Isabel Navarrete Sánchez
María Antonia Cárdenas Viedma
Daniel Sánchez Alvarez
Juan Antonio Botía Blaya
Roque Marín Morales
Rodrigo Martínez Béjar

Departamento de Ingeniería de la Información
y las Comunicaciones
Universidad de Murcia

TEORÍA DE AUTÓMATAS

Y

LENGUAJES FORMALES

Introducción

Aunque no debemos hacer una distinción tajante entre los aspectos prácticos y teóricos de la Informática, es cierto que existen materias que tienen un alto contenido formal, con desarrollos de tipo matemático, al contrario que otros temas más cercanos a la resolución de problemas de tipo práctico. La asignatura de *Teoría de Autómatas y Lenguajes Formales* sin duda trata con las materias del primer tipo y los contenidos que se imparten constituyen el eje fundamental de diversas áreas de conocimiento encuadradas dentro de lo que podríamos denominar Informática Teórica. A veces estas disciplinas resultan para el alumno materias "áridas" y distanciadas de lo que ellos entienden que deberían estudiar en una carrera de Ingeniería Informática. Pero la Informática, como cualquier otra ciencia o ingeniería, tiene unos fundamentos teóricos sobre los que apoyarse y que cualquier ingeniero en Informática debe conocer. Así lo entienden diversos organismos internacionales como ACM e $IEEE$ que recomiendan al menos un curso de Autómatas y Lenguajes Formales en los curricula de las carreras relacionadas con la Informática. Una motivación para el estudio de estas materias formales la expuso Millner en un discurso que dio en 1993 al recoger el prestigioso *premio Turing* que se otorga a distinguidos científicos que trabajan en el área de las Ciencias de la Computación:

> "Estas [las aplicaciones] son altamente necesarias, pero no queremos que esto ocurra en detrimento del trabajo teórico...Las Ciencias de la Computación son tan amplias que si no tienen una teoría básica, estaremos perdidos. Tantas cosas están avanzando...¿Cómo podría ocurrir esto sin una teoría? Esta tiene que ir cogida de la mano de la práctica."

1. Evolución histórica de la Teoría de la Computación

La Teoría de la Computación trata con *modelos de cálculo abstractos* que describen con distintos grados de precisión las diferentes partes y tipos de computadores. Pero estos modelos no se usan para describir detalles prácticos del hardware de un determinado ordenador, sino que más bien se ocupan de cuestiones abstractas sobre la capacidad de los ordenadores, en general. Así, en los *curricula* de Ciencias de la Computación existen cursos separados para tratar materias como Arquitectura de Computadores, Teoría de Circuitos, Algoritmos y Estructuras de Datos, Sistemas Operativos, etc. Todas estas áreas tienen una componente teórica, pero difieren del estudio de la Teoría de la Computación fundamentalmente en dos aspectos:

- Las primeras tratan con computadores que existen realmente, mientras que los modelos abstractos de cálculo abarcan todo tipo de computadores que existen, que puedan llegar a existir o simplemente que uno pueda imaginar.

- En Teoría de la Computación, a diferencia de las otras materias, lo importante no es buscar la mejor manera de hacer las cosas (*optimalidad*) sino estudiar qué puede o no puede hacerse con un ordenador (*computabilidad*).

La historia de la Teoría de la Computación es bastante interesante. Se ha desarrollado gracias a confluencia, por afortunadas coincidencias, de distintos campos de conocimiento y descubrimientos (fundamentalmente matemáticos) realizados a principios del siglo XX. Bajo el nombre Teoría de la Computación se recogen una serie de materias que constituyen hoy en día los fundamentos teóricos de la Informática: *Teoría de Autómatas*, *Teoría de los Lenguajes Formales*, *Computabilidad* y *Complejidad Algorítmica*.

Computabilidad

El primer tema que cae claramente dentro del campo de la Teoría de la Computación es el de Computabilidad. Iniciada por Gödel, Church, Post, Turing y Kleene, tiene sus raíces en la *Lógica Matemática*. Al iniciar el siglo XX, los matemáticos estaban a punto de efectuar grandes descubrimientos. Los logros de los siguientes 40 años estaban destinados a sacudir las bases de las matemáticas y tuvieron consecuencias que se extendieron al campo de las *Ciencias de la Computación*, aún por nacer.
A principios de siglo XX se empezó a fraguar un dilema. Georg Cantor (1845-1918), había inventado por entonces la *Teoría de Conjuntos*, pero al mismo tiempo descubrió algunas paradojas inquietantes. Algunos de sus planteamientos podían ser comprensibles (como que hay "infinitos" de distinto tamaño), pero otros no (por ejemplo, que algún conjunto sea mayor que el conjunto universal). Esto dejó una nube de duda a los matemáticos que ellos necesitaban disipar. El punto de partida de fueron las cuestiones fundamentales que David Hilbert (1845-1918) formuló en 1928, durante el transcurso de un congreso internacional:

1. ¿Son *completas* las Matemáticas, en el sentido de que pueda probarse o no cada aseveración matemática?

2. ¿Son las Matemáticas *consistentes*, en el sentido de que no pueda probarse simultáneamente una aseveración y su negación?

3. ¿Son las Matemáticas *decidibles*, en el sentido de que exista un método definido que se pueda aplicar a cualquier aseveración matemática y que determine si dicha aseveración es cierta o falsa?

La meta de Hilbert era crear un sistema axiomático lógico-matemático *completo y consistente*, del cual podrían deducirse todas las Matemáticas, esto es, cualquier teorema matemático podría derivarse de los axiomas aplicando una serie finita de reglas, es decir, mediante un *proceso algorítmico* o *computacional*. Su idea era encontrar un algoritmo que determinara la verdad o falsedad de cualquier teorema en el sistema formal. A este problema le llamó el '*Entscheidungsproblem*'.
Por desgracia para Hilbert, en la década de 1930 se produjeron una serie de investigaciones que mostraron que esto no era posible. Las primeras noticias en contra surgen en 1931 con Kurt Gödel (1906-1978) y su *Teorema de Incompletitud*: "Todo sistema de primer orden consistente que contenga los teoremas de la aritmética y cuyo conjunto de axiomas sea recursivo no es completo". Como consecuencia no será posible encontrar el sistema formal deseado por Hilbert en el marco de la lógica de primer orden. Una versión posterior y más general del teorema de Gödel elimina la posibilidad de considerar sistemas deductivos más potentes que los sistemas de primer orden, demostrando que no pueden ser consistentes y completos a la vez. Los resultados de Gödel prueban que no sólo no existe un algoritmo que pueda demostrar todos los teoremas en matemáticas, sino que además, no todos los resultados son demostrables. Entonces cabe plantearse las siguientes preguntas:

- ¿Qué pueden hacer los ordenadores (sin restricciones de ningún tipo)?
- ¿Cuales son las limitaciones inherentes a los métodos automáticos de cálculo?

A estas cuestiones pretende responder la Teoría de la Computabilidad. El primer paso en la búsqueda de las respuestas a estas preguntas está en el estudio de los modelos de computación. Los Modelos Abstractos de Cálculo tienen su origen en los años 30, antes de que existieran los ordenadores (el primer computador electrónico de propósito general fue el ENIAC que se desarrolló a partir del año 1943), en el trabajo de los lógicos Church, Gödel, Kleene, Post, y Turing. Estos primeros trabajos han tenido una profunda influencia no sólo en el desarrollo teórico de las Ciencias de la Computación, sino que muchos aspectos de la prácticos de la Informática fueron presagiados por ellos: incluyendo la existencia de ordenadores de propósito general, la posibilidad de interpretar programas, la dualidad entre software y hardware y la representación de lenguajes por estructuras formales basados en reglas de producción.

Alonzo Church propuso la noción de *función λ-definible* como función efectivamente calculable. La demostración de teoremas se convierte en una transformación de una cadena de símbolos en otra, según un conjunto de reglas formales, que se conocen como *lambda cálculo*. En 1936, Church hace un esquema de la demostración de la equivalencia entre las funciones λ-definibles y las funciones recursivas de Herbrand-Gödel (esta equivalencia también había sido probada por Kleene) y conjetura que éstas iban a ser las únicas funciones calculables por medio de un *algoritmo* a través de la tesis que lleva su nombre (**Tesis de Church**) y utilizando la noción de función λ-definible, dio ejemplos de problemas de decisión irresolubles y demostró que el *Entscheidungsproblem* era uno de esos problemas.

Por otra parte Kleene, pocos meses después, demuestra de forma independiente la equivalencia entre funciones λ-definibles y funciones recursivas de Herbrand-Gödel, a través del concepto de *función recursiva* y da ejemplos de problemas irresolubles.

La tercera noción de función calculable proviene del matemático inglés Alan Turing (1912-1954). Turing señaló que había tenido éxito en caracterizar de un modo matemáticamente preciso, por medio de sus máquinas, la clase de las funciones calculables mediante un algoritmo (*funciones Turing-computables*), lo que se conoce hoy como **Tesis de Turing** (1936). Aunque no se puede dar ninguna prueba formal de que una *máquina de Turing* pueda tener esa propiedad, Turing dio un elevado número de argumentos a su favor, en base a lo cual presentó la tesis como un teorema demostrado. Además, utilizó su concepto de máquina para demostrar que existen problemas que no son calculables por un método definido y en particular, que el *Entscheidungsproblem* era uno de esos problemas. Cuando Turing conoció los trabajos de Church y Kleene, demostró que los conceptos de función λ-definible y función calculable por medio de una máquina de Turing coinciden. Naturalmente a la luz de esto la Tesis de Turing resulta ser equivalente a la de Church.

Posteriormente, se demostró la equivalencia entre lo que se podía calcular mediante una máquina de Turing y lo que se podía calcular mediante un sistema formal en general. A la vista de estos resultados, la **Tesis de Church-Turing** es aceptada como un axioma en la Teoría de la Computación y ha servido como punto de partida en la investigación de los problemas que se pueden resolver mediante un algoritmo.

Una de las cuestiones más estudiadas en la Teoría de la Computabilidad ha sido la posibilidad de construir programas que decidan si un determinado algoritmo posee o no una determinada propiedad. Sería interesante responder de forma automática a cuestiones como:

- ¿Calculan los algoritmos A y B la misma función? (*Problema de la equivalencia*)
- ¿Parará el algoritmo A para una de sus entradas? (*Problema de la parada*)
- ¿Parará el algoritmo A para todas sus entradas? (*Problema de la totalidad*)
- ¿Calcula el algoritmo A la función f? (*Problema de la verificación*)

Conforme se fueron obteniendo demostraciones individuales de la no computabilidad de cada una de estas cuestiones, fue creciendo la sensación de que casi cualquier pregunta interesante acerca

de algoritmos era no computable. El Teorema de Rice, confirma esta sensación: "Considérese cualquier propiedad que no sea trivial acerca de la función calculada por un algoritmo, entonces la cuestión de si la función calculada por un algoritmo arbitrario verifica dicha propiedad es no computable".

Complejidad Algorítmica

Después de que la Teoría de la Computabilidad fuera desarrollada, era natural preguntarse acerca de la dificultad computacional de las funciones computables. Este es el objetivo de la parte de las Ciencias de la Computación que se conoce como Complejidad Algorítmica. Rabin fue uno de los primeros en plantear esta cuestión general explícitamente: ¿Qué quiere decir que una función f sea más difícil de computar que otra función g? Rabin sugirió una axiomática que fue la base para el desarrollo del estudio de *medidas de complejidad abstracta* de Blum y otros (1967).

Una segunda aportación que tuvo una influencia relevante en el desarrollo posterior de esta materia fue el artículo de J. Hartmanis y R. Stearns en 1965, cuyo título *On the Complexity of Algorithms* dio nombre a este cuerpo de conocimiento. En él se introduce la noción fundamental de *medida de complejidad* definida como el tiempo de computación sobre una máquina de Turing multicinta y se demuestran los teoremas de jerarquía.

Un tercer hito en los comienzos del tema fue el trabajo de Cobham titulado, *The Intrinsic Computational Difficulty of Functions* (1964). Cobham enfatizó el término "intrínseco", es decir, él estaba interesado en una teoría independiente de las máquinas. Esto nos conduce al un concepto importante desarrollado en 1965: la identificación de la clase de problemas que se pueden resolver en tiempo acotado por un polinomio sobre la longitud de la entrada. La distinción entre *algoritmos de tiempo polinomial* y *algoritmos de tiempo exponencial* fue hecha por primera vez en 1953 por Von Neumann. La notación de P para la clase de los problemas resolubles en tiempo polinomial fue introducida posteriormente por Karp (1972).

La teoría de la *NP-completitud* es seguramente el desarrollo más importante de la Complejidad Algorítmica. La clase NP consta de todos los problemas decidibles en tiempo polinomial por una máquina de Turing no determinista. Cook en 1971 introduce la noción de *problema NP-completo* y demuestra que el problema de la satisfacibilidad booleana es NP-completo. La clase NP incluye una gran cantidad de problemas prácticos que aparecen en la actividad empresarial e industrial. Demostrar que un problema es NP-completo equivale a demostrar que no tiene una solución determinista en tiempo polinomial, salvo que todos los problemas de NP estén en P, cuestión que aún no está demostrada.

Otro área que actualmente está teniendo cada vez más importancia es la **Criptografía**, relacionada con la seguridad de los sistemas informáticos y donde se ha aplicado especialmente la teoría de la complejidad algorítmica. Mediante la criptografía podemos conseguir el manejo de información confidencial en el ordenador de forma más o menos segura.

Máquinas Secuenciales y Autómatas Finitos

La Teoría de Autómatas, que engloba también al estudio de las *Máquinas secuenciales*, tiene su origen en el campo de la *Ingeniería Eléctrica*. El matemático norteameriacano Shanon (que luego se haría famoso por su *Teoría de la Información*) vino a establecer las bases para la aplicación de la Lógica Matemática a los circuitos combinatorios y posteriormente Huffman en 1954 los amplió a circuitos secuenciales y utiliza conceptos como *estado* de un autómata y *tabla de transición*. A lo largo de las décadas siguientes, las ideas de Shanon se desarrollaron considerablemente, dando lugar a la formalización de una Teoría de las Máquinas Secuenciales y de los Autómatas Finitos (1956). Otros trabajos importantes sobre máquinas secuenciales son debidos a Mealy (1955) y Moore.

Desde un frente totalmente distinto, el concepto de *autómata finito* aparece en 1943 con el artículo de de McCulloch y Pitts titulado *A Logical Calculus of the Ideas Immanet in Nervous Activity*, donde describen los cálculos lógicos inmersos en un dispositivo (*neurona artificial*) que habían ideado para simular la actividad de una neurona biológica. A partir de entonces, se han desarrollado asociaciones de neuronas para constituir redes. Podemos considerar una RNA (*Red Neural Artificial*) como una colección de procesadores elementales (neuronas), conectadas a otras neuronas o entradas externas, y con una salida que permite propagar las señales por múltiples caminos. Cada procesador pondera las entradas que recibe y estos pesos pueden ser modificados en aras de conseguir el objetivo previsto. Es lo que llamaremos *función de aprendizaje*. Es decir, una RNA puede "aprender" de sus propios errores, por un proceso inductivo a partir de un conjunto de ejemplos de lo que queremos aprender, frente al proceso deductivo, propio de los *Sistemas Expertos*. Las características que hacen interesantes a las RNAs son su capacidad para *aprender* (reproducir un sistema o función a partir de ejemplos), *memorizar* (almacenar un conjunto de patrones o ejemplos), *generalizar* y *abstraer* (que permita recuperaciones a partir de entradas defectuosas o incompletas). Las redes neuronales, dentro del perfil de Teoría de la Computación, aportan paradigmas interesantes como son el *cálculo paralelo*, el *aprendizaje inductivo* y su capacidad para realizar *cálculos aproximados* por medio de interpolación.

En el verano de 1951 Kleene fue invitado por la *RAND Corporation* para realizar un informe sobre los trabajos de McCulloch-Pitts. En este informe Kleene demuestra la equivalencia entre lo que él llama "dos formas de definir una misma cosa": los *conjuntos regulares*, los cuales pueden ser descritos a partir de sucesos bases y los operadores unión, concatenación y clausura, es decir, mediante *expresiones regulares* y los lenguajes reconocidos por un autómata finito.

Los autómatas finitos son capaces de reconocer solamente un determinado tipo de lenguajes, llamados *lenguajes regulares*, que también se caracterizan mediante un tipo de gramáticas llamadas así mismo regulares. Una forma adicional de caracterizar este tipo de lenguajes es mediante las citadas expresiones regulares, construidas mediante operadores sobre el alfabeto del lenguaje y otras expresiones regulares, incluyendo el lenguaje vacío. Es fácilmente comprobable que, para un alfabeto concreto, no todos los lenguajes que se pueden construir son regulares. Ni siquiera todos los interesantes desde el punto de vista de la construcción de algoritmos para resolver problemas. Hay entonces muchos problemas que no son calculables con estos lenguajes. Esto pone de manifiesto las limitaciones de los autómatas finitos y las gramáticas regulares, y propicia el desarrollo de máquinas reconocedoras de otros tipos de lenguajes y de las gramáticas correspondientes asociadas a los mismos, como veremos en el siguiente apartado.

Desde su nacimiento, la Teoría de Autómatas ha encontrado **aplicación** en campos muy diversos. ¿Qué tienen en común? A primera vista no parece sencillo deducirlo. Sin embargo, podemos vislumbrar la solución si nos damos cuenta de que en todos ellos se manejan conceptos como el 'control', la 'acción', la 'memoria' y además, los objetos controlados o recordados son símbolos, palabras o frases de algún tipo. Algunos de los campos donde ha encontrado aplicación la Teoría de Autómatas son:

- Teoría de la Comunicación.
- Teoría de Control.
- Lógica de Circuitos Secuenciales.
- Reconocimiento de Patrones.
- Fisiología del Sistema Nervioso.
- Estructura y Análisis de los Lenguajes de Programación.
- Traducción Automática de Lenguajes.

- Teoría Algebraica de Lenguajes.

Cuando un autómata se usa para modelar la construcción de hardware (ej. circuitos secuenciales) o software (ej. analizadores léxicos) es muy importante examinar el problema de encontrar el *autómata mínimo* equivalente a uno dado. Tanto Huffman como Moore se ocuparon de este problema y encontraron algoritmos prácticos para minimizar un autómata de estados finitos. Para un autómata de n estados estos algoritmos requerían n^2 pasos. Bastante más tarde, en 1971 Hopcroft encontró un método que lo hacía en $O(n \times log(n))$ pasos. Existe un punto de vista algebraico sobre la minimización y caracterización de autómatas finitos, debida a John Myhill y Anil Nerode. Kleene, en su intento de entender los trabajos de McCullock y Pitts, abstrajo el concepto de autómata finito a partir de las redes de neuronas y el concepto de expresión regular a partir del cálculo lógico del modelo de McCullock y Pitts. De la misma forma, Myhill a partir de los conceptos de autómatas finitos de Kleene obtuvo el de *diagrama de transición* (deterministas) y a los eventos los redujo a la unión de clases de equivalencia. Siguiendo esta línea de trabajo, se ha elaborado en las últimas décadas una teoría abstracta de autómatas con una fuerte base matemática que, según dijo Arbib en 1969, constituye "la matemática pura de la Informática".

Gramáticas y Lenguajes Formales

El desarrollo de los ordenadores en la década de los 40, con la introducción de los programas en la memoria principal y posteriormente con los lenguajes de programación de alto nivel, propician la distinción entre *lenguajes formales*, con reglas sintácticas y semánticas rígidas, concretas y bien definidas, de los lenguajes naturales como el inglés, donde la sintaxis y la semántica no se pueden controlar fácilmente. Los intentos de formalizar los *lenguajes naturales* llevan a la construcción de *gramáticas* como una forma de describir estos lenguajes, utilizando para ello reglas de producción para construir las frases del lenguaje. Se puede entonces caracterizar un lenguaje mediante las reglas de una gramática adecuada.
Noam Chomsky propone en 1956 tres modelos para la descripción de lenguajes, que son la base de su futura jerarquía de los tipos de lenguajes (1959), que ayudó también en el desarrollo de los lenguajes de programación. Chomsky estableció una clasificación de gramáticas de acuerdo con el formato de sus producciones y distinguió cuatro clases fundamentales de lenguajes y relaciones de inclusión entre ellas.
La Teoría de los Lenguajes Formales resultó tener una relación sorprendente con la Teoría de Autómatas y la Computabilidad. Paralelamente a la jerarquía de lenguajes existe otra equivalente de máquinas abstractas, de tal forma que a cada una de las clases de lenguajes definidas en la jerarquía de Chomsky a partir de restricciones impuestas a las gramáticas, le corresponde un tipo de máquina abstracta, que no es otra cosa que un método reconocedor para la descripción de lenguajes. La relación la podemos observar en la figura 1. Cada uno de estos tipos de máquinas es capaz de resolver problemas cada vez más complejos, desde los autómatas finitos (que son los más simples) hasta las máquinas de Turing que determinan el límite de los procesos computables. Se puede llegar así, de una forma casi natural, a considerar las máquinas de Turing, establecidas casi 20 años antes, como máquinas reconocedoras de los lenguajes estructurados por frases (tipo 0) e incluso a interpretar la Tesis de Turing en términos de que un sistema computacional nunca podrá efectuar un análisis sintáctico de aquellos lenguajes que están por encima de los lenguajes estructurados por frases en la jerarquía de Chomsky.

2. Fundamentos Matemáticos

A continuación haremos un repaso breve sobre varias ideas matemáticas que serán utilizadas en los próximos capítulos. Estos conceptos incluyen conjuntos, relaciones, funciones y técnicas de

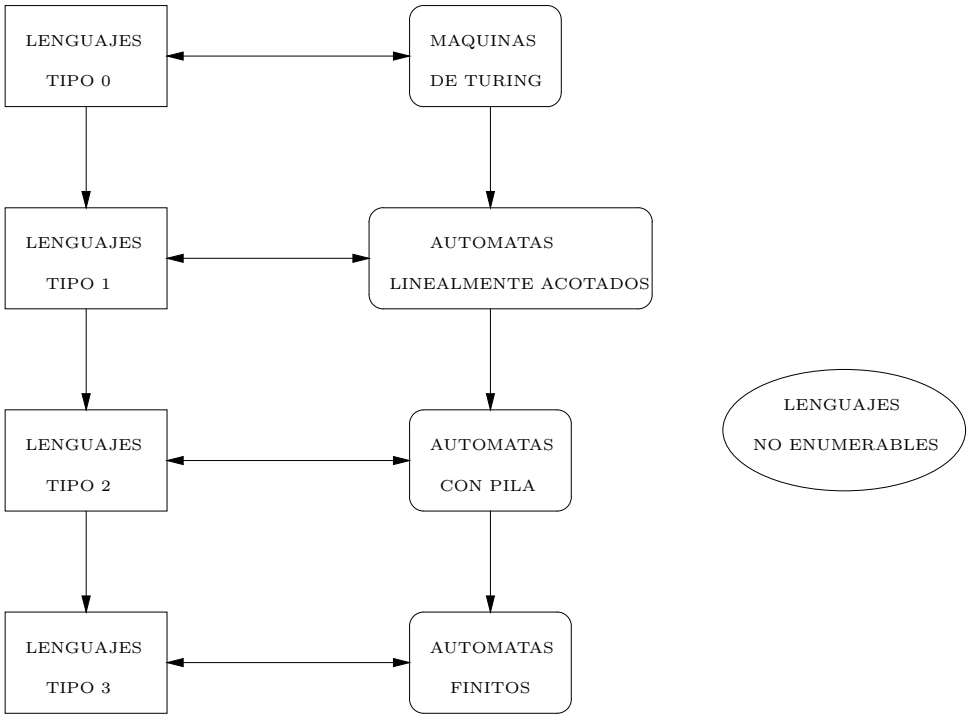


Figura 1: Relación Lenguajes-Máquinas Abstractas

demostración matemáticas.

Conjuntos

Un conjunto es una colección de objetos. Por ejemplo, la colección de las letras vocales forman un conjunto que podemos notar como $V = \{a, e, i, o, u\}$. Los objetos que forman parte del conjunto se llaman *elementos*. Por ejemplo, a es un elemento de V y se escribe $a \in V$; por otra parte podemos decir que $z \notin V$. Dos conjuntos son *iguales* si y sólo si tienen los mismos elementos. No se tienen en cuenta las repeticiones de elementos ni tampoco el orden de éstos. Hay un conjunto que no tiene ningún elemento llamado *conjunto vacío* y lo notaremos por $\varnothing$. Un conjunto se puede especificar enumerando sus elementos entre llaves y separados por comas y esto es lo que se llama *definición por extensión*. Pero a veces esto no es posible hacerlo porque el conjunto es infinito y entonces se usa una *definición por comprensión*, es decir, haciendo referencia a otros conjuntos (*conjuntos referenciales*) y a propiedades que los elementos puedan tener. De forma general se definen:

$$B = \{x \in A \mid x \text{ cumple la propiedad P}\}$$

Un conjunto A es un *subconjunto* de otro conjunto B, $A \subseteq B$, si cada elemento de A es un elmento de B. También podemos decir que A está *incluido* en B. Cualquier conjunto es un subconjunto de sí mismo. Si A es un subconjunto de B pero A no es igual a B se dice que A es un *subconjunto propio* de B y se nota como $A \subset B$. Es obvio que $\emptyset \subseteq A$ para cualquier conjunto A. Para probar que dos conjuntos A y B son iguales debemos probar que $A \subseteq B$ y $B \subseteq A$: cada elemento de A debe ser un elemento de B y viceversa.
Dos conjuntos se pueden combinar para formar un tercero mediante una serie de OPERACIONES SOBRE CONJUNTOS:

unión	$A \cup B = \{x \mid (x \in A) \vee (x \in B)\}$
intersección	$A \cap B = \{x \mid (x \in A) \wedge (x \in B)\}$
diferencia	$A - B = \{x \mid (x \in A) \wedge (x \notin B)\}$

Algunas PROPIEDADES de las operaciones anteriores se pueden deducir fácilmente a partir de sus definiciones:

1. **Idempotencia:** $A \cup A = A \quad ; \quad A \cap A = A$
2. **Conmutatividad:** $A \cup B = B \cup A \quad ; \quad A \cap B = B \cap A$
3. **Asociatividad:** $\begin{array}{l}(A \cup B) \cup C = A \cup (B \cup C) \\ (A \cap B) \cap C = A \cap (B \cap C)\end{array}$
4. **Distributividad:** $\begin{array}{l}A \cup (B \cap C) = (A \cup B) \cap (A \cup C) \\ A \cap (B \cup C) = (A \cap B) \cup (A \cap C)\end{array}$
5. **Absorción:** $A \cap (A \cup B) = A \quad ; \quad A \cup (A \cap B) = A$
6. **Leyes de DeMorgan:** $\begin{array}{l}\overline{A \cap B} = \overline{A} \cup \overline{B} \\ \overline{A \cup B} = \overline{A} \cap \overline{B}\end{array}$

Dos conjuntos son *disjuntos* si no tienen elementos en común, o lo que es lo mismo, si su intersección es el conjunto vacío. Es posible formar intersecciones y uniones de más de dos conjuntos.
La colección de todos los subconjuntos de A es a su vez un conjunto llamado **conjunto potencia** de A y lo notamos como 2^A . Al conjunto potencia de A también se le suele llamar *conjunto de las partes de* A y se nota como $\mathcal{P}(A)$.

