\omega_1, \omega_2)), \underline{\varepsilon})$ con $2 \leq i \leq s$, $\omega_j \in \tau^i$, para $j = 1, 2$. En ese caso la máquina incopora a las cintas de trabajo y la cinta de output, el último dígito contenido en ω_1 y ω_2, esto es
$$\delta'(q', \alpha, \beta) = (q'', \theta_1, \theta_2; -1, -1, -1),$$
donde
$$q'' := (q, (i-1, (\theta_1', \theta_2')), \underline{\varepsilon})$$
de tal modo que
$$\theta_1' \omega_1 = \theta_1, \quad \theta_2' \omega_2 = \theta_2$$
Obsérvese que
$$|\theta_{j,1}'| = i - 1$$
para $j = 1, 2$ y que la cinta de input también aprovecha para ir retrocediendo al mismo ritmo que las cintas de trabajo.

- Supongamos $q' := (q, (1, (\omega_1, \omega_2)), \underline{\varepsilon})$, donde $\omega_1, \omega_2 \in \tau$. En eseta caso termina la escritura y pasa a la fase de movimientos, esto es,
$$\delta'(q', \alpha, \beta) = (q''; \omega_1, \omega_2; 0, 0, 0),$$
donde
$$q'' := (\mathbf{M}, q, \underline{\varepsilon}).$$
Obsérvese que las unidades de control al final de la fase de escritura se encuentran en la misma posición inicial en la que estaban al cominezo de la simulación.

Los estados de la Fase de Movimientos. Definiremos :

$$Q_{2,M} := \{\mathbf{M}\} \times Q \times \{-1,0,+1\}^3 \bigcup Q \times \left(\bigcup_{j=1}^{s} \{j\} \times \{-1,0,+1\}^3 \right).$$

La única idea esencial es realizar los movimientos que indiquen los índices que vienen ya desde la transición. Para ello debemos contar los pasos dados y los que nos restan. Una vez hechos los movimientos, terminamos el proceso de simulación de un paso, volviendo a la fase de LECTURA. Esto es, dado $q' \in Q_{2,M}$, definiremos :

$$\delta'(q', \alpha, \beta),$$

medienta las reglas siguientes :

- Supongamos $q' := (\mathbf{M}, q, , \varepsilon_0, \varepsilon_1, \varepsilon_2) \in Q_{2,M}$. Entonces,

$$\delta'(q', \alpha, \beta) = (q'', \beta; 0, 0, 0),$$

donde

$$q'' := (q, s, (\varepsilon_0, \varepsilon_1, \varepsilon_2)) \in Q_{2,M}$$

(esto es, indicamos que empieza la fase de movimientos y que aún debemos dar s pasos.

- Supongamos $q' := (q, i, (\varepsilon_0, \varepsilon_1, \varepsilon_2)) \in Q_{2,M}$, con $2 \leq i \leq s$. Entonces,

$$\delta'(q', \alpha, \beta) = (q'', \beta; \varepsilon_0, \varepsilon_1, \varepsilon_2),$$

donde

$$q'' := (q, i-1, (\varepsilon_0, \varepsilon_1, \varepsilon_2)) \in Q_{2,M}.$$

Obsérvese que el nuevo estado indica que nos quedan $i-1$ movimientos que hacer y que los movimientos que se hacen en cada caso, son exactamente, los movimientos indicados por la terna $\varepsilon_0, \varepsilon_1, \varepsilon_2$.

- Supongamos $q' := (q, 1, (\varepsilon_0, \varepsilon_1, \varepsilon_2)) \in Q_{2,M}$. Entonces,

$$\delta'(q', \alpha, \beta) = (q, \beta; \varepsilon_0, \varepsilon_1, \varepsilon_2).$$

Esto es, hemos terminado de hacer todos los movimientos y volvemos a iniciar la fase de lectura.

Definición 9.5.3. Definiremos $\mathbf{SimUP}(C') = C'' \in S_{M_1}$ como la configuración resultante de aplicar a C' el proceso de simulación de un paso combinando las operaciones de LECTURA, TRANSICION, ESCRITURA y MOVIMIENTOS descritas más arriba.

Lema 9.5.4. *Con las notaciones anteriores, sea $C \in S_M$ es una configuración de la máquina M y sea $\tilde{\sigma}(C) = C' \in S_{M_1}$ la traducción de C a la máquina M_1. Entonces, para toda configuración $C_1 \in S_M$ tal que $C \to_M C_1$ se tiene :*

$$\mathbf{SImUP}(C') = C_1,$$

esto es, el proceso **SimUP** *simula un paso de cálculo de M.*

Demostración. Ejercicio de comprobación utilizando las definiciones ya expuestas. □

Demostración de la Proposición 9.5.3.– Para demostrar la Proposición 9.5.3 baste con observar que tenemos ya descrita el proceso **SimUP** con su espacio de estados, su función de transición, su estado inicial y sus estados finales aceptadores, que son el subconjunto $F \subseteq Q'$. El Lema previo señala que la simulación se comporta igualmente que la máquina M inicial, pero sobre el nuevo alfebato. De otro lado, señalemos que las operaciones necesarias para ejecutar una vez **SimUP** son del orden $O(s)$ con lo que las estimaciones de tiempo y espacio de M_1 se siguen. □

Con este resultado de simulación, podemos mostrar las siguientes consecuencias.

Corollario 9.5.5. *Sean Σ, τ dos alfabetos finitos y sea $\sigma : \Sigma^r \longrightarrow \tau^s$ un sistema de codificación de caracteres de longitud (r, s). Sea $L \subseteq (\Sigma^r)^*$ un lenguaje recursivamente enumerable y sea*

$$f : L \subseteq (\Sigma^r)^* \longrightarrow (\Sigma^r)^*,$$

una función recursivamente enumerable. Supongamos que existe una máquina de Turing M sobre el alfabeto Σ tal que $L(M) = L$ y $Res_M = f$. Entonces, existe una máquina de Turing M_1 sobre el alfabeto τ tal que

i) $L(M_1) := \tilde{\sigma}(L)$,

ii) $Res_{M_1} : L(M_1) \subseteq \tau^* \longrightarrow \tau^*$ *es tal que, para todo* $y := \tilde{\sigma}(x) \in L(M_1) = \tilde{\sigma}(L)$ *se tiene :*

$$Res_{M_1}(y) := \tilde{\sigma}(f(x)) \in \tau^*.$$

iii)

$$t_{M_1}(n) \leq O(n + st_M(\lfloor \frac{sn}{r} \rfloor),$$

$$s_{M_1}(n) \leq O(ss_M(\lfloor \frac{sn}{r} \rfloor),$$

donde n es la longitud del input medida en trminos de τ.

Demostración. Basta con hacer la simulación en el caso en que el primer alfabeto sea Σ^r. El resto se sigue de lo probado en la Proposición 9.5.3. □

Observación 9.5.3. Los anteriores resultados nos dicen que si un lenguaje sobre un alfabeto es recursivamente enumerable, también lo es cualquier traducción suya. Nos proponemos ahora discutir el recíproco.

Proposición 9.5.6. *Sean Σ, τ dos alfabetos finitos $\sharp(\tau) \geq 2$ y sea $\sigma : \Sigma^r \longrightarrow \tau^s$ un sistema de codificación de caracteres de longitud (r, s). Sea $L \subseteq (\Sigma^r)^*$ un lenguaje y sea*

$$f : L \subseteq (\Sigma^r)^* \longrightarrow (\Sigma^r)^*,$$

una función. Consideremos el lenguaje $\tilde{sigma}(L) \subseteq \tau$ y la función

$$\varphi : \tilde{\sigma}(L) \subseteq (\tau^s)^* \longrightarrow (\tau^s)^*,$$

dada mediante la propiedad siguiente : Para cada $y = \tilde{\sigma}(x) \in \tilde{\sigma}(L)$

$$\varphi(y) := \tilde{\sigma}(f(x)).$$

Supongamos que existe una máquina de Turing M sobre el alfabeto τ tal que

- $L(M) = \tilde{\sigma}(L) \subseteq \tau^*$,
- $Res_M = \varphi$.

Entonces, existe una máquina de Turing M_1 sobre el alfabeto Σ tal que

i) $L(M) = L$,

ii) $Res_M = f$,

iii)

$$t_M(n) \leq O(kn + kt_M(\lfloor \frac{rn}{s} \rfloor),$$

$$s_M(n) \leq O(kn + ks_M(\lfloor \frac{rn}{s} \rfloor),$$

donde $k \geq log_2\sharp(\Sigma)$.

Demostración. La idea esencial de la máquina M_1 funciona como sigue. Sea $\rho : \tau \longrightarrow \Sigma^k$, un sistema de codificación de caracteres. La existencia de este sistema de codificación de caracteres está garantizada por la propiedad $\sharp(\tau) \geq 2$ y existe siempre que $k \geq log_2\sharp(\Sigma)$.. Aplicando la Proposición 9.5.3, existirá una máquina de Turing M_0 sobre el alfabeto Σ tal que M_0 acepta el lenguaje $\tilde{\rho}(L(M))$ y, además, se ha de verificar :

- Para cada $z \in \Sigma^*$, si $z := \tilde{\rho}(y) \in \tilde{\rho}(L(M))$,

$$Res_{M_0}(z) = \tilde{\rho}(\varphi(y)).$$

-

$$t_{M_0}(n) \leq O(n + kt_M(\lfloor \frac{n}{k} \rfloor),$$

$$s_{M_0}(n) \leq O(ks_M(\lfloor \frac{sn}{k} \rfloor).$$

Ahora bastará con que hagamos el proceso siguiente para tener el rersultado apetecido :

INPUT $x \in \Sigma^*$
INCIALIZA
    RECHAZA si $x \notin (\Sigma^r)^*$,
    (usando WT_1) TRADUCE $z := \tilde{\rho}(\tilde{\sigma}(x)) \in \Sigma^*$.
SIMULA (usando las cintas de trabajo necesarias) el cálculo de M_0 sobre $z \in \Sigma^*$.
ACEPTA x si y solamente si M_0 acepta z.
OUTPUT TRADUCE $\tilde{\sigma}^{-1}(\tilde{\rho}^{-1}(Res_{M_0}(z)))$.

Fig. 3.- Definición de la máquina M_1.

La máquina M_1 así definida es esencialmente la máquina M_0 salvo dos etapas. Una primera etapa de "traducción" del input x a una palabra z también sobre el alfabeto Σ y una segunda "traducción" del output de M_0 sobre Σ. En las estimaciones de complejidad tendremos :

- La primera fase de Rechazo cuesta un tiempo $O(n)$.
- La máquina trabaja sobre palabras de longitud n múltiplo de s La primera traducción transforma s símbolos en r símbolos y luego, r símbolos en rk símbolos, así que la primera traducción cuesta un tiempo
$$O(rk\lfloor\frac{n}{s}\rfloor).$$
- La simulación de la máquina M_0 trabaja sobre un input de tamaño $rk\lfloor\frac{n}{s}\rfloor$. Usando las estimaciones, tendremos que la fase de simulación nos cuesta :
$$O(n + kt_M(rk\frac{n}{s}\frac{1}{k}) = O(n + kt_M(\lfloor\frac{rn}{s}\rfloor).$$
- La traducción es simplemente la traducción de un objeto que ha sido escrito por la máquina de Turing M_0. En particular, su tamaño estar acotado por rl tiempo que tarde M_0 en realizar sus cálculos. Concluimos así que la ltima etata de traducción no cuesta más tiempo que la longitud del output de la simulación de M_0, esto es, estar acotada por :
$$O(n + kt_M(\lfloor\frac{rn}{s}\rfloor).$$

Con estas acotaciones tenemos el tiempo total de la máquina M_1 acotado por :

$$O(n + kt_M(\lfloor\frac{rn}{s}\rfloor).$$

Las acotaciones de espacio son también sencillas. y las dejamos como ejercicio.

9.5.2. Los conjuntos recursivamente enumerables como conjuntos imagen, indecidibilidad y otras interpretaciones

Las palabras "recursivamente enumerables" usadas en la Definición 9.3.2 indican en castellano que si el lenguaje L es recursivamente enumerable si se puede "enumerar". Para entender este concepto, dearrollemos la equivalencia entre conjuntos recurisivamente enumerables y conjuntos imagen de funciones computables.

Definición 9.5.4. Sea $L \subseteq \Sigma^*$ un lenguaje sobre un afabeto finito Σ que contiene al afabeto $\{0,1\}$ (o una identificación suya). Decimos que le lenguaje L es imagen de una función computable si existe una función computable

$$f : \mathbb{N} := 1\{0,1\}^* \cup \{0\} \longrightarrow \Sigma^*,$$

de tal modo que $f(\mathbb{N}) = L$.

En otras palabras, un lenguaje L es imagen de una función recursiva si se puede "enumerar" por una función computable. Es decir, si existe una función computable o una máquina de Turing M que recibe $n \in \mathbb{N}$ y devuelve un valor $Res_M(n) \in \Sigma^*$ que es el $n-$ésimo elemento de L.
Para interpretar esa noción, recuperemos el buen orden de Σ^* que le identifica con el conjunto de los números naturales. La idea es combinar un orden lexiográfco con la longitud. Comencemos fijando un orden en los elementos de Σ. Es decir, si $\Sigma := \{\alpha_1, \ldots, \alpha_m\}$, definamos la ordenación de los elementos de Σ mediante:

$$\alpha_1 < \alpha_2 < \cdots < \alpha_m.$$

Ahora definimos, a partir de ese orden, una ordenación en Σ^* dada mediante la longitud y el leicorgráfico como sigue:
Dados $x := \alpha_{i_1} \ldots \alpha_{i_t}$ e $y = \beta_{j_1} \cdots \beta_{j_s}$ en Σ^* diremos que $x <_{lex} y$ si se verifica:
Sea $m := \min\{k \leq \max\{t,s\} \ : \ \alpha_{i_k} \neq \beta_{j_k}\}$, entonces debe ser cierto:

- o bien $m > t$ (en el caso degenerado $m = t+1$, $\beta_{j_m} \neq \lambda = \alpha_{i_m}$),
- o bien $m \leq t := \min\{s,t\}$ y, en ese caso, $\alpha_{i_m} < \beta_{j_m}$.

Denotaremos el orden lexicográfico mediante:

$$x \leq_{lex} y \Longleftrightarrow [x = y] \vee [x <_{lex} y].$$

Definición 9.5.5 (Longitud más lexicográfico)**.** Dado un alfabeto Σ como el usado anteriormente, definimos el orden "longitud+lexicográfio" sobre Σ^* mediante:

$$x \leq_{length+lex} y \Longleftrightarrow [|x| < |y|] \vee [(|x| = |y|) \wedge (x \leq_{loex} y)]\,,$$

es decir, si x tiene una longitud menor que la y o si amboes tienen la misma longitud, pero x es menor que y con el orden lexicográfico.

Proposición 9.5.7. *La relación $\leq_{length+lex}$ es una relación de orden (reflexiva simétrica y transitiva) que es, además, un buen orden sobre Σ^* (i.e. todo subconjunt no vacío posee mínimo). Más aún, $\leq_{length+lex}$ define una biyección*

$$\psi : \mathbb{N} \longrightarrow \Sigma^*.$$

Demostración. Es obvio y no necesita una discusión relevante.

□

Seguidamente podemos establecer un buen orden y una biyección con los naturales en el producto $\Sigma^* \times \mathbb{N}$. La relación de orden extiende la relación $\leq_{length+lex}$ antes descrita (por eso conservaremos la misma notación) del modo siguiente:
Dados $(x,n),(y,m) \in \Sigma^* \times \mathbb{N}$, diremos que (x,n) es menor que (y,m) para el orden "longitud+ lexicográfico" si se verifica las propiedades siguientes:

$$(x,n) \leq_{legth+lex} (y,m)$$

si y solamente si se verifica:

- O bien $|x| + n < |y| + m$,
- o bien $|x| + n = |y| + m$ y $n < m$,

- o bien $|x| + n = |y| + m$, $n = m$ y $x \leq_{length+lex} y$.

Proposición 9.5.8. *La relación $\leq_{length+lex}$ es una relación de orden que es, además, un buen orden sobre $\Sigma^* \times \mathbb{N}$ (i.e. todo subconjunt no vacío posee mínimo). Más aún, $\leq_{length+lex}$ define una biyección*

$$\Psi : \mathbb{N} \longrightarrow \Sigma^* \times,$$

que es computable. Es decir, existe una máquina de Turing que evalúa la siguiente función:

$$next : \Sigma^* \times \mathbb{N} \longrightarrow \Sigma^* \times \mathbb{N},$$

dada mediante:

$$next(x, n) := min\{(y, m) \in \Sigma^* \times \mathbb{N} \ : \ (x, n) <_{length+lex} (y, m)\}.$$

Demostración. Es un sencillo ejercicio que dejamos al alumno. □

Es decir podemos calcular el "elemento siguiente" a uno dado para la ordenación de $\Sigma^* \times \mathbb{N}$ elegida.
De otro lado, a partir de un lenguaje $L \subseteq \Sigma^*$, podemos introducir la siguiente función:

$$\Phi_L : \Sigma^* \times \mathbb{N} \longrightarrow \{0, 1\},$$

Dada mediante:

$$\Phi_L(x, n) := \begin{cases} 1, & \text{si } x \in L \text{ y } T_M(x) \leq n \\ 0, & \text{en caso contrario} \end{cases}$$

Se trata de un Problema de Decisión. Un *Problema de Decisión* consiste en evaluar una función $\chi : \Sigma^* \longrightarrow \{0, 1\}$. Obviamente los problemas de decisión están ligados a los lenguajes por una obvia biyección: A cada lenguaje $L \subseteq \Sigma^*$ le asociamos su función característica (mal llamada, a veces, índice):

$$\chi_L : \Sigma^* \longrightarrow \{0, 1\},$$

dada mediante:

$$\chi_L(x) := \begin{cases} 1, & \text{si } x \in L \\ 0, & \text{en caso contrario} \end{cases}$$

Definición 9.5.6 (Lenguajes Decidibles)**.** Un lenguaje $L \subseteq \Sigma^*$ se denomina decidible si su función característica χ_L es computable.

Proposición 9.5.9. *Si $L \subseteq \Sigma^*$ es un lenguaje recurisvamente enumerable, la función Φ_L anterior es computable o, equivalenetemente, el lenguaje $\mathcal{L} := \Phi_L^{-1}(\{1\})$ es decidible.*

Demostración. Sea M la máquina de Turing que acepta el lenguaje L, esto es, tal que $L := L(M)$. Ahora tomamos una máquina que hace lo siguiente:

INPUT: $(x, n) \in \Sigma^* \times \mathbb{N}$

Inicializar: $c := I_M(c)$ la configuración inicial de M sobre x e $i := 0$
while $c \notin F_M$ (una configuración final aceptadora de M) **do**
    Ejecuta un paso de M sobre la configuración c
    $i := i + 1$
return
OUTPUT: **if** $i \leq n$ ACEPTAR,
    else RECHAZAR
fi

Esta máquina acepta el lenguaje indicado. Nótese que sale del ciclo while solamente si x es aceptada por M y que el "contador" i cuenta el número de pasos realizados. □

Proposición 9.5.10. *Un lenguaje $L \subseteq \Sigma^*$ es decidible si y solamente si es recursivo.*

Demostración. Es obvio, aunque podamos decir unas palabras por razones didácticas. Si un lenguaje L es decidible y M es la máquina de Turing que evalúa χ_L, basta con modificar ligeramente M, añadiendo un nuevo estado (que será el único estado final aceptador) que sólo se activa cuando M termina su ejecución y aparece un 1 en la cinta de output de M. Esta nueva máquina tiene como lenguaje aceptado a L. Indénticamente se puede hacer en el caso de $\Sigma^* \setminus L$.
Para el recíproco, basta con simular en paralelo las dos máquinas de Turing M_1 y M_2 que, respectivamente, verifican $L(M_1) = L$ y $L(M_2) = \Sigma^* \setminus L$. Para simultanearlas, basta con considerar tantas cintas de trabajo como la suma de las de M_1 y las de M_2, de tal modo que cada grupo de cintas reproduce independientemente las computaciones de M_1 y M_2 respectivamente sobre las respectivas configuraciones. Para ello, bastará con tomar como espacio de estados $Q_1 \times Q_2$ (i.e. el producto cartesiano de los de M_1 y M_2). Como conjunto de estados finales aceptadores basta con tomar los que representan la idea "aceptar por la regla de el primero que acepta gana". Es decir, los estdaos finales aceptadores son $F_1 \times Q_2 \bigcup Q_1 \times F_2$. □

Algunos autores prefieren hablar de lenguajes y problemas decidibles, otros prefieren usar el término recursivo. Nosotros los usaremos indistintamente. Finalmente, veamos que un lenguaje es recurviamente enumerable si y solamente si es enumerable por una función computable.

Teorema 9.5.11. *Un lenguaje $L \subseteq \Sigma^*$ es recursivamente enumerable si y solamente si es imagen de una función computable (i.e. recursiva). Un lenguaje es recursivo si y solamente si es decidible.*

Demostración. La segunda de las afirmaciones es innecesaria dado que la acabamos de discutir. Para la primera de las afirmaciones.

- $\Longleftarrow$: Supongamos que un lenguaje es enumerable mediante una función computable

$$\phi : \mathbb{N} \longrightarrow \Sigma^*.$$

Definamos una máquina de Turing del modo siguiente:

INPUT: $x \in \Sigma^*$

Inicializar: $i := 0$

while $\phi(i) \neq x$ **do**

$i := i + 1$

eval $\phi(i)$

return

OUTPUT: Aceptar

Obviamente, esta máquina de Turing acepta si y solamente si x está en $\phi(\mathbb{N})$ y, por tanto, $\phi(\mathbb{N})$ es un lenguaje recursivamente enumerable.

- $\Longrightarrow$: Para el recíproco, upongamos que $L := L(M)$ es un lenguaje recursivamente enumerable. Usaremos dos máquinas de Turing:
  - La máquina M_1 que decide la función Φ_L y que se describe en la Proposición 9.5.9 anterior.
  - La máquina de Turing M_2 que evalúa la función *next* de la Proposición 9.5.8 anterior.

Con ellas dos podemos definir la máquina de Turing siguiente:

INPUT: $n \in \mathbb{N}$

Inicializar:

- $x := \lambda$,
- $i := 0$,
- $k := 0$.

while $i \leq n$ **do**

eval $\Phi_L(x,k)$ (usando la máquina M_1

if $\Phi_L(x,k) = 1$, **do** $i := i+1$, $(x,k) := next(x,k)$ (usando M_2)

else do $i := i$, $(x,k) := next(x,k)$

fi

return

OUTPUT: x

Nótese que el "contador" i lleva la cuenta de cuántas palabras del lenguaje $L \times \mathbb{N}$ son anteriores a la palabra (x,k) que estamos analizando. Salimos del "while" solamente cuando hemos llegado a la $n-$ésima aceptación. Por eso, nos permitimos emitir x que es el $n-$ésimo elemento de L.

□

9.5.3. Independencia del Número de Cintas

En principio, es relativamente fácil mostrar una cierta independencia del número de cintas. Se puede probar el siguiente resultado:

Proposición 9.5.12. *Dada una máquina de Turing con k cintas de trabajo sobre un alfabeto Σ, es posible simular su computación sobre una máquina de Turing sobre el alfabeto Σ^{k+1}, usando 6 cintas de trabajo y tal que:*
El tiempo y espacio requeridos por la máquina Universal $\mathcal{U}$ viene acotado por :

$$T_{M_1}(x) \leq O(|x| + s_M(x))t_M(x)),$$

$$s_{M_1}(c_M, x) \leq O(s_M(x)).$$

Demostración. La prueba se sigue de la demostración que haremos a continuación de la existencia de la máquina Universal de Turing. □
Un resultado mucho más espectacular es el siguiente resultado debido a [HeSt, 66]

Proposición 9.5.13 (Cambio del número de cintas,)**.** *Sea Σ un alfabeto finito y sea $M := (\Sigma, Q, q_0, F, \delta)$ una máquina de Turing sobre el alfabeto Σ. Supongamos que M utiliza k cintas de trabajo. Entonces, existe una máquina de Turing M_1 sobre el mismo alfabeto con menos de 6 cintas de trabajo, y tal que se verifica $L(M_1) = L(M)$, $Res_{M_1} = Res_M$, y las funciones de tiempo y espacio mantienen la siguiente relación.*

$$t_{M_1}(n) \leq O(t_M(n) \log t_M(n)),$$

$$s_{M_1}(n) \leq O(s_M(n)).$$

9.6. La máquina Universal de A. Turing.

En su trabajo de 1936 (cf. [Turing, 37]), A. Turing introdujo un ejemplo de problema recursivamente enumerable que no es recursivo. Ya se conocían los resultados de K. Gödel y A. Church; pero resulta interesante señalarlo aquí. Un problema recursivamente enumerable que no es rescursivo es un *problema que se puede enunciar, pero no se puede resolver por métodos algorítmicos.*

9.6.1. El código de una máquina de Turing.

Añadiremos algunas hipótesis menores:
Por lo visto en la Subsección anterior, podemos dedicarnos a reflexionar sobre el comportamiento de la complejidad cuando disponemos de un alfabeto fijo con solamente dos elementos. En lo que sigue haremos una descripción de la modelización del código de una máquina de Turing sobre el alfabeto $\{0,1\}$.
De paso, iremos introduciendo algunas restricciones al modelo de Turing, por simplicidad. Los resumimos en la siguiente Definición:

Definición 9.6.1. Llamaremos máquina de Turing sobre el alfabeto binario $\{0,1\}$ a todo quíntuplo

$$M := (\Sigma, Q, q_0, A, \delta),$$

donde

i) $\Sigma = \{0,1\}$,

ii) $Q = \{0,1,2,3,\ldots,N\} \subseteq \mathbb{N}$, donde $\sharp(Q) = N+1$.

iii) $q_0 := 0$,

iv) N es el único estado final aceptador,

v) La función de transición viene dada por :

$$\delta \ : Q \times \left(\Sigma \bigcup \{\triangleright, \square\}\right)^{k+1} \longrightarrow Q \times \left(\Sigma \bigcup \{\square\}\right)^{k} \times \{-1,0,+1\}^{k+1}.$$

donde $\triangleright, \square \notin \Sigma$, son símbolos de los que se supone que no pertenecen al alfabeto,

Es relativamente fácil probar que toda máquina de Turing es equivalente a una máquina de este tipo, manteniendo esencialmente las cotas de tiempo y espacio, salvo los cambios causados por el uso del alfabeto binario ya indicados.

9.6.1.1. El código de una máquina de Turing sobre el alfabeto Universal.

Consideraremos el alfabeto universal :

$$\Sigma_0 := \{[,], (,), 0, 1, \text{“,”}, \text{“;”}, +1, -1, \triangleright, \text{“}\square\text{”}\}$$

donde “,” y “;” se refieren a la “coma” y el “punto y coma” usuales, mientras “$\square$” hace refrencia a un símbolo que representa la palabra vacía λ.

Definición 9.6.2. Llamaremos código de una máquina de Turing $M := (\{0,1\}, Q, q_0, A, \delta)$, sobre el alfabeto Σ_0 a toda palabra $c_M \in \Sigma_0^*$ dada por las siguientes propiedades :

$$c_M := (N; K; [B_M]) \in \Sigma_0^*$$

donde

i) $Q := \{0,\ldots,N\}$,

ii) $q_0 = 0$,

iii) $F = \{N\}$, es el conjunto de estados finales aceptadores, formado por un sólo estado. Supondremos que $N \in \{0,1\}^* \subseteq \Sigma_0^*$, es dado en binario.

iv) M usa K cintas de trabajo. Supondremos que $K \in \{1\}^*$, es dado en unario.

v) B_M es una lista finita :

$$B_M := (a_1, b_1), \ldots, (a_M, b_M) \in \Sigma_0^*$$

tal que lo siguiente se verifica :

- Para cada i, $1 \leq i \leq M$,

$$a_i \in Q \times \left(\{0,1\} \bigcup \{\triangleright, \square\}\right)^{k+1},$$

esto es, a_i ha de tener la forma :

$$a_i := (q; w_0, \ldots, w_k) \in \Sigma_0^*,$$

donde $q \in \{0,1\}^*$, $0 \leq q \leq N$ (es dado en binario) y $w_i \in \{0,1,\square,\triangleright\}$.

- Para cada i, $1 \leq i \leq M$,

$$b_i \in Q \times (\{0,1,\square\})^k \times \{-1,0,+1\}^{k+1},$$

esto es, b_i ha de tener la forma :

$$b_i := (q; w_1, \ldots, w_k; \varepsilon_0, \ldots, \varepsilon_k) \in \Sigma_0^*,$$

dondeq $\in \{0,1\}^*$, $0 \leq q \leq N$ (es dado en binario), $w_i \in \{0,1,\square\}$ y $\varepsilon_i \in \{-1,0,+1\}$.

- El grafo de la función de transición δ viene dado por :

$$Gr(\delta) = \{(a_i, b_i) \ : \ 1 \leq i \leq M\}.$$

Observación 9.6.1. Obsérvese que la información descrita en el código de una máquina de Turing es completa para describir la máquina de Turing.

Proposición 9.6.1. *La aplicación que a cada máquina de Turing $M := (\{0,1\}, \{0,\ldots,N\}, 0, N, \delta)$ sobre el alfabeto $\{0,1\}$ le asocia su código $c_M \in \Sigma_0^*$ es una aplicación inyectiva.*

Proposición 9.6.2. *Con las notaciones de la Definición 9.6.2, cualquier palabra $c \in \Sigma_0^*$ que verifique las condiciones i) a vii) expuestas en esa Defínción, es el código de una máquina de Turing.*

Notación 9.6.1. Llamaremos talla de una máquina de Turing al tamaño del anterior código c_M sobre el alfabeto Σ_0 y lo denotaremos por $K(M)$.

Proposición 9.6.3. *Sea $M := (\{0,1\}, \{0,\ldots,N\}, 0, N, \delta)$ una máquina de Turing con K cintas. Sea $c_M \in \Sigma_0^*$ su código. Entonces,*

$$K(M) \leq c(log_2 N + log_2 N + K + \left((N+1)^2 (4)^{2(K+1)} 3^{K+2}\right)).$$

Demostración. Se sigue simplement de observar la lista de objetos implicados. □

9.6.2. La máquina Universal : ejemplo de compilador e intérprete.

La máquina Universal es una de las grandes ideas asociada a los conceptos de A. Turing. En términos informáticos se puede entender como un lenguaje de alto nivel, o un intérprete de las máquinas de Turing. En esencia no es nada más que eso, esto es, un compilador o un intérprete. Sin embargo, no sólo sirve para la concepción de este tipo de procesos, sino que tiene también una interpretación y utilización teórica de cierta relevancia. Lo que haremos aquí es una somera descripción de su funcionamiemto con un modelo que aumenta cuadráticamente su complejidad en tiempo.
La máquina Universal $\mathcal{U}$ es una máquina de Turing sobre el alfabeto universal Σ_0^* antes introducido.
La entrada de una máquina Universal es un par

$$\text{INPUT} : \ (c_M, x) \subseteq \Sigma_1^*$$

donde c_M es el código de una máquina de Turing sobre el alfabeto Σ_0 introducido en la Subsección anterior y $x \in \{0,1\}^* \subseteq \Sigma_0^*$ es una palabra con las siguientes caractersticas.
El output de la máquina universal $\mathcal{U}$ sobre tales inputs, está definido si y solamente si $x \in L(M)$. En caso contrario, no está definido, la máquina no acepta el input. El valor de tal output es justamente

$$Res_M(x) \in \{0,1\}^* \subseteq \Sigma_0^*.$$

EL proceso de funcionamiento de la máquina universal es también una simulación. Para intentar describir este proceso de simulación, tendremos el siguiente código para las configuraciones de una máquina M.

9.6.2.1. El código de una configuración de una máquina de Turing dada por su código.

Pretendemos hacer observar lo siguiente. Podemos expresar una configuración de una máquina de Turing M mediante una lista sobre el alfabeto Σ_0 que depende solamente del código de la máquina de Turing. Así, sea

$$M := (\{0,1\}, \{0,\ldots,N\}, 0, N, \delta)$$

una máquina de Turing sobre el alfabeto $\{0,1\}$ con k cinas de trabajo. Sea

$$c_M := (N; k; [B_M]) \in \Sigma_0^*,$$

su código. Sea $C \in S_M$ una configuración de la máquina M. Supongamos :

$$C := (q; \triangleright w_0, \ldots, \triangleright w_k; n_0, \ldots, n_k) \in S_M$$

Llamaremos código de la configuración C a la palabra :

$$c(C) := (q; \triangleright w_0, \ldots, \triangleright w_k; n_0, \ldots, n_k) \in \Sigma_0^*$$

donde :

- $q \in \{0,1\}^*$ es el estado dado en binario (esto es, $0 \leq q \leq N$,
- $w_j \in \{0,1\}^*$ son los contenidos de las cintas.
- $n_i \in \{1\}^*$ son las posiciones de las unidades de control y son dadas en unario. Obsérvese que la única condición que deben verificar es que $n_i \leq |w_i| + 2$.

Unos primeros resultados esencialmente obvios:

Proposición 9.6.4. *Existe una máquina de Turing M_1 sobre el alfabeto Σ_0 que evalúa la siguiente función. Sea*

$$\mathcal{C}od := \{c \in \Sigma_0^* \ : \ \exists \text{una máquina de Turing } M \text{ sobre } \{0,1\} \ \ c = c_M\}.$$

Entonces, $L(M_1) = \Sigma_0^*$ y $Res_{M_1} : \Sigma_0^* \longrightarrow \{0,1\}$ es la función característica $\chi_{\mathcal{C}od} : \Sigma_0^* \longrightarrow \{0,1\}$ definida por $\mathcal{C}od$.
Además, se verifica:

$$t_{M_1}(c) = O(|c|) \ \text{ (tiempo lineal en el tamaño de la entrada)},$$

usando espacio lineal $s_{M_1}(n) = O(n)$.

Demostración. El espacio constante es debido a que hemos considerado, dentro del código, el número de cintas K y debemos verificarlo. Es decir, debemos verificar que las instrucciones de δ tienen el buen número de entradas y salidas. □

Observación 9.6.2. Se podría hacer con un autómata finito si no fijamos 'a priori' el número de cintas involucradas dentro de c_M. Esto se haría deduciendo el número de cintas del grafo de δ, mediante una semántica un poco más sofisticada, pero evidente y, en ese caso, el lenguaje $\mathcal{C}od$ se podría definir como lenguaje regular. No lo haremos por simplificar la exposición de la máquina Universal.

Proposición 9.6.5. *Existe una máquina de Turing M_2 sobre el alfabeto Σ_0 que evalúa la siguiente función. Sea*

$$\mathcal{C}fg := \{x \in \Sigma_0^* \ : \ \exists \ \text{una máquina de Turing } M \text{ sobre } \{0,1\}$$
$$z = (c, s), c = c_M \ \ s \text{ es el código de una configuración sobre } M\}.$$

Entonces, $L(M_2) = \Sigma_0^*$ y $Res_{M_2} : \Sigma_0^* \longrightarrow \{0,1\}$ es la función característica $\chi_{\mathcal{C}fg} : \Sigma_0^* \longrightarrow \{0,1\}$ definida por $\mathcal{C}fg$.
Además, se verifica:

$$t_{M_2}(z) = O(|z|) \ \text{ (tiempo lineal en el tamaño de la entrada)},$$

usando espacio lineal $s_{M_2}(n) = O(n)$.

Demostración. Se trata de leer en una expresión $z = (c_M, s)$ e ir verificando que s codifica una configuración de c_M. Para ello basta con controlar esencialmente los símbolos y que su secuencia responde a las pautas. El espacio usado depende de modo distinto de la longitud de s que de $|c_M|$. Depende de $|c_M|$ para cotejar que el número de cintas de trabajo es K, que los estados son los estados indicados por c_M. Depende de $|s|$ en la medida en que las posiciones de las unidades de control no se salen de nuestras restricciones. De hecho, si $z = (c_M, s)$ y si s codifica una configuración alcanzable desde la configuración inicial $I_M(x)$ definida por un input $x \in \{0,1\}^*$, entonces, el tiempo y el espacio están acotados por $O(|c_M| s_M(|x|))$. □

Prosiguiendo en la misma dirección, tenemos el resultado de A. Turing del que daremos una pseudo-demostración descriptiva de su acción.

Teorema 9.6.6 (A. Turing)**.** *Existe una máquina de Turing*

$$\mathcal{U} := (\Sigma_0, Q^{(U)}, q_0^{(U)}, F^{(U)}, \delta^{(U)}),$$

sobre el alfabeto Σ_0 verificando las siguientes propiedades :

i) El lenguaje aceptado por $\mathcal{U}$ es el lenguaje $HP \subseteq \Sigma_0^*$ (llamado Halting Problem o Problema de Parada) , dado por la siguiente propiedad : Una palabra $z \in \Sigma_0^*$ está en HP si y solamente si existe una máquina de Turing M sobre el alfabeto $\{0,1\}$ y existe una palabra x sobre el alfabeto binario $\{0,1\}$ tal que :

- $z := (c_M, x) \in \Sigma_0^*$ y
- $x \in L(M) \subseteq \{0,1\}^* \subseteq \Sigma_0^*$.

ii) Si $z := (c_M, x) \in HP$, el resultado de la computación de la máquina universal $\mathcal{U}$ sobre z es dado por :

$$Res_{\mathcal{U}}(c_M, x) := Res_M(x) \in \{0,1\}^* \subseteq \Sigma_0^*.$$

iii) El tiempo y espacio requeridos por la máquina Universal $\mathcal{U}$ viene acotado por :

$$T_{\mathcal{U}}(c_M, x) \leq O((K(M) + |x| + s_M(x))t_M(x)),$$

$$s_{\mathcal{U}}(c_M, x) \leq O(K(M)s_M(x)).$$

Demostración. La demostración de este Teorema se hace mediante una descripción de los cálculos realizados por la máquina Universal. Para ello, distinguiremos varias etapas. La máquina Universal tendr a lo más 6 cintas de trabajo. Los contenidos de esas cintas, irán descritos mediante

- IT : $\triangledown\,(c_M, x)$,
- WT1 : $\triangledown\omega_1$
- WT2 : $\triangledown\omega_2$
- WT3 : $\triangledown\omega_3$
- WT4 : $\triangledown\omega_4$
- WT5 : $\triangledown\omega_5$
- WT6 : $\triangledown\omega_6$

Donde $\triangledown$ será utilizado esta vez como cursor de la máquina universal, para distinguirlo del cursor $\triangleright$ de las máquina de Turing M que van a ser simuladas. Las palabras ω_i estarán en el lenguaje Σ_0^*. El símbolo Λ (con mayscula) será utilizado por la máquina universal para simbolizar celda vacía y poder distinguirlo del símbolo λ (en minscula) que representa celda vacía en M y es un símbolo del alfabeto Σ_0. Asimismo, usaremos el símbolo @ para indicar la celda donde se encuentran las unidades de control de $\mathcal{U}$ si en algún momento deseamos indicar su posición. El significado es : la unidad de control se encuentra leyendo la celda justo siguiente a la aparición del símbolo @. Las fases esenciales, serán las siguientes :

i) **Fase 1.– Inicialización.** Acepta si y solamente si el input es de la forma (c_M, x). En caso afirmativo, copia en $WT1$ el código de la configuración inicial de M sobre x. Después retrocede todas sus unidades de control a la posición inicial. Tendremos, al final de esta fase la figura siguiente :

   - IT : $@\triangledown\,(c_M, x)$,
   - WT1 : $@\triangledown\,(0; \triangleright x, \triangleright, \ldots, \triangleright; 1, 1 \ldots, 1)$
   - WT2 : $@\triangledown\Lambda$
   - WT3 : $@\triangledown\Lambda$
   - WT4 : $@\triangledown\Lambda$
   - WT5 : $@\triangledown\Lambda$
   - WT6 :$@\triangledown\Lambda$

ii) **Fase 2.– Simulación de un Paso de la Máquina M por la máquina Universal : SIMUNIV** La máquina recibe la siguiente información

   - IT : $\triangledown\,(c_M, x)$,

- WT1 : $@\triangledown C$
- WT2 : $@\triangledown\Lambda$
- WT3 : $@\triangledown\Lambda$
- WT4 : $@\triangledown\Lambda$
- WT5 : $@\triangledown\Lambda$
- WT6 :$@\triangledown\Lambda$

donde C es el código sobre el alfabeto Σ_0 de una configuración de M sobre x, esto es,

$$C := (q; \triangleright x, \triangleright w_1, \ldots, \triangleright w_k; n_0, \ldots, n_k) \in \Sigma_0^*$$

donde :

- $q \in \{0,1\}^*$ es el estado dado en binario (esto es, $0 \leq q \leq N$),
- $w_j \in \{0,1\}^*$ son los contenidos de las cintas.
- $n_i \in \{1\}^*$ son las posiciones de las unidades de control y son dadas en unario para facilitar operaciones del tipo movimiento simultáneo. Obsérvese que la única condición que deben verificar es que
$$n_i \leq |w_i| + 2.$$

Supongamos, además, que
$$c_M := (N; k; [B_M]) \in \Sigma_0^*.$$

La simulación es un proceso que se hará en varias etapas. Cada etata supone crear un subconjunto del espacio de estados. Cada etapa esta dada esencialmente con operaciones del tipo:

- *Borra completamente una cinta.*
- *Copia en una cinta el contenido de otra (entre dos símbolos prefijados).*
- *Vuelve las unidades de control (de una o varias cintas) a la posición inicial.*
- *Mueve simultáneamente dos o más cintas (entre dos símbolos prefijados).*

Estas operaciones elementales pueden ser descritas con anterioriddad (ver Hoja de problemas II); así que nos limitaremos a las grandes líneas del proceso.

Procedimiento SIMUNIV :

- *F.2.1.–Criterio de Parada.* Si el estado de la configuración codificada en $WT1$ es aceptador, se acaba el cálculo. En caso contrario, escribe el estado actual (que esta contenido en $WT1$) en la cinta $WT2$. Nos quedar la figura :
  - IT : $@\triangledown\left((N; k; [B_M]), x\right)$,
  - WT1 : $@\triangledown\left(q; \triangleright x, \ldots, \triangleright w_k; n_0, \ldots, n_k\right)$
  - WT2 :$\triangledown(q@;$
  - WT3 : $\triangledown@($
  - WT4 : $@\triangledown\Lambda$
  - WT5 : $@\triangledown\Lambda$
  - WT6 :$@\triangledown\Lambda$
- *F.2.2.–Copiado de las posiciones de las unidades de control.* Lee las posiciones de las unidades de control (descritas en $WT1$) y los copia en $WT3$. Tendremos la figura :
  - IT : $@\triangledown\left((N; k; [B_M]), x\right)$,
  - WT1 : $\triangledown\left(q@; \triangleright x, \ldots, \triangleright w_k; n_0, \ldots, n_k\right)$

- WT2 :$\triangledown(q@;$
- WT3 : $\triangledown@(n_0,\ldots,n_k)$
- WT4 : $@\triangledown\Lambda$
- WT5 : $@\triangledown\Lambda$
- WT6 :$@\triangledown\Lambda$

- *F.2.3.-* **LECTURA**, *esto es, Copiado de las celdas de lectura.* Usando las posiciones en $WT3$ copia en $WT2$ los contenidos de las celdas indicadas por las posiciones de las unidades de control. Tendremos la figura :
  - IT : $@\triangledown\left((N;k;[B_M]),x\right)$,
  - WT1 : $@\triangledown\left(q;\triangleright x,\ldots,\triangleright w_k;n_0,\ldots,n_k\right)$
  - WT2 :$@\triangledown(q;x,y_1,\ldots,y_k)$
  - WT3 : $@\triangledown(n_0,\ldots,n_k)$
  - WT4 : $@\triangledown\Lambda$
  - WT5 : $@\triangledown\Lambda$
  - WT6 :$@\triangledown\Lambda$
- *F.2.4.–* **TRANSICION**, *esto es, Buscando el valor de la función de transición.* Usando el grafo de la función de transición (descrito en c_M en la cinta de input) y la lectura actual (descrito en $WT2$) busca la imagen de la lectura por el valor de transición y lo escribe en $WT4$. Tendremos la figura
  - IT : $@\triangledown\left((N;k;[B_M]),x\right)$,
  - WT1 : $@\triangledown\left(q;\triangleright x,\ldots,\triangleright w_k;n_0,\ldots,n_k\right)$
  - WT2 :$@\triangledown(q;x,y_1,\ldots,y_k)$
  - WT3 : $@\triangledown(n_0,\ldots,n_k)$
  - WT4 : $@\triangledown(q';y'_1,\ldots,y'_k;\varepsilon_0,\ldots,\varepsilon_k)$
  - WT5 : $@\triangledown\Lambda$
  - WT6 :$@\triangledown\Lambda$
- *F.2.5.–* **ESCRITURA**, *esto es, Modificación de los contenidos de la configuración.* Reescribe en $WT6$ la configuración descrita en $WT1$ modificada de acuerdo a las instrucciones de escritura indicadas en $WT4$. Tendremos al figura :
  - IT : $@\triangledown\left((N;k;[B_M]),x\right)$,
  - WT1 : $\triangledown\left(q;\triangleright x,@\ldots,\triangleright w_k;n_0,\ldots,n_k\right)$
  - WT2 :$\triangledown(q;x,y_1,\ldots,y_k)$
  - WT3 : $@\triangledown(n_0,\ldots,n_k@)$
  - WT4 : $\triangledown(q';@y'_1,\ldots,y'_k;@\varepsilon_0,\ldots,\varepsilon_k)$
  - WT5 : $\triangledown(q';\triangleright x,\triangleright w'_1,\ldots,w'_k;@$
  - WT6 : $@\triangledown\Lambda$
- *F.2.6.–* **Movimientos**, *esto es, Modificación de la configuración.* Ahora en $WT6$ escribiremos las nuevas posiciones de las unidades de control, usando la información descrita en $WT3$ (las posiciones viejas) y los movimientos que hay que hacer (escritos en $WT4$). Nos quedar :
  - IT : $@\triangledown\left((N;k;[B_M]),x\right)$,
  - WT1 : $\triangledown\left(q;\triangleright x,@\ldots,\triangleright w_k;n_0,\ldots,n_k\right)$
  - WT2 :$\triangledown(q;x,y_1,\ldots,y_k)$
  - WT3 : $\triangledown(n_0,\ldots,n_k@)$
  - WT4 : $\triangledown(q';@y'_1,\ldots,y'_k;@\varepsilon_0,\ldots,\varepsilon_k)$
  - WT5 : $\triangledown(q';\triangleright x,\triangleright w'_1,\ldots,w'_k;n'_0,\ldots,n'_k@)$
  - WT6 : $@\triangledown;n'_0,\ldots,n'_k@)$

- *F.2.7.– Copiado final.* procedemos a copiar lo escrito en $WT5$ en la cinta $WT1$ (que habremos borrado previamente) y borramos todas las demás. Nos quedar :
  - IT : $@\triangledown\left(\left(N;k;\left[B_M\right]\right),x\right)$,
  - WT1 : $@\triangledown(q';\triangleright x,\triangleright w'_1,\ldots,w'_k;n'_0,\ldots,n'_k)$
  - WT2 :$@\triangledown\Lambda$
  - WT3 : $@\triangledown\Lambda$
  - WT4 :$@\triangledown\Lambda$
  - WT5 : $\triangledown(q';\triangleright x,\triangleright w'_1,\ldots,w'_k;n'_0,\ldots,n'_k@)$
  - WT6 : $@\triangledown\Lambda$

Este proceso verifica la siguiente propiedad.

Lema 9.6.7. *Sea $\mathcal{C}$ la configuración de la máquina Universal $\mathcal{U}$ dada mediante :*

- IT : $\triangledown\left(c_M,x\right)$,
- WT1 : $@\triangledown C$
- WT2 : $@\triangledown\Lambda$
- WT3 : $@\triangledown\Lambda$
- WT4 : $@\triangledown\Lambda$
- WT5 : $@\triangledown\Lambda$
- WT6 :$@\triangledown\Lambda$

Sea $\mathcal{C}'$ la configuración de la máquina universal $\mathcal{U}$ dada mediante :

$$\mathcal{C}' := \mathbf{SIMUNIV}(\mathcal{C}),$$

esto es, $\mathcal{C}'$ es el resultado del cálculo antes descrito y tiene la forma :

- IT : $\triangledown\left(c_M,x\right)$,
- WT1 : $@\triangledown C'$
- WT2 : $@\triangledown\Lambda$
- WT3 : $@\triangledown\Lambda$
- WT4 : $@\triangledown\Lambda$
- WT5 : $@\triangledown\Lambda$
- WT6 :$@\triangledown\Lambda$

Entonces,

i) C' es el código de una configuración de la máquina M.

ii) $C \to C'$ (esto es, C' se obtiene de C mediante un paso de cálculo de M.

iii) El número de pasos que necesita la máquina universal $\mathcal{U}$ para simular el paso

$$\mathcal{C} \to_{\mathcal{U}} \cdots \to_{\mathcal{U}} \mathcal{C}',$$

est acotado por :

$$O(K(M) + |x| + S_M(|x|)).$$

Demostración. El proceso **SIMUNIV** se construye para que haga exactamente esto. □
Se concluye que el número de pasos que realiza $\mathcal{U}$ en la simulación de M es el número de pasos que da M multiplicado por lo que le cuesta a $\mathcal{U}$ simular un paso de M. De ah la cota del enunciado.
□

Corollario 9.6.8. *El lenguaje de parada (HP) es un Lenguaje Recursivamente enumerable.*

9.6.3. El problema de la Parada y cuestiones conexas.

Veamos ahora dos ejemplos de problemas expresables (esto es, enunciables), pero no resolubles algorítmicamente. Asimismo, de la máquina Universal se concluye la siguiente Proposición.

Proposición 9.6.9 (Independencia del número de cintas de trabajo)**.** *Sea Σ un alfabeto finito y sea $M := (\Sigma, Q, q_o, F, \delta)$ una máquina de Turing sobre el alfabeto Σ. Supongamos que M utiliza k cintas de trabajo. Entonces, existe una máquina de Turing M_1 sobre el mismo alfabeto con solamente 6 cintas de trabajo, y tal que se verifica :*

i) $L(M_1) = L(M)$,

ii) $Res_{M_1} = Res_M$,

iii)

$$t_{M_1}(n) \leq O(s_M(n)t_M(n)),$$
$$s_{M_1}(n) \leq O(s_M(n)TO_M(n)).$$

Demostración. Basta con usar la misma filosofía que la máquina Universal $\mathcal{U}$ antes descrita. □

Observación 9.6.3. Existe una estrategia llamada *amortizing analysis* que permite mejorar el tiempo de simulación de una máquina de Turing Universal. Fueron introducidas en [HeSt, 66] y [HaLeSt, 65].

Teorema 9.6.10. *Existe una máquina de Turing universal $\mathcal{U}$ verificando las propiedades a) y b) del Teorema 9.6.6, y tal que*

$$t_{\mathcal{U}}(M, x) \leq C_M t_M(x) \log_2 t_M(x), \quad s_{\mathcal{U}}(M, x) \leq D_M s_M(x),$$

donde C_M y D_M son dos constantes que sólo dependen del tamaño del código de M.

9.7. El Problema de la Parada.

Junto a esta máquina Universal, A. Turing presentó el siguiente enunciado :

Teorema 9.7.1 (Problema de Parada)**.** *El siguiente lenguaje $HP \subseteq \{0,1\}^*$ es un lenguaje recursivamente enumerable que no es recursivo :*

$$HP := \{(c_M, x) \ : \ x \in L(M)\}.$$

Demostración. Para demostrar este enunciado usaremos un argumento de *Diagonalización* del tipo siguiente. Dado que HP es un lenguaje recursivamente enumerable, veamos que no es recursivamente enumerable el lenguaje :

$$HP^c := \Sigma_0^* \setminus HP.$$

Para ello, consideremos un diccionario $\sigma \ : \Sigma_0 \longrightarrow \{0,1\}^4$. Sea $\tilde{\sigma} \ : \Sigma_0^* \longrightarrow \{0,1\}^*$ el correspondiente monomorfismo de monoides.
Razonando por reducción al absurdo, supongamos que HP^c es recursivamente enumerable. Entonces, también será recursivamente enumerable el lenguaje siguiente :

$$L_1 := \{(c, x) \in \Sigma_0^* \ : \ \exists M \text{ máquina de Turing sobre } \{0,1\}, \ \ c = c_M, \\ x \in \{0,1\}^*, (c, x) \notin HP\}.$$

Para decidir L_1, suponiendo que HP^c, es recursivamente enumerable, basta con rechazar todos los inputs que no sean de la forma (c, x) con

- c es el código de una máquina de Turing sobre $\{0,1\}$ (esto es, las propiedades descritas en Definición 9.6.2).

- $x \in \{0,1\}^*$.

Una vez hecho esto, bastará con que sigamos aplicando la máquina de Turing que supuestamente acepta HP^c. Por la Proposición 9.5.3 existirá una máquina de Turing M_1 sobre $\{0,1\}$ que acepta el lenguaje $\tilde{\sigma}(L_1)$. Entonces, será recursivamente enumerable el siguiente lenguaje :

$$L_2 := \{x \in \{0,1\}^* \ : \ x = \tilde{\sigma}(c_M), (c_M, x) \in L_1\}.$$

Este es el lenguaje de la diagonalización. Para ver que es recursivamente enumerable, sea M_1 la máquina de Turing que acepta $\tilde{\sigma}(L_1)$. Ahora construiremos una máquina de Turing M_2 que aceptara L_2 y que viene dada por :

INPUT $z \in \{0,1\}^*$.
SI no existen $c_M \in \Sigma_0^*$ (código de máquina de Turing) y $x \in \{0,1\}^*$ tales que $z = (c_M, x)$, **Rechazar**
    EN CASO CONTRARIO, prosigue la computación.
        SI $x \neq \tilde{\sigma}(c_M)$, **Rechazar**
            EN CASO CONTRARIO, aplica M_1 a $\tilde{\sigma}(c_M, \tilde{\sigma}(c_M))$
        ENDIF
ENDIF
OUTPUT el mismo output que M_1

Sea M_2 la máquina de Turing sobre $\{0,1\}$ que acepta el lenguaje L_2.
Sea, pues, $c := c_{M_2}$ y sea $z := \tilde{\sigma}(c) \in \{0,1\}^*$.
Tendremos que $z \in L_2$ o $z \notin L_2$.
Caso I : $z \in L_2$. En este caso, $(c_{M_2}, z) \in L_1$, luego M_2 no acepta el input $z \in \{0,1\}^*$ (por definición de L_1). En particular, $z \notin L(M_2)$; pero $L(M_2) = L_2$ con lo que habremos llegado a contradicción.
Caso II : $z \notin L_2$. En este caso, $(c_{M_2}, z) \notin L_1$, luego $z \in L(M_2)$ (por definición de L_2. Pero $L(M_2) = L_2$, luego, $z \in L_2$ y habremos llegado también a contradicción.
Por lo tanto, no puede haber máquina de Turing que acepte HP^c. □.
□

La interpretación de estos dos resultados es la siguiente. En primer lugar, la máquina de Turing universal es también el lenguaje al que se transfiere (técnicamente compila, interpreta) todo programa, escrito en algún lenguaje de programación, en cada máquina concreta. Se conoce como *Lenguaje Máquina* o ensamblador y es el lenguaje al que traducen los compiladores los programas escritos en lenguajes de nivel más alto, para obtener un código ejecutable.
El Problema de Parada no es sólo un ejemplo de problema irresoluble algorítmicamente, sino que desmuestra, además, que el sueño de la verificación es imposible. El Teorema de A. Turing dice que no puede existir un verificador universal de programas, dando pie a la *Programación Estructurada* y, a medio plazo, a la *Ingeniería de Software*.

9.8. El final del Problema X de Hilbert

La respuesta al Problema X de Hilbert (cf. Problema 1.6) se obtiene con la secuencia de trabajos en la que intervienen varios autores.
La idea inicial era convertir los conjuntos recursivamente enumerable, caracterizados por K. Gödel en su tesis y trabajos ulteriores, en conjuntos diofánticos, es decir, conjuntos dados mediante un bloque de cuantificadores existenciales y una hipersuperficie.

Definición 9.8.1. Un subconjunto $S \subseteq \mathbb{Z}^n$ se llama diofántico si existe un polinomio con coeficientes diofánticos $f \in \mathbb{Z}[X_1, \ldots, X_n, Y_1, \ldots, Y_m]$ de tal modo que

$$S := \{(x_1, \ldots, x_n) \in \mathbb{Z}^n \ : \ \exists y_1 \in \mathbb{Z}, \ldots, \exists y_m \in \mathbb{Z}, \ \ f(x_1, \ldots, x_n, y_1, \ldots, y_m) = 0\}.$$

Algunas observaciones preliminares son las siguientes:

i) Los conjuntos definidos como las soluciones diofánticas de un número finito de ecuaciones polinomiales con coeficientes enteros también son diofánticos. Es decir, si $S \subseteq \mathbb{Z}^n$ es dado mediante:

$$S := \{x \in \mathbb{Z}^n \ : \ f_1(x) = 0, \ldots, f_s(x) = 0\},$$

Entonces S es diofántico, dado que eligiendo el polinomio $p := f_1^2 + \cdots + f_s^2 \in \mathbb{Z}[X_1, \ldots, X_n]$, tendremos

$$S := \{x \in \mathbb{Z}^n \ : \ p(x) = 0\}.$$

ii) Los conjuntos dados mediante un número finito de intersecciones de igualdades y desigualdadees polinomiales también son diofánticos. Es decir, dado $S \subseteq \mathbb{Z}^n$ dado mediante:

$$S := \{x \in \mathbb{Z}^n \ : \ f_1(x) = 0, \ldots, f_s(x) = 0, g_1(x) \geq 0, \ldots, g_m(x) \geq 0\},$$

entonces S es diofántico. Usaremos el Teorema de Lagrange de los Cuatro Cuadrados:

Teorema 9.8.1 (Lagrange, 1770)**.** *Todo número entero positivo es suma de* 4 *cuadrados. Es decir, para cada $n \in \mathbb{Z}$, son equivalentes $z \geq 0$ y*

$$\exists x_1 \in \mathbb{Z}, \exists x_2 \in \mathbb{Z}, \exists x_3 \in \mathbb{Z}, \exists x_4 \in \mathbb{Z}, \ \ n = x_1^2 + x_2^2 + x_3^2 + x_4^2.$$

Introducimos $4m$ nuevas variables

$$Y_1^{(1)}, Y_1^{(2)}, Y_1^{(3)}, Y_1^{(4)}, Y_2^{(1)}, \ldots, Y_m^{(4)}.$$

Consideramos los polinomios

$$G_i(X_1, \ldots, X_n, Y_1^{(1)}, \ldots, Y_m^{(4)}) := g_i(X_1, \ldots, X_n) - \left(\sum_{j=1}^{4} \left(Y_i^{(j)} \right)^2 \right).$$

Entonces, se tiene para todo $x \in \mathbb{Z}^n$,

$$g_i(x) \geq 0 \Leftrightarrow \exists y_1^{(1)} \in \mathbb{Z}, \ldots, \exists y_m^{(4)} \in \mathbb{Z}, \ \ G_i(x, y_1^{(1)}, \ldots, y_m^{(4)}) = 0.$$

Finalmente, el conjunto S tendrá la forma:

$$S := \{x \in \mathbb{Z}^n \ : \ \exists y_1^{(1)} \in \mathbb{Z}, \ldots, \exists y_m^{(4)} \in \mathbb{Z}, f_1(x) = 0, \ldots, f_s(x) = 0, G_1(x, y...) = 0, \ldots, G_m(x, y...) = 0\}.$$

Para convertirlo en diofántico, basta con sumar cuadrados, es decir, queda

$$S := \{x \in \mathbb{Z}^n \ : \ \exists y_1^{(1)} \in \mathbb{Z}, \ldots, \exists y_m^{(4)} \in \mathbb{Z}, \sum_{i=1}^{s} f_i^2(x) + \sum_{j=1}^{m} G_j(x_1, \ldots, x_n, y_1^{(1)}, \ldots, y_m^{(4)})^2 = 0\}.$$

iii) Finalmente, son conjuntos diofánticos, todos los conjuntos expresables mediante fórmulas que contienen intersecciones finitas de condiciones de signo $= 0$ o ≥ 0 con polinomios diofánticos y un sólo bloque de cuantificadores existenciales, esto es, fórmulas del tipo:

$$\exists Y_1, \ldots, \exists Y_1, \ (\wedge_{i=1}^{s} [f_i(X_1, \ldots, X_n, Y_1, \ldots, Y_m) = 0]) \wedge \left(\wedge_{j=1}^{m} [g_j(X_1, \ldots, X_n, Y_1, \ldots, Y_m) \geq 0] \right).$$

Los pasos iniciales fueron dados por Martin Davis quien, en 1949, demuestra que los conjuntos recursivamente enumerables se pueden representar mediante una fórmula casi-diofántica, que contiene solamente un cuantificador universal $\forall$. Al mismo tiempo demuestra que existe un conjutno diofántico cuyo complementario no es diofántico. Entonces, conjetura que recursivamente enumerable y diofántico debe ser lo mismo.
Casi al mismo tiempo (1950), y sin conocer los trabajos de Davis, Julia Robinson intenta estudiar la función exponencial y su eventual carácter diofántico. Es decir, que el conjutno $EXP := \{(a,b,c) \in \mathbb{Z}^3 \ : \ a = b^c\}$ es diofántico. Enuncia la hipótesis JR:
"Existe un conjunto diofántico D tal que si $(a,b) \in D$, entonces $b < a^a$ y, además, para cada $k > 0$, existen $(a,b) \in D$ tales que $b > a^k$.
Entonces, Julia Robinson demuestra que si existe ese conjunto diofántico, EXP es diofántico también. Añadiendo las exponenciales a nuestras funciones polinomiales, Julia Robinson (en colaboración con Davis y Hilary Putnam) demuestra en 1959 que, con la hipótesis de que "hay infinitas sucesiones de primos en progresión aritmética" (hoy demostrada), los conjuntos recursivamente enumerables son los conjuntos definibles con un bloque de cuantificadores existenciales, polinomios y exponenciales: los exp-diofánticos.
En 1960, J. Robinson redemuestra que los conjuntos recursivamente enumerables son exp-diofánticos sin necesidad de la hipótesis sobre los primos en progresión aritmética. A partir de ese momento, se tiene demostrado que el Problema X no admite ningún algoritmo (i,.e. es indecidible) simplemente probando la hipótesis "JR". Trabaja bastante tiempo en su hipótesis JR sin resultadosd efinitivos.
Finalmente, en 1970, Juri V. Matiyasevich demuestra la hipótesis "JR" (cf. [8]), probando que el siguiente conjunto es diofántico:

$$P := \{(a,b) \in \mathbb{Z}^n \ : \ a > 0, \ b = F_{2a}\},$$

donde F_n es el $n-$ésimo número de Fibonacci. Para obtener su resultado Matiyasevich usa resultados técnicos relevantes de Nikolai Vorob'ev.
Tras el resultado de Matiyasevich, recopilemos:

i) Los conjuntos diofánticos son recursivamente enumerables y hay conjuntos diofánticos cuyo complementario no es diofántico (Davis).

ii) Los conjuntos recursivamente enumerables son los conjuntos exp-diofánticos (Robinson).

iii) Los conjuntos exp-diofánticos son diofánticos si se verifica la hipótesis JR (Robinson).

iv) La hipótesis JR es cierta (Matiyasevich). Ergo, los conjuntos recursivamente enumerables y los conjuntos diofánticos definen la misma clase de subconjuntos de $\mathbb{Z}$.

v) Existe un conjunto diofántico cuyo complementario no es recursivamente enumerable (Gödel).

Corollario 9.8.2 (MRDP Theorem). *No existe algoritmo que decida la existencia de solución diofántica de ecuaciones polinomiales con coeficientes racionales. Ergo, la respuesta al Problema X de Hilbert es negativa.*

Demostración. Dado que existe un conjunto diofántico S que no es recursivo. Sea

$$S := \{(x_1, \ldots, x_n) \in \mathbb{Z}^n \ : \ \exists y_1 \in \mathbb{Z}, \ldots, \exists y_m \in \mathbb{Z}, \ \ p(x_1, \ldots, x_n, y_1, \ldots, y_m) = 0\}, \qquad (9.8.1)$$

ese conjunto, con $p \in \mathbb{Z}[X_1, \ldots, X_n, Y_1, \ldots, Y_m]$. Entonces, no puede existir ningún algoritmo que decida todas las ecuaciones polinomiales (como pretende el enunciado del Problema X de Hilbert). Razonando por reducción al absurdo, sea p el plinomio que define el conjunto diofántico S cuyo complementario $\mathbb{Z} \setminus S$ no es recursivamente enumerable. Y supongamos que $\mathcal{P}$ es un "algoritmo"

[8] Ju. V. Matijasevic, *Enumerable sets are definable*. Soviet Math. Dokl. **11.2** (1970), 354–358.

(máquina de Turing) que resuelve el Problema X, esto es, para cada polinomio $f \in \mathbb{Z}[X_1, \ldots, X_n]$ el algoritmo $\mathcal{P}$ devuelve el valor $\mathcal{P}(f) \in \{0, 1\}$ del modo siguiente:

$$\mathcal{P}(f) := \begin{cases} 1, & \text{si } \exists x \in \mathbb{Z}^n,\ f(x) = 0 \\ 0, & \text{en otro caso.} \end{cases}$$

Definimos el algoritmo siguiente:

INPUT: $x \in \mathbb{Z}^n$
eval $f(Y_1, \ldots, Y_m) := p(x_1, \ldots, x_n, Y_1, \ldots, Y_m)$,
(donde p es el polinomio de la ecuación (9.8.1) anterior)
eval $\mathcal{P}(f)$,
(donde $\mathcal{P}$ es el algoritmo que suponemos que resuelve el Problema X de Hilbert)
if $\mathcal{P}(f) = 1$, OUTPUT: NO
else OUTPUT: SI
fi
end

Este algoritmo resuelve el problema de pertenencia a $\mathbb{Z} \setminus S$, con lo que $\mathbb{Z} \setminus S$ sería recursivamente enumerable y, por ende, S sería recursivo lo que no es cierto. □

9.9. Sistemas de Thue: Problemas de la Palabra

Las gramáticas de tipo 0 son también *Sistemas de Semi–Thue* (véase, por ejemplo, la referencia en [DaWe, 94]) en honor del matemático que las introdujo. Hblaremos de sistemas de Semi–Thue finitamente generados y finitamente presentados cuando el alfabeto subyacente sea finito y las reglas de reescritura sean dadas en número finito. El objetivo de Thue era analizar el siguiente tipo de problemas.

Problema (Problema de la Palabra para Sistemas de Semi–Thue)**.** Dado un sistema de semi–Thue (Σ, R) y dados $x, y \in \Sigma^*$, decidir si $x \vdash_R y$.

Problema (Problema de la Palabra en Semigrupos)**.** Dado R un sistema de semi–Thue sobre un alfabeto finito Σ, consideramos la estructura de semigrupo con unidad de Σ^* (monoide). Dos palabras $x, y \in \Sigma^*$ se dicen relacionadas mediante R, si $x \vdash_R y$ en el sistema de transición asociado (i.e. si y es deducible de x).

Un sistema de Thue es un sistema de semi–Thue en el que R verifica la siguiente propiedad adicional :

$$\forall x, y \in \Sigma^*,\ \ (x, y) \in R \Leftrightarrow (y, x) \in R$$

Entonces, R define una relación de equivalencia $\vdash_R$ en Σ^* y podemos considerar el conjunto cociente :

$$S(\Sigma, R) := \Sigma^* / \vdash_R$$

Claramente se tiene que $S(\Sigma, R)$ es un semigrupo, cuyos elementos son las clases $[x]$ definidas por elementos $x \in \Sigma^*$.
El problema de la palabra para semigrupos se define mediante :
Dados un sistema de Thue (Σ, R) *y dados* $x, y \in \Sigma^*$, *decidir si* $[x] = [y]$

Observación 9.9.1. Esta versión del problema de la palabra está relacionada directamente con un hábito muy común en matemáticas. Supongamos que quiero trabajar con un semigrupo S, no necesariamente conmutativo. Para describirlo, todos pondríamos un conjunto de generadores

(digamos $\{\gamma_1, \ldots, \gamma_n\}$). Sabidos los generadores, sabemos que los elementos son todos de la forma :

$$\gamma_{s(1)} \cdots \gamma_{s(m)}$$

donde $s : \{1, \ldots, m\} \longrightarrow \{1, \ldots, n\}$ es una aplicación, con $m \in \mathbb{N}$. El problema de una representación –tal cual ésta– es que uno no puede hacer cosas tan elementales como comparar dos elementos dados (obsérvese que nadie dijo que las cosas conmuten ni que la representación sea única). Por lo tanto, uno debería dar, al menos, las relaciones entre los generadores (que son inevitables). Estas relaciones tienen la pinta

$$\gamma_{r_i(1)} \cdots \gamma_{r_i(m_i)} = \gamma_{s_i(1)} \cdots \gamma_{s_i(k_i)}$$

para $1 \leq i \leq N$, siendo r_i y k_i aplicaciones con rango $\{1, \ldots, n\}$. Nos conformamos con que sólo haya un número finito de relaciones entre los generadores. Claramente, tenemos un sistema de reescritura sobre el alfabeto $\Sigma = \{1, \ldots, n\}$, siendo

$$R := \{(r_i(1) \cdots r_i(m_i), s_i(1) \cdots s_i(k_i)) \ : \ 1 \leq i \leq N\}$$

Es obvio que nuestro semigrupo S inicial es justamente $S(\Sigma, R)$. Luego el problema de las palabras viene a decir si somos capaces de identificar o distinguir dos elementos de un semigrupo dado por sus generadores y sus relaciones. La respuesta , dada por E. Post[9] en 1947 es que el problema de la palabra para semi–grupos finitamente presentados es indecidible (luego, insoluble).

Teorema 9.9.1 (Post[10], 47)**.** *Los problemas de palabras para sistemas de semi–Thue, y semigrupos son insolubles algorítmicamente.*

El problema de la palabra en grupos El problema anterior se sofistica un poco más, si en lugar de semigrupo hablamos de grupos. Un grupo finitamente generado (no necesariamente abeliano) no es sino un semigrupo asociado a un sistema de Thue (Σ, R) que, además verifica la propiedad siguiente :
existe una aplicación $\sigma : \Sigma \longrightarrow \Sigma$ *tal que :*

$$\forall a \in \Sigma, (a\sigma(a), \lambda) \in R)$$

donde λ *es la palabra vacía.* Escribamos $G(\Sigma, R)$ por el grupo cociente Σ^*/R
El problema de la palabra es también :
`Dado un sistema de grupo` (Σ, R) `y dadas` $x, y \in \Sigma^*$`, decidir si` $[x] = [y]$ `en` $G(\Sigma, R)$`.`
Tras mucho esfuerzo P. Novikov[11] (en 1955) y W.W. Boone[12] (con una demostración mucho más simple, en 1958) lograron demostrar que el enunciado siguente:

Teorema 9.9.2 (Novikov–Boone)**.** *El problema de la palabra para grupos finitamente presentados y finitamente generados es insoluble algorítmicamente.*

Problema de correspondencia de Post. Se trata de otro problema basado en los sistemas de reescritura y que resulta, también insoluble algorítmicamente (cf. E. Post[13] en 1946).

Problema (Post Correspondence)**.** Consideremos un sistema de semi–Thue (Σ, R) y sus elementos como piezas de dominó :

$$R := \{(x_1, y_1), \ldots, (x_n, y_n)\}$$

[9] E. Post. "Recursive unsolvability of a Problem of Thue". *J. of Symb. Logic* **12** (1947) 1–11.
[10] E. Post. "Recursive unsolvability of a Problem of Thue". *J. of Symb. Logic* **12** (1947) 1–11.
[11] P.S. Novikov. "On the algorithmic unsolvability of the word problem in group theory". *Proceedings of the Steklov Institute of Mathematics* **44** (1995), 1-143.
[12] William W. Boone. "The word problem". *Proceedings of the National Academy of Sciences* **44** (1958) 1061-1065.
[13] E. Post . "A variant of a recursively unsolvable problem." *Bull. A.M.S.* **52** (1946) 264–268.

y las piezas

$$\mathcal{D}_i :=| \frac{x_i}{y_i} |$$

Decidir si existe una secuencia de fichas

$$\mathcal{D}_{s(1)} \cdots \mathcal{D}_{s(n)}$$

tal que lo que aparece escrito en las partes superiores de los dominós coincide con lo escrito debajo.

Por ejemplo, sea R (Post prefiere Pairing Lists i.e. PL)

$$R := \{(a, aa), (bb, b), (a, bb)\}$$

para el alfabeto $\Sigma := \{a, b\}$. La siguiente es una solución :

$$| \frac{a}{aa} || \frac{a}{bb} || \frac{bb}{b} || \frac{bb}{b} |$$

Teorema 9.9.3 (Post, 46)**.** *El problema de la correspondencia es insoluble por métodos algorítmicos. En otras palabras, no existe (ni se puede encontrar) un algoritmo que resuelva el problema de correspondencia de Post.*

La prueba de la Indecidibilidad de este Problema puede verse en el [Weh, 97] o en el [DaWe, 94] , entre otros.

9.10. Números reales recursivamente enumerables.

Pongamos la siguiente definición del conjunto de números reales recursivamente enumerables.

Definición 9.10.1. Un número real $x \in \mathbb{R}$ se dice recursivamente enumerable, si existen $a \in \mathbb{Z}$, $y \in [0, 1)$ y una máquina de Turing M sobre el lenguaje $\{0, 1\}$, tales que :

i) $x = a + y$

ii)

$$y = \sum_{k=1}^{\infty} \frac{a_k}{2^k}$$

donde

$$a_k = 1 \Longleftrightarrow k \in L(M)$$

Observación 9.10.1.

i) La base decimal (bien sea 2, 3, 10 u otra), no es relevante en la definición. Hemos usado 2 por su simplicidad.

ii) Obsérvese que el conjunto de los números reales recursivamente enumerables es un conjunto contable, luego es un subconjunto propiamente contenido en el cuerpo de los números reales.

iii) Obsérvese que la definición de recursivamente enumerables indica esencialmente que se trata de números que podemos dar a alguien para que haga algo con ellos. Por ejemplo, se puede hacer con ellos la suma, la resta, la multiplicación. Casi se podra hacer la división salvo por lo que sigue.

iv) Obsérvese que los números racionales y los números reales algebraicos son recursivamente enumerables. Para estos últimos basta con considerar algoritmos de aproximación como los basados en el operador de Newton y los α y γ Teoremas de M. Shub y S. Smale.

v) Obsérvese que esta definición es equivalente a la siguiente : $x \in \mathbb{R}_{re}$ si y solamente si existe $a \in \mathbb{Z}$ y una máquina de Turing que evalua una función :

$$\varphi \ : \mathbb{N} \longrightarrow \mathbb{N}$$

verificándose

$$x = a + \sum_{k=1}^{\infty} \frac{1}{2^{\varphi(k)}}$$

La razón última de este hecho es la coincidencia existente entre los conjuntos recursivamente enumerables y los conjuntos imagen de funciones recursivas.

El siguiente resultado de A. Turing muestra que, sin embargo, no podemos trabajar con números reales recursivamente enumerables.

Teorema 9.10.1. *El siguiente problema no es recursivamente enumerable :*

$$Ineq := \{x, y \in \mathbb{R}_{re}^2 \ : \ x > y\}$$

Demostración. Claramente su complementario es recursivo. Vamos calculando los índices para los que $\frac{1}{3^k}$ aparece en la expansión ternaria de x e y. Cuando aparezca algún dígito distinto ya son distintos.
Para ver nuestro enunciado :
Sea dado un par (c_M, x) de objetos sobre el alfabeto Σ_U de la máquina universal. A partir de estos objetos construimos un número racional, cuando c_M es una máquina de Turing y $x \in \{0,1\}^*$. El método es el siguiente :
Asociamos a x la expansión binaria de un número natural. Denotemos por $\iota(x) \in \mathbb{N}$ el número natural cuya expansión binaria es $1x$ o 0 si $x = \lambda$.
Construimos el siguiente número real recursivamente enumerable :

$$r_{M,x} := \sum_{\iota(y) \geq \iota(x), y \in L(M)} \frac{1}{2^{2\iota(y)}}$$

Este número está en el intervalo $[0,1]$ y es recursivamente enumerable porque el lenguaje

$$L := \{\iota(y) \ : \ \iota(y) \geq \iota(x), y \in L(M)\}$$

es recursivamente enumerable.
Observamos que para cada $x \in \{0,1\}^*$ tenemos que si $x \in L(M)$ ocurre que :

$$r_{M,x} \geq \frac{1}{2^{2\iota(x)}}$$

En cambio, si $x \notin L(M)$ se tiene :

$$r_{M,x} \leq \sum_{k \geq \iota(x)+1} \frac{1}{2^{2k}} = \frac{1}{4^{\iota(x)+1}} \left(\sum_{k \geq 0} \frac{1}{4^k} \right) = \frac{1}{4^{\iota(x)+1}} \left(\frac{1}{1 - \frac{1}{4}} \right) = \frac{1}{4^{\iota(x)+1}} \frac{4}{3} = \frac{1}{3} \frac{1}{4^{\iota(x)}} < \frac{1}{4^{\iota(x)}}$$

Concluiremos así que

$$x \notin L(M) \Leftrightarrow r_{M,x} < \frac{1}{4^{\iota(x)}}.$$

En particular, el conjunto $Ineq$ es la imagen del complementario del conjunto de parada por una función recursiva. Si $Ineq$ fuera recursivamente enumerable, también lo sería HP^c y llegaríamos a contradicción. □

9.11. Tape Compression Lemma y Linear Speed-Up

La utilización de las máquinas de Turing para el análisis de la compejidad de algoritmos se remonta a los trabajos de J. Hartmanis y R. Stearns en 1965 [14]. Sus primeros resultados, muestran que la complejidad debe ser entendida como la asintótica de las fuciones de tiempo y espacio a partir del tamaño del input. Estos primeros resultados son el Tape Compression Lemma y el Linear Speed–Up Lemma que daremos a continuación.
Antes de comenzar retomemos nuestros resultados sobre el cambio del alfabeto ya discutidos anteriormente,
Por último, es claro que la máquina M_1 realiza los cálculos previstos. □

Definición 9.11.1. Sean Σ, τ dos alfabetos. Diremos que dos lenguajes $L \subseteq \Sigma^*$ y $L' \subseteq \tau^*$ están identificados salvo sistema de codificación de caracteres, si existe un sistema de codificación de caracteres de longitud (r, s) $\sigma : \Sigma^r \longrightarrow \tau^s$ tal que $\tilde{\sigma}(L) = L'$.

Corollario 9.11.1. *Sean Σ y τ dos alfabetos finitos de cardinal mayor que 2. Sea $f : \mathbb{R} \longrightarrow \mathbb{R}$ una función monótona creciente. Entonces, existen una constantes $c, d \in \mathbb{R}, c > 0, d > 0$ tales que se verifica, salvo identificación de caracteres:*

i) $DTIME_\Sigma(f) \subseteq DTIME_\tau(cf)$ *y* $DTIME_\tau(f) \subseteq DTIME_\Sigma(df)$,

ii) $DSPACE_\Sigma(f) \subseteq DSPACE_\tau(cf)$ *y* $DSPACE_\tau(f) \subseteq DSPACE_\Sigma(df)$,

iii) $NTIME_\Sigma(f) \subseteq NTIME_\tau(cf)$ *y* $NTIME_\tau(f) \subseteq NTIME_\Sigma(df)$,

iv) $NSPACE_\Sigma(f) \subseteq NSPACE_\tau(cf)$ *y* $NSPACE_\tau(f) \subseteq NSPACE_\Sigma(df)$.

Demostración. Basta con usar los diccionarios y cambios de alfabeto descritos en la Subsección 9.5.1 y la Proposición 9.5.3.

□

Observación 9.11.1. El Corolario anterior ya indica que cambiando el alfabeto no modifica la clase de complejidad, siempre que se preserven las clases asintóticas de complejidad, es decir, clases dadas mediante $O(f)$ y no mediante f solamente. Esto se verá con más detalle en los resultados de la SUbsección siguiente.

9.11.1. Tape Compression Lemma

Se llama Tape Compression Lemma al siguiente Teorema.

Teorema 9.11.2. *[HaSt, 65] Sea $L \subseteq \Sigma^*$ un lenguaje aceptado por una máquina de Turing (determinstica o no) usando espacio acotado por una función $s : \mathbb{N} \longrightarrow \mathbb{R}_+$ monótona creciente (i.e.$L \in NSPACE_\Sigma(s)$ o $L \in DSPACE_\Sigma(s)$), y sea $c \in \mathbb{R}, 0 < c < 1$. Entonces, existe un alfabeto τ tal que $\Sigma \subseteq \tau$ y una máquina de Turing (del mismo tipo de determinismo) sobre el alfabeto τ , tal que :*

i) $L(M_c) = L$,

ii) $s_{M_c}(n) \leq cs_M(n)$.

En otras palabras, para cada función monótona creciente $s : \mathbb{N} \longrightarrow \mathbb{R}_+$, para cada alfabeto Σ y para cada constante $0 < c < 1$ existe un alfabeto τ que contiene a Σ y tal que

$$DSPACE_\Sigma(s) \subseteq DSPACE_\tau(cs),$$

$$NSPACE_\Sigma(s) \subseteq NSPACE_\tau(cs).$$

[14] J. Hartmanis, R. Stearns. "On the Computational Complexity of Algorithms". *Trans. of the A.M.S.* **117** (1965) 285–306.

Demostración. Supongamos que la máquina M usa solamente una cinta de trabajo (en caso de varias cintas de trabajo el formalismo sera el mismo aunque más lioso de expresar) y que viene dada por :

$$M := (\Sigma, Q, q_0, F, \delta),$$

donde

$$\delta \ : Q \times \Sigma^2 \longrightarrow Q \times \sigma \times \{-1, 0, +1\}^2$$

Sea $r > 0$, $r \in \mathbb{N}$ tal que

$$\frac{2}{r} < c$$

A partir de este número natural r tendremos una nueva máquina de Turing M_r que vendrá dada por :
El proceso de construcción de la nueva máquina seguirá como sigue :

i) *El nuevo alfabeto :*

$$\tau := \Sigma \dot{\bigcup} \left(\Sigma \dot{\bigcup} \{\triangledown\} \right)^r$$

ii) *El nuevo espacio de estados :*

$$Q' := (Q \times \{1, \ldots, r\}) \dot{\bigcup} \{(q_0, A), (q_0, B), \}$$

iii) *El nuevo estado inicial :*

$$q_0' := (q_0, A)$$

iv) *El nuevo conjunto de estados finales aceptadores :*

$$F' := F \times \{1, \ldots, r\}$$

Habrá que dedicar un poco más de tiempo para manejar la nueva función de transición :

$$\delta' \ : Q' \times \tau^2 \longrightarrow Q' \times \tau \times \{-1, 0, +1\}^2.$$

La máquina M_r tendrá dos fases.
Una primera fase de **Inicialización** en la que la máquina lee el input y simplemente decide si el input $z \in \tau^*$ está en Σ^* o no. Si no está en Σ^* rechaza el input dando **Error**. En caso contrario procede a computar. Para esta fase de inicilización usaremos los estados $\{(q_0, A), (q_0, B), \}$. En este proceso sólo hacemos LECTURA Y MOVIMIENTOS en la cinta de input. El proceso de Inicilialización se puede describir como sigue : Esto se puede describir fácilmente mediante la expresión siguiente :

- *Si el estado es* (q_0, A). Mientras leas en la cinta de input un elemento en $\Sigma \bigcup \{\triangleright\}$ avanza un paso a la derecha y mantén el estado (q_0, A). Si lees un símbolo λ no hagas movimientos y pasa al estado (q_0, B). Si lees un símbolo en $\tau \setminus \Sigma$ rechaza pasando al estado **Error**.
- *Si el estado es* (q_0, B). Retrocede hasta que encuentres el símbolo $\triangleright$. Cuando lo encuentres, pasa al estado $(q_0, 1)$.

El estado $(q_0, 1)$ es el estado donde empiezan los trabajos. En el resto de los trabajos la cinta de trabajo sólo contendrá símbolos del tipo $\triangleright z$ donde

$$z \in \left(\left(\Sigma \dot{\bigcup} \{\triangledown\} \right)^r \right)^* .$$

Para poder entender el proceso de simulación de M que pretendemos, trabajemos sobre traducciones del sistema de transición de M y el de M_r.
Supongamos dada una configuración de la máquina M :

$$C := (q, \triangleright x, \triangleright y; n_0, n_1) \in S_M,$$

donde

- $q \in Q$
- $x := x_1 \cdots x_n \in \Sigma^*$, con $|x| = n$.
- $y := y_1 \cdots y_m \in \Sigma^*$, con $|y| = m$.
- $n_0 \leq n + 2$,
- $n_1 \leq m + 2$.

Una *compresión de la configuración* C será dada por una lista :

$$\mathcal{C} := (q', \xi, \eta; n_0', n_1') \in S_{M_r},$$

donde

- $q' := (q, i) \in Q'$
- $\xi := x_1 \cdots x_n = x \in \Sigma^* \subseteq \tau^*$.
- $\eta := \eta_1 \cdots \eta_{m'} \in \tau^*$, con
$$m' := \lfloor \frac{m}{r} \rfloor = \frac{m}{r}, \text{ cuando } r \mid m$$
$$m' := \lfloor \frac{m}{r} \rfloor + 1 \text{ en caso contrario .}$$
La palabra η estar dada del modo siguiente :
  - Para cada j, $1 \leq j \leq \lfloor \frac{m}{r} \rfloor$
$$\eta_j := (y_{(j-1)r+1}, \ldots, y_{jr}) \in \tau.$$
  - Si r no divide a m, sea $t := \lfloor \frac{m}{r} \rfloor$, entonces,
$$\eta_{t+1} = \eta_{m'} := (y_{(t-1)r+1}, \ldots, y_m, \triangledown, \ldots, \triangledown) \in \tau.$$
- Las posiciones verifican $n_0' = n_0$, mientras que :
  - Si r no divide a n_1,
$$n_1' := \lfloor \frac{n_1}{r} \rfloor,\ i := n_1 - rn_1' := rem(n_1, r)$$
  - Si r divide a n_1,
$$n_1' := \lfloor \frac{n_1}{r} \rfloor - 1,\ i = r.$$

Obsérvese que la aplicación

$$\text{Compresión} \quad : S_M \longrightarrow S_{M_r}$$

es inyectiva, esto es a cada configuración de M le corresponde una única configuración de M_r. Obsérvese que si realizamos un paso de computación en M $C \to_M C_1$ y si C', C_1' só las correspódientes compresiones, debemos definir la máquina M_r para que

$$C' \to_{M_r} C_1'.$$

Una máquina de este tipo es fcil de concebir, aunque de complicada expresión. Se trata de que la nueva función de transición δ' simule el efecto de la compresión. En otras palabras, la nueva máquina hará lo siguiente :
Dada la configuración C,

- LEER el símbolo i de $\eta_{n_1'}$ y el símbolo x_{n_0} de ξ. Leer el estado q de (q, i).
- HACER LA TRANSICION de la máquina M original.

- ESCRIBIR el nuevo valor de $\eta_{n_1'}$ conforme al cambio de un único símbolo de esa lista.
- MOVERSE del modo siguiente : si la posición i (del estado (q,i)) está en medio (esto es $2 \leq i \leq r-1$ mover el índice $i \to i+\varepsilon_1$ y no mover la unidad de control. Si, por el contrario, $i = 1$ o $i = r$, y la unidad de control tiende a salir del bloque de r palabras en el que está, entonces debo cambiar el índice a r cuando $i = 1$ y tienda hacia la izquierda. Pasar el índice a 1, cuando $i = r$ y tienda a la derecha.

Lo que sigue es un intento pormenorizado de definir una función de transición δ' que hace este proceso. Nótese que el espacio de trabajo ocupado por M_r es ahora

$$m' \leq \lfloor \frac{m}{r} \rfloor + 1.$$

Ahora bien dada una configuración C en la máquina M tal que $m = s_M(n)$, la correspobndiente configuración de la máquina M_r ocupar espacio :

$$m' \leq \lfloor \frac{s_M(n)}{r} \rfloor + 1 \leq 2 \lfloor \frac{s_M(n)}{r} \rfloor \leq c s_M(n).$$

Y tenemos el resultado apetecido.

9.11.1.0.1. Intento pormenorizado de describir la máquina del Tape Compression Lemma. Nos interesa pues explicitar el valor D dado por :

$$D := \delta' \left(q', x, w \right),$$

en los casos siguientes :

- $q' = (q,i) \in (Q \times \{0,1,\ldots,r\})$,
- $x \in \Sigma \bigcup \{\triangleright, \lambda\}$,
- $w = \triangleright, \lambda$ o
$$w := (w_1, \ldots, w_r) \in \left(\Sigma \dot{\bigcup} \{\triangledown\} \right)^r.$$

En el resto de los casos no contemplados, la función manda al estado **Error** y rechaza el input. La función de transición tomar los siguientes valores :

- Si $w = \triangleright$, entonces
$$D := ((q_1, 1), \triangleright; \varepsilon_0, \varepsilon_1),$$
donde
$$\delta(q, x, \triangleright) := (q_1, \triangleright; \varepsilon_0, \varepsilon_1),$$
- Si $w = \lambda$, $i > 1$,, entonces
$$D := (\mathbf{Error}, \lambda; 0, 0),$$
- Si $w = \lambda$, $i = 1$, y si
$$\delta(q, x, \lambda) = (q_1, \theta; \varepsilon_0, -1),$$
entonces
$$D := ((q_1, r), w'; \varepsilon_0, -1),$$
donde
$$w' := (\theta, \triangledown, \ldots, \triangledown) \in \tau.$$

- Si $w = \lambda$, $i = 1$,y si
$$\delta(q, x, \lambda) = (q_1, \theta; \varepsilon_0, 0),$$
entonces
$$D := ((q_1, 1), w'; \varepsilon_0, 0),$$
donde
$$w' := (\theta, \triangledown, \ldots, \triangledown) \in \tau.$$

- Si $w \in \left(\Sigma \dot{\bigcup} \{\triangledown\}\right)^r$ distinguimos los casos siguientes :
  - Si $2 \leq i \leq r-1$ y $w_j \neq \triangledown$ para cada j, $1 \leq j \leq i$ y si
$$\delta(q, x, w_i) = (q_1, \theta; \varepsilon_0, \varepsilon_1),$$
con $\theta = \lambda$ entonces
$$D := ((q_1, i + \varepsilon_1), w'; \varepsilon_0, 0),$$
donde
$$w' := (w_1, \ldots, w_{i-1}, \triangledown, \triangledown, \ldots, \triangledown) \in \tau.$$
  - Si $2 \leq i \leq r-1$ y $w_j \neq \triangledown$ para cada j, $1 \leq j \leq i$ y si
$$\delta(q, x, w_i) = (q_1, \theta; \varepsilon_0, \varepsilon_1),$$
con $\theta \in \Sigma$, $\theta \neq \lambda$ entonces
$$D := ((q_1, i + \varepsilon_1), w'; \varepsilon_0, 0),$$
donde
$$w' := (w_1, \ldots, w_{i-1}, \theta, w_{i+1}, \ldots, w_r) \in \tau.$$
  - Si $2 \leq i \leq r-1$ y $w_i = \triangledown$, si $w_j \in \Sigma$ para cada j, $1 \leq j \leq i-1$ y si
$$\delta(q, x, \lambda) = (q_1, \theta; \varepsilon_0, \varepsilon_1),$$
entonces
$$D := ((q_1, i + \varepsilon_1), w'; \varepsilon_0, 0),$$
donde
$$w' := (w_1, \ldots, w_{i-1}, \theta, \triangledown, \ldots, \triangledown),$$
  - Si $i = 1$ y $w_1 \neq \triangledown$ y si
$$\delta(q, x, w_1) = (q_1, \theta; \varepsilon_0, -1),$$
con $\theta \neq \lambda$ entonces
$$D := ((q_1, r), w'; \varepsilon_0, -1),$$
donde
$$w' := (\theta, w_2, \ldots, w_r).$$
  - Si $i = 1$ y $w_1 \neq \triangledown$ y si
$$\delta(q, x, w_1) = (q_1, \theta; \varepsilon_0, -1),$$
con $\theta = \lambda$ entonces
$$D := ((q_1, r), w'; \varepsilon_0, -1),$$
donde
$$w' := \lambda.$$

- Si $i = 1$ y $w_1 \neq \triangledown$ y si
$$\delta(q, x, w_1) = (q_1, \theta; \varepsilon_0, \varepsilon_1),$$
con $\varepsilon_1 \neq -1$, $\theta \neq \lambda$, entonces
$$D := ((q_1, 1 + \varepsilon_1), w'; \varepsilon_0, 0),$$
donde
$$w' := (\theta, w_2, \ldots, w_r).$$
- Si $i = 1$ y $w_1 \neq \triangledown$ y si
$$\delta(q, x, w_1) = (q_1, \theta; \varepsilon_0, \varepsilon_1),$$
con $\varepsilon_1 \neq -1$, $\theta = \lambda$, entonces
$$D := ((q_1, 1), w'; \varepsilon_0, 0),$$
donde
$$w' := (\triangledown, \ldots, \triangledown).$$
- Si $i = 1$, $w_1 = \triangledown$, y si
$$\delta(q, x, \lambda) = (q_1, \theta; \varepsilon_0, -1),$$
con $\theta \neq \lambda$, entonces
$$D := ((q_1, r), w'; \varepsilon_0, -1),$$
donde
$$w' := (\theta, \triangledown, \ldots, \triangledown) \in \tau.$$
- Si $i = 1$, $w_1 = \triangledown$, y si
$$\delta(q, x, \lambda) = (q_1, \lambda; \varepsilon_0, -1),$$
entonces
$$D := ((q_1, r), w'; \varepsilon_0, -1),$$
donde
$$w' := \lambda.$$
- Si $i = 1$, $w_1 = \triangledown$, y si
$$\delta(q, x, \lambda) = (q_1, \lambda; \varepsilon_0, \varepsilon_1),$$
con $\varepsilon_1 \neq -1$, entonces
$$D := ((q_1, 1), w'; \varepsilon_0, 0),$$
donde
$$w' := (\triangledown, \triangledown, \ldots, \triangledown) \in \tau.$$
- Si $i = 1$, $w_1 = \triangledown$, y si
$$\delta(q, x, \lambda) = (q_1, \theta; \varepsilon_0, \varepsilon_1),$$
con $\theta \neq \lambda$ y con $\varepsilon_1 \neq -1$, entonces
$$D := ((q_1, 1 + \varepsilon_1), w'; \varepsilon_0, 0),$$
donde
$$w' := (\theta, \triangledown, \ldots, \triangledown) \in \tau.$$
- Si $i = r$ y $w_r \neq \triangledown$
$$\delta(q, x, w_r) = (q_1, \theta; \varepsilon_0, +1),$$
entonces $\theta \in \Sigma$ y
$$D := ((q_1, 1), w'; \varepsilon_0, +1),$$
donde
$$w' := (w_1, \ldots, w_{r-1}, \theta) \in \tau.$$

- Si $i = r$ y $w_r \neq \triangledown$

$$\delta(q, x, w_r) = (q_1, \theta; \varepsilon_0, \varepsilon_1),$$

con $\theta \in \Sigma$ y $\varepsilon_1 \neq +1$, entonces

$$D := ((q_1, r + \varepsilon_1), w'; \varepsilon_0, 0),$$

donde

$$w' := (w_1, \ldots, w_{r-1}, \theta) \in \tau.$$

- Si $i = r$ y $w_r \neq \triangledown$

$$\delta(q, x, w_r) = (q_1, \lambda; \varepsilon_0, \varepsilon_1),$$

con $\varepsilon_1 \neq +1$, entonces

$$D := ((q_1, r + \varepsilon_1), w'; \varepsilon_0, 0),$$

donde

$$w' := (w_1, \ldots, w_{r-1}, \triangledown) \in \tau.$$

- Si $i = r$ y $w_r = \triangledown$

$$\delta(q, x, \lambda) = (q_1, \theta; \varepsilon_0, +1),$$

con $w_j \in \Sigma$, para cada j, $1 \leq j \leq r - 1$, entonces

$$D := ((q_1, 1), w'; \varepsilon_0, +1),$$

donde

$$w' := (w_1, \ldots, w_{r-1}, \theta) \in \tau.$$

- Si $i = r$ y $w_r = \triangledown$

$$\delta(q, x, w_r) = (q_1, \lambda; \varepsilon_0, +1),$$

con

$$D := ((q_1, r), w'; \varepsilon_0, 0),$$

donde

$$w' := (w_1, \ldots, w_{r-1}, \triangledown) \in \tau.$$

- Si $i = r$ y $w_r = \triangledown$

$$\delta(q, x, \lambda) = (q_1, \theta; \varepsilon_0, \varepsilon_1),$$

con $\varepsilon_1 \neq +1$, y $\theta \neq \lambda$, entonces

$$D := ((q_1, r + \varepsilon_1), w'; \varepsilon_0, +1),$$

donde

$$w' := (w_1, \ldots, w_{r-1}, \theta) \in \tau.$$

- Si $i = r$ y $w_r = \triangledown$

$$\delta(q, x, \lambda) = (q_1, \theta; \varepsilon_0, \varepsilon_1),$$

con $\varepsilon_1 \neq +1$, y $\theta = \lambda$, entonces

$$D := ((q_1, r + \varepsilon_1), w'; \varepsilon_0, +1),$$

donde

$$w' := (w_1, \ldots, w_{r-1}, \triangledown) \in \tau.$$

□

9.11.2. Linear Speed–Up.

Se llama Linear Speed–Up al siguiente enunciado :

Teorema 9.11.3. *[HaSt, 65] Sea $L \subseteq \Sigma^*$ un lenguaje aceptado por una máquina de Turing (determinstica o no) usando espacio acotado por una función $t : \mathbb{N} \longrightarrow \mathbb{R}_+$ monótona creciente (i.e.$L \in NTIME_\Sigma(t)$ o $L \in DTIME_\Sigma(t)$) y sea $c \in \mathbb{R}, 0 < c < 1$. Entonces, existe un alfabeto τ tal que $\Sigma \subseteq \tau$ y una máquina de Turing (del mismo tipo de determinismo) sobre el alfabeto τ , tal que :*

i) $L(M_c) = L$,

ii) $t_{M_c}(n) \leq 2n + ct_M(n)$.

En otras palabras, para cada función monótona creciente $t : \mathbb{N} \longrightarrow \mathbb{R}_+$, para cada alfabeto Σ y para cada constante $0 < c < 1$ existe un alfabeto τ que contiene a Σ y tal que

$$DTIME_\Sigma(t) \subseteq DTIME_\tau(2n + ct),$$

$$NTIME_\Sigma(t) \subseteq NTIME_\tau(2n + ct).$$

Demostración. Para demostrar este resultado usaremos el mismo proceso de compresión que se realizaba con la máquina M_r del Teorema anterior. La diferencia estriba en que, en nuestro caso, por cada r pasos de la máquina original, daremos solamente 6 pasos. Este proceso de simulación de denomina un "baile". Trataremos de describir el proceso de un "baile"para simular r pasos de la máquina original y dejaremos como Ejercicio la descripción promenorizada de la máquina correspondiente. Sea $r \in \mathbb{N}$ tal que :

$$\frac{12}{r} < c.$$

Definamos una máquina M'_r sobre el alfabeto de la máquina del Tape Compression Lemma. Esto es, el nuevo alfabeto será

$$\tau := \Sigma \dot{\bigcup} \left(\Sigma \dot{\bigcup} \{\triangledown\} \right)^r$$

La máquina nueva funcionar como sigue :

INPUT $x \in \Sigma^*$,
COMPRIME x a una palabra z con símbolos en $\left(\Sigma \dot{\bigcup} \{\triangledown\}\right)^r$
SIMULA el comportamiento de la máquina M_r del Tape Compression Lemma sobre z, mediante bailes.
ACEPTA si y solamente si M_r acepta z.

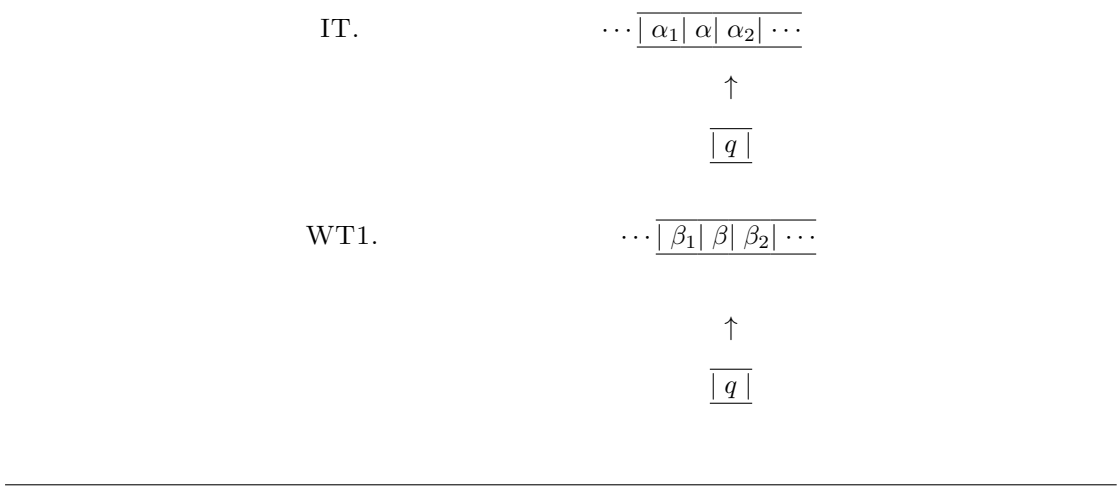
Lo que haremos será la descripción de un baile. Supondremos una sola cinta de trabajo.
Supongamos que la máquina M_r se encuentra en una situación como la descrita por :

$$C := ((q, i), \alpha, \beta; n_0, n_1) \in S_{M_r}.$$

donde

$$\alpha, \beta \in \left(\Sigma \dot{\bigcup} \{\triangledown\} \right)^r$$

Ahora los próximos r pasos de la máquina M_r sólo pueden afectar a la celda en la que está y a las dos celdas contiguas. Grficamente,



AHora ejecutamos un baile que tiene las siguientes fases :

- **Paso 1.**– Guarda el contenido de las celdas que estás leyendo en la unidad de control y mueve un paso a la derecha ambas cintas.
- **Paso 2.**– Añade el contenido de la cinta que estás leyendo a la unidad de control y da un paso a la izquierda con ambas cintas. Obsérvese que estamos ahora en la posición de la que partíamos, aunque la Unidad de control tiene muchas más información (una cantidad finita, por supuesto)
- **Paso 3.**– Da un paso a tu izquierda con ambas cintas.
- **Paso 4.**– Añade el contenido de la cinta que estás leyendo a la Unidad de control y da un paso a la derecha. Obsérvese que tras estos cuatro primeros pasos, tendremos en la Unidad de control la siguiente información :
  - El estado del que partíamos.
  - Las posiciones respectivas que tendra la máquina original dentro del bloque de r símbolos del que partíamos.
  - Los contenidos de las cintas adyacentes, esto es, $(\alpha_1, \alpha, \alpha_2)$ y $(\beta_1, \beta, \beta_2)$
- **Paso 5.**– La transición de M'_r sobre la información de la Unidad de control, consiste en la aplicación que da r pasos de la máquina original. Obsérvese que la máquina original no puede hacer más de r desplazamientos en r pasos. Así pues, no podríamos modificar el contenido nada más que el de la cinta que estamos y una de las adyacentes, pero no en las dos adyacentes. AL mismo tiempo, en esos r movimientos de M_r no podemos ir sino o bien a la celda de la izquierda, o bien a la celda de la derecha o quedarnos donde estabmos. Y estos son los movimientos de la transición. Asi que hacemos lo siguiente : modifica el contenido de la celda en la que estamos conforme a los r pasos de la máquina original. Para los movimientos del Paso 5, si la cinta de input original pasa a la celda de la izquierda en r pasos, da un paso a la izquierda con la cinta de input. Si la cinta de input original pasa a la celda de la derecha tras r pasos, da un paso a la derecha con la cinta de input. Haz la misma tarea con la cinta de trabajo.
- **Paso 6.**– Modifica el contenido de la posición en la que estás.

Estos 6 pasos simulan r pasos de la máquina M. Por lo tanto,

$$t_{M_1'}(x) \leq 2|x| + 6\lfloor \frac{T_M(x)}{r} \rfloor$$

donde $2n$ es el tiempo necesario para la compresión. En particular,

$$t_{M_1'}(n) \leq 2n + \frac{6}{r} T_M(n) \leq 2n + ct_M(n).$$

como pretendamos.

□

Capítulo 10

Euclides y Ecuaciones Diofánticas

El objetivo de este Capítulo es un estudio de la complejidad de los algoritmos que resuelven el siguiente problema de carácter diofántico :

Problema 10.0.1. Dados números enteros $a_1, \ldots, a_n, b \in \mathbb{Z}$, resolver la ecuación en $\mathbb{Z}^n$ definida mediante :

$$a_1X_1 + \cdots a_nX_n = b \qquad (*)$$

Para tratar de atacar este problema, su análisis de complejidad y algunos resultados al respecto, dividamos su análisis en dos partes : *Semántica y Sintaxis*. La Semántica se ocupará de la discusión de los objetos matemáticos que aparecen involucrados, mientras la sintaxis procederá a un estudio de la complejidad a partir de las componentes sintácticas de descripción de inputs y outputs. Debe señalarse que hemos utilizado las estimaciones de tiempo y espacio de las operaciones elementales con números enteros basadas en los algoritmos escolares.
Para poder expresar un estudio del problema, comencemos recordando algunos hechos matemáticos.

Proposición 10.0.1. *Si R es un dominio de ideales principales y M un $R-$módulo son equivalentes :*

- *M un módulo libre de torsión,*
- *M es un $R-$módulo libre.*

En particular, todo submódulo de un módulo libre sobre un dominio de ideales principales es libre y proyectivo a la vez.

Demostración. Ver cualquier texto clásico del Algebra

Obsérvese que $\mathbb{Z}$ es un dominio de ideales principales y que la ecuación $(*)$ es compatible si y solamente si el término independiente b está en el ideal $(a_1, \ldots, a_n)$ generado en el anillo $\mathbb{Z}$ por los coeficientes de la ecuación. En particular, tenemos la siguiente observación obvia :

Proposición 10.0.2. *La ecuación $(*)$ es compatible (consistente) si y solamente si*

$$gcd(a_1, \ldots, a_n) \mid b,$$

donde $gcd(a_1, \ldots, a_n)$ significa máximo común divisor de los coeficientes.

Como primer conclusión observamos que la discusión de la consistencia de una ecuación como $(*)$ nos lleva a un cálculo de máximos comunes divisores. Pero aún hay más,

Proposición 10.0.3. *Dados $a_1, \ldots, a_n \in \mathbb{Z}$, sea h su máximo común divisor. Entonces, existen $\alpha_1, \ldots, \alpha_n \in \mathbb{Z}$ tales que :*

$$h = \alpha_1 a_1 + \cdots + \alpha_n a_n$$

Más aún, los coeficientes de esta combinación lineal pueden obtenerse de tal modo que

$$||\alpha_i|| \leq (n-1) max\ \{||a_i||\ :\ 1 \leq i \leq n\} + 1,$$

donde $||a||$ para un número entero $a \in \mathbb{Z}$ denota su valor absoluto para el valor absoluto arquimediano usual en $\mathbb{R}$ (o el hermítico de $\mathbb{C}$).

La primera observación siendo evidente, nos conduce a la última en la que acotamos las tallas de los objetos que aparecen en la combinación lineal. No tratamos de obtener los mejores posibles (pregunta interesante que nos llevaría seguramente a la $NP-$completitud) sino unos suficientemente buenos. La idea de esta transformación es la siguiente : Supongamos sin pérdida de la generalidad que

$$||a_1|| = min\ \{||a_i||\ :\ 1 \leq i \leq n\}$$

Y escribamos

$$\alpha_i = q_i a_1 + r_i$$

siendo $0 \leq r_i < ||a_1||$, para $2 \leq i$.
Definamos

$$\alpha_i' := r_i, \quad \text{para} \quad i \geq 2$$

$$\alpha_1' := \alpha_1 + \sum_{i=2}^{n} q_i a_i$$

Tenemos la identidad siguiente :

$$h = \alpha_1' a_1 + \cdots + \alpha_n' a_n$$

En particular, tenemos la siguiente desigualdad :

$$||\alpha_1'||||a_1|| \leq ||h|| + \sum_{i=2}^{n} ||\alpha_i'||||a_i||$$

Tomando $A := max\ \{||a_i||\ :\ 1 \leq i \leq n\}$, obtenemos :

$$||\alpha_1'|| \leq 1 + (n-1)A$$

con lo que queda garantizado el anunciado.

□

Nótese que el enunciado anterior nos permite "reducir"los tamaños de los coeficientes que puedan aparecer en cualquier combinación lineal de las ajustadas. Pero además nos permite pasar a una definición semántica de lo que significa "resolver "la ecuación $(*)$ (Obsérvese mi permanente insistencia en las buenas definiciones de este tipo de nociones que siempre traen problemas por su imprecisión en la práctica).

Definición 10.0.1. Sea dada una ecuación diofántica del tipo $(*)$. Por "resolver.$^{\text{en}}$tendemos la respuesta a las siguientes preguntas y el cálculo de los objetos que se especifican :

i) Consistencia : La ecuación posee alguna solución en $\mathbb{Z}^n$?

ii) Resolución : En caso de respuesta afirmativa hallar :

- *Una solución particular :* Un punto $x := (x_1, \ldots, x_n) \in \mathbb{Z}^n$ solución de $(*)$

- *Una base de las sizygias :* Es decir, una base $\beta := \{v_1, \ldots, v_t\}$ del núcleo $Ker\varphi$ del morfismo de módulos libres :

$$\varphi : \mathbb{Z}^n \longrightarrow \mathbb{Z}$$

dado por :

$$\varphi(y_1, \ldots, y_n) := a_1 y_1 + \cdots + a_n y_n$$

Es claro que a partir de la información descrita en la resolución, uno dispone de todas las soluciones, puesto que

$$\forall y \in \mathbb{Z}^n \text{ solución de la ecuación } (*)$$

$$\exists r_1, \ldots, r_t \text{ tales que } y = x + \sum_{i=1}^{t} r_i v_i$$

De otro lado, supongamos que disponemos de una expresión del máximo común divisor de $a_1, \ldots, a_n$ de la forma :

$$h := \alpha_1 a_1 + \cdots + \alpha_n a_n$$

En este caso, y para un sistema compatible (i.e. $h \mid b$) se tiene el siguiente medio de calcular una solución particular : Sea $\lambda h = b$, con $\lambda \in \mathbb{Z}$. Una solución particular viene dada por :

$$x_1 := \alpha_1 \lambda, \ldots, X_n := \alpha_n \lambda$$

Esto justifica que el algoritmo de Euclides tenga la forma que a continuación vamos a darle. Por razones puramente didácticas hemos dividido el proceso en dos estudios independientes :

10.1. Caso $n = 2$.

Suponemos una ecuación del tipo :

$$a_1 X_1 + a_2 X_2 = b$$

Seguimos el algoritmo clásico de Euclides para la discusión de la Consistencia y, por ende, el cálculo del máximo común divisor :

$$a_1 := q_0 a_2 + r_0$$

$$a_2 := q_1 r_0 + r_1$$

$$\vdots$$

$$r_{n-2} := q_n r_{n-1} + r_n$$

$$r_{n-1} := q_{n+1} r_n$$

siendo, entonces,

$$r_n := gcd(a_1, a_2)$$

Es claro que este procedimiento permite concluir la consistencia mediante una división adicional, aunque no permite hallar una solución particular. Para remdiarlo introducimos unas variables locales

$$A_1^{(k)}, A_2^{(k)}$$

definidas del modo siguiente :

- *Inicializar :*

$$A_1^{(0)} := 1, A_2^{(0)} := -q_0$$

$$A_1^{(1)} := -q_1, A_2^{(1)} := 1 + q_1 q_0$$

- *Recursión :* Dadas $A_i^{(k-2)}, A_i^{(k-1)}$ para $1 \leq i \leq 2$, definimos :

$$A_1^{(k)} := A_1^{(k-2)} - q_k A_1^{(k-1)}$$

$$A_2^{(k)} := A_2^{(k-2)} - q_k A_2^{(k-1)}$$

Proposición 10.1.1. *Con las notaciones anteriores, la cadena de números enteros $A_i^{(k)}$ verifica :*

$$r_k := A_1^{(k)} a_1 + A_2^{(k)} a_2$$

para todo $k \geq 0$

Demostración. Mera comprobación siguiendo la secuencia de Euclides. □

Sin embargo, el proceso anterior así definido produce un crecimiento excesivo de las tallas de los resultados intermedios. Por mejorar ese crecimiento, supongamos $a_1 > a_2 > 0$ y definamos la siguiente nueva secuencia :

- *Inicializar :*

$$A_1^{(0)} := 1, A_2^{(0)} := -q_0$$

$$B_1^{(0)} := rem(A_1^{(0)}, a_2); B_2^{(0)} := quo(A_1^{(0)}, a_2)a_1 + A_2^{(0)}$$

$$A_1^{(1)} := -q_1 B_1^{(0)}, A_2^{(1)} := 1 - q_1 B_1^{(0)}$$

$$B_1^{(1)} := rem(A_1^{(1)}, a_2), B_2^{(1)} := quo(A_1^{(1)}, a_2)a_1 + A_2^{(1)}$$

- *Recursión :* Dadas $B_i^{(k-2)}, B_i^{(k-1)}$ para $1 \leq i \leq 2$, definimos :

$$A_1^{(k)} := B_1^{(k-2)} - q_k B_1^{(k-1)}$$

$$A_2^{(k)} := B_2^{(k-2)} - q_k B_2^{(k-1)}$$

$$B_1^{(k)} := rem(A_1^{(k)}, a_2)$$

$$B_2^{(k)} := quo(A_1^{(k)}, a_2)a_1 + A_2^{(k)}$$

donde *rem* y *quo* significan resto y cociente de los objetos indicados. Tenemos el siguiente resultado crucial :

Proposición 10.1.2. *Con las notaciones anteriores se tiene :*

i) Para todo $k \geq 0$:

$$r_k := B_1^{(k)} a_1 + B_2^{(k)} a_2$$

ii) Además :

$$||B_1^{(k)}|| \leq ||a_2||$$

$$||B_2^{(k)}|| \leq ||a_1|| + 1$$

Dicho de otra manera, los resultados intermedios que permiten calcular la combinación lineal que expresa el gcd en términos de las entradas tienen tallas bit lineales en el tamaño de la entrada. Finalmente, tenemos la siguiente proposición :

Proposición 10.1.3. *Con las anteriores notaciones, dados $a_1, a_2 \in \mathbb{Z}$ y siendo $h := gcd(a_1, a_2)$ se tiene que una base del módulo de sizygias puede obtenerse mediante :*

$$(-a_2', a_1')$$

donde

$$a_i' := \frac{a_1}{h}$$

Dicho de otra manera, el algoritmo de Euclides proporciona suficiente información para resolver completamente el caso $n = 2$.
Describamos el algoritmo de la manera obvia en la Figura 1.

INPUT : $a_1, a_2, b \in \mathbb{Z}$, supongamos $a_1 > a_2 > 0$

Variables locales (Cintas de Trabajo que vamos a activar)

$$V, W, A_1^{(1)}, A_2^{(1)}, A_1^{(2)}, A_2^{(2)}, B_1, B_2$$

Initialize :
Divide a_1 por a_2.
Si el resto es nulo hacer : $V := a_2, A_1^{(2)} := 1, A_2^{(2)} := -quo(a_1, a_2)$
else

- $V := rem(a_1, a_2), W := rem(a_2, V)$
- $A_1^{(1)} := rem(1, a_2); A_2^{(1)} := quo(1, a_2)a_1 - quo(a_1, a_2)$
- $A_1^{(2)} := rem(-quo(a_1, V), a_2), A_2^{(2)} := (1 - quo(a_1, a_2)quo(a_2, r_0)) + quo(-quo(a_1, V), a_2)a_1$
- $B_1 = 0, B_2 = 0$

Recursión :
While $W \neq 0$ do :

- Calcula $rem(V, W)$ y $quo(V, W)$
- $B_1 := A_1^{(1)} - quo(V, W)A_1^{(2)}$
- $B_2 := A_2^{(1)} - quo(V, W)A_2^{(2)}$
- $V := W, W := rem(V, W)$
- $A_1^{(1)} := A_1^{(2)}, A_2^{(1)} := A_2^{(2)}$
- $A_1^{(2)} := rem(B_1, a_2)$
- $A_2^{(2)} := B_2 + quo(B_1, a_2)a_1$

OUTPUT :

$$[V, A_1^{(2)}, A_2^{(2)}]$$

Fig. 1.– Descripción del Algoritmo de Euclides en el caso de 2 *enteros.*

Para analizar la complejidad de este procedimiento, nos falta solamente una guinda :

Proposición 10.1.4 (G. Lamé, 1845)**.** *Sea $n \in \mathbb{N}$. Sea $u > 0$ el menor número natural tal que existe $v \in \mathbb{N}$ verificando :*

$$u > v > 0$$

y el número de divisiones necesarias en el algoritmo de Euclides aplicado a u, v es n. Entonces,

$$F_{n+2} \leq u$$

donde F_k es el $k-$ésimo término en la sucesión de Fibonacci.

Demostración. Supongamos todo con números naturales y procedamos con la sucesión de Euclides usual :

$$u := q_0 v + r_0$$
$$v := q_1 r_0 + r_1$$
$$\vdots$$
$$r_{n-2} := q_n r_{n-1} + r_n$$
$$r_{n-1} := q_{n+1} r_n$$

donde se han realizado n divisiones y r_n es el gcd. Tenemos

$$r_n \geq 1 = F_1$$
$$r_{n-1} \geq 1 = F_2$$
$$r_{n-2} \geq r_{n-1} + r_n \geq F_3 = F_1 + F_2$$

$\vdots$

$$r_1 \geq F_{n-1}$$
$$r_0 \geq F_n$$
$$v \geq r_1 + r_0 \geq F_{n+1}$$
$$u \geq v + r_0 \geq F_{n+2}$$

□

En particular tenemos el siguiente Corolario :

Corollario 10.1.5. *Dados $u > v > 0$ dos números enteros, el número de divisiones que realiza el algoritmo de Euclides sobre u, v es. a lo más :*

$$n \leq \frac{log(\sqrt{5}u)}{\left(\frac{1+\sqrt{5}}{2}\right)} - 2$$

Demostración. Se basa en una acotación inferior de la sucesión de Fibonacci del tipo :

$$\frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^k \leq F_k$$

□

Una segunda cota inferior para el crecimiento de la sucesión de Fibonacci puede ser obtenido del modo siguiente :

Proposición 10.1.6. *Sea F_n el $n-$ésimo término de loa sucesión de Fibonacci. Entonces,*

$$F_n \geq 2^{\lfloor \frac{n}{2} \rfloor},$$

donde para cada número real $x \in \mathbb{R}$, $\lfloor x \rfloor \in \mathbb{Z}$ denota la parte entera de x.

Demostración. Utilizaremos el siguiente argumento inductivo

$$F_n = F_{n-1} + F_{n-2} = 2F_{n-2} + F_{n-3} \geq 2F_{n-2}.$$

Por lo tanto, para todo $k \in \mathbb{N}$, tendremos

$$F_n \geq 2^k F_{n-2k}.$$

Si $n = 2k$ habremos obtenido

$$F_n \geq 2^{n/2} F_0 = 2^{n/2}.$$

En otro caso $n = 2k + 1$, conducirá al resultados siguiente :

$$F_n \geq 2^k F_1 = 2^k.$$

En ambos casos,

$$F_n \geq 2^{\lfloor n/2 \rfloor},$$

como pretendíamos.

□

Para cada número natrural $x \in \mathbb{N}$, denotaremos por $|x|$ su talla en la codificación binaria, esto es su longitud cuando está representado sobre el alfabeto $\Sigma := \{\triangleright, 0, 1\}$. Un número natural $x \in \mathbb{N}$ queda pues identificado con una lista de dígitos $x_0 x_1 \cdots x_k \in \{0, 1\}^*$, donde

$$x := x_0 + 2x_1 + \cdots + x_k 2^k,$$

con $x_k \neq 0$. La longitud binaria de x será, en este caso

$$|x| = k + 1,$$

y se verificarán las siguientes relaciones :

$$\lfloor log_2 x \rfloor \leq |x| \leq \lfloor log_2 x \rfloor + 1.$$

Lema 10.1.7. *Las operaciones elementales de suma, resta, producto y división con resto de números naturales se realizan con las siguientes estimaciones de complejidad mediante los algoritmos escolares.*

- *Operación + de números naturales sobre el alfabeto binario : tiempo $O(n)$, espacio $O(n)$.*
- *Operación $\geq$ entre números naturales sobre el alfabeto binario (esto es, operación tal que dados $x, y \in \mathbb{N}$, decide si $x \geq y$? o no; además calcula el máximo de los dos números naturales dados) : tiempo $O(n)$, espacio $O(n)$.*
- *Operación $-$ de números naturales (esto es, operación $x - y$ cuasndo $x, y \in \mathbb{N}$, $x \geq y$) sobre el alfabeto binario : tiempo $O(n)$, espacio $O(n)$.*
- *Operación $\times$ de números naturales sobre el alfabeto binario : tiempo $O(n^2)$, espacio $O(n)$.*
- *Operación división con resto de números naturales sobre al alfabeto binario (esto es, dados $x, y \in \mathbb{N}$ calcular quo(x, y) el cociente de la división de x por y y rem(x, y), el resto de la división de x por y) : tiempo $O(n^2)$, espacio $O(n)$.*

Demostración. La estimación de estas complejidades se ha hecho dentro de la Hoja de Problemas I.

□

Observación 10.1.1. Debe observarse que estas estimaciones de complejidad son manifiestamente mejo- rables. Como se ha observado en la Hoja de problemas I, el algoritmo de Karatsuba permite realizar la multiplicación de números naturales en tiempo $O(n^{log_2 2})$ que es estrictamente mejor que la estimación $O(n^2)$. Asimismo, el algoritmo de Schonhage–Strassen de multiplicación rápida de números naturales, permite realizar esta multiplicaion en tiempo $O(nlog_2 nlog_2 log_2 n)$. Hemos elegido la simplificación de los métodos escolares para facilitar la comprensión del modelo de complejidad de las máquinas de Turing.