Ejemplo 0.1 *Sea* $A = \{c, d\}$. *Entonces* $2^A = \{\emptyset, \{c\}, \{d\}, \{c, d\}\}$

Una **partición** de un conjunto no vacío A es un subconjunto, Π, de 2^A tal que:

1. cada elemento de Π es no vacio;
2. los elementos de Π son disjuntos;
3. $\bigcup \Pi = A$

Ejemplo 0.2 $\{\{a, b\}, \{c\}, \{d\}\}$ *es una partición de* $\{a, b, c, d\}$ *pero* $\{\{a, b, c\}, \{c, d\}\}$ *no lo es. Los conjuntos de números pares e impares forman una partición de* $\mathbb{N}$.

Relaciones y funciones

De forma general podemos definir una *relación* como un conjunto de elementos , que son en esencia combinaciones de objetos de un determinado tipo que están relacionados de alguna forma. Llamamos *par ordenado* a una pareja de objetos escritos entre paréntesis y separados por comas. Por ejemplo, (a, b) es un par ordenado y a, b son los *componentes* del par ordenado. No es lo mismo (a, b) que $\{a, b\}$ por varios motivos:

- el orden influye: no es lo mismo (a, b) que (b, a), sin embargo $\{a, b\} = \{b, a\}$
- los dos componentes de un par ordenado no tienen porqué ser distintos; por ejemplo, $(2, 2)$ es un par válido.

El **producto cartesiano** de dos conjuntos A y B, que notamos $A \times B$, es el conjunto de todos los pares ordenados (a, b) donde $a \in A$ y $b \in B$.

Ejemplo 0.3 *Dados los conjuntos* $\{1, 3, 9\}$ *y* $\{b, c, d\}$, *el producto cartesiano es,*

$$\{1, 3, 9\} \times \{b, c, d\} = \{(1, b), (1, c), (1, d), (3, b), (3, c), (3, d), (9, b), (9, c), (9, d)\}$$

Una **relación binaria** entre dos conjuntos A y B es un subconjunto de $A \times B$.

Ejemplo 0.4 $\{(9,b),(1,c),(3,d)\}$ *es una relación binaria entre los conjuntos* $\{1,3,9\}$ *y* $\{b,c,d\}$. *La relación "menor que" entre los números naturales es una relación binaria,*

$$<= \{(i,j) \mid (i,j \in \mathbb{N}\ ) \wedge (i<j)\}$$

Sea n un número natural, entonces $(a_1,a_2,...,a_n)$ es una *n-tupla ordenada*. Para cada $i \in \{1,...,n\}$, a_i es la i-ésima componente de la n-tupla. Dos n-tuplas $(b_1,b_2,...,b_n)$ y $(a_1,a_2,...,a_m)$ son iguales si y sólo si $m=n$ y $a_i=b_i$ para cada $i \in \{1,...,n\}$. Si $A_1,...,A_n$ son conjuntos cualesquiera, el producto cartesiano de todos ellos, $A_1 \overbrace{\times \ldots \times} A_n$, es el conjunto de todas las n-tuplas $(a_1,...,a_n)$ con $a_i \in A_i$ para cada $i \in \{1,...,n\}$. En el caso de que todos los A_i sean iguales el producto cartesiano $A \overbrace{\times \ldots \times} A$ se puede escribir como A^n . Una **relación n-aria** entre los conjuntos $A_1,...,A_n$ es un subconjunto del producto cartesiano $A_1 \overbrace{\times \ldots \times} A_n$.
Vamos a tratar ahora con relaciones binarias entre un conjunto y el mismo, es decir, con $R \subseteq A \times A$. Si $(a,b) \in R$ podemos escribirlo con una notación infija como $a\,R\,b$. Por ejemplo, en la relación de igualdad se suele decir que $a=b$, en lugar de $(a,b) \in =$.

Sea R una relación binaria sobre un conjunto A. Decimos que:

- R es **reflexiva** sii $\forall\, a \in A : aRa$
- R es **irreflexiva** sii $\forall\, a \in A : \neg(aRa)$
- R es **transitiva** sii $\forall\, a,b,c \in A : (aRb) \wedge (bRc) \Rightarrow (aRc)$
- R es **simétrica** sii $\forall\, a,b \in A : aRb \Rightarrow bRa$
- R es **antisimétrica** sii $\forall\, a,b \in A : aRb \Rightarrow \neg(bRa)$

Una relación $R \subseteq A \times A$ que cumpla las propiedades reflexiva, simétrica y transitiva se dice que es una **relación de equivalencia**. Usaremos la notación $[a]_R$ para indicar la *clase de equivalencia* de la relación R representada por el elemento $a \in A$ y se define:

$$[a]_R = \{b \in A \mid (a,b) \in R\}$$

Al conjunto formado por todas las clases de equivalencia de una relación de equivalencia $R \subseteq A \times A$ se le denomina *conjunto cociente de A modulo R* y se nota como A/R:

$$A/R = \{[a] \mid a \in A\}$$

Una relación $R \subseteq A \times A$ que cumpla las propiedades reflexiva, antisimétrica y transitiva se dice que es una **relación de orden**. Al par (A,R) lo llamaremos *conjunto ordenado*. Si además la relación de orden R verifica que todo par de elementos de A son comparables, entonces se dice que R es una *relación de orden total* o lineal en A y el par (A,R) es un *conjunto totalmente ordenado*. Un orden no total se llama *parcial*.

Supongamos que P es un conjunto de propiedades sobre relaciones. La **P-clausura** de $R \subseteq A \times A$ es la menor relación R' que incluye todos los pares ordenados de R y cumple las propiedades de P. Por ejemplo, la **clausura transitiva** de R, que notaremos R^+, se define de la siguiente manera:

1. Si $(a, b) \in R$ entonces $(a, b) \in R^+$.
2. Si $(a, b) \in R^+$ y $(b, c) \in R$, entonces $(a, c) \in R^+$.
3. Sólo están en R^+ los pares introducidos por 1 y 2.

La **clausura reflexiva y transitiva** de R, que notamos R^*, se define:

$$R^* = R^+ \cup \{(a, a) \mid a \in A\}$$

Ejemplo 0.5 *Sea* $R = \{(1, 2), (2, 2), (2, 3)\}$ *una relación sobre el conjunto* $\{1, 2, 3\}$. *Entonces tenemos que,*

$$R^+ = \{(1, 2), (2, 2), (2, 3), (1, 3)\}$$
$$R^* = \{(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)\}$$

Una **función** de un conjunto A en un conjunto B, que notamos como $f : A \longrightarrow B$, es una relación binaria $f \subseteq A \times B$ con la siguiente *propiedad:* para cada elemento $a \in A$ hay exactamente un par ordenado en f cuya primera componente sea a. Llamamos a A el *dominio* de la función f y a B el *codominio* de f. Si a es un elemento cualquiera de A, $f(a)$ será un elemento $b \in B$ tal que $(a, b) \in f$ y además por ser f una función este b será único. Al elemento $f(a)$ lo llamaremos *imagen de a bajo f.* Si tenemos la función f anterior y A' es un subconjunto de A, definimos:

$$f\left[A'\right] = \left\{f(a) \mid a \in A'\right\}$$

que es la *imagen de A' bajo f.* El *rango* de una función f es la imagen de su dominio. Por convenio, si el dominio de una función es un producto cartesiano, no hace falta que especifiquemos las parejas de paréntesis de los elementos.

Ejemplo 0.6 *Si* $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ *está definida de forma que la imagen de un par ordenado* (m, n) *es la suma de* m *y* n, *podemos escribir* $f(m, n) = m + n$, *en lugar de* $f((m, n)) = m + n$ *y además podemos decir que* m, n *son los argumentos de* f *y* $m + n$ *el correspondiente valor o resultado de* f.

Monoides

El par $(M, \circ)$ es un *semigrupo* si M es un conjunto y $\circ$ es una operación interna binaria asociativa. Es decir, $\circ$ es una función de $M \times M$ en M que verifica lo siguiente:

$$\forall\, x, y, z \in M : x \circ (y \circ z) = (x \circ y) \circ z$$

Un elemento $e \in M$ es la *identidad* de un semigrupo $(M, \circ)$ si se verifica:

$$\forall\, x \in M : e \circ x = x \circ e = x$$

Un **monoide** es un semigrupo con identidad. Sea el monoide $(M, \circ, e)$, $x \in M$ y un número natural n. La **n-ésima potencia** de x, representada por x^n, se define inductivamente de la siguiente manera:

1. $x^0 = e$
2. $x^n = x \circ x^{n-1}$, para $n > 0$

Sean A y B subconjuntos del monoide $(M, \circ, e)$. La operación $\circ$ induce de forma natural una operación binaria $\circ$ sobre 2^M, el conjunto de todos los subconjuntos de M. Esta operación se define por:

$$\forall\, A, B \in 2^M : A \circ B = \{x \circ y \mid (x \in A) \wedge (y \in B)\}$$

Definición 0.1 *Sea* $(M, \circ, e)$ *un monoide. Entonces* $\left(2^M, \circ, \{e\}\right)$*, donde* $\circ$ *es la operación inducida, es también un monoide que llamaremos* ***monoide inducido*** *por* $(M, \circ, e)$ *sobre* 2^M*.*

Definición 0.2 *Si* A *es un subconjunto del monoide* $(M, \circ, e)$*. Entonces:*

- A *es* ***cerrado positivo*** *sii* $\forall\, x, y \in A : x \circ y \in A$*.*
- A *es* ***cerrado*** *sii es cerrado positivo y además contiene a la identidad* e*.*

Definición 0.3 *Sea* A *un subconjunto cerrado de un monoide. Entonces* $(A, \circ, e)$ *donde* $\circ$ *es la restricción de la operación de* M *para los elementos de* A *es también un monoide. A tal monoide se le denomina* ***submonoide*** *de* $(M, \circ, e)$*.*

Definición 0.4 *Sea* A *cualquier subconjunto de un monoide* $(M, \circ, e)$*. El* ***cierre positivo*** *de* A*, representado por* A^+ *, se define por:*

$$A^+ = \bigcup_{n=1}^{\infty} A^n$$

El ***cierre*** *de* A*, que notaremos como* A^**, se define como:*

$$A^* = \bigcup_{n=0}^{\infty} A^n$$

donde A^n *representa la n-ésima potencia de A en el monoide inducido* $\left(2^M, \circ, \{e\}\right)$*.*

Un subconjunto B de un monoide M se dice que **genera** M sii $B^* = M$. Al conjunto B se le llama *base* (o generador) de M. Si B genera M entonces, por definición, cualquier $x \in M$ (distinto de e) se puede representar como $x = x_1 \circ \ldots \circ x_n$, donde $x_1, \ldots, x_n \in B$ y $n > 0$. Se dice que B **genera libremente** a M si la representación anterior es única (salvo el orden, si la operación $\circ$ es conmutativa). M se dice que es un **monoide libre** si contiene un subconjunto B que lo genera libremente.

Conjuntos finitos e infinitos

Una propiedad básica de los conjuntos finitos es su tamaño o cardinalidad. Algunos aspectos sobre el tamaño de los conjuntos finitos son obvios, como por ejemplo, si $A \subseteq B$ entonces el cardinal de A es menor o igual que el cardinal de B; si A es subconjunto propio de B será de menor tamaño que B. Sin embargo esto no es tan simple cuando tenemos conjuntos infinitos. Por ejemplo, ¿hay más números naturales que números pares? Aunque la intuición nos dice que sí, formalmente no podemos afirmarlo.
Se dice que dos conjuntos A y B son **equinumerables** o *equipotentes* si podemos encontrar una función $f : A \longrightarrow B$ donde f es biyectiva. Decimos que un conjunto es **finito** si es equinumerable con $\{1, 2, \ldots, n\}$, para algún $n \in \mathbb{N}$, y diremos que el cardinal de A es n, esto es, $|A| = n$.
Un conjunto A es **infinito** si puede establecerse una aplicación biyectiva entre A y un subconjunto propio de A. No todos los conjuntos infinitos son equinumerables, por ejemplo $\mathbb{N}$ y $\mathbb{R}$ no tienen la misma cardinalidad. Un conjunto se dice que es **infinito numerable** si es equinumerable con $\mathbb{N}$ y se dice que es **numerable** si es finito o infinito numerable. En caso contrario se dice que es **no numerable**, como por ejemplo el conjunto de los numeros reales $\mathbb{R}$.

Teorema **0.1** *Si A es un conjunto cualquiera (incluso infinito) entonces* $|A| < |\mathcal{P}(A)|$*. Además si* $\mathcal{A}$ *es infinito numerable entonces* $\mathcal{P}$(A) *es no numerable.*

Teorema **0.2** *La cardinalidad del conjunto de los números naturales es menor o igual que la cardinalidad de cualquier conjunto infinito.*

Principio de inducción

El *principio de inducción matemática* afirma lo siguiente,

> Si A es un subconjunto de numeros naturales, $A \subseteq \mathbb{N}$, y satisface las condiciones:

1. $0 \in A$
2. si $k \in A$ entonces $k+1 \in A$

> entonces debe ser $A = \mathbb{N}$.

En la práctica, el principio de inducción es usado para probar afirmaciones del tipo "para todo número natural k la propiedad P se cumple". Esto es lo mismo que probar que el conjunto

$$A = \{k \in \mathbb{N} \mid P(k) \text{ se cumple}\}$$

coincide con el conjunto de números naturales, esto es, debemos probar que $A = \mathbb{N}$. Esto es lo que se llama *demostración por inducción* y el procedimiento a seguir es el siguiente:

> ETAPA BASE Probar que la propiedad P se cumple para 0.
>
> ETAPA DE INDUCCIÓN Suponer que la propiedad se sumple para k (hipótesis de inducción) y probar que esto implica que se cumple para $k+1$.
>
> CONCLUSIÓN Puesto que hemos probado en la etapa base que $0 \in A$ y en la etapa de inducción que si $k \in A$ entonces también $k+1 \in A$, resulta que, por el principio de inducción, podemos deducir que $A = \mathbb{N}$, como queríamos demostrar.

A veces, interesa demostrar que cierta propiedad se cumple para todo $k \geq m$. En este caso debemos demostrar que el conjunto,

$$A = \{n \in \mathbb{N} \mid P(n+m) \text{ se cumple}\}$$

coincide con el conjunto N. Para ello seguimos el siguiente razonamiento:

> ETAPA BASE ($n = 0$) Probar que $P(m)$ se cumple.
>
> ETAPA DE INDUCCIÓN ($n > 0$) Suponer que $P(k)$ se cumple, siendo $k \geq m$ y probar que $P(k+1)$ se cumple.
>
> CONCLUSIÓN Por las etapas anteriores y el principio de inducción tenemos que $A = \mathbb{N}$ y por tanto P se cumple para todo $k \geq m$.

El principio de inducción también se usa para definir conjuntos de objetos donde definimos el primer objeto y el objeto k se define en términos del $(k-1)$-ésimo objeto. Esto es lo que se llama *definición inductiva*.

Ejemplo 0.7 *El factorial de un número natural puede ser definido inductivamente como,*

1. $0! = 1$
2. $k! = k \cdot (k-1)!$ *para* $k > 0$.

CAPÍTULO 1:
LENGUAJES Y GRAMÁTICAS FORMALES

Contenidos Teóricos

1. Alfabetos y palabras

Un *alfabeto* es un conjunto finito y no vacío de elementos llamados *símbolos* o letras. Una *palabra* o cadena sobre un alfabeto V es una cadena finita de símbolos del alfabeto. La *longitud* de una cadena w, que notaremos como $|w|$, es el número de letras que aparecen en w. A la cadena que no tiene símbolos, o lo que es lo mismo, que tiene longitud 0, la llamaremos *palabra vacía* y se nota por λ (o también ϵ, según los autores).
Si V es un alfabeto, llamaremos V^n al conjunto de todas las palabras de longitud n sobre V. Un elemento de V^n será una cadena del tipo $a_1 a_2 \dots a_n$ donde cada $a_i \in V$. Llamaremos V^0 al conjunto cuyo único elemento es la palabra vacía, es decir, $V^0 = \{\lambda\}$. El conjunto de todas las cadenas de cualquier longitud sobre V es:

$$V^* = \bigcup_{n=0}^{\infty} V^n$$

Llamamos V^+ al conjunto de todas las cadenas sobre el alfabeto V excepto la vacía. Por tanto, $V^+ = V^* - \{\lambda\}$.

1.1. Concatenación de Palabras

La operación de concatenación, que notaremos '$\cdot$', es una operación binaria entre palabras sobre un alfabeto V, esto es:

$$\cdot : V^* \times V^* \longrightarrow V^*$$

de forma que si tenemos dos palabras $x, y \in V^*$ donde $x = a_1 a_2 \dots a_n,\ y = b_1 b_2 \dots b_m$ entonces, x *concatenado* con y será una palabra $w \in V^*$ con $|w| = |x| + |y|$, de forma que:

$$w = x \cdot y = a_1 a_2 \dots a_n b_1 b_2 \dots b_m$$

Nota A veces se suele suprimir el '$\cdot$' y se puede escribir directamente $w = xy$

Algunas PROPIEDADES de la concatenación son:

- operación **cerrada** $\dashrightarrow \quad \forall\, x, y \in V^* :\ x \cdot y \in V^*$
- propiedad **asociativa** $\dashrightarrow \quad \forall\, x, y, z \in V^* :\ x \cdot (y \cdot z) = (x \cdot y) \cdot z$
- **elemento neutro** λ $\dashrightarrow \quad \forall\, x \in V^* :\ \lambda \cdot x = x \cdot \lambda = x$

Por tener estas propiedades $(V^*, \cdot, \lambda)$ es un *monoide*. Además cada palabra de V^* se representa de *forma única* como concatenación de símbolos de V, por eso es además un *monoide libre*.
Todo monoide libre cumple la ley de **cancelación izquierda y derecha**, en este caso, $\forall\, x, y, z \in V$ se cumple que:

$$(x \cdot y = x \cdot z) \Rightarrow (y = z) \qquad \| \qquad (y \cdot x = z \cdot x) \Rightarrow (y = z)$$

Decimos que una cadena z es *subcadena* de otra cadena w si existen cadenas $x, y \in V^*$ tal que $w = x \cdot z \cdot y$. Vamos a ver dos conjuntos especiales de subcadenas:

$$\text{Prefijo}(w) = \{x \in V^* \mid \exists\ z \in V^* :\ w = x \cdot z\} \ \|\ \text{Sufijo}(w) = \{x \in V^* \mid \exists\ z \in V^* :\ w = z \cdot x\}$$

Diremos que x es un *prefijo* de w si $x \in \text{Prefijo}(w)$ y será un *prefijo propio* si $x \neq w$. Por otra parte, diremos que x es un *sufijo* de w si $x \in \text{Sufijo}(w)$ y será un *sufijo propio* si $x \neq w$.

Ejemplo 1.1 *Si $w = abab$ es una palabra sobre el alfabeto $\{a, b\}$, o lo que es lo mismo, $w \in \{a, b\}^*$, tenemos que:*

ab es un prefijo propio de w
$abab$ es un prefijo de w, pero no es propio
b es un sufijo de w

1.2. Potencia de una palabra

Llamamos *potencia n-ésima* de una palabra, a la operación que consiste en concatenar la palabra consigo misma n veces. Dada una palabra $w \in V^*$, se define inductivamente la potencia n-ésima de w, que notaremos w^n, como:

1. $w^0 = \lambda$
2. $w^n = w \cdot w^{n-1}$ para $n > 0$

Ejemplo 1.2 *Si $w = aba$ es una palabra sobre el alfabeto $\{a, b\}$ entonces:*

$$w^0 = \lambda$$
$$w^1 = aba$$
$$w^2 = abaaba$$

1.3. Inversión de palabras

Si $w = a_1 a_2 \dots a_n$ es una palabra sobre un alfabeto V entonces la *palabra inversa* o *refleja* de w se define como: $w^R = a_n a_{n-1} \dots a_1$

Ejemplo 1.3 *Si $w = aaba$ es una palabra sobre el alfabeto $\{a, b\}$, entonces $w^R = abaa$.*

2. Lenguajes formales

Llamamos LENGUAJE SOBRE EL ALFABETO V a cualquier subconjunto de V^*. Así tenemos que, $V^*, \varnothing$, y V pueden considerarse como lenguajes. Puesto que un lenguaje es tan sólo una clase especial de conjunto, podemos especificar un *lenguaje finito* por extensión enumerando sus elementos entre llaves. Por ejemplo, $\{aba, czr, d, f\}$ es un lenguaje sobre el alfabeto $\{a, b, c, ..., z\}$. Sin embargo, la mayoría de los lenguajes de interés son *infinitos*. En este caso podemos especificar un lenguaje por comprensión de la siguiente forma:

$$\boxed{L = \{w \in V^* \mid w \text{ cumple la propiedad } P\}}$$

En la definición anterior vemos que V^* es el conjunto *referencial*, que podemos llamar también *lenguaje universal* sobre V.

Ejemplo 1.4 $L = \{w \in \{0,1\}^* \mid ceros(w) = unos(w)\}$*, palabras que tienen el mismo número de ceros que de unos.*

2.1. Operaciones del algebra de conjuntos

Sean L_1 y L_2 dos lenguajes definidos sobre el alfabeto V. Se define la *unión* de estos dos lenguajes como el lenguaje L sobre V que se especifica como:

$$L = L_1 \cup L_2 = \{w \in V^* \mid (w \in L_1) \vee (w \in L_2)\}$$

La unión de lenguajes sobre el mismo alfabeto es un operación *cerrada* y además cumple las propiedades *asociativa*, *conmutativa*, y existe un *elemento neutro* que es el lenguaje vacío $\varnothing$ (no es lo mismo $\varnothing$ que el lenguaje que contiene la palabra vacía $\{\lambda\}$). El conjunto $\mathcal{P}(V^*)$ (esto es, el conjunto de las partes de V^*, también llamado 2^{V^*}), está formado por todos los lenguajes posibles que se pueden definir sobre el alfabeto V. Entonces, por cumplir la unión las propiedades anteriores tenemos que $(\mathcal{P}(V^*), \cup, \varnothing)$ es un *monoide abeliano*.
De forma análoga a la unión se pueden definir otras operaciones del álgebra de conjuntos como la *intersección, diferencia, y complementación* de lenguajes. Por ejemplo, el complementario del lenguaje L sobre el alfabeto V será: $\overline{L} = V^* - L$.

2.2. Concatenación, potencia e inversión de lenguajes

Sean L_1 y L_2 dos lenguajes definidos sobre el alfabeto V, la *concatenación* de estos dos lenguajes es otro lenguaje L definido como:

$$\boxed{L_1 \cdot L_2 = \{x \cdot y \in V^* \mid (x \in L_1) \wedge (y \in L_2)\}}$$

La definición anterior sólo es válida si L_1 y L_2 contienen al menos un elemento. Podemos extender la operación de concatenación al lenguaje vacío de la siguiente manera:

$$\varnothing \cdot L = L \cdot \varnothing = \varnothing$$

La concatenación de lenguajes sobre un alfabeto es una operación *cerrada*, y además cumple la propiedad *asociativa* y tiene un *elemento neutro* que es el lenguaje $\{\lambda\}$. Con lo cual, tenemos que $(\mathcal{P}(V^*), \cdot, \{\lambda\})$ es el *monoide inducido* por el monoide $(V^*, \cdot, \lambda)$ sobre $\mathcal{P}(V^*)$. Esto es, la operación de concatenación de palabras *induce* la operación de concatenación de lenguajes y ésta conserva las propiedades de la primera.

Teorema **1.1** *Dados los lenguajes A, B, C sobre un alfabeto V, la concatenación de lenguajes es distributiva con respecto a la unión, esto es, se cumple que:*

1. $A \cdot (B \cup C) = (A \cdot B) \cup (A \cdot C)$
2. $(B \cup C) \cdot A = (B \cdot A) \cup (C \cdot A)$

Dem.- La demostración se deja como ejercicio. En el primer caso se debe probar que:

$$A \cdot (B \cup C) \subseteq (A \cdot B) \cup (A \cdot C) \quad \text{y} \quad (A \cdot B) \cup (A \cdot C) \subseteq A \cdot (B \cup C)$$

para demostrar la igualdad y el segundo caso se demuestra de forma análoga. ■

Una vez definida la concatenación de lenguajes, podemos definir la *potencia n-ésima* de un lenguaje como la operación que consiste en concatenar el lenguaje consigo mismo n veces. La definición inductiva es:

1. $L^0 = \{\lambda\}$
2. $L^n = L \cdot L^{n-1}$, $\forall\, n > 0$

Ejemplo 1.5 *Si $L = \{ab, c\}$ es un lenguaje sobre el alfabeto $\{a, b, c\}$ entonces,*

$$\begin{aligned} L^0 &= \{\lambda\} \\ L^1 &= L = \{ab, c\} \\ L^2 &= L \cdot L^1 = \{abab, abc, cab, cc\} \\ L^3 &= L \cdot L^2 = \{ababab, ababc, abcab, abcc, cabab, cabc, ccab, ccc\} \end{aligned}$$

Las definiciones de prefijo y sufijo de una palabra podemos extenderlas a lenguajes de la siguiente forma:

$$\text{Prefijo}(L) = \bigcup_{w \in L} \text{Prefijo}(w) \quad \| \quad \text{Sufijo}(L) = \bigcup_{w \in L} \text{Sufijo}(w)$$

También podemos definir el *lenguaje inverso* o reflejo de L como:

$$L^R = \left\{ w^R \mid w \in L \right\}$$

2.3. Clausura de un lenguaje

Dado un lenguaje L sobre un alfabeto V se define la *clausura positiva* (o cierre positivo) de L, denotado L^+, como:

$$\boxed{L^+ = \bigcup_{n=1}^{\infty} L^n}$$

Definimos L^* como la *clausura* (o cierre) de L, como:

$$\boxed{L^* = \bigcup_{n=0}^{\infty} L^n}$$

En ambos casos, L^n se refiere a la potencia n-ésima del lenguaje L en el monoide inducido $(\mathcal{P}(V^*), \cdot, \{\lambda\})$. El cierre o clausura de un lenguaje, por definición, contiene cualquier palabra que se obtenga por concatenación de palabras de L y además la palabra vacía.

3. Gramáticas formales

Hasta ahora hemos descrito los lenguajes formales como se describen los conjuntos: por extensión (si son finitos) o por comprensión. Aquí vamos a introducir otra forma general y rigurosa de describir un lenguaje formal: mediante el uso de gramáticas. Las gramáticas son mecanismos *generadores* de lenguajes, es decir, nos dicen cómo podemos obtener o construir palabras de un determinado lenguaje.