Siguiendo con la estructura de las operaciones elementales escolares, tendremos la siguiente estimación para las complejidades de las operaciones elementales con números enteros. Para ello, representaremos los números enteros sobre el alfabeto $\Sigma := \{\triangleright, 1, 0, -, +\}$. Un número entero $x \in \mathbb{Z}$ estará representado por una lista $\varepsilon x_0 x_1 \cdots x_k \in \{1, 0, -, +\}$, donde

- $\varepsilon \in \{+, -\}$
- $$||x|| = x_0 + 2x_1 + \cdots + 2^k x_k.$$

  Denotaremos por $|x|$ la talla sobre el alfabeto binario de un número entero $x \in \mathbb{Z}$. Entonces, se verificarán las siguienets propiedades.

$$\lfloor log_2||x|| \rfloor + 1 \leq |x| \leq \lfloor log_2||x|| \rfloor + 2.$$

Lema 10.1.8. *Sea $\Sigma := \{\triangleright, 1, 0, -, +\}$ el alfabeto finito del código binario de los números enteros. Las operaciones elementales de suma, resta, producto y división con resto de números enteros se realizan con las siguientes estimaciones de complejidad mediante los algoritmos escolares.*

- *Operación $+$ de números enteros sobre el alfabeto binario : tiempo $O(n)$, espacio $O(n)$.*
- *Operación $\geq$ entre números enetros sobre el alfabeto binario (esto es, operación tal que dados $x, y \in \mathbb{Z}$, decide si $x \geq y$? o no y, eventualmente calcula el máximo de los dos números enteros dados) : tiempo $O(n)$, espacio $O(n)$.*
- *Operación $-$ de números enteros sobre el alfabeto binario : tiempo $O(n)$, espacio $O(n)$.*
- *Operación $\times$ de números enteros sobre el alfabeto binario : tiempo $O(n^2)$, espacio $O(n)$.*
- *Operación división con resto de números enteros sobre al alfabeto binario (esto es, dados $x, y \in \mathbb{N}$ calcular $quo(x, y)$ el cociente de la división de x por y y $rem(x, y)$, el resto de la división de x por y) : tiempo $O(n^2)$, espacio $O(n)$.*

Demostración. Es un ejercicio elemental a partir del Lema anterior puesto que sólo debemos aadir las condieraciones sobre el comportamiento del signo.

□

Con todo esto observado es fácil concluir el siguiente enunciado :

Teorema 10.1.9 (Complejidad del Algoritmo de Euclides en el caso de 2 variables)**.** *Existe una máquina de Turing que resuelve una ecuación diofántica del tipo*

$$a_1 X_1 + a_2 X_2 = b$$

en tiempo $O(n^3)$ y espacio $O(n)$ donde n es la talla de la entrada

$$\triangleright a_1 \sharp a_2 \sharp b$$

sobre el alfabeto

$$\Sigma := \{\triangleright, 0, 1, \sharp, -\}$$

Demostración. La prueba ha sido hecha con las disquisiciones anteriores y basta con verificar la cota tiene el buen comportamiento deseado.
Como conclusiones tenemos las siguientes estimaciones de complejidades para las operaciones aritméticas elementales sobre los cuerpos primos.

Corollario 10.1.10. *Sea p un número primo y $\mathbb{F}_p$ el cuerpo primo de p elementos. Entonces, las operaciones en $\mathbb{F}_p$ tienen las siguientes complejidades :*

i) Operaciones $+, -$ tienen complejidad en tiempo $O(log_2 p)$ y necesitan espacio $O(log_2 p)$.

ii) Operación $\times$ *: Tiene complejidad en tiempo* $O(log_2^2 p)$ *y necesita espacio* $O(log_2 p)$*. item Operacion* $^{-1}, /$ *: Tiene una complejidad en tiempo* $O(log_2^3 p)$ *y necesita un espacio* $O(log_2 p)$*.*

Demostración. Obsérvese simplemente que el cálculo del inverso a^{-1} de un número no nulo $a \in \mathbb{F}_p$ se hace resolviendo la siguiente ecuación diofántica :

$$AX + pX = 1.$$

□

Obsérvse asimismo que el código de un elemento del cuerpo primo $\mathbb{F}_p$ es simplemente uno de los posibles restos módulo p. En otras palabras, para cada $x \in \mathbb{F}_p$, se tiene

$$|x| \leq \lfloor log_2 p \rfloor + 1.$$

Los números racionales, en cambio serán representados sobre el alfabeto

$$\Sigma := \{\triangleright, 0, 1, -, +, |\}.$$

Un número racional vendrá dado como una lista

$$a \mid b,$$

donde $a, b \in \mathbb{Z}$ viene representados sobre el alfabeto $\{0, 1, +, -\}$ de los números enteros y deben satisfacer las siguientes propiedades

- $gcd(a, b) = 1$,
- $b \neq 0$,
- $b \in \mathbb{N}$.

En particular la talla de un número racional sobre el alfabeto indicado, viene dada por la siguiente estimación :

$$|x| = |a| + |b| + 1,$$

lo que, a su vez, tiene la siguiente estimación :

$$\lfloor log_2 ||a|| \rfloor + \lfloor log_2 ||b|| \rfloor + 2 \leq |x| \leq \lfloor log_2 ||a|| \rfloor + \lfloor log_2 ||b|| \rfloor + 3.$$

Corollario 10.1.11. *El cuerpo* $\mathbb{Q}$ *de los números racionales es un cuerpo computable, siendo sus elementos codificados sobre el alfabeto* $\Sigma := \{\triangleright, 0, 1, -, +, |\}$ *en la forma*

$$\triangleright a \mid b$$

donde $a, b \in \mathbb{Z}$, $b \in \mathbb{N} \setminus \{0\}$, $gcd(a, b) = 1$*. Además sus operaciones elementales tienen las siguientes complejidades :*

i) Operaciones $+, -$ *: tiempo* $O(n^3)$ *espacio* $O(n)$*.*

ii) Cálculo del opuesto para la $+$ *: tiempo* $O(n^3)$*, espacio* $O(n)$*,*

iii) Operaciones $\times$ *: tiempo* $O(n^3)$*, espacio* $O(n)$*,*

iv) Operaciones $/, ^{-1}$ *: Tiempo* $O(n^3)$*, espacio* $O(n)$*,*

v) Operaciones $= 0?, =?$ *: Tiempo* $O(n^2)$*, espacio* $O(t)$*.*

vi) Operación cálculo del representante canónico :

$$\textit{tiempo} \quad O(n^3), \quad \textit{espacio} \quad O(n)$$

Donde n *es la talla del input considerado.*

Obsérvese que la suma de dos números racionales es casi tan compleja (o incluso más) qeu su producto por cuento necesitamos al menos 3 multiplicaciones de núemros naturales para realizar la suma y solamente 2 para realizar el producto. Obsérvese asimismo que todas las operaciones quedan afectadas por el cálculo del representante canónico del número racional considerado y, por ende, por el uso del algortimo de Euclides.

Definición 10.1.1. Llamaremos altura logarítmica (o simplemente altura) de un número racional $x \in \mathbb{Q}$, al número

$$ht(x) := max\{log_2 \mid a \mid, log_2 \mid b \mid : \ x = \frac{a}{b}, \wedge gcd(a,b) = 1\}$$

La noción de altura logarítmica de un número racional es una noción esencial en Aproximación y Geometría Diofánticas. Introducide originalmente por A. Weil para estudios de problemas de transcendencia de números reales, tiene una traducción clara a los asuntos de complejidad. Se tienen las siguientes relaciones entre altura logarítmica y talla binaria de un número racional $x \in \mathbb{Q}$:

$$ht(x) + 2 \leq |x| \leq 2ht(x) + 3.$$

10.2. Caso $n \geq 3$.

La estrategia en este caso es sutilmente diferente (aunque sólo en apariencia). Tratamos de resolver el problema diofántico (sobre $\mathbb{Z}^n$) siguiente : Hallar todas las soluciones de la ecuación :

$$a_1X_1 + \cdots + a_nX_n = b \qquad (*)$$

Uno podría seguir dos estrategias básicas.

- Construir un algoritmo de cálculo del gcd para n números enteros.
- Utilizar el algoritmo de cálculo del gcd para dos números naturales.

Comencemos con el primero y después expresemos las diversas estrategias propias del primero.

10.2.0.1. Divisiones–Reducciones.

La idea de hacer divisiones reducciones se plantea del modo siguiente :

- Comenzamos con la lista de coeficientes :

$$\beta := [a_1, \ldots, a_n]$$

- Elegimos i tal que

$$||a_i|| := min\{||a_k|| \ : \ 1 \leq k \leq n\}$$

- Escribimos

$$\beta := [a_i, a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n\}$$

- efectuamos las divisiones a_k entre a_i para $k \neq i$ y escribimos :

$$a_1 := a_i$$

$$a_k := rem(a_{k-1}, a_i) \quad \text{para} \ \ 2 \leq k \leq i$$

$$a_k := rem(a_k, a_i) \quad \text{para} \ \ i+1 \leq k \leq n$$

- Escribimos

$$\beta := [a_1, \ldots, a_n]$$

- Si $a_k = 0$ para $2 \leq k \leq n$, terminamos exhibiendo β En caso contrario, volvemos a empezar.

Nótese que la instrucción que se utiliza es un while y que podríamos haber escrito :

Initialize :

$$\beta := [a_1, \ldots, a_n]$$

While there exists k, $2 \leq k \leq n$ s.t. $a_k \neq 0$ do

Choose i s.t.

$$||a_i|| := min\{||a_k|| \ : \ 1 \leq k \leq n\}$$

Write

$$\beta := [a_i, a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n\}$$

Do :

$$a_1 := a_i$$

$$a_k := rem(a_{k-1}, a_i) \quad \text{para} \ \ 2 \leq k \leq i$$

$$a_k := rem(a_k, a_i) \quad \text{para} \ \ i+1 \leq k \leq n$$

Write

$$\beta := [a_1, \ldots, a_n]$$

Output β
end

Fig.2.– Una descripción de Euclides como un proceso de Divisiones–Reducciones.

Proposición 10.2.1. *El anterior proceso da como resultado una lista $\beta := [h, 0, \ldots, 0]$ tal que*

$$h := gcd(a_1, \ldots, a_n)$$

Hay dos inconvenientes a considerar en la anterior construcción. De una parte, tenemos que acotar el número de veces que se ejecuta el while. De otra parte, la información del gcd sólo nos aporta respuesta a la cuestión de la consistencia, pero no nos ayuda a resolver. Para tratar de hacer ambos intentos, usaremos las siguientes consideraciones.
Sea $\beta_0, \ldots, \beta_m$ los sucesivos conjuntos obtenidos por este proceso hasta lograr que

$$\beta_m := [h, 0, \ldots, 0]$$

siendo

$$\beta_0 := [a_1, \ldots, a_n]$$

Entonces, tenemos la siguiente Proposición :

Proposición 10.2.2. *Sea*

$$u := max\{log_2||a_i|| \ : \ 1 \leq k \leq n\}$$

Entonces,

$$m \in O(log_2 u)$$

Demostración. La prueba sigue los mismos pasos que el caso de dos variables y el resultado de Lamé. Supongamos que todos son positivos por simplicidad. Comencemos denotando :

$$u_k := ||\beta_k|| := max\{||a_i^{(k)}|| \ : \ 1 \leq k \leq n\}$$

donde

$$\beta_k := [a_1^{(k)}, \ldots, a_n^{(k)}]$$

Tenemos

$$u_m \geq 1$$

$$u_{m-1} \geq 1$$

$$u_k \geq F_{n+2-k}$$

$$u_0 \geq F_{n+2}$$

usando la cota inferior de la sucesión de Fibonacci tenemos el resultado buscado.

□

De otro lado, haremos la siguiente construcción :
donde Id_n es la matriz identidad $n \times n$. El procedimiento que seguiremos es simple : Sobre la matriz superior haremos exactamente los cálculos diseñados por el algoritmo anterior, sobre la matriz de abajo simplemente ejecutaremos, por columnas las operaciones que se señalan en la primera fila.
Al final del proceso, llegaremos a una estructura del tipo siguiente : donde la matriz Q ha sido obtenida des la matriz identidad con los siguientes tipos de operaciones elementales :

- Intercambiar dos columnas.
- Sumar a una columna otra columna multiplicada por una constante entera

Proposición 10.2.3. *La matriz Q tiene determinante unidad (es unimodular) en $\mathbb{Z}$ y, por tanto, define un isomorfismo de grupos abelianos :*

$$\mathbb{Z}^n \longrightarrow \mathbb{Z}^n$$

mediante
Además, la matrix anterior verifica :

$$(a_1, \ldots, a_n)Q := (h, 0, \ldots, 0)$$

Las columnas de la matriz regular Q definen una base de $\mathbb{Z}^n$ como $\mathbb{Z}-$módulo. La primera columna nos da una relación de los datos con el gcd : Entonces,

$$h := q_{1,1}a_1 + \cdots + q_{n,1}a_n$$

Siendo las $n-1$ columnas de la matriz Q una base del núcleo $Ker\varphi$. Para demostrar este hecho, obsérvese que la relación :

$$(a_1, \ldots, a_n)Q := (h, 0, \ldots, 0)$$

implica que para todo $v = (x_1, \ldots, x_n) \in \mathbb{Z}^n$ ortogonal a $(a_1, \ldots, a_n)$ se tiene :

$$v := y_1v_1 + \cdots + y_nv_n$$

siendo $v_1, \ldots, v_n$ las columnas de la matriz Q. En particular concluimos que, dado que $h \neq 0$ (en otro caso no habría ecuación que considerar) la igualdad anterior se trasnforma en

$$hy_1 = 0$$

y tenemos la afirmación buscada.

Para analizar la complejidad del proceso, haremos intervenir la cota de la Proposición 10.2.2.
Demostración. Obsérvese que las operaciones realizadas en cada etapa son, o bien de cambiar una columna por otra (reescritura) o bien sumar a una columna otra multiplicada por un número entero. Obsérvese, además, que el número entero utilizado es el cociente de una división, por cuento su tamaño está acotado por el tamaño del dividendo. Finalmente, obsérvese que se inicializa con la matriz identidad.
Pongamos que la ejecución del proceso tiene m pasos de cálculo, con matrices

$$Q_0 := Id_n, Q_1, \ldots, Q_m$$

Definamos

$$||Q_k|| := max\{||a|| \ : \ a \text{ es una coordenada entera de } Q_k\}$$

Tenemos

$$||Q_0|| = 1$$

Para Q_1 a lo más hemos sumado una columna a la otra multiplicada por una cantidad acotada por el máximo de los valores absolutos de las coordenadas del input (sea A esta cantidad) Luego las nuevas columnas de Q_1 tienen coordenadas acotadas por :

$$||Q_1|| \leq (A+1)$$

Finalmente, si observamos que Q_{k+1} se obtiene de Q_k a partir de una operación de las anteriores, tendremos :

$$||Q_{k+1}|| \leq (A+1)||Q_k||$$

En conclusión, obtenemos la cota :

$$||Q_k|| \leq (A+1)^k$$

Dado que el número de pasos de cálculo está acotado por $O(log_2 A)$, los números que aparecen en los sucesivos cálculos está acotados por

$$(A+1)^{O(log_2 A)} = 2^{O(log_2^2 A)}$$

en otras palabras, su talla bit está acotada por $O(log_2^2 A)$. Obsérvese además que las divisiones se realizan solamente entre los elementos de la primera fila y que el número de divisiones en cada etapa es $n-1$, luego el proceso de dividir se realiza en tiempo $(n-1)O(log_2^2 A)$ porque los objetos que aparecen son restos y siempre son menores en valor absoluto que A. Las operaciones sobre la matriz son $n-1$ sumas de la primera columna multiplicada por un número menor que A a las demás columnas. Esto supone, para cada columna, n multiplicaciones, n sumas, de números de talla bit a lo sumo $O(log_2^2 A)$ por otro de talla $O(log_2 A)$. Esto produce, para cada columna, un tiempo del orden $nO(log_2^3 A)$ y hay que hacerlo para $n-1$ columnas, luego cada paso cuesta

$$O(n^2 log_2^3 A)$$

Como hay $O(log_2 A)$ iteraciones, el coste total será del orden :

$$O(n^2 log_2^4 A)$$

contando intercambios de columnas y otras menudencias.
En cuanto al espacio, debemos guardar siempre : n números de talla bit acotada por $O(log_2 A)$ (la primera fila) n^2 números de talla bit acotada por $O(log_2^2 A)$ (la matriz)
Luego el espacio de trabajo es $O(n^2 log_2^2 A)$. Lo que conduce al siguiente Teorema :

Teorema 10.2.4. *Existe una máquina de Turing M que resuelve ecuaciones diofánticas del tipo :*

$$a_1 X_1 + \cdots + a_n X_n = b$$

El tiempo de cálculo de M es del orden :

$$O(n^2 log_2^4 A + n log_2 ||b||)$$

Y el espacio necesario para su ejecución es del orden :

$$O(n^2 log_2^2 A + log_2 ||b||)$$

donde

$$A := max\{||a_i|| \ : \ 1 \leq i \leq n\}$$

En términos de la talla de la entrada n, la complejidad de este algoritmo es del orden :

$$T_M \in O(n^6), \ \ S_M \in O(n^4)$$

Parte III

Algunos Rudimientos con la Complejidad Computacional

Capítulo 11

Clases de Complejidad y Primeras Relaciones

Contents

11.1. Terminología Básica. 219
11.2. El indeterminismo en juego. 220
11.2.1. Clases de complejidad funcionales 223
11.3. Mezclando Clases. 227
11.3.1. Teoremas de Jerarquía. 227
11.3.2. Unas palabras sobre grafos orientados. 232
11.3.2.1. Clausura Transitiva 233
11.3.2.2. Alcanzable 234
11.3.3. Una codificación más corta de las configuraciones 234
11.3.4. Espacio indeterminista frente a tiempo determinista. 235
11.3.5. Tiempo indeterminista frente a espacio determinista. 236
11.4. El Teorema de Savitch. 236
11.5. Un pequeño grafo con las relaciones conocidas. 237

Este Capítulo está dedicado a establecer algunas propiedades generales de las clases de complejidad tal y como se han presentado en el Capítulo Rudimentos con máquinas de Turing anterior (Capítulo 9). Lo esencial de estos contenidos responde a un estándar de la Complejidad Estructural y puede seguirse también en textos como [Papa, 94], [BaDiGa, 88], [WaWe, 86]. Recordemos que el tratamiento de la complejidad en términos de complejidad de máquinas de Turing se remonta a [HaSt, 65].

11.1. Terminología Básica.

Comenzaremos por una somera descripción del significado del indeterminismo, mediante una caracterización usando el *guessing*. Antes de comenzar la discusión, habida cuenta de los resultados demostrados en el Capítulo de Rudimentos con máquina de Turing, nos couparemos solamente de funciones de complejidad (en tiempo y/o espacio) del caso peor definidas salvo una constante (es decir clases de complejidad en términos asintóticos). Para ello introducimos la siguiente notación :

Definición 11.1.1. Con las notaciones descritas en la Capítulo referente a Rudimentos, sea $f : \mathbb{N} \to \mathbb{R}_+$ una función monótona creciente. Definiremos, las siguientes clases de complejidad :

$$\mathbf{DTIME}(f) := \bigcup_{g \in O(f)} DTIME(f),$$

$$\mathbf{NTIME}(f) := \bigcup_{g \in O(f)} NTIME(f),$$

$$\mathbf{DSPACE}(f) := \bigcup_{g \in O(f)} DSPACE(f),$$

$$\mathbf{NSPACE}(f) := \bigcup_{g \in O(f)} NSPACE(f),$$

11.2. El indeterminismo en juego.

El indeterminismo aparece como noción relevante a partir de la introduciión de la clase **NP**. Esta clase fue introducida por S. Cook ([Cook, 71]), aunque el formalismo es debido a Richard Karp ([Krp, 72]). Todo comienza a princios de los años 70.
La idea del indeterminismo se expresa bien mediante la diferencia existente en Buscar una solución y Comprobar que un cierto candidato es solución. El problema hubiera sido meramente filosófico si no fuera porque La mayor parte de los problemas naturales esenciales para la informática tiene la cualidad siguiente "es fácil comprobar si un candidato es solución, pero nadie sabe resolverlos de manera eficaz". Más abajo veremos algunos de esos ejemplos clásicos. Un ejemplo, propuesto por el propio S. Cook, que ayuda a comprender el fenómeno a los no iniciados es el siguiente :
"... Supongamos que te invitan a una fiesta en la que temes aburrirte y en la que, probablemente, no conoces a nadie. Supongamos que, al llegar, el anfitrión te dice :
– Hemos invitado a un amigo tuyo, John Smith. Lleva una camisa roja y un pantalón negro y estaba en el salón hace un momento.
En este caso, un vistazo rápido a las personas que se encuentran en el salón te ayuda a encontrar con cierta rapidez a ese conocido. Sin embargo, supongamos que el anfitrión nos dice :
– Hemos invitado a algún amigo tuyo; pero no sé ni cómo se llama, ni si ha venido o no.
En este segundo caso te ves obligado a buscar por la casa, uno por uno, a todos los invitados hasta comprobar si está o no está alguno de tus conocidos.
Claramente, en el primer caso tardarás poco en encontrar a alguien conocido con quien entablar conversación, mientras que, en el segundo caso, puedes pasarte toda la fiesta verificando uno por uno a los invitados, sin dar con nadie conocido.
Esta sencilla idea determina el llamado indeterminismo. En el primer caso aparece un candidato a solución a tu problema (*John Smith, con una camisa roja, un pantalón negro y está en el salón*). En el segundo caso no dispones de candidato a solución y debes verificar sistemáticamente con todos tus conocidos y todos los invitados por ver si conoces a alguien. Si la fiesta es grande y hay muchos individuos que vienen y se van, es probable que no llegues a saber nunca si algún conocido tuyo estuvo en la fiesta.
La noción es muy común en matemáticas y nos recuerda la experiencia personal de cada uno de nosotros. Dado un problema matemático $\mathcal{P}$ y dado un candidato a solución x, suele ser fácil verificar si x es o no solución, pero no suele ser tan fácil Calcular la Solución. Un ejemplo sencillo es el cálculo de funciones primitivas. Derivar una función dada como composición de funciones elementales es un proceso de reescritura que nos permite comprobar si un "candidato" F es o no la primitiva de una función f dada. En cambio, hallar F sin información adicional suele ser una tarea compleja.
En informática, esta dualidad entre Comprobación y Cálculo de una solución se expresa mediante el proceso de Guessing.
Para explicar este fenómeno, comencemos discutiendo algunos ejemplos de problemas naturales :

Ejemplo 11.2.1.

Problema (PRIMES). Dado un número natural, decidir si este número es un número primo o no.

El lenguaje, en este caso puede ser

$$PRIMES := \{n \in \mathbb{N} \ : \ n \text{ es un número primo}\}$$

o bien

$$COMP := \{n \in \mathbb{N} \ : \ n \text{ no es un número primo}\}$$

Es fácil encontrar un algoritmo indeterminista para el lenguaje $COMP$ que funcione en tiempo polinomial.

INPUT : $n \in \mathbb{N}$

Guess $a, b \in \mathbb{N}$
if $a \times b - n = 0$ then ACCEPT
else NO PARES
fi
END

El proceso del "guessing" de la elección de un candidato (certificado) es esencial para comprender el fenómeno que yace bajo el indeterminismo.
Para hacer un guessing basta con poner en la función de transición de una máquina de Turing una subrutina tras el estado inicial. Esta subrutina procede como sigue :
Primero escribimos en una cinta de trabajo (digamos en la primera cinta de trabajo $WT1$) el contenido de input. Después, dando una opción indeterminista que consiste en un "guessing" (escribiendo algo tras el input en $WT1$), más tarde se procede con una máquina de Turing determinística M que efecta sus cálculos como si el contenido de $WT1$ fuera su cinta de input. Este proceso que modeliza al guessing, puede reescribirse mediante :

INPUT : $x \in \Sigma^*$

Guess $y \in \Sigma^*$
Apply M on $xy \in \Sigma^*$
ACCEPT if and only if M accepts $xy \in \Sigma^*$
end

Formalemente, podemos definir las máquinas de Turing con guessing del modo siguiente:

Definición 11.2.1 (Máquinas de Turing con guessing). Una máquina de Turing con guessing sobre un alfabteo Σ es una máquina de Turing indeterminística $M = (Q, \Sigma, q_0, F, \delta)$ con un estado especial **Guess** $\in Q$ y una cinta especial (cinta de guessing of WT1) que verifica las propiedades siguientes:

i) Dada una lectura $\ell := (q; x_0, x_1, \ldots, x_k) \in Q \times (\Sigma \cup \{\triangleright, \square)^{k+1}$ si $q \neq$ **Guess**,

$$\sharp(\delta(\ell)) \leq 1.$$

Es decir, cuando el estado no es **Guess**, la función de transición se comporta como una máquina determinística.

ii) Dada una lectura con estado **Guess**, $\ell := (\textbf{Guess}; x_0, x_1, \ldots, x_k) \in Q \times (\Sigma \cup \{\triangleright, \square)^{k+1}$, el resultado de la función de transición $\delta(\ell)$ puede tener más de una imagen. Los elementos $s \in \delta(\ell)$ pueden ser de dos tipos:

a) Las instrucciones descritas en s vuelven a estado **Guess**. En ete caso, se puede escribir cualquier símbolo en la cinta de "guessing" (WT1) y el resto de las cintas no se modifican. Esto es, si

$$s := (\mathbf{Guess}; \alpha, x_2, \ldots, x_k; 0, +1, 0, \ldots, 0),\ \alpha \in I \subseteq \Sigma.$$

Nótese que la cinta de "guessing" avanza un paso a la derecha después de escribir α. Nótese también que no necesariamente todos los símbolos de Σ son utilizados, sino solamente algunos en un conjunto $I \subseteq \Sigma$.

b) Las instrucciones descritas en s pasan a un estado p distinto de **Guess**. En ete caso, no se modifica ninguna de las cintas y se pasa a la parte determinística del proceso.

$$s := (p; x_1, x_2, \ldots, x_k; 0, 0, 0, \ldots, 0),\ \alpha \in I \subseteq \Sigma.$$

Nótese que la cinta de "guessing" no se mueve y re-comienza un proceso determinístico.

El proceso indeterminista así generado tiene un sentido claro en las matemáticas más clásicas : Supongamos que nos dan un problema que implica la búsqueda de una solución. En realidad tenemos dos subproblemas significativos :

- RESOLVER. El problema original que consiste en buscar una solución.
- COMPROBAR. Suponiendo que alguien nos da una solución, podemos decidir si la solución es buena o no.

El "guessing" es el proceso que nos introduce *un candidato a solución o "certificado"* dentro de un proceso determinístico. Sin embargo, comprobar si es solución lo podemos hacer determinsticamente. Así, para los problemas donde se pregunta sobre la existencia de una solución, puede ocurrir que comprobar si algo es solución sea fácil, mientras hallar una solución no lo parezca tanto.
Algunos ejemplos más :

Problema (**El problema de la mochila (Knapsack Problem,KnP)).** Sean dados números naturales $x_1, \ldots, x_n, k \in \mathbb{N}$. Decidir si

$$\exists S \subseteq \{1, \ldots, n\},\ \sum_{i \in S} x_i = k$$

Problema (**El problema del Viajante (Travelling Salesman Problem, TSP)).** Sea dada una matriz triangular superior con coordenadas números naturales :

$$A := (a_{i,j})_{1 \leq i,j \leq n}$$

Y sea dado $k \in \mathbb{N}$. Decidir si

$$\exists \sigma \in \Sigma_n,\ \sum_{i=1}^{n} a_{\sigma(i),\sigma(i+1)} \leq k$$

donde $\sigma(n+1)$ se supone igual a $\sigma(1)$.

En ambos problemas la comprobación de un candidato "guessado" es fácil; aunque para hallar una posible solución sea necesario probar respectivamente con todos los subconjuntos S de $\{1, \ldots, n\}$ o con todas las permutaciones σ del grupo de permutaciones Σ_n. Esto nos lleva en el primer caso a 2^n pruebas y en el segundo a $n!$ pruebas.
Otros ejemplos de este mismo pelaje son los que siguen :

Ejemplo 11.2.2 (**Polinomios Compuestos e Irreducibles).**

$$COMPOL := \{f \in \mathbb{Z}[X] \ :\ f \text{ es primitivo y reducible en } \mathbb{Q}[X]\}$$

$$PRIMPOL := \{f \in \mathbb{Z}[X] \ :\ f \text{ es primitivo y primo en } \mathbb{Q}[X]\}$$

Algoritmos indeterministas que resuelve $COMPOL$ son fáciles de diseñar. Los algoritmos deterministas serán mostrados en capítulos posteriores.
Un ejemplo más de relativa importancia es el siguiente :

11.2.1. Clases de complejidad funcionales

Sea dada una función recursiva

$$\varphi \ : D(\varphi) \subseteq \Sigma^* \longrightarrow \Sigma^*$$

Llamaremos tiempo de esta función al tiempo de la mejor máquina de Turing que acepta $L(M) = D(\varphi)$ y tiene como resultado $Res_M = \varphi$. De modo análogo haremos con la noción de espacio. Podemos así definir clases de complejidad de funciones de la misma forma que lo hemos hecho con los problemas decisionales :

$$\mathbf{DTIMEF}(f), \mathbf{NTIMEF}(f), \mathbf{DSPACEF}(f), \mathbf{NSPACEF}(f)$$

Nos podemos preguntar si un estudio de las complejidades de la evaluación de funciones está suficientemente garantizado con el estudio de los problemas decisionales siguientes :

Problema 11.2.1 (Pertenencia al grafo.)**.** Dados $x, y \in \Sigma^*$ decidir si $(x, y) \in Gr(\varphi)$.

Problema 11.2.2 (Pertenencia a la Imagen)**.** Con las notaciones anteriores, sea $Im(\varphi) := \varphi(D(\varphi))$ la imagen de φ. Entonces, dado $y \in \Sigma^*$, decidir si $y \in Im(\varphi)$.

Este procedimiento pasa de problemas de evaluación a problemas decisionales.

Proposición 11.2.1. *Con las anteriores notaciones, tenemos la siguiente cadena de implicaciones.*

- *Si* $\varphi \in \mathbf{DTIMEF}(f) \Rightarrow Gr(\varphi) \in \mathbf{DTIME}(2f)$
- *Si* $\varphi \in \mathbf{DSPACEF}(f) \Rightarrow Gr(\varphi) \in \mathbf{DSPACE}(f + max\{|\varphi(x)| \ : \ x \in D(\varphi), \ |x| \leq n\})$

Demostración. Son evidentes, dado $(x, y) \in (\Sigma^*)^2$, primero evaluamos $\varphi(x)$. Si coincide con y, habremos terminado aceptando, si no coincide (o si y es demasiado grande, en comparación con $\varphi(x)$), no aceptaremos. □

El indeterministmo puede aparecer de dos formas que se contemplan en las Proposiciones siguientes:

Proposición 11.2.2. *Con las anteriores notaciones, supongamos que* $f : \mathbb{N} \longrightarrow \mathbb{N}$ *es una función monótona creciente que verifica:*

i) la función f *es super-linear, es decir* $f(n) \geq n$, $\forall n \in \mathbb{N}$,

ii) f *es tiempo constructible (ver Secciones siguientes),*

iii) para todo $x \in D(\varphi)$, $|\varphi(x)| \leq f(|x|)$.

Entonces, se tiene:

- *Si* $Gr(\varphi) \in \mathbf{DTIME}(f) \Rightarrow \varphi \in \mathbf{NTIMEF}(g)$,
- *Si* $Gr(\varphi) \in \mathbf{DSPACE}(f) \Rightarrow \varphi \in \mathbf{NSPACEF}(g)$,

donde $g : \mathbb{N} \longrightarrow \mathbb{N}$ *es la función dada mediante:*

$$g(n) := f(2f(n)), \ \ \forall n \in \mathbb{N}.$$

Demostración. Aquí se hace aparecer el guessing de manera natural. El procedimiento sería como sigue. Supongamos que existe una máquina de Turing que decide $Gr(\varphi)$ en tiempo acotado por f. Definamos:

Input: $x \in \Sigma^*$
Eval: $f(|x|)$ (en tiempo $O(f)$ porque es constructible en tiempo)

Guess indeterministically: $y \in \Sigma^*$, $|y| \leq f(|x|)$.
if si $(x, y) \in Gr(\varphi)$, OUTPUT: y
else, OUTPUT: NO IDEA
fi
end

En primer lugar, sabemos que $|\varphi(x)| \leq f(|x|)$, luego la imagen y, si es imagen, debe tener talla acotada por $f(|x|)$. De donde se sigue que el guessing está bien limitado. Tras haber seleccionado el guessing, sólo nos queda verificar que $(x, y) \in Gr(\varphi)$. Pero ésto se hace en tiempo acotado por

$$f(|x| + |y|) \leq f(|x| + f(|x|)) \leq f(f(|x|) + f(|x|)),$$

por la primera de las hipótesis dobre f.

□

También la Imagen tiene una interpretación reseñablemente comparable. Será el uso más habitual en la omprensión del indeterminismo:

Proposición 11.2.3. *Con las anteriores notaciones, supongamos que $f : \mathbb{N} \longrightarrow \mathbb{N}$ es una función monótona creciente que verifica:*

i) la función f es super-linear, es decir $f(n) \geq n$, $\forall n \in \mathbb{N}$,

ii) f es tiempo constructible (ver Secciones siguientes),

iii) para todo $y \in Im(\varphi)$, exite $x \in D(\varphi)$ tal que $|x| \leq f(|y|)$, con $\varphi(x) = y$.

Entonces, se tiene:

- *Si $D(\varphi), Gr(\varphi) \in \mathbf{DTIME}(f) \Rightarrow Im(\varphi) \in \mathbf{NTIMEF}(g)$,*
- *Si $D(\varphi), Gr(\varphi) \in \mathbf{DSPACE}(f) \Rightarrow Im(\varphi) \in \mathbf{NSPACEF}(g)$,*

donde $g : \mathbb{N} \longrightarrow \mathbb{N}$ es la función dada mediante:

$$g(n) := 2f(2f(n)), \quad \forall n \in \mathbb{N}.$$

Demostración. La prueba es similar a la anterior. Supongamos que existe una máquina de Turing que decide $Gr(\varphi)$ en tiempo acotado por f. Definamos:

INPUT: $y \in \Sigma^*$
Eval: $f(|y|)$ (en tiempo $O(f)$ porque es constructible en tiempo)
Guess indeterministically: $x \in \Sigma^*$, $|x| \leq f(|y|)$.
Decidir si $x \in D(\varphi)$
if si $(x, y) \in Gr(\varphi)$, OUTPUT: YES
else, OUTPUT: NO IDEA
fi
end

En primer lugar, sabemos que dado y, si $y \in Im(\varphi)$, es porque existe $x \in D(\varphi)$ $|x| \leq f(|y|)$. Por tanto, el "guessing" de la anti-imagen $\varphi^{-1}(y)$ lo podemos buscar entre aquellos x con $|x| \leq f(|y|)$. Tras haber seleccionado el guessing, sólo nos queda verificar primero que $x \in D(\varphi)$ y luego que (x, y) está en $Gr(\varphi)$. Ésto se hace en tiempo acotado por dos veces la cantidad siguiente:

$$f(|x| + |y|) \leq f(|x| + f(|x|)) \leq f(f(|x|) + f(|x|)),$$

por la primera de las hipótesis sobre f. □

Finalmente, el "guessing" tiene una interpretación geométrica como proyección o como prefijos, aunque aquí lo de geométrico sea más bien una sutil idea que será entendida más adelante.

Proposición 11.2.4. *Los lenguajes $L \subseteq \Sigma^*$ aceptados mediante una máquina de Turing indeterminística via el proceso de "guessing" son exactamente las proyecciones, a través de*

$$\pi \ : \Sigma^* \times \Sigma^* \longrightarrow \Sigma^*$$

de los lenguajes aceptados por máquinas de Turing deterministas. Más aún, las funciones de tiempo y espacio se relacionan del modo siguiente (M es la máquina indeterminista que hace "guessing" y N es la máquina que acepta $L' \subseteq \Sigma^* \times \Sigma^*$ tal que $\pi(L') = L$) :

$$t_M(x) = min\{t_N(x,y) \ : \ (x,y) \in L'\}$$

$$s_M(x) = min\{s_N(x,y) \ : \ (x,y) \in L'\}$$

Lo interesante del caso es que el proceso del "guessing" recoge todas las posibles opciones que puedan nacer del uso de máquinas indeterministas :

Teorema 11.2.5. *Sea Σ un alfabeto finito y $L \subseteq \Sigma^*$ un lenguaje. Entonces, L es aceptado por una máquina de Turing indeterminista M en tiempo acotado por f y espacio acotado por g si y solamente si existe un alfabeto $\tau \supseteq \Sigma$ y una máquina de Turing determinstica N sobre τ cuyas funciones de tiempo y espaico están en $O(f)$ y $O(g)$ respectivamente, y tal que si $L(N) \subseteq \tau^*$ es el lenguaje aceptado por N, se tiene :*

$$\forall x \in \Sigma^*, \ x \in L \ \Leftrightarrow \ \exists y \in \tau^*, \ xy \in L(N)$$

Demostración. Hagamos la parte menos fácil. Supongamos L aceptado por M máquina de Turing indeterminista con k cintas de trabajo, sobre el alfabeto Σ y construyamos el alfabeto

$$\tau_1 := Q \times \Sigma^{k+1} \times \{-1,0,1\}^{k+2}$$

$$\tau := \sigma \dot{\bigcup} \tau_1 \dot{\bigcup} \{\sharp, |\}$$

donde $\sharp$ se usar para designar celda vacía cuando convenga y $|$ es un separador que no estáen Σ. La idea de construcción de la máquina N es simple. En cada etapa de cálculo de N verificamos que hemos "guessado" el buen paso de la máquina M. Si no lo hemos hecho bien, rechazamos. Así pues, la máquina N funciona del modo siguiente : Si M tiene k cintas de trabajo y una cinta de input, N tiene $k+3$ cintas de trabajo. Las cintas $WT1$ hasta WT_{k+1} de N van haciendo la computación de M. En la cinta $WT1$

- INPUT : $x \mid y_1, \ldots, y_m$
- **Inicialización :**
  - Verificamos que $y_i \in \tau_1$ para $1 \leq i \leq m$, (es decir, verificamos que son jugadores y que efectivamente, podemos seguir el paso que determinan esos jugadores, en caso contrario ir a un estado de no aceptación, a un ciclo infinito que designaremos por el estado **Error**).
  - Escribimos x en WT_1 Para que comience a trabajar la máquina M.
  - En la cinta WT_{k+2} escribimos y_1.
- **Simulamos** la máquina M sobre $k+1$ cintas de trabajo de N con el siguiente proceso (tipo **while**).
  - **While** la cinta de input de N no alcance la celda vacía (esto es, mientras no se le acabe la lista de jugadores propuestos), realiza la siguiente tarea :

- Usando la información contenida en las cintas WT_1 hasta WT_{k+1} calcula todas las acciones posibles que hubiera podido hacer la máquina M. Escríbelas en la cinta de trabajo WT_{k+3}, separadas por el símbolo |.(Obsérvese que esto sólo depende la lectura concreta que está haciendo la máquina M y que los símbolos son elementos del alfabeto τ_1).
- Después comparamos el contenido de la cinta WT_{k+2} con la lista escrita en WT_{k+3}. Si nunca coinciden, pasamos el estado de ciclo infinito **Error**. Si alguna vez coinciden es ésa y solamente ésa transición es la que se aplica.
- Aplicamos esa única transición (que ha de ser el contenido en la cinta $WT_{k+2} := y_i$. Borramos la cinta WT_{k+3}, buscamos en la cinta de input de N la transición siguiente a la contenida en WT_{k+2}, esto es, el smbolo y_{i+1}. Escribimos en WT_{k+2} el smbolo y_{i+1} y volvemos a realizar el mismo proceso.

- **Parada :** La máquina N para si y solamente si despus de aplicar el proceso a todos los jugadores de la lista $y_1, \ldots, y_m$ (esto es, cuando la cinta de input alcanza una celda vacía) la configuración obtenida es una configuración final aceptadora de M.

- OUTPUT : El output de la máquina M.

De las varias maneras que disponemos de modelizar este proceso, una de ellas es la siguiente. Sea

$$N := (\tau, Q', q_0', F', \delta'),$$

donde

- $$Q' := Q \dot{\bigcup} \mathcal{P}(\tau_1) \dot{\bigcup} (Q \times \{A, B, C\}),$$
  donde $\mathcal{P}(X)$ significa partes de X.
- $q_0' = q_0$ es el estado inicial.
- $F' = F$

La función de transición estará definida como sigue.

$$\delta'(q'; \alpha, w_1, \ldots, w_{k+1}, w_{k+2}) := D,$$

- Si $q' \in Q$, procederemos como sigue :
  $$D := (H, w_1, \ldots, w_{k+1}, w_{k+2}; 0, \ldots, 0),$$
  donde H es el conjunto de todas las posibles acciones de la función de transición de M, esto es,
  $$H := \{z \in \tau_1 \ : \ z = \delta(q', \alpha, w_1, \ldots, w_{k+1})\} \in \mathcal{P}(\tau_1).$$
- si $q' = H \in \mathcal{P}(\tau_1)$, supongamos
  $$w_{k+2} = (q_1; \theta_1, \ldots, \theta_k; \varepsilon_0, \ldots, \varepsilon_k) \in \tau_1$$
  - Si $w_{k+2} \in q' = H$, hacemos :
    $$D := ((q_1, A); \theta_1, \ldots, \theta_k, \lambda; \varepsilon_0, \ldots, \varepsilon_k, +1).$$
  - Si $w_{k+2} \notin q' = H$, hacemos :
    $$D := (\ \textbf{Error}; w_1, \ldots, w_{k+1}, \lambda); 0, \ldots, 0).$$

- Para el resto de los casos, haremos lo siguiente
  - Si $q' := (q, A)$, con $q \in Q$, y $\alpha =,$, avanza un paso la cinta de input :
$$D := ((q, B); w_1, \ldots, w_{k+1}, w_{k+2}; 0, \ldots, 0, +1),$$
  - Si $q' = (q, B)$, con $q \in Q$ $\alpha \neq,$, copia lo que lees en la cinta de input, en la cinta $k+2$, esto es
$$D := ((q, C); w_1, \ldots, w_{k+1}, \alpha; 0, \ldots, 0, 0),$$
  - Si $q' = (q, C)$, con $q \in Q$ y $\alpha \neq,$, pasa al ciclo **Mientras**, esto es,
$$D := (q; w_1, \ldots, w_{k+1}, \alpha; 0, \ldots, 0, 0),$$

El resto de los casos no contemplados, ira a **Error**. □
Este Teorema nos da una interpretación del indeterminismo que siempre se puede transformar en un proceso tipo "guessing". De alguna manera el resultado es comparable al paso de formas de primer orden con cuantificadores a formas del mismo tipo pero en forma prenexa (i.e. los cuantificadores van por delante).

11.3. Mezclando Clases.

Un podra pensar que las clases de complejidad antes definidas no suponen ciertamente una clasificación de los problemas tratados. Ciertamente, existen los llamados Teoremas de Jerarquía que demuestran que tales clases son clases distintas. Esta Subsección estará dedicada a demostrar este hecho. Ambos resultados se remontan a los fundadores de esta modelización (hoy comúnmente aceptada) [HaSt, 65].

11.3.1. Teoremas de Jerarquía.

La definición de clases de complejidad lleva implcita la necesidad de que las clases estn bien definidas, para lo cual es necesario decidir si no son todas las clases iguales. En términos más técnicos, se trata de ver que funciones de complejidad distintas, definen clases de complejidad distintas. A esta idea se la denomina los Teoremas de Jerarquía en Tiempo y en Espacio. Para comenzar hagamos observar que no todas las funciones que acoten tiempo y/o espacio son realmente necesarias, sino esencialmente las llamadas funciones tiempo y/o espacio constructibles.

Definición 11.3.1. Sea $f : \mathbb{N} \longrightarrow \mathbb{N}$ una función monótona.

i) Diremos que f es constructible en tiempo si existe una máquina de Turing determinista M sobre el alfabeto unario $\Sigma := \{1\}$ tal que M para en todos los inputs y tal que existe una constante $c > 0$ verificando que para todo $n \in \{1\}^* = \mathbb{N}$, la máquina M calcula $f(n) \in \{1\}^* = \mathbb{N}$ en tiempo $t_M(n) \leq cf(n)$.

ii) Diremos que f es constructible en espacio si existe una máquina de Turing determinista M sobre el alfabeto unario $\Sigma := \{1\}$ tal que M para en todos los inputs y tal que existe una constante $c > 0$ verificando que para todo $n \in \{1\}^* = \mathbb{N}$, la máquina M calcula $f(n) \in \{1\}^* = \mathbb{N}$ en espacio $s_M(n) \leq cf(n)$.

Observación 11.3.1. A partir de este momento, consideraremos clases de complejidad **DTIME**(t), **NTIME**(t) para funciones $t : \mathbb{N} \longrightarrow \mathbb{N}$ constructibles en tiempo y clases **DSPACE**(s), **NSPACE**(s) para funciones $s : \mathbb{N} \longrightarrow \mathbb{N}$ constructibles en espacio.

Ejemplo 11.3.1. i) Son funciones constructibles en tiempo las siguientes :

- $t(n) := n^k$, para $k \in \mathbb{N}$ fijo,
- $t(n) := n!$,

- $t(n) := 2^{c.n}$, para c fijo,
- $t(n) := 2^{n^k}$, para $k \in \mathbb{N}$ fijo,
- $t(n) := n\lfloor log_2 n \rfloor^k$, para $k \in \mathbb{N}$ fijo,
- $t(n) := 2^{2^{c.n}}$, para $c > 0$ fijo.

ii) Son funciones constructibles en espacio las siguientes :

- Todas las constructibles en tiempo y, por ejemplo,
- $t(n) := \lfloor log_2 n \rfloor^k$, para $k \in \mathbb{N}$ fijo,

En general, uno no trata de analizar todos los problemas recursivamente enumerables cuando de trata de analizar complejidad de los procedimientos y problemas esenciales. Para explicar este fenómeno, obsérvese el siguiiente enunciado :

Proposición 11.3.1. *Sea Σ un alfabeto finito y M una máquina de Turing sobre Σ. Sea $t : \mathbb{N} \longrightarrow \mathbb{N}$ una función constructible en tiempo y supongamos $t_M(n) \leq t(n)$ para cada $n \in \mathbb{N}$. Sea $L \subseteq \Sigma^*$ el lenguaje aceptado por M. Entonces, existe una máquina de Turing N tal que se verifican las propiedades siguientes :*

- $L(N) := \Sigma^*$,
- $t_N(n) \in O(t)$,
- $Res_N(x) := \chi_L$, *donde :*
$$\chi_L : \Sigma^* \to \{0, 1\},$$
es la función característica de L.
- *El sistema de transición de N no repite configuraciones, esto es, para cada $c \in S_N$ no es cierto $c \to_N c$.*

Demostración. El proceso consiste en poner un contador a la máquina M, esto es, usaremos un número de cintas igual al máximo entre el número de cintas que usa la máquina M y el número de cintas que se necesitan para evaluar la función constructible en tiempo t. La máquina N funcionará como sigue :

INPUT : $x \in \Sigma^*$.
Evalúa $t(\mid x \mid)$ y guarda el resultado en alguna cinta de trabajo w_1 en modo unario.
Simula el cálculo de la máquina M (usando cuantas cintas de trabajo adicionales se necesiten) con la siguiente restricción : a cada paso de cálculo, borra un dígito de lo que haya escrito en w_1 y da un paso hacia atrs con esa cinta.
if la máquina M acepta el input x antes de que se agote el contenido de la cinta w_1, devuelve **1**
else devuelve **0**
fi
END

El tiempo de cálculo de la máquina N es $O(t)$ para evaluar la función constructible en tiempo $t(n)$, más tiempo $t(n)$ para simular la máquina M. Esta segunda acotación es obvia puesto que a cada paso de cálculo se borra un dígito en la cinta w_1, con lo que la máquina se detiene cuando se detiene M o cuando se acaba el contenido en w_1, esto es, parta siempre en tiempo acotado en $O(t)$. Además, si $x \in L$, la simulación de la máquina M tiene tiempo $t(n)$ para completar sus cálculos, asíque N aceptar x. En otro caso, rechazar x. En particular, evaluar la función carcaterística de L. Finalmente, no repite ninguna configuración puesto que a cada paso va borrando un dígito en la cinta de trabajo w_1, con lo que no hay dos configuraciones iguales que se sigan una a la otra. □

Observación 11.3.2. Al proceso antes descrito se le denomina poner un contador a una máquina de Turing, el contador es el contenido de la cinta de trabajo donde guardamos el tiempo disponbile $t(n)$. A partir de este momento nos couparemos solamente de las máquinas de Turing que verifican las propiedades anteriores. Podemos modelizarlas como siguen : máquina de Turing $M := (\Sigma, Q, q_0, F_A, F_R, \delta)$, donde

- $L(M) := \Sigma^*$
- $F := F_A \bigcup F_R$ son los estados finales de cálculo, F_A representa los estadops finales aceptadores y F_R los estados finales de rechazo. Los estados finales aceptadores se alcanzan cuando la máquina de la proposición anterior alcanza un **1**, mientras que los estados finales de rechazo se alcanzan cuando la máquina de la proposición anterior alcanza un **0**. Para los primeros diremos que M acepta el input y para los segundos diremos que M rechaza el input.
- El sistema de transición de la máquina M no repite configuraciones.

Definición 11.3.2. Sean $f, g : \mathbb{N} \longrightarrow \mathbb{R}_+$, dos funciones monótonas crecientes. Diremos que $g \in \omega(f)$ si para cada $c > 0$, existe un conjunto infinito $I_c \subseteq \mathbb{N}$, tal que

$$g(n) > cf(n).$$

Una manera distinta de entender la clase $\omega(f)$ es la siguiente :

$$g \in \omega(f), \text{ si y solamente si } g \notin O(f).$$

Teorema 11.3.2 (Teorema de Jerarquía en Espacio, [HaSt, 65])**.** *Sean $s, s' : \mathbb{N} \longrightarrow \mathbb{N}$ dos funciones monótonas crecientes y supongamos que s' es constructible en espacio. Supongamos, además, que $s' \in \omega(s)$ (esto es, $s' \notin O(s)$). Entonces,*

$$\mathbf{DSPACE}(s') \setminus \mathbf{DSPACE}(s) \neq \emptyset,$$

$$\mathbf{NSPACE}(s') \setminus \mathbf{NSPACE}(s) \neq \emptyset.$$

Demostración. Consideremos el alfabeto Σ_0 de la máquina universal y $\{0,1\}$ el alfabeto binario. Sea $\sigma : \Sigma_0 \longrightarrow \{0,1\}^4$ un diccionario. Consideremos el lenguaje siguiente : $L \subseteq \Sigma_0^*$ dado mediante : para cada $x \in \{0,1\}^*$, $x \in L$ si y solamente si se verifican las siguiente propiedades :

- Existe $j \in \mathbb{N}$, tal que $x = 1 \cdots^{(j} \cdots 10w$, donde
- Existe una máquina de Turing determinista M cuyo código es $c_M \in \Sigma_1^*$ y

$$w := \tilde{\sigma}(c_M)$$

- $(c_M, x) \notin HP$ (esto es, $x \notin L(M)$) y, finalmente,
- $s_{\mathcal{U}}(c_M, x) \leq s'(|x|)$, donde $\mathcal{U}$ es la máquina universal descrita en el Capítulo 4 y $S_{\mathcal{U}}$ es la función de espacio de la máquina Universal.

Veamos que este lenguaje L es recursivamente enumerable y que $L \in \text{DSPACE}(s')$. Para lo cual diseamos la siguiente máquina de Turing $\mathcal{M}$:

Input $x \in \Sigma_0^*$
Calcula $n := |x|$.
Dado que s' es espacio constructible, calcula $s'(n)$ y guarda este valor en unario en alguna cinta de trabajo (se llamar Contador o reloj).
Verifica que x tiene la forma $1^j 0w$ donde w es la traducción al alfabeto Σ_0 del código de una máquina de Turing M y 1^j representa una lista de j símbolos del tipo $1 \in \Sigma_0$.

Simula la máquina universal $\mathcal{U}$ (traducida al alfabeto Σ_0 como en la demostración del Halting Problem) sobre el par (w, x). Mantn la simulación siempre que el espacio utlizado no supere el número de celdas "reservadas" por el Contador.
ACEPTAR si y solamente si $\mathcal{U}$ rechaza (c_M, x) usando menos espacio del reservado por el contador.

Es un mero ejercicio el verificar que el lenguaje aceptado por la máquina que acabamos de describir. Además el espacio requerido por esta máquina de Turing no es nunca mayor que el espacio $O(s')$ usado para calcular la función s' (que era espacio constructible) más el espacio requerido para la simulación de la máquina universal $\mathcal{U}$ traducida sobre Σ_0^*. Por lo visto en la Subsección 4.2 (La independencia del alfabeto) el espacio requerido por la traducción es $O(s_{\mathcal{U}}(c_M, x)$. Por lo visto en la Subsección 4.4 (Teorema de la máquina Universal) :

$$s_{\mathcal{U}}(c_M, x) \in O(s_M(x)).$$

Finalmente, por la corrección usando el contador, si la máquina $\mathcal{M}$ que acabamos de describir, cuando simula $\mathcal{U}$ intentase ocupar más espacio que $s'(n)$, entonces, se parará. Luego, la máquina $\mathcal{M}$ que acabamos de definir, trabaja en espacio

$$O(min\{s'(|x|), s_M(|x|)\}) = O(s').$$

En conclusión $L \in \text{DSPACE}(s')$.
Razonemos por reducción al absurdo. Supongamos que $L \in \text{DSPACE}(s)$ y sea M_1 una máquina de Turing que acepta el lenguaje L en espacio $O(s)$. Sea $w := \tilde{\sigma}(c_{M_1}) \in \Sigma_0^*$ la traducción del código de la máquina M_1. Sea $c > 0$ una constante tal que para cada $x \in L = L(M_1)$, se tenga

$$s_{M_1}(|x|) \leq cs(|x|).$$

Sea $c_1 > 0$ una constante tal que :

$$s_{\mathcal{U}}(c_M, x) \leq c_1 s_{M_1}(|x|) \leq (c_1 \cdots c)s(|x|).$$

Finalmente, sea $c_2 > 0$ tal que la máquina obtenida por la traducción de la máquina universal $\mathcal{U}$ al alfabeto binarioa, requiere espacio acotado por

$$c_2 s_{\mathcal{U}}(c_M, x) \leq (c_2 \cdots c_1 \cdots c)s(|x|).$$

Pongamos $K := c_2 c_1 c > 0$. Como $s' \in \omega(s)$, existe una infinidad de números naturales n tales que

$$s'(n) > Ks(n).$$

Sea $m \in \mathbb{N}$ un número natural tal que la palabra

$$x := 1 \cdots^{(m} \cdots 10w$$

tiene talla n para algn n tal que $s'(n) > Ks(n)$. Entonces, la máquina $\mathcal{M}$ tiene suficiente espacio para simular sobre x los cálculos que la máquina universal hace sobre el input (c_{M_1}, x). Ocurre que $x \in L$ si y solamente si $\mathcal{M}$ acepta x, lo cual es equivalente a que $\mathcal{U}$ rechaza (c_{M_1}, x), luego $x \notin L(M_1)$ contradiciendo el hecho de que el lenguaje aceptado por M_1 era L. Concluimos así que $x \notin L$, ergo como la máquina universal tiene suficiente espacio para realizar sus cálculos, esto quiere decir que $\mathcal{U}$ acepta (c_{M_1}, x), luego $x \in L = L(M_1)$. En conclusión, la hiptesis es falsa y $L \neq L(M_1)$.
Hemos hecho un outline de la demostración en el caso determinista. El caso indeterminista es análogo usando máquinas indeterministas. □
El segundo resultado de la jerarquía es el siguiente :

Teorema 11.3.3 (Teorema de Jerarquía en Tiempo, [HaSt, 65]). *Sean $t, t' : \mathbb{N} \longrightarrow \mathbb{N}$ dos funciones monótonas crecientes y supongamos que t' es constructible en tiempo. Supongamos, además, las hiptesis siguientes :*

i) $t' \in \omega(t\log(t))$,

ii) $n \in O(t')$.

Entonces,

$$\mathbf{DTIME}(t') \setminus \mathbf{DTIME}(t) \neq \emptyset,$$
$$\mathbf{NTIME}(t') \setminus \mathbf{NTIME}(t) \neq \emptyset.$$

Demostración. La prueba es esencialmente igual al caso de la complejidad en espacio. Para obtener el resultado enunciado podemos usar la máquina de Turing universal que requiere tiempo $t_{\mathcal{U}}(c_M, x) \leq O\left(t_M(x)\log\left(t(x)\right)\right)$ (cf. Observación 9.6.3 y en el Teorema 9.6.10 de [HeSt, 66] y [HaLeSt, 65]. El lenguaje en $\mathbf{DTIME}(t') \setminus \mathbf{DTIME}(t)$ ser el lenguaje $L \subseteq \Sigma_0^*$ dado mediante : para cada $x \in \Sigma_0^*$, $x \in L$ si y solamente si se verifican las siguiente propiedades :

- Existe $j \in \mathbb{N}$, tal que $x = 1 \cdots^{(j} \cdots 10w$, donde
- Existe una máquina de Turing determinista M cuyo código es $c_M \in \Sigma_1^*$ y
$$w := \tilde{\sigma}(c_M)$$
- $(c_M, x) \notin HP$ (esto es, $x \notin L(M)$) y, finalmente,
- $t_{\mathcal{U}}(c_M, x) \leq t'(|x|)$, donde $\mathcal{U}$ es la máquina universal descrita en el Capítulo 4 y $t_{\mathcal{U}}$ es la función de tiempo de la máquina Universal.

La máquina que realiza este lenguaje, vendrá dada por :

INPUT $x \in \Sigma_0^*$
Calcula $n := |x|$.
Dado que t' es tiempo constructible, calcula $t'(n)$ y guarda este valor en unario en alguna cinta de trabajo (se llamar Contador o reloj).
Verifica que x tiene la forma $1^j 0w$ donde w es la traducción al alfabeto Σ_0 del código de una máquina de Turing M y 1^j representa una lista de j símbolos del tipo $1 \in \Sigma_0$.
Simula la máquina universal $\mathcal{U}$ (traducida al alfabeto Σ_0 como en la demostración del Halting Problem) sobre el par (w, x). Mantén la simulación siempre que el tiempo utlizado no supere el número de pasos "reservados" por el Contador, esto es, en menos de $t'(|x|)$ pasos.
ACEPTAR si y solamente si $\mathcal{U}$ rechaza (c_M, x) usando menos tiempo del reservado por el contador.

El argumento de diagonalización queda como ejercicio (es análogo al Teorema de Jerarquía en Espacio). □

Corollario 11.3.4. *Los siguientes son contenidos estrictos :*

- *En términos de espacio determinista tenemos los siguientes contenidos estrictos, para* $k \geq 2$, $a > b > 2$ *:*
$$\mathbf{LOG} := \mathbf{DSPACE}(logn) \subset \mathbf{DSPACE}(log^k n) \subset \mathbf{DSPACE}(n) \subset \mathbf{DSPACE}(n^k)$$
$$\mathbf{DSPACE}(n^k) \subset \mathbf{DSPACE}(n^{k+1}) \subset \mathbf{DSPACE}(2^n) \subset \mathbf{DSPACE}(b^n) \subset \mathbf{DSPACE}(a^n)$$
$$\mathbf{DSPACE}(a^n) \subset \mathbf{DSPACE}(2^{n^k}) \subset \mathbf{DSPACE}(2^{2^n})$$
- *En términos de espacio indeterminista tenemos los siguientes contenidos estrictos, para* $k \geq 2$, $a > b > 2$ *:*
$$\mathbf{NLOG} := \mathbf{NSPACE}(logn) \subset \mathbf{NSPACE}(log^k n) \subset \mathbf{NSPACE}(n) \subset \mathbf{NSPACE}(n^k)$$
$$\mathbf{NSPACE}(n^k) \subset \mathbf{NSPACE}(n^{k+1}) \subset \mathbf{NSPACE}(2^n) \subset \mathbf{NSPACE}(b^n) \subset \mathbf{NSPACE}(a^n)$$
$$\mathbf{NSPACE}(a^n) \subset \mathbf{NSPACE}(2^{n^k}) \subset \mathbf{NSPACE}(2^{2^n})$$

- *En términos de tiempo determinista tenemos los siguientes contenidos estrictos, para $k \geq 2$, $a > b > 2$:*

$$\mathbf{DTIME}(n) \subset \mathbf{DTIME}(n^k) \subset \mathbf{DTIME}(n^{2k}) \subset \mathbf{DTIME}(2^n)$$

$$\mathbf{DTIME}(2^n) \subset \mathbf{DTIME}(b^n) \subset \mathbf{DTIME}(a^n) \subset \mathbf{DTIME}(2^{n^k}) \subset \mathbf{DTIME}(2^{2^n})$$

- *En términos de tiempo indeterminista tenemos los siguientes contenidos estrictos, para $k \geq 2$, $a > b > 2$:*

$$\mathbf{NTIME}(n) \subset \mathbf{NTIME}(n^k) \subset \mathbf{NTIME}(n^{2k}) \subset \mathbf{NTIME}(2^n)$$

$$\mathbf{NTIME}(2^n) \subset \mathbf{NTIME}(b^n) \subset \mathbf{NTIME}(a^n) \subset \mathbf{NTIME}(2^{n^k}) \subset \mathbf{NTIME}(2^{2^n})$$

11.3.2. Unas palabras sobre grafos orientados.

Para atacar una mezcla de las clases de complejidad, comenzaremos definiendo algunas nociones básicas de la teoría de grafos y un par de algoritmos elementales sobre ellos.

Definición 11.3.3. i) Llamaremos grafo a todo par $G := (V, E)$ donde V es un conjunto finito (llamado conjunto de vértices o de nodos del grafo) y $E \subseteq \mathcal{P}(V)$ (llamado conjunto de ejes o aristas del grafo) de tal manera que para cada $X \in E$, $\geq 1\sharp(X) \leq 2$.

ii) Llamaremos grafo orientado a todo par $G := (V, E)$ donde V es un conjunto finito y $E \subseteq V \times V$.

iii) Llamaremos camino en un grafo orientado $G = (V, E)$ a toda sucesin finita de nodos de G :

$$v_1, \ldots, v_r$$

de tal modo que :

$$\forall i, 1 \leq i \leq r-1, (v_i, v_{i+1}) \in E$$

iv) Llamaremos ciclo en un grafo orientado G a todo camino $v_1, \ldots, v_r$ tal que $v_1 = v_r$. Diremos que un grafo es acclico si no presenta ningn ciclo.

Notación 11.3.1. Hay bastante terminología en torno a la noción de grafo, sólo alguna poca va a ser útil en nuestra discusión; pero prefiero añadir algún que otro término por facilitar la escritura. Para comenzar se llama conexo en un grafo orientado $G = (V, E)$ a todo subconjunto cerrado por caminos, es decir, S es conexo si todo par de nodos x, y de S están conectados por un camino de G cuyos nodos están todos en S. Se llaman componentes conexas de un grafo a los conjuntos maximales con esta propiedad. Finalmente, se llama clausura transitiva de un subconjunto B de un grafo a la unión de todas las componentes conexas del grafo que cortan a B. Es fácil observar que la clausura transitiva de B es justamente el conjuntos de puntos de V que son alcanzables por caminos desde algún punto de B. Se suele decir que un grafo orientado y acíclico es un bosque y que cada componente conexa es un árbol.
Otros dos términos útiles son las nociones de abanico de entrada y abanico de salida (fan–in y fan–out). El fan–in de un vértice en un grafo orientado es el número de aristas o ejes que llegan a él (para un vértice $x \in V$, es el número de pares $(y, x) \in E$). En cambio, el fan–out es el conjunto de aristas que "salen" de él (para un vértice $x \in V$, es el número de pares $(x, y) \in E$). Se suelen llamar binarios a los árboles con fan–out 2 y fan–in 1.
Se denomina longitud (o altura) de un grafo orientado a la longitud del mayor de los caminos que está contenido en él. Se denominan árboles balanceados a todo árbol binario en el que todo camino tiene la misma longitud.

En lo que respecta a nuestras discusiones actuales, trataremos solamente de utilizar un par de algoritmos elementales sobre el cálculo de la clasura transitiva y la decisin de si algo es alcanzable dentro de un grafo orientado.

11.3.2.1. Clausura Transitiva

Dado un grafo orientado $G := (V, E)$ y dado un subconjunto de vértices $B \subseteq V$ hallar la clausura transitiva de B en G. El procedimiento se describe como sigue :

INPUT :

- $G := (V, E)$ donde $V := \{1, \ldots, N\} \subseteq \mathbb{N}$ (codificados en binario), y $E \subseteq V \times V$ (dado como una lista)
- $B \subseteq V$ (dado como una lista).

OUTPUT :

- La lista T de los vrtives en la clausura transitiva de B.

DESCRIPCIÓN DEL PROCEDIMIENTO :

INPUT : $G := (V, E)$, $B \subseteq V$

INITIALIZE :

- $S := B$
- $T := \{v \in V \; \exists u \in B, \; (u, v) \in E\}$

While $S \not\subseteq T$, **do** $S := T$
Para cada $u \in S$ y cada $v \in V \setminus T$ hacer
If $[(u, v) \in E] \wedge [v \notin T]$, **do**

$$T := T \bigcup \{v\}$$

else $T := T$
endif
Next v, **next** u
Return
endwhile
OUTPUT T
end

Proposición 11.3.5. *Existe una máquina de Turing que calcula la clausura transitiva con la estructura de datos anterior en tiempo $O(\sharp(E)\sharp(V)^3)$ y, por tanto, en tiempo polinomial $O(n^5)$ en el tamaño de la entrada.*

Demostración. Basta con modelizar del modo más obvio el algoritmo anterior en máquinas de Turing. □

Un problema menos sofisticado, del tipo decisional, que sigue al anterior, es el siguiente :

11.3.2.2. Alcanzable

Dado un grafo orientado $G := (V, E)$, dado un subconjunto de vértices $B \subseteq V$ y dado $F \subseteq V$, decidir si F corta a la clausura transitiva de B (i.e. si algn vértice de F es alcanzable desde algún vértice de B). El procedimiento se describe como sigue :

Input :

- $G := (V, E)$ donde $V := \{1, \ldots, N\} \subseteq \mathbb{N}$ (codificados en binario), y $E \subseteq V \times V$ (dado como una lista)
- $B \subseteq V$ (dado como una lista de vértices).
- $F \subseteq V$ (dado como una lista de vértices).

Output :

- 1 o 0 según se verifiquen las propiedades buscadas.

Descripción del procedimiento :
Básicamente como el anterior, salvo por el hecho de que la condición de parada se pone solamente cuando el resultado intermedio T corta al conjunto F.

Proposición 11.3.6. *Existe una máquina de Turing determinística tal que dados G un grafo orientado, B y F dos subconjuntos de vértices, decide si la clausura transitiva de B corta al conjunto F o no. El tiempo de ejecución de esta máquina de Turing es del orden $O(n^5)$ donde n es el tamaño de la entrada.*

11.3.3. Una codificación más corta de las configuraciones

Hasta ahora hemos condificado las configuraciones de una máquina de Turing $M := (Q, \Sigma, q_0, F, \delta)$ mediante una lista del tipo:

$$s := (q; \triangleright x_0, \triangleright x_1, \ldots, \triangleright x_k; n_0, \ldots, n_k),$$

donde

- $q \in Q$ es un estado,
- $\triangleright x_i \in \triangleright\Sigma^*$, es una palabra sobre el alfabeto Σ,
- $n_i \in \{1\}^*$ es un número entero que indica la posición de la unidad de control.

La observación trivial es que para cada lista $(q; \triangleright x_0, \triangleright x_1, \ldots, \triangleright x_k;$ puede haber muchas configuraciones que dependen de las distintas posiciones $n_0, \ldots, n_k$. De hecho, con nuestra formulación el número posible es $\prod_{i=0}^{k} (|x_i| + 2)$.
Esto se puede simplificar del modo siguiente. Añadamos un nuevo elemento $\diamondsuit \notin \Sigma \cup \{\triangleright\}$. Definimos una configuración en codificación corta sobre M como una lista

$$(q; \triangleright y_0, \triangleright y_1, \ldots, \triangleright y_k),$$

donde

- $q \in Q$,

- $\triangleright y_i$ está en $\triangleright\Sigma^*$ o en el lenguaje regular

$$\triangleright\Sigma^*\diamondsuit\Sigma^* := \{\triangleright y \ : \ \exists x_1, x_2 \in \Sigma^*, \ \ y := x_1\diamondsuit x_2\}.$$

La interpretación obvia es que la posición n_i de la unidad de control es la celda inmediatamente siguiente a $\diamondsuit$, es decir, si $\triangleright y$, está en una cinta de una máquina de Turing, la posición de la unidad de control n_i es dada por:

$$n_i := \begin{cases} 0, & \text{si } y \in \Sigma^* \\ |x_1| + 1 & \text{si } y = x_1\diamondsuit x_2 \end{cases}$$

Obviamente, dada una configuración en codificación corta, en tiempo lineal en la talla de la configuración de obtiene la codificación usada en la máquina de Turing Universal. Adicionalmente, se tiene

Proposición 11.3.7. *Sea $M = (Q, \Sigma, q_0, F, \delta)$ una máquina de Turing, $x \in \Sigma^*$ y $f : \mathbb{N} \longrightarrow \mathbb{N}$ una función monótona creciente. El número de configuraciones de M de talla acotada por $f(|x|)$ y que tienen a x en la cinta de input está acotado por*

$$2^{Kf(|x|)+d},$$

donde K es una constante que depende solamente del número de cintas de M, del cardinal de Q y del cardinal de Σ.

Demostración. La idea es que, con la codificación corta, el número de posibles configuraciones será, a lo sumo,

$$\sharp(Q) \times \left((\sharp\left(\Sigma \cup \{\triangleright, \diamondsuit\}\right))^{f(|x|)}\right)^k.$$

El resto es poner las constantes, tomando

$$K := k\log_2(\sharp(\Sigma) + 2), \ \ d := \log_2 \sharp(Q).$$

□

11.3.4. Espacio indeterminista frente a tiempo determinista.

Con estos elementos tan simples de la teoría de grafos ya estamos en condiciones de ofrecer un primer resultado razonable.

Teorema 11.3.8. *Sea $t \ : \mathbb{N} \longrightarrow \mathbb{N}$ una función constructible en tiempo, $t(n) \geq n$. Entonces,*

$$NSPACE(t) \subseteq DTIME(2^{O(t)})$$

Demostración. Supongamos dado $L \subseteq \Sigma^*$ un lenguaje aceptado por una máquina de Turing indeterminista en espacio acotado por t. Sea $M := (\Sigma, Q, q_0, F.\delta)$ la tal máquina de Turing, sea k el número de cintas de trabajo y sea $(S_M, \rightarrow_M)$ el sistema de transición asociado. Ahora, sea dado $x \in \Sigma^*$ y definamos la proyección

$$\pi \ : S_M \longrightarrow \overline{S_M},$$

siendo π la proyección que olvida la coordenada que expresa el contenido de la cinta de input (i.e. la x) trabajaremos en el conjunto

$$\{x\} \times \overline{S_M}$$

Sea V_x el conjunto de configuraciones en $\overline{S_M}$ cuyo espacio ocupado está acotado por t.Por la Proposición precedente, existe una constante $c > 0$ tal que el cardinal de V_x es a lo sumo

$$2^{ct(n)}$$

donde $n := |x|$.
Claramente, codificar un elemento de V_x se puede hacer en talla (juntando separadores y $\{0,1\}$ eventualmente a Σ)

$$O(log_2\sharp(Q) \times kt(n) \times log_2 n \times klog_2 t(n))$$

Fijada la máquina estos códigos no ocupan espacio mayor que :

$$O(t(n)log_2 t(n)log_2 n) \leq O(t(n)log_2^2 t(n))$$

A partir de este punto, vamos a definir una estructura de grafo sobre V_x de la manera obvia. Definamos $F_x \subseteq V_x$ como aquellos objetos en los cuales el estado que aparece es un estado final aceptador.
Dados $s_1, s_2 \in V_x$, diremos que $(s_1, s_2) \in E_x$ si, dentro del sistema de transición de la máquina M, ocurre que

$$(x, s_1) \rightarrow_M (x, s_2)$$

A partir de aquí el procedimiento funciona como sigue.

INPUT : $x \in \Sigma^*$
Eval $|x| = n$
Eval $t(n)$
Write down all configurations in V_x
Write down E_x
Apply Reachable to $(V_x, E_x), \pi(I(x)), F_x$, donde $I(x)$ es la configuración inicial en x

---

Corollario 11.3.9. *En las condiciones anteriores*

$$DSPACE(t) \subseteq NSPACE(t) \subseteq DTIME(2^{O(t)})$$

11.3.5. Tiempo indeterminista frente a espacio determinista.

Proposición 11.3.10. *Si $t : \mathbb{N} \longrightarrow \mathbb{N}$ es un función tiempo constructible, se tiene :*

$$NTIME(t) \subseteq DSPACE(t)$$

Demostración. Es análoga a la máquina en la que demostrbamos que el guessing era equivalente al indeterminismo. Lo único que hay que hacer es escribir a priori todos los posibles pasos de instrucción (t(n) pasos a lo sumo) y ejecutar el cálculo mientras se respeten esos pasos de instrucción. Si acaso, observar que una utilización brutal del grafo anterior nos dará, de modo casi inmediato, una cota del tipo

$$NTIME(t) \subseteq DSPACE(t^2)$$

Lo que evitamos es guardar toda configuración y nos conformamos solamente con guardar toda instrucción en un cierto camino.

□

11.4. El Teorema de Savitch.

Este resultado es uno de los importantes en el desarrollo de la Complejidad Estructural, puesto que permite clarificar que el indeterminismo es irrelevante cuando se trata de analizar el espacio de cálculo (i.e. la memoria). Como el curso trata esencialmente de entender los problemas de complejidad relativos al tiempo y al indeterminismo, dejaremos la prueba de este resultado a uno de los trabajos. En todo caso, se puede encontrar el varios de los textos recomendados (como [Papa, 94] o [BaDiGa, 88]). Digamos simplemente que la esencia del resultado consiste en un proceso de búsqueda en grafos usando muy pocos recursos de memoria o, si se prefiere, juagando un pebble game sobre el grafo de todas las configuraciones.

Teorema 11.4.1. *[Sa, 70] Si $s : \mathbb{N} \longrightarrow \mathbb{N}$ es una función constructible en espacio y $s(n) \geq log_2 n$, se tiene :*

$$\mathbf{NSPACE}(s) \subseteq \mathbf{DSPACE}(s^2)$$

Demostración. En realidad, este es un pequeño bosquejo de la prueba basado en la función Alcanzable. Daremos la máquina de Turing , mediante esta función y dejaremos al trabajo correspondiente, la disquisición sobre cómo generar una máquina de Turing que haga el proceso. Sea $M := (\Sigma, Q, q_o, F, \delta)$ una máquina de Turing indeterminsita que evalúa la función característica de un lenguaje $L \in \mathbf{NSPACE}(s)$. Para cada $x \in \Sigma^*$, consideremos el grafo (V_x, E_x) donde V_x es el conjunto de todas las configuraciones en S_M conteniendo x en la cinta de input y con talla menor que $s(|x|)$. Por su parte, E_x estáformado por los pares $(c, c') \in V_x \times V_x$ de configuraciones, tales que $c \to_M c'$ en $(S_M, \to_M)$.
Ahora definiremos la función Alcanzable para un grafo cualquiera $\mathcal{G} := (V, E)$.

Input $(A, B) \in V \times V$, $r \in \mathbb{N}$
Alcanzable (A,B,r) es dado por :

Si $r \leq 1$ Alcanzable($\mathcal{G}$,A,B,1) $= 1$ si y solamente si existe $C \in V$ tal que $(A, C) \in E$ y $(C, B) \in E$ (es decir, si puedo pasar en el grafo $\mathcal{G}$ de A a B con un camino de dos pasos)

Si $r \geq 2$ Alcanzable($\mathcal{G}$,A,B,r) $= 1$ si y y solamente si existe $C \in V$ tal que Alcanzable($\mathcal{G}$,A,C, r-1) $= 1$ y Alcanzable($\mathcal{G}$,C,B,r-1) $= 1$ (es decir, si puedo pasar de A a B en 2^r pasos dentro del grafo).

Así definamos la máquina N del modo siguiente :

Input $x \in \Sigma^*$
Evaluar $s(|x|) \in \mathbb{N}$
Para cada configuración final aceptadora $B_x \in V_x$, aplicar :

$$\text{Alcanzable}(\mathcal{G}_x, I(x), B_x, s(|x|))$$

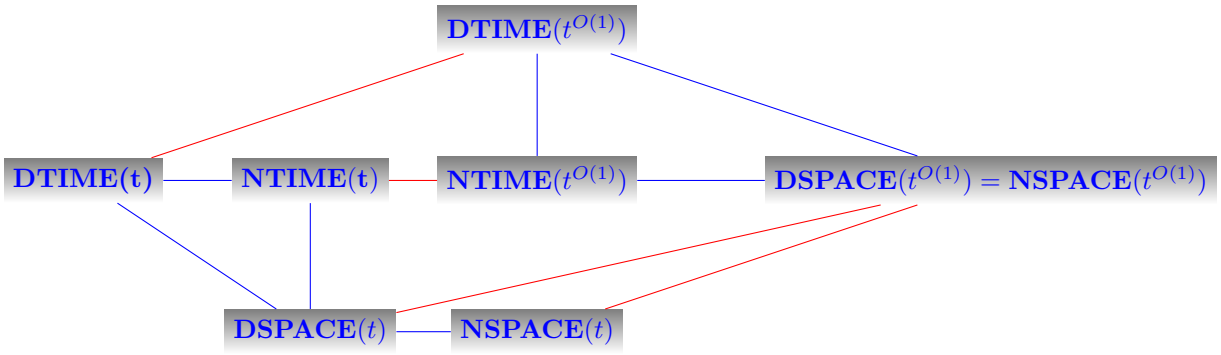
Aceptar si y solamente si Alcanzable produce el valor **1**.

La idea a definir es cómo transformar la definición recursiva de Alcanzable como proceso iterativo, lo que se transforma en recorrer el grafo $\mathcal{G}_x := (V_x, E_x)$ pasando por todos los caminos posibles (de longitud mxima $2^{O(s(|x|))}$ sin escribir completamente los caminos, sino solamente $s(n)$ nodos. Este proceso queda como ejercicio.

□

11.5. Un pequeño grafo con las relaciones conocidas.

Un pequeño grafo entre las clases ya definidas. El color rojo de las aristas denota contenido estricto, mientras que el color azul denota contenido del que se desconoce si hay igualdad o no.



Nótese que los únicos contenidos estrictos hacen referencia a los Teoremas de Jerarquía. No hemos incluido algunas de las otras relaciones por no embrollar más el dibujo.

Capítulo 12

Clases Centrales de Complejidad.

Contents

12.1. Introducción 239
12.1.1. Tesis de Cobham-Edmonds: a partir de unos experimentos teóricos . . 239
12.1.2. Clases Centrales de Complejidad. 241
12.2. La clase NP. 241
12.2.1. Ejemplos naturales de problemas indeterminísticos. 243
12.2.2. Ejemplo: Primalidad y el Teorema de Pratt. 244
12.2.3. El Teorema de Pratt :PRIMES $\in$ **NP** 244
12.2.4. Máquinas con Oráculos 250
12.3. El Cálculo Proposicional y su Cálculo de Predicados 251
12.3.0.1. El Cálculo Proposicional : Fórmulas Booleanas. 251
12.4. NP−completitud : Teoremas de Cook y Karp. 258
12.4.1. Reducciones 259
12.4.1.1. Problemas de Búsqueda (Search Problem). 259
12.4.1.2. Clausura bajo reducciones 260
12.4.2. El Teorema de Cook: Problemas **NP**–completos. 261
12.4.3. El Teorema de Ladner 268
12.5. La clase PSPACE 269
12.5.1. Problemas **PSPACE**-completos 269
12.5.2. La Jerarquía Polinomial **PH** 271
12.6. Un grafo final 272

12.1. Introducción

A Pesar de las disquisiciones anteriormente expuestas, pocas son las clases de complejidad verdaderamente interesantes para la informática práctica. Las clases esenciales son las clases relacionadas (o, en el entorno) de la **Tesis de Cobham-Edmonds**(cf. [Co, 65], [Ed, 65a, Ed, 65b]) sobre los problemas *Tratables* informáticamente.

12.1.1. Tesis de Cobham-Edmonds: a partir de unos experimentos teóricos

Para hacer la discusión, tomemos como ejemplo las funciones de tiempo siguientes

$$f_1(n) := n^7, f_2(n) := 2^n, f_3(n) := 2^{n^2}, f_4(n) := 2^{2^n},$$

que corresponden a cotas de tiempo polinomial (f_1), exponencial (f_2), expo-polinomial (f_3) y doblemente exponencial (f_4). Consideremos, asimismo, el más moderno y potente ordenador del mundo en la actualidad.[1] : se trata del superordenador DTF de IBM y la NSF. Sobre este ordenador, ejecutamos algoritmos cuyas funciones de tiempo vienen acotadas, respectivamente, por f_1, f_2, f_3 y f_4.
Veamos la evolución de estos algoritmos sobre el DTF :

i) $n = 10$, es decir, nos dan un input con 10 dígitos. El DTF tardará 2^{800} TRILLONES DE AÑOS EN ACABAR LA EJECUCIÓN DEL ALGORITMO 4. Los algoritmos $1, 2$ y 3 serán concluidos en tiempos respectivos :

   - ALGORITMO 1 : menos de 1 milésima de segundo,
   - ALGORITMO 2 : menos de 1 diez–milésima de segundo,
   - ALGORITMO 3 : aproximadamente 1 mes,

ii) $n = 30$, es decir nos dan un input con 30 dígitos. No sabría contar el tiempo para el algoritmo 4 que, ciertamente ya queda descartado. También el algoritmo 3 : DTF tardaría unos 2^{700} TRILLONES DE AÑOS EN RESPONDER. Para los otros dos algoritmos nos queda :

   - ALGORITMO 1 : menos de 1 centésima de segundo,
   - ALGORITMO 2 : menos de 1 centésima de segundo,

iii) $n = 100$, es decir nos dan un input con 100 dígitos. Los algoritmos 1 y 2 aún son comparables :

   - ALGORITMO 1 : menos de 1 décima de segundo,
   - ALGORITMO 2 : aproximadamente 1 mes para terminar sus cálculos,

iv) $n = 124$, es decir nos dan un input con 124 dígitos. Los algoritmos 1 y 2 ya no son comparables :

   - ALGORITMO 1 : menos de 1 décima de segundo,
   - ALGORITMO 2 : aproximadamente 1 MILLÓN DE AÑOS,

Del análisis anterior se deduce que los problemas cuyas funciones de tiempo crecen por encima de la función exponencial (2^n) no van a servir para problemas naturales. Tener 150 o 200 bits (dígitos) de input no es una cantidad excesiva en ningún problema aplicado. Por lo tanto, *se define la clase de problemas tratables como la clase de problemas cuya función de tiempo es polinomial en el tamaño de la entrada.*

Definición 12.1.1 (**Tesis de Cobham-Edmons).** Se llaman algoritmos y problemas tratables a los algoritmos de la clase **P** definida del modo siguiente :

$$\mathbf{P} := \bigcup_{k \in \mathbb{N}} \mathbf{DTIME}(n^k).$$

Es decir son tratables los algoritmos cuyo tiempo de ejecución sea polinomial en el tamaño de la entrada. Todos los demás se consideran intratables.

[1] La NSF ha financiado con 53 millones de dólares el proyecto DTF (Distributed Teracomputer Facility), gestionado por IBM. Su entrada en funcionamiento está prevista para finales del año 2001. El DTF sea unas mil veces más potente que el ordenador *Deep Blue* que derrotó a Kasparov en 1997. Se estima que su velocidad de cálculo sea de unos $14{,}10^6$ Mips, esto es, unas $2^4 10^{12}$ operaciones bit por segundo.

12.1.2. Clases Centrales de Complejidad.

i) Clases Determinadas por el Tiempo: Se definen las clases deterministas e indeterministas (prefijo **N**). El Teorema de Jerarquía en Tiempo (Teorema 11.3.3) nos garantiza que tenemos contenidos estrictos en cada una de las columnas. La clase **NP** es la única de la que aún se pueden albergar esperanzas (muy pequeñas, a estas alturas) de que fuera "tratable", porque se desconoce su exacta relación con **P**:

Deterministas	*Indeterministas*
P	$\mathbf{NP} := \bigcup_{k\in\mathbb{N}} \mathbf{NTIME}(n^k)$
$\mathbf{EXTIME} := \mathbf{DTIME}(2^{O(n)})$	$\mathbf{NEXTIME} := \mathbf{NTIME}(2^{O(n)})$.
$\mathbf{EXPTIME} := \bigcup_{k\in\mathbb{N}} \mathbf{DTIME}(2^{n^k})$	$\mathbf{NEXPTIME} := \bigcup_{k\in\mathbb{N}} \mathbf{NTIME}(2^{n^k})$
$\mathbf{DDEXTIME} := \mathbf{DTIME}(2^{2^{O(n)}})$	

ii) Clases de Funciones/Correspondencias: Se denotan añadiendo el sufijo **F**: **PF**, **NPF**, **EXTIMEF**, etc.

iii) Clases Determinadas por el Espacio. Exceptuando el caso de **NLOG**, las clases de espacio indeterminista no se consideran, como consecuencia del Teorema de Savitch (cf. Teorema 11.4.1). El Teorema de Jerarquía en Espacio (Teorema 11.3.2) nos garantiza que tenemos contenidos estrictos en cada una de las columnas y el contenido estricto $\mathbf{NLOG} \subsetneq \mathbf{PLOG}$. La clase **PSPACE** es la única de la que aún se pueden albergar esperanzas (muy pequeñas, a estas alturas) de que fuera "tratable", porque se desconoce su exacta relación con **P**:

Deterministas	*Indeterministas*
$\mathbf{LOG} := \mathbf{DSPACE}(logn)$	$\mathbf{NLOG} := \mathbf{NSPACE}(logn)$
$\mathbf{PLOG} := \bigcup_{k\in\mathbb{N}} \mathbf{DSPACE}(log^k n)$	
$\mathbf{PSPACE} := \bigcup_{k\in\mathbb{N}} \mathbf{DSPACE}(n^k)$	
$\mathbf{EXSPACE} := \mathbf{DSPACE}(2^{O(n)})$	

12.2. La clase NP.

Es quizá la clase más relevante de la informática teórica pues en ella han ido apareciendo una serie de problemas naturales, sencillos y esenciales, que, por desgracia, no ha sido posible resolver con pocos recursos de tiempo y/o espacio. Fue introducida por S. Cook ([Cook, 71]), aunque el formalismo es debido a Richard Karp ([Krp, 72]).
Recordamos las nociones de indeterminismo y guessing" introducidas en la Sección anterior. El concepto de "guessing" se expresa para la clase **NP** de la manera siguiente:

Teorema 12.2.1. *Sea Σ un alfabeto finito y $L \subseteq \Sigma^*$ un lenguaje. Entonces, $L \in \mathbf{NP}$ si y solamente si existen :*

i) Dos funciones polinomiales $p, q : \mathbb{N} \longrightarrow \mathbb{N}$

ii) Un alfabeto finito $\tau \supseteq \Sigma$,

iii) Una máquina de Turing determinística M sobre el alfabeto τ,

iv) Un lenguaje $\tilde{L} \subseteq \tau^*$,

tales que :

i) El lenguaje aceptado por M es $\tilde{L}$ (i.e. $\tilde{L} = L(M)$),

ii) El tiempo de ejecución de la máquina M está acotado por p, i.e.

$$t_M(n) \leq p(n), \forall n \in \mathbb{N}$$

iii) El lenguaje L está caracterizado por la propiedad siguiente :

$$\forall x \in \Sigma^*, \ \ |x| = n \in \mathbb{N}, \ \ \Big[[x \in L] \Leftrightarrow \Big[\exists y \in \tau^*, \ \ \Big[[|y| \leq q(n)] \wedge \Big[xy \in \tilde{L}\Big]\Big]\Big]\Big]$$

Demostración. Obvio por lo discutido en torno al guessing. Insistimos en la idea de que el guessing es solamente la proyección, con recursos de longitud y tiempo de ejecución acotados. □

Observación 12.2.1. La anterior Proposición debe entenderse del modo siguiente. Los problemas en la clase **NP** son aquellos problemas en los que *"si alguien me sugiere un candidato a solución a mi problema"* (i.e. el GUESSING o CERTIFICADO) yo puedo *"testar en tiempo polinomial, esto es, con un algoritmo tratable, si el candidato que me sugieren es o no solución de mi problema.* Lo que no sé, lo que no sabe nadie como veremos, es si esta propiedad es suficiente para decir que un problema con tales propiedades es tratable. Y esta es la dificultad. Esto es, se sabe que $\mathbf{P} \subseteq \mathbf{NP}$ pero nadie sabe decidir si este contenido es estricto. Se sospecha que, tras casi treinta años de esfuerzos de la comunidad de informática teórica sin avances, la respuesta a esta pregunta debe ser que ambas clases no son iguales. Por eso, a la pregunta sobre la relación entre **P** y **NP** , que se suele llamar la **Conjetura de Cook**, se le da la forma :

$$\mathbf{P} \neq \mathbf{NP} \ ?.$$

Observación 12.2.2. Una manera alternativa de presentar esta idea, debida a [GaJo, 79] se resume en la siguiente anécdota :
Supongamos que usted trabaja para una empresa y se le pide desarrollar un programa que resuelva un problema P. Le pueden ocurrir dos cosas inicialmente :

- *Usted encuentra rápidamente un algoritmo que funciona en la práctica y resuelve el problema P.*
- *Usted no encuentra tal algoritmo y, después de mucho reflexionar, encuentra un argumento que le permite decir a su empresa que tal algoritmo no existe.*

En ambos casos su empleo estará asegurado. Pero puede ocurrir, y de hecho ocurre muy a menudo, que ninguna de estas dos opciones se da. Ocurre, y mucho más a menudo de lo que se piensa fuera del contexto informático, que usted es incapaz de encontrar un algoritmo que funcione en la práctica y que es también incapaz de demostrar que tal algoritmo no existe. En tal caso, debe buscar alguna justificación para salvar su empleo. La justificación propuesta por Cook y Karp se expresa en los términos siguientes :
– Verá, jefe, no puedo encontrar lo que me pidió y no sé demostrar que tal cosa no existe; pero puedo demostrarle que no encontrará a nadie en el mundo que le realice la tarea que me pidió.
La idea se expresa también en los términos siguientes : de los resultados conocidos sabemos que los problemas tratables en la práctica **P** están en la clase **NP**. Sin embargo, nadie sabe si estas dos clases coinciden. A esta pregunta se la conoce como

$$\textit{Conjetura de Cook } \mathbf{P} \neq \mathbf{NP} \ ?.$$

Su resolución está valorada en un millón de dólares, pagable por el Instituto Clay que dirige A. Wiles. Aunque nadie sabe resolverla, sí se conocen algunos estudios relevantes.

En ocasiones es esencial discutir la presencia de clases de complejidsad definidas por complementarios.

Definición 12.2.1. Sea $\mathcal{C}$ una clase de complejidad. La clase $co-\mathcal{C}$ se define como la clase de los complementarios de los lenguajes que están en $\mathcal{C}$, esto es

$$co-\mathcal{C} := \{L \subseteq \Sigma^* \ : \ \Sigma^* \setminus L \in \mathcal{C}\}.$$

Obviamente, en clases determiníticas en tiempo esta noción carece de sentido. Esto es,

$$co-\mathbf{P} = \mathbf{P}, \quad co-\mathbf{EXTIME} = \mathbf{EXTIME}, \ldots$$

Donde toma sentido esta nociónn es el las clases inteterminísticas. Por ejemplo, nadie conoce las relaciones que puedan existir entreb las clases :

$$co-\mathbf{NP}?\mathbf{NP}, \quad co-\mathbf{NEXTIME}?\mathbf{NEXTIME}, \ldots$$

12.2.1. Ejemplos naturales de problemas indeterminísticos.

Algunos ejemplos ya han sido presentados en la Sección 12.2: como **PRIMES** (cf, Problema 11.2.1, con los lenguajes PRIMES y COMP, **Knapsack Problem, KnP** (cf. Problema 11.2), **Traveling Salesman Problem, TSP** (cf. Problema 11.2) o los lenguajes *COMPOL* y *PRIMPOL* (cf. Ejemplo 11.2.2).

Proposición 12.2.2. *Los siguientes son problemas en la clase* **NP***:*

i) $COMP \in \mathbf{NP}$, $PRIMES \in \text{co} - \mathbf{NP}$

ii) $\mathbf{KnP} \in \mathbf{NP}$,

iii) $\mathbf{TSP} \in \mathbf{NP}$,

iv) $COMPOL \in \mathbf{NP}$, $PRIMEPOL \in \text{co} - \mathbf{NP}$

Demostración. Es un sencillo ejercicio para los alumnos. □

Observación 12.2.3. Obsérvese que sólo decimos $PRIMES \in \text{co} - \mathbf{NP}$ o $PRIMEPOL \in \text{co} - \mathbf{NP}$. Para poder demostrar que están en **NP** hay que trabajar un poco más. Para el problema dado por el lenguaje **PRIMES** V.R. Pratt [2] encontró en 1975 un algoritmo indeterminista basado en el Teorema pequeño de Fermat. Lo veremos más adelante.

Otros ejemplos pueden ser:

Problema (**Optimización Entera (Integer Programming, IP)).** Dados $A \in \mathcal{M}_{n\times m}(\mathbb{Z})$ una matriz con coeficientes enteros y dado $b \in \mathbb{Z}^m$ un vector con coordenadas enteras, decidir si

$$A \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \leq b,$$

tiene solución en $\mathbb{Z}^n$.

Un interesante ejercicio sería probar el siguiente resultado:

Teorema 12.2.3. $\mathbf{IP} \in \mathbf{NP}$

Nótese que la dificultad estriba en probar que si un problema de Optimización Lineal Entera tiene solución, entonces posee solución entera con talla polinomial en la talla de la entrada. Es decir, probar que existe certificado de talla apropiada.

Ejemplo 12.2.1 (**Hilber Nullstellensatz, HN).** Los siguientes problemas se conocen como Hilbert Nullstellensatz o Problema de Consistencia de Sistemas de Ecuaciones Polinomiales Multivariadas (véase [Pa, 95] y las referencias que allí se citan). Veamos una versión global de este Nullstellensatz:

Sea K un cuerpo y supongamos que nos dan un número finito de polinomios multivariados $f_1, \ldots, f_s \in K[X_1, \ldots, X_n]$ de grado a lo sumo 3. Sea L un cuerpo más grande que K (i.e. $L \supset K$). Se define el conjunto algebraico

$$V_L(f_1, \ldots, f_s) := \{x \in L^n \ : \ f_i(x) = 0,\ 1 \leq i \leq s\}.$$

Se pide :

Problema (**Satistactibilidad en una extensión del cuerpo de coeficientes).** Decidir si $V_L(f_1, \ldots, f_s)$ es vacío o no.

[2] V.R. Pratt. "Every Prime has a succinct certificate". *SIAM J. on Comput.* **4** (1975) 214–220.

En otras palabras, se pide decidir si el sistema de ecuaciones polinomiales en varias variables

$$\begin{cases} f_1(X_1,\ldots,X_n) &= 0 \\ \vdots & = 0 \\ f_s(X_1,\ldots,X_n) &= 0 \end{cases}$$

posee o no solución en L^n.
Algunos casos notables son :

i) En el caso $K=\mathbb{Q}$ y $L=\mathbb{Z}$. Se trata del problema X de Hilbert y por lo dicho en el Capítulo 1 y en la Sección 1.6, el Teorema de J. Matjasievicz nos dice que no hay ningún algoritmo que resuelva este problema.

ii) $K=\mathbb{Z}/p\mathbb{Z}=L$. Se trata del problema SAT de S. Cook expuesto anteriormente.

iii) $K=\mathbb{Q}$ y $L=\mathbb{C}$ se llama propiamente Teoremas de los Ceros de Hilbert (Hilbert Nullstellensatz)

iv) $K=\mathbb{Q}$ y $L=\mathbb{R}$ se llama Teorema de los Ceros Real (o Real Nullstellensatz).

Trataremos este Problema en la Parte IV de este manuscrito.

Una lista con más de 400 ejemplos naturales de problemas, provenientes de ámbitos muy diversos, con la cualidad de que esos problemas son resolubles fácilmente de modo indeterminista es la Guía de lo Intratable de [GaJo, 79].

12.2.2. Ejemplo: Primalidad y el Teorema de Pratt.

Vamos a ver un ejemplo de las dificultades que pueden existir con la caracterización como lenguaje **NP** de un problema sencillo y usual : los tests de primalidad de números enteros :

Problema 12.2.1. Dado $n\in\mathbb{N}$, en código binario, decidir si n es un número primo, i.e. evaluar la función caracterstica del siguiente lenguaje :

$$\mathbb{P} := \{n\in\{0,1\}^* \ : \ n\in\mathbb{N} \quad \text{es un número primo}\}$$

Los métodos más clásicos al respecto pueden comenzar con el siguiente :

12.2.2.0.1. Criba de Eratóstenes (s. III a. de C.) : Dado $n\in\mathbb{N}$, probar con todos los números menores que $\sqrt{n}$ si dividen o no a n. Si no hallamos ninguno, concluir que es primo.

Proposición 12.2.4. *El tiempo de ejecución de la Criba de Eratóstenes sobre un input $n\in\mathbb{N}$ es del orden $O(n^{1/2}log_2^3 n)$ y, por tanto, es un algoritmo exponencial en la talla de la entrada (i.e.* **PRIMES** $\in$ **EXTIME**$)$

12.2.3. El Teorema de Pratt :PRIMES $\in$ NP

Analicemos un poco cuidadosamente qué ocurre con la condición de ser primo.
Para ello introduzcamos un poco más de material

Definición 12.2.2. Definiremos la función de Euler $\varphi \ : \mathbb{N}\longrightarrow\mathbb{N}$ del modo siguiente :

$$\varphi(n) := \sharp\{m\in\mathbb{N} \ : \ m\leq n,\ gcd(m,n)=1\}$$

Tenemos las siguiente propiedades elementales de la función de Euler :

Lema 12.2.5. *En las anteriores notaciones, sea $n\in\mathbb{N}$ y sea $\mathbb{Z}/n\mathbb{Z}$ el anillo de restos módulo n. Sea $(\mathbb{Z}/n\mathbb{Z})^*$ el grupo de las unidades de $\mathbb{Z}/n\mathbb{Z}$ para la operación producto. Se tiene :*

i) $\varphi(n)\leq n$,

ii)

$$\varphi(n) = \sharp(\mathbb{Z}/n\mathbb{Z})^*$$

Demostración. Mero ejercicio sin mayor inters. □

Lema 12.2.6. *Para cada número natural $n \in \mathbb{N}$, $n \geq 2$, las siguientes propiedades son equivalentes :*

i) $n \in \mathbb{P}$,

ii) $\mathbb{Z}/n\mathbb{Z}$ *es un dominio de integridad.*

iii) $\mathbb{Z}/n\mathbb{Z}$ *es un cuerpo*

iv) $\varphi(n) := n - 1$

v) $(\mathbb{Z}/n\mathbb{Z})^*$ *con la operación producto es un grupo abeliano de orden* $n - 1$.

Demostración. Obvio de nuevo por las propias definiciones. □

Lema 12.2.7. *Sea $p \in \mathbb{N}$ un número primo y $n \in \mathbb{N}$ un número natural, entonces*

$$\varphi(p^n) = p^n - p^{n-1}.$$

Demostración. Consideremos los conjuntos siguientes :

$$A := \{m \in \mathbb{N} \ : \ 1 \leq m \leq p^n - 1\},$$

y

$$B := \{m \in \mathbb{N} \ : \ 1 \leq m \leq p^n - 1, p \mid m\}.$$

Se tiene

$$\varphi(p^n) = \sharp(A) - \sharp(B) = (p^n - 1) - \sharp(B).$$

Ahora bien, es fácil comprobar que

$$B = \{pk \ : \ 1 \leq k \leq p^{n-1} - 1\},$$

con lo que $\sharp(B) = p^{n-1} - 1$ y la igualdad se sigue. □

Lema 12.2.8. *Sean $n, m \in \mathbb{N}$, $n, m \geq 2$ dos números naturales coprimos (i.e. $gcd(m, n) = 1$). Entonces, se tiene :*

$$(\mathbb{Z}/m\mathbb{Z})^* \times (\mathbb{Z}/n\mathbb{Z})^* \cong (\mathbb{Z}/mn\mathbb{Z})^*$$

y el isomorfismo, como grupos abelianos, viene justamente del Teorema Chino de los Restos. En particular, para cada par de números naturales $n, m \in \mathbb{N}$ coprimos, se tiene

$$\varphi(nm) = \varphi(n)\varphi(m)$$

y la función de Euler es multiplicativa

Demostración. Verificar a través del Teorema Chino de los Restos. El Teorema Chino de los restos induce un isomorfismo de los grupos de unidades :

$$(\mathbb{Z}/mn\mathbb{Z}) \longrightarrow (\mathbb{Z}/m\mathbb{Z}) \times (\mathbb{Z}/n\mathbb{Z})$$

$$x + mn\mathbb{Z} \longmapsto (x + m\mathbb{Z}, x + n\mathbb{Z})$$

□

En otros términos, la función de Euler es una función multiplicativa.

Corollario 12.2.9 (Teorema Pequeño de Fermat)**.** *Sea $n \in \mathbb{N}$ un número natural, $n \geq 2$. Entonces, para cada $x \in \mathbb{Z}/n\mathbb{Z} \setminus \{0\}$, se tiene :*

$$x^{n-1} \equiv 1 \ \ mod\ n$$

Demostración. Justamente, como el orden del grupo de las unidades $(\mathbb{Z}/n\mathbb{Z})^*$ es $n-1$, se tiene la afirmación buscada.
Esta propiedad no es suficiente para caracterizar la condición de número primo. Desgraciadamente hay más números que los números primos verificando esta propiedad. Son los llamados *números de Carmichael*. El más pequeño número de Carmichael conocido es

$$561 := 3{,}11{,}17$$

que, obviamente no es un número primo. Además se conoce, desde hace poco tiempo, de la existencia de infinitos números de Carmichael. De hecho, si $C(n)$ es el cardinal del conjunto de número de Carmichael menores que n, se tiene que (cf. [**?**])

$$C(n) \sim n^{0,1}$$

Para poder afinar en la caracterización de los números primos, necistamos hacer un esfuerzo adicional.

Teorema 12.2.10 ([Pr, 75], Versión Fuerte del Teorema Pequeño de Fermat)**.** *Un número natural $n \in \mathbb{N}$, $n \geq 2$ es un número primo si y solamente si una cualquiera de las siguienets condiciones son equivalentes :*

i) $(\mathbb{Z}/n\mathbb{Z})^*$ *es un grupo cclio de orden* $n-1$,

ii) La ecuación $X^{n-1} - 1$ *posee una raíz primitiva en el anillo* $\mathbb{Z}/n\mathbb{Z}$.

Para realizar la demostración introduzcamos un poco más de notación y algunos resultados ms.

Proposición 12.2.11. *Con la definición*

$$\varphi(1) = 1$$

Se tiene :

$$\sum_{m|n} \varphi(m) = n$$

Demostración. Por inducción en N.
Si $n = 1$ es claro.
Para $n \geq 2$, si n es primo,

$$\{m \in \mathbb{N};: , m \geq 1, m \mid n\} = \{1, n\}$$

Luego

$$\sum_{m|n} \varphi(m) = \varphi(1) + \varphi(n) = 1 + (n-1) = n$$

Si $n = p^b$ es una potencia de primo, se tiene:

$$\sum_{m|n} \varphi(m) = \varphi(1) + \varphi(p) + \varphi(p^2) + \cdots + \varphi(p^b) = 1 + (p-1) + (p^2 - p) + \cdots + (p^b - p^{b-1}) = p^b,$$

y la afirmación se sigue en el caso de potencia de primo. Supongamos que n no es ni primo ni potencia de primo. Entonces, existen $a, b \in \mathbb{N}$, $a, b \geq 2$, tales que :

$$a.b = n, \qquad gcd(a, b) = 1$$

Se tendrá que, para todo divisor $m \mid n$, si definimos $m_1 := gcd(m, a)$ y $m_2 = gcd(m, b)$, entonces $m = m_1.m_s$ (justamente por ser a y b coprimos).
Por hipótesis inductiva, tenemos

$$a = \sum_{m_1|a} \varphi(m_1), \qquad b = \sum_{m_2|b} \varphi(m_2)$$

Luego,

$$n = ab = \sum_{m_1|a,\, m_2|b} \varphi(m_1)\varphi(m_2) = \sum_{m_1|a,\, m_2|b} \varphi(m_1.m_2)$$

Porque si $m_1 \mid a$ y $m_2 \mid b$ entonces, $gcd(m_1, m_2) = 1$.
Para concluir la prueba, basta con notar que :

$$\{m \in \mathbb{N} \ : \ m \mid n, m \geq 1\} = \{m_1 m_2 \ : \ m_1 \mid a, m_2 \mid b\}$$

Un contenido es claro pues el segundo conjunto esta contenido en el primero. De otro lado, si $m \mid n$, se concluye que :

$$m_1 := gcd(m, a), \quad m_2 := gcd(m, b)$$

han de verificar

$$m = m_1.m_2$$

□

Para $n \in \mathbb{N}$ fijo definamos los siguientes conjuntos :

$$K(d) := \{x \in \mathbb{Z}/n\mathbb{Z} \ : \ x^d - 1 = 0 \ mod \ n\}$$

y los conjuntos de cardinales :

$$R(d) := \sharp(K(d) \setminus \bigcup_{i=1}^{d-1} K(i))$$

Tenemos las siguientes propiedades :

Lema 12.2.12. *Con las notaciones anteriores, si n es un número primo, se tienen las propiedades siguientes :*

i) $R(d) = 0$ *si* $d \not| \, n - 1$,

ii) $R(d) \leq \varphi(d)$,

iii) $\sum_{d=1}^{n-1} R(d) = n - 1$

Demostración. i) Si $d \not| \, n - 1$,

$$x^d - 1 \neq 0$$

La razón es, obviamente, que en $(\mathbb{Z}/n\mathbb{Z})^*$, todos los elementos han de tener orden divisor de $n - 1$. Pero d no divide $n - 1$, ergo....
ii) Tenemos la siguiente situación : si $x \in K(d)$ se tiene :

$$x^d - 1 = 0 \ mod \ n$$

Si, además,

$$x \notin \bigcup_{j=1}^{d-1} K(j)$$

Entonces, x es una raíz primitiva $d-$ésima de la unidad, es decir,

$$T^d - 1 = \prod_{i=0}^{d-1}(T - x^i)$$

es una factorización en el dominio de ideales principales

$$\mathbb{Z}/n\mathbb{Z}[T]$$

En particular, si $R(d) \geq 1$, las raíces primitivas $d-$ésimas de la unidad estarán en la clase

$$\{x^s \ : \ (gcd(s,d) = 1\}$$

Esto se ve fácilmente, notando que si $gcd(s,d) = 1$, x^s es primitiva. El recíproco es idéntico. En particular, estarán en biyección con

$$(\mathbb{Z}/d\mathbb{Z})^*$$

Luego,

$$R(d) \leq \varphi(d)$$

iii) Todos los elementos de $(\mathbb{Z}/n\mathbb{Z})^*$ tienen que tener algún orden. Este orden es un divisor de $n-1$ y son primitivos $d-$ésimos con respecto a ese orden. luego están en algún $K(d) \setminus \cup_{j=1}^{d-1} K(j)$, con lo que cada uno aporta para algún $R(d)$.

□

Teorema 12.2.13 (Versión Fuerte del Teorema Pequeño de Fermat)**.** *Sea $n \in \mathbb{N}$ un número natural, $n \geq 2$. Entonces, n es un número primo si y solamente si se verifica la siguiente propiedad :*

$$R(n-1) \geq 1$$

es decir, si existe una raíz primitiva $(n-1)-$ésima de la unidad en $(\mathbb{Z}/n\mathbb{Z})^*$.

Demostración. Una de las implicaciones es obvia. Si $R(n-1) \geq 1$ es claro que $(\mathbb{Z}/n\mathbb{Z})^*$ tiene $n-1$ elementos :

$$\{1, x, x^2, \ldots, x^{n-2}\}$$

Veamos el recíproco. Tenemos

$$n - 1 = \sum_d R(d) = \sum_{d|n-1} R(d) \leq \sum_{d|n-1} \varphi(d) = n - 1$$

En particular, como

$$R(d) \leq \varphi(d)$$

para cada d, tendremos

$$R(n-1) = \varphi(n-1) \geq 1$$

□

Corollario 12.2.14. *Un número natural $n \in \mathbb{N}$ impar, verifica $n \in PRIMES$ si y solamente si existe $x \in (\mathbb{Z}/n\mathbb{Z})^*$ tal que:*

i) $x^{n-1} - 1 = 0$ en $\mathbb{Z}/n\mathbb{Z}$ y

ii) Para todo factor irreducible p de $n-1$, se tiene

$$x^{\frac{n-1}{p}} - 1 \neq 0, \text{en } \mathbb{Z}/n\mathbb{Z}.$$

Corollario 12.2.15 ([Pr, 75])**.**

$$\textbf{PRIMES} \in \textbf{NP}$$

Demostración. La prueba se basa en el siguiente algoritmo recursivo.

INPUT $n \in \{0,1\}^* = \mathbb{N}$, impar.
guess indeterministically $p_1, \ldots, p_s \in \{0,1\}^*$ tales que

$$\sum_{i=1}^{s} log_2 p_i < log_2 n.$$

guess indeterministically $x \in /\mathbb{Z}/n\mathbb{Z})^*$.
if

- $\prod_{i=1}^{s} p_i = (n-1)$,
- $p_i \in PRIMES$, para $i = 1, \ldots, s$,
- $x^{n-1} - 1 = 0$ en $\mathbb{Z}/n\mathbb{Z}$, y
- $$x^{\frac{n-1}{p_i}} - 1 \neq 0, \text{en } \mathbb{Z}/n\mathbb{Z}.$$

then OUPUT PRIMO
fi
end

El algoritmo es indeterminista y el lenguaje aceptado es primo. Para ver que es polinomial, obsérvese que el tiempo de ejecución de esta máquina verifica la siguiente propiedad:

$$T_M(n) \leq \sum_{i=1}^{s} T_M(p_i) + c log_2^3 n,$$

con la condición

$$\sum_{i=1}^{s} log_2 p_i = log_2(n-1) < log_2 n.$$

Un sencillo argumento inductivo demostrará que tal función ha de verificar,

$$T_M(n) \in c\,(log_2 n)^4\,.$$

El argumento inductivo será el siguiente:
Demostrar por inducción en s que se verifica:
Dados $a_1, \ldots, a_s \in \mathbb{R}$ números reales positivos, $a_i \geq 1$, $s \geq 2$, se tiene

$$\sum_{i=1}^{s} a_i^4 + \left(\sum_{i=1}^{s} a_i\right)^3 \leq \left(\sum_{i=1}^{s} a_i\right)^4.$$

La demostración se sigue de la mera aplicación (inducción) de esta desigualdad, sabiendo que $n-1$ es par (i.e. $s > 1$). □

12.2.4. Máquinas con Oráculos

Definición 12.2.3. Una máquina de Turing con oráculo $L \subseteq \Sigma^*$ es una máquina de Turing (determinística o no) que posee:

- una cinta especial en la que puede escribir llamada la cinta del oráculo,
- tres estados especiales $\{\text{QUERY}, q_{\text{yes}}, q_{\text{no}}\} \subseteq Q$

cuyo funcionamiento es el siguiente:
En cualquier momento del cálculo, si la máquina accede al estado QUERY, la máquina lee (en un sólo paso) el contenido $\omega \in \Sigma^*$de la cinta del oráculo y devuelve o bien q_{yes} o q_{no} según $\omega \in L$ o no. Depués sigue su computación.

Para una clase de complejidad $\mathbf{C}$ y un lenguaje L, denotaremos por $\mathbf{C}^L$ la clase formada por todos los lenguajes acotados por el recurso descrito por $\mathbf{C}$, pero admitiendo máquinas con oráculo L. Así podemos definir las clases $\mathbf{P}^L, \mathbf{NP}^L$, etc...
Obviamente, si $L \in \mathbf{P}$, se tiene $\mathbf{P}^L = \mathbf{P}$ y así para cada clase de complejidad. La presencia de oráculos aumenta el poder computacional de una clase, aunque uno debe ser cuidadoso con su presencia.
Una de las primeras observaciones que se hicieron en torno al problema de Cook era la dificultad de utilizar argumentos basados en diagonalización (a la Gödel, Turing o como en los Teoremas de Jerarquía anterior): los argumentos basados en diagonalización tienen que ser «especiales» en la medida de que no son aplicables a máquinas de Turing con oráculos. Es el caso del siguiente resultado:

Teorema 12.2.16 ([BaGiSo, 75])**.** *Existe un lenguaje A tal que*

$$\mathbf{P}^A = \mathbf{NP}^A = \mathbf{EXPTIME}^A.$$

Y también existe un lenguaje B tal que

$$\mathbf{P}^B \neq \mathbf{NP}^B.$$

Demostración. Como resumen de la prueba, un lenguaje A que satisface el enunciado es el siguiente:

$$A := \{(M, x, n) \ : \ x \in L(M),\ T_M(x) \leq 2^n\}.$$

Para un lenguaje cualquiera B, definamos $U_B := \{1^n \ : \ \exists x \in B, |x| = n\}$. Claramente $U_B \in \mathbf{NP}^B$, pero *se puede construir un lenguaje B tal que $U_B \notin \mathbf{P}^B$.*
El lenguaje se define del modo siguiente (diagonalización): Para cada $i \in \{0,1\}^*$, sea M_i la máquina de Turing con oráculo B cuyo código es precisamente i. Definimos B inductivamente en función de i. En cada paso añadimos un número finito de elementos nuevos (o no añadimos ninguno).
Supongamos que ya hemos definido algunos elementos B_{i-1} de B en pasos anteriores. Ahora elijamos n mayor que la longitud de todos los elementos de B_{i-1} y ejecutamos la máquina M_i sobre 1^n. Consideramos todas las palabras que se guardan en la cinta del oráculo y alcanzan el estado QUERY. Si alguna está en B_{i-1} procedemos con q_{yes}, si alguna no ha sido predeterminada (no está en B_{i-1}), respondemos q_{no} y continuamos. Estamos ejecutando $M_i^{B_{i-1}}$, de hecho. Detendremos la computación tras $2^n/10$ pasos.
Si la máquina $M_i^{B_{i-1}}$ termina su computación aceptando antes de realizar los $2^n/10$ pasos, escribiremos $B_i = B_{i-1}$ y, en particular, $1^n \notin U_B$. En caso contrario, elijamos una palabra $x \in \{0,1\}^*$, de longitud n, que no ha aparecido en la cinta del oráculo (existen porque el tiempo está acotado por $2^n/10$ y no hemos podido pasar por todos los $x \in \{0,1\}^n$) y definimos $B_i := B_{i-1} \cup \{x\}$.
Definimos finalmente $B := \cup_{i \in \mathbb{N}} B_i$. Con esta construcción garantizamos que la máquina M_i siempre devolverá una respuesta incorrecta sobre 1^n en menos de $2^n/10$ pasos. Por tanto, $U_B \notin \mathbf{P}^B$. □

12.3. El Cálculo Proposicional y su Cálculo de Predicados

12.3.0.1. El Cálculo Proposicional : Fórmulas Booleanas.

Lo que sigue pretende ser una disquisición que presenta las Fórmulas Booleanas y los problemas de SATIFACIBILIDAD y TEOREMA.
Los elementos que definen una teoría formal son esencialmente los siguientes :

- SINTAXIS : Se trata , como en la gramtica de los lenguajes naturales, de fijar la forma en que están escritas las frases, palabras o fórmulas aceptables para esta teoría formal en cuestin. Sus elementos son fundamentalmente :
  - *Alfabeto :* Es un conjunto finito de símbolos sobre los que escribiremos palabras, cada palabra es potencialmente una fórmula de la teoría formal.
  - *Reglas sintácticas :* Son las reglas que permiten definir la clase de fromulas bien escritas para la teoría en discusin. A esta clase de fórmulas se la denomina clase de las *fórmulas bien formadas* o *fbf*.
- DEDUCCIÓN : Pretende reflejar el proceso matemtico de la demostración. El camino que se ha de recorrer para escribir formalmente el paso

  $$\textit{Hipótesis} \rightarrow \textit{Tesis}$$

  Consta de dos elementos fundamentalmente :
  - *Axiomas :* Son las propiedades que se dan por válidas, las hipótesis en suma, sobre las que sigue la arguimentación.
  - *Reglas Deductivas :* Son las reglas que permiten pasar de unas fbf's a otras y, por tanto, describen el proceso seguido en la demostración.
- SEMÁNTICA : Son las asignaciones de valores concretos a las fbf's : las *interpretaciones*. A través de ellas podremos tratar de entender la verdad o falsedad de ciertas fórmulas cuando se especializan a interpretaciones concretas.

Para explicar un poco más el proceso imaginemos un libro y su lectura. Las reglas sintácticas son como las reglas gramaticales en las que se funda el lenguaje que vamos a utilizar para escribir nuestro libro. Las reglas deductivas son los procesos que nos permiten escribir el texto (pongamos por caso el funcionamiento de la máquina de escribir con la permanente verificación de la correción de las frases). Finalmente, el libro es depositado en la estantería. Ahora viene el lector indeterminado que lo abre. Mientras estuvo cerrado no hubo interpretación, sólo un libro dispuesto a tener lector. Ahora que se abre y se lee, el lector puede interpretar lo que allí está escrito. Para ello necesita disponer de los registros que transforman los símbolos inscritos en el libro en significantes subjetivos. Este proceso último es el proceso de la semántica y, obviamente, depende de la interpretación que se haga de los signos el que vayamos a aceptar lo que el libro dice o no (del mismo modo que podemos o no aceptar la lectura de un texto de tipo religioso, según nuestra subjetividad).
Veamos estos objetos a través de una Teoría Formal sencilla.

12.3.0.1.1. La Sintaxis.

- SINTAXIS : Dividida en los siguientes elementos :
  - *Alfabeto :*

    $$\Sigma := \{X_n \ : \ n \in \mathbb{N}\} \bigcup \{\Rightarrow, \neg, \} \bigcup \{1, 0\} \bigcup \{(,)\}$$

    A los elementos $\{X_n \ : \ n \in \mathbb{N}\}$ se les denomina variables de la teoría, a los elementos $\{1, 0\}$ se les denomina constantes de la teoría, y a los elementos $\{\Rightarrow, \neg, \}$ se les denomina conectivas o, simplemente, funcionales de la teoría. Los símbolos en el conjunto $\{(,)\}$ son meros símbolos auxiliares.

- *Reglas sintácticas :* A su vez, estará compuestas de dos elementos por su defínción recursiva : los términos y las reglas de construcción. Estas últimas requieren del uso de metavariables para ser definidas, pero esto no es grave por lo que respecta a esta teoría :
  - *Términos :* Son términos los elementos :

$$1, 0, X_n \ : \ n \in \mathbb{N}$$

  - *Fórmulas bien formadas :* Son fórmulas bien formadas aquellas en Σ^* que pertenecen al menor subconjunto conteniendo a los términos y tal que si A, B son fórmulas bien formadas, también lo sean :

$$(\neg A), (A \Rightarrow B)$$

- DEDUCCIÓN :Consta de los siguiente elementos :
  - *Axiomas :* Sean A, B, C tres fórmulas bien formadas de esta teoría. Son axiomas los siguientes :
    - $(A \Rightarrow (B \Rightarrow A))$
    - $(((A \Rightarrow (B \Rightarrow)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$
    - $(((\neg B) \Rightarrow (\neg A)) \Rightarrow (((\neg B) \Rightarrow A) \Rightarrow B))$
  - *Reglas Deductivas :* Esencialmente una sola Regla Deductiva :
    - *Modus Ponens :* Si A, B son fórmulas bien formadas, se tiene :

$$A, (A \Rightarrow B) \rightarrow B$$

(es decir, si A y $(A \Rightarrow B)$, entonces B)

Usualmente se suele denominar *fórmula booleana* a toda fórmula bien formada de esta teoría del Cálculo Proposicional. Normalmente escribiremos

$$\Phi(X_0, \ldots, X_n)$$

para denotar la fórmula booleana Φ en la cual aparecen solamente variables contenidas en el conjunto $\{X_0, \ldots, X_n\}$.
Con estos elementos ya podemos comenzar a discernir algunos elementos propios del análisis de la Lógica.

Definición 12.3.1. Dada una teoría formal T, llamaremos demostración a toda sucesión de fórmulas bien formadas :

$$s_1, \ldots, s_K$$

tal que para cada $i, 1 \leq i \leq K$, se tiene :

- O bien s_i es un axioma de la teoría,
- o bien existen una regla deductiva R de la teoría que depende de t variables y existen $s_{i_1}, \ldots, s_{i_t}$ con $i_j < i$ tales que :

$$R(s_{i_1}, \ldots, s_{i_t}) \rightarrow s_i$$

Definición 12.3.2. Dada una teoría formal T llamaremos teorema de T a toda fórmula bien formada Φ tal que existe una demostración en T :

$$s_1, \ldots, s_K$$

verificándose $\Phi := s_K$.

La clase de los teoremas es una clase muy especial pues refleja lo que uno puede alcanzar con los recursos deductivos. En ocasiones, una teoría formal puede contener autnticos desastres para nuestra intuición lógica. Por eso se discuten casos como el siguiente :

Definición 12.3.3. Una teoría formal T se denomina consistente si T contiene $\neg$ entre sus símbolos y no es posible demostrar Φ y $(\neg\Phi)$ a partir de los axiomas y con las reglas deductivas de T. En caso contrario diremos que la teoría es inconsistente.

Teorema 12.3.1. *El Cálculo Proposicional define una teoría consistente.*

Demostración. No la incluyo aquí, pero puede verse en el texto [RED, 89], con mecanismos muy elementales. □

Vamos a añadir algunos conceptos del lenguaje del Cálculo Proposicional que serán de utilidad más adelante. En primer lugar, introduzcamos algunas abreviaturas. Son representaciones de fórmulas bien formadas que se describen añadiendo algunas conectivas al alfabeto inicial. Las más comunes son las siguientes :

- *Disyunción :* Dadas dos fórmulas booleanas A, B escribiremos $(A \vee B)$ en lugar de $((\neg A) \Rightarrow B)$
- *Conjunción :* Dadas dos fórmulas booleanas A, B escribiremos $(A \wedge B)$ en lugar de
$$(\neg(A \Rightarrow (\neg B)))$$
- *Equivalencia :* Dadas dos fórmulas booleanas A, B escribiremos $(A \Leftrightarrow B)$ en lugar de
$$((A \Rightarrow B) \wedge (B \Rightarrow A))$$
- *Diferencia Simétrica :* Dadas dos fórmulas booleanas A, B escribiremos $(A \oplus B)$ en lugar de
$$((A \wedge (\neg B)) \vee (B \wedge (\neg A)))$$

Podemos aceptar que nuestras fórmulas booleanas contienen el conjunto completo de conectivas :

$$\mathcal{C} := \{\Rightarrow, \neg, \vee\wedge, \Leftrightarrow, \oplus\}$$

También podemos observar que este conjunto de conectivas posee subconjuntos suficientemente potentes para poder reconstruir todas las demás como abreviaturas. A tales conjuntos se les denomina *conjuntos suficientes de conectivas.*

Algunos conjuntos suficientes de conectivas son los siguientes :

- $\{\Rightarrow, \neg\}$
- $\{\vee, \wedge, \neg\}$
- $\{\wedge, \oplus, \neg\}$

En lo que sigue usaremos como conjunto suficiente de conectivas $\{\Rightarrow, \neg\}$, entendiendo que las demás son abreviaturas.