3.1. Definiciones básicas

Definición 1.1 *Una* GRAMÁTICA *es una cuadrupla* $G = (V_N, V_T, S, P)$ *donde:*

V_T *es el alfabeto de símbolos terminales*

V_N *es el alfabeto de símbolos no terminales o variables, de forma que debe ser* $V_N \cap V_T = \varnothing$ *y denotamos con* V *al alfabeto total de la gramática, esto es,* $V = V_N \cup V_T$.

S *es el símbolo inicial y se cumple que* $S \in V_N$

P *es un conjunto finito de reglas de producción*

Definición 1.2 *Una* REGLA DE PRODUCCIÓN *es un par ordenado* (α, β) *de forma que:*

$$(\alpha, \beta) \in (V^* \cdot V_N \cdot V^*) \times V^*$$

Es decir, $\alpha = \gamma_1 A \gamma_2$ *donde* $\gamma_1, \gamma_2 \in (V_N \cup V_T)^*$, $A \in V_N$ *y* $\beta \in (V_N \cup V_T)^*$. *Una producción* $(\alpha, \beta) \in P$ *se suele escribir de forma infija como* $\alpha \to \beta$.

Por **convenio** usaremos letras mayúsculas para los símbolos no terminales; dígitos y las primeras letras minúsculas del alfabeto para los símbolos terminales; las últimas letras minúsculas del alfabeto para palabras que pertenezcan a V_T^* y letras griegas para cualquier palabra que pertenezca a V^*. Usando este convenio, a veces se suele describir una gramática enumerando únicamente sus reglas de producción y cuando varias reglas tienen la misma parte izquierda, se suelen agrupar separándolas con |.

Ejemplo 1.6 *Sea la gramática* G *cuyas producciones son:*

$$S \to aSa \mid bSb \mid a \mid b \mid \lambda$$

Esta gramática tiene una sola variable S *que además es el símbolo inicial.* $V_T = \{a, b\}$ *y* P *contiene 5 reglas de producción.*

Definición 1.3 *Sea* G *una gramática y sean las cadenas* $\alpha, \beta \in V^*$. *Decimos que* α *deriva directamente en* β, *que notamos como* $\alpha \Rightarrow \beta$ (DERIVACIÓN DIRECTA), *si y sólo si existe una producción* $\delta \to \sigma \in P$ *tal que* $\alpha = \gamma_1 \delta \gamma_2$, $\beta = \gamma_1 \sigma \gamma_2$ *con* $\gamma_1, \gamma_2 \in V^*$.

Esto quiere decir que α deriva directamente en β, si β puede obtenerse a partir de α sustituyendo una ocurrencia de la parte izquierda de una producción que aparezca en α por la parte derecha de la regla de producción.

Nota Si $\alpha \to \beta$ es una regla de producción de G, entonces se cumple siempre que $\alpha \Rightarrow \beta$. Cuando sea necesario distinguir entre varias gramáticas, escribiremos $\alpha \Rightarrow_G \beta$, para referirnos a un derivación directa en G.

Por la definición anterior se deduce que $\Rightarrow$ es una relación binaria en el conjunto de cadenas de la gramática, esto es: $\Rightarrow \subseteq V^* \times V^*$. Aquí usamos una notación infija para indicar que $\alpha \Rightarrow \beta$ en lugar de $(\alpha, \beta) \in \Rightarrow$.

Definición 1.4 *Decimos que α deriva en β, o bien que, β es derivable de α, y lo notamos como $\alpha \Rightarrow^* \beta$ (*DERIVACIÓN*) si y sólo si se verifica una de las dos condiciones siguientes:*

1. *$\alpha = \beta$, (son la misma cadena), o bien,*
2. *$\exists\, \gamma_0, \gamma_1, \ldots, \gamma_n \in V^*$ tal que $\gamma_0 = \alpha,\ \gamma_n = \beta$ y $\forall\, 0 \leq i < n$ se cumple que $\gamma_i \Rightarrow \gamma_{i+1}$*

A la secuencia $\gamma_0 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_n$ la llamaremos secuencia de derivaciones directas de longitud n, o simplemente derivación de longitud n.

Nota Por la definición anterior está claro que $\overset{*}{\Rightarrow}$ es también una relación binaria en V^* y además es la *clausura reflexiva y transitiva* de la relación de derivación directa $\Rightarrow$. Esto quiere decir que $\overset{*}{\Rightarrow}$ es la *menor relación* que cumple lo siguiente:

- Si $\alpha \Rightarrow \beta$ entonces $\alpha \overset{*}{\Rightarrow} \beta$. Esto es, si dos cadenas están relacionadas mediante $\Rightarrow$ entonces también lo están mediante la relación $\overset{*}{\Rightarrow}$
- $\overset{*}{\Rightarrow}$ es reflexiva, ya que $\forall\, \alpha \in V^*$ se cumple que $\alpha \overset{*}{\Rightarrow} \alpha$
- $\overset{*}{\Rightarrow}$ es transitiva. En efecto, si $\alpha \overset{*}{\Rightarrow} \beta$ y $\beta \overset{*}{\Rightarrow} \gamma$, entonces $\alpha \overset{*}{\Rightarrow} \gamma$

Definición 1.5 *Sea una gramática $G = (V_N, V_T, S, P)$. Una palabra $\alpha \in (V_N \cup V_T)^*$ se denomina* FORMA SENTENCIAL *de la gramática, si y sólo si se cumple que: $S \overset{*}{\Rightarrow} \alpha$. Una forma sentencial w tal que $w \in V_T^*$ se dice que es una* SENTENCIA.

Ejemplo 1.7 *Sea la gramática $S \to aSa \mid bSb \mid a \mid b \mid \lambda$, podemos afirmar lo siguiente:*

- *$aaSb \Rightarrow aabSbb$, aunque ni $aaSb$ ni $aabSbb$ son formas sentenciales de G*
- *$aabb \overset{*}{\Rightarrow} aabb$, aunque $aabb$ no es una sentencia de G*
- *S, aSa, $abSba$, λ son formas sentenciales de G y además λ es una sentencia*
- *$aabaa$ es una sentencia de G, ya que existe una derivación de longitud 3 por la que $S \overset{*}{\Rightarrow} aabaa$. En efecto:*

$$S \Rightarrow aSa \Rightarrow aaSaa \Rightarrow aabaa$$

Definición 1.6 *Sea una gramática $G = (V_N, V_T, S, P)$. Se llama* LENGUAJE GENERADO *por la gramática G al lenguaje $L(G)$ formado por todas las cadenas de símbolos terminales que son derivables del símbolo inicial de la gramática (sentencias):*

$$\boxed{L(G) = \left\{ w \in V_T^* \mid S \overset{*}{\Rightarrow} w \right\}}$$

Ejemplo 1.8 *Sea $L = \left\{ w \in \{a,b\}^* \mid w = w^R \right\}$. Este lenguaje está formado por todos los palíndromos sobre el alfabeto $\{a,b\}$. Puede probarse que la gramática $S \to aSa \mid bSb \mid a \mid b \mid \lambda$ genera el lenguaje L. En general no existe un método exacto para probar que una gramática genera un determinado lenguaje. Para este caso tan sencillo podemos probarlo de "manera informal" haciendo una serie de derivaciones hasta darnos cuenta de que $S \overset{*}{\Rightarrow} w$ si y sólo si $w = w^R$. Luego veremos una demostración formal por inducción en la sección de aplicaciones.*

Definición 1.7 *Dos gramáticas G y G' son* EQUIVALENTES *si y sólo si generan el mismo lenguaje, es decir, sii $L(G) = L(G')$.*

3.2. Notación BNF

A veces se utiliza una notación especial para describir gramáticas llamada notación BNF (*Backus-Naus-Form*). En la notación BNF los símbolos no terminales o variables son encerrados entre ángulos y utilizaremos el símbolo ::= para las producciones, en lugar de $\rightarrow$. Por ejemplo, la producción $S \rightarrow aSa$ se representa en BNF como $\langle S\rangle ::= a\,\langle S\rangle\,a$. Tenemos también la notación *BNF-extendida* que incluye además los símbolos [ ] y { } para indicar elementos opcionales y repeticiones, respectivamente.

Ejemplo 1.9 *Supongamos que tenemos un lenguaje de programación cuyas dos primeras reglas de producción para definir su sintaxis son:*

$$\langle programa\rangle ::= [\langle cabecera\rangle]\ \ begin\ \langle sentencias\rangle\ \ end$$
$$\langle sentencias\rangle ::= \langle sentencia\rangle\,\{\langle sentencia\rangle\}$$

Esto viene a decir que un programa se compone de una cabecera opcional, seguido de la palabra clave "begin", a continuación una lista de sentencias (debe haber al menos una sentencia) y finaliza con la palabra clave "end". Podemos transformar las producciones anteriores para especificarlas, según la notación que nosotros hemos introducido (estándar), de la siguiente forma:

$$P \rightarrow C\,\text{begin}\,A\,\text{end} \mid \text{begin}\,A\,\text{end}$$
$$A \rightarrow B\,A \mid B$$

donde P es el símbolo inicial de la gramática y corresponde a la variable $\langle programa\rangle$, C corresponde a $\langle cabecera\rangle$, A se refiere a la variable $\langle sentencias\rangle$ y B a $\langle sentencia\rangle$.

La simbología utilizada para describir las gramáticas en notación estándar y en notación BNF nos proporcionan un herramienta para describir los lenguajes y la estructura de las sentencias del lenguaje. Puede considerarse a esta simbología como un *metalenguaje*, es decir un lenguaje que sirve para describir otros lenguajes.

3.3. Jerarquía de Chomsky

En 1959 *Chomsky* clasificó las gramáticas en cuatro familias, que difieren unas de otras en la forma que pueden tener sus reglas de producción. Si tenemos una gramática $G = (V_N, V_T, S, P)$ clasificaremos las gramáticas y los lenguajes generados por ellas, de la siguiente forma:

- TIPO 3 (*Gramáticas regulares*). Pueden ser, a su vez, de dos tipos:

  - *Lineales por la derecha.* Todas sus producciones son de la forma:

    $$\boxed{\begin{array}{l} A \rightarrow bC \\ A \rightarrow b \\ A \rightarrow \lambda \end{array}}$$

    donde $A, C \in V_N$ y $b \in V_T$.

- *Lineales por la izquierda.* Con producciones del tipo:

$$\boxed{\begin{array}{l} A \to Cb \\ A \to b \\ A \to \lambda \end{array}}$$

  Los lenguajes generados por estas gramáticas se llaman LENGUAJES REGULARES y el conjunto de todos estos lenguajes es la clase $\mathcal{L}_3$.

- TIPO 2 (*Gramáticas libres del contexto*). Las producciones son de la forma:

$$\boxed{A \to \alpha}$$

  donde $A \in V_N$ y $\alpha \in (V_N \cup V_T)^*$. Los lenguajes generados por este tipo de gramáticas se llaman LENGUAJES LIBRES DEL CONTEXTO y la clase es $\mathcal{L}_2$.

- TIPO 1 (*Gramáticas sensibles al contexto*). Las producciones son de la forma:

$$\boxed{\alpha A \beta \to \alpha \gamma \beta}$$

  donde $\alpha, \beta \in V^*$ y $\gamma \in V^+$ Se permite además la producción $S \to \lambda$ siempre y cuando S no aparezca en la parte derecha de ninguna regla de producción.

  El sentido de estas reglas de producción es el de especificar que una variable A puede ser reemplazada por γ en una derivación directa sólo cuando A aparezca en el "contexto" de α y β, de ahí el nombre "sensibles al contexto". Además, las producciones de esa forma cumplen siempre que la parte izquierda tiene longitud menor o igual que la parte derecha, pero nunca mayor (excepto para $S \to \lambda$). Esto quiere decir que la gramática es *no contráctil*.

  Los lenguajes generados por las gramáticas de tipo 1 se llaman LENGUAJES SENSIBLES AL CONTEXTO y su clase es $\mathcal{L}_1$.

- TIPO 0 (*Gramáticas con estructura de frase*) Son las gramáticas más generales, que por ello también se llaman *gramáticas sin restricciones*. Esto quiere decir que las producciones pueden ser de cualquier tipo permitido, es decir, de la forma $\alpha \to \beta$ con $\alpha \in (V^* \cdot V_N \cdot V^*)$ y $\beta \in V^*$.

  Los lenguajes generados por estas gramáticas son los LENGUAJES CON ESTRUCTURA DE FRASE, que se agrupan en la clase $\mathcal{L}_0$. Estos lenguajes también se conocen en el campo de la Teoría de la Computabilidad como *lenguajes recursivamente enumerables*.

Teorema **1.2 (Jerarquía de Chomsky)** *Dado un alfabeto V, el conjunto de los lenguajes regulares sobre V está incluido propiamente en el conjunto de los lenguajes libres de contexto y este a su vez está incluido propiamente en el conjunto de los lenguajes sensibles al contexto, que finalmente está incluido propiamente en el conjunto de lenguajes con estructura de frase. Esto es:*

$$\boxed{\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0}$$

La demostración de este teorema la iremos viendo a lo largo del curso.

Nota En este tema hemos hecho referencia al termino *lenguaje formal* para diferenciarlo de *lenguaje natural*. En general, un lenguaje natural es aquel que ha evolucionado con el paso del tiempo para fines de la comunicación humana, por

ejemplo el español o el inglés. Estos lenguajes evolucionan sin tener en cuenta reglas gramaticales formales. Las reglas surgen después con objeto de explicar, más que determinar la *estructura* de un lenguaje, y la sintaxis es difícil de determinar con precisión. Los lenguajes formales, por el contrario, están definidos por reglas de producción preestablecidas y se ajustan con todo rigor o "formalidad" a ellas. Como ejemplo tenemos los lenguajes de programación y los lenguajes lógicos y matemáticos. No es de extrañar, por tanto, que se puedan construir compiladores eficientes para los lenguajes de programación y que por contra la construcción de traductores para lenguaje natural sea una tarea compleja e ineficiente, en general. Veremos que las gramáticas regulares y libres de contexto, junto con sus máquinas abstractas asociadas tienen especial interés en la construcción de traductores para lenguajes de programación.

4. Nociones básicas sobre traductores

Hace apenas unas cuantas décadas, se utilizaban los llamados *lenguajes de primera generación* para hacer que los computadores resolvieran problemas. Estos lenguajes operan a nivel de código binario de la máquina, que consiste en una secuencia de ceros y unos con los que se instruye al ordenador para que realice acciones. La programación, por tanto, era difícil y problemática, aunque pronto se dio un pequeño paso con el uso de código octal o hexadecimal.

El código de máquina fue reemplazado por los *lenguajes de segunda generación*, o *lenguajes ensambladores*. Estos lenguajes permiten usar abreviaturas nemónicas como nombres simbólicos, y la abstracción cambia del nivel de flip-flop al nivel de registro. Se observan ya los primeros pasos hacia la estructuración de programas, aunque no puede utilizarse el término de *programación estructurada* al hablar de programas en ensamblador. Las desventajas principales del uso de los lenguajes ensambladores son, por un lado, la dependencia de la máquina y, por otro, que son poco legibles.

Para sustituir los lenguajes ensambladores, se crearon los *lenguajes de tercera generación* o *lenguajes de alto nivel*. Con ellos se pueden usar estructuras de control basadas en objetos de datos lógicos: variables de un tipo específico. Ofrecen un nivel de abstracción que permite la especificación de los datos, funciones o procesos y su control en forma independiente de la máquina. El diseño de programas para resolver problemas complejos es mucho más sencillo utilizando este tipo de lenguajes, ya que se requieren menos conocimientos sobre la estructura interna del computador, aunque es obvio que el ordenador únicamente entiende código máquina.

Por lo tanto, para que un computador pueda ejecutar programas en lenguajes de alto nivel, estos deben ser traducidos a código máquina. A este proceso se le denomina *compilación*, y la herramienta correspondiente se llama *compilador*. Nosotros vamos a entender el término *compilador* como un programa que lee otro, escrito en *lenguaje fuente*, y lo traduce a *lenguaje objeto*, informando, durante el proceso de traducción, de la presencia de errores en el programa fuente. Esto se refleja en la figura 1.1.

En la década de 1950, se consideró a los compiladores como programas notablemente difíciles de escribir. El primer compilador de FORTRAN, por ejemplo, necesitó para su implementación, 18 años de trabajo en grupo. Desde entonces, se han descubierto técnicas sistemáticas para manejar muchas de las importantes tareas que surgen en la compilación. También se han desarrollado buenos lenguajes de implementación, entornos de programación y herramientas de software. Con estos avances, puede construirse un compilador real incluso como proyecto de estudio en una asignatura sobre diseño de compiladores.

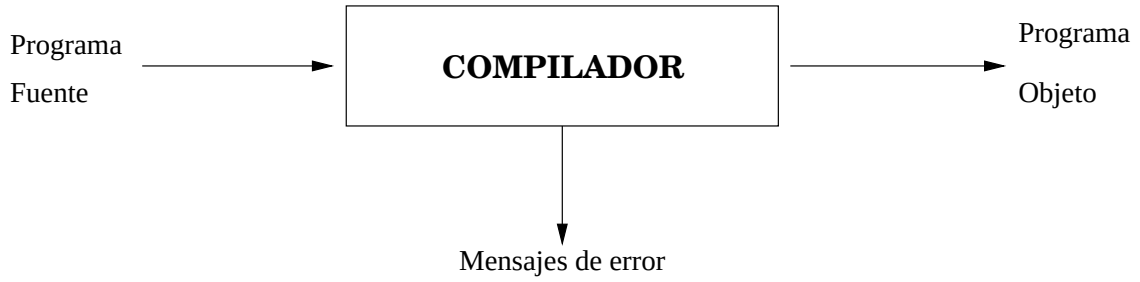


Figura 1.1: Definición de un compilador

4.1. Traductores y compiladores

Un *traductor* es un programa que acepta cualquier texto expresado en un lenguaje (el **lenguaje fuente** del traductor) y genera un texto semánticamente equivalente expresado en otro lenguaje (su **lenguaje destino**).
Un *ensamblador* traduce un lenguaje ensamblador en su correspondiente código máquina. Generalmente, un ensamblador genera una instrucción de la máquina por cada instrucción fuente.
Un *compilador* traduce desde un lenguaje de alto nivel a otro lenguaje de bajo nivel. Generalmente, un compilador genera varias instrucciones de la máquina por cada comando fuente.
Los ensambladores y compiladores son las clases más importantes de traductores de lenguajes de programación, pero no son las únicas clases. A veces se utilizan los *traductores de alto nivel* cuya fuente y destino son lenguajes de alto nivel. Un *desensamblador* traduce un código máquina en su correspondiente lenguaje ensamblador. Un *descompilador* traduce un lenguaje de bajo nivel en un lenguaje de alto nivel.
Nosotros estamos interesados en la traducción de textos que son programas. Antes de realizar cualquier traducción, un compilador comprueba que el texto fuente sea un programa correcto del lenguaje fuente. (En caso contrario genera un informe con los errores). Estas comprobaciones tienen en cuenta la *sintaxis* y las *restricciones contextuales* del lenguaje fuente. Suponiendo que el programa fuente es correcto, el compilador genera un programa objeto que es semánticamente equivalente al programa fuente, es decir, que tiene los efectos deseados cuando se ejecuta. La generación del programa objeto tiene en cuenta tanto la *semántica* del lenguaje fuente como la *semántica* del lenguaje destino.
Los traductores, y otros procesadores de lenguajes, son programas que manipulan programas. Varios lenguajes se ven implicados: no sólo el lenguaje fuente y el lenguaje destino, sino también el lenguaje en el cual el traductor se ha escrito. Este último es el llamado *lenguaje de implementación*.

4.2. Intérpretes

Un compilador nos permite preparar un programa para que sea ejecutado en una máquina, traduciendo el programa a código máquina. El programa entonces se ejecuta a la velocidad de la máquina. Este método de trabajo no está libre de inconvenientes: todo el programa debe ser traducido antes que pueda ejecutarse y producir resultados. En un entorno interactivo, la *interpretación* es un método de trabajo más atractivo.
Un *intérprete* es un programa que acepta otro programa (el *programa fuente*) escrito en un determinado lenguaje (el *lenguaje fuente*), y ejecuta el programa inmediatamente. Un intérprete trabaja cargando, analizando y ejecutando una a una las instrucciones del programa fuente. El programa fuente comienza a ejecutarse y produce resultados desde el momento en que la primera instrucción ha sido analizada. El intérprete no traduce el programa fuente en un código objeto.

La interpretación es un buen método cuando se dan las siguientes circunstancias:

- El programador está trabajando en forma interactiva, y quiere ver el resultado de cada instrucción antes de entrar la siguiente instrucción.
- El programa se va a utilizar sólo una vez, y por tanto la velocidad de ejecución no es importante.
- Se espera que cada instrucción se ejecute una sola vez.
- Las instrucciones tiene un formato simple, y por tanto pueden ser analizadas de forma fácil y eficiente.

La interpretación es muy lenta. La interpretación de un programa fuente, escrito en un lenguaje de alto nivel, puede ser 100 veces más lenta que la ejecución del programa equivalente escrito en código máquina. Por tanto la interpretación no es interesante cuando:

- El programa se va a ejecutar en modo de producción, y por tanto la velocidad es importante.
- Se espera que las instrucciones se ejecuten frecuentemente.
- Las instrucciones tienen formatos complicados, y por tanto su análisis es costoso en tiempo.

Algunos intérpretes más o menos conocidos son:

(a) Un intérprete Caml: Caml es un lenguaje funcional. El intérprete lee cada cada línea hasta el símbolo ";;" y la ejecuta produciendo una salida, por lo que el usuario ve el resultado de la misma antes de entrar la siguiente. Existen versiones tanto para Windows como para distintas versiones de Linux. Existen también varios compiladores para distintos sistemas operativos.

(b) Un intérprete Lisp: Lisp es un lenguaje en el que existe una estructura de datos (árbol) tanto para el código como para los datos.

(c) El intérprete de comandos de Unix (*shell*): Una instrucción para el sistema operativo del usuario de Unix se introduce dando el comando de forma textual. El programa *shell* lee cada comando, lo analiza y extrae un nombre de comando junto con algunos argumentos y ejecuta el comando por medio de un sistema de llamadas. El usuario puede ver el resultado de un comando antes de entrar el siguiente. Los comandos constituyen un lenguaje de comandos, y el *shell* es un intérprete para tal lenguaje.

(d) Un intérprete SQL: SQL es un lenguaje de preguntas (*query language*) a una base de datos. El usuario extrae información de la base de datos introduciendo una pregunta SQL, que es analizada y ejecutada inmediatamente. Esto es realizado por el intérprete SQL que se encuentra dentro del sistema de administración de la base de datos.

4.3. Compiladores interpretados

Un compilador puede tardar mucho en traducir un programa fuente a código máquina, pero una vez hecho esto, el programa puede correr a la velocidad de la máquina. Un intérprete permite que el programa comience a ejecutarse inmediatamente, pero corre muy lento (unas 100 veces más lento que el programa en código máquina).
Un *compilador interpretado* es una combinación de compilador e intérprete, reuniendo algunas de las ventajas de cada uno de ellos. La idea principal es traducir el programa fuente en un *lenguaje intermedio*, diseñado para cumplir los siguiente requisitos:

- tiene un nivel intermedio entre el lenguaje fuente y el código máquina
- sus instrucciones tienen formato simple, y por tanto pueden ser analizadas fácil y rápidamente.
- la traducción desde el lenguaje fuente al lenguaje intermedio es fácil y rápida.

Por tanto un compilador interpretado combina la rapidez de la compilación con una velocidad tolerable en la ejecución.
El código de la Máquina Virtual de Java (el JVM-code) es un lenguaje intermedio orientado a Java. Nos provee de potentes instrucciones que corresponden directamente a las operaciones de Java tales como la creación de objetos, llamadas de métodos e indexación de matrices. Por ello la traducción desde Java a JVM-code es fácil y rápida. Además de ser potente, las instrucciones del JVM-code tienen un formato tan sencillo como las instrucciones del código máquina con campos de operación y campos de operandos, y por tanto son fáciles de analizar. Por ello la interpretación del JVM-code es relativamente rápida: alrededor de 'sólo' diez veces más lenta que el código máquina. JDK consiste en un traductor de Java a JVM-code y un intérprete de JVM-code, los cuales se ejecutan sobre alguna máquina M.

4.4. Contexto de un compilador

En el proceso de construcción de un programa escrito en código máquina a partir del programa fuente, suelen intervenir, aparte del compilador, otros programas:

- *Preprocesador*: Es un traductor cuyo lenguaje fuente es una forma extendida de algún lenguaje de alto nivel, y cuyo lenguaje objeto es la forma estándar del mismo lenguaje. Realiza la tarea de reunir el programa fuente, que a menudo se divide en módulos almacenados en archivos diferentes. También puede expandir abreviaturas, llamadas macros, a proposiciones del lenguaje fuente. El programa objeto producido por un preprocesador puede , entonces, ser traducido y ejecutado por el procesador usual del lenguaje estándar.
- *Ensamblador*: Traduce el programa en lenguaje ensamblador, creado por el compilador, a código máquina.
- *Cargador* y *linkador*: Un *cargador* es un traductor cuyo lenguaje objeto es el código de la máquina real y cuyo lenguaje fuente es casi idéntico. Este consiste usualmente en programas de lenguaje máquina en forma reubicable, junto con tablas de datos que especifican los puntos en dónde el código reubicable debe modificarse para convertirse en verdaderamente ejecutable. Por otro lado, un *linkador* es un traductor con los mismos lenguajes fuente y objeto que el cargador. Toma como entrada programas en forma reubicable que se han compilado separadamente, incluyendo subprogramas almacenados en librerías. Los une en una sola unidad de código máquina lista para ejecutarse. En general, un editor de carga y enlace une el código máquina a rutinas de librería para producir el código que realmente se ejecuta en la máquina.

En la figura 1.2 aparece resumido el contexto en el que un compilador puede trabajar, aunque es necesario tener en cuenta que no han de cumplirse estos pasos estrictamente. En cualquier caso, dependerá del lenguaje que se esté traduciendo y el entorno en el que se trabaje.