En segundo lugar, no escribiremos siempre los paréntesis, al menos cuando sean entendidos y no ofrezcan confusión. Este es un convenio matemático para simplificar la escritura, pero debemos seguir manteniendo la idea de que, en la escritura correcta de los objetos deben aparecen tantos paréntesis como sea necesario. Así, por ejemplo, escribiremos :

$$(A_1 \vee A_2 \vee \cdots \vee A_m)$$

para representar

$$(\ldots((A_1 \vee A_2) \vee A_3) \vee \cdots \vee A_m)$$

Y lo mismo haremos con la conjunción $\wedge$.

Definición 12.3.4. i) Una fórmula booleana Φ se denomina clausula disyuntiva (o simplemente clausula) si tiene la forma siguiente :

$$\Phi = A_1 \vee A_2 \vee \cdots \vee A_m$$

donde

$$A_i \in \{0,1\} \bigcup \{X_n \ : \ n \in \mathbb{N}\} \bigcup \{(\neg X_n) \ : \ n \in \mathbb{N}\}$$

ii) Una fórmula booleana Φ se denomina clausula conjuntiva si tiene la forma siguiente :

$$\Phi = A_1 \wedge A_2 \wedge \cdots \wedge A_m$$

donde

$$A_i \in \{0,1\} \bigcup \{X_n \ : \ n \in \mathbb{N}\} \bigcup \{(\neg X_n) \ : \ n \in \mathbb{N}\}$$

Definición 12.3.5. i) Una fórmula booleana Φ se denomina en forma normal conjuntiva si tiene la forma :

$$\Phi = A_1 \wedge A_2 \wedge \cdots \wedge A_m$$

donde A_i es una clausula disyuntiva.

ii) Una fórmula booleana Φ se denomina en forma normal disyuntiva si tiene la forma siguiente :

$$\Phi = A_1 \vee A_2 \vee \cdots \vee A_m$$

donde A_i es una clausula conjuntiva.

12.3.0.1.2. Semántica. Consta de los elementos interpretación y sustitución.

Definición 12.3.6. Una interpretación de las fórmulas bien formadas de una teoría formal T está compuesta de los siguientes elementos :

i) Un conjunto no vacío D, llamado dominio.

ii) Unos elementos del dominio D asignados a las constantes de la teoría formal.

iii) Unas funciones definidas sobre el dominio que están asignadas a los símbolos de función de la teoría formal.

iv) Unas relaciones sobre el dominio que están asignadas a los símbolos de relación de la teoría formal.

Normalmente las relaciones llevan acompañadas unas asignaciones de verdad o falsedad correspondientes a la función característica de la relación. En nuestro caso, por ahora, el Cálculo Proposicional no presenta símbolos de relación, por lo que no nos vamos a preocupar de la discusión de este aspecto.

La interpretación booleana del Cálculo Proposicional.

Consideramos el dominio $D := \{V, F\}$. Además, consideramos la asignación de funciones y constantes siguientes :

$$1 \longmapsto V$$

$$0 \longmapsto F$$

Y las funciones :

$$imp \ : D^2 \longrightarrow D$$

dada mediante la siguiente tabla de valores :

x	y	imp(x,y)
V	V	V
V	F	F
F	V	V
F	F	F

Adems, consideraremos la siguiente función :

$$neg \ : D \longrightarrow D$$

dada mediante la siguiente table de valores :

x	neg(x)
V	F
F	V

Mediante estas dos funciones podemos proceder a una interpretación de los símbolos del alfabeto como funcionales :

$$\Rightarrow \longmapsto \text{ imp}$$

$$\neg \longmapsto \text{neg}$$

Obviamente la interpretación nos definirá los parámetros a partir de los cuales podemos pasar a definir el valor que toma cada fórmula booleana en términos de unos datos concretos del dominio.

Definición 12.3.7. Dada una teoría formal con variables $\{X_n \ : \ n \in \mathbb{N}\}$, y una interpretación de esa teoría con dominio D, llamaremos sustitución a toda sucesión $\varepsilon \in D^{\mathbb{N}}$ (es decir, $\varepsilon := (\varepsilon_n \ : \ n \in \mathbb{N})$, donde $\varepsilon_n \in D$).

A partir de una interpretación, un dominio D y una sustitución ε, podemos proceder a definir la siguiente transformación de fórmulas booleanas en valores :

$$\tilde{\varepsilon} \ : \ \{ \text{ fmulas booleanas } \} \ \longrightarrow D$$

donde $\varepsilon := (\varepsilon_n \ : \ n \in \mathbb{N}) \in D^{\mathbb{N}}$
Sobre los términos :

- $\tilde{\varepsilon}(0) = F, \tilde{\varepsilon}(1) = V$,
- $\tilde{\varepsilon}(X_n) := \varepsilon_n$,

Sobre las fórmulas booleanas : Mediante el siguiente procedimiento recursivo, que aprovecha de la definición recursiva de la clase de fórmulas booleanas :
Si A y B son fórmulas booleanas, definiremos :

$$\tilde{\varepsilon}((A \Rightarrow B)) := imp(\tilde{\varepsilon}(A), \tilde{\varepsilon}(B))$$

$$\tilde{\varepsilon}((\neg A)) := neg(\tilde{\varepsilon}(A))$$

Fácilmente podemos concluir las siguientes afirmaciones :

Proposición 12.3.2. *La relación $\tilde{\varepsilon}$ es una aplicación.*

Demostración. Gracias a la definición recursiva usada en la construcción de la clase de fórmulas booleanas. □

Definición 12.3.8. i) Dada una teoría formal y una interpretación con dominio D, diremos que una fórmula Φ de la teoría es satisfacible sobre D si existe una sustitución $\varepsilon \in D^{\mathbb{N}}$ tal que $\tilde{\varepsilon}(\Phi) = V$.

ii) Dada una teoría formal y una interpretación con dominio D, diremos que una fórmula Φ es una tautología sobre D si para cualquier sustitución $\varepsilon \in D^{\mathbb{N}}$, se tiene :

$$\tilde{\varepsilon}(\Phi) = V$$

Definición 12.3.9. Un interpretación de una teoría se denomina modelo de la teoría si todo teorema de la teoría es tautológico sobre la interpretación.

En el caso actual se tienen las siguientes propiedades :

Proposición 12.3.3. *Dada la teoría formal del Cálculo Proposicional antes introducida, y la interpretación D antes definida, se tiene :*

i) Todo axioma de la teoría es tautológico sobre el dominio D,

ii) La regla deductiva Modus Ponens es tautológica en el sentido siguiente : Para toda interpretación $\varepsilon \in D^{\mathbb{N}}$ y cualesquiera dos fórmulas booleanas A, B, si $\tilde{\varepsilon}(A) = V$ y $\tilde{\varepsilon}((A \Rightarrow B)) = V$, entonces también $\tilde{\varepsilon}(B) = V$.

iii) Todo teorema del Cálculo Propsicional es una tautología sobre el anterior dominio y con la anterior interpretación, es decir lo anterior es un modelo de la teoría.

Demostración. Mero ejercicio de comprobación. □

Hay una implicación relativamente más fácil de demostrar que relaciona teoremas y tautologías del modo siguiente :

Definición 12.3.10. Una teoría formal se denomina completa si existe un modelo y en ese modelo toda tautología es demostrable.

Teorema 12.3.4. *El Cálculo Proposicional es una teoría formal completa.*

Demostración. La prueba de este hecho es un poco más liosa y no aporta nada sustancialmente esencial al contenido del curso. Por ello prefiero no incluirla y remitir al lector al bonito libro [RED, 89] (o a cualquier manual básico de teoría de modelos). □

Hay un formalismo notacional de relativa conveniencia en la interpretación del proceso que acabamos de definir mediante sustituciones. Hemos dicho que denotamos mediante $\Phi(X_0, \ldots, X_n)$ las fórmulas booleanas en las que solamente aparecen variables en el conjunto $\{X_0, \ldots, X_n\}$. De la misma manera observamos el hecho siguiente :

Proposición 12.3.5. *Sea $\Phi(X_0, \ldots, X_n)$ una fórmula booleana que solamente depende de variables en el conjunto $\{X_0, \ldots, X_n\}$. sean $\varepsilon, \tau \in D^{\mathbb{N}}$ dos sustituciones tales que :*

$$\varepsilon_i = \tau_i, \quad para \;\; 0 \leq i \leq n$$

Entonces,

$$\tilde{\varepsilon}(\Phi) = \tilde{\tau}(\Phi)$$

Además, basta con conocer los valores $(\varepsilon_0, \ldots, \varepsilon_n) \in D^n$ para conocer el valor $\tilde{\varepsilon}(\Phi)$. A esta cantidad la denotaremos mediante :

$$\Phi(\varepsilon_0, \ldots, \varepsilon_n) := \tilde{\varepsilon}(\Phi)$$

Demostración. Por inducción en el número de veces que se han utilizado $\Rightarrow$ y $\neg$ en la construcción de Φ, usando la definición de las reglas de sustitución, se observa obviamente la dependencia exclusiva de las variables que son usadas en la descripción de Φ. □

Corollario 12.3.6. *Sea dada una fórmula booleana $\Phi(X_0, \ldots, X_n)$. Tenemos las siguientes equivalencias :*

i) La fórmula Φ es satisfacible si y solamente si

$$\exists (\varepsilon_0, \ldots, \varepsilon_n) \in D^n, \; \Phi(\varepsilon_0, \ldots, \varepsilon_n) = V$$

ii) La fórmula Φ es una tautología si y solamente si

$$\forall(\varepsilon_0,\ldots,\varepsilon_n)\in D^n,\ \Phi(\varepsilon_0,\ldots,\varepsilon_n)=V$$

Demostración. Obvio □

La conclusión fundamental de este Corolario hace referencia al posible análisis de propiedades como ser satisfacible o ser tautológica mediante procedimientos algorítmicos y conecta de modo fuerte con otro elemento esencial de la Teoría de Modelos.

Definición 12.3.11. Una teoría formal se dice decidible si existe un algoritmo que decide cuando una fórmula de la teoría es o no teorema. Es decir si existe una máquina de Turing M sobre un alfabeto finito que contiene al alfabeto de la teoría y tal que se para sobre todas las fórmulas bien formadas evaluando la función característica del lenguaje de teoremas de la teoría. Más formalmente sea Σ el alfabeto de M (que contiene al alfabeto de la teoría) :

$$FBF := \{\text{ fórmulas bien formadas }\}\subseteq\Sigma^*$$

$$TEOR\subseteq FBF\subseteq\Sigma^*$$

el lenguaje de las fórmulas bien formadas que son demostrables en la teoría. Entonces, $L(M):=FBF$ y la función que evala M es

$$Res_M := \aleph_{TEOR}\ : L(M)\subseteq\Sigma^*\longrightarrow\{0,1\}$$

siendo $\aleph_{TEOR}$ la función característica de la clase $TEOR$.

Una segunda manera de definir las teorías decidibles es diciendo que es una teoría en la que las fórmulas bien formadas son un conjunto recursivo y la clase de fórmulas bien formadas que son teoremas es también un conjunto recursivo. Obsérvese que la noción de decidible depende solamente de las reglas sintcticas y de las reglas de deducción y los axiomas, pero no depende a priori de la semántica. Sin embargo, en el caso del Cálculo Propocicional, la buena relación entre sintaxis y semántica permite una demostración de la decidibilidad de esta teoría mediante el uso de las propiedades semánticas. El Corolario anterior es esencial en este sentido puesto que garantiza que para ver si una fórmula booleana es tautológica no necesitamos compobarlo con todas las posibles sutituciones, sino solamente con valores en un conjunto finito D^{n+1}, donde $n+1$ es una cota para las variables que aparecen en la fórmula dada.
El Corolario anterior se puede usar del modo siguiente :

Teorema 12.3.7. *El Cálculo Proposicional es una teoría decidible.*

Demostración. Para decidir si una fórmula booelana es o no un teorema, basta con deicidir si esa fórmula booleana es una tautología. Ahora bien, los alumnos disponen ya de un algoritmo (aunque no siempre bien formulado) con el cual son capaces de reflejar el proceso siguiente :
INPUT : $\Phi(X_0,\ldots,X_n)$, $(\varepsilon_0,\ldots,\varepsilon_n)\in\{V,F\}^n$
OUTPUT : $\Phi(\varepsilon_0,\ldots,\varepsilon_n)$
Bastar pues con ejecutar ese algoritmo sobre todos los valores del conjunto $\{V,F\}^n$ para poder decidir si Φ es o no teorema (es teorema si y solamente si es tautología, si y solamente si el valor que sale para cualquier $(\varepsilon_0,\ldots,\varepsilon_n)\in\{V,F\}^n$ es V). □
Será conveniente que los alumnos fueran refrescando el procedimiento que usan para el cálculo de las tablas de verdad de una fórmula booleana dada.
Hay una forma natural de interpretar el proceso de sustitución :

Definición 12.3.12. Llamaremos función booleana de n parámetros a toda aplicación :

$$f\ :\{V,F\}^n\longrightarrow\{V,F\}$$

Denotaremos por $\mathcal{B}_n$ el conjunto de las funciones booleanas de n parámetros.

Como primera observación tenemos que el cardinal de $\mathcal{B}_n$ es

$$2^{2^n}$$

Para probarlo, baste observar que la funciones booleanas en $\mathcal{B}_n$ están en biyección con los subconjuntos del conjunto $\{V, F\}^n$ que tiene cardinal 2^n. Así podemos decir que una interpretación del Cálculo Propsicional como la anterior es simplemente la construcción de una aplicación :

$$D \ : \{ \text{ fórmulas booleanas}\} \longrightarrow \bigcup_{n\in\mathbb{N}} \mathcal{B}_n$$

La función booleana asociada a una fórmula del Cálculo Proposicional viene dada obviamente por las sustituciones :

$$\Phi(X_0, \ldots, X_n) \longmapsto \Phi \ : D^{n+1} \longrightarrow D$$

con la defínción natural. Estas asignaciones hacen aparecer nociones relevantes en la modelización clásica de la informática como los circuitos booleanos que evalúan funciones booleanas y que serán discutidos en pginas posteriores de este manuscrito. Nótese que varias fórmulas booleanas pueden definir una misma función booleana, con lo que el estudio de fórmulas booleanas no resulta tan evidente.
Unas clases relevantes de fórmulas booleanas se reinterpretan como problemas o como lenguajes cuya complejidad ser esencial estudiar posteriormente.

Definición 12.3.13. En las notaciones anterios, definimos los siguientes lenguajes :

i)
$$SAT := \{\Phi \ : \ \Phi \text{ es una fórmula booleana satisfacible}\}$$

ii)
$$CNF - SAT := \{\Phi \ : \ \Phi \in SAT, \ \Phi \text{ está en forma normal conjuntiva}\}$$

iii)
$$3SAT := \{\Phi \ : \ \Phi \in CNF - SAT, \ \text{ las claúsulas de } \Phi \text{ involucran solamente 3 variables } \}$$

iv)
$$TAUT := \{\Phi \ : \ \Phi \text{ es una tautología}\}$$

12.4. NP−completitud : Teoremas de Cook y Karp.

Ya introdujimos anteriormente las clases de problemas **NP** y co−**NP**. Los problemas del cálculo Proposicional antes descritos verifican las siguientes propiedades :

Teorema 12.4.1. *Los problemas anteriores, verifican las siguientes propiedades :*

i) $SAT, CNF - SAT, 3SAT \in$ **NP**,

ii) $TAUT \in co-$**NP**.

Demostración. Usaremos el Ejercicio III.1 de la Hoja III, para observar que hay una máquina de Turing determinística tal que si le damos $\Phi(X_1, \ldots, X_n)$ una fórmula bien formada del cálculo Proposicional, codificada sobre el alfabeto del Cálculo Proposicional y dado $(\varepsilon_1, \ldots, \varepsilon_n) \in \{\mathbf{V}, \mathbf{F}\}^n$ un valor de verdad, la máquina M devuelve :

$$\Phi(\varepsilon_1, \ldots, \varepsilon_n) \in \{\mathbf{V}, \mathbf{F}\}.$$

Para el problema SAT, el proceso indeterminista irá como sigue :

INPUT $\Phi(X_1, \ldots, X_n)$ una fórmula booleana del Cálculo Proposicional
Guess $(\varepsilon_1, \ldots, \varepsilon_n) \in \{\mathbf{V}, \mathbf{F}\}^n$
Eval $D := \Phi(\varepsilon_1, \ldots, \varepsilon_n) \in \{\mathbf{V}, \mathbf{F}\}$.
ACEPTAR si $D = \mathbf{V}$.
END

Del mismo modo, podemos mostrar que CNF–SAT y 3SAT están en la clase **NP**. En cuanto a **TAUT**, es claramente un problema cuyo complementario está en **NP**.

□

12.4.1. Reducciones

En la literatura se pueden encontrar varios conceptos de reducción. Una reducción es una simplificación de un problema en otro. Normalmente, en cuanto sigue, haremos referencia a reducciones de Karp (también llamadas polynomial-time many-one reductions); pero, por completitud mostraremos las tres reducciones descritas por los padres de la **NP**–completitud: Cook, Karp y Levin.

Definición 12.4.1 (**Reducción de Cook).** Dados dos lenguajes $L, L' \subseteq \Sigma^*$, decimos que L es Cook reducible a L' (también llamada reducción de Turing) si existe una máquina de Turing M con oráculo L', que finaliza sus computaciones en tiempo polinomial en el tamaño de la entrada, tal que el lenguaje aceptado por $M^{L'}$ es L.

En esencia, se trata de lo siguiente: Para un input $x \in \Sigma^*$, el problema de pertenencia a L (i.e. evaluar $\chi_L(x)$) se resuelve mediante la aplicación de la máquina M con oráculo L' a x. Por ejemplo, si $L' \in \mathbf{P}$ y L es Cook reducible a L', entonces $L \in \mathbf{P}$.

Definición 12.4.2 (**Reducción de Karp).** Dados dos lenguajes $L, L' \subseteq \Sigma^*$, decimos que L es Karp reducible a L' si existe una función $f \in \mathbf{PF}$ (evaluable en tiempo polinomial) tal que para cada $x \in \Sigma^*$, se verifica:L

$$x \in L \Longleftrightarrow f(x) \in L'.$$

De nuevo, para resolver el problema de pertenencia a L para un input $x \in \Sigma^*$, primero evaluamos $f(x)$ y luego aplicamos cualquier algoritmo que resuelva L' a $f(x)$. En particular, si $L' \in \mathbf{P}$ y si L es Karp reducible a L', entonces $L' \in \mathbf{P}$.
Es claro que una reducción de Karp induce una reducción de Cook: La máquina con oráculo M es una máquina que evalúa f, que trata la cinta de output como cinta del oráculo y que, al alcanzar un estado final aceptador pasa al estado QUERY con oráculo L' y acepta si y solamente el oráculo devuelve aceptar.

12.4.1.1. Problemas de Búsqueda (Search Problem).

Los problema de búsqueda y sus reducciones, fueron la motivación de la aproximación de Levin a la clase **NP**.
Un problema de búsqueda se define del modo siguiente: Sea $R \subseteq (\Sigma^*)^2$ una relación (en ocasiones variedad de incidencia o solution variety, según autores y contexto). Tenemos dos proyecciones canónicas $\pi_i^{(R)} : R \longrightarrow \Sigma^*$, $i = 1, 2$. Para cada $x \in \Sigma^*$ disponemos de dos fibras $(\pi_1^{(R)})^{-1}(x)$ y $(\pi_2^{(R)})^{-1}(x)$. Nos ocuparemos de la primera, aunque la segunda es simétrica para nuestras disquisisiones.
Por ejemplo, sea $\mathbb{F}_q$ un cuerpo finito y para cada lista de grados $(d) := (d_1, \ldots, d_n)$ consideremos $\mathcal{P}_{(d)}$ el conjunto formado por todas listas $(f_1, \ldots, f_n)$ de polinomios $f_i \in \mathbb{F}_q[X_1, \ldots, X_n]$ con $deg(f_i) = d_i$, $1 \leq i \leq n$. Con una ordenación adecuada, de coeficientes, tomando $\Sigma^* := \mathbb{F}_q$, podemos considerar la relación $V_{(d)} \subseteq \mathcal{P}_{(d)} \times \mathbb{F}_q^n \subseteq (\Sigma^*)^2$ dada por la siguiente igualdad:

$$V_{(d)} := \{(f, x) \ : \ f(x) = 0\}.$$

La fibra $(\pi_1^{(R)})^{-1}(f)$ son las soluciones diofánticas (en $\mathbb{F}_q$) del sistema de ecuaciones definido por la lista f, mientras que $(\pi_2^{(R)})^{-1}(x)$ son los sistemas de ecuaciones que se anulan en $x \in \mathbb{F}_q$.
Dada una relación R, una función $\varphi : \Sigma^* \longrightarrow \Sigma^* \cup \{\emptyset\}$ resuelve el problema de búsqueda R si para cada $x \in \Sigma^*$ viene dada por:

$$\varphi(x) := \begin{cases} y \in (\pi_1^{(R)})^{-1}(x), & \text{para algún } y, \text{ si } (\pi_1^{(R)})^{-1}(x) \neq \emptyset \\ \emptyset & \text{en otro caso} \end{cases}$$

Es decir, φ devuelve algún punto de la fibra en el caso de fibra no vacía. En el caso anterior, una función que resuelve el problema de búsqueda definido por la primera proyección sería un resolvedor de ecuaciones polinomiales sobre cuerpos finitos, mientras que el problema de búsqueda simétrico sería un interpolador.
Un problema decisional asociado a un problema de búsqueda $R \subseteq \Sigma^*$, es el problema de decidir si la fibra es no vacía, es decir el lenguaje siguiente:

$$S_R := \{x \in \Sigma^* \ : \ \left(\pi_1^{(R)}\right)^{-1}(x) \neq \emptyset\}.$$

En un sentido amplio, tanto el Problema X de Hilbert como el Nullstellensatz de Hilbert son problemas decisionales asociados a problemas de búsqueda donde la relación es la variedad de incidencia de Room–Kempf o la *"solution variety"* de M. Shub y S. Smale, adaptada, en cada caso, al cuerpo correspondiente. Lo mismo puede decirse de problemas de optimización o de factibilidad de solución de sistemas de ecuaciones sobre los reales. En todo caso, Levin introdujo la siguiente reducción:

Definición 12.4.3 (Reducción a la Levin). Dados $R, R' \subseteq (\Sigma^*)^2$ dos problemas de búsqueda, una reducción de Levin de R a R' es un par de funciones (f, g) dadas mediante:

- La función $f : \Sigma^* \longrightarrow \Sigma^*$ es una reducción de Karp de S_R a $S_{R'}$, es decir, $f \in \mathbf{PF}$ y para cada $x \in \Sigma^*$, $x \in S_R$ si y solamente si $f(x) \in S_{R'}$.
- La función $g : D(g) \subseteq (\Sigma^*)^2 \longrightarrow \Sigma$, también está en $\mathbf{PF}$ y verifica que si $x \in S_R$ y si $x' = f(x)$ entonces,
$$\forall y' \in (\pi_1^{(R')})^{-1}(x') \Longrightarrow (x, g(x, y')) \in R.$$

En suma, si R es Levin reducible a R' y si disponemos de un algoritmo polinomial que resuelve R', podemos resolver el problema de búsqueda para $x \in R$, comenzando con la aplicación de f, obteniendo $f(x)$. Si $f(x) \notin S_{R'}$, devolvemos $\emptyset$, en otro caso, resolvemos el problema de búsqueda para $f(x)$ (con respecto a R') obteniendo y' y terminamos devolviendo $g(x, y')$. Si R' se resuelve en tiempo polinomial, entonces R también se resuelve en tiempo polinomial.
No insistiremos mucho más en los problemas de búsqueda en este mini–curso. Indiquemos solamente que los problemas de búsqueda en la clase $\mathcal{PC}$ (relaciones [con tallas] polinomialmente acotadas que admiten "checking" en tiempo polinomial) son Cook reducibles a problemas en **NP**. Véase [Go, 08] para un tratamiento más pormenorizado de los problemas de búsqueda en relación con la Conjetura de Cook.
Dentro de la clase **P** y sus subclases (**LOG**, **NLOG**, **NC**,...) se suelen usar reducciones *log–space* (i.e. reducciones en **LOGF**,..).

Definición 12.4.4. Decimos que una clase de complejidad **C** es reducible (Cook, Karp, Levin, log–space...) a otra clase **C'** si los problemas de la primera son (Cook, Karp, Levin, log–space...) reducibles a problemas en la segunda.

12.4.1.2. Clausura bajo reducciones

Definición 12.4.5. Sea $\mathcal{C}$ una clase de complejidad. Decimos que $\mathcal{C}$ es cerrada bajo reducciones de Karp (resp. Turing, Levin, log-space, ...) si para cualesquiera dos lenguajes L y L' tales que L es reducible Karp (resp. Cook, log-space,.....) a L' y tales que $L' \in \mathcal{C}$, entonces, $L \in \mathcal{C}$

Proposición 12.4.2. *Las siguientes clases son cerradas bajo reducciones a la Karp:*

$$\mathbf{P}, \mathbf{NP}, \mathbf{PSPACE}.$$

Demostración. Obvio. □

Debe indicarse que no se sabe si las reducciones de Cook son más fuertes que las reducciones de Karp. De hecho, ni siquiera se sabe si la clase **NP** es cerrada por reducciones de Cook (aunque se sospecha que no es así).

Problema Abierto 2. *¿Es la clase* **NP** *cerrada bajo reducciones de Cook?*

12.4.2. El Teorema de Cook: Problemas NP–completos.

Aunque en ocasiones se usan reducciones de Cook para probar que ciertos problemas son **NP**–completos, nos restringirenos a las reducciones de Karp siempre que sea posible.

Definición 12.4.6. Sea **C** una clase de complejidad, decimos que un lenguaje L es **C**–duro para reducciones Karp (resp. Cook) si todos los lenguajes S de la clase **C** son Karp reducibles (resp. Cook reducibles) a L.

Definición 12.4.7. Sea **C** una clase de complejidad, decimos que un lenguaje L es **C**–completo si verifica:

- $L \in \mathbf{C}$
- L es **C**–duro.

Proposición 12.4.3. *Sea* $\mathbf{C}' \subseteq \mathbf{C}$ *dos clases de complejidad y supongamos que* **C**' *es cerrada por reducciones à la Karp (resp. à la Cook). Sea* L *un lenguaje* **C***–completo para reducciones de Karp (resp. Cook, etc...), entonces*

$$L \in \mathbf{C}' \Longrightarrow \mathbf{C} = \mathbf{C}'.$$

Demostración. Obvio por la noción de cerrada para reducciones de cierto tipo. □

Esta Proposición muestra la potencialidad de los problemas completos en una clase: ellos parecen condensar todo el potencial de la clase de complejidad y, por tanto, si "caen" a una clase menor, pero estable por las reducciones consideradas, toda la clase en la que son completas "cae" también en esa subclase. Diremos que ambas clases colapsan.

Corollario 12.4.4. *Con las notaciones anteriores, se tiene*

i) Sean $L, L' \in \mathbf{NP}$ *y supongamos*

- L *es Karp reducible a* L'
- $L' \in \mathbf{P}$

Entonces, también $L \in \mathbf{P}$. *En particular,* $\mathbf{P} = \mathbf{NP}$ *si y solamente si existe un problema* **NP***-completo* L *tal que* $L \in \mathbf{P}$.

ii) Sean $L, L' \in \mathbf{PSPACE}$ *y supongamos*

- L *es Karp reducible a* L'
- $L' \in \mathbf{P}$

Entonces, también $L \in \mathbf{P}$. *En particular,* $\mathbf{P} = \mathbf{PSPACE}$ *si y solamente si existe un problema* **NP***-completo* L *tal que* $L \in \mathbf{P}$.

Demostración. La demostración es obvia □

Teorema 12.4.5 ([Krp, 72]). *El siguiente problema es* **NP**$-$ *completo para reducciones de Karp.*

$$K := \{(c_M, x, t) \ : \ t \in \{1\}^*, \ x \in L(M), \ t_M(x) \leq t\}.$$

donde $c_M \in \Sigma_1^*$ *es el código de una máquina de Turing indeterminista,* $x \in \Sigma_0^*$ *y* t *es dado en unario.*

Demostración. Usando la máquina universal, podemos simular el cálculo de una máquina de Turing indeterminista dada por su código sobre un dato y verificar que el número de pasos que da es a lo sumo t. esta simulación es, obviamente, indeterminista, y tarda un tiempo polinomial (i.e. cuadrático) en t_M, luego como t actúa como contador, el tiempo es a lo sumo $O(t^2)$ y habremos terminado pues el tamaño del input es $|c_M| + |x| + t$. Veamos que K es **NP**$-$ completo. Para ello, sea L un lenguaje en la clase **NP** y sea M la máquina de Turing indeterminista que acepta L, sea $p(n)$ el polinomio tal que $t_M(n) \leq p(n)$ Definamos la siguiente reducción de Karp, dada por la máquina T :

INPUT $x \in \Sigma_0^*$,
Hallar $n := |x|$,
EVAL $p(n)$ y escribe el resultado en unario.
ESCRIBE $(c_M, x, p(n))$.
END.

El tiempo de cálculo de esta máquina está acotado por :

- El tiempo de hallar n (tiempo $O(n)$)
- El tiempo de evaluar $p(n)$ (como los polinomios son constructibles en tiempo $O(p)$).
- El tiempo de escribir $p(n)$ en unario (obviamente, a lo sumo $p(n)$).
- El tiempo de escribir el código c_M (obviamente constante $O(1)$ porque el código de M no depende de x)
- El tiempo de escribir x (i.e. tiempo lineal $O(n)$
- El tiempo de escribir $p(n)$ (i.e. $p(n)$).

Luego esta máquina trabaja en tiempo polinomial en n. Adems $Res_T(x) := (c_M, x, p(n))$ y claramente :

$$x \in L(M) \Longleftrightarrow Res_T(x) \in K$$

□

EL anterior problema no es un problema muy natural (i.e. un problema con el que uno se encuentre a lo largo de un proceso de computación. Por eso, se encontraron otros problemas **NP**$-$completos más naturales como los siguientes :

Teorema 12.4.6. *[Cook, 71] Los siguientes problemas son* **NP**$-$*completos :*

$$SAT, CNF - SAT, 3SAT.$$

En particular, el siguiente problema es $co - NP-$*completo :*

$$TAUT.$$

Demostración. □

Corollario 12.4.7. *El Hilbert Nullstellensatz* **HN** *sobre cuerpos primos es un problema* **NP**$-$*duro y el siguiente es* **NP**$-$*completo:*

$$\mathbf{SAT - HN} := \{f = (f_1, \ldots, f_{n+1}) \in \mathcal{P}_{(3)}^K \ : \ \exists x \in K^n, f_1(x) = 0, \ldots, f_{n+1}(x) = 0\}.$$

Demostración. En Problema **??** se define el Nullstellensatz como el lenguaje:

$$\mathbf{HN} := \{f = (f_1, \ldots, f_{n+1}) \in \mathcal{P}^K_{(3)} \ : \ \exists x \in \mathbb{K}^n, f_1(x) = 0, \ldots, f_{n+1}(x) = 0\}.$$

Se tratará de un problema **NP**–duro porque `SAT` es Karp reducible a él. Pero no se sabe si es **NP**–completo puesto que no podemos "adivinar" de manera controlada y simple las soluciones en $\mathbb{K}^n$ (recuérdese que $\mathbb{K}$ es un cuerpo de cardinal infinito) a partir de los coeficientes y, por tanto, no podemos garantizar que esté en **NP**.
En cambio sí está en **NP** su versión **SAT**–**HN**: la búsqueda de soluciones no ya en $\mathbb{K}^n$ sino en $K^n = \mathbb{F}_q^n$, que son fáciles de "adivinar". Como **SAT** también es reducible a **SAT**–**HN**, será un problema **NP**–completo. □

Problema 12.4.1 (CLIQUE). Un grafo $G := (V, E)$ se dice completo si E contiene todas las posibles aristas entre cualesquiera dos nodos de V. El lenguaje `CLIQUE` es el lenguaje dado por los pares (G, k) donde $G = (V, E)$ es un grafo y k es un entero positivo, de tal modo que G contiene un subgrafo completo de cardinal mayor o igual que k.

Teorema 12.4.8 ([Krp, 72])**.** *El problema* `CLIQUE` *es* **NP***–completo.*

Demostración. (Sketch) Reduciremos `SAT--CNF` a `CLIQUE` del modo siguiente. Supongamos que la fórmula Φ es la conjunción

$$\Phi := C_1 \wedge \cdots \wedge C_r,$$

donde $C1, \ldots, C_r$ son cláusulas que involucran variables en $\{X_1, \ldots, X_n\}$ y literales $\{x_1, \ldots, x_k\}$. Definimos un grafo $G = (V, E)$ del modo siguiente:

- $V := \{(x_j, C_i) \ : \text{ el literal } x_j \text{ aparece en la cláusula } C_i\}$,
- $E := \{((x_j, C_i), (x_m, C_s) \ : \ C_i \neq C_s, \ \ \neg x_j \neq x_m\}$, donde hemos supuesto que la doble negación es la variable original.
- $k = r$ (i.e. el número de cláusulas).

Se trata de probar que el grafo G así construido posee un subgrafo completo de cardinal mayor o igual que r si y solamente si la fórmula $\Phi = C_1 \wedge \cdots \wedge C_r$ es satisfactible.
Para probar $\Longrightarrow$, supongamos que G posee un subgrafo completo U de cardinal r. Observemos que dados dos nodos $(x_{k(i)}, C_i), (x_{k(j)}, C_j) \in U$, entonces $C_i \not C_j$ porque, al estar en U, tienen que estar conectados (U es completo). Pero, además, cono U tiene cardinal r, no puede haber más de un literal asociado a cada nodo y cláusula en U. En suma, tenemos

$$U := \{(x_{k(1)}, C_1), \ldots, (x_{k(r)}, C_r)\}.$$

Además, es perfectamente posible que $x_{k(i)} = x_{k(j)}$ para $i \neq j$, pero, como $(x_{k(i)}, C_i)$ y $(x_{k(i)}, C_j)$ están conectados en U, es seguro que $x_{k(i)} \neq \neg x_{k(j)}$ para cada $i \neq j$. Por tanto, considero una asignación de verdad del tipo siguiente:
Dada la variable X_i, si existe ℓ, $1 \leq \ell \leq r$ tal que $X_i = x_{k(\ell)}$, elijamos $\varepsilon_i = 1$. Si, por el contrario, existe j, $1 \leq j \leq r$ tal que $x_{k(j)} = \neg X_i$, entonces elejimos $\varepsilon_i = 1$. Finalmente, si X_i y $\neg X_i$ no aparecen el la lista de literales $x_{k(1)}, \ldots, x_{k(r)}$, asigamos $\varepsilon_i \in \{0, 1\}$ cualquier valor que deseemos. La lista

$$\underline{\varepsilon} := (\varepsilon_1, \ldots, \varepsilon_n) \in \{0, 1\}^n,$$

está bien definida con las reglas anteriores. La razón es, obviamente, que las reglas anteriores no pueden afectar en dos casos a ninguna variable X_i porque no pueden aparecer $x_{k(i)}$ y $\neg x_{k(i)}$ en la lista $x_{k(1)}, \ldots, x_{k(r)}$. Además, es claro que con estas asignaciones, cada clásula C_i es satisfactible.
Para probar $\Longleftarrow$, supongamos que Φ es satisfactible. Sea dada la asignación de verdad $\underline{\varepsilon} := (\varepsilon_1, \ldots, \varepsilon_n) \in \{0, 1\}^n$ se tenga que $eval(\Phi, \underline{\varepsilon}) = 1$. Entonces, para cada cláusula, C_i hay un literal $x_{k(i)}$ tal que $eval(x_{k(i)}, \underline{\varepsilon}) = 1$. Definamos, por tanto, el subgrafo

$$U := \{(x_{k(1)}, C_1), \ldots, (x_{k(r)}, C_r)\}.$$

Veamos que es completo. Obviamente, dados $i \neq j$, $C_i \neq C_j$. Pero, además, dados $i \neq j$, $eval(x_{k(i)}, \underline{\varepsilon}) = 1$ and $eval(x_{k(j)}, \underline{\varepsilon}) = 1$, luego no es posible que $x_{k(j)} = \neg x_{k(i)}$. En conclusión, para $i \neq j$, existe arista entre $(x_{k(i)}, C_i)$ y $(x_{k(j)}, C_j)$. Y U es un subgrafo completo de cardinal r.

□

Problema 12.4.2. 3COLOR Dado un grafo $\mathcal{G} := (V, E)$ y una paleta con tres colores $\{A, R, N\}$, decidir si se pueden asignar colores a los vértices de tal manera que dos vértices adyacentes no tengan el mismo color. Es decir, si $V := \{1, \ldots, n\}$, una aplicación:

$$\sigma : V \longrightarrow \{A, R, N\},$$

tal que para cada $i \neq j$, $\sigma(i) \neq \sigma(j)$.

El Problema está, obviamente, en la clase **NP**...

Problema 12.4.3 (**3COLOR**, versión Nullstellensatz)**.** Dado un grafo $\mathcal{G} := (V, E)$, con $V = \{1, \ldots, n\}$, decidir si el siguiente sistema de ecuaciones posee solución en $\mathbb{C}^n$:

$$X_1^3 - 1 = 0, \ldots, X_n^3 - 1 = 0, X_i^2 + X_i X_j + X_j^2 = 0, \ \ \forall \{i, j\} \in E.$$

Hemos identificado los tres colores con las tres raíces cúbicas de la unidad: $\{1, e^{\frac{2\pi i}{3}}, e^{\frac{4\pi i}{3}}\}$. De ahí las ecuaciones $X_i^3 - 1 = 0$.
Además, módulo $(X_i^3 - 1 = X_j^3 - 1 = 0)$

$$X_i^2 + X_i X_j + X_j^2 = \frac{X_i^3 - X_j^3}{X_i - X_j} = (X_i - e^{\frac{2\pi i}{3}} X_j)(X_i - e^{\frac{4\pi i}{3}} X_j),$$

$$[X_i \neq X_j] \Longleftrightarrow \left[X_i^2 + X_i X_j + X_j^2 = 0\right].$$

De hecho, es fácil probar que el número total de coloraciones es

$$\frac{deg(V)}{3!},$$

donde $deg(V)$ es el número de soluciones complejas del sistema de ecuaciones:

$$X_1^2 - 1 = 0, \ldots, X_n^3 - 1 = 0, X_i^2 + X_i X_j + X_j^2 = 0, \ \ \forall \{i, j\} \in E.$$

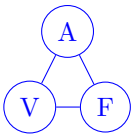
Teorema 12.4.9 ([Krp, 72])**.** *El problema* `3-COLOR` *es* **NP***–completo.*

Demostración. Veremos una reducción de **3SAT** a **3 COLOR** del modo siguiente:
Dada una fórmula $\varphi := \bigwedge_{i=1}^{m} C_i$, cláusulas $C_1, \ldots, C_m$ cada una con 3 variables.
Defino un grafo $\mathcal{G} := (V, E)$, donde V es dado por:

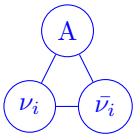
- Un vértice por cada literal $\{X_1, \ldots, X_n, (\neg X_1), \ldots, (\neg X_n)\}$ que denotaremos mediante ν_i y $\bar{\nu_i}$, $1 \leq i \leq n$
- 5 vértices por cláusula que denotaremos mediante $j_1, \ldots, j_5$ para $1 \leq j \leq m$.
- Tres vértices adicionales $\{V, F, A\}$.

Introduciremos las aristas E mediante:

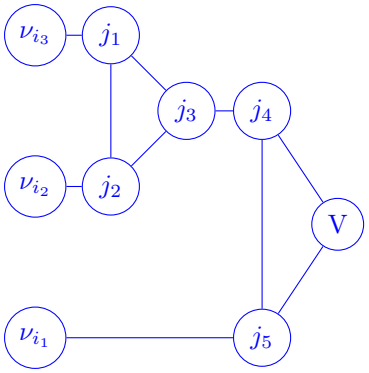
- Un triángulo con los tres nodos especiales:



- Un triángulo con los nodos ν_i y $\bar{\nu_i}$ y A para cada i:



- Y un grafo especial para cada cláusula:

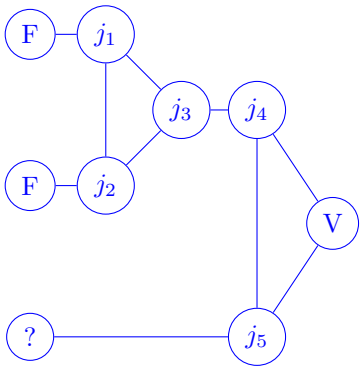


- Si el triángulo formado por $\{V, F, A\}$ es 3–coloreable, cada uno de estos símbolos es un color distinto... V, F, A.

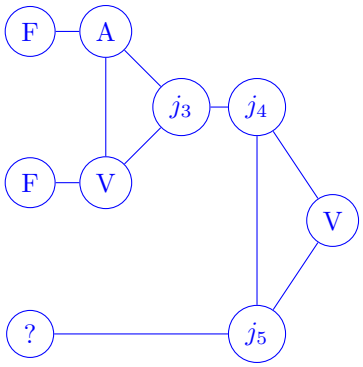
- Si los triángulos formados por $\{\nu_i, \bar{\nu_i}, A\}$ son 3−coloreables, los "colores" asignables a ν_i y $\bar{\nu_i}$ sólo pueden ser V o F y además son complementarios.

- El último grafo sólo es 3−colorable (con estas restricciones) si alguno de los ν_i's está coloreado con V.

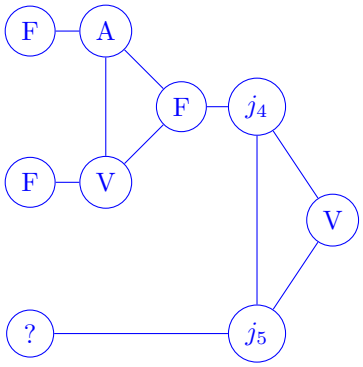
En suma, el grafo es 3−coloreable si y solamente si la fórmula original es satisfactible. Para verlo supongamos que alguna entrada tiene dos falsos en dos de los índices iniciales:



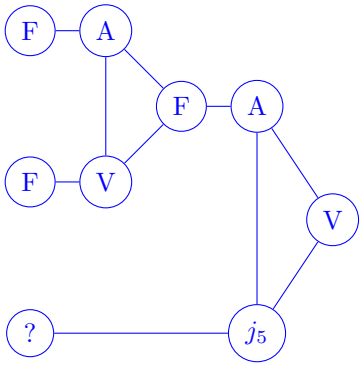
Entonces, los restantes nodos sólo se pueden colorear como siguen (o simétricamente en las posiciones de A y V):



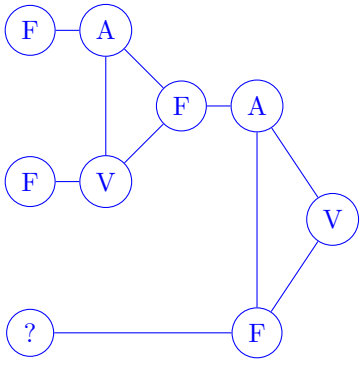
Seguidamente obligamos al tercer nodo del trángulo superior

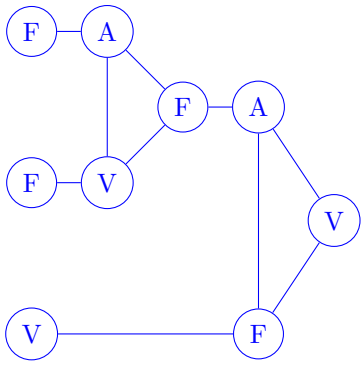


Que fuerza también el color del siguiente:



Lo que fuerza los colores de los otros dos nodos:





El resto de los casos se siguen igualmente.

□

Problema 12.4.4 (**HC** Hamiltonian Circuit)**.** Un circuito hamiltoniano en un grafo orientado $G = (V, E)$ es una secuencia cerrada de vértices $\nu_1, \ldots, \nu_s$ de tal modo que se pude pasar de cada uno al siguiente $(\nu_i, \nu_{i+1}) \in V$ y $(\nu_s, \nu_1) \in V$. El lenguaje **HC** es el lenguaje formado por todos los grafos orientados G que poseen un circuito hamiltoniano pasando por todos los nodos del grafo.

Teorema 12.4.10. *El problema* **HC** *es* **NP***–completo.*

Demostración. (Hint) Reducir `3SAT` a **HC**. □

Problema 12.4.5 (**SUBSET SUM**)**.** Se trata de las listas finitas de números enteros $\{a_i \ : \ 1 \leq i \leq n\} \subseteq \mathbb{Z}$ tales que existe $S \subseteq \{1, \ldots, n\}$ verificando

$$\sum_{i \in S} a_i = 0.$$

Una variante de este problema es el Problema de la Mochila.

Problema 12.4.6 (**KANPSACK**)**.** Dada una lista de enteros $\{a_i \ : \ 1 \leq i \leq n\} \subseteq \mathbb{Z}$ y dado $k \in \mathbb{N}$, decidir si

$$\exists S \subseteq \{1, \ldots, n\}, \sum_{i \in S} a_i = l.$$

Observación 12.4.1. Ambos problemas pueden reescribirse como un problema de eliminación (Hilbert Nullstellensatz):
Decidir si el siguiente sistema de ecuaciones polinomiales con coeficientes en $\mathbb{Z}$ *posee una solución en* $\mathbb{C}^n$*:*

$$X_1^2 - X_1 = 0, X_2^2 - X_2 = 0, \ldots, X_n^2 - X_n = 0, k - \sum_{i=1}^{n} a_i X_i = 0.$$

Teorema 12.4.11. *Tanto* **SUBSET SUM** *como* **KNAPSACK** *son problemas* **NP***–completos.*

Problema 12.4.7 (**Minimum Distance**)**.** En Teoría de Códigos Correctores de Errores trabajamos sobre un cuerpo finito $\mathbb{F}_q := GF_q$ que actúa como alfabeto. El espacio $\mathbb{F}_q^n$ es un espacio métrico con la distancia de Hamming:

$$d_H(x, y) := \sharp\{i \ : \ 1 \leq i \leq n, \ x_i \neq y_i\}.$$

Un código es un subespacio lineal $C \subseteq \mathbb{F}_q^n$, que podemos definir mediante sus ecuaciones lineales por una matriz $H \in \mathcal{M}_{m \times n}(\mathbb{F}_q)$. La capacidad de un código viene determinada por el peso mínimo de las palabras en C o, equivalentemente, por el mínimo de las distancias de sus elementos.
Definimos el lenguaje **Minimum Distance** como los pares (H, r) donde H es una matriz en $\mathcal{M}_{m \times n}(\mathbb{F}_q)$ y r es un número natural tal que existe x verificando

$$Hx^t = 0, \ \ weight(x) := d_H(x, 0) \leq r.$$

Teorema 12.4.12. *[Vrd,97] El problema* **Minimum Distance** *es* **NP**–*completo.*

Teorema 12.4.13. *El problema de Optimización Entera* **IP** *es* **NP**−*completo. Esto es, el problema siguiente :*

Problema 12.4.8 (**IP**, Optimización Entera)**.** Dados $A \in \mathcal{M}_{n\times m}(\mathbb{Z})$ una matrix con coeficientes enteros y dado $b \in \mathbb{Z}^m$ un vector con coordenadas enteras, decidir si

$$A \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \leq b,$$

tiene solución en $\mathbb{Z}^n$.

Insisto en que la galería de problemas **NP**−completos conocidos es más amplia. Uno de los trabajos del curso nos dará unos cuantos de entre ellos. Una idea importante (esencial en la evolución de la informática teórica) será hallar un algoritmo polinomial para cualquiera de esos problemas con lo que todos ellos tendrán algoritmos de tipo polinomial.
Los problemas **NP**–completos adolecen del *don de la ubicuidad*. Se encuentran en casi cualquier ámbito de la computación. Lo que sigue es una galería de problemas **NP**–completos de diversos ámbitos del conocimiento. La galería no pretende ser completa, dado que se conocen miles de ejemplos, sino simplemente ilustrar algunos de esos casos. El lector interesado puede acudir al ya clásico [GaJo, 79] o a la lisa/resumen de Wikipedia en

`http://en.wikipedia.org/wiki/List_of_NP-complete_problems`

12.4.3. El Teorema de Ladner

El Teorema de Ladner introduce la clase siguiente:

$$\mathbf{NPI} := \{L \in \mathbf{NP} \ : \ L \notin \mathbf{NP} - \text{completo}, \ \ L \notin \mathbf{P}\}.$$

Teorema 12.4.14. *[Lad, 75]*

$$\mathbf{P} \neq \mathbf{NP} \iff \mathbf{NPI} \neq \emptyset.$$

Por ello, se buscan ejemplos de problemas en la clase **NP** que no sean **NP**-completos y de los que se pueda probar que tampoco son problemas en **P**. Los intentos, hasta ahora infructuosos, se han orientado hacia problemas como los siguientes:

i) Graph Isomorphism

ii) Factoring Integers

iii) Group Isomorphism, Ring Isomorphism, Ring automorphism

iv) Discrete Log Problem

v) Turnpike Problem

vi) Winning parity games

vii) Highest chance of winning stochastic games

viii) .....

12.5. La clase PSPACE

La clase de problemas en **PSPACE** se define del modo siguiente :

Definición 12.5.1. Con las notaciones anteriores definiremos la clase de complejidad :

$$\mathbf{PSPACE} := \bigcup_{k\in\mathbb{N}} \mathbf{DSPACE}(n^k).$$

Observación 12.5.1. Podríamos haber definido también la clase de complejidad en espacio indeterminista siguiente :

$$\mathbf{NPSPACE} := \bigcup_{k\in\mathbb{N}} \mathbf{NSPACE}(n^k).$$

Sin embargo, en función del Teorema de Savitch

$$\mathbf{NSPACE}(s) \subseteq \mathbf{DSPACE}(s^2),$$

uno concluye obviamente la siguiente igualdad :

$$\mathbf{NPSPACE} = \mathbf{PSPACE}.$$

Por las mismas razones, concluimos la siguiente cadena de contenidos :

Proposición 12.5.1. *Con las anteriores notaciones, se tiene la siguiente cadena de inclusiones :*

$$\mathbf{LOG} \subseteq \mathbf{NLOG} \subseteq \mathbf{P} \subseteq \mathbf{BPP} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} = \mathbf{NPSPACE},$$

donde

$$\mathbf{LOG} := \mathbf{DSPACE}(log\ n),\ \ \mathbf{NLOG} := \mathbf{NSPACE}(log\ n).$$

12.5.1. Problemas PSPACE-completos

De nuevo, el desconocimiento de la relación entre **P** y **PSPACE**, lleva a la búsqueda de problemas completos en la clase **PSPACE** para reducciones de Karp. En este curso no entraremos en profundidad en disquisiciones relativas a espacio, por lo que nos conformamos con indicar uno de los ejempolos esenciales de problema **PSPACE**−completo : QBF. Para definir este problema completo, retomemos las fórmulas del Cálculo Proposicional.

Definición 12.5.2. Llamaremos fórmula booleana cuantificada (en forma prenexa) con variables libres a toda lista :

$$\varepsilon_1 X_1 \cdots \varepsilon_n X_n\ \Phi(X_1, \ldots, X_n, Y_1, \ldots, Y_m),$$

donde

i) $\varepsilon_1, \ldots, \varepsilon_n \in \{\forall, \exists\}$ (etso es, son cuantificadores existenciales (i.e. $\exists$) o universales (i.e. $\forall$)).

ii) $\Phi(X_1, \ldots, X_n, Y_1, \ldots, Y_m)$ es una fórmula bien formada del Cálculo Proposicional que involucra al conjunto de variables $\{X_1, \ldots, X_n, Y_1, \ldots, Y_m\}$.

Las variables $\{X_1, \ldots, X_n\}$ se denominan variables ligadas o cuantificadas y las variables $\{Y_1, \ldots, Y_m\}$ se denominan variables libres.
Llamaremos fórmulas booleanas cuantificadas sin variables libres a aquellas que involucran solamente variables afectadas por un cuantificador, es decir, sin variables libres o con todas las variables ligadas, esto es, fórmulas de la forma :

$$\varepsilon_1 X_1 \cdots \varepsilon_n X_n\ \Phi(X_1, \ldots, X_n),$$

con $\varepsilon_1, \ldots, \varepsilon_n \in \{\forall, \exists\}$ y $\Phi(X_1, \ldots, X_n)$ una fórmula bien formada del Cálculo Proposicional.

Llamaremos fórmula booleana cuantificada libre de cuantificadores a las que tiene la siguiente expresión :

$$\Phi(Y_1, \ldots, Y_m)$$

Esto es, todas las variables son libres y no aparecen cuantificadores al principio de la palabra. Más simplemente, las fórmulas libres de cuantificadores son simplemente las fórmulas bien formadas del Cálculo Proposicional.

Observación 12.5.2. Obsrvese que las fórmulas booelanas cuantificadas son palabras sobre un alfabeto finito : el alfabeto Σ_3 dado mediante :

$$\Sigma_3 := \{\forall, \exists\} \dot{\bigcup} \Sigma_2,$$

donde Σ_2 es el alfabeto para escribir fórmulas boolenas (esto es, fórmulas bien formadas del Cálculo Proposicional). A la teoría que considera las fórmulas booleanas cuantificadas se la denomina Cálculo de Predicados del Cálculo Proposicional.

A las fórmulas booleanas cuantificadas en forma prenexa se les pueden asignar valores de verdad o falsedad conforme a las reglas esperadas. Para ello, utilizaremos la siguiente definición recursiva :

Definición 12.5.3. Definiremos la función de asignación de verdad para fórmulas boolenas cuantificadas de la forma siguiente : Sea Φ una fórmula booleana cuantificada que depende de las variables (libres y ligadas) $\{X_1, \ldots, X_n, Y_1, \ldots, Y_m\}$, con $\{X_1, \ldots, X_n\}$ ligadas y $\{Y_1, \ldots, Y_m\}$ libres. Sea

$$\underline{\tau} := (\tau_1, \ldots, \tau_m) \in \{\mathbf{V}, \mathbf{F}\}^m,$$

una asignación de verdad. Definiremos

$$Asig(\Phi, \underline{\rho})$$

del modo siguiente :

i) Si Φ es una fórmula boolena libre de cuantificadores, definiremos :

$$Asig(\Phi, \underline{\rho}) := \Phi(\underline{\rho}),$$

del modo usual. Nótese que si Φ no viene precedida de cuantificadores,luego no hay variables ligadas y es una fórmula del Cálculo Proposicional. Se aplica entonces lo descrito en la Sección correspondiente.

ii) Supongamos que Φ tiene la forma :

$$\Phi := \exists X_1 \Phi'(X_1, \ldots, X_n, Y_1, \ldots, Y_m),$$

donde $\Phi'(X_1, \ldots, X_n, Y_1, \ldots, Y_m)$ es una fórmula booleana cuantificada con variables ligadas $\{X_2, \ldots, X_n\}$ y variables libres $\{X_1, Y_1, \ldots, Y_m\}$. Entonces, definimos :

$$Asig(\Phi, \underline{\rho}) := \mathbf{V} \iff \text{existe } \mathbf{B} \in \{\mathbf{V}, \mathbf{F}\}, \text{ tal que } Asig(\Phi', (\mathbf{B}, \underline{\rho})) = \mathbf{V}.$$

En caso contrario, definiremos

$$Asig(\Phi, \underline{\rho}) := \mathbf{F}.$$

iii) Supongamos que Φ tiene la forma :

$$\Phi := \forall X_1 \Phi'(X_1, \ldots, X_n, Y_1, \ldots, Y_m),$$

donde $\Phi'(X_1, \ldots, X_n, Y_1, \ldots, Y_m)$ es una fórmula booleana cuantificada con variables ligadas $\{X_2, \ldots, X_n\}$ y variables libres $\{X_1, Y_1, \ldots, Y_m\}$. Entonces, definimos :

$$Asig(\Phi, \underline{\rho}) := \mathbf{V} \iff \text{para todo } \mathbf{B} \in \{\mathbf{V}, \mathbf{F}\}, \ Asig(\Phi', (\mathbf{B}, \underline{\rho})) = \mathbf{V}.$$

En caso contrario, definiremos

$$Asig(\Phi, \underline{\rho}) := \mathbf{F}.$$

Definición 12.5.4. Dada una fórmula booleana cuantificada en forma prenexa Φ con variables ligadas $\{X_1, \ldots, X_n\}$ y variables libres $\{Y_1, \ldots, Y_m\}$, diremos que Φ es un Teorema (o que es tautología, ambas nociones son equivalentes en el cálculo de predicados del Cálculo Proposicional) si para todo valor posibles $\underline{\rho} \in \{\mathbf{V}, \mathbf{F}\}^m$, se tiene :

$$asig(\Phi, \underline{\rho}) = \mathbf{V}.$$

As podemos definir el siguiente lenguaje : $QBF \subseteq \Sigma_3^*$ es lenguaje formado por las palabras sobre Σ_3^* que son fórmulas booleanas cuantificadas sin variables libres y son Teorema. Se tiene entonces,

Teorema 12.5.2 ([BaDiGa, 88], [Papa, 94])**.** *Con las notaciones anteriores, QBF es* **PSPACE**$-$*completo para reducciones de Karp, esto es,*

i) $QBF \in$ **PSPACE***,*

ii) Para cualquier lenguaje $L \subseteq \Sigma_3^*$, existe una máquina de Turing M que funciona en tiempo polinomial y que evalúa una función Res_M tal que para cada $x \in \Sigma_3^*$, $x \in L$ si y solamente si $Res_M(x) \in QBF$.

Otros ejemplos de Problemas **PSPACE**–completos son:

- Problema de Palabra para gramáticas sensibles al contexto.
- Dada una expresión regular α, decidir si el lenguaje que describe $L(\alpha)$ coincide con Σ^*.
- Generalizaciones de juegos (extendidos a tableros $n \times n$) como Hex, Sokoban o Mah Jong....

Si bien los dos primeros son relevantes en el procesamiento de lenguajes (dejando los lenguajes "tratables" en clases más simples como las libres de contexto), la gran variedad de juegos para los que se ha demostrado que la búsqueda de estrategias ganadoras es **PSPACE**–completo, da un cierto toque de popularidad y marketing del que pienso huir. El lector interesado bien puede acudir a Papadimitrou en [Papa, 94] o [**?**] y continuadores.
Obviamente, una estrategia polinomial que resuelva cualquiera de esos problemas, implicaría la igualdad de todos los contenidos descritos al inicio de esta Sección. Pero es poco esperable.

12.5.2. La Jerarquía Polinomial PH

Dado un grafo $\mathcal{G} := (V, E)$, un *conjunto independiente* (también llamado estable) es un subconjunto de vértices $S \subseteq V$ de tal modo que dos vértices cualesquiera $x, y \in S$ no están unidos por ninguna arista (en E) del grafo $\mathcal{G}$. Es decir, el subgrafo inducido es totalmente aislado y tiene tantas componentes conexas como vértices.
El siguiente es un claro problema en la clase **NP**:

$$\texttt{IND} := \{(\mathcal{G}, k) \ : \ \text{existe un conjunto independiente } S \text{ con } \sharp(S) \geq k\}.$$

Pero podríamos convertirlo en un problema decisional del tipo siguiente:

$$\texttt{EXACT IND}\{(\mathcal{G}, k) \ : \ k \text{ es el máximo cardinal de subconjuntos independientes}\}.$$

El segundo no parece pertenecer a la clase **NP** sino que parece añadir un cuantificador universal $\forall$. La concatenación alternada de cuantificadores existenciales y universales da pie a la *Jerarquía Polinomial*: **PH**

Definición 12.5.5 (PH). Se definen las clases de lenguajes siguientes:

- Se dice que un lenguaje $L \subseteq \Sigma^*$ está en la clase Σ_i, $i \in \mathbb{N}$, $i \geq 1$, si existe un lenguaje $\mathcal{L}$ en **P** y existe un polinomio univariado q, de tal modo que una palabra $x \in \Sigma^*$, $|x| = n$, está en L si y solamente si

$$Q_1 u_1 \in \Sigma^{q(n)} Q_2 u_2 \in \Sigma^{q(n)} \cdots Q_i u_i \in \Sigma^{q(n)}, \ \ (x, u_1, \ldots, u_i) \in \mathcal{L},$$

  donde $Q_j \in \{\exists, \forall\}$, $Q_j \neq Q_{j+1}$, $Q_1 = \exists$.

- Se dice que un lenguaje $L \subseteq \Sigma^*$ está en la clase Π_i, $i \in \mathbb{N}$, $i \geq 1$ si existe un lenguaje $\mathcal{L}$ en **P** y existe un polinomio univariado q, de tal modo que una palabra $x \in \Sigma^*$, $|x| = n$, está en L si y solamente si

$$Q_1 u_1 \in \Sigma^{q(n)} Q_2 u_2 \in \Sigma^{q(n)} \cdots Q_i u_i \in \Sigma^{q(n)}, \ \ (x, u_1, \ldots, u_i) \in \mathcal{L},$$

  donde $Q_j \in \{\exists, \forall\}$, $Q_j \neq Q_{j+1}$, $Q_1 = \forall$.

Como primeras observaciones se tiene:

$$\mathbf{NP} = \Sigma_1,$$

$$\text{co} - \mathbf{NP} = \Pi_1,$$

$$\Sigma_i \subseteq \Pi_{i+1},$$

$$\Pi_i \subseteq \Sigma_{i+1},$$

$$\Pi_i = \text{co} - \Sigma_i.$$

Definición 12.5.6 (**PH**)**.** Se denomina jerarquía polinomial **PH** a la clase dada mediante:

$$\mathbf{PH} = \bigcup_{i=1}^{\infty} \Sigma_i.$$

Proposición 12.5.3 (Colapsos en **PH**)**.** *Se tienen los siguientes resultados:*

- *Si existiera i tal que $\Sigma_i = \Pi_i$, entonces* **PH** $= \Sigma_i$ *(la jerarquía polinomial colapsaría al nivel i).*
- *Si* **P** $=$ **NP**, *entonces* **PH**$=$ **P** *(la jerarquía polinomial colapsaría al nivel* **P***).*

No se sabe si la jerarquía polinomial posee problemas completos.

Problema Abierto 3. *¿Existe algún problema completo (para reducciones à la Karp) en la jerarquía polinomial?*

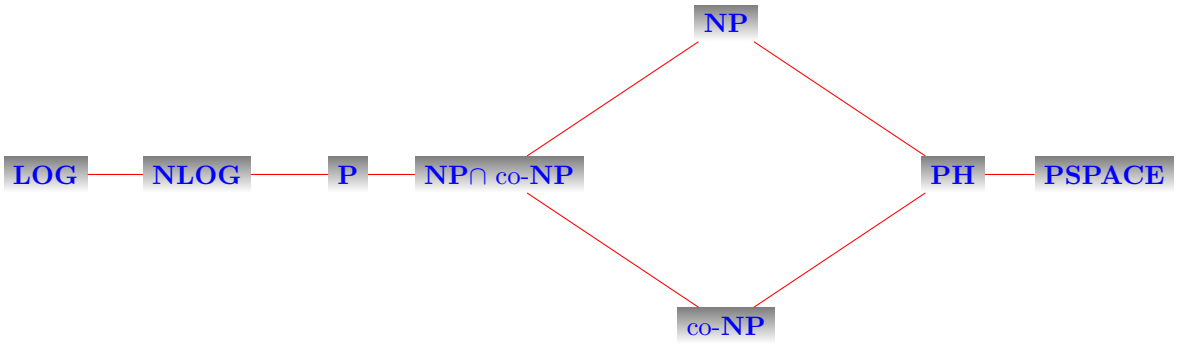
En cambio sí se conocen algunas de sus consecuencias de su existencia

Proposición 12.5.4. *Si existiera algún problema completo para la clase* **PH***, entonces la jerarquía polinomial colapsaría a algún nivel i.*

En particular, es obvio que **PH** $\subseteq$ **PSPACE**, pero si la jerarquía polinomial no colapsa, entonces **PH** $\neq$ **PSPACE**. Porque ya hemos visto que sí hay problemas completos en **PSPACE**.
Existe una generalización de las máquinas indeterministas denominadas Máquinas de Turing que Alternan *ATM*. De la misma manera que el indeterminismo refelja la clase **NP**, las ATM modelizan las clase Σ_i y Π_i. Es conocido que el tiempo polinomial en ATM's coincide con espacio polinomial; pero evitaremos esta discusión sobre ATM's y dirigiremos al lector a cualquiera de las referencias al uso.

12.6. Un grafo final

A modo de pequeño resumen de este Capítulo las relaciones entre las distintas clases inolucradas. Las flechas rojas denotan problemas abiertos.


NP
LOG
NLOG
P
NP∩ co-NP
PH
PSPACE
co-NP

Capítulo 13

Algoritmos Probabilistas

Contents

13.1. Clases de Algoritmos Aleatorios : BPP, RP, ZPP 275
13.2. La clase P/poly: circuitos booleanos 278
13.3. Una disgresión : el Sistema Criptográfico RSA. 279
13.3.1. Criptología 279
13.3.1.1. Contexto de la Teoría de la Información : 280
13.3.2. Criptografía y Criptoanálisis. 281
13.3.3. El Sistema Criptográfico RSA. 281
13.3.3.1. El sistema RSA : Definición de Clave Pública : 281
13.3.4. El sistema RSA : Codificación de un mensaje 281
13.3.5. El sistema RSA : Descodificación de un mensaje 281
13.4. Test de Primalidad de Miller-Rabin: COMPUESTO $\in$ RP 282
13.4.1. La búsqueda de primos de talla dada y otras cuestiones. 291
13.4.2. Comentarios sobre Criptoanálisis :Factorización. 293
13.4.2.1. Comentario Preliminar. 293
13.4.2.2. Algoritmos de Factorización basados en Fermat. 293
13.4.2.3. El método ρ de Pollard. 294
13.5. Reciprocidad Cuadrática: El algoritmo de Solovay-Strassen 295
13.5.0.1. Símbolo de Lengendre. Criterio de Euler 295
13.5.1. La demostración de Einsenstein 297
13.5.2. El Lema de Gauss y las raíces cuadradas modulares de 2 299
13.5.3. El símbolo de Jacobi 299
13.5.4. El Test de Solovay-Strassen 301
13.6. Tests de Nulidad para Polinomios. 303
13.6.1. El Test de Schwartz–Zippel. 303
13.6.2. Cuestores. 305
13.6.3. Witness Theorem. 306
13.6.4. Tests de Nulidad para Números Dados por Esquemas de Evaluación. . 307

Esta Capítulo discute algunos aspectos de los algoritmos probabilistas, expresando sus propiedades esenciales e ilustrando a través de ejemplos:

i) Tests Probabilistas de Primalidad.

ii) Tests Probabilistas de Nulidad para polinomios.

13.1. Clases de Algoritmos Aleatorios : BPP, RP, ZPP

La tesis de Cobham–Edmonds, a la que hacíamos referencia en la Subsección 12.1.1, puede y debe extenderse hasta la clase de algoritmos tratables que incorporan un ingrediente de aleatoriedad: los algoritmos probabilistas con tiempo polinomial. En la práctica son los algoritmos más utilizados, tienden a ser más eficientes que los determinísticos conocidos para problemas análogos y su robustez teórica los hace deseables y valiosos.
Históricamente, nacen con los primeros tests de primalidad en los trabajos de Solovay y Strassen (cf. [SoSt, 77]) o Miller y Rabin (cf. [Mi, 76], [Ra, 80]). Depués vinieron los tests de nulidad de polinomios dados por programas que los evalúan (*straight–line program*) como en los trabajos [Schwa, 80], [Zi, 90] (cf. también [Zi, 93]) o en las versiones con conjuntos questores [HeSc, 82] (cf. [Pa, 95] para un histórico del tema). Los tests de primalidad tendrán gran impacto en el diseño de protocolos criptográficos como RSA, mientras que los tests probabilistas de nulidad (para polinomios y números) influirán en el diseño de algoritmos eficientes en Teoría de la Eliminación y en el tratamiento algorítmico del Nullstellensatz, por ejemplo. Más recientemente, los algoritmos probabilistas servirán, por ejemplo, para el tratamiento numérico eficiente de ecuaciones polinomiales multivariadas, lo que permitirá resolver el PROBLEMA 17 de los propuestos por S. Smale para el siglo XXI (cf. [BePa, 11s] para un resumen histórico del tema). Una monografía sobre algoritmos probabilistas o aleatorios es [MoRa, 95]. En las páginas que siguen supondremos la distribución uniforme en $\{0,1\}^n$ o en Σ^n (siguiendo la notación de [Pa, 12]).
A primera vista, los algoritmos probabilistas tienen un aspecto similar a los indeterminísticos: admitimos una etapa de "guessing" sobre un conjunto de elementos de longitud polinomial en el tamaño de la entrada. En el caso probabilista, *disponemos, además, de un control de la probabilidad de cometer errores*. A partir de un polinomio univariado p y de una máquina de Turing determinística N, podemos imaginar un modelo de máquina de la forma siguiente:

INPUT : $x \in \Sigma^*$
Guess at random $y \in \Sigma^*$, $|y| \leq p(|x|)$
Aplicar N sobre $x \cdot y \in \Sigma^*$.
if N acepta $x \cdot y$,
    OUTPUT x es aceptado y $Res_N(x \cdot y)$.
else OUTPUT rechazar x.
fi

Definición 13.1.1 (BPP). Un lenguaje $L \subseteq \Sigma^*$, con función característica $\chi_L : \Sigma^* \longrightarrow \{0,1\}$, pertenece a la clase **BPP**[1] si existe un par (p, N) donde:

- p es un polinomio univariado,
- N es una máquina de Turing determinística de tiempo polinomial,

de tal modo que para cada $x \in \{0,1\}^*$, la probabilidad de error del algoritmo probabilista diseñado en el modelo anterior verifica

$$\texttt{Prob}_{y \in \{0,1\}^{p(|x|)}}[N(x,y) \neq \chi_L(x)] \leq 1/3.$$

Esta clase asume la probabilidad de error a ambos lados, pero son también habituales los algoritmos probabilistas que yerran solamente hacia uno de los lados (éste es el caso, por ejemplo, en los tests de primalidad probabilistas citados anteriormente). Son las clases **RP** y co–**RP** siguientes:

Definición 13.1.2 (RP). Un lenguaje $L \subseteq \Sigma^*$ pertenece a la clase **RP** si existe un par (p, N), donde $p \in \mathbb{Z}[X]$ y N es una máquina determinística de tiempo polinomial, tales que para cada $x \in \{0,1\}^*$, se verifica:

$$\begin{array}{lccl} \textbf{Completitud:} & x \in L & \Longrightarrow & \texttt{Prob}_{y \in \{0,1\}^{p(|x|)}}[N(x,y) = \text{accept}] \geq 2/3, \\ \textbf{Solidez:} & x \notin L & \Longrightarrow & \texttt{Prob}_{y \in \{0,1\}^{p(|x|)}}[N(x,y) = \text{accept}] = 0. \end{array}$$

[1] Bounded error probability polynomial time

La clase co–**RP** es la clase de lenguajes cuyos complementarios están en **RP**.

A modo de ejemplo, el clásico Test de "Primalidad" de Miller–Rabin es un algoritmo que, en realidad, prueba que el lenguaje de los números primos PRIMES $\subseteq \mathbb{N}$ está en co–**RP**: Si el input n es primo, el test de Miller–Rabin devuelve PRIMO con probabilidad 1, mientras que si el input n es compuesto, devuelve PRIMO con probabilidad estrictamente menor que 1/3. Propiamente hablando, el Test de Miller–Rabin es un "Test de Composición ", como también lo es el Test de Solovay–Strassen, por ejemplo.
Estos modelos de algoritmo suelen recibir también el nombre de algoritmos de tipo *Monte Carlo*. Una clase más fina son los algoritmos *Las Vegas*.

Definición 13.1.3 (**ZPP** o Las Vegas)**.** Un lenguaje $L \subseteq \Sigma^*$ pertenece a la clase **ZPP** si existe un par (p, N) con las propiedades anteriores tal que la probabilidad de error es nula y, adicionalmente, para cada $x \in \Sigma^*$, la esperanza de la función de tiempo es polinomial en la talla de x, esto es,

$$E_{y\in\{0,1\}^{p(|x|)}}[T_N(x, y)] \in |x|^{O(1)}.$$

Algunas primeras propiedades (obvias) son las siguientes:

$$\begin{array}{ll} \mathbf{RP} \subseteq \mathbf{BPP}, & \text{co–}\mathbf{RP} \subseteq \mathbf{BPP}, \\ \mathbf{RP} \subseteq \mathbf{NP}, & \text{co–}\mathbf{RP} \subseteq \text{co–}\mathbf{NP}. \end{array}$$

Tiene algún interés la siguiente

Proposición 13.1.1.

$$\mathbf{ZPP} = \mathbf{RP} \bigcap \text{co–}\mathbf{RP}.$$

Demostración. Supongamos que disponemos de una máquina M_1 que resuelve el L en **RP** y otra máquina M_2 que resuelve el mismo lenguaje en co–**RP**. Procedamos como sigue:
INPUT $x \in \Sigma^*$
while No hay respuesta **do**
apply la máquina M_1 sobre x,
if M_1 responde afirmativamente, OUTPUT: **1**
else do
apply la máquina M_2 sobre x,
if M_2 responde negativamente, OUTPUT: **0**
else return to while
od
end
Es claro que esta combinación produce un algoritmo en **ZPP**. Para probar el otro contenido, baste con diseñar una máquina $\widetilde{M}$ modificando la máquina M, que demuestra que un cierto lenguaje L está en **ZPP**, en la forma siguiente: Sobre un input x de talla n, la máquina $\widetilde{M}$ trabaja sobre x en, al menos, el doble de la esperanza $E_{y\in\{0,1\}^{p(n)}}[T_N(x, y)]$. Si la máquina $\widetilde{M}$ llega a aceptar, damos respuesta afirmativa y si no concluye su ejecución antes del doble de la esperanza, respondemos negativamente. La probabilidad de error estará por 1/2, como consecuencia inmediata de la Desigualdad de Markov. □

Una pregunta abierta más en nuestra lista es la siguiente:

Problema Abierto 4. *Con las anteriores notaciones, ¿***BPP** $\subseteq$ **NP***?.*

Un resultado importante, con lo ya expuesto, y en relación con la Jerarquía polinomial discutida en la Subsección 12.5.2 es el siguiente:

Teorema 13.1.2 ([Si, 83], [Lau, 83])**.** **BPP** $\subseteq \Sigma_2 \cap \Pi_2 \subseteq$ **PH**.

13.2. La clase P/poly: circuitos booleanos

Un circuito booleano $\mathcal{C}$ es un grafo orientado y acícliclo cuyos nodos tienen abanico de entrada de cardinal a lo más 2. El grafo define una relación de orden parcial sobre los nodos y están etiquetados de la manera siguiente:

- Los nodos de entrada (abanico de entrada 0) están en biyección con las etiquetas $\{X_1, \ldots, X_n, 0, 1\}$, representando variables y constantes booleanas.
- Los nodos interiores con abanico de entrada 2, contienen una etiqueta de la forma *op*, con $op \in \{\vee, \wedge\}$. Se interpretan como la acción de la conectiva *op* sobre los resultados aportados por los nodos inmediatamente "anteriores" i_1 e i_2.
- Los nodos interiores con abanico de entrada 1, contienen etiquetas de la forma $\neg$, y se interpreta como la acción de $\neg$ sobre el resultado aportado por el nodo i_1 inmediatamente "anterior".
- Hay un único nodo con abanico de salida 0, que devuelve el resultado del circuito.

Los circuitos booleanos son programas finitos que evalúan funciones booleanas $f : \{0,1\}^n \longrightarrow \{0,1\}$. Si en lugar de las conectivas $\{\vee, \wedge, \neg\}$, hubiésemos identificado $\mathbb{F}_2 = \{0,1\}$ y usado las operaciones de $\mathbb{F}_2$ como cuerpo, los hubiéramos llamado circuitos aritméticos. Esto nos llevaría a la *Teoría de la Complejidad Algebraica* y a otra interesantísima historia, que no es la pretendida. La talla del circuito es la *talla del grafo* y la altura (o *profundidad*) se suele identificar con la complejidad paralela (como en la clase **NC** de problemas bien paralelizables, que no trataremos aquí).
Si $\mathcal{B}_n$ denota la clase de funciones booleanas con domino $\mathbb{F}_2^n$, C.E. Shannon y O. Lupanov ([Sha, 49], [Lup, 58]) demostraron que casi todas las funciones booleanas exigen circuitos de talla $2^n/n$ para ser evaluadas y que esa talla basta para evaluar cualquier función booleana. En la interpretación algebraica, $\mathcal{B}_n$ es isomorfo, como $\mathbb{F}_2-$álgebra al anillo de clases de restos $\mathbb{F}_2[X_1, \ldots, X_n]/\mathfrak{a}$, donde $\mathfrak{a}$ es el ideal generado por $\{X_1^2 - X_1, \ldots, X_n^2 - X_n\}$. Un problema abierto clásico es el siguiente:

Problema Abierto 5. *Mostrar una función booleana $\varphi \in \mathcal{B}_n$ que necesite circuitos de talla $2^n/n$ para ser evaluada.*

Solamente nos ocuparemos de la clase **P**/poly definida mediante:

Definición 13.2.1 (**P**/poly)**.** Un lenguaje L, está en la clase **P**/poly si existe un polinomio univariado p y una familia de circuitos booleanos $\{\mathcal{C}_n \ : \ n \in \mathbb{N}\}$ tal que:

- Para cada $n \in \mathbb{N}$, la talla del circuito $\mathcal{C}_n$ está acotada por $p(n)$.
- Para cada $n \in \mathbb{N}$, el circuito $\mathcal{C}_n$ evalúa la función booleana dada como la función característica de $L_n \subseteq \{0,1\}^n$, donde $L_n := \{x \in L \ : \ |x| = n\}$.

Algunos resultados básicos que relacionan **P**/poly con las otras clases son:

Teorema 13.2.1 ([KrpLi, 80])**.** *Con las anteriores notaciones se tiene:*

- **BPP** $\subseteq$ **P**/poly.
- *Si* **NP** $\subseteq$ **P**/poly*, entonces la jerarquía polinomial colapsa* **PH** $= \Sigma_2$.
- *Si* **EXPTIME** $\subseteq$ **P**/poly*, entonces* **EXPTIME** $= \Sigma_2$.
- *Si* **P**=**NP***, entonces* **EXPTIME** $\not\subseteq$ **P**/poly.

13.3. Una disgresión : el Sistema Criptográfico RSA.

13.3.1. Criptología

Parece que la existencia de los números primos ya era conocido por los pitagóricos. En los "Elementos" de Euclides aparece su famosa demostración de la existencia de un número infinito de números primos. Desde entonces, estos números han fascinado a los matemáticos de todas la generaciones. Sería imposible citar aquí a cuantos han hecho aportaciones significativas a la Teoría de los números primos.
Sin embargo, su relevancia se ha visto incrementada notablemente en los últimos años, no tanto por la riqueza de sus propiedades, sino por sus aplicaciones en el contexto de la Teoría de la Información.
El Capítulo se ocupa de los dos elementos esenciales de la Criptología :

- Criptografía : La Criptografía se dedica al desarrollo de técnicas que transforman un texto inteligible en otro (llamado *criptograma*) cuyo contenido de información (Semántica) es igual al del texto inteligible, pero que sólo pueden interpretar personas autorizadas. En esta parte del curso nos dedicaremos esencialmente a la Criptografía.

- Criptoanálisis : El Criptoanálisis, por su parte, se dedica a descrifrar criptogramas sin tener la autorización (romper un sistemas criptográfico).

El contexto de la Criptografía moderna es también el contexto de la Teoría de Números, de los números primos y del diseño y análisis de algoritmos de factorización de números enteros. Desde el año 1978, a partir de la aparición del sistema criptográfico RSA (por sus autores [RSA, 78] su utilización en sistemas de telecomunicación y, especialmente, en Internet se ha extendido al punto de ser la base de las comunicaciones seguras por internet (Criptografía de mensajes y correos electrónicos, el sistema de acceso seguro SSH, etc.). Tres son los ingredientes fundamentales que intervienen en la comprensión del sistema RSA.
En primer lugar, el criptosistema funciona como consecuencia del Teorema Pequeño de Fermat (véase la Subsección 12.2.3). Gracias al Teorema Pequeño de Fermat y a la función de Euler somos capaces de enviar mensajes codificados y de descodificar los mensajes que nos han sido enviados.
El siguiente elemento esencial para el sistema RSA son los Test de Primalidad. Gracias a ellos el usuario puede definir su clave pública, la clave que permite la recepción de mensajes cifrados. Uno de los resultados más impactantes del verano del 2002 (que se publicará como [AKS, 04]) fué el anuncio por parte de los autores M. Agrawal, N. Kayal y N. Saxena [2] de que disponen de un algoritmo polinomial para decidir si un número entero dado es primo o no. La sencillez del método fue uno de los rasgos esenciales. Este resultado relevante, del cual dispongo de un manuscrito de 9 páginas, puede cambiar notablemente la impartición de los tests de primalidad. Este sencillo algoritmo será incluido más adelante en las notas.
Sin embargo, hay razones de tipo histórico que permiten reflexionar sobre la noción de algoritmos probabilistas y que serán tratados con mayor profusión. He preferido mantener un cierto tono clásico al incluir esencialmente los Tests de Primalidad probabilísticos. Los Tests de Primalidad son los primeros algoritmos probabilistas y no deterministas que aparecen en el contexto informático. El primer algoritmo es el debido a R. Solovay y V. Strassen de 1977, que será discutido en la Subsección 13.5.4 (cf. [SoSt, 77]). Es muy común el Test de Miller–Rabin por necesitar menos Teoría de Números para ser establecido, que será discutido en la Sección **??** siguiente. Debe señalarse que estos Test de Primalidad no son exactamente los mejores. De hecho, se trata de algoritmos de tipo **BPP** y **RP** (i.e. Monte Carlo) para el problema de Composición (es decir, son buenos para detectar números compuestos y no números primos). La demostración de que existe un Test de Primalidad **ZPP** (i.e. Las Vegas) puede seguirse en el texto [AdHu, 92]. Expondremos los tests de Miller-Rabin y Solovay-Strassen en este Capítulo, como complementos a las nociones.

[2] M. Agrawal, N. Kayal, N. Saxena. "PRIMES is in P", manuscrito de 6 de Agosto (2002).

El último elemento de los sistemas criptográficos de clave pública es el *criptoanálisis*, esto es, la búsqueda de procedimientos para romper los criptosistemas. En el caso del sistema critográfico RSA, basta con factorizar la clave pública de un individuo para tener acceso a la descodificación de todos los mensajes que recibe. Por tanto, el énfasis se hace en el diseño de algoritmos para la factorización de números enteros. Si existiera un algoritmo que en tiempo polinomial factorizara números enteros, entonces todo el sistema de clave pública sería vulnerable y no se podría utilizar con seguridad. Por ello, la esencia del asunto es que *no se conoce (y, a lo mejor, no puede existir) ningún algoritmo polinomial que factorice números enteros.* Discutiremos este asunto en la Sección 13.4.2 siguiente. Aquí hay que hacer una disquisición sobre la elección de los métodos propuestos. En el comienzo de la Sección 13.4.2 se exhibe una tabla de tiempos que mide el comportamiento de los algoritmos de factorización esenciales. Hemos seleccionado impartir un algoritmo sencillo, cuyo comportamiento se ha establecido de modo "heurístico" y con un excelente (dentro de lo posible) comportamiento experimental. Se trata del algoritmo ρ de J.M. Pollard [Po, 75]. Citando a [GatGe, 99] podemos decir : *" The time bounds of Pollard's ρ method are "heuristic" and have not been rigourously established".*
Los métodos más eficaces (como el método de las curvas elípticas de H.W. Lenstra [3] o el método de la Criba del Cuerpo de Números [4]) aumentarían excesivamente las escuetas disponibilidades de tiempo necesarias para una exposición relativamente honesta de cualquiera de esos dos métodos. Por ejemplo, el método de la curva elíptica de Lenstra exige, entre otras cosas, el estudio de la cota de Hasse para el número de puntos racionales de una curva elíptica sobre un cuerpo finito de cardinal q y característica distinta de 2 y 3 ($|\sharp(\mathcal{E}) - (q+1)| \leq \sqrt{q}$). La demostración con detalle de las estimaciones de probabilidad propuestas por Lenstra es aún más larga en términos de tiempo docente. Como ejemplo, podría hacerse lo que propone el [GatGe, 99] : hacer un "amago" del Teorema de Hasse (sin demostrar), establecer el algoritmo de Lenstra (sin demostrar) y proponer el enunciado sobre la probabilidad de que el algoritmo de Lenstra funciona (sin demostrar) añadiendo una frase amable que dice "the proof is outside the scope of this book". Personalmente, he preferido no incluir nada en este documento.
Las referencias básicas de este Capítulo son diversas. Para los Test de Primalidad hemos seguido el [Kn, 97] y el [Ko, 92]. El algoritmo ρ de Pollard ha sido tomado de [Kn, 97] y [GatGe, 99].

13.3.1.1. Contexto de la Teoría de la Información :

La Teoría de la Información se divide en tres ramas esenciales :

- Criptología : Comprende dos ramas la *Criptografía* y el *Criptoanálisis*. La Criptografía se dedica al desarrollo de técnicas que transforman un texto inteligible en otro (llamado *criptograma*) cuyo contenido de información es igual al del texto inteligible, pero que sólo pueden interpretar personas autorizadas. El Criptoanálisis, por su parte, se dedica a descrifrar criptogramas sin tener la autorización (romper un sistema criptográfico). En esta parte del curso nos dedicaremos esencialmente a la Criptografía.

- Compresión de la Información : Básicamente se trata de estudiar los límites de la compresión de información mediante códigos. Se conoce también como complejidad de Kolmogoroff[5] (o, por ser generosos con los simultáneos descubridores, como complejidad de Kolmogoroff–Chaitin–Solomonoff). Cae un tanto lejos de los objetivos de este curso.

- Teoría de Códigos Correctores de Errores : Fundamentalmente se trata de generar códigos seguros en los que el objetivo no es la protección de la información frente a personas

[3] que fue introducido en H.W. Lenstra. "Factoring integers with elliptic curves". *Annals of Math.* **126** (1987) 649–673.
[4] A.K. Lenstra, H.W. Lenstra, M.S. Manasse, J.M. Pollard. "The Number Field Sieve". In *proc. 22nd ACM SYmp. on Theory of Comput.* (1990) 564–572.
[5] Para mayor información sobre la complejidad de Kolmogoroff, véanse los excelentes textos de Li y Vitanyi M. Li, P. Vitanyi. "Kolmogorov Complexity and its Applications". *Handbook of Theoretical Computer Science*, Ch. 4, Elsevier (1990) 187–251; Li, Ming, Vitanyi, Paul. *" An introduction to Kolmogorov complexity and its applications".* Springer, (1997). También se pueden seguir otras referencias en mi trabajo con J.L. Montaña : J.L. Montaña, L.M. Pardo."On the Kolmogorov Complexity in the Real Turing Machine Setting". *Inf. Proc. Letters.* vol. **67**, 2 (1998) 81–86.

no autorizadas, sino la *calidad* de la información en evitación de errores propios de la transmisión mediante equipamiento electrónico.

13.3.2. Criptografía y Criptoanálisis.

Hay excelentes tratados de Criptografía cuyos contenidos superan con mucho los objetivos propuestos en este curso.

- Sistemas clásicos de Criptografía. Se trata de hacer un resumen superficial de sistemas criptográficos en desuso que sirva para motivar el sistema RSA.
- Sistemas Asimétricos : En 1976, W. Diffie y M.E. Hellman inician la utilización de los criptosistemas de calve pública con su trabajo [DiHe, 76] En él definen los elementos necesarios para disponer de un buen sistema criptográfico de clave pública, en los siguientes términos : *Confidencialidad, Autentificación y Firma Digital.* Hasta la fecha han aparecido varios de estos sistemas, pero el más difundido es el RSA por el trabajo de 1978 en [RSA, 78].Éste es el criptosistema que desarrollaremos en esta parte del curso.

13.3.3. El Sistema Criptográfico RSA.

13.3.3.1. El sistema RSA : Definición de Clave Pública :

Cada individuo de una red de clave pública selecciona los siguientes elementos :

i) Dos números primos **grandes** p y q.

ii) Hallar $n = pq$, $\phi(n) := (p-1)(q-1)$.

iii) Elegir un entero e tal que , $1 < e < \phi(n)$ y $mcd(e, \phi(n)) = 1$. El número e puede ser elegido como el resto de la división de n por $\phi(n)$. Obviamente, el máximo común divisor de n y $\phi(n)$ es 1 en este caso. Para evitar los casos triviales, el número e ha de verificar $2^e > pq$. Ahora, obsérvese que $e = p + q - 1$ y todo irá bien si p y q son suficientemente grandes.

iv) Hallar $d = e^{-1}$ (el inverso de e en $\mathbb{Z}/\phi(n)\mathbb{Z}$).

v) Definir Clave Pública de usuario como **(n,e)**.

13.3.4. El sistema RSA : Codificación de un mensaje

El individuo A (con clave pública (n_A, e_A) quiere enviar un mensaje M al individuo B (con clave pública (n_B, e_B)). Procede como sigue :

- Calcula el criptograma $C := M^{e_B} \ (mod\ n_B)$ y lo expone públicamente. Todo el mundo puede leer el Criptograma C.

13.3.5. El sistema RSA : Descodificación de un mensaje

El individuo B lee el criptograma C emitido por A. Lo descodifica : Procede como sigue :

- Calcula el criptograma $C^{d_B} \ (mod\ n_B) = (M^{e_B})^{d_B} \ (mod\ n_B) = M \ (mod\ n_B)$ y lee el mensaje.

Observación 13.3.1 (Vulnerabilidad de **RSA**)**.** La seguridad del sistema RSA se fundamenta en la carencia de algoritmos eficientes de factorización de números enteros (esto es, algoritmos que funcionan en tiempo polinomial en el tamaño de la entrada). Para romper un sistema criptográfico del tipo RSA basta con aplicar el algoritmo que se muestra en la Figura 3. A partir de ese momento, todo mensaje que se le envíe al individuo "criptoanalizado" será fácilmente descrifrado y, muy probablemente, no sabrá que están leyendo sus mensajes cifrados. Sin embargo, esto no es tan fácil. En estos momentos, claves públicas de 1024 bits (esto es, p y q se escriben con

155 cifras cada uno) son todavía casi invulnerables con la algorítmica actual y las potencias de cálculo disponibles[6] . En la parte del Criptoanálisis de RSA se han venido desarrollando técnicas muy poderosas (con matemáticas que exceden con mucho los contenidos de este curso (Criba del Cuerpo de Números, Curvas Elípticas, etc.) que han rebajado el tiempo de ejecución de algoritmos de factorización de números; aunque sigue sin ser eficiente. Libros de alto nivel, alejado de las posibilidades de un curso como éste, en los que se introducen y demuestran algoritmos de factorización de números enteros más eficaces concidos son [Coh, 93], [Rie, 85], [AdHu, 92], [Kn, 97], [Ko, 92] de la Sección 13.4.2 más adelante.

Observación 13.3.2 (Autentificación Firma Digital.)**.** Cuando un individuo A de clave (n_A, e_A) quiere enviar un mensaje M cifrado y firmado al individuo B (de clave (n_B, e_B)), bastará con que le envía la información :

$$C_1 := M^{d_A} \mod n_A,$$

y

$$C_2 := C_1^{e_B} \mod n_B.$$

Para decodificar el mensaje cifrado y firmado, B debe calcular :

$$C_1 := C_2^{d_B} \mod n_B,$$

y

$$M := C_1^{e_A} \mod n_A.$$

INPUT : La clave Pública (n, e) de un individuo.

Factoriza n **:** Hallar p, q tales que $n = p.q$

Hallar $\phi(n)$ **:** Basta con mutiplicar $(p-1)*(q-1)$.

Hallar la Clave de Descodificación d **:** Basta con aplicar el Algoritmo Extendido de Euclides al par dado por $(p-1)*(q-1)$ y e. Se trata de hallar $d := e^{-1}$ módulo $(p-1)*(q-1)$. esto es fácil de hacer como consecuencia del teorema de G. Lamé.

OUTPUT : d

13.4. Test de Primalidad de Miller-Rabin: COMPUESTO $\in$ RP

Consideraremos el siguiente procedimiento :

INPUT $n \in 2\mathbb{N} + 1$

Compute ℓ such that $n - 1 = 2^\ell m$, m impar.

Guess randomly $a \in \{1, \ldots, n-1\}$

Compute

$$a^m (mod\ n)$$

if $a^m = 1$ (mod n), aceptar y OUTPUT PRIMO

[6]El sistema criptográfico RSA-155 (por 512 bits) se rompiró en Agosto de 1999, con un curioso procedimiento. En Agosto de 1999, aprovechando que todas las máquinas de los principales centros académicos de París estaban infrautilizadas por vacaciones, usando las redes de Cálculo de varios de los centros científicos más importantes de Francia (École Polytechnique, École Normale Superieure, etc.) trabajando en red durante un mes con el algoritmo de factorización de Lenstra, basado en curvas elípticas, se pudo romper RSA–155. Véase un informe en el trabajo J. Marchand *et al.*. "Factorization of a 512-bits RSA modulus". En *Eurocrypt* (2000).

else compute

$$a^m(mod\ \ n), a^{2m}(mod\ \ n), a^{2^2m}(mod\ \ n), \ldots, a^{2^\ell m}(mod\ \ n)$$

if $a^{n-1} \neq 1\ (mod\ \ n)$, aceptar y OUTPUT COMPUESTO.
else encontrar el mayor k such that $a^{2^k m} \neq 1\ (mod\ \ n)$
if $a^{2^k m} = -1\ (mod\ \ n)$, aceptar y OUTPUT PRIMO
else OUTPUT COMPUESTO.
fi
fi
fi
end;

El algoritmo se probó como algoritmo en **RP** en [Mi, 76], bajo la hipótesis de Riemann. Posteriormente, [Ra, 80] demuestra que está en **RP** sin necesidad de la hipótesis de Riemman. Esta segunda es la prueba que vamos a describir aquí.

Teorema 13.4.1 ([Mi, 76], [Ra, 80]). *i) Si $n \in \mathbb{N}$ es un número primo impar, el algoritmo siempre produce PRIMO.*

ii) Si $n \in \mathbb{N}$ es un número compuesto, el algoritmo produce COMPUESTO, con probabilidad $\geq 1/2$.

Demostración. La demostración de este resultado requerirá varias etapas. Comencemos con la más simple :

Lema 13.4.2. *Si $n \in \mathbb{N}$ es un número primo impar, el anterior procedimiento siempre responde PRIMO.*

Demostración. Dado $a \in \{2, \ldots, n-1\}$, necesariamente ha de ocurrir

$$a^{n-1} = 1\ (mod\ n)$$

Dada una factorización

$$n - 1 = 2^\ell m,$$

donde m es un número impar, tomemos la secuencia:

$$a^m(mod\ \ n), a^{2m}(mod\ \ n), a^{2^2m}(mod\ \ n), \ldots, a^{2^\ell m}(mod\ \ n)$$

Puesto que n es un número primo, la ecuación $T^2 - 1$ se factoriza de modo único en el D.I.P.

$$\mathbb{Z}/n\mathbb{Z}[T]$$

del modo obvio :

$$T^2 - 1 = (T + 1)(T - 1)$$

Si k es el máximo tal que $a^{2^k m} \neq 1\ (mod\, n)$, necesariamente :

$$a^{2^{k+1} m} = 1\ (mod\, n).$$

Por tanto,

$$a^{2^k m} = -1\ (mod\ n)$$

con lo que nuestro algoritmo siempre responderá PRIMO. □

Para poder demostrar la segunda de las afirmaciones, necesitaremos un poco más de esfuerzo. Recuperemos las notaciones utilizadas en la demostración del teorema de Pratt.
Para $n \in \mathbb{N}$, sea $1 \leq d \leq n - 1$ y definamos el grupo de las raíces $d-$ésimas de la unidad :

$$K(d) := \{a \in \{1, \ldots, n-1\}\ :\ a^d - 1 = 0(mod\ n)\}$$

Lema 13.4.3. *Para cada $n \in \mathbb{N}$, $K(d)$ es un subgrupo del grupo de las unidades $(\mathbb{Z}/n\mathbb{Z})^*$. En particular,*

$$\sharp(K(n-1)) \mid \sharp(\mathbb{Z}/n\mathbb{Z})^*$$

luego,

$$\sharp(K(n-1)) \mid \varphi(n)$$

La demostración es, por demás, obvia. □

Lema 13.4.4. *Sea $n \in N$ un número compuesto que no es un número de Carmichael. El algoritmo de Miller–Rabin responder COMPUESTO con probabilidad $> 1/2$.*

Demostración. Puesto que n es un número compuesto impar, que no es un número de Carmichael, ha de ocurrir :

$$K(n-1) \quad \text{es un subconjunto propio de} \quad (\mathbb{Z}/n\mathbb{Z})^*$$

Como también es un subgrupo, se tiene :

$$\sharp(K(n-1)) \leq \frac{1}{2}\sharp(\mathbb{Z}/n\mathbb{Z})^* = \frac{1}{2}\varphi(n)$$

Adicionalmente, como n es compuesto, $\varphi(n) < n-1$.
El algoritmo responde PRIMO solamente en el caso en que encuentre $a \in \{1, \ldots, n-1\}$ tal que :

- O bien $a^m = 1 \ (mod\ n)$,
- o bien $a^{n-1} - 1 = 0 \ (mod\ n)$ y existe k tal que

$$a^{2^k m} \neq 1 \ (mod\ n) \quad y \quad a^{2^k m} = -1 \ (mod\ n)$$

Por lo tanto, responde PRIMO, solamente en los casos en los que $a \in K(n-1)$. La probabilidad de que responda PRIMO estar acotada por :

$$\frac{\sharp(K(n-1))}{\sharp\{1, \ldots, n-1\}} \leq \frac{1}{2}\frac{\varphi(n)}{(n-1)} < \frac{1}{2}\frac{(n-1)}{(n-1)} = \frac{1}{2}.$$

□

Nos queda por ver el caso en el que tengamos un número compuesto que sea número de Carmichael.

Lema 13.4.5. *Las potencias de primos no son números de Carmichael.*

Demostración. Sea $n := p^d$, con p un número primo y $a := p^{d-1} + 1$.
Como $p^{d-1} + 1$ y p^d son coprimos,

$$a \in (\mathbb{Z}/n\mathbb{Z})^*.$$

De otro lado :

$$a^{n-1} = (p^{d-1} + 1)^{p^d - 1} = \sum_{i=0}^{p^d - 1} \binom{p^d - 1}{i} p^{(d-1)i}$$

Para cada $i \geq 2$, $p^{(d-1)i}$ es divisible por p^d, luego

$$a^{n-1} = 1 + (p^d - 1)p^{d-1} \pmod{p^d} = 1 + p^{d-1} \pmod{p^d} \neq 1 \pmod{p^d}$$

Luego, n no es un número de Carmichael. □

Lema 13.4.6. *Si p es un número primo impar, el anillo*

$$\mathbb{Z}/p^d\mathbb{Z}$$

tiene exactamente dos raíces cuadradas de la unidad. Son exactamnente las clases módulo p^d de los enteros $\{1, -1\}$.

Demostración. Si $a^2 = 1 \pmod{p^d}$:

$$p^d \mid (a+1)(a-1) \Longrightarrow p \mid (a+1) \quad \vee \quad p \mid (a-1)$$

Dado que $p \neq 2$, no puede ser que p divida a ambas cantidades (porque

$$(a+1) = p\alpha \wedge (a-1) = p\beta \Longrightarrow (a+1) - (a-1) = 2 = (\alpha - \beta)p)$$

Por lo tanto p sólo puede ser divisor de una de esas cantidades. Inductivamente, concluiremos

$$p^d \mid (a+1) \vee p^d \mid (a-1)$$

de donde concluimos

$$a = 1 \pmod{p^d} \vee a = -1 \pmod{p^d}$$

□

Lema 13.4.7. *Si $c, d \in \mathbb{N}$ son coprimos $c, d \leq n-1$, se tiene :*

$$K(cd) \cong K(c) \times K(d)$$

Demostración. Definamos

$$\psi \ : K(cd) \longrightarrow K(c) \times K(d)$$

$$a \longmapsto (a^d, a^c)$$

Como $gcd(c, d) = 1$ sean

$$\alpha d + \beta c = 1$$

Luego,

$$a^{\alpha d} a^{\beta c} = a^1 = a$$

La inversa ψ^{-1} vendrá dada por :

$$K(c) \times K(d) \longrightarrow K(cd)$$

$$(x, y) \longmapsto x^\alpha y^\beta$$

Se tiene :

$$(x^\alpha y^\beta)^{cd} = 1 \pmod{n}$$

Para la composición :

$$a \in K(cd) \longmapsto (a^d, a^c) \longmapsto a^{\alpha d} a^{\beta d} = a$$

$$(x, y) \in K(c) \times K(d) \longmapsto x^\alpha y^\beta \longmapsto \left((x^\alpha y^\beta)^d, (x^\alpha y^\beta)^c\right) = (x^{\alpha d}, y^{\beta c}) = (x, y)$$

□

Lema 13.4.8. *Sea $n \in \mathbb{N}$ un número compuesto, supongamos $n = p_1^{\alpha_1} \cdots p_k^{\alpha_k}$, con $p_1, \ldots, p_k$ números primos distintos. Entonces*

$$\mathbb{Z}/n\mathbb{Z} \cong (\mathbb{Z}/p_1^{\alpha_1}\mathbb{Z}) \times \cdots \times (\mathbb{Z}/p_k^{\alpha_k}\mathbb{Z})$$

es un isomorfismo de anillos. Además, el número de raíces cuadradas de la unidad en $\mathbb{Z}/n\mathbb{Z}$ es exactamente 2^k .

Demostración. La primera afirmación no necesita prueba. En cuanto a la segunda, baste ver cmo, a través del isomorfimso del Teorema Chino de los Restos, se identifican las raíces cuadradas de la unidad de $\mathbb{Z}/n\mathbb{Z}$ con las listas

$$(\varepsilon_1 \ mod \ p_1^{\alpha_1}, \ldots, \varepsilon_k \ mod \ p_k^{\alpha_k})$$

donde $\varepsilon_i \in \{1, -1\}$, $1 \leq i \leq k$. □

Observación 13.4.1. Obsérvese que los números de Carmichael se distinguen de los números primos en cuanto la unidad posee dos raíces cuadradas módulo n si y solamente si el número n es primo. Esto es precisamente lo que se mide en el test de Solovay y Strassen mediante el número de Jacobi.

13.4.0.0.1. El grafo $K(2^\ell)$. Para cada número natural impar se tiene :

$$K(n-1) \cong K(2^\ell) \times K(m)$$

con ℓ maximal. Vamos a construir una estructura de grafo en $K(2^\ell)$.
La clase de vectores en el grafo será la siguiente :

$$E := \{(a^2, a) \ : \ a \in K(2^\ell)\}$$

Esto genera una estructura del tipo siguiente :

$$\begin{array}{cccc} & \{1\} & & \\ & \swarrow\downarrow\searrow & & \\ \{-1\} & \{w\} & \{-w\} & \cdots \\ \vdots & \vdots & \vdots & \end{array}$$

Nótese que los elementos de profundidad i dentro del grafo son, justamente, las raíces primitivas $i-$ésimas de la unidad.
Para cada elemento $x \in K(2^\ell)$, denotamos por T_x el árbol formado por sus sucesores, es decir, por sus raíces cuadradas y las raíces cuadradas de sus raíces cuadradas, etc. Formalmente,

$$T_x := \{a \in K(2^\ell) \ : \ \exists k, \ a^{2^k} = x \ mod \ n\}.$$

La observación crucial para el algoritmo Miller–Rabin es la siguiente :

Proposición 13.4.9. *Con las anteriores notaciones :*

i)

$$K(2^\ell) = T_1 = \{1\} \bigcup T_{-1} \bigcup \left(\bigcup_{\omega^2 = 1, \omega \neq -1} T_\omega \right).$$

ii) Dados $\omega, \omega' \in K(2^\ell) \setminus \{1\}$, tales que $\omega^2 = 1 \ mod \ n$, $(\omega')^2 = 1 \ mod \ n$ y $\omega \neq \omega'$, entonces,

$$T_\omega \bigcap T_{\omega'} = \emptyset.$$

En particular,

$$\sharp(K(2^\ell)) = 1 + \sharp(T_{-1}) + \sum_{\omega^2 = 1, \omega \neq -1} \sharp(T_\omega).$$

Demostración. La primera de las afirmaciones es evidente. En cuanto a la segunda, la demostraremos como sigue :
Sean, por tanto, dadas dos raíces cuadradas de la unidad distintas de 1 : $\omega, \omega' \in K(2^\ell) \setminus \{1\}$, tales que $\omega^2 = 1 \ mod \ n$, $(\omega')^2 = 1 \ mod \ n$ y $\omega \neq \omega'$. Supongamos $a \in T_\omega \cap T_{\omega'} \neq \emptyset$. Entonces, existen $k, r \in \mathbb{N}$ tales que

$$\left[a^{2^k} = \omega \ mod \ n\right] \ y \ \left[a^{2^r} = \omega' \ mod \ n\right].$$

Supongamos $k > r$. Entonces,

$$a^{2^k} \ mod \ n = \left(a^{2^r}\right)^{2^{k-r}} \ mod \ n = (\omega')^{2^{k-r}} \ mod \ n.$$

Como $k > r$, se ha de tener, $2 \mid 2^{k-r}$ y, por tanto,

$$a^{2^k} \ mod \ n = \left((\omega')^2\right)^{2^{k-r-1}} \ mod \ n = 1 \ mod \ n.$$

De otro lado, teníamos $a^{2^k} = \omega \ mod \ n$ y $\omega \neq 1 \ mod \ n$, con lo que habremos llegado a contradicción. En particular, debe ocurrir que $k = r$ y se tiene :

$$\omega = a^{2^k} \ mod \ n = a^{2^r} \ mod \ n = \omega',$$

con lo que concluimos una nueva contradicción y queda probado que $T_\omega \cap T_{\omega'} = \emptyset$. □
Adicionalmente, los cardinales de los árboles T_ω están relacionados entre sí. Es lo que indica la siguiente Proposición s

Proposición 13.4.10. *Sea $w \in K(2^\ell)$, y supongamos que $w \neq 1, -1$, con $w^2 = 1 \ (mod \ n)$. Entonces,*

$$\sharp(T_w) \geq \sharp(T_{-1})$$

Demostración. Para demostrar esta Proposición, nos proponemos una serie de etapas intermedias. En primer lugar, definiremos la profundidad de un elemento $x \in T_\omega$ del modo siguiente : Sea ω tal que $\omega^2 = 1 \ mond \ n$ y supongamos $\omega \neq 1$. Denotaremos por $\ell(x)$ al mínimo número natural k tal que

$$x^{2^k} = \omega \ mod \ n.$$

En otras palabras,

$$\ell(x) = k \Longleftrightarrow \left[x^{2^k} = \omega \ mod \ n\right] \ \wedge \ \left[x^{2^j} \neq \omega \ mod \ n, \ 0 \leq j \leq k-1\right].$$

Sea $\omega \in K(2^\ell)$ una raíz cuadrada de la unidad distinta de la unidad (i.e. $\omega^2 = 1 \ mod \ n$ y $\omega \neq 1$). Definamos el conjunto :

$$T_{\omega,k} := \{a \in (\mathbb{Z}/n\mathbb{Z})^* \ : \ a^{2^k} = x \ mod \ n, \ell(a) = k\}.$$

Lema 13.4.11. *Con las anteriores notaciones, la siguiente es una descomposición como unión disjunta de T_ω :*

$$T_\omega := \bigcup_{k=0} T_{\omega,k}.$$

En particular,

$$\sharp(T_\omega) = \sum_{k=0} \sharp(T_{\omega,k}).$$

Demostración. Claramente por la definición de $\ell(x)$ se tiene que es una descomposición como unión disjunta y hemos terminado. □
Para cada $\omega \in K(2^\ell)$ una raíz cuadrada de la unidad distinta de la unidad (i.e. $\omega^2 = 1 \ mod \ n$ y $\omega \neq 1$), definamos :

$$L(\omega) := max \ \{k \ : \ T_{\omega,k} \neq \emptyset\}.$$

En otras palabras, $L(\omega) = k$ si y solamente si existe $x \in T_\omega$ con $\ell(x) = k$ y no hay ningún $y \in T_\omega$ con $\ell(\omega) > k$. En particular, tenemos :

Corollario 13.4.12. *Para cada $\omega \in K(2^\ell)$ una raíz cuadrada de la unidad distinta de la unidad (i.e. $\omega^2 = 1 \ mod \ n$ y $\omega \neq 1$), se tiene :*

$$\sharp(T_\omega) = \sum_{k=0}^{L(\omega)} \sharp(T_{\omega,k}).$$

La Proposición 13.4.10 se sigue del siguiente Lema, junto con el Corolario anterior :

Lema 13.4.13. *Con las anteriores notaciones, se tiene :*

i) Para cada raíz cuadrada de la unidad $\omega^2 = 1$, con $\omega \neq 1$ t para cada $k \leq L(\omega)$, si $T_{\omega,k} \neq \emptyset$, se tiene :

$$\sharp(T_{\omega,k}) = \sharp(K(2^k)).$$

ii) Con las notaciones del apartado i), se tiene que si $T_{\omega,k} \neq \emptyset$, para algún k, entonces, $T_{\omega,j} \neq \emptyset$, para todo $j < k$.

iii) Finalmente, $L(-1) \leq L(\omega)$ para todo ω tal que $\omega^2 = 1 \ mod \ n$, $\omega \neq 1, -1 \ mod \ n$.

En particular, concluimos la afirmación de la Proposición 13.4.10 anterior, es decir

$$\sharp(T_{-1}) := \sum_{k=0}^{L(-1)} \sharp(T_{-1,k}) = \sum_{k=1}^{L(-1)} \sharp(T_{\omega,k}) \leq \sum_{k=1}^{L(\omega)} \sharp(T_{\omega,k}) = \sharp(T_\omega).$$

Demostración. Probemos primero la afirmación *i*). Para ello, sea $c \in T_{\omega,k} \neq \emptyset$. Definamos la siguiente aplicación :

$$\psi \ : K(2^k) \longrightarrow T_{\omega,k}$$

dada mediante

$$\psi(x) := cx, \ \forall x \in K(2^k).$$

Veamos que está bien definida. Claramente se tiene :

$$\psi(x)^{2^k} = (xc)^{2^k} = x^{2^k} c^{2^k} = \omega \ mod \ n.$$

En particular $\psi(K(2^k)) \subseteq T_\omega$. Además, para $j < k$, si $(xc)^{2^j} = \omega \ mod \ n$, se tiene :

$$\psi(x)^{2^k} = \left((xc)^{2^j}\right)^{2^{k-j}} = (\omega)^{2^{k-j}} \ mod \ n = 1 \ mod \ n,$$

por ser $\omega^2 = 1 \ mod \ n$ y ser $2 \mid 2^{k-j}$. Pero $\omega \neq 1 \ mod \ n$, y $\psi(x) \in T_\omega$, con lo que conluimos $\psi(x) \in T_{\omega,k}$ y ψ está bien definida.
Claramente, por ser c una unidad en $(\mathbb{Z}/n\mathbb{Z})^*$, la aplicación ψ es inyectiva.
Para concluir la biyección, nótese que si $y \in T_{\omega,k}$, entonces $c^{-1}y \in K(2^k)$. Esto es,

$$\left(c^{-1}y\right)^{2^k} = \omega^{-1}\omega \ mod \ n = 1 \ mod \ n.$$

Por tanto, ψ es una biyección entre $K(2^k)$ y $T_{\omega,k}$ con lo que tenemos la igualdad de cardinales en este caso.
Parala afirmación *ii*), se tiene que si $c \in T_{\omega,k}$, entonces, $c^{2^j} \in T_{\omega,k-j}$ y la afirmación se sigue de modo inmediato.
La afirmación *iii*) requiere del siguiente argumento : comencemos observando que n es un número de Carmichael impar, luego :

$$n := p_1^{\alpha_1} \cdots p_k^{\alpha_k},$$

donde p_i y p_j son primos distintos. Por el teorema de los Restos Chinos, tenemos un isomorfismo de anillos, del tipo :

$$\mathbb{Z}/n\mathbb{Z} \longrightarrow \mathbb{Z}/p_1^{\alpha_1}\mathbb{Z} \times \cdots \times \mathbb{Z}/p_k^{\alpha_k}\mathbb{Z}$$
$$a \ mod \ n \longmapsto (a \ mod \ p_1^{\alpha_1}, \ldots, a \ mod \ p_k^{\alpha_k})$$

En particular, las raices cuadradas de la unidad en $(\mathbb{Z}/n\mathbb{Z})$ son exactamente 2^k y vienen dadas por listas del tipo :

$$(\varepsilon_1 \ mod \ p_1^{\alpha_1}, \ldots, \varepsilon_k \ mod \ p_k^{\alpha_k})$$

donde $\varepsilon_i \in \{1, -1\}$. Ahora, supongamos que $\omega \in (\mathbb{Z}/n\mathbb{Z})^*$ es una raíz cuadrada de la unidad y tiene la pinta :

$$\omega := (\varepsilon_1 \ mod \ p_1^{\alpha_1}, \ldots, \varepsilon_k \ mod \ p_k^{\alpha_k}).$$

Sea ahora $a = (a_1 \ mod \ p_1^{\alpha_1}, \ldots, a_k \ mod \ p_k^{\alpha_k}) \in \mathbb{Z}/n\mathbb{Z}$ un elemento cualquiera. Definamos :

$$a_i^{(\varepsilon_i)} = a_i^2 \ mod \ p_i^{\alpha_i}, \ \text{ si } \varepsilon_i = 1$$
$$a_i^{(\varepsilon_i)} = a_i \ mod \ p_i^{\alpha_i}, \ \text{ si } \varepsilon_i = -1.$$

Definamos

$$a^{(\omega)} = (a_1^{(\varepsilon_1)}, \ldots, a_k^{(\varepsilon_k)}) \in \mathbb{Z}/n\mathbb{Z}$$

Supongamos que $a \in T_{-1,k}$, entonces

$$(a^{(\omega)})^{2^k} = \omega \ mod \ n,$$

con lo que concluimos que $a \in T_\omega$.
Más aún, supongamos que existe $j < k$ tal que

$$(a^{(\omega)})^{2^j} = \omega \ mod \ n.$$

Si así fuera, concluiríamos :

$$\omega \ mod \ n = (a^{(\omega)})^{2^k} = \left((a^{(w)})^{2^j}\right)^{2^{k-j}} = \omega^{2^{k-j}} = 1 \ mod \ n.$$

Dado que $\omega \neq 1 \ mod \ n$, llegaríamos a una contradicción.
Por tanto, hemos definido una aplicación :

$$\psi \ : T_{-1,k} \longrightarrow T_{\omega,k}$$
$$a \longmapsto a^{(\omega)}.$$

En particular, si $T_{-1,k} \neq \emptyset$, entonces $T_{\omega,k} \neq \emptyset$ y, por tanto,

$$L(-1) \leq L(\omega),$$

como se pretendía. La conclusión de Lema se sigue de las tres afirmaciones y, adicionalmente, la Proposición 13.4.10. □

Lema 13.4.14 (Final)**.** *Si $n \in \mathbb{N}$ es un número de Carmichael impar, que no es un número primo, el algoritmo de Miller–Rabin responde PRIMO con probabilidad menor estricto que* $1/2$.

Demostración. Sea $a \in \{1, \ldots, n-1\}$. Si $a \notin (\mathbb{Z}/n\mathbb{Z})^*$ es seguro que este algoritmo no responde PRIMO.
Además,

$$A := \{1, \ldots, n-1\} \setminus (\mathbb{Z}/n\mathbb{Z})^* \neq \emptyset$$

por ser n un número compuesto.Sea

$$t := \frac{\sharp(A)}{n-1} > 0$$

Revisando el algoritmo, la respuesta puede ser PRIMO, solamente cuando $a \in (\mathbb{Z}/n\mathbb{Z})^*$. Dado que n es un número de Carmichael, tenemos :

$$(\mathbb{Z}/n\mathbb{Z})^* = K(n-1) \cong K(2^\ell) \times K(m)$$

Recordemos, además, el isomorfismo :

$$K(n-1) \longrightarrow K(2^\ell) \times K(m)$$

$$a \longmapsto (a^m, a^{2^\ell})$$

En tal caso, el algoritmo responde PRIMO solamente si

$$a^m = 1 \ (mod\ n)$$

o si

$$\exists k, (a^m)^{2^k} = -1 \ (mod\ N)$$

En otros términos, responde PRIMO, solamente si

$$a \in K(m) \quad \text{o} \quad a^m \in T_{-1}$$

Por lo tanto, tenemos la siguiente estimación para la probabilidad de error :

$$\frac{\sharp\{a \in (\mathbb{Z}/n\mathbb{Z})^* \ : \ a \in K(m) \text{ o } a^m \in T_{-1}\}}{(n-1)} \leq$$

$$< \frac{\sharp\{a \in (\mathbb{Z}/n\mathbb{Z})^* \ : \ a \in K(m) \text{ o } a^m \in T_{-1}\}}{\sharp(K(n-1))} =$$

$$= \frac{\sharp\{a \in (\mathbb{Z}/n\mathbb{Z})^* \ : \ a \in K(m) \text{ o } a^m \in T_{-1}\}}{\sharp(K(m))\sharp(K(2^\ell))} \leq$$

$$\leq \frac{\sharp(K(m)) + \sharp\{a \in K(n-1) \ : \ a^{2^\ell} \in K(m) \wedge a^m \in T_{-1}\}}{\sharp(K(m))\sharp(K(2^\ell))} =$$

Si $\psi\ ; K(n-1) \longrightarrow K(2^\ell) \times K(m)$ es el isomorfismo de grupos dado mediante :

$$\psi(a) := (a^m, a^{2^\ell}),$$

tendremos

$$\psi(\{a \in K(n-1) \ : \ a^{2^\ell} \in K(m) \wedge a^m \in T_{-1}\}) := K(m) \times T_{-1}.$$

Por lo que deducimos que la probabilidad de error está acotada por :

$$\frac{\sharp(K(m)) + \sharp(K(m)) \times \sharp(T_{-1})}{\sharp(K(m)) \times \sharp(K(2^\ell))} = \frac{1 + \sharp(T_{-1})}{\sharp(K(2^\ell))}.$$

Ahora, como n es un número de Carmichael, el número de raíces cuadradas de la unidad es 2^r donde r es el número de factores primos distintos.
Adicionalmente, por la Proposición 13.4.9 tendremos :

$$\sharp(K(2^\ell)) = 1 + \sharp(T_{-1}) + \sum_{\omega^2=1, \omega \neq 1,-1} \sharp(T_\omega).$$

Por la Proposición 13.4.10, tendremos :

$$\sharp(K(2^\ell)) \geq 1 + \sharp(T_{-1}) + 2^{r-1}\sharp(T_{-1}).$$

Como n es de Carmichael, $2^{r-1} \geq 2$, de donde concluimos :

$$\sharp(K(2^\ell)) \geq 1 + 3\sharp(T_{-1}).$$

Por tanto, la probabilidad de error será, a lo sumo :

$$< \frac{1 + \sharp(T_{-1})}{1 + 3\sharp(T_{-1})} = \frac{1+b}{1+3b} \leq \frac{1}{2},$$

para $b \geq 1$. Lo que nos lleva a la conclusión del enunciado. □

13.4.1. La búsqueda de primos de talla dada y otras cuestiones.

Comencemos con un somero recordatorio del Teorema de los Números Primos (véase, por ejemplo, el excelente texto clásico [HarWri, 60]). Este Teorema fue postulado tanto por Adrien–Marie Legendre[7] como por C.F. Gauss[8] a base de evidencia numérica. Una indicación de la posible demostración de este resultado es debida a Chebyshev en 1848. Esta indicación fue seguida por B. Riemann en 1851, quien ofreció un esquema de demostración que quedó incompleto porque el Análisis Complejo de la época aún no estaba preparado para demostrar el resultado. Finalemente, el Teorema de los Números Primos fue demostrado por Jacques Hadamard[9] y simultánea e independientemente por Charles de la Vallée Poussin[10]. Se trata de unos de los grandes resultados en Teoría de Números. Una demostración puede verse casi en cualquier texto básico de Teoría de Números o de Variable Compleja. Una referencia concreta puede ser el texto de H. E. Rose ([Ro, 94], Capítulos 12 y 13).
Comencemos con un poco de notación :

Notación 13.4.1.

- Dadas dos funciones $f, g : \mathbb{R} \longrightarrow \mathbb{R}$, escribiremos $f \in \theta(g)$ cuando

$$lim_{x\to\infty}\frac{f(x)}{g(x)} = 1$$

  Obsérvese que $f \in \theta(g) \Longleftrightarrow g \in \theta(f)$.

- Definamos la función $\pi : \mathbb{N} \longrightarrow \mathbb{N}$ dada por $\pi(x)$ es el cardinal del conjunto de números primos p tales que $2 \leq p \leq x$.

- Definamos la función $Li(x)$ mediante la siguiente integral :

$$Li(x) := \int_2^x \frac{dt}{lnt}$$

Teorema 13.4.15 (Teorema de los Números Primos). *En las anteriores notaciones,*

$$\pi \in \theta\left(\frac{x}{lnx}\right)$$

Observación 13.4.2. La estimación más precisa (experimentalmente debida a Gauss), nos diría : $\pi \in \theta(Li(x))$

Estamos en condiciones de hacer una disquisición del asunto que nos interesa (cf. también [Mign, 89], p.309).

---

[7] Legendre tuvo en Gauss a su gran competidor científico. Por ejemplo, Legendre descubrió el método de mínimos cuadrados, pero debió asignárselo a Gauss porque Gauss afirmó haberlo descubierto antes. Varios otros aspectos en Teoría de Números de la obra de Legendre tienen este mismo problema. Su libro más relevante en Teoría de Números es su texto *"Essai su la Théorie des nombres"* cuya primera edición data de 1798. En la edición de 1808, Legendre incluyó su estimación (hecha a base de cálculos extensos y sin demostración) $\pi(n) = \frac{n}{log(n)} - 1{,}08366$.