4.5. Fases y estructura de un compilador

Podemos distinguir, en el proceso de compilación, dos tareas bien diferenciadas:

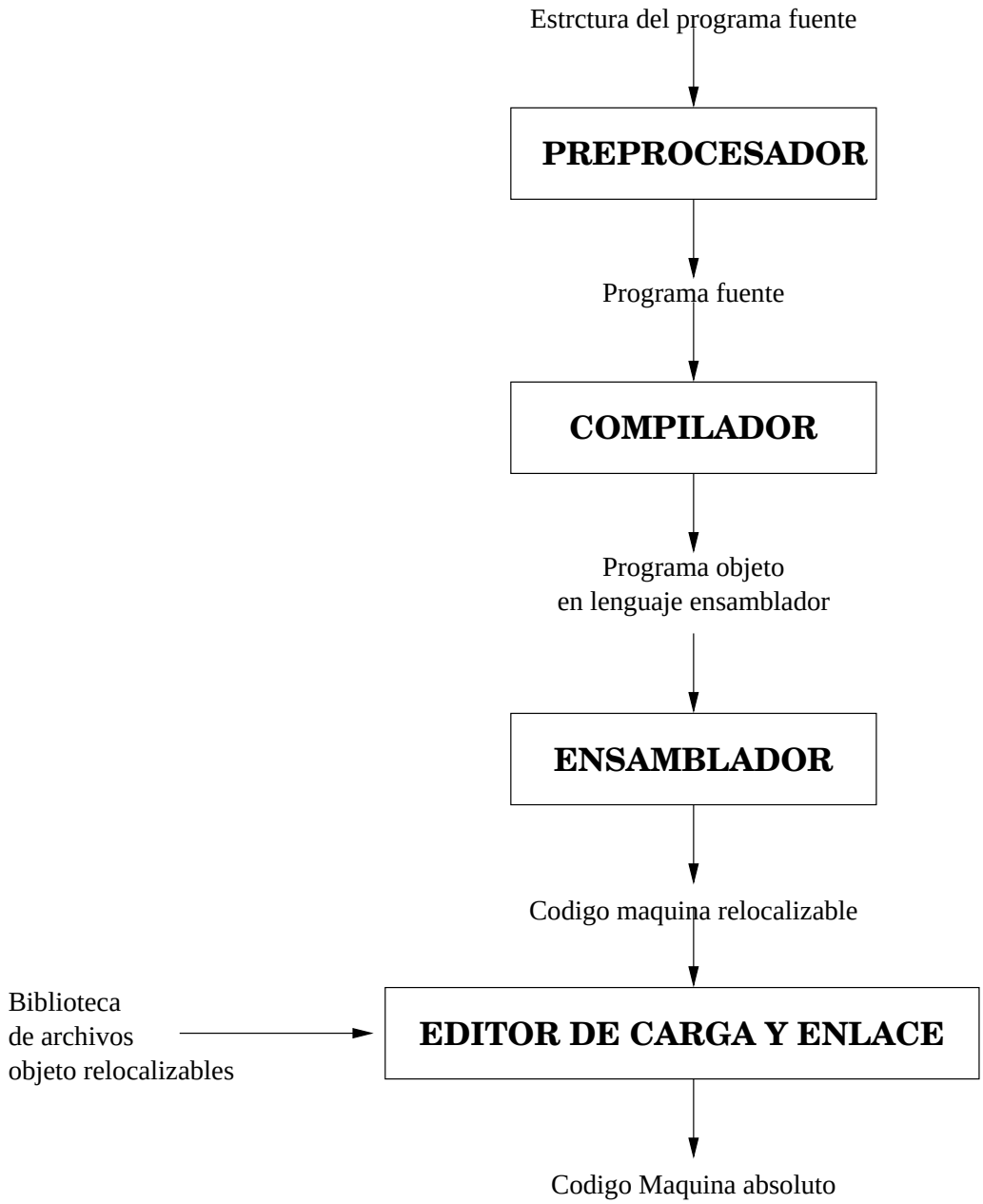


Figura 1.2: Contexto de un compilador

- *Análisis*: Se determina la estructura y el significado de un código fuente. Esta parte del proceso de compilación divide al programa fuente en sus elementos componentes y crea una representación intermedia de él, llamada *árbol sintáctico*.

- *Síntesis*: Se traduce el código fuente a un código de máquina equivalente, a partir de esa representación intermedia. Aquí, es necesario usar técnicas mas especializadas que durante el análisis.

Conceptualmente, un compilador opera en estas dos etapas, que a su vez pueden dividirse en varias fases. Estas pueden verse en la figura 1.3, dónde se muestra la descomposición típica de un compilador. En la práctica, sin embargo, se pueden agrupar algunas de estas fases, y las representaciones intermedias entre ellas pueden no ser construidas explícitamente.

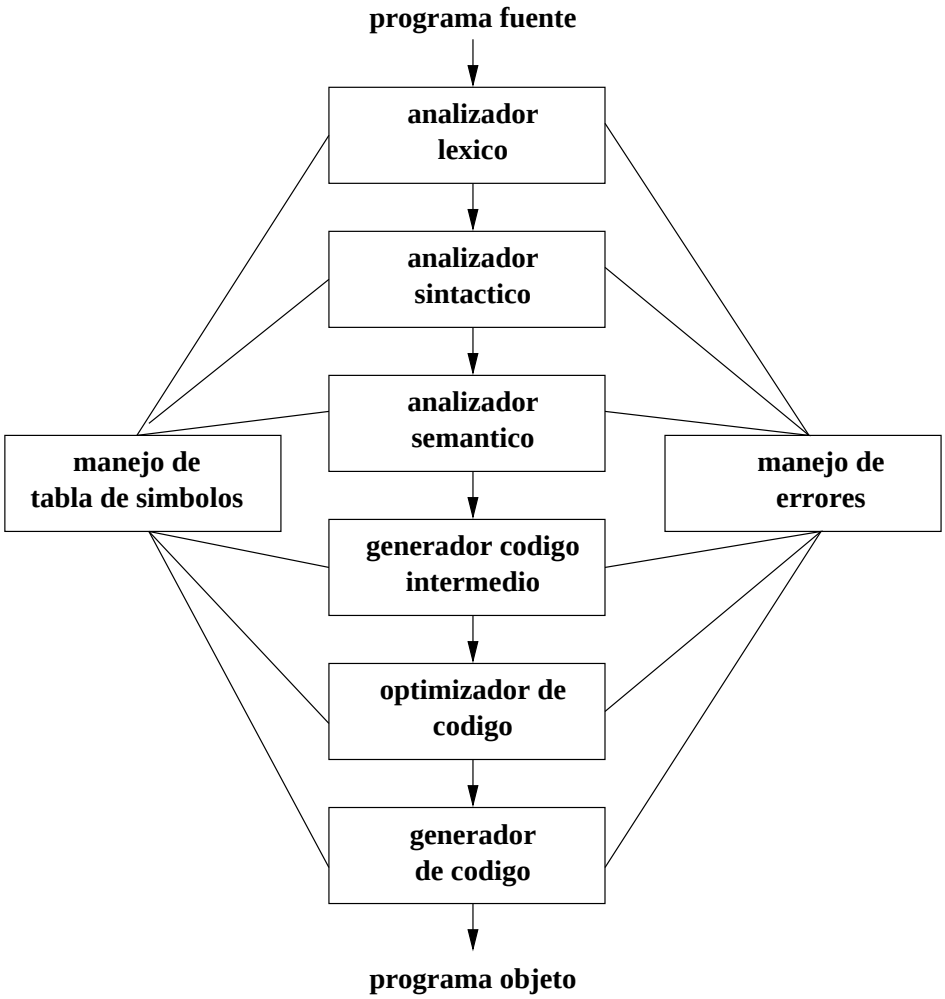


Figura 1.3: Fases de un compilador

Las tres primeras fases de la figura 1.3 conforman la mayor parte de la tarea de *análisis* en un compilador, mientras que las tres últimas pueden considerarse como constituyentes de la parte de *síntesis* del mismo.
Durante el *análisis léxico*, la cadena de caracteres que constituye el programa fuente, se lee de izquierda a derecha, y se agrupa en componentes léxicos, que son secuencias de caracteres con un significado colectivo.
En el *análisis sintáctico*, los componentes léxicos se agrupan jerárquicamente en colecciones anidadas con un significado común.
En la fase de *análisis semántico* se realizan ciertas revisiones para asegurar que los componentes de un programa se ajustan de un modo significativo.
Las *tres últimas fases* suelen variar de un compilador a otro. Existen, por ejemplo, compiladores que no generan código intermedio, o no lo optimizan y pasan directamente del análisis semántico a la generación de código.
De manera informal, también se consideran fases al *administrador de la tabla de símbolos* y al *manejador de errores*, que están en interacción con todas las demás:

- *Administración de la tabla de símbolos*: Una función esencial de un compilador es registrar los identificadores utilizados en el programa fuente y reunir información sobre los distintos atributos de cada identificador. Estos atributos pueden proporcionar información sobre la memoria asignada a un identificador, su tipo, su ámbito (la parte del programa dónde

tiene validez), y, en el caso de los procedimientos, cosas como el número y tipo de sus argumentos, el método por que que cada argumento es pasado (valor, referencia,...) y el tipo que devuelve, si lo hay.

Una *tabla de símbolos* es una estructura de datos que contiene un registro por cada identificador, con campos para los atributos del identificador. La estructura de datos debe permitir encontrar rápidamente datos de ese registro.

Cuando el analizador léxico detecta un identificador en el programa fuente, este identificador se introduce en la tabla de símbolos. Sin embargo, normalmente los atributos de un identificador no se pueden determinar durante el análisis léxico. Por ejemplo, cuando el analizador léxico reconoce los componentes léxicos de la declaración de PASCAL

```
var x, y, z : real;
```

no relaciona unos componentes con otros, y, por tanto, no puede establecer el significado de la frase (x, y y z son variables reales).

Las fases restantes introducen información sobre los identificadores en la tabla de símbolos, y después la utilizan de varias formas. Por ejemplo, cuando se está haciendo el análisis semántico y la generación de código intermedio, se necesita conocer los tipos de los identificadores, para poder comprobar si el programa fuente los usa de una forma válida, y, así, poder generar las operaciones apropiadas con ellos. El generador de código, por lo general, introduce y utiliza información detallada sobre la memoria asignada a los identificadores.

- *Detección e información de errores*: Cada fase dentro de proceso de compilación, puede encontrar errores. Sin embargo, después de detectar un error, cada fase debe tratar de alguna forma ese error, para poder continuar la compilación, permitiendo la detección de nuevos errores en el programa fuente. Un compilador que se detiene cuando encuentra el primer error, no resulta tan útil como debiera. Las fases de análisis sintáctico y semántico, por lo general, manejan una gran porción de errores detectables por el compilador.

  La *fase de análisis léxico* puede detectar errores donde los caracteres restantes de la entrada no forman ningún componente léxico del lenguaje.

  Los errores dónde la cadena de componentes léxicos viola las reglas de la estructura del lenguaje (sintaxis) son determinados por la *fase de análisis sintáctico*.

  Durante la *fase de análisis semántico*, el compilador intenta detectar construcciones que tengan la estructura sintáctica correcta, pero que no tengan significado para la operación implicada. Por ejemplo, se cometería un error semántico si se intentaran sumar dos identificadores, uno de los cuales fuera el nombre de una matriz, y el otro el nombre de un procedimiento.

4.5.1. *Análisis léxico (o lineal)*

Es la primera fase de la que consta un compilador. La parte del compilador que realiza el análisis léxico se llama *analizador léxico* (AL), scanner o explorador. La tarea básica que realiza el AL es transformar un flujo de caracteres de entrada en una serie de componentes léxicos o *tokens*. Se encargaría, por tanto, de reconocer identificadores, palabras clave, constantes, operadores, etc.
La secuencia de caracteres que forma el token se denomina *lexema*. No hay que confundir el concepto de token con el de lexema. A un mismo token le pueden corresponder varios lexemas. Por ejemplo, se pueden reconocer como tokens de tipo ID a todos los identificadores. Aunque para analizar sintácticamente una expresión, sólo nos hará falta el código de token, el lexema

debe ser recordado, para usarlo en fases posteriores dentro del proceso de compilación. El AL es el único componente del compilador que tendrá acceso al código fuente. Por tanto, debe de encargarse de almacenar los lexemas para que puedan ser usados posteriormente. Esto se hace en la *tabla de símbolos*. Por otro lado, debe enviar al analizador sintáctico, aparte del código de token reconocido, la información del lugar dónde se encuentra almacenado ese lexema (por ejemplo, mediante un apuntador a la posición que ocupa dentro de la tabla de símbolos). Posteriormente, en otras fases del compilador, se irá completando la información sobre cada item de la tabla de símbolos.

Por ejemplo, ante la sentencia de entrada

```
coste = precio * 0'98
```

el AL podría devolver una secuencia de parejas, como la siguiente:

```
[ID,1] [=,] [ID,2] [*,] [CONS,3]
```

dónde ID, =, * y CONS corresponderían a códigos de tokens y los números a la derecha de cada pareja sería índices de la tabla de símbolos.
Si durante la fase de análisis léxico, el AL se encuentra con uno o más lexemas que no corresponden a ningún token válido, debe dar un mensaje de *error léxico* e intentar recuperarse.
Finalmente, puesto que el AL es el único componente del compilador que tiene contacto con el código fuente, debe encargarse de eliminar los símbolos no significativos del programa, como espacios en blanco, tabuladores, comentarios, etc.

Es conveniente siempre separar esta fase de la siguiente (análisis sintáctico), por razones de eficiencia. Además, esto permite el uso de representaciones diferentes del programa fuente, sin tener que modificar el compilador completo.

4.5.2. *Análisis sintáctico (o jerárquico)*

Esta es la segunda fase de la que consta un compilador. La parte del compilador que realiza el análisis sintáctico se llama *analizador sintáctico* o parser. Su función es revisar si los tokens del código fuente que le proporciona el analizador léxico aparecen en el orden correcto (impuesto por la gramática), y los combina para formar unidades gramaticales , dándonos como salida el *árbol de derivación* o *árbol sintáctico* correspondiente a ese código fuente.
De la forma de construir este árbol sintáctico se desprenden los dos tipos de analizadores sintácticos existentes:

- Cuando se parte del axioma de la gramática y se va descendiendo, utilizando derivaciones más a la izquierda, hasta conseguir la cadena de entrada, se dice que el *análisis* es *descendente*
- Por el contrario, cuando se parte de la cadena de entrada y se va generando el árbol hacia arriba mediante reducciones más a la izquierda (derivaciones más a la derecha), hasta conseguir la raíz o axioma, se dice que el *análisis* es *ascendente*.

Si el programa no tiene una estructura sintáctica correcta, el analizador sintáctico no podrá encontrar el árbol de derivación correspondiente y deberá dar mensaje de *error sintáctico*.
La división entre análisis léxico y sintáctico es algo arbitraria. Generalmente se elige una división que simplifique la tarea completa del análisis. Un factor para determinar cómo realizarla es comprobar si una construcción del lenguaje fuente es inherentemente recursiva o no. Las construcciones léxicas no requieren recursión, mientras que las sintácticas suelen requerirla.

Las *gramáticas libres de contexto* (GLC) formalizan la mayoría de las reglas recursivas que pueden usarse para guiar el análisis sintáctico. Es importante destacar, sin embargo, que la mayor parte de los lenguajes de programación pertenecen realmente al grupo de *lenguajes dependientes del contexto*.

4.5.3. *Análisis semántico*

Para que la definición de un lenguaje de programación sea completa, aparte de las especificaciones de su sintaxis (estructura o forma en que se escribe un programa), necesitamos también especificar su semántica (significado o definición de lo que realmente hace un programa).
La sintaxis de un lenguaje de programación se suele dividir en componentes libres de contexto y sensibles al contexto. La *sintaxis libre de contexto* define secuencias legales de símbolos, independientemente de cualquier noción sobre el contexto o circunstancia particular en que aparecen dichos símbolos. Por ejemplo, una sintaxis libre de contexto puede informarnos de que $A := B + C$ es una sentencia legal, mientras que $A := B*$ no lo es.
Sin embargo, no todos los aspectos de un lenguaje de programación pueden ser descritos mediante este tipo de sintaxis. Este es el caso, por ejemplo, de las reglas de alcance para variables, de la compatibilidad de tipos, etc. Estos son *componentes sensibles al contexto* de la sintaxis que define al lenguaje de programación. Por ejemplo, $A := B + C$ podría no ser legal si las variables no están declaradas, o son de tipos incompatibles.
Puesto que en la mayoría de los casos, como ya apuntamos en la sección anterior, se utilizan por simplicidad GLC para especificar la sintaxis de los lenguajes de programación, tenemos que hacer un tratamiento especial con las restricciones sensibles al contexto. Estas pasarán a formar parte de la semántica del lenguaje de programación.
La fase de análisis semántico revisa el programa fuente para tratar de encontrar *errores semánticos*, y reúne la información sobre los tipos para la fase posterior de generación de código. Para esto se utiliza la estructura jerárquica que se construye en la fase de análisis sintáctico, para, por ejemplo, identificar operadores y operandos de expresiones y proposiciones. Además, accede, completa y actualiza con frecuencia la tabla de símbolos.
Una tarea importante a realizar en esta fase es la *verificación de tipos*. Aquí, el compilador comprueba si cada operador tiene operandos permitidos por la especificación del lenguaje fuente. Muy frecuentemente, esta especificación puede permitir ciertas *conversiones de tipos* en los operandos, por ejemplo, cuando un operador aritmético binario se aplica a un número entero y a otro real. En este caso, el compilador puede requerir la conversión del número entero a real, por ejemplo.
Resumiendo, algunas de las comprobaciones que puede realizar, son:

- Chequeo y conversión de tipos.
- Comprobación de que el tipo y número de parámetros en la declaración de funciones coincide con los de las llamadas a esa función.
- Comprobación del rango para índices de arrays.
- Comprobación de la declaración de variables.
- Comprobación de las reglas de alcance de variables.

4.5.4. *Generación de código*

La generación de código constituye la última fase dentro del proceso de compilación. Después de examinar el código fuente y comprobar que es correcto desde el punto de vista léxico, sintáctico

y semántico, se debe llevar a cabo la traducción del programa fuente al *programa objeto*. Este consiste, normalmente, en un programa equivalente escrito en un lenguaje máquina o ensamblador. Por equivalente queremos decir que tiene el mismo significado, es decir, que produce los mismos resultados que nuestro programa fuente original.
El árbol de derivación obtenido como resultado del análisis sintáctico, junto con la información contenida en la tabla de símbolos, se usa para la construcción del código objeto. Existen varios métodos para conseguir esto. Uno de ellos, que es particularmente efectivo y elegante, es el que se conoce como *traducción dirigida por la sintaxis*. Esta consiste básicamente en asociar a cada nodo del árbol de derivación una cadena de código objeto. El código correspondiente a un nodo se construye a partir del código de sus descendientes y del código que representa acciones propias de ese nodo. Por tanto, se puede decir que este método es ascendente, pues parte de las hojas del árbol de derivación y va generando código hacia arriba, hasta que llegamos a la raíz del árbol. Esta representa el símbolo inicial de la gramática y su código asociado será el programa objeto deseado.
A veces, el proceso de generación de código se puede dividir en las siguientes fases:

- **Generación de código intermedio**

  Algunos compiladores generan una representación intermedia explícita del programa fuente tras la etapa de análisis. Esta representación intermedia se puede considerar como un programa para una máquina abstracta, y debe cumplir dos propiedades:

  - Debe ser fácil de producir.
  - Debe ser fácil de traducir a código objeto.

  En general, las representaciones intermedias deben hacer algo más que calcular expresiones; también deben manejar construcciones de flujo de control y llamadas a procedimientos. El código generado a partir del intermedio suele ser, por lo general, menos eficiente que el código máquina generado directamente, debido al nivel de traducción adicional.

- **Optimización del código**

  La fase de optimización de código trata de mejorar el código intermedio, de modo que finalmente se obtenga un código máquina más eficiente en tiempo de ejecución. Hay mucha variación en la cantidad de optimización de código que ejecutan los distintos compiladores. En los que realizan muchas operaciones de optimización, denominados *compiladores optimizadores*, una parte significativa del tiempo del compilador se ocupa en esta tarea. Sin embargo, hay optimizaciones sencillas que mejoran sensiblemente el tiempo de ejecución del programa objeto, sin necesidad de retardar demasiado la compilación.

  A veces, a causa del tiempo requerido en esta fase, hay compiladores que no la llevan a cabo y pasan directamente a la generación de código objeto. De hecho, en muchos casos, también se suele suprimir la fase de generación de código intermedio, aunque ésta tiene otras utilidades. Suele ser usual que el compilador ofrezca al usuario la posibilidad de desactivar la opción de optimización del generador de código durante la fase de desarrollo o depuración de programas.

  La generación de código óptimo es un problema NP-completo, y, por tanto, incluso los compiladores optimizadores no tienen por qué producir código óptimo. Es decir, no debemos malinterpretar el término optimización, pues al tratarse de un problema NP-completo, sólo supone, en general, la obtención de código mejorado, pero esto no significa que sea el mejor código posible.

- **Generación de código objeto**

La fase final del compilador es la generación de código objeto, que, por lo general, consiste en código de máquina reubicable o código ensamblador. Para cada una de las variables usadas por el programa se seleccionan posiciones de memoria. Después, cada una de las instrucciones intermedias se traduce a una secuencia de instrucciones máquina que ejecutarán la misma tarea. Una aspecto muy importante a tener en cuenta es la asignación de variables a registros.

Si durante el proceso de compilación se ha generado código intermedio, y se ha pasado por la fase de optimización, sólo quedaría general el código objeto correspondiente al código intermedio optimizado. En otro caso, podría generarse directamente código objeto después del análisis semántico. Incluso puede realizarse al mismo tiempo que el análisis sintáctico y semántico (*compiladores de una pasada*).

En cualquier caso, existen varias posibilidades en cuanto al formato que puede tener el código objeto:

- Generar directamente código máquina, que estaría, por tanto, listo para ejecutarse en la máquina correspondiente. En este caso, debe resolverse, entre otras cuestiones, la de reservar memoria para los identificadores que aparezcan en el programa. Esto hace necesario construir un mapa de direcciones que asocie a cada identificador su correspondiente dirección en memoria.
- Generar código en lenguaje ensamblador de la máquina destino. Posteriormente, habría que traducirlo, mediante un ensamblador, a código objeto reubicable. Este, haciendo uso del cargador-linkador, se transformaría en código ejecutable. Esta forma de generar código es más sencilla, y permite poder compilar por separado distintos programas que pueden interactuar entre sí, usando librerías de rutinas, etc. De hecho, esta técnica es muy común en compiladores que trabajan bajo entorno UNIX, aunque en otros casos se evita, por hacer más ineficiente el proceso de compilación.

En cualquier caso, la generación de código es una tarea complicada, que requiere profundos conocimientos del hardware de la máquina destino, con objeto de aprovechar al máximo los recursos de la misma para que el programa ejecutable resulte lo más eficiente posible.

4.5.5. *Un ejemplo sencillo*

En la figura 1.4 se esquematiza un ejemplo de traducción de la proposición:

```
posicion := inicial + velocidad * 60
```

siguiendo cada una de las fases del proceso de compilación, desde el análisis léxico hasta la generación de código en lenguaje ensamblador. Se supone que las constantes no se almacenan en la tabla de símbolos. Por otro lado, se realiza una conversión de tipos (la constante entera 60 se convierte a real), dentro del análisis semántico. Asimismo, se genera código intermedio de tres direcciones, que es optimizado antes de generar el código en lenguaje ensamblador.

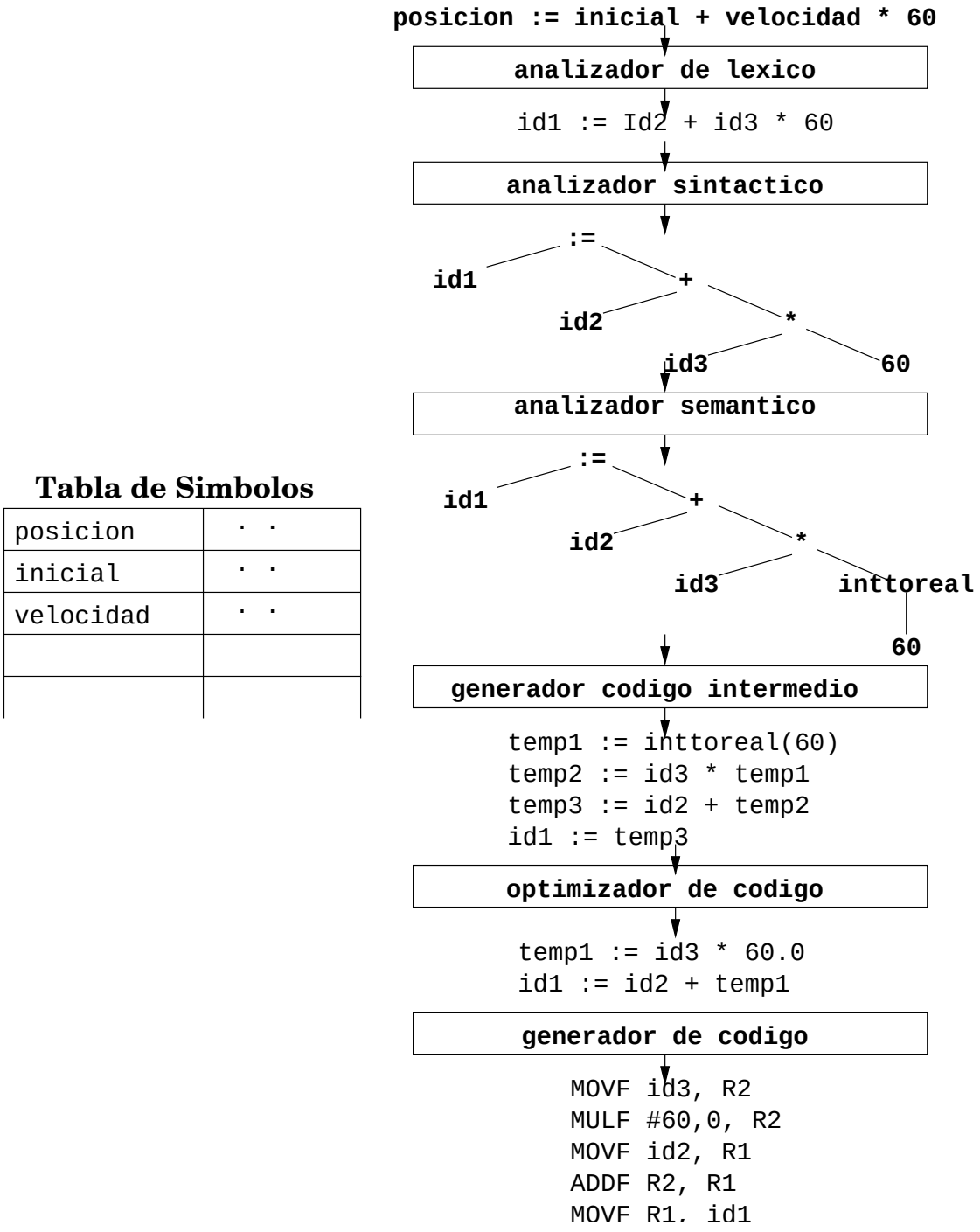


Figura 1.4: Traducción de una sentencia

EJERCICIOS RESUELTOS

---

1. Sea $L = \{\lambda, a\}$. Obtener L^n para $n = 0, 1, 2, 3$. ¿Cuantos elementos tiene L^n, en general? Describir por comprensión L^+.