[8] Parece que Gauss postuló este famoso Teorema cuando era un niño de unos 11 o 12 años. A la sazón, Gauss (que era un gran calculador) disponía de un texto con una colección de números primos "pequeños" (menores que 1.000). Se puso a contar el número de primos que salían en aquella tabla y le dijo a un amigo que sólo dedicaría quince minutos en aquella niñería. Durante el resto de su vida llegó a calular todos los primos menores que 3.000.000 y observó que el comportamiento de la función $\pi(n)$ era muy similar a la obtenida años después por Hadamard o de la Vallée Poussin. La estimación de Gauss es la función $Li(x)$ que se expone más arriba.

[9] En 1892, Hadamard leyó su tesis Doctoral y ese mismo año ganó el "Grand Prix des Sciences Mathématiques" por su trabajo sobre la "Determination du nombre des nombres premiers plus petits qu'un nombre donné" en el cual resolvía algunos aspectos (demostraciones incompletas) del trabajo de B. Riemann sobre la función zeta. Posteriormente, en 1896, simultáneo con de la Vallée Poussin, obtuvo el enunciado sobre el número de números primos.

[10] Aunque su interés era la fundamentación del Análisis Complejo, su resultado más famoso fué este Teorema de los Números Primos que obtuvo simultánea e independientemente de Hadamard en 1896.

Lema 13.4.16 ([IbMo, 83]). *Existe una constante $c \in \mathbb{R}$ universal, tal que para cualquier un número natural $n \in \mathbb{N}$, $2^{2n} \geq c$ y para cada $k \in \mathbb{N}$ tal que $1 \leq \mid k \mid \leq 2^{n2^n}$ se verifica : Dado el conjunto $\mathcal{P}$ de todos los números primos menores que 2^{2n}. Entonces, existe un primo $p \in \mathcal{P}$ tal que k no es divisible por p.*

Demostración. Por el Teorema de los Números Primos, se tiene $\pi \in \theta(x/lnx)$. Por lo tanto, existe $c \in \mathbb{R}$ tal que para cada $n \in \mathbb{N}$ tal que 2^{2n},

$$\sharp(\mathcal{P}) = \pi(2^{2n}) \geq \frac{2^{2n}}{ln2^{2n}} \geq \frac{2^{2n}}{2n}$$

De otro lado, sean $p_1, \ldots, p_s$ los divisores primos de k. Como $p_1 \geq 2$ se tiene :

$$s \leq log_2 k \leq 2n2^n$$

Finalmente, observemos que

$$\frac{2^{2n}}{2n} - 2n2^n = 2^n \left(\frac{2^n}{2n} - 2n \right) = 2^n \left(\frac{2^n - 4n^2}{2n} \right) > 0$$

para $n \geq 3$. El resto es obvio. □

Observación 13.4.3. Digamos que el argumento de Ibarra y Moran en [IbMo, 83] trata de obtener un resultado del tipo siguiente (que será utilizado más adelante) :
En las mismas hipótesis del Lema anterior, dado el conjunto $\{1, \ldots, 2^{2n}\}$ la probabilidad de que un cierto elemento m (en ese conjunto) elegido al azar no divida a k es mayor o igual que $\frac{1}{4n}$.

Corollario 13.4.17. *Existe un algoritmo polinomial que realiza el siguiente proceso :*
Dado $k \in \mathbb{Z}$, hallar $p \in \mathbb{N}$ un número primo tal que p no divide a k.

Demostración. Consideremos $n := log_2 log_2 \mid k \mid$. Entonces, $\mid k \mid \leq 2^{2n2^n}$. Se trata de hallar todos los primos de la clase $\mathcal{P}$ del Lema anterior. Para ello, obsérvese que el cardinal de $\mathcal{P}$ es, como mucho, 2^{2n}, luego es polinomial en la talla bit de k. Además, cualquier primo en $\mathcal{P}$ tiene talla bit $O(n)$, luego aplicando uno cualquiera de los algoritmos al uso (Eratóstenes, p.ej.) de tipo determinístico, uno puede describir $\mathcal{P}$ en tiempo polinomial en latalla bit de k. Finalmente, calcular el resto de la división de k por cada $p \in \mathcal{P}$ es polinomial en la talla bit de k y no haya nada más que discutir. □

Observación 13.4.4. El proceso se mejora sustancialmente si uno está dispuesto a calcular $\mathcal{P}$ usando algoritmos probablilísticos (tipo Solovay-Strassen [SoSt, 77], Miller–Rabin [Mi, 76], [Ra, 80] ver tambien [BuMü, 95] o, sobre todo, [BaSh, 96]) o algoritmos deterministas tipo AKS.

Un argumento similar es el siguiente :

Corollario 13.4.18. *Existe una algoritmo probabilista polinomial que realiza la tarea siguiente :*
Dado $k \in \mathbb{N}$, hallar algún número primo entre 2^k y $2^{k+1} - 1$ (ambos inclusive)

Demostración. De nuevo usaremos la filosofía probabilista y el Teorema de los Números Primos anterior. Tenemos la acotación siguiente : el cardinal de los números primos entre 2^k y $2^{k+1} - 1$ está acotado por :

$$\pi(2^{k+1}) - \pi(2^k) \geq \frac{2^{k+1}}{k+1} - \frac{2^k}{k} = 2^k \left(\frac{2}{k+1} - \frac{1}{k} \right) = 2^k \left(\frac{k-1}{(k+1)k} \right)$$

La probabilidad de que un tal número elegido al azar sea primo está acotada por :

$$\left(\frac{k-1}{(k+1)k} \right)$$

La probabilidad de que tras un intento no hayamos dado con un primo es como mucho

$$1 - \left(\frac{k-1}{(k+1)k} \right) = 1 - \left(\frac{1}{\frac{(k+1)k}{k-1}} \right)$$

Si tras $(k+1)k$ intentos seguimos sin dar con un número primo, nuestra probabilidad resulta del orden :

$$\left(1 - (\frac{1}{\frac{(k+1)k}{k-1}}) \right)^{(k+1)k} = \left(1 - (\frac{1}{\frac{(k+1)k}{k-1}})^{\frac{(k+1)k}{k-1}} \right)^{k-1}$$

Como la función $(1 - \frac{1}{x})^x$ es una función decreciente para $x > 1$ con la cota (**sic**) :

$$(1 - \frac{1}{x})^x \leq \frac{1}{e}$$

Luego la tal probabilidad anterior es del orden

$$\frac{1}{e^{k-1}}$$

Basta con repetir el algrotimo obvio con un test de primalidad para alcanzar un primo con alta probablidad. Nótese que, en este caso, la cosa es polinomial en k, pero la dependencia de la probablidad es alta. □

13.4.2. Comentarios sobre Criptoanálisis :Factorización.

13.4.2.1. Comentario Preliminar.

Ya señalamos en la Introducción del Capítulo que los tests más eficaces de factorización de números naturales escapan con mucho de un curso con el propuesto. Para expresar el estado actual de los métodos conocidos presentamos la siguiente Tabla procedente del texto [GatGe, 99]. Así, sea N un número natural a factorizar y supongamos $n := \lfloor log_2 N \rfloor$ la talla binaria de N.

Procedimiento	**Tiempo**
Criba de Eratóstenes	$2^{n/2} n^{O(1)}$
Pollard ρ	$2^{n/4} n^{O(1)}$
Lenstra C. Elíptica [11]	$2^{n^{1/2}} n^{O(1)}$
Criba del Cuerpo de Números [12]	$2^{n^{1/3}} n^{O(1)}$

De esta Tabla se sigue que el método que propondremos en esta Sección no es el más eficaz de los conocidos; pero, en aras de su simplicidad y amplia difusión, hemos escogido presentar el método ρ de J.M. Pollar e informar a los alumnos de la existencia de métodos alternativos en la literatura del tema.

13.4.2.2. Algoritmos de Factorización basados en Fermat.

En el año 1643, P. de Fermat introduce un concepto de factorización de números que ha permanecido, con diversas variantes, hasta nuestros días. El concepto para la factorización en Fermat se explica del modo siguiente :
Dado un número natural impar $n \in \mathbb{N}$, supongamos que queremos calcular dos números naturales u y v tales que $n = uv$. Consideremos los números racionales

$$x := \frac{(u+v)}{2}, \quad y := \frac{(v-u)}{2}. \tag{13.4.1}$$

Obsérvese que si n es impar, también lo son u y v y por ende, x e y son números enteros. Entonces, el punto $(x, y) \in \mathbb{Z}^2$ pertenece a la có3nica plana dada por la ecuación :

$$n = x^2 - y^2.$$

Los métodos de factorización llamados de Fermat se basan en la búsqueda sistemática de puntos (x, y) en esa cónica. A partir de ellos, invirtiendo las relaciones descritas en la Ecuación (13.4.1) tendremos los valores u y v buscados. Los métodos son esencialmente métodos de tipo criba que prueban con muchos posibles pares (x, y) hasta dar con alguno de ellos. Las variantes esenciales de este método son descritas en [Kn, 97] con bastante detalle. Citemos esencialmente las siguientes :

- Factorizar Mediante sumas y productos.
- Factorizar mediante Cribas.

El método no es muy eficaz (la complejidad sigue siendo alta) así que surgen alternativas, algunas de las cuales citaremos a continuación.

13.4.2.3. El método ρ de Pollard.

Es uno de los métodos usados, se puede encontrar en las versiones actuales de Maple, por ejemplo, entre los algoritmos de factorización más destacados. Incluiremos la variante de Brent sobre el concepto original de Pollard. El princio del método de Pollard–Brent se basa en el principio de que los "generadores de sucesiones alateorias no son muy aleatorios". Una buena referencia son los ejercicios descritos en [Kn, 97].
Sea $f \in \mathbb{Z}[T]$ un polinomio univariado de grado mayor que 2. Sea N el número entero dado. Consideremos la sucesión :

$$x_0 = A,\ x_{m+1} = f(x_m)\ (mod\ n),\ \forall m \geq 1.$$

De otro lado, sea p un factor primo de N desconocido y consideremos la sucesión

$$y_0 = A,\ y_{m+1} = f(y_m)\ (mod\ p),\ \forall m \geq 1.$$

Obsérvese que $y_m(modp) = x_m(modp)$.

Proposición 13.4.19 (R. Brent)**.** *Con las anteriores notaciones, se tiene que la órbita de A tiene que tener una longitud acotada por $\sqrt{p}$. Más aún, si $\ell(m)$ es la máxima potencia de 2 menor que m, se tiene*

$$y_m - y_{\ell(m)-1} = 0\ (mod\ p).$$

El enunciado fundamental del Método de Pollard es el siguiente :

Teorema 13.4.20. *Sea $N \in \mathbb{N}$ un número compuesto, p su factor primo más pequeño y $f(x) := x^2 + 1$. Si la sucesión $\{f(x_m) : m \in \mathbb{N}\}$ anterior se comporta como una sucesión aleatoria, el método de Pollard calcula el factor primo más pequeño de N en tiempo*

$$O(\sqrt{p} log^2 N),$$

y N es factorizado completamente en tiempo $2^{n/4} n^{O(1)}$, donde $n := log_2 N$.

Por lo tanto, el procedimiento de Pollard funciona buscando el entero m tal que $gcd(x_m - x_{\ell(m)-1}, N) \neq 1$. Esto es realizado por el siguiente procedimiento.

INPUT N (entero compuesto) y $m \leq N^{1/4}$
c:=1; A:=2; $\ell := \ell(m)$.
$x := x_{\ell(m)-1}$ (mod N)
$y := x_m$ (mod N)

```
    g := gcd(x − y, N)
      if g = N, then Output "FAILURE"
        elif g>1, then Output g
        else Ouput "Failure"
      fi
end
```

Pseudo–código del Método de Pollard simple.

Este algoritmo genera un factor no trivial de N o falla. En el caso de que falle, se aplica heurísticamente modificando los valores iniciales c, A y m. De ahí el carácter heurístico sugerido por [GatGe, 99] en su comentario.

13.5. Reciprocidad Cuadrática: El algoritmo de Solovay-Strassen

El Teorema de la Reciprocidad Cuadrática es un clásico Teorema de la Teoría de Números. Los primeros resultados parciales fueron obtenidos por Leonard Euler. En su trabajo de 1798, Adien-Marie Legendre mejora los resultados de Euler e introduce el llamado símbolo de Legendre (cf. [Leg, 1798]). Pero será Carolo Federico Gavss quien, a sus 24 años, publica seis demostraciones de la Ley de reciprocidad cuadrática (cf. [Gauss. 1801]). Desde entonces, diversos autores han publicado demostraciones alternativas o extensiones de la misma. Entgre ellos L. Kronecker, E. Kummer, Jacobi, D. Hilbert o E. Artin y su "Reciprocidad de Artin". Aquí hemos elegido la demostración publicada por G. Einsenstein en 1850 (cf. [Eins, 1850]). Luego estudiaremos la aplicación de este resultado al diseño de un algoritmo aleatorio para la primalidad debido a Solovay y V. Strassen (cf. [SoSt, 77]).

13.5.0.1. Símbolo de Lengendre. Criterio de Euler

Definición 13.5.1 (Símbolo de Lengendre). Sea $p \in \mathbb{N}$ un número primo impar y sea $m \in \mathbb{N}$ un entero positivo. Definimos el símbolo de Legendre mediante la indetidad siguiente:

$$\left(\frac{m}{p}\right) := m^{\frac{p-1}{2}} \quad \text{mód } p.$$

Es decir, es el resto módulo p (i.e. en $\mathbb{Z}/p\mathbb{Z}$) de $m^{\frac{p-1}{2}}$.

Para entender el significado, introducimos la noción de residuo (o resto) cuadrático.

Definición 13.5.2 (Residuo Cuadrático). Sea $p \in \mathbb{N}$ un número primo y $m \in \mathbb{Z}$ un entero. Decimos que m es un residuo cuadrático módulo p si la siguiente ecuación posee solución en $\mathbb{Z}/p\mathbb{Z}$:

$$X^2 - m = 0.$$

O, equivalentemente, si el polinomio $X^2 - m$ es reducible en $\mathbb{Z}/p\mathbb{Z}[X]$.

Lema 13.5.1 (Euler). *Si $p \in \mathbb{N}$ es un número primo impar, entonces el número de residuos cuadráticos módulo p en $(\mathbb{Z}/p\mathbb{Z})^*$ es exactamente $\frac{p-1}{2}$.*

Demostración. □

Teorema 13.5.2 (Criterio de Euler). *Sea $p \in \mathbb{N}$ un número primo impar y $m \in \mathbb{N}$ entero no negativo, se tiene:*

$$\left(\frac{m}{p}\right) = \begin{cases} 1, & \text{si } p \nmid m \text{ y } m \text{ es un residuo cuadrático módulo } p \\ -1, & \text{si } m \text{ no es un residuo cuadrático módulo } p \\ 0, & \text{si } p \mid m \end{cases} \tag{13.5.1}$$

Demostración. □

Corollario 13.5.3. *Con las notaciones anteriores:*

i) EL producto de residuos cuadráticos es residuo cuadrático.

ii) EL producto de un residuo cuadrático por un no residuo cuadrático da un elemento que no es residuo cuadrático.

iii) El producto de dos números que no son residuos cuadráticos es un residuo cuadrático.

Proposición 13.5.4 (**Propiedades Elementales del Símbolo de Lengendre).** *Con las notaciones anteriores:*

i) Se puede extender de manera natural el símbolo de Legendre a enteros cualesquiera (y no solamente positivos).

ii) El símbolo de Legendre es cíclico de orden p.

iii) El símbolo de Legendre es una función completamente multiplicativa, i.e.

$$\left(\frac{mn}{p}\right) = \left(\frac{m}{p}\right)\left(\frac{n}{p}\right).$$

Demostración. □

Observación 13.5.1. Como consecuencia del carácter completamente multiplicativo del símbolo de Legendre, podemos concluir que, para conocer el símbolo de Legendre de un número entero $m \in \mathbb{Z}$ basta con conocer el símbolo de Legnedre de sus factores y una factorización. Es decir, si

$$m = q_1^{\alpha_1} \cdots q_r^{\alpha_r},$$

es una factorización de m en producto de primos, entonces,

$$\left(\frac{m}{p}\right) = \left(\frac{q_1}{p}\right)^{\alpha_1} \cdots \left(\frac{q_r}{p}\right)^{\alpha_r}.$$

Proposición 13.5.5 (**Algunos valores del símbolo de Legendre).** *Con las notaciones anteriores:*

i)

$$\left(\frac{m^2}{p}\right) = \begin{cases} 1, & \text{si } p \nmid m \\ 0, & \text{si } p \mid m \end{cases}$$

ii)

$$\left(\frac{-1}{p}\right) = \begin{cases} 1, & \text{si } p \equiv 1 \mod 4 \\ -1, & \text{si } p \equiv 3 \mod 4 \end{cases}$$

iii)

$$\left(\frac{2}{p}\right) = \begin{cases} 1, & \text{si } p \equiv 1 \vee 7 \mod 8 \\ -1, & \text{si } p \equiv 3 \vee 5 \mod 8 \end{cases}$$

Demostración. Para la propiedad c se necesita el Lema de Gauss para la Ley de Reciprocidad Cuadrática. □

13.5.1. La demostración de Einsenstein

13.5.1.0.1. Una Construcción Geométrica Se recomienda al lector hacer los dibujos de cada paso de esta demostración. Sean $p \in \mathbb{N}$ un número primo impar, $q \in \mathbb{N}$ un entero positivo impar tal que $p \not| q$. Consideremos el rectángulo abierto $R := (0,p) \times (0,q) \subseteq \mathbb{R}^2$. Consideremos el retículo $\mathbb{Z}^2 \subseteq \mathbb{R}^2$ y sea Λ la parte de ese retículo en el rectángulo R anterior. Es decir,

$$\Lambda := \mathbb{Z}^2 \cap R = \mathbb{Z}^2 \cap ((0,p) \times (0,q)) .$$

Nótese que los puntos en Λ son los puntos en

$$\{1,2,\ldots,p-1\} \times \{1,2,\ldots,q-1\}.$$

En el rectángulo R consideramos la diagonal $D \subseteq R$. Es decir, la recta D que, pasando por el origne, pasa por el punto $C := (p,q)$. Se trata de la recta:

$$D := \{(x,y) \in \mathbb{R}^2 \ : \ y = \frac{q}{p}x\}.$$

Como $p \not| q$, entonces la diagonal D no contiene ningún punto de Λ.
Añadamos dos ejes a nuestra figura. El eje vertical pasando por el punto $(p/2,0) \in \mathbb{R}^2$ y el eje horizontal pasando por el punto $(0,q/2) \in \mathbb{R}^2$. COmo p y q son impares, tampoco hay ningún punto de Λ en ninguno de los dos ejes.

Lema 13.5.6. *Sea $u \in \{1,\ldots,p-1\}$ y sea L_u la recta vertical pasando por el punto $(u,0) \in \mathbb{R}^2$. Entonces,*

$$\sharp (L_u \cap \Lambda \cap (\triangle ABC)) = \left\lfloor \frac{uq}{p} \right\rfloor.$$

donde $\triangle ABC$ es el triángulo determinado por $A = (0,0)$, $B = (p,0)$ y $C = (p,q)$.

Demostración. Nótese que estamos contando los puntos de L_u que "caen" debajo de la diagonal D en el conjunto Λ. □

Dada cualquier región $S \subseteq \Lambda$, denotaremos mediante:

$$S_1 := \{(x,y) \in S \ : \ x \notin 2\mathbb{Z}\}.$$

$$S_2 := \{(x,y) \in S \ : \ x \in 2\mathbb{Z}\}.$$

Lema 13.5.7. *Con las notaciones anteriores, sean X ey Y los puntos:*

$$X := (p/2,0) , \ \ Y = (p/2,q/2).$$

Entonces, se tiene:

i)

$$\sharp (\Lambda \cap (\triangle ABC)) = \sum_{u=1}^{p-1} \left\lfloor \frac{uq}{p} \right\rfloor.$$

ii)

$$\sharp (\Lambda \cap (\triangle ABC)_2) = \sum_{u \in 2\mathbb{Z} \cap \{1,\ldots,p-1\}} \left\lfloor \frac{uq}{p} \right\rfloor.$$

iii)

$$\sharp (\Lambda \cap (\triangle AXY)_2) = \sum_{u \in 2\mathbb{Z} \cap \{1,\ldots,\frac{p-1}{2}\}} \left\lfloor \frac{uq}{p} \right\rfloor.$$

Donde $\triangle ABC$ y $\triangle AXY$ son los triángulos determinados por esos puntos.

Demostración. □

13.5.1.0.2. Una Reflexión actuando sobre el rectángulo R: Consideremos la siguiente refelxión actuando sobre el rectángulo R:

$$\begin{array}{rccc} \varphi: & R & \longrightarrow & R \\ & (x,y) & \longmapsto & (p-x, q-y). \end{array}$$

Obsérvese que φ es reflexión (i.e. $\varphi^2 = Id_R$) y que $\varphi(\mathbb{Z}^2) \subseteq \mathbb{Z}^2$. Además,s e tiene:

Proposición 13.5.8. *Con las notaciones anteriores:*

$$\varphi\left((\triangle CYZ)_2\right) = (\triangle AXY)_1,$$

$$\varphi\left((\triangle CYZ)_1\right) = (\triangle AXY)_2,$$

donde A, X e Y son como antes y $Z = (p/2, q)$.

Demostración. □

Corollario 13.5.9. *Con las anteriores notaciones,*

$$\sharp\left(\Lambda \cap (\triangle AXY)\right) = \sum_{u=1}^{\frac{p-1}{2}} \left\lfloor \frac{uq}{p} \right\rfloor = \sharp\left((\triangle AXY)_2\right) + \left((\triangle CYZ)_2\right).$$

Demostración. Como φ es biyectiva, basta con observar que

$$\sharp\left((\triangle CYZ)_2\right) = \sharp\left((\triangle AXY)_1\right).$$

□

Proposición 13.5.10. *Consideremos la región $\overline{BXYC}$ determinada por el polígono cuyos vértices son $B = (p, 0), X = (p/2, 0), Y = (p/2, q/2), C = (p, q)$. Entonces,*

$$\sharp\left(\left(\overline{BXYC}\right)_2\right) = \sharp\left((\triangle CYZ)_2\right) \quad \text{mód } 2.$$

Es decir,

$$(-1)^{\sharp\left(\left(\overline{BXYC}\right)_2\right)} = (-1)^{\sharp\left((\triangle CYZ)_2\right)}.$$

Demostración. □

Corollario 13.5.11. *Con las notaciones anteriores, se tiene:*

$$(-1)^{\sum_{u\in 2\mathbb{Z}\cap\{1,\ldots,p-1\}} \lfloor \frac{uq}{p} \rfloor} = (-1)^{\sum_{u=1}^{\frac{p-1}{2}} \lfloor \frac{uq}{p} \rfloor}.$$

Demostración. □

Teorema 13.5.12 (Lema de Einsenstein). *Sea $p \in \mathbb{N}$ un número primo impar y $m \in \mathbb{N}$ tal que $p \not| m$. Supongamosque m es impar, entonces*

$$\left(\frac{m}{p}\right) = (-1)^{\sum_{u\in 2\mathbb{Z}\cap\{1,\ldots,p-1\}} \lfloor \frac{um}{p} \rfloor} = (-1)^{\sum_{u=1}^{\frac{p-1}{2}} \lfloor \frac{uq}{p} \rfloor}.$$

Demostración. □

Teorema 13.5.13 (Ley de Reciprocidad Cuadrática). *Sea $p, q \in \mathbb{N}$ dos números primos impares. Entonces.*

$$\left(\frac{q}{p}\right)\left(\frac{p}{q}\right) = (-1)^{\frac{p-1}{2}\frac{q-1}{2}}.$$

Demostración. Usar el Lema de Einsenstein. □

13.5.2. El Lema de Gauss y las raíces cuadradas modulares de 2

Lema 13.5.14 (Lema de Gauss). *Sea $p \in \mathbb{N}$ un número primo impar y sea $m \in \mathbb{N}$ un entero positivo coprimo con p. Consideremos el conjunto*

$$S := \{um \ : \ 1 \leq u \leq \frac{p-1}{2}\}.$$

Definamos el conjunto

$$S' := \{x \mod p \ : \ x \in S\} \subseteq \{1, \ldots, p-1\}.$$

Sea $n := \sharp\{r \in S' \ : \ p/2 \leq r\}$, entonces

$$\left(\frac{m}{p}\right) = (-1)^n .$$

Demostración. Sigue esencialmente las ideas de Lema de Einsenstein, quien se inspiró en este Lema de Gauss, □

Corollario 13.5.15. *Si $p \in \mathbb{N}$ es un número primo impar, entonces*

$$\left(\frac{2}{p}\right) = \begin{cases} 1, & \text{si } p = 1 \vee 7 \mod 8 \\ -1, & \text{si } p = 3 \vee 5 \mod 8 \end{cases}$$

Demostración. Aplicar el Lema de Gauss □

13.5.3. El símbolo de Jacobi

Lo introdujo C.G.J. Jaconi en su trabajo de 1837 [Jac, 1837].

Definición 13.5.3 (Símbolo de Jacobi). Sean $m, n \in \mathbb{N}$ dos números enteros positivos, n impar. Sea $n = p_1^{\alpha_1} \cdots p_r^{\alpha_r}$ una factorización de n. Definimos el símbolo de Jacobi $\left(\frac{m}{n}\right)$ mediante:

$$\left(\frac{m}{n}\right) := \left(\frac{m}{p_1}\right)^{\alpha_1} \cdots \left(\frac{m}{p_r}\right)^{\alpha_r} .$$

El símbolo de Jacobi es una generalización del símbolo de Legendre que permite poner números enteros no primos, aunque impares, en el denominador.

Proposición 13.5.16. *El símbolo de Jacobi es completamente multiplicativo. Es decir,*

$$\left(\frac{m_1 m_2}{n}\right) = \left(\frac{m_1}{n}\right)\left(\frac{m_2}{n}\right),$$

$$\left(\frac{m}{n_1 n_2}\right) = \left(\frac{m}{n_1}\right)\left(\frac{m}{n_2}\right),$$

donde n, n_1, n_2 son impares.

Demostración. □

Observación 13.5.2. El símbolo de Jacobi depende solamente del resto de m por n, es decir,

$$\left(\frac{m}{n}\right) = \left(\frac{rem(m,n)}{n}\right),$$

donde $rem(m,n)$ s el resto de la división de m por n. El símbolo de Jacobi también verifica una Ley de reciprocidad cuadrática, que pasaremos a probar a continuación.

Lema 13.5.17. *Sea $n \in \mathbb{N}$ un número impar y supongamos una factorización $n = p_1 \cdots p_r$, con primos repetidos si fuera necesario. Entonces,*

$$\frac{n-1}{2} - \left(\sum_{i=1}^{r} \frac{p_i - 1}{2} \right) = 0 \quad \text{mód } 2.$$

Lema 13.5.18. *Sean $m, n \in \mathbb{N}$ dos números impares y supongamos dos factorizaciones con m y n con premios repetidos*

$$n = p_1 \dots p_r, \;\; m = q_1 \cdots q_s.$$

Entonces,

$$\left[\frac{m-1}{2} \right] \left[\frac{n-1}{2} \right] = \left(\sum_{i=1}^{r} \frac{p_i - 1}{2} \right) \left(\sum_{j=1}^{s} \frac{q_j - 1}{2} \right) \quad \text{mód } 2.$$

Demostración. □

Teorema 13.5.19 (Ley de Reciprocidad Cuadrática de Jacobi). *Con las motaciones anteriores, dados $m, n \in \mathbb{N}$ dos enteros impares, se tiene*

$$\left(\frac{m}{n} \right) \left(\frac{n}{m} \right) = (-1)^{\frac{m-1}{2} \frac{n-1}{2}}.$$

En los casos -1 y 2 se tiene

$$\left(\frac{-1}{n} \right) = (-1)^{\frac{n-1}{2}}, \;\; \left(\frac{2}{n} \right)^{\frac{n^2-1}{8}}.$$

Observación 13.5.3. La Ley de Reciprocidad de Jacobi nos permite diseñar un algoritmo para el cálculo del símdolo de Jacobi de dos números impares sin conocer su factorización.
Es claro que la propia defínción nos permite calcular $\left(\frac{m}{n}\right)$, si n es primo impar. Si no fuera primo, tendríamos que factorizar n y usar el carácter multiplicativo del símbolo de Jacobi. Pero queremos evitar esa necesidad de la factorización. Por eso usamos la Reciprocidad Cuadrática como sigue:
En primer ligar, basta con saber hallar el símbolo $\left(\frac{m}{n}\right)$ cuando $m < n$. Si no fuera así, bastará con reemplazar m por $rem(m, n)$.
Para seguir reduciendo, procedemos como sigue:

- Si m es impar, hallar

$$\left(\frac{m}{n} \right) = \left(\frac{n}{m} \right) (-1)^{\frac{n-1}{2} \frac{m-1}{2}} = \left(\frac{rem(n,m)}{m} \right) (-1)^{\frac{n-1}{2} \frac{m-1}{2}}.$$

- Si m es par, hallar

$$\left(\frac{m}{n} \right) = \left(\frac{2}{n} \right)^{r} \left(\frac{m_1}{n} \right),$$

  donde m_1 es impar $m = 2^r m_1$.

Estas dos transformaciones permiten reducir el cálculo de $\left(\frac{m}{n}\right)$ al cálculo del símbolo de Jacobi con un número estrictamente más pequeño en la parte inferior. Además, los números involucrados se reducen a un ritmo superior que en el algoritmo de Euclides.

13.5.4. El Test de Solovay-Strassen

Introducido en el trabajo de Solovay y Strassen [SoSt, 77].

Lema 13.5.20. *Sea $p \in \mathbb{N}$ un número primo. El polinomio $X^{p-1} - 1 \in \mathbb{Z}/p\mathbb{Z}[X]$ factoriza completamente en $\mathbb{Z}/p\mathbb{Z}[X]$ y su factorización tiene la forma siguiente:*

$$X^{p-1} - 1 = \prod_{i=1}^{p-1}(X - i).$$

Todas sus raíces son distintas y, además, si $n \mid (p-1)$, es un divisor de $(p-1)$, entonces en polinomio $X^n - 1$ se escinde completamente en $\mathbb{Z}/p\mathbb{Z}$ y posee n raíces distintas.

Demostración. Usar bien el teorema Pequeño de Fermat. □

Lema 13.5.21. *Sean $q_1, g_2 \in (\mathbb{Z}/p\mathbb{Z})^*$ dos elementos del grupo de las unidades y supongamos que verifican:*

- *$ord(g_1) = d_1$, $ord(g_2) = d_2$,*
- *$gcd(d_1, d_2) = 1$.*

Entonces, el orden del elemento $g_1 g_2$ es $d_1 d_2$.

Demostración. Lo habitual. □

Teorema 13.5.22 (Teorema Pequeño de Fermat). *Un número $n \in \mathbb{N}$ es primo si y solamente si el grupo multiplicativo $(\mathbb{Z}/p\mathbb{Z})^*$ es cíclico de orden $p-1$.*

Proposición 13.5.23. *Sea $p \in \mathbb{N}$ un número primo impar y sea $g \in (\mathbb{Z}/p\mathbb{Z})^*$ un elemento primitivo. Entonces,*

i) Los residuos cuadráticos módulo p son dados mediante:

$$\{g^r \ : \ 0 \leq r \leq p-2, r \in 2\mathbb{Z}\}.$$

ii) Los que no son residuos cuadráticos módulo p son dados mediante:

$$\{g^r \ : \ 0 \leq r \leq p-2, r \in 2\mathbb{Z}+1\}.$$

Además, el número de residuos cuadráticos y el de los que no son residuos cuadráticos es el mismo $\frac{p-1}{2}$.

Demostración. Usar el símbolo de Jacobi. □

Corollario 13.5.24. *Si $p \in \mathbb{N}$ es un primo impar hay elementos en $(\mathbb{Z}/p\mathbb{Z})^*$ que no son residuos cuadráticos.*

Proposición 13.5.25. *Sea $p \in \mathbb{N}$ un número primo impar, entonces el grupo multiplicativo $\left(\mathbb{Z}/p^2\mathbb{Z}\right)^*$ es un grupo cíclico de orden $p(p-1)$. En particular, posee al menos un generador primitivo como grupo multiplicativo.*

Demostración. □

Definición 13.5.4. Sea $n \in \mathbb{N}$ un número compuesto impar y sea $a \in \mathbb{Z}/n\mathbb{Z}$ tal que $gcd(a, n) = 1$. Decimos que a es un testigo de Euler para n si verifica:

$$a^{\frac{n-1}{2}} \neq \left(\frac{a}{n}\right) \quad \text{mód } n,$$

donde $\left(\frac{a}{n}\right)$ es el símbolo de Jacobi.

Proposición 13.5.26. *Si $n \in \mathbb{N}$ es un número compuesto impar, entonces hay, al menos, un testigo de Euler en $(\mathbb{Z}/n\mathbb{Z})^*$. Obviamente, si $n \in \mathbb{N}$ es primo impar no posee testigos de Euler. Luego un número impar es compuesto si y solamente si posee testigos de Euler.*

Demostración. La idea es que si $n \in \mathbb{N}$ es primo, el símbolo de Jacobi $\left(\frac{a}{n}\right)$ coincide con el de Legendre y, por tanto, no puede poseer testigos de Euler. La parte dura es probar que si n es compuesto posee un testigo de Euler. □

Definición 13.5.5. Sea $n \in \mathbb{N}$ un número impar y sea $a \in \mathbb{Z}/n\mathbb{Z}$ tal que $gcd(a, n) = 1$. Decimos que a es un mentiroso de Euler para n si verifica:

$$a^{\frac{n-1}{2}} = \left(\frac{a}{n}\right) \quad \text{mód } n,$$

donde $\left(\frac{a}{n}\right)$ es el símbolo de Jacobi.

Proposición 13.5.27. *Sea $n \in \mathbb{N}$ un número impar. Entonces, si n es compuesto el número de testigos de Euler es mayor que el número de mentirosos de Euler. De hecho, es una equivalencia, dado que, si n es primo, no hay testigos de Euler.*

Demostración. □

El Test de Solovay Strassen funcionaría del modo siguiente:

INPUT: $n \in \mathbb{N}$ un número impar.
Guess randomly $a_1, \ldots, a_k \in \{2, \ldots, k-1\}$
eval $x_i := \left(\frac{a_i}{n}\right)$, $1 \leq i \leq k$
(usando la técnica de las reducciones via la Recirpocidad Cuadrática de Jacobi)
eval $y_i := a_i^{\frac{n-1}{2}}$ mód n, $1 \leq i \leq k$
(usando el cálculo de las potencias con $O(\log_2(n))$ multiplicaciones)
if $x_i = 0$ o $x_i \neq y_i$ para algún i, OUTPUT: Compuesto
else, OUTPUT: PRIMO
fi
end

Teorema 13.5.28. *[SoSt, 77] El lenguaje $COMP := \mathbb{N} \setminus PRIMES$ está en* **RP**. *Es decir, el algoritmo verifica lo siguiente:*

i) El algoritmo se ejecuta en tiempo polinomial en k y el $\log(n)$,

ii) SOLIDEZ: *Si $n \in \mathbb{N}$ es primo, el algoritmo responde PRIMO.*

iii) Si $n \in \mathbb{N}$ es compuesto, el algoritmo responde COMPUESTO con probabilidad mayor que

$$1 - \frac{1}{2^k}.$$

Tomando k tal que $1/2^k < 1/3$, habremos terminado.

Demostración. Si n es primo no hay testigos de Euler, así que siempre entra en el **else**. En otro caso, □

13.6. Tests de Nulidad para Polinomios.

Si bien antes hemos dicho que los esquemas de evaluación son una estructura de datos muy adecuada para tratar y manipular polinomios multivariados, hay una dificultad esencial para que sea tan precisa como la codificación densa o rala de polinomios. No es fácil decidir si dos códigos de dos esquemas de evaluación evalúan el mismo polinomio. Para resolverlo, tenemos dos opciones básicas :

i) Interpolar obteniendo todos los coeficientes y comparando a posteriori. Esta no es una excelente idea, Por ejemplo, dadas dos formas de Cayley–Chow de dos variedades cero–dimensionales, interpolar para decidir si definen la misma variedad supone un esfuerzo considerablemente caro que nos lleva, usualmente, a una complejidad del orden d^{n^2}.

ii) Desarrollar Tests de Nulidad para Esquemas de evaluación. Se trata de algoritmos que evalúan los códigos dados en un número finito (y "pequeño") de puntos. Con los valores de esas "pocas" evaluaciones, tratamos de decidir si ambos esquemas de evaluación evalúan la misma función.

En esta Sección nos ocuparemos de estos Tests de Nulidad que pertenecen a tres tipos fundamentalmente :

i) *Los Tests de Zippel–Schwartz.* Se trata de seleccionar un conjunto fijo y, para cada polinomio, hay muchos puntos en los que no se anula. Se trata de un Test Probabilista que depende del polinomio y el esquema particular que tratamos. Fueron introducidos en los trabajos de J.T. Schwartz (cf. [Schwa, 80]) y R. Zippel (cf. [Zi, 90]).

ii) *Los Cuestores o Correct Test Sequences.* Es un método alternativo al método de Zippel–Schwarzt. En ocasiones genera gran confusión entre los especialistas que no entienden la diferencia. El método de los cuestores es un método que no depende del polinomio particular que se discute sino de una clase de polinomios (y para ser más precisos de una familia uniracional). Por ejemplo, depende la clase de complejidad y es válido para toda ella. Fue introducido en el trabajo de J. Heintz y C.P. Schnorr (cf. [HeSc, 82]) y refinado en el trabajo [KrPa, 96]. La versión que aquí se incluye es la versión refinada de [KrPa, 96].

iii) *Witness Theorem.* La existencia de un sólo punto donde un polinomio no se anula aparece ya en el trabajo de L. Kronecker y se conoce como "Esquema de Kronecker". En el caso de polinomios con coeficientes en $\mathbb{Z}$ el resultado aparece recogido, y atribuido a Kronecker, en el trabajo de J. Heintz y C.P. Schnorr de 1982, citado al pie. Posteriormente, M. Shub, S. Smale y sus co-autores redescubren el esquema de Kronecker para polinomios con coeficientes en un cuerpo de números y lo denomina "Witness Theorem" (véase también [BlCuShSm, 98]. Sin embargo, las estimaciones de Smale y sus colaboradores eran muy groseras. Las estimaciones se mejoran en el trabajo [CaHaMoPa, 01].

Haremos la demostración del Test de Schwartz-Zippel y dejaremos los otros resultados para cuando tengamos una mejor fundamentación matemática.

13.6.1. El Test de Schwartz–Zippel.

La clave del Test de Schwartz–Zippel es el siguiente enunciado :

Lema 13.6.1. *Sea $f \in K[X_1, \ldots, X_n]$ un polinomio no nulo. Definamos recursivamente los siguientes polinomios :*

$$Q_1 := f \in k[X_1, \ldots, X_n],\ d_1 := deg_{X_1}(Q_1)$$

Sea $Q_2 \in K[X_2, \ldots, X_n]$ el coeficiente de $X_1^{d_1}$ en Q_1. Para $i \leq 2$ definamos recursivamente :

$$d_i := deg_{X_i} Q_i, Q_{i+1} \in K[X_{i+1}, \ldots, X_n]$$

el coeficiente de $X_i^{d_i}$ in Q_i. Para $1 \leq i \leq n$ sea I_i un subconjunto finito de K.
Entonces, el número de ceros de F in $I_1 \times \cdots \times I_n$ es a lo sumo

$$\sharp(I_1 \times \cdots \times I_n)\left(\frac{d_1}{\sharp(I_1)} + \cdots + \frac{d_n}{\sharp(I_n)}\right)$$

Demostración. La prueba se sigue por inducción en n. En el caso $n = 1$ es obvio que un polinomio univariado de grado d_1 con coeficientes en un cuerpo no posee más de d_1 ceros en el cuerpo. La razón última es que $K[X]$ es un dominio de factorización única.
Consideremos ahora el caso $n > 1$. Consideremos el polinomio $Q_2 \in K[X_2, \ldots, X_n]$ que es el coeficiente director de f con respecto a la variable X_1. Por construcción, la secuencia de polinomios $Q_2, \ldots, Q_n$ y de grados $d_2, \ldots, d_n$ es la misma comenzando por Q_2 o comenzando por f. Por tanto, podemos aplicar la hipótesis inductiva a Q_2 y tenemos que el número de elementos de $x = (x_2, \ldots, x_n) \in \prod_{i=2}^n I_i$ en los que Q_2 no se anula es de cardinal mayor que:

$$\prod_{i=2}^n \sharp(I_i) - \prod_{i=2}^n \sharp(I_i)\left(\sum_{i=2}^n \frac{d_i}{\sharp(I_i)}\right) = \prod_{i=2}^n \sharp(I_i)\left(1 - \sum_{i=2}^n \frac{d_i}{\sharp(I_i)}\right).$$

Para cada uno de los $x = (x_2, \ldots, x_n) \in \prod_{i=2}^n I_i$ en los que $Q_2(x) \neq 0$, el polinomio f toma la forma:

$$f(X_1, x_2, \ldots, x_n) = Q_2(x)Y_1^{d_1} + h(X_1),$$

donde h es un polinomio de grado a lo sumo $d_1 - 1$. En consecuencia, este polinomio univariado no se puede anular en, al menos, $\sharp(I_1) - d_1$ elementos de I_1. Y esto para cada x con esas condiciones. Por tanto, f no se anula en, al menos, el siguiente número de elementos de $\prod_{i=1}^n I_i$:

$$P := (\sharp(I_1) - d_1)\prod_{i=2}^n \sharp(I_i)\left(1 - \sum_{i=2}^n \frac{d_i}{\sharp(I_i)}\right).$$

Desarrollando este producto obtenemos

$$P := \sharp(I_1)\prod_{i=2}^n \sharp(I_i)\left(1 - \sum_{i=2}^n \frac{d_i}{\sharp(I_i)}\right) - d_1\prod_{i=2}^n \sharp(I_i)\left(1 - \sum_{i=2}^n \frac{d_i}{\sharp(I_i)}\right).$$

Luego

$$P := \prod_{i=1}^n \sharp(I_i) - \prod_{i=1}^n \sharp(I_i)\left(\sum_{i=2}^n \frac{d_i}{\sharp(I_i)}\right) - \frac{d_1}{\sharp(I_1)}\prod_{i=1}^n \sharp(I_i) + \frac{d_1}{\sharp(I_1)}\prod_{i=1}^n \sharp(I_i)\left(\sum_{i=2}^n \frac{d_i}{\sharp(I_i)}\right).$$

Por tanto,

$$P \geq \prod_{i=1}^n \sharp(I_i)\left(1 - \sum_{i=2}^n \frac{d_i}{\sharp(I_i)}\right) - \frac{d_1}{\sharp(I_1)}\prod_{i=1}^n \sharp(I_i) = \prod_{i=1}^n \sharp(I_i)\left(1 - \sum_{i=1}^n \frac{d_i}{\sharp(I_i)}\right),$$

y se sigue el enunciado previsto. □

Tenemos la siguiente aplicación inmediata:

Corollario 13.6.2. *Con las notaciones previas, sea I un subconjunto finito de K and $F \in K[X_1, \ldots, X_n]$ un polinomio de grado d. La probabilidad de que una elección aletaoria de un punto $x \in I^n$ sea un cero de F es, a lo sumo :*

$$\frac{d}{\sharp(I)}$$

En particular, si $\sharp(I) \geq 2d + 1$,la probabilidad de que una elección aletoria en I^n de un valor no nulo de F es, al menos, $1/2$.

Demostración. Basta con usar el Lema precedente con $I = I_1 = \cdots = I_n$. □

Esto genera el siguiente algoritmo probablistia polinomial (**RP** o MonteCarlo) para detectar polinomios no nulos.

INPUT : El código de un esquema de evaluación bien paralelizable $\mathcal{G}$ en n variables, que evalúa un polinomio de grado d.

guess indeterministically

$$x = (x_1, \ldots, x_n) \in \{-d, \ldots, 0, 1, \ldots, d\}$$

Eval $\mathcal{G}$ en x.
if $\mathcal{G}(x) \neq 0$, OUTPUT : "Es un polinomio no nulo",
else OUTPUT : "Probablemente sea nulo",
fi
end

Para aumentar la "certeza" de que el polinomio probablemente sea el polinomio nulo, basta con repetir el proceso varias veces, observando que tras k reiteraciones, si nos hubiera salido siempre nulo, el polinomio sería nulo con probabilidad al menos

$$1 - \frac{1}{2^k}.$$

Corollario 13.6.3. *Con las anteriores notaciones, si existe un subconjunto I de K de, al menos, $2d$ elementos, entonces para todo polinomio $f \in K[X_1, \ldots, X_n]$ de grado a lo sumo d existe $(t_1, \ldots, t_n) \in K^n$ tal que $f(t_1, \ldots, t_n) \neq 0$.*

Demostración. Consecuencia inmediata del resultado precedente. □

13.6.2. Cuestores.

Definición 13.6.1. Dado un subconjunto (no necesariamente finito) $\mathcal{F} \subset K[X_1, \ldots, X_n]$ (que contiene al polinomio nulo) Diremos que un conjunto finito $\mathcal{Q} \subset \mathbb{K}^n$ es un *questor* (o una "Correct Test Sequence") *para* $\mathcal{F}$ si y sólo si para todo $F \in \mathcal{F}$ se tiene :

$$P \mid_{\mathcal{Q}} = 0 \Longrightarrow P \equiv 0 \ .$$

El resultado depende fuertemente de la desigualdad de Bézoutque analizaremos posteriormente. El primer resultado significativo es el siguiente :

Lema 13.6.4 ([KrPa, 96])**.** *Sea $O(L, \ell, n)$ el conjunto de todos los polinomios en $\mathbb{K}[X_1, \ldots, X_n]$ que se pueden evaluar mediante un esquema de evaluación de talla L y profundidad ℓ. Sea $W(L, \ell, n)$ la clausura Zariski de ese conjunto. Entonces, se verifica*

$$\deg W(L, \ell, n) \leq (2^{\ell+1} - 2)^{2L(L-(n+1))}.$$

Teorema 13.6.5 (Existencia de Conjuntos Cuestores)**.** *Sea K un cuerpo y sean $n, \ell, L \in \mathbb{N}$, $L \geq n + 1$. Sean*

$$u := (2^{\ell+1} - 2)\,(2^{\ell} + 1)^2 \qquad \text{and} \qquad t := 6\,(\ell L)^2.$$

Supongamos que la característica de K es mayor que u o que la característica de K es cero. Entonces, el conjunto $\{1, \ldots, u\}^n \subset K^n$ contiene al menos $u^{nt}\,(1 - u^{-\frac{t}{6}})$ conjuntos cuestores de longitud t para $W(L, \ell, n)$. En particular, contiene al menos uno.

Observe el lector que un elección aleatoria de un subconjunto cualquiera de t elementos del conjunto $\{1,\ldots,u\}^n \subset K^n$ es un conjunto cuestor para $W(L,\ell,n)$ con probabilidad mayor que

$$(1-u^{-\frac{t}{6}}) > 1/2.$$

Por tanto, el algoritmo del Tests de Zippel–Schwartz se transforma en un algoritmo **RP** mediante el siguiente esquema :

INPUT : El código de un esquema de evaluación bien paralelizable $\mathcal{G}$ en n variables, que evalúa un polinomio de grado d. Supongamos que $\mathcal{G}$ es de talla L y profundidad ℓ.
Compute u y t (como en el Teorema anterior)
    guess indeterministically $\mathcal{Q} \subseteq \{1,\ldots,u\}^n$ de cardinal t.
    Eval $\mathcal{G}$ en x para cada $x \in \mathcal{Q}$.
    if $\mathcal{G}(x) \neq 0$, para algún x OUTPUT : "Es un polinomio no nulo",
        else OUTPUT : "Probablemente sea nulo",
    fi
end

En este caso, la probabilidad de no cometer errores es, al menos

$$(1-u^{-\frac{t}{6}}).$$

13.6.3. Witness Theorem.

Comencemos fijando la terminología con la siguiente Definición :

Definición 13.6.2. Un testigo (Witness) para un polinomio $F \in K[X_1,\ldots,X_n]$ es un punto $\underline{\omega} \in K^n$ tal que si $F(\underline{\omega}) = 0$ implica $P = 0$.

En otras palabras, un testigo es un punto $\underline{\omega} \in K^n$ fuera del conjunto de puntos $K-$racionales de la hipersuprficie $V(F)$ (si hubiera alguno). La manera de obtenerlo de modo explícito es el siguiente Teorema

Teorema 13.6.6 (Witness Theorem)**.** *Sea K un cuerpo de números, $F \in K[X_1,\ldots,X_n]$ un polinomio no nulo evaluable por un esquema de evaluación Γ de talla L, profundidad ℓ y parámetros en $\mathcal{F} \subseteq K$. Sea $\omega_0 \in K$ tal que se verifica la siguiente desigualdad :*

$$ht(\omega_0) \geq \max\{\log 2, ht(\mathcal{F})\}.$$

Sea $N \in \mathbb{N}$ un número natural tal que se verifica la siguiente desigualdad :

$$\log N > \log(\ell+1) + (\ell+2)(\log 2)\,(\log\log(4L))\,.$$

Definamos recursivamente la siguiente secuencia de números algebraicos (conocida como Esquema de Kronecker) :

$$\omega_1 = \omega_0^N,$$

y para cada i, $2 \leq i \leq n$, definamos

$$\omega_i = \omega_{i-1}^N.$$

Entonces, el punto $\underline{\omega} := (\omega_1,\ldots,\omega_n) \in K^n$ es un testigo para F (i.e. $F(\underline{\omega}) \neq 0$).

La demostración se sigue por un argumento inductivo, que usa fuertemente una Generalización de la Desigualdad de Liouville, descrito en [CaHaMoPa, 01].

Corollario 13.6.7. *Sea $F \in K[X_1, \ldots, X_n]$un polinomio no nulo evaluable por un esquema de evaluación de talla L, profundidad ℓ y parámetros en $\mathcal{F} := \{x_1, \ldots, x_r\} \subseteq K$. Sea $\omega_{-1} \in K$ tal que*

$$ht(\omega_{-1}) := \text{máx}\{\log 2, ht(x_1), \ldots, ht(x_r)\}.$$

Definamos $\omega_0 \in K$ como $\omega_0 := \omega_{-1}^{2L^2}$. Sea $N \in \mathbb{N}$ un número natural tal que

$$\log N > \log(\ell + 1) + (\ell + 2)(\log 2)\,(\log\log(4L))\,.$$

Definamos recursivamente la siguiente secuencia de números algebraicos (Esquema de Kronecker) :

$$\omega_1 = \omega_0^N,$$

y para cada i, $2 \leq i \leq n$, definamos $\omega_i = \omega_{i-1}^N$. Entonces, el punto $\underline{\omega} := (\omega_1, \ldots, \omega_n) \in K^n$ es un Testigo para F (i.e. $F(\underline{\omega}) \neq 0$).

Observación 13.6.1.

i) El resultado nos da, codificado como un esquema de evaluación, un punto en el que no se anula el polinomio dado. Sin embargo, el tal Testigo es un punto que, en expansión binaria, resulta excesivo para poder manejarlo del modo adecuado. Por ello, el uso de métodos tipo Witness Theorem exigen poner un especial cuidado con el tamaño de los resultados intermedios o, en su defecto, usar Tests Probabilistas para números dados por esquemas de evaluación como los que se introducen en la Subsección 13.6.4 siguiente.

ii) El Caso Denso . Para la mayoría (genéricamente) de los polinomios $F \in K[X_1, \ldots, X_n]$ de grado d, el esquema de evaluación óptimo tiene talla

$$L = \binom{d+n}{n},$$

y profundidad $\ell = \log d + O(1)$. Los parámetros en este caso genérico son los coeficientes de F. El Teorema 13.6.6 anterior dice que existe una pequeña constante universal $c_2 > 1$, tal que la cota que debe verificar N es simplemente la cota siguiente :

$$\log N > c_2 n \log^2 d.$$

iii) El caso Ralo (Sparse/Fewnomials). Supongamos que nuestro polinomio $F \in K[X_1, \ldots, X_n]$ tiene pocos términos no nulos. Supongamos que F tiene grado a lo sumo d y que a lo sumo M de sus términos tienen coeficientes no nulos. Entre estos polinomios, el esquema de evaluación óptimo que los evalúa tiene talla del orden $L = c_3 M d$ (donde $c_3 > 0$ es una constante universal), y profundidad $\log_2 d + O(1)$. Entonces, el Teorema 13.6.6 anterior dice que existe una pequeña constante $c_3 > 1$, tal que la condición para definir N en el esquema de Kronecker es la siguiente :

$$\log N > c_3 \log d\,(\log\log d + \log\log M)\,.$$

13.6.4. Tests de Nulidad para Números Dados por Esquemas de Evaluación.

Del mismo modo que los esquemas de evaluación pueden ser la buena estructura de datos para codificar polinomios que aparecen en Teoría de la Elminación, la misma estructura de datos se aplica a la representación de números enteros y racionales que aparecen como resultados de eliminación. Del mismo modo que ocurre con los polinomios, los esquemas de evaluación de números son muy adecuados para realizar operaciones aritméticas entre números codificados mediante esquemas. Sin embargo, los Tests de Igualdad (o Tests de Nulidad) son problemáticos. En este sentido, la operación correspondiente a la evaluación de un polinomio es la operación de

evaluar un esquema de evaluación módulo una constante dada. La buena capacidad de adaptación de los esquemas de evaluación para estas propiedades hace que los Tests de Nulidad para esquemas de evaluación representando números pasen por los cálculos modulares.
Los algoritmos esenciales en esta Sección vienen de los trabajos de O.H. Ibarra, S. Moran (cf. [IbMo, 83] y del trabajo de A. Schönhage (cf. [Schö, 79]).
El resultado esencial es el siguiente Teorema que aprovecha ampliamente del Teorema de Densidad de los Números Primos.

Teorema 13.6.8. *Existe un algoritmo probabilista que, en tiempo polinomial decide la nulidad de todo número entero evaluado por un esquema de evaluación.*

El resultado técnico esencial es el siguiente Lema.

Lema 13.6.9. *Sea N un número entero no nulo tal que*

$$|N| \leq 2^{2n2^n}$$

Etonces, para n suficientemente grande, la probabilidad de que $N \neq 0$ mod m, para una elección aleatoria de $m \in \{1, \ldots, 2^{2n}\}$ es, al menos,

$$\frac{1}{4n}$$

El algoritmo correspondiente se define del modo siguiente :

INPUT : Γ el código de un esquema de evaluación de talla L evaluando un número entero.
Gess un conjunto D_L de $4L$ números enteros en el conjunto $\{1, \ldots, 2^{2L}\}$,
if $\Gamma \neq 0 mod m$, para algún $m \in D_L$, OUTPUT : "El número es no nulo".
else OUTPUT :"El número es probablemente nulo.
fi
end

La probabilidad de error en este algoritmo es del orden

$$(1 - \frac{1}{4L})^{4L} < e^{-1} < 1/2,$$

donde e es el número de Neper.

Parte IV

Algunas ideas más sutiles

Apéndice A

Sucinta Introducción al Lenguaje de la Teoría Intuitiva de Conjuntos

Contents

A.1. Introducción 311
A.2. Conjuntos. Pertenencia. 311
A.2.1. Algunas observaciones preliminares. 312
A.3. Inclusión de conjuntos. Subconjuntos, operaciones elementales. . 312
A.3.1. El retículo $\mathcal{P}(X)$. 313
A.3.1.1. Propiedades de la Unión. 313
A.3.1.2. Propiedades de la Intersección. 314
A.3.1.3. Propiedades Distributivas. 314
A.3.2. Leyes de Morgan. 314
A.3.3. Generalizaciones de Unión e Intersección. 314
A.3.3.1. Un número finito de uniones e intersecciones. 314
A.3.3.2. Unión e Intersección de familias cualesquiera de subconjuntos. 314
A.3.4. Conjuntos y Subconjuntos: Grafos No orientados. 315
A.4. Producto Cartesiano (`list`). Correspondencias y Relaciones. . . . 315
A.4.1. Correspondencias. 316
A.4.2. Relaciones. 317
A.4.2.1. Relaciones de Orden. 318
A.4.2.2. Relaciones de Equivalencia. 319
A.4.3. Clasificando y Etiquetando elementos: Conjunto Cociente. 319
A.5. Aplicaciones. Cardinales. 320
A.5.1. Determinismo/Indeterminismo. 321
A.5.2. Aplicaciones Biyectivas. Cardinales. 323

A.1. Introducción

A finales del siglo XIX, G. Cantor introduce la Teoría de Conjuntos. Su propósito inicial es el modesto propósito de fundamentar matemáticamente el proceso de "contar". Eso sí, no se trataba solamente de contar conjuntos finitos sino también infinitos, observando, por ejemplo, que hay diversos infinitos posibles ($\aleph_0, 2^{\aleph_0}$ o $\aleph_1$, por ejemplo). Más allá del propósito inicial de Cantor, la Teoría de Conjuntos se transformó en un instrumento útil para las Matemáticas, como un

lenguaje formal sobre el que escribir adecuadamente afirmaciones, razonamientos, definiciones, etc...
Lo que aquí se pretende no es una introducción formal de la Teoría de Conjuntos. Para ello sería necesario hacer una presentación de los formalismos de Zermelo–Frænkel o de Gödel–Bernays, lo cual nos llevaría un tiempo excesivo y sería de todo punto infructuoso e ineficaz. Al contrario, pretendemos solamente unos rudimentos de lenguaje que nos serán de utilidad durante el curso, un "apaño", para poder utilizar confortablemente el lenguaje si tener que produndizar en honduras innecesarias para la Ingeniería Informática. El recurso, también usado corrientemente en Matemáticas, es acudir a la Teoría Intuitiva de Conjuntos tal y como la concibe Haussdorff. Éste es el propósito de estas pocas páginas.

A.2. Conjuntos. Pertenencia.

Comencemos considerando los conjuntos como conglomerados de objetos. Estos objetos pasarán a denominarse *elementos* y diremos que *pertenecen a un conjunto*. Para los objetos que no están en un conjunto dado, diremos que *no pertenecen al conjunto* (o no son elementos suyos).
Como regla general (aunque con excepciones, que se indicarán en cada caso), los conjuntos se denotan con letras mayúsculas:

$$A, B, C, \ldots, X, Y, Z, A_1, A_2, ...,$$

mientras que los elementos se suelen denotar con letras minúsculas:

$$a, b, c, d, \ldots, x, y, z, a_1, a_2, ...$$

El símbolo que denota pertenencia es $\in$ y escribiremos

$$x \in A, \quad ; x \notin B,$$

para indicar "el elemento x pertenece al conjunto A" y "el elemento x no pertenece al conjunto B", respectivamente.
Hay muchas formas para definir un conjunto. Los símbolos que denotan conjunto son las dos llaves { y } y su descripción es lo que se escriba en medio de las llaves.

- *Por extensión:* La descripción de todos los elementos, uno tras otro, como, por ejemplo:

  $$X := \{0, 2, 4, 6, 8\}.$$

- *Por una Propiedad que se satisface:* Suele tomar la forma

  $$X := \{x \ : \ P(x)\},$$

  donde P es una propiedad (una fórmula) que actúa sobre la variable x. Por ejemplo, el conjunto anterior puede describirse mediante:

  $$X := \{x \ : \ [x \in \mathbb{N}] \wedge [0 \leq x \leq 9] \wedge [2 \mid x]\},$$

  donde hemos usado una serie de propiedades como $[x \in \mathbb{N}]$ (es un número natural), $[0 \leq x \leq 9]$ (entre 0 y 9), $[2 \mid s]$ (y es par). Todas ellas aparecen ligadas mediante la conectiva $\wedge$ (conjunción). Sobre la forma y requisitos de las propiedades no introduciremos grandes discusiones.

A.2.1. Algunas observaciones preliminares.

- Existe un único conjunto que no tiene ningún elemento. Es el llamado **conjunto vacío** y lo denotaremos por $\emptyset$. La propiedad que verifica se expresa (usando cuantificadores) mediante:

  $$\neg\left(\exists x, x \in \emptyset\right),$$

  o también mediante la fórmula

  $$\forall x, x \notin \emptyset.$$

- La **Estructura de Datos** relacionada con la noción de conjunto es el tipo `set`, ya visto es el curso correspondiente y que no hace sino reflejar la noción global.

A.3. Inclusión de conjuntos. Subconjuntos, operaciones elementales.

Se dice que un conjunto X está incluido (o contenido) en otro conjunto Y si *todos los elementos de X son también elementos de Y*. También se dice que X es subconjunto de Y en ese caso. Se usa el símbolo $\subseteq$ para indicar inclusión y la propiedad se "define" mediante:

$$X \subseteq Y := [\forall x, x \in X \Longrightarrow x \in Y] .$$

- Nótese la identificación entre la inclusión $\subseteq$ y la implicación $\Longrightarrow$ (o $\longrightarrow$, en la forma más convencional de la Lógica).
- Obviamente, a través de esa identificación, el conjunto vacío está contenido en cualquier conjunto. Es decir,
$$\emptyset \subseteq X,$$
para cualquier conjunto X.
- Dos conjuntos se consideran iguales si poseen los mismos elementos. En términos formales:
$$(A = B) \Leftrightarrow ((A \subseteq B) \wedge (B \subseteq A)) .$$
Lo que también puede escribirse con elementos mediante:
$$(A = B) \Leftrightarrow \forall x, ((x \in A) \Longleftrightarrow (x \in B)) .$$
- La familia de todos los subconjuntos de un conjunto X dado se denomina la *clase de partes de X* y se suele denotar mediante $\mathcal{P}(X)$.

Ejemplo A.3.1. Es fácil, por ejemplo, mostrar la siguiente igualdad que describe las partes del conjunto $X := \{0, 1, 2\}$:

$$\mathcal{P}(\{0, 1, 2\}) = \{\emptyset, \{0\}, \{1\}, \{2\}, \{0, 1\}, \{0, 2\}, \{1, 2\}, \{0, 1, 1\}\} .$$

No resulta tan fácil probar que la clase $\mathcal{P}(\mathbb{N})$ es justamente el intervalo $[0, 1]$ de la recta real $\mathbb{R}$. Lo dejamos para más tarde (en forma puramente esquemática).

Las conectivas lógicas del cálculo proposicional, permiten definir operaciones entre subconjuntos de un conjunto dado. Supongamos que tenemos un conjunto X dado y sean $A, B \in \mathcal{P}(X)$ dos de sus subconjuntos. Definimos:

- **Unión:**
$$A \cup B := \{x \in X \ : \ (x \in A) \vee (x \in B)\}.$$
- **Intersecc\'on:**
$$A \cap B := \{x \in X \ : \ (x \in A) \wedge (x \in B)\}.$$
- **Complementario:**
$$A^c := \{x \in X \ : \ x \notin A\}.$$

Obsérvese que $\emptyset^c = X$ y que $(A^c)^c = A$ para cualquier $A \in \mathcal{P}(X)$.
Adicionalmente, podemos reencontrar la diferencia entre conjuntos y la traslación del EXCLUSIVE OR (denotado por XOR en Electrónica Digital) o por $\oplus$ (en Teoría de Números, hablando de restos enteros módulo 2, i.e. $\mathbb{Z}/2\mathbb{Z}$; aunque, en este caso se suele denotar simplemente mediante $+$).

- **Diferencia:**
$$A \setminus B := \{x \in X \ : \ (x \in A) \wedge (x \notin B)\}.$$
- **Diferencia Simétrica:**
$$A\Delta B := \{x \in X \ : \ (x \in A) \oplus (x \in B)\}.$$

Las relaciones evidentes con estas definiciones se resumen en las siguientes fórmulas:

$$A \setminus B := A \cap B^c, \ \ A\Delta B = (A \cup B) \setminus (A \cap B) = (A \setminus B) \cup (B \setminus A).$$

A.3.1. El retículo $\mathcal{P}(X)$.

Sería excesivo e innecesario expresar aquí con propiedad las nociones involucradas, pero dejemos constancia de la propiedades básicas de estas operaciones:

A.3.1.1. Propiedades de la Unión.

Sean A, B, C subconjuntos de un conjunto X.

- **Idempotencia:** $A \cup A = A$, $\forall A \in \mathcal{P}(X)$.
- **Asociativa:** $A \cup (B \cup C) = (A \cup B) \cup C$, $\forall A, B, C \in \mathcal{P}(X)$.
- **Conmutativa:** $A \cup B = B \cup A$, $\forall A, B \in \mathcal{P}(X)$.
- **Existe Elemento Neutro:** El conjunto vacío $\emptyset$ es el elemento neutro para la unión:
$$A \cup \emptyset = \emptyset \cup A = A, \quad \forall A \in \mathcal{P}(X).$$

A.3.1.2. Propiedades de la Intersección.

Sean A, B, C subconjuntos de un conjunto X.

- **Idempotencia:** $A \cap A = A$, $\forall A \in \mathcal{P}(X)$.
- **Asociativa:** $A \cap (B \cap C) = (A \cup B) \cup C$, $\forall A, B, C \in \mathcal{P}(X)$.
- **Conmutativa:** $A \cap B = B \cap A$, $\forall A, B \in \mathcal{P}(X)$.
- **Existe Elemento Neutro:** El conjunto total X es el elemento neutro para la intersección:
$$A \cap X = X \cap A = A, \quad \forall A \in \mathcal{P}(X).$$

A.3.1.3. Propiedades Distributivas.

Sean A, B, C subconjuntos de un conjunto X.

- $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$, $\forall A, B, C \in \mathcal{P}(X)$.
- $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$, $\forall A, B, C \in \mathcal{P}(X)$.
- $A \cap (B\Delta C) = (A \cap B)\Delta(A \cap C)$, $\forall A, B, C \in \mathcal{P}(X)$.

A.3.2. Leyes de Morgan.

Por ser completos con los clásicos, dejemos constancia de las Leyes de Morgan. Sean A, B subconjuntos de un conjunto X.

$$(A \cap B)^c = (A^c \cup B^c), \quad (A \cup B)^c = (A^c \cap B^c).$$

A.3.3. Generalizaciones de Unión e Intersección.

Tras todas estas propiedades, dejemos las definiciones y generalizaciones de la unión e intersección en el caso de varios (o muchos) subconjuntos de un conjunto dado. Nótese la identificación entre $\cup$, $\vee$ y el cuantificador existencial $\exists$ (y, del mismo modo, la identificación entre $\cap$, $\wedge$ y el cuantificador universal $\forall$.

A.3.3.1. Un número finito de uniones e intersecciones.

Dados $A_1, \ldots, A_n$ unos subconjuntos de un conjunto X. Definimos:

$$\bigcup_{i=1}^{n} A_i := A_1 \cup A_2 \cup \cdots \cup A_n = \{x \in X \ : \ \exists i,\ 1 \leq i \leq n,\ x \in A_i\}.$$

$$\bigcap_{i=1}^{n} A_i := A_1 \cap A_2 \cap \cdots \cap A_n = \{x \in X \ : \ \forall i,\ 1 \leq i \leq n,\ x \in A_i\}.$$

A.3.3.2. Unión e Intersección de familias cualesquiera de subconjuntos.

Supongamos $\{A_i \ : \ i \in I\}$ es una familia de subconjuntos de un conjunto X. Definimos:

$$\bigcup_{i \in I} A_i := \{x \in X \ : \ \exists i \in I,\ x \in A_i\}.$$

$$\bigcap_{i \in I} A_i := \{x \in X \ : \ \forall i \in I,\ x \in A_i\}.$$

En ocasiones nos veremos obligados a acudir a uniones e intersecciones de un número finito o de un número infinito de conjuntos.

A.3.4. Conjuntos y Subconjuntos: Grafos No orientados.

Recordemos que un grafo no orientado (o simplemente un grafo) es una lista $\mathcal{G} := (V, E)$ formada por dos objetos:

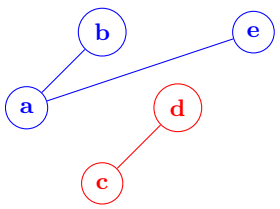
- **Vértices:** son los elementos del conjunto V (que usualmente se toma finito[1]) aunque podremos encontrar "grafos" con un conjunto "infinito" de vértices.
- **Aristas:** Es un conjunto de subconjuntos de V, es decir, $E \subseteq \mathcal{P}(V)$ con la salvedad siguiente: los elementos $A \in E$ (que, recordemos, son subconjuntos de V) son no vacíos y tienen a lo sumo dos elementos distintos.

Ejemplo A.3.2. Un sencillo ejemplo sería:

- **Vértices:** $V := \{a, b, c, d, e\}$
- **Aristas:**

$$E := \{\{a, b\}, \{a, e\}, \{c, d\}\} \subseteq \mathcal{P}(V).$$

Al ser no orientado la matriz de adyacencia es simétrica y las componentes conexas son, también, subconjuntos de V, aunque de mayor cardinal. Gráficamente:



[1] Aceptemos esta disgresión sin haber definido finitud

Nótese que podríamos haber aceptado aristas que van desde un nodo a sí mismo (tipo $\{a\}$ o $\{e\}$, por ejemplo) pero que el orden en que son descritos los elementos de una arista no es relevante: por eso hablamos de grafos no orientados.

A.4. Producto Cartesiano (`list`). Correspondencias y Relaciones.

Si en los grafos no orientados considerábamos aristas descritas de forma $\{a, b\}$ y el orden no interviene ($\{a, b\} = \{b, a\}$) ahora nos interesa destacar el papel jugado por el orden, hablamos de pares ordenados (a, b) y se corresponde al tipo de datos `list`. Así, por ejemplo, $(a, b) = (b, a)$ si y solamente si $a = b$. Una manera de representar las listas mediante conjuntos podría ser escribiendo (a, b) como abreviatura de $\{\{a\}, \{a, b\}\}$. Pero nos quedaremos con la intuición del tipo de datos `list`.

Dados dos conjuntos A y B definimos el producto cartesiano de A y B como el conjunto de las listas de longitud 2 en las que el primer elemento está en el conjunto A y el segundo en B. Formalmente,

$$A \times B := \{(a, b) \ : \ a \in A, b \in B\}.$$

Pero podemos considerar listas de mayor longitud: dados $A_1, \ldots, A_n$ definimos el producto cartesiano $\prod_{i=1}^{n} A_i$ como las listas de longitud n, en las que la coordenada $i-$ésima está en el conjunto A_i.

$$\prod_{i=1}^{n} A_i := \{(x_1, \ldots, x_n) \ : \ x_i \in A_i, \ 1 \leq i \leq n\}.$$

En ocasiones, se hacen productos cartesianos de familias no necesariamente finitas $\{A_i \ : \ i \in I\}$ (como las sucesiones, con $I = \mathbb{N}$) y tenemos el conjunto:

$$\prod_{i \in I} A_i := \{(x_i \ : \ i \in I) \ : \ x_i \in A_i, \ \forall i \in I\}.$$

En otras ocasiones se hace el producto cartesiano de un conjunto consigo mismo, mediante las siguientes reglas obvias:

$$A^1 = A, \quad A^2 := A \times A, \quad A^n := A^{n-1} \times A = \prod_{i=1}^{n} A.$$

Algunos casos extremos de las potencias puedebn ser los siguientes:

- **Caso $n = 0$:** Para cualquier conjunto A se define A^0 como el conjunto formado por un único elemento, que es el mismo independientemente de A, y se conoce con la palabra vacía y se denota por λ. No se debe confundir $A^0 := \{\lambda\}$ con el conjunto vacío $\emptyset$.

- **Caso $I = \mathbb{N}$:** Se trata de las sucesiones (infinitas numerables) cuyas coordenadas viven en A. Se denota por $A^{\mathbb{N}}$. Los alumnos han visto, en el caso $A = \mathbb{R}$ el conjunto de todas las sucesiones de números reales (que se denota mediante $A^{\mathbb{N}}$).

Observación A.4.1 (Palabras sobre un Alfabeto)**.** El conjunto de las palabras con alfabeto un conjunto A jugará un papel en este curso, se denota por A^* y se define mediante

$$A^* := \bigcup_{n \in \mathbb{N}} A^n.$$

Volveremos con la noción más adelante

A.4.1. Correspondencias.

Una correspondencia entre un conjunto A y otro conjunto B es un subconjunto R del producto cartesiano $A \times B$. En esencia es un grafo bipartito que hace interactuar los elementos de A con elementos de B. Los elementos que interactúan entre sí son aquellos indicados por los pares que están en R. En ocasiones se escribirán una notación funcional de la forma $R : A \longrightarrow B$, aunque poniendo gran cuidado porque no siempre son funciones.

Ejemplo A.4.1. Tomando $A = B = \mathbb{R}$, podemos definir la relación $R_1 \subseteq \mathbb{R}^2$ mediante:

$$R_1 := \{(x, y) \in \mathbb{R}^2 \ : \ x = y^2\}.$$

Estaremos relacionando los número reales con sus raíces cuadradas. Obsérvese que los elementos x tales que $x < 0$ no están relacionados con ningún número y (no tienen raíz cuadrada real). El 0 se relaciona con un único número (su única raíz cuadrada) y los número reales positivos se relacionan con sus dos raices cuadradas.

Ejemplo A.4.2. Tomando los mismos conjuntos $A = B = \mathbb{R}$, podemos definir la relación $R_2 \subseteq \mathbb{R}^2$ distinta de la anterior:

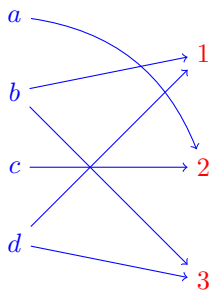
$$R_1 := \{(x, y) \in \mathbb{R}^2 \ : \ x^2 = y\}.$$

En este caso tenemos una función que relaciona cualquier x en $\mathbb{R}$ con su cuadrado.

Ejemplo A.4.3. Un grafo bipartito podría ser, por ejemplo, $A := \{a, b, c, d\}$, $B := \{1, 2, 3\}$ y una relación como $R \subseteq A \times B$:

$$R := \{(a, 2), (b, 1), (b, 3), (c, 2), (d, 1), (d, 3)\},$$

cuyo grafo sería:



Observación A.4.2. En ocasiones abusaremos de la notación, escribiendo $R(x) = y$ o xRy, para indicar que los elementos $x \in A$ e $y \in B$ están en correspondencia a través de $R \subseteq A \times B$.

A.4.2. Relaciones.

Las relaciones son correspondencia $R \subseteq A \times A$, es decir, aquellas correspondencias donde el conjunto de primeras coordenadas es el mismo que el conjunto de las segundas coordenadas.

Observación A.4.3 (Una Relación no es sino un grafo orientado.)**.** Aunque, por hábito, se suele pensar en que los grafos orientados son relaciones sobre conjuntos finitos, pero admitiremos grafos con un conjunto infinito de vértices.

Pongamos algunos ejemplos sencillos:

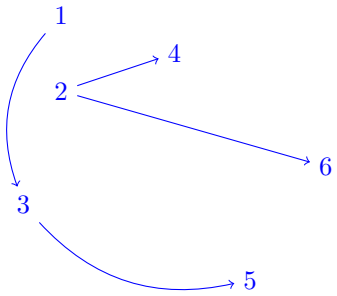
Ejemplo A.4.4 (Un ejemplo al uso)**.** Consideremos el grafo $\mathcal{G} := (V, E)$ donde V es el conjunto de vértices dado por:

$$V := \{1, 2, 3, 4, 5, 6\},$$

y $E \subseteq V \times V$ es el conjunto de aristas orientadas siguiente:

$$E := \{(1, 3), (3, 5), (2, 4), (2, 6)\}.$$

Gráficamente tendremos



Ejemplo A.4.5 (La circunferencia unidad)**.** Es un grafo infinito cuyos vértices son los números reales $V = \mathbb{R}$ y cuyas aristas son dadas mediante:

$$E := \{(x, y) \in \mathbb{R}^2 \ : \ x - y \in \mathbb{Z}\}.$$

No lo dibujaremos (tenemos demasiados vértices y demasiadas aristas) pero las componentes conexas están identicadas con los puntos de la circunferencia unidad

$$S^1 := \{(x, y) \in \mathbb{R}^2 \ : \ x^2 + y^2 - 1 = 0\}.$$

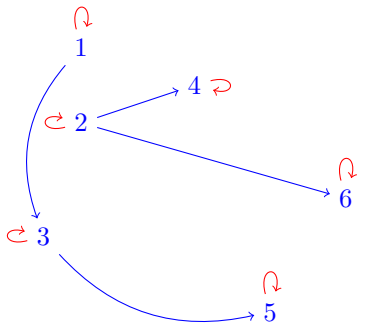
Algunos tipos de relaciones son más relevantes que otras por sus consecuencias. Destaquemos dos clases:

A.4.2.1. Relaciones de Orden.

Son aquellas relaciones $R \subseteq V \times V$, que verifican las propiedades siguientes:

- *Reflexiva:* $\forall x \in V, \ \ (x, x) \in R$. La relación descrita en el Ejemplo A.4.4 anterior no verifica esta propiedad. Para verificarla, se necesitaría que también fueran aristas las siguientes:

$$\{(1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6)\} \subseteq E.$$



  En cambio el ejemplo de la circunferencia verifica la propiedad reflexiva.

- *Antisimétrica:* Que se expresa mediante:

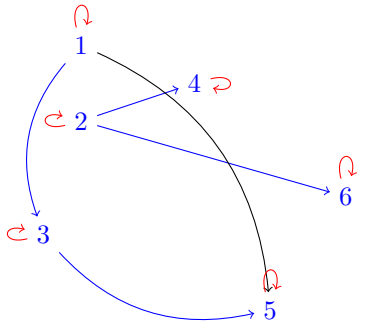
$$\forall x, y \in V, \ \ ((x, y) \in R) \wedge ((y, x) \in R) \Rightarrow (x = y)\,.$$

  La relación descrita en el Ejemplo A.4.4 sí verifica la propiedad antisimétrica porque no se da ningún caso que verifique simultáneamente las dos hipótesis. Incluso si añadimos todas las refelxivas todo funciona bien. En el ejemplo de la circunferencia, sin embargo, no se da la antisimétrica: por ejemplo 1 y 0 verifican que $(1, 0) \in R$, $(0, 1) \in R$ y, sin embargo, $1 \neq 0$.

- *Transitiva:* Que se expresa mediante:

$$\forall x, y, z \in V, \quad ((x, y) \in R) \wedge ((y, z) \in R) \Rightarrow ((x, z) \in R)\,.$$

La relación descrita en el Ejemplo A.4.4 no verifica la transitiva. Tenemos que $(1, 3) \in E$ y $(3, 5) \in E$, pero $(1, 5) \notin E$. Tendríamos que añadirla para tener un grafo como:



Este último grafo ya nos dará una relación de orden. En el ejemplo de la circunferencia, sin embargo, se da la Transitiva, aunque no es relación de orden por no satisfacerse la anti–siméttrica.

A.4.2.2. Relaciones de Equivalencia.

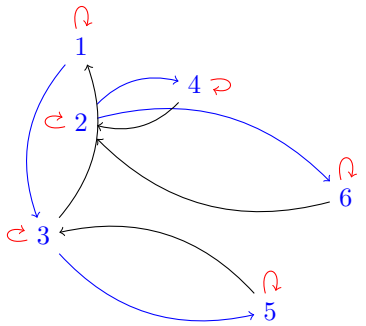
Son aquellas relaciones $R \subseteq V \times V$, que verifican las propiedades siguientes:

- *Reflexiva:* (como en el caso anterior) $\forall x \in V, \quad (x, x) \in R$. En el Ejemplo A.4.4 debemos completar nuestra lista de aristas, mientras que el ejemplo de la circunferencia ya la verifica.

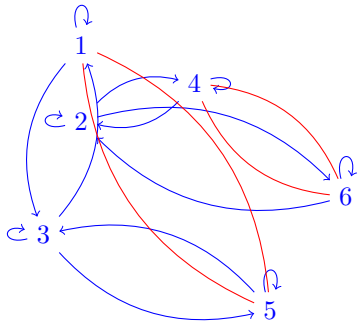
- *Simétrica:* $\forall x \in V, \quad (x, y) \in R \Leftrightarrow (y, x) \in R$. En el caso de la circunferencia ya se satisface. Mientras que en el caso del Ejemplo A.4.4 debems completar nuestra lista, añadiendo las aristas:

$$\{(3, 1), (5, 3), (4, 2), (6, 2)\},$$

para que se satisfaga. Esto nos da un grafo como:



- *Transitiva:* Que se expresa como ya se ha indicado. Es claro que el caso de la circunferencia tenemos una relación de equivalencia y en el caso del Ejemplo A.4.4 habrá que completar con todos los casos. Esto nos dará una figura como la siguiente:



Este último grafo ya nos dará una relación de equivalencia.

A.4.3. Clasificando y Etiquetando elementos: Conjunto Cociente.

Mientras las relaciones de orden pretenden establecer una preferencia de unos elementos sobre otros, dentro de un cierto conjunto, las relaciones de equivalencia pretenden clasificar los elementos del mismo conjunto para, posteriormente etiquetarlos. Se llamará conjunto cociente al conjunto de etiquetas finales.
El término etiqueta no es tan espúreo dado que las etiquetas son lo que definen, de manera bastante desafortunada en ocasiones, cosas tan dispares como la sociedad de consumo o la claisificación e Linneo de los seres vivos, por ejemplo.
Así, tomemos un conjunto X y una relación de equivalencia $\sim \subseteq X \times X$ definida sovre él. Para un elementos $x \in X$, consideraremos su clase de equivalencia como el conjunto formado por todos los elementos de X equivalentes a x, es decir,

$$[x] := \{y \in X \ : \ x \sim y\}.$$

Las clases son sunconjuntos de X y se verifican las siguientes tres propiedades que indican que se trata de una partición de X:

- $X = \bigcup_{x \in X} [x]$,
- $[[x] \cap [y] = \emptyset] \vee [[x] = [y]]$.
- $[x] \neq \emptyset$.

Así, retomando los ejemplos, podemos clasificar un colectivo X de personas (los habitantes de una ciudad, por ejemplo) mediante la relación de equivalencia *"$x \sim y$ si y solamente si [x tiene el mismo modelo de coche que y]"*. Se trata claramente de una relación de equivalencia sobre X. La relación no es fina como clasificador puesto que hay individuos que poseen más de un coche y, por tanto, más de un modelo, con lo que podríamos tener que un "Dacia" y un BMW están relacionados. Admitamos que la relación se refina mediante *"$x \sim y$ si y solamente si [x e y poseen un mismo modelo de coche y ambos le prefieren entre los de su propiedad]"*.
Ciertamente cada clase de euivalencia recorre todos los individuos de la una ciudad que poseen el mismo modelo de coche. Así, podríamos tener la clase [Luis], formada por todas las personas que no tienen coche o [Juan] formada por todas las personas que tienen un Dacia Logan del 96. De hecho, la etiqueta del coche define la clase. Podríamos usar el símbolo $\emptyset$ para describir la clase de quienes poseen ningún coche y el símbolo TT para quienes posean un Audi TT. Recíprocamente, en la sociedad de consumo, la publicidad no nos vende el coche que sale en un anuncio sino todos los coches equivalentes a él, es decir, todos los que tienen las mismas caracteísticas de los que fabrica esa empresa...Es lo que se llama "Marca" o "etiqueta" y es lo que los ciudadanos de las sociedades llamadas avanzadas compran.
En la clasificación de Linneo también tenemos una relación de equivalencia, esta vez entre todos los seres vivos. Dos seres vivos serían equivalentes si pertenecen al mismo Reino, Orden, Familia, Género, Especie....Luego se imponen las etiquetas. Así, la *rana muscosa* esla etiqueta que define la

clase de equivalencia de todas las ranas de montaña de patas amarillas y no distingue entre ellas como individuos: una de tales ranas pertenece a la clase (etiqueta) pero ella no es toda la clase. En tiempos más recientes, el afán clasificatorio de Linneo se reconvierte en el afán clasificatorio de los genetistas: dos seres vivos son equivalentes si poseen el mismo sistema cromosósico (mapa genético), quedando el código genético como etiqueta individual.
Con ejemplos matemáticos, es obvio que en un grafo no orientado, las clases de equivalencia son las clausuras transitivas (o componentes conexas) de cada elemento. En el caso de los número racionales, por ejemplo, la clase de equivalencia de 2/3 está formada por todos los pares de números enteros a/b, con $b \neq 0$, tales que $2b = 3a$.
Una vez queda claro que disponemos de clases de equivalencia, podemos considerarlas como elementos. Nace así el conjunto cociente que es el conjunto formado por las clases de equivalencia, es decir,

$$X/\sim := \{[x] \ : \ x \in X\}.$$

En los ejemplos anteriores, el conjunto cociente es el conjunto de las etiquetas de coches, el conjunto de los nombres propuestos por Linneo para todas las especies de animales, etc. Nótese que el conjunto cociente es algo que, en muchas ocasiones, se puede escribir (por eso el término etiqueta) aunque hay casos en los que los conjuntos cocientes no son "etiquetables" en el sentido de formar un lenguaje. El ejemplo más inmediato es el caso de los números reals $\mathbb{R}$ que son las etiquetas de las clases de equivalencia de sucesiones de Cauchy, pero que no son expresabels sobre un alfabeto finito.

A.5. Aplicaciones. Cardinales.

Las aplicaciones son un tipo particular de correspondencias.

Definición A.5.1 (Aplicaciones)**.** Una aplicación entre dos conjuntos A y B es una correspondencia $R \subseteq A \times B$ que verifica las propiedades siguientes:

- Todo elemento de A está realcionado con algún elemento de B:

$$\forall x \in A, \ \exists y \in B, \ (x, y) \in R.$$

- No hay más de un elemento de B que pueda estar relacionado con algún elemento de A:

$$\forall x \in A, \forall y, y' \in B, \ ((x, y) \in R) \wedge ((x, y') \in R) \Longleftarrow y = y'.$$

En ocasiones se resume con la expresión:

$$\forall x \in A, \exists \mid y \in B, \ \ (x, y) \in R,$$

donde el cuantificador existencial modificado $\exists \mid$ significa "existe uno y sólo uno".

Notación A.5.1. Notacionalmente se expresa $R : A \longrightarrow B$, en lugar de $R \subseteq A \times B$ y, de hecho, se suelen usar letras minúsculas del tipo $f : A \longrightarrow B$ para indicar la aplicación f de A en B. Al único elemento de B relacionado con un $x \in A$ se le representa $f(x)$ (es decir, escribimos $x \longmapsto f(x)$ en lugar de $(x, f(x)) \in f$).
Por simplicidad, mantendremos la notación (inadecuada, pero unificadora) $f : A \longrightarrow B$ también para las correspondencias, indicando en cada caso si hacemos referencia a una aplicación o a una correspondencia, y reservaremos la notación $R \subseteq A \times A$ para las relaciones.

Ejemplo A.5.1 (Aplicación (o función) característica de un subsonjunto)**.** Sea X un conjunto $L \subseteq X$ un subconjunto. De modo natural tenemos definda una aplicación que toma como entradas los elementos de X y depende fuertemente de L: el la función característica

$$\chi_L : X \longrightarrow \{0, 1\}$$

y que viene definida para cada $x \in X$ mediante:

$$\chi_L(x) := \begin{cases} 1, & \text{si } \ x \in L \\ 0, & \text{en otro caso-} \end{cases}$$

Se usa la expresión función cuando se trata de aplicaciones $f : \mathbb{R}^n \longrightarrow \mathbb{R}$, expresión que viene de la tradición del Análisis Matemático.

Definición A.5.2 (Composición)**.** Dados tres conjuntos A, B y C y dos aplicaciones $f : A \longrightarrow B$, $g : B \longrightarrow C$, podemos definir una aplicacin (llamada composición de f y g) que denotaremos $g \circ f : A \longrightarrow C$ y viene definida por la regla:

$$x \longmapsto g(f(x)),\ \forall x \in A,$$

es decir, "primero aplicamos f sobre x y luego aplicamos g a $f(x)$".

Para una aplicación (o correspondencia) $f : A \longrightarrow A$ podemos definir la potencia mediante:

$$\begin{array}{ll} f^0 := & Id_A, \ \text{(la identidad)}, \\ f^1 := & f, \\ f^n := & f^{n-1} \circ f, \ \ \forall n \geq 2. \end{array}$$

A.5.1. Determinismo/Indeterminismo.

A partir de una aplicación (o correspondencia) $f : A \longrightarrow A$, podemos definir una estructura de grafo orientado "natural", definiendo los vértices como los elementos de A y las aristas mediante

$$V := \{(x, f(x)) \ : \ x \in A\}.$$

En algunos casos, los alumnos habrán llamdo a este conjunto de vértices *"el grafo de la función f"*.
Dentro de ese grafo orientado, podemos considerar la parte de la "componente conexa" de x que son descendientes de x. Este conjunto vendrá dado por las iteraciones de f, es decir:

$$\{, x, f(x), f^2(x), f^3(x), \ldots, f^n(x), \ldots\}.$$

La diferencia entre el hecho de ser f aplicación o correspondencia se traduce en términos de "determinismo" o "indeterminismo":

- En el caso de ser aplicación, el conjunto de sucesores es un conjunto, posiblemente, infinito, en forma de camino (un árbol sin ramificaciones):

  $$x \longrightarrow f(x) \longrightarrow f^2(x) \longrightarrow f^3(x) \longrightarrow \cdots .$$

  Se dice que (A, f) tiene una dinámica *determinista*.

- En el caso de ser correspondencia, el conjunto de los sucesores de x toma la forma de árbol (con posibles ramificaciones): algunos valores no tendrán descendientes y otros tendrán más de un descendiente directo. Se dice que (A, f) tiene una dinámica *indeterminista*.

Ejemplo A.5.2. Tomemos $A := \mathbb{Z}/5\mathbb{Z} := \{0, 1, 2, 3, 4\}$, las clases de restos módulo 5 y consideremos $f := A \longrightarrow A$, dada mediante:

$$x \longmapsto f(x) = x^2, \forall x \in A.$$

Es una aplicación, por lo que los descendientes de cada valor $x \in A$ forman un camino (un árbol sin ramificaciones). Por ejemplo, $\{0\}$ es el conjunto de todos los descendientes de 0, mientras que, si empezamos con 3 tendremos:

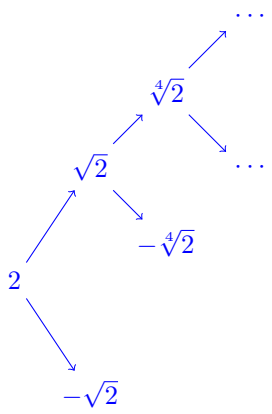
$$3 \longmapsto f(3) = 4 \longmapsto f^2(3) = 1 \longmapsto f^3(3) = 1 \longmapsto 1 \longmapsto \cdots ,$$

COmo f es aplicación tendremos, para cada $x \in A$ una dinámica determinista.

Ejemplo A.5.3. Un ejemplo de indeterminismo sería $A = \mathbb{R}$ y la correspondencia:

$$R := \{(x, y) \ : \ x = y^2\}.$$

En este caso, si $x < 0$ no hay descendientes, si $x = 0$ hay solamente un deecendiente, y si $x > 0$ tenemos una infinidad de descendientes en forma de árbol no equilibrado. Por ejemplo,



Los vértices $-\sqrt{2}, -\sqrt[4]{2}, \ldots$ no tendrán descendientes, mientras los positivos tienen un par de descendientes directos.
Debe señalarse que este ejemplo muestra un indeterminismo fuertemente regular (sabemos la regla) pero, en general, el indeterminismo podría presentar una dinámica muy impredecible.

A.5.2. Aplicaciones Biyectivas. Cardinales.

Sólo un pequeño resumen del proceso de contar el número de elementos de un conjunto, noción que preocupaba originalmente a G. Cantor.

Definición A.5.3 (Aplicaciones inyectivas, suprayectivas, biyectivas)**.** Sea $f : A \longrightarrow B$ una aplicación.

- Decimos que f es inyectiva si verifica:

$$\forall x, x' \in A, \ \ (f(x) = f(x')) \Longrightarrow x = x'.$$

- Decimos que f es suprayectiva si verifica:

$$\forall y \in B, \exists x \in A, \ \ f(x) = y.$$

- Decimos que f es biyectiva si es, a la vez, inyectiva y suprayectiva.

Obsérvese que una aplicación $f : A \longrightarrow B$ es biyectiva si y solamente si disponemos de una aplicación (llamada inversa de f) que se suele denotar por $f^{-1} : B \longrightarrow A$ y que satisface:

$$f \circ f^{-1} = Id_B, \ \ f^{-1} \circ f = Id_A,$$

donde $\circ$ es la composición y Id_A e Id_B son las respectivas aplicación identidad en A y en B.
En general las aplicaciones no tienen inversa, es decir, no podemos suponer siempre que sean biyectivas.
El proceso de contar no es sino la fundamentación del proceso "infantil" de contar mediante identificación de los dedos de las manos con los objetos a contar. Este proceso es una biyección.

Definición A.5.4 (Cardinal)**.** Se dice que dos conjuntos A y B tienen el mismo cardinal (o número de elementos) si existe una biyección $f : A \longrightarrow B$. También se dice que son biyectables.

- Un conjunto A se dice finito si existe un número natural $n \in \mathbb{N}$ y una biyección
$$f : A \longrightarrow \{1, 2, 3, \ldots, n\}.$$
Por abuso de lenguaje se identifican todos los conjuntos del mismo cardinal y escribiremos, en el caso finito, $\sharp(A) = n$, cuando A sea biyectable a $\{1, 2, \ldots, n\}$.
- Un conjunto A se dice (infinito) numerable si hay una biyección $f : A \longrightarrow \mathbb{N}$
- Un conjunto se dice contable si es finito o numerable.

Proposición A.5.1. *Si dos conjuntos A e B son biyectables, también son biyectables $\mathcal{P}(A)$ y $\mathcal{P}(B)$ (i.e. , las familias de sus subconjuntos).*

Demostración. Baste con disponer de una biyección $f : A \longrightarrow B$ para poder definir:
$$\widetilde{f} : \mathcal{P}(A) \longrightarrow \mathcal{P}(B),$$
dada mediante:
$$X \longmapsto \widetilde{f}(X) := \{f(x) \in B \ : \ x \in X\} \subseteq B.$$
La inversa de esta transformación será:
$$\widetilde{f}^{-1} : \mathcal{P}(B) \longrightarrow \mathcal{P}(A),$$
dada mediante
$$Y \longmapsto \widetilde{f}^{-1}(Y) := \{x \in A \ : \ f(x) \in Y\}.$$
□

Usualmente se utiliza la notación $f(X)$ y $f^{-1}(Y)$ en lugar de $\widetilde{f}(X)$ y $\widetilde{f}^{-1}(Y)$, usadas en la prueba anterior.
Algunos cardinales y propiedades básicas:

i) Los conjuntos $\mathbb{N}, \mathbb{Z}, \mathbb{Q}$ son conjuntos numerables, mientras que $\mathbb{R}$ o $\mathbb{C}$ son conjuntos infinitos (no son finitos) y son no numerables (no son biyectables con $\mathbb{N}$).

ii) Los subconjuntos de un conjunto finito son también finitos. Entre los subconjuntos A, B de un conjunto finito se tiene la propiedad
$$\sharp(A \cup B) + \sharp(A \cap B) = \sharp(A) + \sharp(B).$$

iii) Los subconjuntos de un conjunto contable son también contables.

iv) Si A y B son finitos tendremos:
$$\sharp\,(A \times B) = \sharp(A)\sharp(B).$$

v) Si A es un conjunto finito, el cardinal de $\mathcal{P}(A)$ (el número de todos sus subconjuntos) es dado por
$$\sharp\,(\mathcal{P}(A)) = 2^{\sharp(A)}.$$

vi) Si A es un conjunto finito, el número $\sharp(\mathcal{A})$ de aplicaciones $f : A \longrightarrow \{0, 1\}$ verifica:
$$\sharp(\mathcal{A}) = \sharp\,(\mathcal{P}(A)) = 2^{\sharp(A)}.$$

vii) Si A es un conjunto finito,
$$\sharp(A^n) = (\sharp(A))^n\,.$$
Por ejemplo, si K es un cuerpo finito de la forma $K := \mathbb{Z}/p\mathbb{Z}$, donde $p \in \mathbb{N}$ es un número primo, el cardinal
$$\sharp(K^n) = \sharp(K)^n,$$
por lo que se tiene que para cada espacio vectorial V de dimensión finita sobre un cuerpo K finito se tiene:
$$\dim V = \log_{\sharp(K)} \sharp(V).$$

viii) Si A es un conjunto finito $\sharp(A) = n$, el número de permutaciones (es decir, biyecciones de A en sí mismo) es $n!$. Además, el número de subconjuntos de cardinal k de A es dado por el número combinatorio:

$$\binom{n}{k} := \frac{n!}{k!(n-k)!}.$$

De ahí que se tenga:

$$2^n := \sum_{k=0}^{n} \binom{n}{k}.$$

Algunas propiedades elementales de los cardinables contables se resumen en:

Proposición A.5.2. *Productos finitos de conjuntos contables es un conjunto contable. Es decir, dados $\{A_1, \ldots, A_n\}$ una familia finita de conjuntos contables, entonces el producto cartesiano $\prod_{i=1}^{n} A_i$ es también contable.*
La unión numerable de conjuntos contables es contable, es decir, dados $\{A_n \ : \ n \in \mathbb{N}\}$ una familia numerable de conjuntos, de tal modo queo cada A_n es contable, entonces, tambiés es contable el conjunto:

$$A := \bigcup_{n\in\mathbb{N}} A_n.$$

Si A es un conjunto contable (finito o numerable), el conjunto de palabras A^* también es contable.

Ejemplo A.5.4 (Los subconjuntos de $\mathbb{N}$)**.** Por lo anterior, los subconjuntos de $\mathbb{N}$ son siempre conjuntos contables (finitos o numerables) pero la cantidad de subconjuntos de $\mathbb{N}$ es infinita no numerable (es decir, el cardinal de $\mathcal{P}(\mathbb{N})$ es infinito no numerable). Para comprobarlo, vamos a mostrar una biyección entre $\mathcal{P}(\mathbb{N})$ y el intervalo $[0,1] \subseteq \mathbb{R}$ de números reales. Nótese que el cardinal del intervalo $[0,1]$ es igual al cardinal de los números reales. Usaremos la función característica asociada a cada subconjunto $L \subseteq \mathbb{N}$. Así, dado $L \in \mathcal{P}(\mathbb{N})$, definiremos el número real:

$$L \longmapsto x_L := \sum_{i=1}^{\infty} \frac{\chi_L(i)}{2^i} \in [0,1].$$

Nótese que el número real asociado al conjunto vacío $\emptyset$ es el número real $x_\emptyset = 0$, mientras que el número real $x_\mathbb{N} \in [0,1]$ es precisamente $x_\mathbb{N} = 1 \in [0,1]$.
Recíprocamente, dado cualquier número real $x \in [0,1]$, éste posee una única expansión "decimal" en base dos (para ser más correcto, digamos, una única expansión binaria):

$$x := \sum_{i=1}^{\infty} \frac{x_i}{2^i}.$$

Definamos el sunconjunto $L_x \subseteq \mathbb{N}$ mediante:

$$L_x := \{i \in \mathbb{N} \ : \ x_i = 1\}.$$

Ambas aplicaciones ($x \longmapsto L_x$ y $L \longmapsto x_L$) son una inversa de la otra y definen biyecciones entre $[0,1]$ y $\mathcal{P}(\mathbb{N})$ y recíprocamente).
Dejamos al lector el esfuerzo de verificar que hay tantos número reales (en todo $\mathbb{R}$) como número reales en el intervalo $[0,1]$.

Bibliografía

[AdHu, 92] L.M. Adleman, M.-D- A- Huang, Primality testing and Abelian varieties over finite fields. Springer, 1992.

[AKS, 04] M. Agrawal, N. Kayal, N. Saxena, *PRIMES is in P*. Annals of Mathematics **160**, (2), (2004), 781793.

[AhHoUl, 75] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algotrithms. Addison-Wesley (1975).

[AhUl, 72a] A.V. Aho, J.D. Ullman, The Theory of Parsing, Translation and Compiling. Vol I: Parsing. Prentice Hall, 1972.

[AhUl, 72b] A.V. Aho, J.D. Ullman, The Theory of Parsing, Translation and Compiling. Vol II: Compilers. Prentice Hall, 1972.

[AaUl, 95] A.V. Aho, J.D. Ullman, Foundations of Computer Science. W. H. Freeman (1995).

[Al, 07] . Alfonseca, Teoría De Autómatas y Lenguajes Formales. McGraw-Hill, 2007.

[AlMi, 85] N. Alon, V. D. Milman, λ_1, *isoperimetric inequalities for graphs, and superconcentrators*. J. Combin. Theory Ser. B, **38** (1985), 73-88.

[ALMSS, 98] S. Arora, C. Lund, R. Motwani, M. Sudan, M. Szegedy, *Proof verification and the hardness of approximation problems*. J. of the Assoc. Comput. Mach. **45** (1998), 501-555.

[AS, 98] S. Arora, S. Safra, *Probabilistic checking of proofs: A new characterization of* **NP**. J. of the Assoc. Comput. Mach. **45** (1998), 70-122.

[ArBa, 09] S. Arora, B. Barak, Computational Complexity (A Modern Approach). Cambridge University Press, 2009.

[BaFoLu, 90] L. Babai, L. Fortnow, C. Lund, *Nondeterministic exponential time has two-prover interactive protocols*. In Proc. of the 31st Annual Symp. Found. of Comput. Sci. (FoCS), IEEE Comput. Soc., 1990, 16-25.

[BaSh, 96] E. Bach, J. Shallit, Algorithmic Number Theory. Vol 1 : Efficient Algorithms. MIT Press, 1996.

[BaDiGa, 88] J.L. Balcazar, J.L. Díaz and J. Gabarró, Structural Complexity I. EATCS Mon. on Theor. Comp. Sci. **11**, Springer (1988).

[BaDiGa, 90] J.L. Balcázar, J.L. Díaz and J. Gabarró, Structural Complexity II. EATCS Mon. on Theor. Comp. Sci., Springer, 1990.

[BaGiSo, 75] T. Baker, J. Gill y R. Solovay, *Relativizations of the* **P** *=?* **NP** *question*. SIAM J. on Comput. **4** (1975), 431-442.

[Ba (ed.), 77] J. Barwise (ed.), Handbook of Mathematical Logic. Noth-Holland, 1977.

[BePa, 11s] C. Beltrán y L. M. Pardo, *Efficient Polynomial System Solving by Numerical Methods*, In "Randomization, Relaxation, and Complexity in Polynomial Equation Solving", L. Gurvits, P. Pébay, J.M. Rojas, D. Thompson, eds., Contemporary Mathematics, vol. **556**, Amer. Math. Soc., 2011, 1-35.

[Bl, 67] M. Blum, *A machine independent theory of the complexity of recursive functions.* Journal of the Assoc. Comput. Mach. **14**, N.2, (1967), 322-336.

[BlSuShSm, 96] L. Blum, F. Cucker, M. Shub, S. Smale, *Algebraic settings for the problem* $\mathbf{P}\neq\mathbf{NP}$*?*. In The mathematics of numerical analysis (Park City, UT, 1995), Amer. Math. Soc., Providence, 1996, 125-144.

[BlCuShSm, 98] L. Blum, F. Cucker, M. Shub, S. Smale, Complexity and real computation. Springer-Verlag, New York, 1998.

[BuMü, 95] J. Buchmann, V. Müller, *Algorithms for Factoring Integers.* Preprint Universität des Saarlandes, 1995(?).

[Bus, 92] P. Buser, *A note on the isoperimetric constant.* Ann. Sci. École Norm. Sup. **15** (1982), 213-230.

[Ca, 95] T. Cahill, How The Irish Saved Civilization: The Untold Story of Ireland's Heroic Role from the Fall of Rome to the Rise of Medieval Europe. Doubledya, New York, 1995.

[CaHaMoPa, 01] D. Castro, K. Hägele, J.E. Morais, L. M. Pardo, *Kronecker's and Newton's approaches to solving: a first comparison*, J. of Complexity **17** (2001), 212-303.

[Che, 70] J. Cheeger, *A lower bound for the smallest eigenvalue of the Laplacian.* In *Problems in analysis (Papers dedicated to Salomon Bochner, 1969)*, Princeton Univ. Press, Princeton, N. J., 1970, 195-199.

[Che, 74] A. Chenciner, Géometrie Algébrique Elementaire. Cours de Maitrise, Université de Paris VII, 1974/75.

[Ch, 57] N. Chomsky, Syntactic Structures. Mouton and Co., The Hague, 1957.

[ChMi, 57] N. Chomsky, G. A. Miller, *Finite state languages.* Information and Control **1** (1957), 91-112.

[Ch, 59a] N. Chomsky, *On certain formal properties of grammars.* Information and Control **2** (1959), 137-167.

[Ch, 59b] N. Chomsky, *A note on phrase structure grammars.* Information and Control **2** (1959), 393-395.

[Ch, 62] N. Chomsky *Context-free grammarsand pushdown storage.* Quarterly Progress Report No. **65**. Research Laboratory of Electronics, M. I. T., Cambridge, Mass., 1962.

[Ch, 65] N. Chomsky, *Three models for the description of language.* IEEE Trans. on Information Theory **2** (1965), 113-124.

[Chu, 35] A. Church, *An unsolvable problem of elementary number theory (abstract).* Bull. Amer. Math. Soc. **41** (1935), 332-333.

[Chu, 36] A. Church, *An unsolvable problem of elementary number theory.* Amer. J. Math. **58** (1936), 345-363.

[Co, 65] A. Cobham. *The intrinsic computational difficulty of functions.* In Proc. Logic, Methodology, and Philosophy of Science II (Proc. 1964 Internat. Congr.), North Holland (1965) 2430.

[CoSc, 70] . Cocke, J.T. Shwartz, Programming Languages and their Compilers. Courant Institute of MathematicalSciences, NYU, 1970.

[Coh, 93] H. Cohen, A Course in Computational Algebraic Number Theory. GTM **138**, Springer, 1993.

[Cook, 71] S. Cook, *The complexity of theorem-proving procedures.* In Proc. 3rd Ann. ACM SToC, ACM Press, 1971, 151-158.

[Danzig, 79] G.B. Dantzig,Linear Programming and Extensions. Princeton, NJ: Princeton University Press, 1963.

[RED, 89] R.E. Davis, Truth, Deduction and Computation(Logic and Semantics for Computer Science). W.H. Freeman, 1989.

[MaD, 82] M. Davis, Computability and Unsolvability. Dover, 1982.

[MaD, 97] M. Davis, *Unsolvable Problems.* Handbook of Mathematical Logic, North-Holland, 1997, 567-594.

[DaWe, 94] M.D. Davis, E.J.Weyuker, Computability, Complexity, and Languages (Fundamentals of Theoretical Computer Science), 2nd Ed., Academic Press, 1994.

[DiHe, 76] W. Diffie, M.E. Hellman, *New directions in cryptography.* IEEE Trans. Information Theory IT-22, no. 6, (1976), 644-654.

[Di, 07] I. Dinur, *The* **PCP** *theorem by gap amplification.* J. of the Assoc. Comput. Mach. **54**, vol 3 (2007), Art. 12.

[Do, 84] J. Dodziuk, *Difference equations, isoperimetric inequality and transience of certain random walks.* Trans. Amer. Math. Soc. **284** (1984), 787-794.

[Du, 87] D. Duval, Diverses Questiones Relatives au Calcul Formel avec des Nombres Algébriques. Thése d'Etat, Université de Grenoble, 1987.

[Ed, 65a] J. Edmonds,*Minimum partition of a matroid into independent sets.* J. of Res. of the Nat. Bureau of Standards (B) **69** (1965) 67-72.

[Ed, 65b] J. Edmonds, *Maximum mathcing and a polyhedron with 0,1-vertices.* J. of Res. of the Nat. Bureau of Stand. (B) **69** (1965), 125-130.

[Eil, 74] S. Eilenberg, Automata,Languages and Machines, vol. A. Academica Press, Pure and App. Math. **59**-A, 1974.

[Eins, 1850] G. Einsenstein, *Beweiss der Allgemeisten Reciprocitätgesetze zwischen reellen und komplexen Zahlen.* Verhandlungen der Königlick Pr. Akad. des Wissen. sur Berlin (1850), 189-198.

[End, 77] H.B. Enderton, *Elements of Recursion Theory.*In Handbook of Mathematical Logic, J. Barwise (ed.), North-Holland, 1977.

[FoLuKa, 90] L. Fortnow, C. Lund, H. Karloff, *Algebraic methods for interactive proof systems.* J. of the Assoc. Comput. Mach. **39** (1992), 859-868. Anunciado, con N. Nisan de co-autor adicional, en Proc. of 31st Symp. Found. of Comput. Sci., IEEE, New York, 1990, pp. 290.

[GabGal, 81] O. Gabber y Z. Galil, *Explicit constructions of linear-sized superconcentrators.* J. Comput. System Sci. **22**(1981), 407-420.

[GaJo, 79] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completness. W.H. Freeman, 1979.

[GatGe, 99] J. von zur Gathen, J. Gerhard, Modern Computer Algebra. Cambridge University Press, 1999.

[Gauss. 1801] Carolo Federico Gauss, Disquisitiones Aithmeticae. LIPSIAE in Commissis apud Gerh. Fleiser, Jun. 1801.

[Gödel, 31] K. Gödel, *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I.* Monatsh. Math. Phys **38** (1931), 173-198.

[Gödel, 65] K. Gödel, *On undecidable propositions of formal mathematical systems.* In The undecidable, Basic Papers on Undecidable Propositions, Unsolvable Problemas and Computable Functions, Raven Press, Hewlett, NY, 1965, 41-71.

[Go, 99] O. Goldreich, Modern Cryptography, Probabilistic Proofs and Pseudo-Randomness. Algorithmc and Combinatorics **17**, Springer, 1999.

[Go, 08] O. Goldreich, Computational Complexity: A Conceptual Approach. Cambridge University Press, 2008.

[HarWri, 60] G. Hardy, E. Wright, The Theory of Numbers. Oxford Univ. Press, 4th. ed., 1960.

[HaSt, 65] J. Hartmanis, R. Stearns. *On the computational complexity of algorithms.* Trans. of the A.M.S. **117** (1965), 285-306.

[HaLeSt, 65] J. Hartmanis, P. M. Lewis II, R. E. Stearns.*Hierarchies of memory limited computations.* In Proc. 6th Annual IEEE Symp. on Switching Circuit Theory and Logical Design, 1965, 179-190.

[HeSc, 82] J. Heintz, C.P. Schnorr, *Testing polynomials wich are easy to compute.* In In Logic and Algorithmic (an International Symposium in honour of Ernst Specker), L'Enseignement Mathématique **30** (1982), 237-254.

[HeSt, 66] F. Hennie, R. Stearns, *Two-tape simulation of multitape Turing machines.* J. of the Assoc. Comput. Mach. **13** (1966), 533-546.

[He, 26] G. Hermann, *Die Frage der endlich vielen Schritte in der Theorie der Polynomideale.* Math. Ann. **95** (1926), 736-788.

[HoMoUl, 07] J. E. Hopcroft, R. Motwani, J. D. Ullman. Introduction to Automata Theory, Languages, and Computation, 3/Ed. Addison-Wesley, 2007.

[HoLiWi, 06] S. Hoory, N. Linial, A. Widgerson, *Expander graphs and their applications.* Bull. (New ser.) of the Amer. Math. Soc. **43** (2006), 439561.

[IbMo, 83] O.H. Ibarra, S. Moran, *Equivalence of Straight-Line Programs.* J. of the Assoc. Comput. Mach. **30** (1983), 217-228.

[Jac, 1837] Carl Gustac Jacob Jacobi, Über die Kreisteilung und ahre Anwendung auf die Zahlentheorie. Bericht Akademi Wissen. Berlin, 1837, 127-136.

[Krp, 72] R. Karp, *Reducibility among combinatorial problems.* In Complexity of Computer Computations, (R.E. Miller & J.W. Hatcher, eds.), Plenum Press, 1972, 85-103.

[KrpLi, 80] R. Karp y J. Lipton, *Some connections between nonuniform and uniform complexity classes.* In Proc. of the 12th Annual ACM Symp. Theor. of Compu., 1980, 302309.

[Kha, 79] L. G. Khachiyan, *A polynomial algorithm in linear programming* (in Russian). Doklady Akademiia Nauk SSSR, **224** (1979), 1093-1096. English Translation: Soviet Mathematics Doklady, Volume **20**, 191-194.

[KlMi, 72] V. Klee, G.J. Minty, *How Good is the Simplex Algorithm?*. In Inequalities **3**, O. Shisha, editor, New York: Academic Press, 1972, 159-175.

[Kl, 36] S.C. Kleene, *$\lambda-$definability and recursiveness*. Duke Math. J. **2** (1936), 340-353.

[Kl, 52] S.S. Kleene. INTRODUCTION TO METAMATHEMATICS. Van Nostrand Reinhold, New York, 1952.

[Kl, 56] S.S. Kleene. *Representation of events in nerve nets*. In Automata Studies, Shannon and McCarthy eds. Princeton University Press, Princeton, N.J., 1956, 3-40.

[Kn, 97] D.E. Knuth, THE ART OF COMPUTER PROGRAMMING (2ND ED.), VOL. 2 SEMINUMERICAL ALGORITHMS. Addison-Wesley, 1997-98.

[Ko, 92] D.C. Kozen. THE DESIGN AND ANALYSIS OF ALGORITHMS. Texts and Monographs in Computer Science, Springer Verlag, 1992.

[KrPa, 96] T. Krick, L.M. Pardo, *A Computational Method for Diophantine Approximation*. In Algorithms in Algebraic Geometry and Applications, Proc. MEGA'94, Progress in Mathematics **143**, Birkhäuser Verlag, 1996, 193-254.

[Ku, 64] S.Y. Kuroda, *Classes of Languages and Linear-Bounded Automata*. Information and Control **7** (1964), 207-223.

[Lad, 75] R. Ladner, *On the Structure of Polynomial Time Reducibility*. Journal of the ACM (JACM) **22** (1975), 155171.

[La, 1985] S. Landau, *Factoring Polynomials over algebraic number fields*. SIAM J. Comput. **14** (1985), 184-195.

[LaMi, 85] S. Landau, G. L. Miller, *Solvability by Radicals is in Polynomial Time*. J. of Comput. and Systems. Sci. **30** (1985), 179-208.

[Law, 63] P.S. Landweber, *Three theorems on phrae structure grammars of type 1*. Information and Control **6** (1963), 131-137.

[Lau, 83] C. Lautemann, **BPP** *and the polynomial hierarchy*. Inf. Proc. Lett. **14** (1983), 215217.

[HB, 92] Van Leeuwen, J. (ed.).HANDBOOK OF THEORETICAL COMPUTER SCIENCE. Elsevier, 1992.

[Leg, 1798] A.M. Legendre, ESSAI SUR LA THÉORIE DES NOMBRES. Paris, 1798.

[Lev, 73] L.A. Levin. *Universal search problems*. Probl. Pred. **7** (1973), 115-116. (English tranlation in Proble. Inf. Trans. **9** (1973), 265-266).

[LePa, 81] H.L. Lewis, C.H. Papadimitriou, ELEMENTS OF THE THEORY OF COMPUTATION. Prentice-Hall, 1981.

[LuPhSa, 88] A. Lubotzky, R. Phillips, P. Sarnak, *Ramanujan graphs*. Combinatorica **8** (1988), 261277.

[Lup, 58] O.B. Lupanov, *A method of circuit synthesis*. Izves. VUZ, Radiofizika **1** (1958), 120-140.

[Mart, 03] J. Martin, INTRODUCTION TO LANGUAGES AND THE THEORY OF COMPUTATION, 3RD EDITION. McGraw Hill, 2003.

[Marc, 67] S. Marcus, ALGEBRAIC LINGUISTICS; ANALYTIC MODELS. Mathematics in Science and Engineering, vol. **29**, Academic Press, 1967.

[Marg, 73] G. A. Margulis, *Explicit constructions of expanders*. Problemy Peredaci Informacii **9** (1973), 71-80.

[Mign, 89] M. Mignotte, MATHÉMATIQUES POUR LE CALCUL FORMEL. Presses Univ. de France, 1989.

[Mi, 76] G.L. Miller, *Riemann's hypothesis and tests for primality*. J. Comput. Syst. Sci. **13** (1976), 300-317.

[MoRa, 95] R. Motwani, P. Raghavan, RANDOMIZED ALGORITHMS, Cambridge University Press, 1995.

[My, 60] J. Myhill, *Linear Bounded Automata*. WADD Tech. Note No. 60-165, Wright-Patterson Air Force Base, Ohio, 1960.

[Papa, 94] C. H. Papadimitrou, COMPUTATIONAL COMPLEXITY. Addison-Wesley, 1994.

[Pa, 95] L.M. Pardo, *How lower and upper complexity bounds meet in elimination theory*. In Proc. 11th International Symposium Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, AAECC-11, Paris 1995, G. Cohen, M.Giusti and T. Mora, eds., Springer Lect. Notes Comput. Sci. **948**, 1995 33-69.

[Pa, 12] L.M. Pardo, *La Conjetura de Cook ('?***P** = **NP***?). Parte I: Lo Básico*. La Gaceta de la RSME **15** (2012), 117-147.

[PoZas, 89] M. Pohst, H. Zassenhaus, ALGORITHMIC ALGEBRAIC NUMBER THEORY. Cambridge Univ. Press, 1989.

[Po, 75] J.M. Pollard, *A Monte Carlo Method for Factorization*. BIT **15** (1975), 331-334.

[Pr, 75] V.R. Pratt, *Every Prime has a succinct certificate*. SIAM J. on Comput. **4** (1975), 214-220.

[Ra, 60] M. O. Rabin. *Degree of difficutly of computing a function and a partial ordering of recursive sets*. Tech. Rep. No. **2**, Hebrew University, 1960.

[Ra, 66] M. O. Rabin. *Mathematical theory of automata*. In Proc. of 19th ACM Symposium in Applied Mathematics, 1966, 153-175.

[Ra, 80] M.O. Rabin. *Probabilistic algorithms for testing primality*. J. Number Theory **12** (1980), 128-138.

[RdSu, 07] J. Radhakrishnan, M. Sudan, *On Dinur's proof of the* **PCP***-Theorem*. Bull. of the Amer. Math. Soc. **44** (2007), 19-61.

[ReVaWi, 02] O. Reingold, S. Vadhan, A. Wigderson, *Entropy waves, the zig-zag graph product, and new constant-degree expanders*. Ann. of Math. **155** (2002), 157187.

[Ri, 87] K. Ríbnikov, HISTORIA DE LAS MATEMÁTICAS. Mir, 1987.

[Rie, 85] H. Riesel, PRIME NUMBERS AND COMPUTER METHODS FOR FACTORIZATION. Birkhäuser, 1985.

[RSA, 78] R.L. Rivest, A. Shamir, L.A. Adleman, *A method for obtaining digital signatures and public-key cryptosystems*. Comm. ACM **21**, no. 2 ,(1978,) 120-126.

[Ro, 94] H.E. Rose, A COURSE IN NUMBER THEORY, 2nd. ed.. Oxford Sci. Publications, 1994.

[Sa, 04] P. Sarnak, *What is an Expander?*. Notices of the Amer. Math. Soc. **51** (2004), 762-763.

[Sh, 92] A. Shamir, **IP** = **PSPACE**. J. of the Assoc. Comput. Mach. **39** (1992), 869-877.

[Sha, 49] C.E. Shannon, *The synthesis of two-terminal switching circuits.* Bell System Technical J. **28** (1949), 5998.

[Sa, 70] W.J. Savitch, *Relationships between nondeterministic and deterministic tape complexities.* J. Comput. System. Sci. **4** (1970), 177-192.

[Schö, 79] A. Schönhage, *On the power of random access machines.* In Proc. of the 6th Colloquium on Automata, Languages and Programming, H. A. Maurer (ed.), Lect. Notes Compu. Sci. **71**, Springer, 1979, 520-529.

[SchöVe, 94] A. Schönhage, E. Vetter, FAST ALGORITHMS. A MULTITAPE TURING MACHINE IMPLEMENTATION. Wissenschaftverlag, 1994.

[Schwa, 80] J.T. Schwartz, *Fast Probabilistic Algorithms for Verification of Polynomial Identities.* J. of the ACM **27**, (1980), 701-717.

[Si, 83] M. Sipser, *A complexity theoretic approach to randomness.* In Proc. of the 15th ACM Symp. Theor. of Comput., 1983, 330-335.

[Si, 97] M. Sipser (1997), INTRODUCTION TO THE THEORY OF COMPUTATION. PWS Publishing, 1997.

[SoSt, 77] R. Solovay, V. Strassen, *A fast Monte Carlo test for primality.* SIAM J. on Comput. **6** (1977), 84-85.

[St, 89] I. Stewart, GALOIS THEORY (2nd Edition). Chapman and Hall, 1989.

[Turing, 37] A. M. Turing, *On Computable Numbers, with an Application to the Entscheidungsproblem.* Proceedings of the London Mathematical Society, Series 2, **42** (1936-37) pp. 23065. See also A.M. Turing, *On Computable Numbers, with an Application to the Entscheidungsproblem: A correction.* Proceedings of the London Mathematical Society. Series 2, **43** (1937), pp. 5446.

[Val, 76] L. Valiant, *Graph-theoretic properties in computational complexity.* J. Comput. Syst. Sci. **13** (1976), 278-285.

[Va, 94] A. Valibouze, THÉORIE DE GALOIS CONSTRUCTIVE. Mémoire d'Habilitation, Univ. Pierre et Marie Curie, 1994.

[Va, 93] A. Valibouze. "Résolvantes de Lagrange". *Rapport interne LITP* **93.61** (1993).

[vdW, 49] B.L. van der Waerden, MODERN ALGEBRA (vols. 1 y 2). F. Ungar, 1949-5).

[Vrd,97] A. Vardy.*Algorithmic Complexity in Coding Theory and the Minimum Distance Problem.* In Proc. STOC'97, 1997, 92-109.

[WaWe, 86] K. Wagner, G. Wechsung, COMPUTATIONAL COMPLEXITY. D. Reidel, 1986.

[Wal, 50] R.J. Walker, ALGEBRAIC CURVES. Dover, 1950.

[Weh, 97] K. Weihrauch, COMPUTABILITY. EATCS monographs on Theor. Comp. Sci. **9**, Springer Verlag, 1987.

[Wi, 96] N. Wirth, COMPILER CONSTRUCTION. Addison-Wesley International Computer SCience Service, 1996.

[Zi, 90] R. Zippel, *Interpolating Polynomials from their Values.* J. Symbol. Comput. **9** (1990), 375-403.

[Zi, 93] R. Zippel, EFFECTIVE POLYNOMIAL COMPUTATION. Kluwer Academic Publishers, 1993.