$L^0 = \{\lambda\}$ $\qquad$ $L^1 = L \cdot \text{Ł}^0 = \{\lambda, a\} \cdot \{\lambda\} = \{\lambda, a\}$

$L^2 = L \cdot \text{Ł}^1 = \{\lambda, a\} \cdot \{\lambda, a\} = \{\lambda, a, aa\}$ $\qquad$ $L^3 = L \cdot \text{Ł}^2 = \{\lambda, a\} \cdot \{\lambda, a, aa\} = \{\lambda, a, aa, aaa\}$

Para todo $n \geq 0$ se tiene que $|L^n| = n + 1$. Podemos definir la clausura positiva de L como:

$$L^+ = \bigcup_{n=1}^{\infty} L^n = \{a^m \mid m \geq 0\}$$

---

2. Sean los lenguajes $A = \{a\}$ y $B = \{b\}$. Describir $(AB)^*$ y $(AB)^+$.

$$(AB)^* = \{(ab)^n \mid n \geq 0\} = \{\lambda, ab, abab, ababab, \ldots\}$$
$$(AB)^+ = \{(ab)^n \mid n > 0\} = \{ab, abab, ababab, \ldots\}$$

---

3. Demostrar que la concatenación de lenguajes no es distributiva respecto de la intersección.

No se cumple que para tres lenguajes cualesquiera $A \cdot (B \cap C) = (A \cdot B) \cap (A \cdot C)$. Lo vamos a demostrar con un contraejemplo. Sean los lenguajes $A = \{a, \lambda\},\ B = \{\lambda\},\ C = \{a\}$. Tenemos que:

$$A \cdot (B \cap C) = \{a, \lambda\} \cdot (\{\lambda\} \cap \{a\}) = \varnothing$$
$$(A \cdot B) \cap (A \cdot C) = \{a, \lambda\} \cap \{aa, a\} = \{a\}$$

Como vemos, se obtienen resultados diferentes. Luego la concatenación no es distributiva respecto de la intersección.

---

4. Dadas dos cadenas x e y sobre V, demostrar que $|xy| = |x| + |y|$ (*).

Primero definimos por inducción la longitud de una cadena, de forma que:

$$1)\ |\lambda| = 0,\ |a| = 1, \forall\, a \in V$$
$$2)\ |wa| = |w| + 1$$

Ahora demostramos (*) por inducción. Para el caso **base** cuando y tienen longitud cero o uno, se cumple (*) por la definición inductiva. Por hipótesis de **inducción** suponemos que (*) se cumple para toda palabra x de cualquier longitud y para toda palabra y de longitud $0 \leq |y| \leq n$. Ahora consideramos una palabra cualquiera y de longitud $n + 1$. Entonces y tendrá al menos un símbolo, de forma que $y = wa$ y por la definición inductiva tenemos que $|y| = |w| + 1$. También por definición se tiene que $|xy| = |xwa| = |xw| + 1$. Pero $|xw| = |x| + |w|$ puesto que se cumple la hipótesis de inducción para w por tener longitud n. En definitiva tenemos que:

$$|xy| = |xwa| = |xw| + 1 = |x| + |w| + 1 = |x| + |y|, \text{ c.q.d.}$$

---

5. Sea el alfabeto $V = \{0, 1\}$ y los lenguajes:

$$L_1 = \{w \in \{0,1\}^* \mid ceros(w) \text{ es par}\}$$
$$L_2 = \{w \in \{0,1\}^* \mid w = 01^n, n \geq 0\}$$

Demostrar que la concatenación L_1L_2 es el lenguaje: $L = \{w \in \{0,1\}^* \mid ceros(w) \text{ es impar}\}$

Tenemos que demostrar que $L_1 \cdot L_2 \subseteq L$ y que $L \subseteq L_1 \cdot L_2$

$\boxed{L_1 \cdot L_2 \subseteq L}$ Se cumple ya que la concatenación de una palabra de L_1 con otra de L_2 nos da una palabra con un número impar de 0's. En efecto, una palabra de L_1 tiene un número par de ceros y una palabra de L_2 sólo tiene un cero al principio y va seguida de

cualquier número de unos. Por tanto al concatenar las dos palabras, la palabra resultante tendrá un número impar de ceros.

$\boxed{L \subseteq L_1 \cdot L_2}$ Se cumple que cada palabra w con un número impar de 0's puede obtenerse como concatenación de una palabra de L_1 seguida de una palabra de L_2. Haremos una **demostración por casos** revisando todas las posibles formas de la palabra: que termine en 0 o que termine en 1.

a) Supongamos que $w = x0$. Entonces x debe tener un número par de ceros, por tanto:

$$w = \underbrace{x}_{\in L_1} \cdot \underbrace{0}_{\in L_2}$$

b) Supongamos que $w = x1$. Nos fijamos en el último cero de x (tiene que haberlo a la fuerza) y partimos la cadena x de forma $x = z_1 \cdot z_2$ donde z_1 llega hasta el último cero de x (no incluido) y por tanto z_2 empezará con un cero e irá seguida de cero o más unos. Por tanto:

$$w = x \cdot 1 = \underbrace{z_1}_{\in L_1} \cdot \underbrace{z_2 \cdot 1}_{\in L_2}$$

---

6. Sea G una g.l.c. con $V_N = \{S\}, V_T = \{a, b\}, P = \{S \to aSb \mid \lambda\}$. Demostrar formalmente que $L(G)$ es el lenguaje L definido como:

$$L = \{a^n b^n \mid n \geq 0\}$$

Para probar que $L(G)$ es realmente el lenguaje que se indica, tenemos que probar dos cosas:

$\boxed{L \subseteq L(G)}$ Hay que demostrar que $\forall\, n \geq 0$ la cadena $w = a^n b^n \in L(G)$ y lo vamos a hacer por inducción sobre n. **Base:** $(n = 0)$, la cadena $a^0 b^0 = \lambda \in L(G)$, ya que $S \Rightarrow \lambda$ (aplicando la regla $S \to \lambda$). **Inducción:** suponemos que $a^n b^n \in L(G)$ y vamos a demostrar que $a^{n+1} b^{n+1} \in L(G)$. En efecto, tenemos que (1) $S \Rightarrow aSb$ (aplicando la regla $S \to aSb$) y (2) $S \overset{*}{\Rightarrow} a^n b^n$ por hipótesis. Luego de (1) y (2) se deduce que

$$S \overset{*}{\Rightarrow} aSb \overset{*}{\Rightarrow} aa^n b^n b$$

Y por la propiedad transitiva de la relación de derivación se tiene que $S \overset{*}{\Rightarrow} a^{n+1} b^{n+1}$. Es decir $a^{n+1} b^{n+1} \in L(G)$, c.q.d.

$\boxed{L(G) \subseteq L}$ Todas las formas sentenciales que no son sentencias, son de la forma $a^n S b^n$, por aplicación de la regla $S \to aSb$ n veces. Y la única forma de llegar a obtener una sentencia es aplicando en el último paso de derivación, la regla $S \to \lambda$ a la forma sentencial $a^n S b^n$, obteniéndose así la sentencia $a^n b^n$. Luego todas las sentencias siguen el patrón $a^n b^n, \forall\, n \geq 0$, y esto significa que $L(G) \subseteq L$, c.q.d.

---

7. Sea el lenguaje $L = \{a^n b^n c^n \mid n \geq 0\}$. Encontrar una gramática que genere L y decir de qué tipo es.

Este lenguaje no es libre del contexto y se demostrará en el capítulo 8. Una gramática que genere este lenguaje puede ser la gramática G siguiente:

$$S \to abDSc \mid \lambda$$
$$bDa \to abD$$
$$bDb \to bbD$$
$$bDc \to bc$$

$\boxed{L \subseteq L(G)}$ Vamos a ver que para cualquier $n \geq 0$, la palabra $a^n b^n c^n$ es derivable de S. Para $n = 0$ está claro pues $S \Rightarrow \lambda$. Para $n > 0$, aplicando n veces la regla $S \to abDSc$ tenemos que $S \overset{*}{\Rightarrow} (abD)^n S c^n$ y aplicando ahora la regla $S \to \lambda$ se obtiene la forma sentencial $(abD)^n c^n$, que tiene n a's, n b's y n c's, pero están "descolocadas". Para conseguir generar la sentencia $a^n b^n c^n$ tenemos que aplicar el resto de reglas empezando a sustituir por la D más a la derecha. Por ejemplo, para $n = 2$ tendríamos:

$$S \overset{*}{\Rightarrow} abDabDcc \Rightarrow abDabcc \Rightarrow aabDbcc \Rightarrow aabbDcc \Rightarrow aabbcc$$

Siguiendo este proceso se puede generar cualquier palabra del tipo $a^n b^n c^n$.

$\boxed{L(G) \subseteq L}$ Como hemos visto, la única forma de añadir terminales a una forma sentencial es aplicando la regla $S \to abDSc$ repetidas veces. El resto de reglas hacen desaparecer una variable de la forma sentencial (como es el caso de $S \to \lambda$ o la regla $bDc \to bc$), o bien, cambian los terminales de posición en la forma sentencial. Una vez que se aplica la regla $S \to \lambda$ a una forma sentencial, dicha forma sentencial tendrá n a's, n b's y n c's y las únicas sentencias que se pueden generar, si aplicamos una secuencia correcta de reglas de producción en las que intervenga la variable D, son palabras que siguen el patrón $a^n b^n c^n$.

La gramática G de este ejemplo es una gramática con estructura de frase (tipo 0). G no es sensible al contexto aunque $L(G)$ sí es sensible al contexto. Esto quiere decir que debe existir una gramática G' equivalente a G que es sensible al contexto.

EJERCICIOS PROPUESTOS

Se proponen los siguientes ejercicios para resolver en pizarra.

1. Encontrar una gramática libre del contexto y otra equivalente regular para cada uno de los dos lenguajes siguientes:

$$L_1 = \{ab^n a \mid n \geq 0\} \quad \| \quad L_2 = \{0^n 1 \mid n \geq 0\}$$

2. Los lenguajes L_3 y L_4 siguientes son libres del contexto. Encontrar gramáticas que los generen.

$$L_3 = \{0^m 1^n \mid m \geq n \geq 0\} \quad \| \quad L_4 = \{0^k 1^m 2^n \mid n = k + m\}$$

3. Dado el lenguaje $L_5 = \{z \in \{a, b\}^* \mid z = ww\}$. Describir una gramática (no puede ser libre del contexto) que lo genere y justificar la respuesta.

4. Clasificar las siguientes gramáticas (dadas por sus reglas de producción) y los lenguajes generados por ellas, haciendo una descripción por comprensión de los mismos.

   a) $\{S \to \lambda \mid A,\ A \to AA \mid c\}$

   b) $\{S \to \lambda \mid A,\ A \to Ad \mid cA \mid c \mid d\}$

   c) $\{S \to c \mid ScS\}$

   d) $\{S \to AcA,\ A \to 0,\ Ac \to AAcA \mid ABc \mid AcB,\ B \to A \mid AB\}$

5. Dada la gramática cuyas producciones son:

$$\begin{aligned} S &\to 0B \mid 1A \\ A &\to 0 \mid 0S \mid 1AA \\ B &\to 1 \mid 1S \mid 0BB \end{aligned}$$

   Demostrar que $L(G) = \{w \in \{0,1\}^* \mid ceros(w) = unos(w) \wedge |w| > 0\}$.

6. Probar que si $L = \{w \in \{0,1\}^* \mid ceros(w) \neq unos(w)\}$ entonces se cumple que $L^* = \{0,1\}^*$

7. Dada la siguiente definición inductiva del lenguaje L sobre el alfabeto $\{a, b\}$:

   1) $\lambda \in L$
   2) Si $w \in L$ entonces $a\,w\,b \in L$ y $b\,w\,a \in L$
   3) Si $x, y \in L$ entonces $x\,y \in L$

   Describir el lenguaje L por comprensión y comprobar que el lenguaje descrito se ajusta a la definición inductiva.

CUESTIONES BREVES

1. ¿Se cumple la propiedad distributiva de la concatenación respecto de la diferencia de lenguajes?

2. Dada la gramática cuyas producciones son $S \to \lambda \mid aSa \mid bSb$, ¿genera la gramática el lenguaje de los palíndromos sobre el alfabeto $\{a, b\}$?

3. Si una gramática G no tiene ninguna producción de la forma $A \to a$, ¿podemos afirmar que G no es regular?

4. Dados dos lenguajes A, B sobre cierto alfabeto V, ¿es cierto que $(A \cdot B)^R = B^R \cdot A^R$?

5. Dar una definición inductiva de w^R.

NOTAS BIBLIOGRÁFICAS

- La parte de alfabetos y lenguajes puede consultarse en el libro de [Kel95] (capítulo 1).
- La parte de gramáticas formales puede consultarse en [Alf97] (capítulo 3), aunque utiliza una notación ligeramente diferente a la nuestra. Otro libro que sigue nuestra notación es el [Lin97] (capítulo 1).

En este capítulo hemos seguido la clasificación original de Chomsky de gramáticas y lenguajes. En otros libros se da una definición diferente (aunque equivalente) de las gramáticas regulares y sensibles al contexto.

CAPÍTULO 2:
EXPRESIONES REGULARES

Contenidos Teóricos

1. Definición de expresión regular (ER)
2. Lenguaje descrito por una expresión regular
3. Propiedades de las expresiones regulares
4. Derivada de una expresión regular
5. Ecuaciones de expresiones regulares
6. Expresiones regulares y gramáticas regulares

1. Definición de expresión regular

Dado un alfabeto V, los símbolos $\varnothing$, λ y los operadores $+$ (unión), $\cdot$ (concatenación) y $*$ (clausura), definimos (de forma recursiva) una EXPRESIÓN REGULAR (ER) sobre el alfabeto V como:

1. el símbolo $\varnothing$ es una expresión regular
2. el símbolo λ es una ER
3. cualquier símbolo $a \in V$ es una ER
4. si α y β son ER entonces también lo es $\alpha + \beta$
5. si α y β son ER entonces también lo es $\alpha \cdot \beta$
6. si α es una ER entonces también lo es α^*

Nota El orden de prioridad de los operadores es, de mayor a menor: $*$, $\cdot$, $+$. Este orden puede alterarse mediante paréntesis, de forma análoga a como se hace con las expresiones aritméticas.

Ejemplo 2.1 *$aa + b^*a$ es una e.r sobre el alfabeto $\{a, b\}$ (por simplicidad omitimos el operador $\cdot$) y esta ER es distinta a la ER $(aa + b^*)\, a$. Por otra parte, la cadena $(+b^*a)$ no es una ER sobre $\{a, b\}$.*

2. Lenguaje descrito por una expresión regular

Cada expresión regular α sobre un alfabeto V describe o representa un lenguaje $L(\alpha) \subseteq V^*$. Este lenguaje se define de forma recursiva como:

1. si $\alpha = \varnothing$ entonces $L(\alpha) = \varnothing$
2. si $\alpha = \lambda$ entonces $L(\alpha) = \{\lambda\}$
3. si $\alpha = a$ y $a \in V$ entonces $L(\alpha) = \{a\}$
4. si α y β son ER entonces $L(\alpha + \beta) = L(\alpha) \cup L(\beta)$
5. si α y β son ER entonces $L(\alpha \cdot \beta) = L(\alpha) \cdot L(\beta)$
6. si α^* es una ER entonces $L(\alpha^*) = (L(\alpha))^*$

Ejemplo 2.2 *Dado $V = \{0, 1\}$ y la ER $\alpha = 0^*10^*$, tenemos que:*

$$L(0^*10^*) = L(0^*) \cdot L(1) \cdot L(0^*) = (L(0))^* \cdot L(1) \cdot (L(0))^* = \{0\}^* \cdot \{1\} \cdot \{0\}^* = \{0^n 10^m \mid n, m \geq 0\}$$

3. Propiedades de las expresiones regulares

Decimos que dos expresiones regulares α y β son *equivalentes*, y lo notamos como $\alpha = \beta$, si describen el mismo lenguaje, es decir, si $L(\alpha) = L(\beta)$.
A continuación enumeramos una serie de propiedades que cumplen las expresiones regulares, derivadas de las propiedades de las operaciones con lenguajes:

1. $\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma$	9. $\lambda^* = \lambda$
2. $\alpha + \beta = \beta + \alpha$	10. $\varnothing^* = \lambda$
3. $\alpha + \varnothing = \alpha$	11. $\alpha \cdot \alpha^* = \alpha^* \cdot \alpha$
4. $\alpha + \alpha = \alpha$	12. $\alpha^* = \alpha^* \cdot \alpha^* = (\alpha^*)^*$
5. $\alpha \cdot \lambda = \alpha$	13. $\alpha^* = \lambda + \alpha \cdot \alpha^*$
6. $\alpha \cdot \varnothing = \varnothing$	14. $(\alpha + \beta)^* = (\alpha^* + \beta^*)^*$
7. $\alpha \cdot (\beta \cdot \gamma) = (\alpha \cdot \beta) \cdot \gamma$	15. $(\alpha + \beta)^* = (\alpha^* \cdot \beta^*)^* = (\alpha^* \cdot \beta)^* \cdot \alpha^*$
8. $\alpha \cdot (\beta + \gamma) = \alpha\beta + \alpha\gamma,\ (\beta + \gamma) \cdot \alpha = \beta\alpha + \gamma\alpha$	16. $\alpha \cdot (\beta \cdot \alpha)^* = (\alpha \cdot \beta)^* \cdot \alpha$

Nota Si tenemos dos expresiones regulares tales que $L(\beta) \subseteq L(\alpha)$ entonces se cumple que $\alpha + \beta = \alpha$.

Estas propiedades muestran ciertas equivalencias que se cumple entre expresiones regulares. Por tanto, la demostración de cada propiedad se haría demostrando que se cumple la igualdad de los lenguajes descritos por las expresiones regulares equivalentes.

Ejemplo 2.3 *Para demostrar la propiedad $\varnothing^* = \lambda$ basta probar que $L(\varnothing^*) = L(\lambda)$. En efecto, teniendo en cuenta la definición de lenguaje descrito por una ER tenemos que:*

$$L(\varnothing^*) = (L(\varnothing))^* = \bigcup_{n=0}^{\infty} \varnothing^n = \varnothing^0 = \{\lambda\} = L(\lambda), \quad \textit{c.q.d.}$$

Las propiedades de las expresiones regulares son útiles porque en algunos casos nos permiten simplificar una ER (ver ejercicio 2 en la sección de aplicaciones).

4. Derivada de una expresión regular

Sea α una ER sobre cierto alfabeto V y sea $a \in V$. La *derivada de α respecto del símbolo a*, y lo notamos como $D_a(\alpha)$, es una expresión regular que describe el siguiente lenguaje:

$$\boxed{L(D_a(\alpha)) = \{w \in V^* \mid a \cdot w \in L(\alpha)\}}$$

En realidad, lo que estamos haciendo al derivar α respecto de un símbolo a, es describir el lenguaje que resulta de eliminar el prefijo a de todas las palabras de $L(\alpha)$. Teniendo esto en cuenta, para calcular la derivada de una expresión regular aplicamos de forma recursiva las siguientes *reglas de derivación*:

1. $D_a(\varnothing) = \varnothing$
2. $D_a(\lambda) = \varnothing$
3. $D_a(a) = \lambda, \quad D_a(b) = \varnothing, \ \forall\, b \in V \ b \neq a$
4. $D_a(\alpha + \beta) = D_a(\alpha) + D_a(\beta)$
5. $D_a(\alpha \cdot \beta) = D_a(\alpha) \cdot \beta + \delta(\alpha) \cdot D_a(\beta) \quad$ donde $\delta(\alpha) = \begin{cases} \varnothing & \text{si } \lambda \notin L(\alpha) \\ \lambda & \text{si } \lambda \in L(\alpha) \end{cases}$
6. $D_a(\alpha^*) = D_a(\alpha) \cdot \alpha^*$

Ejemplo 2.4 *Sea la expresión regular $\alpha = a^*ab$. Vamos a derivar α respecto de a y de b:*

- $D_a(\alpha) = D_a(a^*) \cdot ab + \delta(a^*) \cdot D_a(ab) = D_a(a)\, a^*ab + \lambda\,(D_a(a)\, b + \delta(a)\, D_a(b)) = a^*ab + b = \underbrace{(a^*a + \lambda)}_{(13)} b = a^*b$
- $D_b(\alpha) = D_b(a)\, a^*ab + \lambda\,(D_b(a)\, b + \delta(a)\, D_b(b)) = \varnothing$

  Nota También podemos derivar una expresión regular α respecto de una cadena x de símbolos del alfabeto, teniendo en cuenta que:

$$L(D_x(\alpha)) = \{w \in V^* \mid x \cdot w \in L(\alpha)\}$$

5. Ecuaciones de expresiones regulares

Definición 2.1 *Llamamos* ECUACIÓN DE EXPRESIONES REGULARES *(en forma estándar) con incógnitas o variables $x_1, x_2, \ldots, x_n$ a una ecuación del tipo:*

$$\boxed{x_i = \alpha_{i_0} + \alpha_{i_1} x_1 + \ldots + \alpha_{i_n} x_n}$$

donde cada coeficiente α_{i_j} es una expresión regular.

Puede ser que alguno de los coeficientes sea $\alpha_{i_j} = \varnothing$, en cuyo caso el término para la incognita x_j no aparece en la ecuación y α_{i_0} es el término independiente. Una solución para x_i es una expresión regular.

Definición 2.2 *A una ecuación de la forma* $\boxed{x = \alpha x + \beta}$ *donde α y β son expresiones regulares, la llamaremos* ECUACIÓN FUNDAMENTAL *de expresiones regulares.*

Lema 2.1 (de Arden) *Se puede probar que* $\boxed{x = \alpha^*\beta}$ *es una soluciónpara la ecuación fundamental y esta solución es única si* $\lambda \notin L(\alpha)$. *En otro caso la ecuación tiene infinitas soluciones de la forma* $\boxed{x = \alpha^*(\beta + \gamma)}$ *donde* γ *es cualquier* ER

> Nota Aunque la ecuación fundamental tenga infinitas soluciones, se tiene que $\alpha^*\beta$ es la *menor solución* o menor punto fijo de la ecuación. Esto quiere decir que no existe otra expresión regular r que sea solución y cumpla que $L(r)$ sea subconjunto propio de $L(\alpha^*\beta)$.

En la figura 2.1 mostramos un algoritmo resuelve **sistemas de ecuaciones** de expresiones regulares. El algoritmo toma como entrada n ecuaciones de ER con n incógnitas $x_1, x_2, \ldots, x_n$ y proporciona como salida una solución para cada variable. El método es similar al de eliminación gaussiana: primero diagonalizamos el sistema para dejarlo triangular inferior, de forma que la primera ecuación sea fundamental y luego se realiza una sustitución progresiva de las soluciones obtenidas para la primera variable en adelante.

Entrada: n ecuaciones de ER con n incógnitas $x_1, x_2, \ldots, x_n$
Salida: una solución para cada incognita x_i

1. $i \leftarrow n$; {comenzamos a tratar la última ecuación}
2. while $i \geq 2$ {bucle de diagonalización}
3. expresar ecuación para x_i como $x_i = \alpha x_i + R$ {R es la suma del resto de términos}
4. obtener $x_i \leftarrow \alpha^* R$;
5. desde $j = i - 1$ hasta 1 sustituir en ecuación para x_j la variable x_i por $\alpha^* R$;
6. $i \leftarrow i - 1$;
7. end-while
8. $i \leftarrow 1$; {comenzamos a tratar la primera ecuación}
9. while $i \leq n$ {bucle de sustitución progresiva}
10. obtener solución $x_i \leftarrow \alpha^*\beta$; {la ecuación para x_i ya es fundamental}
11. desde $j = i + 1$ hasta n sustituir en ecuación para x_j la variable x_i por $\alpha^*\beta$;
12. $i \leftarrow i + 1$;
13. end-while

Figura 2.1: Algoritmo de resolución de sistemas de ecuaciones de ER

Ejemplo 2.5 *Vamos a resolver el sistema de ecuaciones de expresiones regulares sobre el alfabeto* $\{0, 1\}$*:*

$$\begin{aligned} x_1 &= \lambda + 1x_1 + 0x_2 \\ x_2 &= 1x_2 + 0x_3 \\ x_3 &= 0x_1 + 1x_3 \end{aligned}$$

Tenemos un sistema de 3 ecuaciones con 3 incógnitas. Las ecuaciones no contienen todos los términos, por ejemplo, a la ecuación para x_2 *le falta el término independiente y el término para* x_1*. Según el algoritmo primero tenemos que diagonalizar el sistema:*

$$\begin{aligned} x_1 &= \lambda + 1x_1 + 0x_2 \\ x_2 &= 0 \cdot 1^*0x_1 + 1x_2 \\ x_3 &= 1^* \cdot 0x_1 \end{aligned} \quad \dashrightarrow \quad \begin{aligned} x_1 &= \lambda + (1 + 0 \cdot 1^*01^*0)\, x_1 \\ x_2 &= 1^* \cdot 01^*0x_1 \\ x_3 &= 1^*0x_1 \end{aligned}$$

Ahora obtenemos soluciones y sustituimos variables por soluciones:

$$x_1 = (1 + 01^*01^*0)^*$$
$$x_2 = 1^*01^*0\,(1 + 01^*01^*0)^*$$
$$x_3 = 1^*0\,(1 + 01^*01^*0)^*$$

6. Expresiones regulares y gramáticas regulares

En esta sección vamos a probar que el lenguaje descrito por una expresión regular es un lenguaje que puede ser generado por una gramática regular, esto es, es un lenguaje regular. Y por otra parte, todo lenguaje regular veremos que puede ser descrito por una expresión regular. Para ello veremos dos métodos:

- $ER \longrightarrow GR$ para pasar de una expresión regular a una gramática regular
- $GR \longrightarrow ER$ para obtener una expresión regular a partir de una gramática regular

6.1. Cálculo de la gramática a partir de la expresión regular $\boxed{ER \longrightarrow GR}$

Dada una expresión regular α sobre cierto alfabeto $V = \{a_1, \ldots, a_k\}$, vamos a aplicar el *método de las derivadas* para obtener una gramática regular G tal que $L(\alpha) = L(G)$:

1. S es el símbolo inicial de G que asociaremos a la expresión regular α
2. Inicialmente $V_N = \{S\}$, $V_T = V$, $P = \varnothing$
3. Obtenemos las reglas de producción para S de la siguiente forma:
   - *a)* Si $\lambda \in L(\alpha)$ entonces añadimos a P la regla $S \to \lambda$
   - *b)* Desde $i = 1$ hasta k
     1) calculamos $D_{a_i}(S)$ y si $\lambda \in L(D_{a_i}(S))$ entonces añadimos a P la regla $S \to a_i$
     2) si $D_{a_i}(S) \neq \lambda, \varnothing$ entonces añadimos la regla $S \to a_i A_i$ donde A_i es la variable que asociamos a $D_{a_i}(S)$ y la añadimos a V_N (si es nueva)
4. Obtenemos reglas para el resto de variables de la gramática por derivadas sucesivas hasta que no podamos añadir variables nuevas a la gramática:
   - *a)* Para cada variable B asociada a una ER no derivada y desde $i = 1$ hasta k
     1) calculamos $D_{a_i}(B)$ y si $\lambda \in L(D_{a_i}(B))$ entonces añadimos a P la regla $B \to a_i$
     2) si $D_{a_i}(B) \neq \lambda, \varnothing$ entonces añadimos la regla $B \to a_i C_i$ donde C_i es la variable que asociamos a $D_{a_i}(B)$ y la añadimos a V_N (si es nueva)

Ejemplo 2.6 *Sea la expresión regular $\alpha = aa^*bb^* + ab$ sobre el alfabeto $V = \{a, b\}$. Vamos a calcular la gramática correspondiente por el método de las derivadas:*

1. $D_a(S) = a^*bb^* + b = A$
2. $D_b(S) = \varnothing$
3. $D_a(A) = D_a(a^*bb^* + b) = D_a(a^*) \cdot bb^* + \delta(a^*) \cdot D_a(bb^*) + \varnothing = a^*bb^* = A$, *ya que* $a^*bb^* + b = a^*bb^*$, *puesto que* $L(b) \subseteq L(a^*bb^*)$

4. $D_b(A) = D_b(a^*) \cdot bb^* + \delta(a^*) \cdot D_b(bb^*) + \lambda = b^* + \lambda = b^* = B$

5. $D_a(B) = D_a(b^*) = \varnothing$

6. $D_b(B) = b^* = B$, *y ya no hay variables nuevas para derivar*

   La gramática que se obtiene es la siguiente:

$$\begin{array}{l} S \to aA \\ A \to aA \mid bB \mid b \\ B \to bB \mid b \end{array}$$

6.2. Cálculo de la ER a partir de la gramática $\boxed{GR \longrightarrow ER}$

Supongamos que tenemos una gramática $G = (V_N, V_T, A_1, P)$ lineal derecha (si fuera lineal izquierda se puede obtener otra lineal derecha equivalente, aunque no vamos a ver el método), vamos a aplicar el **método de resolución de ecuaciones** para obtener una expresión regular α tal que $L(G) = L(\alpha)$:

1. Supongamos que $V_N = \{A_1, A_2, \ldots A_n\}$. Obtenemos un sistema de n ecuaciones de ER, una para cada variable de la gramática, teniendo en cuenta las reglas de producción para esa variable. La ecuación para la variable A_i será de la forma:

$$A_i = \alpha_{i_0} + \alpha_{i_1} A_1 + \alpha_{i_2} A_2 + \ldots + \alpha_{i_n} A_n$$

   y los coeficientes se obtienen:

   a) término independiente: Si $A_i \to a_1 \mid \ldots \mid a_k \mid \lambda$, donde cada $a_j \in V_T$, entonces el término independiente será la suma de los terminales y λ, esto es, $\alpha_{i_0} = (a_1 + \ldots + a_k + \lambda)$. Si A_i no deriva en ningún símbolo terminal ni en λ entonces $\alpha_{i_0} = \varnothing$

   b) coeficiente para variable A_j: Si $A_i \to b_1 A_j \mid \ldots \mid b_m A_j$, donde cada $b_j \in V_T$, entonces α_{i_j} será la suma de los terminales que acompañan a A_j, esto es, $\alpha_{ij} = (b_1 + \ldots + b_m)$. Si no tenemos ninguna producción para A_i donde A_j aparezca en la parte derecha entonces $\alpha_{i_j} = \varnothing$

2. Resolvemos el sistema de ecuaciones obtenido en el paso anterior y la solución para A_1 (símbolo inicial) será la expresión regular α tal que $L(G) = L(\alpha)$. En general, la expresión regular solución a una variable A_i describe el conjunto de palabras que pueden generarse a partir de la variable A_i.

Ejemplo 2.7 *Encontrar la expresión regular correspondiente a la siguiente gramática:*

$$\begin{array}{l} S \to aA \mid \lambda \\ A \to aA \mid aC \mid bB \mid aB \\ B \to bB \mid bA \mid b \\ C \to aC \mid bC \end{array}$$

Vamos a obtener el sistema de ecuaciones para la gramática y resolvemos (en realidad no es necesario resolverlo completo, sólo la primera ecuación):

$$\begin{array}{l} S = aA + \lambda \\ A = aA + aC + (b+a)\,B \\ B = bB + bA + b \\ C = (a+b)\,C \end{array} \dashrightarrow \begin{array}{l} S = aA + \lambda \\ A = aA + (b+a)\,B \\ B = bB + bA + b \\ C = (a+b)^* \cdot \varnothing = \varnothing \end{array} \dashrightarrow \begin{array}{l} S = aA + \lambda \\ A = aA + (b+a)\,b^*bA + (b+a)\,b^*b \\ B = b^*\,(bA + b) \end{array}$$

$$
\begin{array}{rl}
 & S = a \cdot (a + (b + a)\, b^*b)^*\, (b + a)\, b^*b + \lambda \\
\dashrightarrow & A = (a + (b + a)\, b^*b)^* \cdot (b + a)\, b^*b \\
 & B = b^* \cdot (bA + b)
\end{array}
$$

Para resumir los resultados expuestos en esta sección enunciamos el siguiente teorema.

Teorema 2.1 *Un lenguaje puede ser generado por una gramática regular si y solo si puede ser descrito por una expresión regular.*

EJERCICIOS RESUELTOS

---

1. Sea la ER $\alpha = a + bc + b^3a$. ¿Cuál es el lenguaje descrito por α? ¿Qué expresión regular corresponde al lenguaje universal sobre el alfabeto $\{a, b, c\}$?

En primer lugar, esta no es estrictamente hablando una ER, ya que no se permite b^3a. Sin embargo, aceptamos como válida la expresión $a + bc + b^3a$, como una simplificación de la ER $a + bc + bbba$. En ese caso, $L(\alpha) = \{a, bc, bbba\}$, que como vemos es un lenguaje finito sobre el alfabeto $\{a, b, c\}$. La ER que describe el lenguaje universal sobre este alfabeto es $(a + b + c)^*$.

---

2. Simplificar la ER $\alpha = a + a\,(b + aa)\,(b^*aa)^*\, b^* + a\,(aa + b)^*$.

Aplicando las propiedades de las expresiones regulares, podemos obtener una ER equivalente con tan sólo 4 operadores:

$$a + a\,(b + aa)\,\underbrace{(b^*aa)^*\, b^*}_{(15)} + a\,(aa + b)^* = \underbrace{a + a\,(b + aa)\,(b + aa)^*}_{(8)} + a\,(aa + b)^* =$$

$$a(\underbrace{\lambda + (b + aa)\,(b + aa)^*}_{(13)}) + a\,(aa + b)^* = a(\underbrace{b + aa}_{(2)})^* + a\,(aa + b)^* =$$

$$\underbrace{a\,(aa + b)^* + a\,(aa + b)^*}_{(4)} = a\,(aa + b)^*$$

---

3. Calcular $D_{ab}(\alpha)$ siendo $\alpha = a^*ab$.

Teniendo en cuenta que $D_{ab}(\alpha) = D_b(D_a(\alpha))$, y que $D_a(\alpha) = a^*b$ (calculada en el ejemplo 2.4), entonces $D_b(a^*b) = D_b(a^*) \cdot b + \delta(a^*) \cdot D_b(b) = \varnothing \cdot b + \lambda \cdot \lambda = \lambda$.

---

4. Demostrar que $D_a(\alpha^*) = D_a(\alpha) \cdot \alpha^*$ (regla de derivación para la clausura).

Podemos afirmar que $\alpha^* = \sum_{n=0}^{\infty} \alpha^n$, porque ambos miembros de la igualdad describen el mismo lenguaje. Teniendo esto en cuenta y según la regla de derivación para la suma de expresiones regulares, se cumple que:

$$D_a(\alpha^*) = D_a(\alpha^0) + D_a(\alpha^1) + D_a(\alpha^2) + D_a(\alpha^3) + \ldots$$

donde $\alpha^0 = \lambda$. Ahora aplicamos la regla de la concatenación a cada término:

$$D_a(\alpha^*) = \varnothing + D_a(\alpha) + D_a(\alpha) \cdot \alpha + \delta(\alpha) \cdot D_a(\alpha) + D_a(\alpha) \cdot \alpha^2 + \delta(\alpha) \cdot D_a(\alpha^2) + \ldots$$

De aquí se pueden eliminar los términos $\delta(\alpha) \cdot D_a(\alpha^i)$, ya que $D_a(\alpha^i)$ siempre tiene que calcularse, con lo que $\delta(\alpha) \cdot D_a(\alpha^i)$ resultaría redundante, independientemente de lo que valga $\delta(\alpha)$. Ahora podemos sacar factor común y queda:

$$D_a(\alpha^*) = D_a(\alpha) \cdot (\lambda + \alpha + \alpha^2 + \alpha^3 + \ldots) = D_a(\alpha) \cdot \alpha^*, \ c.q.d.$$

---

5. Demostrar que $x = \alpha^*\beta$ es una solución para la ecuación fundamental $x = \alpha x + \beta$ y razonar por qué la ecuación fundamental puede tener en algunos casos infinitas soluciones.

Para probar que es una solución tenemos que sustituir x por $\alpha^*\beta$ en ambos miembros de la ecuación y ver que se cumple la igualdad. En efecto:

$$\alpha^*\beta = \alpha\alpha^*\beta + \beta = (\alpha^*\alpha + \lambda)\beta = \alpha^*\beta, \ c.q.d.$$

Según el lema de Arden, la ecuación puede tener infinitas soluciones cuando $\lambda \in L(\alpha)$ y estas soluciones deben ser de la forma $\alpha^*(\beta + \gamma)$. Comprobemos que es solución:

$$\alpha^*(\beta + \gamma) = \alpha(\alpha^*(\beta + \gamma)) + \beta = \alpha\alpha^*\beta + \alpha\alpha^*\gamma + \beta = \alpha^*\beta + \alpha\alpha^*\gamma$$

La igualdad se cumple sólo en el caso de que $\alpha^*\gamma = \alpha\alpha^*\gamma$, pero dado que γ puede ser cualquier ER, debe ser $\alpha^* = \alpha\alpha^*$, y para que esto se cumpla es necesario que $\lambda \in L(\alpha)$, como afirma el lema de Arden.

---

6. Simplificar la expresión regular $1^*01^*0\,(01^*01^*0 + 1)^*\,01^* + 1^*$ de forma que sólo aparezca un operador $+$.

$$1^*01^*0\underbrace{(01^*01^*0 + 1)^*}_{(15)}01^* + 1^* = \underbrace{1^*01^*0\,(1^* \cdot 01^*01^*0)^*}_{(16)}1^* \cdot 01^* + 1^* =$$

$$\underbrace{(1^*01^*0 \cdot 1^*0)^*\,1^*01^*01^*01^* + 1^*}_{(8)} = \left(\underbrace{(1^*01^*01^*0)^*\,1^*01^*01^*0 + \lambda}_{(13)}\right)1^* =$$

$$\underbrace{(1^* \cdot 01^*01^*0)^*\,1^*}_{(15)} = (1 + 01^*01^*0)^*$$

EJERCICIOS PROPUESTOS

1. Obtener la ER correspondiente a la siguiente gramática y aplicar el método de las derivadas a la expresión regular obtenida:

$$S \to aA \mid cA \mid a \mid c$$
$$A \to bS$$

2. Obtener la gramática que genera el lenguaje descrito por la ER $\alpha = (b + ab^*a)^*ab^*$

3. Comprobar que la equivalencia $(b + ab^*a)^*ab^* = b^*a(b + ab * a)^*$

4. Dada la expresión regular $(ab + aba)^*$: aplicar el método de las derivadas para obtener la gramática y resolver el sistema de ecuaciones de la gramática obtenida.

5. Dada una ER α sobre cierto alfabeto V, demostrar que si $\alpha^2 = \alpha$ entonces $\alpha^* = \alpha + \lambda$

6. Dada la expresión regular $\alpha = a(bc)^*(b + bc) + a$: obtener G a partir de α y resolver el sistema de ecuaciones para G.

7. Obtener la expresión regular equivalente a la siguiente gramática:

$$S \to bA \mid \lambda$$
$$A \to bB \mid \lambda$$
$$B \to aA$$

8. Obtener la expresión regular que describe el lenguaje generado por la gramática:

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

9. Aplicar el método de las derivadas para calcular la gramática correspondiente a la expresión regular $(d(ab)^*)^* da(ba)^*$

10. Demostrar que para cualquier expresión regular se cumple $\alpha^* = \alpha^*\alpha + \lambda$

CUESTIONES BREVES

1. ¿Pertenece *acdcdb* al lenguaje descrito por la expresión regular $(a(cd)^*b)^* + (cd)^*$?

2. Si L es el lenguaje formado por todas las cadenas sobre $\{a, b\}$ que tienen al menos una ocurrencia de la subcadena b, ¿podemos describir L mediante la expresión regular $a^*(ba^*)^*bb^*(b^*a^*)^*$?

3. Dada cualquier expresión regular α, ¿se cumple que $\alpha^*\alpha = \alpha^*$?

4. Dadas α, β, γ ER cualesquiera, ¿es cierto que $\alpha + (\beta \cdot \gamma) = (\alpha + \beta) \cdot (\alpha + \gamma)$?

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

NOTAS BIBLIOGRÁFICAS

- Para este tema el libro básico que recomendamos es el de [Isa97] (capítulo 3).
- La teoría sobre ecuaciones de expresiones regulares y el método para obtener la expresión regular a partir de la gramática puede consultarse en el libro de [Aho72] (pag. 105). El lema de Arden aparece en [Alf97] (pag. 166) y [Kel95] (pag.79).

CAPÍTULO 3:
AUTÓMATAS FINITOS

Contenidos Teóricos

1. Arquitectura de un autómata finito (AF)
2. Autómatas finitos deterministas
3. Autómatas finitos no deterministas
4. Autómatas finitos con λ-transiciones
5. Lenguaje aceptado por un AF
6. Equivalencia entre autómatas finitos
7. Autómatas finitos, expresiones regulares y gramáticas regulares
8. Minimización de un AFD
9. Aplicaciones: análisis léxico

1. Arquitectura de un autómata finito (AF)

Un autómata finito es una estructura matemática que representa un sistema o máquina abstracta cuya arquitectura puede verse en la figura 3.1

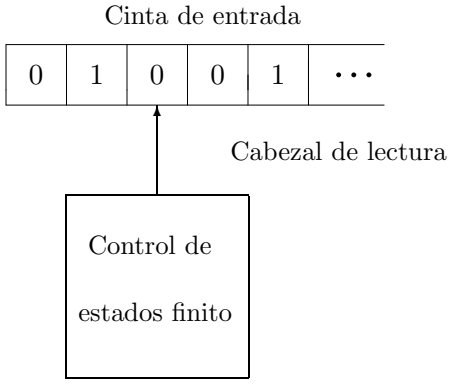


Figura 3.1: Arquitectura de un AF

La *cinta de entrada* (que se extiende infinitamente hacia la derecha) está dividida en celdas, cada una de las cuales es capaz de almacenar un sólo símbolo de un cierto alfabeto. La máquina es capaz de leer los símbolos de esta cinta de izquierda a derecha por medio de un *cabezal de lectura*. Cada vez que se lee un símbolo, el cabezal de lectura se mueve a la siguiente celda a la derecha y la máquina efectúa un cambio de estado o *transición*. Esta transición está determinada por

el *mecanismo de control* (que contiene un número finito de estados), programado para conocer cual debe ser el nuevo estado, que dependerá de la combinación del estado actual y el símbolo de entrada leído.
Los autómatas finitos pueden considerarse como mecanismos *aceptadores* o reconocedores de palabras. De manera informal decimos que un autómata finito aceptará una palabra de entrada si, comenzando por un estado especial llamado *estado inicial* y estando la cabeza de lectura apuntando al primer símbolo de la cadena, la máquina alcanza un *estado final* o de aceptación después de leer el último símbolo de la cadena.

2. Autómatas finitos deterministas

Un AUTÓMATA FINITO DETERMINISTA (AFD) se define como una quintupla $M = (Q, V, \delta, q_0, F)$, donde:

Q es un *conjunto finito de estados*
V es el *alfabeto de entrada*
q_0 es el *estado inicial*
$F \subseteq Q$ es el *conjunto de estados finales*
$\delta : Q \times V \longrightarrow Q$ es la *función de transición*

El nombre "determinista" viene de la forma en que está definida la función de transición: si en un instante t la máquina está en el estado q y lee el símbolo a entonces, en el instante siguiente $t + 1$ la máquina cambia de estado y sabemos con seguridad cual es el estado al que cambia, que es precisamente $\delta(q, a)$.
El AFD es **inicializado** con una palabra de entrada w como sigue:

1. w se coloca en la cinta de entrada, con un símbolo en cada celda
2. el cabezal de lectura se apunta al símbolo más a la izquierda de w
3. el *estado actual* pasa a ser q_0

Una vez que se ha inicializado el AFD, comienza su "ejecución" sobre la palabra de entrada. Como cualquier computador tiene un **ciclo de ejecución básico**:

1. se lee el *símbolo actual*, que es el apuntado por el cabezal de lectura. Si el cabezal apunta a una celda vacía entonces el AFD termina su ejecución, *aceptando* la palabra en caso de que el estado actual sea final y *rechazando* la palabra en caso contrario. Esto ocurre cuando se ha leído toda la palabra de entrada, y se produce una situación similar a tener una condición "fin de fichero" en la ejecución de un programa
2. se calcula el *estado siguiente* a partir del estado actual y del símbolo actual según la función de transición, esto es, $\delta(estado\ actual, simbolo\ actual) = estado\ siguiente$
3. el cabezal de lectura se mueve una celda a la derecha
4. el estado siguiente pasa a ser el estado actual y vuelve al paso 1

La función de transición de un AFD se puede **representar** de dos formas: mediante una tabla de transición o mediante un diagrama de transición.

Tabla de transición	Cada fila corresponde a un estado $q \in Q$ El estado inicial se precede del símbolo $\rightarrow$ Cada estado final se precede del símbolo # Cada columna corresponde a un símbolo de entrada $a \in V$ En la posición (q, a) está el estado que determine $\delta(q, a)$
Diagrama de transición	Los nodos se etiquetan con los estados El estado inicial tiene un arco entrante no etiquetado Los estados finales están rodeados de un doble círculo Habrá un arco etiquetado con a desde el nodo q_i al q_j si $\delta(q_i, a) = q_j$

Ejemplo 3.1 *Supongamos que tenemos el autómata finito determinista dado por*

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\})$$

donde la función $\delta : \{q_0, q_1, q_2\} \times \{0, 1\} \longrightarrow \{q_0, q_1, q_2\}$ *viene dada por*

$$\begin{array}{ll} \delta(q_0, 0) = q_0 & \delta(q_0, 1) = q_1 \\ \delta(q_1, 0) = q_0 & \delta(q_1, 1) = q_2 \\ \delta(q_2, 0) = q_2 & \delta(q_2, 1) = q_1 \end{array}$$

La tabla de transición correspondiente a este autómata será:

δ	0	1
$\rightarrow q_0$	q_0	q_1
$\# \; q_1$	q_0	q_2
q_2	q_2	q_1

y el diagrama de transición correspondiente se muestra en la figura 3.2.

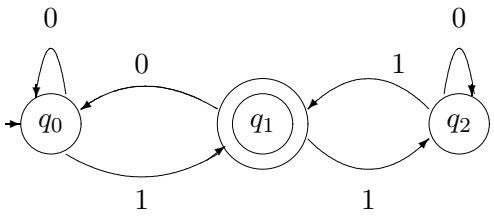


Figura 3.2: Diagrama de transición del ejemplo 3.1

Nota El diagrama de transición de un AFD tiene por cada nodo un sólo arco etiquetado con cada uno de los símbolos del alfabeto. Algunos autores consideran que la función de transición puede ser *parcial*, es decir, no estar definida para algún $\delta(q, a)$. En ese caso se dice que el AFD es *incompleto*, y en el diagrama de transición faltarían los arcos correspondientes a los casos no definidos de la función de transición. Nosotros consideraremos que los $AFDs$ son *completos*.

3. Autómatas finitos no deterministas

Un AUTÓMATA FINITO NO DETERMINISTA ($AFND$) es una quintupla $M = (Q, V, \Delta, q_0, F)$ donde todos los componentes son como en los $AFDs$, excepto la función de transición que se define ahora como:

$$\boxed{\Delta : Q \times V \longrightarrow \mathcal{P}(Q)}$$

donde $\mathcal{P}(Q)$ denota el conjunto de las partes de Q (o conjunto potencia 2^Q).
El hecho de que el codominio de la función de transición sea $\mathcal{P}(Q)$ es lo que añade esta característica de "no determinismo": a partir del estado actual y del símbolo actual de entrada no se puede determinar de forma exacta cuál será el estado siguiente. Por ejemplo, podemos tener $\Delta(q, a) = \{q_1, q_2, \ldots, q_m\}$ y esto indica que dado el estado actual q y el símbolo de entrada a, el estado siguiente puede ser cualquier estado entre q_1 y q_m. También puede darse el caso de que $\Delta(q, a) = \varnothing$, lo que indica que el estado siguiente no está definido.
Intuitivamente, un $AFND$ *acepta* una palabra de entrada w siempre que sea posible comenzar por el estado inicial y que exista una secuencia de transiciones que nos lleven a consumir la palabra y acabe el autómata en un estado final. Puede que tengamos otras secuencias de transiciones que no acaben en estado final, pero basta que exista una que acabe en estado final para que la palabra sea aceptada.
Los $AFND$ también se representan mediante tablas o diagramas de transición. En el diagrama de transición, el no determinismo se descubre porque hay algún nodo del que parten dos o más arcos etiquetados con el mismo símbolo del alfabeto, o falta algún arco para algún símbolo del alfabeto. En la figura 3.3 podemos ver un ejemplo de tabla y diagrama de transición de un $AFND$.

δ	a	b
$\rightarrow q_0$	$\{q_0, q_3\}$	$\{q_0, q_1\}$
q_1	$\emptyset$	$\{q_2\}$
$\# q_2$	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_4\}$	$\emptyset$
$\# q_4$	$\{q_4\}$	$\{q_4\}$

Tabla de transición

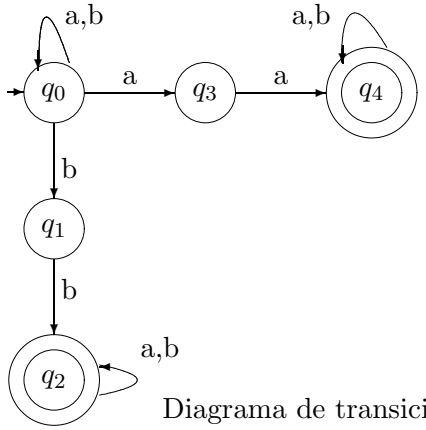


Diagrama de transición

Figura 3.3: Ejemplo de $AFND$

4. Autómatas finitos con λ-transiciones

Un AUTÓMATA FINITO CON λ-TRANSICIONES ($AFND$-λ) es básicamente un $AFND$ al que se le permite cambiar de estado sin necesidad de consumir o leer un símbolo de la entrada. Por eso la función de transición de un $AFND$-λ se define

$$\boxed{\Delta : Q \times (V \cup \{\lambda\}) \longrightarrow \mathcal{P}(Q)}$$

La tabla de transición de un $AFND$-λ es como la de un $AFND$ excepto que se le añade una columna correspondiente a λ, de forma que en la posición $T[(q, \lambda)]$ estará el conjunto de estados que determine $\Delta(q, \lambda)$.

Ejemplo 3.2 *Supongamos que tenemos un $AFND$-λ cuyo diagrama de transición corresponde al de la figura 3.4. Entonces si el autómata está en el estado q_1 en un cierto instante y el símbolo actual es b, en el instante siguiente, el autómata puede decidir de forma no determinista entre "leer el símbolo b y cambiar al estado q_4", o bien, "cambiar al estado q_2 sin mover el cabezal de lectura". Además, el conjunto de cadenas que es capaz de aceptar este autómata es $\{b, bb, bbb\}$.*

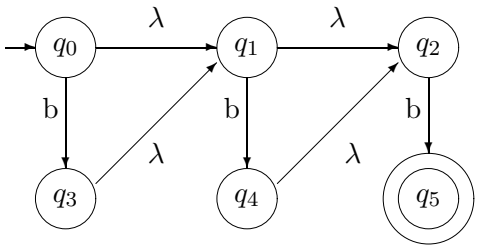


Figura 3.4: Ejemplo de $AFND$-λ

5. Lenguaje aceptado por un AF

Un autómata finito sirve para reconocer cierto tipo de lenguajes. Antes de definir formalmente el concepto de lenguaje aceptado por un AF necesitamos definir los conceptos de *configuración* y *cálculo* en un autómata finito.
La configuración de un autómata finito (sin importar el tipo) en cierto instante viene dada por el estado del autómata en ese instante y por la porción de cadena de entrada que le queda por leer o procesar. La porción de cadena leída hasta llegar al estado actual no tiene influencia en el comportamiento futuro de la máquina. En este sentido podemos decir que un AF es una máquina sin memoria externa; son los estados los que resumen de alguna forma la información procesada.
Formalmente una CONFIGURACIÓN de un AF es un elemento $(q, w) \in (Q \times V^*)$. Algunos tipos de configuraciones especiales son:

- *Configuración inicial*: (q_0, w), donde q_0 es el estado inicial y w la palabra de entrada.
- *Configuración de parada*: cualquier configuración en la que el autómata puede parar su ejecución, bien porque se haya procesado toda la entrada o bien porque se haya llegado a una situación donde no es aplicable ninguna transición.
- *Configuración de aceptación*: (q_F, λ), donde q_F es un estado final del autómata. Una vez alcanzada esta configuración el autómata puede aceptar la palabra.

Si consideramos el conjunto de las configuraciones de un autómata finito, podemos definir una relación binaria $\vdash \subseteq (Q \times V^*) \times (Q \times V^*)$ que llamaremos *relación de cálculo en un paso*. Intuitivamente si dos configuraciones C_i y C_j están relacionadas mediante la relación $\vdash$ y lo notamos como $C_i \vdash C_j$, quiere decir que podemos pasar de la configuración C_i a la C_j aplicando una sola transición y diremos que "la configuración C_i alcanza en un paso la configuración C_j".

Para definir formalmente la relación de cálculo en un paso $\vdash$, distinguiremos tres casos correspondientes a los tres tipos de autómatas que hemos visto:

- Si tenemos un AFD, la relación de cálculo en un paso se define de la siguiente forma:

$$(q, w) \vdash (q', w') \Leftrightarrow \begin{cases} w = aw', \text{ donde } a \in V \\ q' = \delta(q, a) \end{cases}$$

- Si tenemos un $AFND$, la relación de cálculo en un paso la se define:

$$(q, w) \vdash (q', w') \Leftrightarrow \begin{cases} w = aw', \text{ donde } a \in V \\ q' \in \Delta(q, a) \end{cases}$$

- Si tenemos un $AFND$-λ, la relación de cálculo en un paso se define:

$$(q, w) \vdash (q', w') \Leftrightarrow \begin{cases} w = \sigma w', \text{ donde } \sigma \in V \cup \{\lambda\} \\ q' \in \Delta(q, \sigma) \end{cases}$$

  Cuando queramos distinguir el autómata M al que refiere la relación, se usará $\vdash_M$.

La clausura reflexiva y transitiva de la relación $\vdash$ es otra relación binaria $\vdash^* \subseteq (Q \times V^*) \times (Q \times V^*)$, que llamaremos *relación de cálculo*. Diremos que la "configuración C_i alcanza (en cero o más pasos) la configuración C_j", y lo notamos como $C_i \vdash^* C_j$, si se cumple una de las dos condiciones siguientes:

1. $C_i = C_j$, o bien,
2. $\exists\, C_0, C_1, \ldots C_n$, tal que $C_0 = C_i$, $C_n = C_j$, y $\forall\, 0 \leq k \leq n-1$ se cumple que $C_k \vdash C_{k+1}$

A una secuencia del tipo $C_0 \vdash C_1 \vdash \ldots \vdash C_n$ la llamaremos *cálculo en n pasos*, abreviadamente $C_1 \vdash^*_{n\ pasos} C_n$.

Ejemplo 3.3 *Considerando el AFD de la figura 3.2 podemos decir que* $(q_0, 01) \vdash (q_0, 1)$, $(q_0, 1) \vdash (q_1, \lambda)$ *y por tanto* $(q_0, 01) \vdash^* (q_1, \lambda)$. *También* $(q_1, 101) \vdash (q_2, 01)$ *y en varios pasos* $(q_2, 0011) \vdash^* (q_1, 1)$.
Por otra parte para el AFND de la figura 3.3 tenemos, por ejemplo, que $(q_0, abb) \vdash (q_0, bb)$ *y también* $(q_0, abb) \vdash (q_3, bb)$. *Al ser el autómata no determinista vemos que a partir de una misma configuración, en este caso* (q_0, abb), *se puede llegar en un paso de cálculo a dos o más configuraciónes distintas. Esta situación no puede producirse en un AFD.*
Para el AFND-λ *de la figura 3.4 el cálculo* $(q_1, bb) \vdash (q_2, bb)$ *es un ejemplo donde se produce una transición que implica un cambio de estado sin consumir símbolos de entrada. Esto es posible porque* $q_2 \in \Delta(q_1, \lambda)$.

Si tenemos un autómata finito $M = (Q, V, \delta, q_0, F)$, se define el *lenguaje aceptado por M* y lo notamos $L(M)$, como:

$$\boxed{L(M) = \{w \in V^* \mid (q_0, w) \vdash^* (q_F, \lambda) \text{ donde } q_F \in F\}}$$

Es decir, una palabra w será aceptada por el autómata M, si partiendo de la configuración inicial con w en la cinta de entrada, el autómata es capaz de alcanzar una configuración de aceptación. Dependiendo del tipo de autómata de que se trate, $\vdash^*$ hará referencia a la clausura reflexiva y transitiva de la relación $\vdash$ en un AFD, en un $AFND$ o en un AF con λ-transiciones. En un autómata finito determinista, el hecho de que una palabra w sea aceptada por el autómata nos asegura que existe un único camino en el diagrama de transición que nos lleva del nodo etiquetado con el estado inicial al nodo etiquetado con el estado final y cada arco que se recorre

en este camino está etiquetado con un símbolo de la palabra. Podríamos simular la ejecución de un autómata finito determinista mediante un programa que codifique la función de transición y simule los cambios de estado. Si $|w| = n$ entonces el programa puede determinar si la palabra es aceptada o no en $O(n)$.
En el caso de un $AFND$ o un $AFND$-λ no podemos asegurar que exista un único camino en el diagrama que nos lleve del estado inicial a un estado final consumiendo los símbolos de la palabra. Incluso puede que para una palabra $w \in L(M)$ podamos tener una camino que no acabe en estado final o que llegue a un estado desde el que no se pueda seguir leyendo símbolos. Esto es debido al no determinismo, que hace que los cálculos en estos autómatas no estén perfectamente determinados. Si quisiéramos simular un autómata no determinista para decidir si una palabra es aceptada o no, tendríamos que usar alguna técnica de retroceso o *backtracking* para explorar distintas posibilidades hasta encontrar un cálculo correcto que reconozca la palabra o determinar que la palabra no es aceptada si se han explorado todos los posibles cálculos y ninguno de ellos conduce a un estado final. Esto nos llevaría a un algoritmo de tiempo exponencial para reconocer una palabra. De ahí que a efectos prácticos, como en la construcción de analizadores léxicos o reconocimiento de patrones en un texto, lo deseable es tener un autómata finito determinista. Afortunadamente siempre es posible pasar de un AF no determinista a un AF determinista, como veremos en la siguiente sección.

Ejemplo 3.4 *Recordemos los AFs ya vistos y veamos ahora cuál es el lenguaje aceptado por ellos. El diagrama de la figura 3.3 correspondiente a un AFND permite ver de forma intuitiva que $L(M)$ es el lenguaje descrito por la expresión regular $(a+b)^*(aa+bb)(a+b)^*$ que consiste en aquellas cadenas sobre el alfabeto $V = \{a, b\}$ que contienen al menos una ocurrencia de la subcadena* aa *ó* bb. *Por ejemplo, la cadena* abb *es aceptada, ya que tenemos el cálculo:*

$$(q_0, abb) \vdash (q_0, bb) \vdash (q_1, b) \vdash (q_2, \lambda), \quad \text{y } q_2 \in F$$

Sin embargo podemos tener otro cálculo que no conduce a estado final:

$$(q_0, abb) \vdash (q_0, bb) \vdash (q_0, b) \vdash (q_1, \lambda), \quad q_1 \notin F$$

e incluso un cálculo que no llega a consumir la palabra:

$$(q_0, abb) \vdash (q_3, bb) \vdash \text{ (y no puede seguir)}$$

A partir del diagrama del AFD de la figura 3.2 no es tan sencillo ver cuál es el lenguaje aceptado. Pero, según veremos en la sección 7, hay un método exacto para encontrar este lenguaje. En este caso el lenguaje aceptado es el descrito por la expresión regular $(0 + 1\,(10^*1)^*)^*\,1\,(10^*1)^*$

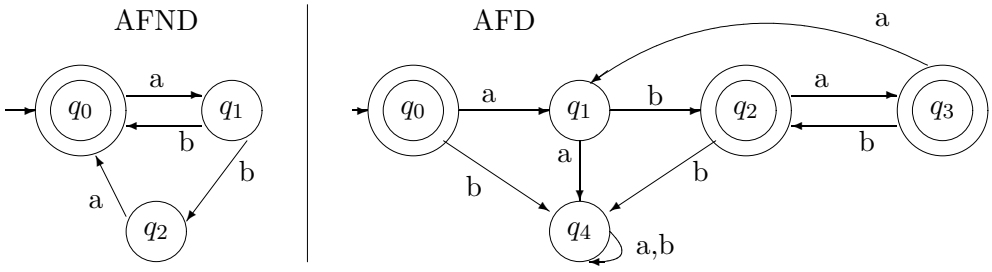


Figura 3.5: AFs que aceptan $L(\alpha)$ donde $\alpha = (ab + aba)^*$

6. Equivalencia entre autómatas finitos

Decimos que dos autómatas finitos M y M' son *equivalentes* si y sólo si aceptan el mismo lenguaje, esto es, $L(M) = L(M')$. Veremos ahora que, en contra de lo que parece, los autómatas no deterministas (con o sin λ-transiciones) son igual de potentes que los autómatas finitos deterministas, en el sentido de que son capaces de reconocer los mismos lenguajes: los lenguajes regulares.

Ya hemos dicho que a efectos de simular la ejecución de un AF con un programa conviene que el autómata sea determinista para poder reconocer las palabras en tiempo polinomial. ¿Qué sentido tienen entonces los autómatas no deterministas? Desde el punto de vista teórico son interesantes porque permiten modelizar algoritmos de búsqueda y retroceso y también son de gran utilidad, sobre todo los $AFND$-λ, para demostrar algunos teoremas sobre autómatas y lenguajes formales. Otra ventaja que supone el uso de autómatas no deterministas es que a veces resulta más intuitivo y sencillo diseñar un autómata no determinista para reconocer un determinado lenguaje, que pensar directamente en el autómata determinista. Un ejemplo lo tenemos en la figura 3.5. Se puede observar que es más sencillo averiguar el lenguaje aceptado a partir del $AFND$ que del AFD.

Para demostrar formalmente que los tres tipos de autómatas vistos aceptan la misma clase de lenguajes, vamos a ver dos teoremas, uno de ellos establece la equivalencia entre los modelos de AFD y $AFND$ y el otro teorema demuestra la equivalencia entre $AFND$ y $AFND$-λ. Las demostraciones de estos teoremas tienen una parte constructiva que muestra un método algorítmico para pasar de un tipo de autómata a otro, y una parte inductiva que prueba la validez de dicho método.

Teorema **3.1** *Un lenguaje es aceptado por un $AFND$ si y sólo si es aceptado por un AFD.*

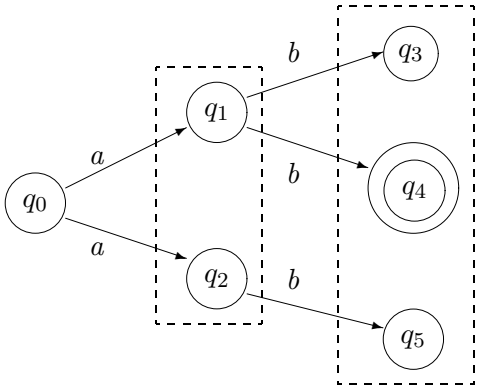
Dem.- Esta claro que un AFD se puede considerar como un caso particular de $AFND$ donde $|\Delta(q,a)| = 1, \forall\, q \in Q,\, a \in V$. Por tanto si un lenguaje L es aceptado por un AFD también será aceptado por un $AFND$.

Supongamos que un lenguaje L es aceptado por un autómata finito no determinista $M_N = (Q, V, \Delta, q_0, F)$. Vamos a construir a partir de él un autómata finito determinista $M_D = (Q', V, \delta, q_0', F')$ que tiene las siguientes componentes:

$$
\begin{array}{l}
Q' = \mathcal{P}(Q) \\
q_0' = \{q_0\} \\
F' = \{S \in \mathcal{P}(Q) \mid S \cap F \neq \varnothing\} \\
\forall\, S \in Q',\, a \in V \text{ se define } \delta(S,a) = \bigcup_{q \in S} \Delta(q,a) = \{p \in Q \mid p \in \Delta(q,a) \wedge q \in S\}
\end{array}
$$

Tenemos pues, que el conjunto de estados del AFD así construido está formado por estados que a su vez son conjuntos de estados del $AFND$. La idea es la siguiente: cuando el $AFND$ lee una cadena w, no sabemos exactamente en que estado quedará el autómata, pero podemos decir que estará en un estado de un posible conjunto de estados, por ejemplo, $\{q_i, q_j, \ldots, q_k\}$. Un AFD equivalente después de leer la misma entrada quedará en un estado que está perfectamente determinado, y para que el autómata AFD sea equivalente haremos que este estado corresponda al conjunto de estados $\{q_i, q_j, \ldots, q_k\}$. Por eso, si originalmente tenemos $|Q|$ estados en el $AFND$, entonces el AFD tendrá como mucho $2^{|Q|}$ estados.

Ejemplo 3.5 *Supongamos que tenemos el $AFND$ siguiente:*



Al leer la palabra $w = ab$ se pueden dar los siguientes cálculos:

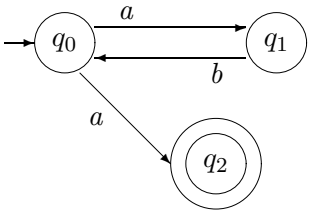
$$(q_0, ab) \vdash_{M_N} \begin{cases} \nearrow (q_1, b) \vdash_{M_N} \begin{cases} \nearrow (q_3, \lambda) \\ \searrow (q_4, \lambda) \end{cases} \\ \searrow (q_2, b) \vdash_{M_N} (q_5, \lambda) \end{cases}$$

El autómata acepta la palabra porque uno de los posibles cálculos conduce a un estado final. Consecuentemente, el AFD deberá reconocer la palabra. De hecho es así y el único cálculo que llega a consumir la palabra y acabar en estado final es:

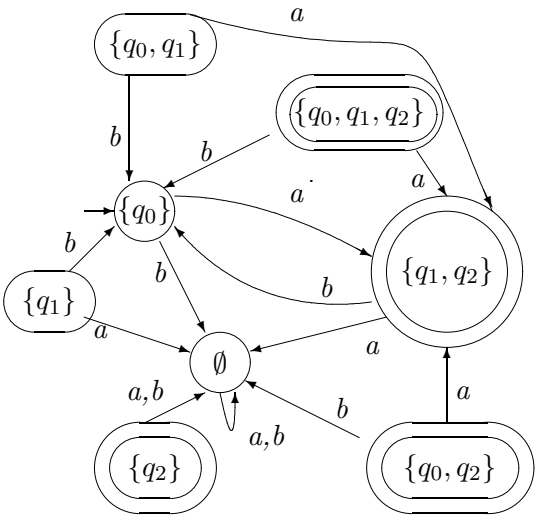
$$(\{q_0\}, ab) \vdash_{M_D} (\{q_1, q_2\}, b) \vdash_{M_D} (\{q_3, q_4, q_5\}, \lambda)$$

Para acabar de demostrar el teorema necesitamos probar que $L(M_D) = L(M_N)$, o lo que es lo mismo, que $w \in L(M_D) \Leftrightarrow w \in L(M_N)$, $\forall\, w \in V^*$. Lo dejamos para la sección de aplicaciones. ■

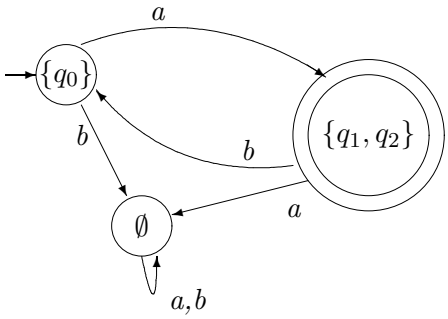
Ejemplo 3.6 *Dado el lenguaje descrito por la expresión regular $(ab)^*a$, un $AFND$ que acepta dicho lenguaje es el siguiente:*



Si aplicamos el método del teorema 3.1 anterior se obtiene un AFD equivalente con 8 estados, correspondientes a los 8 subconjuntos de estados del $AFND$:



En el diagrama anterior podemos ver que hay estados (los enmarcados en un recuadro oval) que son inaccesibles*, es decir, no se puede llegar hasta ellos desde el estado inicial. Por tanto podrían eliminarse del AFD sin que ello afecte al lenguaje aceptado, obteniéndose el siguiente AFD simplificado:*



El método del teorema 3.1 considera todos los posibles elementos de $\mathcal{P}(Q)$, pero ya hemos visto que no siempre son todos necesarios. En la figura 3.6 se presenta un **algoritmo** que calcula un AFD a partir de un $AFND$ siguiendo la misma idea que antes, pero *sin generar estados inaccesibles.*

Teorema 3.2 *Un lenguaje es aceptado por un $AFND$-λ si y sólo si es aceptado por un $AFND$.*

Dem.- Es obvio que un $AFND$ puede considerarse un caso restringido de $AFND$-λ, donde $\Delta(q, \lambda) = \varnothing, \forall\, q \in Q$. Por tanto si un lenguaje L es aceptado por un $AFND$ también será aceptado por un $AFND$-λ.
Se define la λ-*clausura de un estado* q en una $AFND$-λ como:

$$\boxed{\lambda\text{-}clau(q) = \{p \in Q \mid (q, \lambda) \vdash^* (p, \lambda)\}}$$

Esto es, la λ-clausura de un estado nos da el conjunto de estados que se pueden alcanzar siguiendo todos los caminos en el diagrama de transición que parten del nodo q y sólo pasan por arcos etiquetados con λ. También se tiene, por definición, que $q \in \lambda\text{-}clau(q)$. Esta definición

Entrada: Un $AFND$ $M_N = (Q, V, \Delta, q_0, F)$

Salida: Un AFD $M_D = (Q', V, \delta, q'_0, F')$ tal que $L(M_D) = L(M_N)$

1. $Q' \leftarrow \{\{q_0\}\}$; $\quad q'_0 \leftarrow \{q_0\}$;
2. for all $a_i \in V$
3. $\quad \delta(q'_0, a_i) \leftarrow \Delta(q_0, a_i)$;
4. $\quad$ AÑADIR $(Q', \delta(q'_0, a_i))$; /* sólo se añade si es nuevo */
5. end-for;
6. MARCAR (Q', q'_0)); /* para saber que se han calculado sus transiciones */
7. while haya estados no marcados en Q'
8. $\quad$ sea S_j un estado no marcado de Q'
9. $\quad$ for all $a_i \in V$
10. $\quad\quad \delta(S_j, a_i) \leftarrow \bigcup_{q \in S_j} \Delta(q, a_i)$;
11. $\quad\quad$ AÑADIR $(Q', \delta(S_j, a_i))$;
12. $\quad$ end-for;
13. $\quad$ MARCAR (Q', S_j));
14. end-while;
15. $F' = \{S \in Q' \mid S \cap F \neq \varnothing\}$;

Figura 3.6: Algoritmo de paso de $AFND$ a AFD

se puede extender y se define la λ-*clausura de un conjunto de estados* S como:

$$\boxed{\lambda\text{-}clau(S) = \bigcup_{q \in S} \lambda\text{-}clau(q)}$$

Cuando un $AFND$-λ parte de un estado q y lee un símbolo a, el autómata podrá alcanzar un conjunto de estados, que son los estados a los que se puede llegar desde q siguiendo todos los caminos con el siguiente procedimiento:

- llegar a los estados que se pueden alcanzar desde q y sólo pasan por arcos etiquetados con λ. Estos estados son los correspondientes a λ-$clau(q)$
- de todos los estados de λ-$clau(q)$ nos quedamos con aquellos estados r que tienen definida la transición $\Delta(r, a)$
- se calcula el conjunto de estados a los que se puede llegar desde estos estados r consumiendo en un paso el símbolo a y a este conjunto lo llamamos S
- calcular todos los estados alcanzables desde cualquier estado de S pasando sólo por arcos etiquetados con λ. Estos estados son los correspondientes a λ-$clau(S)$

El $AFND$ que simule al $AFND$-λ debe poder alcanzar, para cada estado q y para cada símbolo a, los mismos estados a los que llega el $AFND$-λ a partir de q y leyendo el símbolo a. Por eso la nueva función de transición se calculará teniendo en cuenta el procedimiento anterior. Dado el autómata $M_\lambda = (Q, V, \Delta, q_0, F)$ un $AFND$ equivalente $M_N = (Q, V, \Delta', q_0, F')$ se obtiene definiendo:

- Si $\lambda\text{-}clau(q_0) \cap F \neq \varnothing$ entonces $F' = F \cup \{q_0\}$, en otro caso $F' = F$
- $\forall\, q \in Q$, $a \in V$ se define $\Delta'(q, a) = \lambda\text{-}clau(S)$,
  donde $S = \{p \in Q \mid p \in \Delta(r, a) \wedge r \in \lambda\text{-}clau(q)\}$

Para demostrar que este método es correcto habría que probar que $w \in L(M_\lambda) \Leftrightarrow w \in L(M_N)$, $\forall\, w \in V^*$. Para $w = \lambda$ tenemos que $\lambda \in L(M_\lambda)$ sii $[(q_0, \lambda) \vdash_\lambda^* (p_F, \lambda) \wedge p_F \in F]$. Pero esto se cumple sii $[(p_F = q_0) \vee (p_F \in \lambda\text{-}clau(q_0))]$ sii $(q_0 \in F')$ sii $\lambda \in L(M_N)$, puesto que $(q_0, \lambda) \vdash_{M_N}^* (q_0, \lambda)$. Ahora se debe demostrar que $w \in L(M_\lambda) \Leftrightarrow w \in L(M_N)$ para todas las palabras de longitud mayor o igual que uno. El proceso a seguir es similar al del teorema 3.1 y necesitamos demostrar la hipótesis:

HIPÓTESIS : $\underset{(1)}{(q, w) \vdash_{M_\lambda}^* (p, \lambda)} \quad \Leftrightarrow \quad \underset{(2)}{(q, w) \vdash_{M_N}^* (p, \lambda)}$

Una vez demostrada la hipótesis, tomando $q = q_0$ se tiene claramente que $w \in L(M_\lambda) \Leftrightarrow w \in L(M_N)$. Luego $L(M_\lambda) = L(M_N)$, como queríamos demostrar. La demostración de la hipótesis se deja como ejercicio. ■

Ejemplo 3.7 *Supongamos que queremos reconocer el lenguaje* $L = \left\{0^i 1^j 2^k \mid i, j, k \geq 0\right\}$*, o lo que es lo mismo, el lenguaje descrito por la expresión regular* $0^*1^*2^*$*. Podemos diseñar un autómata finito con* λ*-transiciones* $M_\lambda = (\{q_0, q_1, q_2\}, \{0, 1, 2\}, \Delta, q_0, \{q_2\})$ *y función* Δ *representada en la figura 3.7. El AFND equivalente es* $M_N = (\{q_0, q_1, q_2\}, \{0, 1, 2\}, \Delta', q_0, \{q_0, q_2\})$*, donde* Δ' *se define como:*

$\Delta'(q_0, 0) = \lambda\text{-}clau\{q_0\} = \{q_0, q_1, q_2\}$ $\quad \Delta'(q_1, 0) = \varnothing$ $\quad \Delta'(q_2, 0) = \varnothing$
$\Delta'(q_0, 1) = \lambda\text{-}clau\{q_1\} = \{q_1, q_2\}$ $\quad \Delta'(q_1, 1) = \lambda\text{-}clau\{q_1\} = \{q_1, q_2\}$ $\quad \Delta'(q_2, 1) = \varnothing$
$\Delta'(q_0, 2) = \lambda\text{-}clau\{q_2\} = \{q_2\}$ $\quad \Delta'(q_1, 2) = \lambda\text{-}clau\{q_2\} = \{q_2\}$ $\quad \Delta'(q_2, 2) = \lambda\text{-}clau\{q_2\} = \{q_2\}$

En la figura 3.7 aparecen los diagramas de los dos autómatas finitos. La palabra λ *es aceptada por el AFND-*λ *por el siguiente cálculo:* $(q_0, \lambda) \vdash_{M_\lambda} (q_1, \lambda) \vdash_{M_\lambda} (q_2, \lambda)$ y $q_2 \in F$*. También es aceptada por el AFND ya que* $(q_0, \lambda) \vdash_{M_N}^* (q_0, \lambda)$ *y* $q_0 \in F'$*. Otra palabra aceptada es 012 ya que:*

$$(q_0, 012) \vdash_{M_\lambda} (q_0, 12) \vdash_{M_\lambda} (q_1, 12) \vdash_{M_\lambda} (q_1, 2) \vdash_{M_\lambda} (q_2, 2) \vdash_{M_\lambda} (q_2, \lambda) \text{ y } q_2 \in F$$
$$(q_0, 012) \vdash_{M_N} (q_1, 12) \vdash_{M_N} (q_2, 2) \vdash_{M_N} (q_2, \lambda) \text{ y } q_2 \in F'$$

Como vemos el cálculo es más corto (3 pasos) en el AFND que en el AFND-λ *(5 pasos), ya que debido a las* λ*-transiciones el AFND-*λ *puede hacer "movimientos" sin consumir símbolos.*

7. Autómatas finitos, expresiones regulares y gramáticas regulares

En el tema anterior estudiamos las expresiones regulares y vimos que pueden utilizarse para describir lenguajes regulares generados por gramáticas regulares o de tipo 3, según la clasificación de Chomsky. Ahora vamos a ver que si un lenguaje es aceptado por un autómata finito también se puede describir mediante una expresión regular y al contrario, todo lenguaje descrito por una expresión regular puede ser aceptado por un autómata finito.

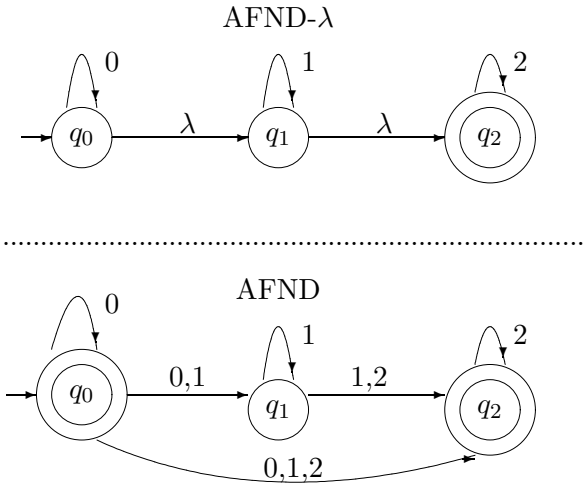


Figura 3.7: Ejemplo de paso de $AFND$-λ a $AFND$

7.1. Teorema de Análisis de Kleene

> Si L es un lenguaje aceptado por un autómata finito M entonces existe una expresión regular α tal que $L = L(M) = L(\alpha)$.

Podemos suponer que el autómata finito M no tiene λ-transiciones (si las tuviera ya sabemos que podemos encontrar autómata equivalente sin λ-transiciones). Sea $M = (Q, V, \Delta, q_0, F)$. A partir de este autómata podemos obtener un sistema de ecuaciones de expresiones regulares que llamaremos *ecuaciones características del autómata*. Estas ecuaciones se obtienen a partir del diagrama de transición del autómata del siguiente modo:

- A cada nodo q_i le corresponde una ecuación y cada estado se puede considerar como una incógnita de la ecuación.
- La ecuación para el estado q_i tiene en el primer miembro el estado q_i y el segundo miembro de la ecuación está formado por una suma de términos, de forma que por cada arco del diagrama de la forma $q_i \xrightarrow{a} q_j$ tenemos un término aq_j. Si el estado q_i es final, añadimos además el termino λ al segundo miembro.

Cada incógnita q_i del sistema de ecuaciones representa el conjunto de palabras que nos llevan del nodo q_i a un estado final, en el diagrama de transición. Por tanto, si resolvemos el sistema de las ecuaciones características del autómata tendremos soluciones de la forma $q_i = \alpha_i$, donde α_i es una expresión regular sobre el alfabeto V y como hemos dicho el lenguaje descrito por esta expresión regular es:

$$L(\alpha_i) = \{w \in V^* \mid (q_i, w) \vdash^* (q_F, \lambda)\,,\ q_F \in F\} \tag{3.1}$$

El **método** que se propone para obtener una expresión regular α a partir de un AF es el siguiente:

1. Obtener las ecuaciones características del autómata;
2. Resolver el sistema de ecuaciones;
3. $\alpha \leftarrow$ solución para el estado inicial;

Para comprobar que este método es válido tendríamos que probar que se cumple 3.1 para toda solución $q_i = \alpha_i$ del sistema de ecuaciones, y en particular la solución para el estado inicial es la expresión regular correspondiente al autómata. Aunque no lo vamos a demostrar formalmente, por la forma de obtener las ecuaciones características del autómata y la validez del método de resolución de sistemas de ecuaciones de expresiones regulares vista en el tema anterior, podemos intuir que el método anterior es correcto. En realidad, no es necesario resolver todas las incógnitas, sólo necesitamos despejar la incognita correspondiente al estado inicial.

Ejemplo 3.8 *Consideremos de nuevo el autómata finito M de la figura 3.2. Las ecuaciones características correspondientes a este autómata son:*

$$\begin{aligned} q_0 &= 0q_0 + 1q_1 \\ q_1 &= 0q_0 + 1q_2 + \lambda \\ q_2 &= 1q_1 + 0q_2 \end{aligned}$$

Comenzando por la última ecuación se tiene que $q_2 = 0^*1q_1$ y sustituyendo en la segunda ecuación queda

$$q_1 = 0q_0 + 10^*1q_1 + \lambda$$

de donde se obtiene que $q_1 = (10^*1)^* (0q_0 + \lambda)$ y este valor se sustituye en la primera ecuación

$$q_0 = 0q_0 + 1 (10^*1)^* (0q_0 + \lambda) = 0q_0 + 1 (10^*1)^* 0q_0 + 1 (10^*1)^*$$

Esta ecuación es fundamental y por el lema de Arden tiene como única solución

$$q_0 = (0 + 1 (10^*1)^* 0)^* 1 (10^*1)^*$$

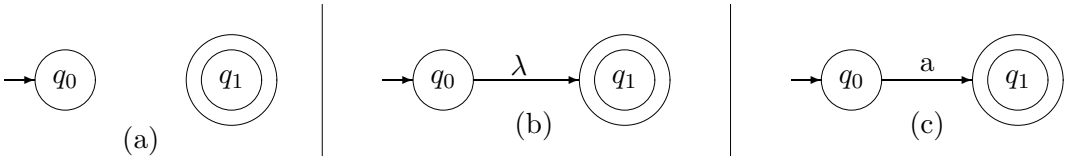
y por tanto $(0 + 1 (10^*1)^* 0)^* 1 (10^*1)^*$ es la expresión regular que describe el lenguaje $L(M)$.

7.2. Teorema de Síntesis de Kleene

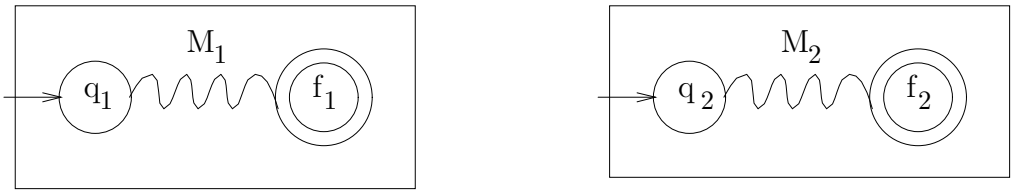
> Si L es un lenguaje asociado a una expresión regular α entonces existe un autómata finito M tal que $L = L(\alpha) = L(M)$.

Vamos a demostrar por inducción sobre el número de operadores de α $(+, \cdot, *)$ que existe un $AFND$-λ M con un sólo estado final sin transiciones y distinto del estado inicial, de forma que $L(\alpha) = L(M)$.

Base.- (cero operadores) α puede ser: $\varnothing$, λ, a, donde $a \in V$. Los autómatas que aceptan el lenguaje vacío, el lenguaje $\{\lambda\}$ y el lenguaje $\{a\}$, son, por este orden, los siguientes: (a), (b) y (c)



Inducción.- (uno o más operadores en α). Supongamos que se cumple la hipótesis para expresiones regulares de menos de n operadores. Sean las expresiones regulares α_1 y α_2 donde $op(\alpha_1)$, $op(\alpha_2) < n$. Entonces, por hipótesis existen dos autómatas finitos M_1 y M_2 tal que $L(M_1) = L(\alpha_1)$ y $L(M_2) = L(\alpha_2)$, donde $M_1 = (Q_1, V_1, \Delta_1, q_1, \{f_1\})$ y $M_2 = (Q_2, V_2, \Delta_2, q_2, \{f_2\})$ y podemos suponer sin pérdida de generalidad que $Q_1 \cap Q_2 = \varnothing$. Estos autómatas podemos representarlos esquemáticamente como:



Supongamos que tenemos una expresión regular α con n operadores. Vamos a construir un automata M tal que $L(M) = L(\alpha)$ y para eso distinguimos tres casos correspondientes a las tres formas posibles de expresar α en función de otras expresiones regulares con menos de n operadores.

1. $\boxed{\alpha = \alpha_1 + \alpha_2}$ tal que $op(\alpha_1)$, $op(\alpha_2) < n$. Los autómatas correspondientes a α_1 y α_2 son respectivamente M_1 y M_2, como hemos dicho antes. A partir de M_1 y M_2 construimos otro autómata $M = (Q_1 \cup Q_2 \cup \{q_0, f_0\}, V_1 \cup V_2, \Delta, q_0, \{f_0\})$ donde Δ se define como:

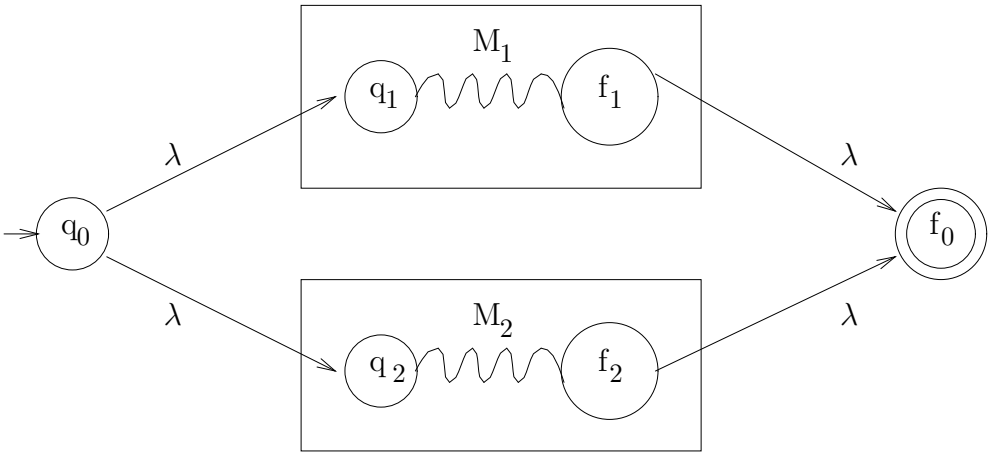
   a) $\Delta(q_0, \lambda) = \{q_1, q_2\}$

   b) $\Delta(q, \sigma) = \Delta_1(q, \sigma), \quad \forall\, q \in Q_1 - \{f_1\},\ \sigma \in V_1 \cup \{\lambda\}$

   c) $\Delta(q, \sigma) = \Delta_2(q, \sigma), \quad \forall\, q \in Q_2 - \{f_2\},\ \sigma \in V_2 \cup \{\lambda\}$

   d) $\Delta(f_1, \lambda) = \Delta(f_2, \lambda) = \{f_0\}$

   M se puede representar gráficamente del siguiente modo:



   Cualquier camino de q_0 a f_0 debe pasar forzosamente a través del autómata M_1 o del autómata M_2. Si una cadena w es aceptada por M, entonces debe ser aceptada también por M_1 o por M_2.

   Es decir, $L(M) = L(M_1) \cup L(M_2) =$

   $= L(\alpha_1) \cup L(\alpha_2)$, por hipótesis de inducción

   $= L(\alpha_1 + \alpha_2)$, por definición de lenguaje asociado a $\alpha_1 + \alpha_2$

   $= L(\alpha)$, como queríamos demostrar.

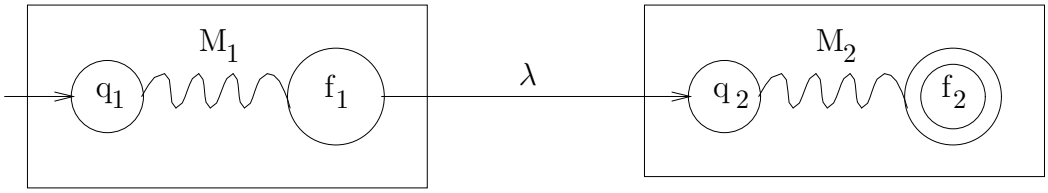
2. $\boxed{\alpha = \alpha_1 \cdot \alpha_2}$ tal que $op(\alpha_1)$, $op(\alpha_2) < n$. A partir de M_1 y M_2 construimos otro autómata $M = (Q_1 \cup Q_2, V_1 \cup V_2, \Delta, q_1, \{f_2\})$ donde Δ se define como:

   a) $\Delta(q, \sigma) = \Delta_1(q, \sigma), \quad \forall\, q \in Q_1 - \{f_1\},\ \sigma \in V_1 \cup \{\lambda\}$

   b) $\Delta(f_1, \lambda) = \{q_2\}$

   c) $\Delta(q, \sigma) = \Delta_2(q, \sigma), \quad \forall\, q \in Q_2,\ \sigma \in V_2 \cup \{\lambda\}$

M se puede representar esquemáticamente como:



Cualquier camino de q_1 a f_2 debe pasar forzosamente a través del autómata M_1 y del autómata M_2. Si una cadena w es aceptada por M, entonces esa cadena se puede descomponer como $w = w_1.w_2$, de forma que w_1 debe ser aceptada por M_1 y w_2 por M_2. Según esto, $L(M) = L(M_1) \cdot L(M_2) =$

$= L(\alpha_1) \cdot L(\alpha_2)$, por hipótesis de inducción

$= L(\alpha_1 \cdot \alpha_2)$, por definición de lenguaje asociado a $\alpha_1 \cdot \alpha_2$

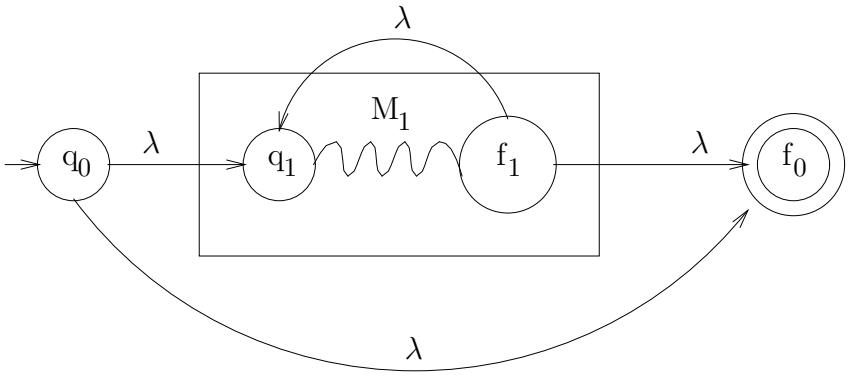
$= L(\alpha)$, como queríamos demostrar.

3. $\boxed{\alpha = (\alpha_1)^*}$ tal que $op(\alpha_1) = n - 1$. El autómata correspondiente a α_1 es M_1, a partir del cual construimos otro autómata $M = (Q_1 \cup \{q_0, f_0\}, V_1, \Delta, q_0, \{f_0\})$ donde Δ se define como:

   a) $\Delta(q_0, \lambda) = \Delta(f_1, \lambda) = \{q_1, f_0\}$

   b) $\Delta(q, \sigma) = \Delta_1(q, \sigma), \quad \forall\, q \in Q_1 - \{f_1\},\ \sigma \in V_1 \cup \{\lambda\}$

   M se puede representar del siguiente modo:



   Este autómata acepta cadenas de la forma $w = w_1 w_2 \cdots w_j$, donde $j \geq 0$ y cada subcadena w_i es aceptada por M_1. Por tanto,

   $L(M) = \bigcup_{n=0}^{\infty} (L(M_1))^n =$

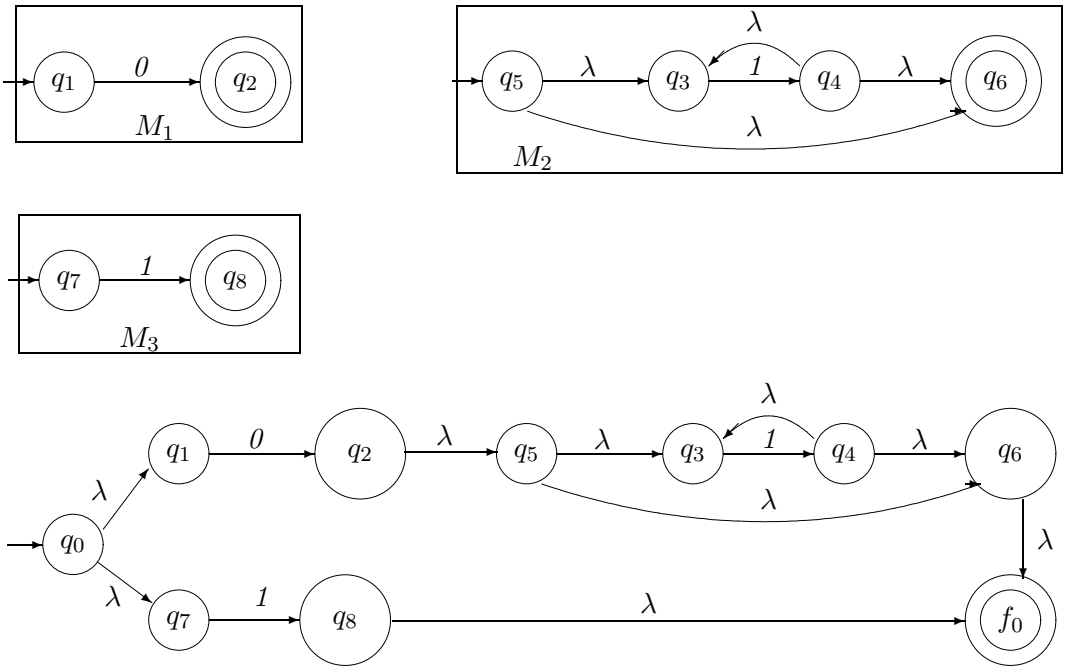
   $= \bigcup_{n=0}^{\infty} (L(\alpha_1))^n$, por hipótesis de inducción

   $= (L(\alpha_1))^*$, por definición de clausura de un lenguaje

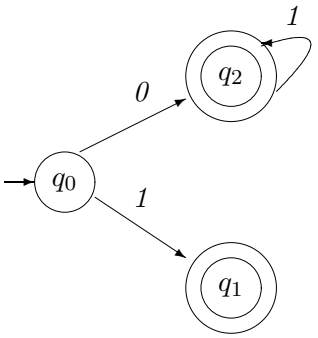
   $= L(\alpha_1^*)$, por definición de lenguaje asociado a α_1^*

   $= L(\alpha)$, como queríamos demostrar.

Ejemplo 3.9 *Siguiendo el método anterior, en la figura siguiente se ha construido un autómata para la expresión regular* $01^* + 1$*, donde* M_1 *representa el autómata para la expresión regular* 0*,* M_2 *representa* 1^* *y* M_3 *la expresión regular* 1*. En el autómata final se han integrado simultáneamente los autómatas para la concatenación (*0 *con* 1^**) y la suma de expresiones regulares* $01^* + 1$*.*



Este método de construcción de AFs para expresiones regulares está pensado para ser implementado de forma automática mediante un programa. Para nosotros podría haber sido más sencillo pensar directamente, por ejemplo, en el siguiente autómata:



7.3. Autómatas finitos y gramáticas regulares

Los autómatas finitos son mecanismos *reconocedores* de lenguajes y las gramáticas regulares son mecanismos *generadores* y vamos a ver que ambos tratan con la misma clases de lenguajes: los lenguajes regulares. Primero vamos a ver un teorema ($AF \longrightarrow GR$) cuya demostración nos proporciona un método para obtener una GR a partir de un AF y luego presentamos el teorema ($GR \longrightarrow AF$) para obtener un AF a partir de una GR.

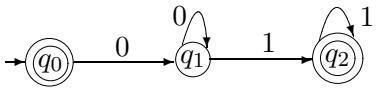
Teorema **3.3** **($AF \longrightarrow GR$)** *Si L es un lenguaje aceptado por una autómata finito M, entonces existe una gramática regular G tal que $L = L(M) = L(G)$.*

Dem.- Podemos suponer sin pérdida de generalidad que $M = (Q, V, \Delta, q_0, F)$ es un AF que no tiene λ-transiciones. Podemos obtener la gramática $G = (Q, V, q_0, P)$ a partir del diagrama de transición del AF con el siguiente **método**:

1.	Si tenemos el arco $q \xrightarrow{a} p$ entonces añadimos a P la regla $q \to ap$
2.	Si $q_F \in F$ añadimos la regla $q_F \to \lambda$

Esta es la *parte constructiva* de la demostración. Falta la *parte inductiva* para probar que el método es válido: hay que demostrar que $w \in L(G) \Leftrightarrow w \in L(M)$. Lo dejamos para la sección de aplicaciones. ■

Ejemplo 3.10 *Dado el siguiente AF, obtener la gramática correspondiente y la expresión regular.*



La gramática que se obtiene es:

$$G = (\{q_0, q_1, q_2\}, \{0, 1\}, q_0, \{q_0 \to 0q_1 \mid \lambda,\ q_1 \to 0q_1 \mid 1q_2,\ q_2 \to 1q_2 \mid \lambda\})$$

Para obtener la expresión regular podemos obtener las ecuaciones características del autómata, o bien, obtener el sistema de ecuaciones para la gramática. En ambos casos se obtienen las mismas ecuaciones, que pasamos a resolver:

$$\begin{array}{l} q_0 = 0q_1 + \lambda \\ q_1 = 0q_1 + 1q_2 \\ q_2 = 1q_2 + \lambda \end{array} \dashrightarrow \begin{array}{l} q_0 = 0q_1 + \lambda \\ q_1 = 0q_1 + 1 \cdot 1^* \\ q_2 = 1^* \end{array} \dashrightarrow \begin{array}{l} \boxed{q_0 = 00^*11^* + \lambda} \\ \\ q_1 = 0^*11^* \end{array}$$

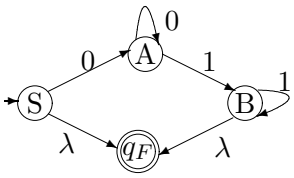
Teorema 3.4 ($GR \longrightarrow AF$) *Si L es un lenguaje generado por una gramática regular G, entonces existe un autómata finito M tal que $L = L(G) = L(M)$.*

Dem.- Podemos suponer sin pérdida de generalidad que $G = (V_N, V_T, S, P)$ es una gramática lineal derecha. Podemos obtener el diagrama del autómata finito $M = (V_N \cup \{q_F\}, V_T, \Delta, S, \{q_F\})$ a partir de la gramática con el siguiente **método**:

1.	Si la regla $A \to aB \in P$ entonces añadimos el arco $A \xrightarrow{a} B$
2.	Si la regla $A \to a \in P$ añadimos el arco $A \xrightarrow{a} q_F$
3.	Si la regla $A \to \lambda \in P$ añadimos el arco $A \xrightarrow{\lambda} q_F$

Esta es la *parte constructiva* de la demostración. Falta la *parte inductiva* para probar que el método es válido. Como en el caso del teorema anterior, hay que demostrar que $w \in L(G) \Leftrightarrow w \in L(M)$. Lo dejamos como ejercicio. ■

Ejemplo 3.11 *Dada la gramática con reglas: $P = \{S \to 0A \mid \lambda,\ A \to 0A \mid 1B,\ B \to 1B \mid \lambda\}$. El AF que reconoce el lenguaje generado por esta gramática es:*



8. Minimización de un AFD

En esta sección vamos a ver cómo podemos obtener un AFD equivalente a uno dado que tenga el menor número de estados posibles. Para ello es necesario definir antes una relación de equivalencia en el conjunto de estados de un AFD.

Definición 3.1 *Un estado q de un AFD es* ACCESIBLE *si* $\exists\, x \in V^*$ tal que $(q_0, x) \vdash^* (q, \lambda)$. *En otro caso el estado es* INACCESIBLE.

Definición 3.2 *Decimos que dos estados p y q de un AFD son* EQUIVALENTES *y lo notamos $p \approx q$, si cumplen:*

$$\boxed{\forall\, w \in V^* : \quad \text{si } \left[(q, w) \vdash^* (q', \lambda) \wedge (p, w) \vdash^* (p', \lambda)\right] \text{ entonces } \left[q' \in F \Leftrightarrow p' \in F\right]}$$

Por otro lado, decimos que p y q son DISTINGUIBLES *si no son equivalentes, o lo que es lo mismo, existe una cadena w tal que se produce una de las dos condiciones siguientes:*

- $q' \in F$ *y* $p' \notin F$, *o bien,*
- $q' \notin F$ *y* $p' \in F$

Nota A partir de la definición anterior podemos hacer las siguientes observaciones:

1. Si tenemos dos estados $q \in F$ y $p \notin F$ entonces podemos asegurar que son distinguibles ya que tomando $w = \lambda$ se cumple la definición.
2. Si sabemos que dos estados (p_a, q_a) son distinguibles y se tiene que $p_a = \delta(p, a)$, $q_a = \delta(q, a)$ entonces podemos asegurar que (p, q) son distinguibles, ya que si una cadena w hace distinguibles a (p_a, q_a) entonces la cadena aw hace distinguibles a (p, q).
3. $\approx$ define una RELACIÓN DE EQUIVALENCIA en el conjunto de estados del autómata y una forma de reducir el número de estados de un AFD será encontrar las clases de equivalencia en el conjunto de estados y a partir de ahí construir un autómata cuyos estados sean las clases de equivalencia.

Definición 3.3 *Dado un AFD $M = (Q, V, \delta, q_0, F)$ se define el* AUTÓMATA COCIENTE *de M como $M_{\approx} = (Q', V, \delta', q_0', F')$ donde:*

$$\boxed{\begin{array}{l} Q' = Q \,/\, \approx \\ \delta\prime\,([q]\,, a) = [\delta(q, a)] \\ q_0' = [q_0] \\ F' = F \,/\, \approx \end{array}}$$

Teorema **3.5** *Dado un autómata finito determinista M, el autómata cociente $M_{\approx}$ es el autómata mínimo equivalente a M. Este autómata mínimo es único, salvo isomorfismos (renombramiento de estados).*

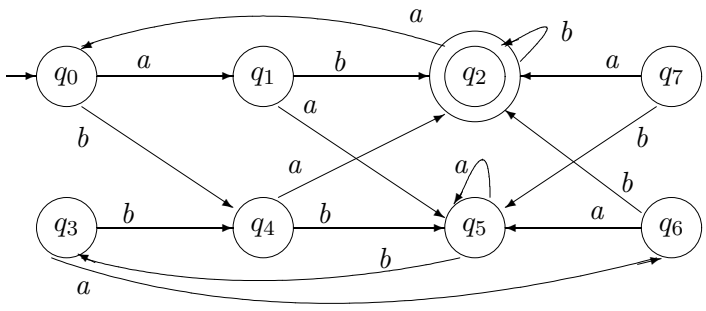
Dem.- Primero tenemos que probar que M y $M_{\approx}$ son equivalentes. Pero para eso basta probar que $\forall\, w \in V^* :\ w \in L(M) \Leftrightarrow w \in L(M_{\approx})$. Para $w = \lambda$ está claro que $\lambda \in L(M) \Leftrightarrow q_0 \in F \Leftrightarrow [q_0] \in F' \Leftrightarrow \lambda \in L(M_{\approx})$. Cuando tenemos una palabra de longitud mayor que cero, por ejemplo $w = a_1 a_2 \dots a_n$ entonces $w \in L(M) \Leftrightarrow$ existe un cálculo donde $q_{i_n} \in F$ y:

$$(q_0, a_1 a_2 \dots a_n) \vdash_M (q_{i_1}, a_2 \dots a_n) \vdash_M \dots \vdash_M (q_{i_{n-1}}, a_n) \vdash_M (q_{i_n}, \lambda)$$

$\Leftrightarrow q_{i_j} = \delta(q_{i_{j-1}}, a_j), \quad \forall\, 1 \le j \le n$ (por definición de la relación $\vdash$)
$\Leftrightarrow [q_{i_j}] = [\delta(q_{i_{j-1}}, a_j)], \quad \forall\, 1 \le j \le n$ (por definición de clase de equivalencia)
$\Leftrightarrow [q_{i_j}] = \delta'([q_{i_{j-1}}], a_j), \quad \forall\, 1 \le j \le n$ (por definición de δ')
$\Leftrightarrow ([q_0], a_1 a_2 \dots a_n) \vdash_{M_{\approx}} ([q_{i_1}], a_2 \dots a_n) \vdash_{M_{\approx}} \dots \vdash_{M_{\approx}} ([q_{i_{n-1}}], a_n) \vdash_{M_{\approx}} ([q_{i_n}], \lambda)$ donde $[q_{i_n}] \in F'$
$\Leftrightarrow w \in L(M_{\approx})$, como queríamos demostrar.
Ahora tenemos que probar no hay otro autómata equivalente a $M_{\approx}$ con menos estados que él. Supongamos que M' es equivalente a M y tiene menos estados que $M_{\approx}$. Entonces M' tiene que ser equivalente a $M_{\approx}$, lo cual implica que deben existir al menos dos estados distintos $[p]$ y $[q]$ en $Q \,/\, \approx$ que son equivalentes pero esto implica que $[p] = [q]$, lo cual es absurdo. Luego $M_{\approx}$ es único salvo isomorfismo (renombramiento de estados). ■

Una vez demostrada la existencia y unicidad del autómata mínimo mostramos en la figura 3.8 un **algoritmo que calcula el autómata cociente** o autómata mínimo. Aunque no vamos a demostrar formalmente la validez de este algoritmo, sería sencillo hacerlo a partir de la definición que hemos dado de la relación de equivalencia de estados, del autómata cociente y de los resultados expuestos en la nota anterior y el teorema 3.5. Aclaramos que cuando hablamos de PAR (q_i, q_j) nos referimos a la posición de la tabla $T[i, j]$ cuando $i > j$ o a $T[j, i]$ cuando $i < j$. El hecho de utilizar una **tabla triangular** se justifica porque si $q_i \approx q_j$ entonces $q_j \approx q_i$, ya que la relación de equivalencia $\approx$ es simétrica. De esta forma se ahorra espacio de memoria. El **marcado recursivo** de $lista(q_i, q_j)$ significa que accedemos a todas posiciones correspondientes a las parejas de estados en $lista(q_i, q_j)$; marcamos estas celdas de la tabla y las celdas correspondientes a los pares de las listas asociadas a estas posiciones y así sucesivamente hasta que no se puedan marcar más.

Ejemplo 3.12 *Vamos a calcular el autómata mínimo correspondiente al siguiente autómata:*

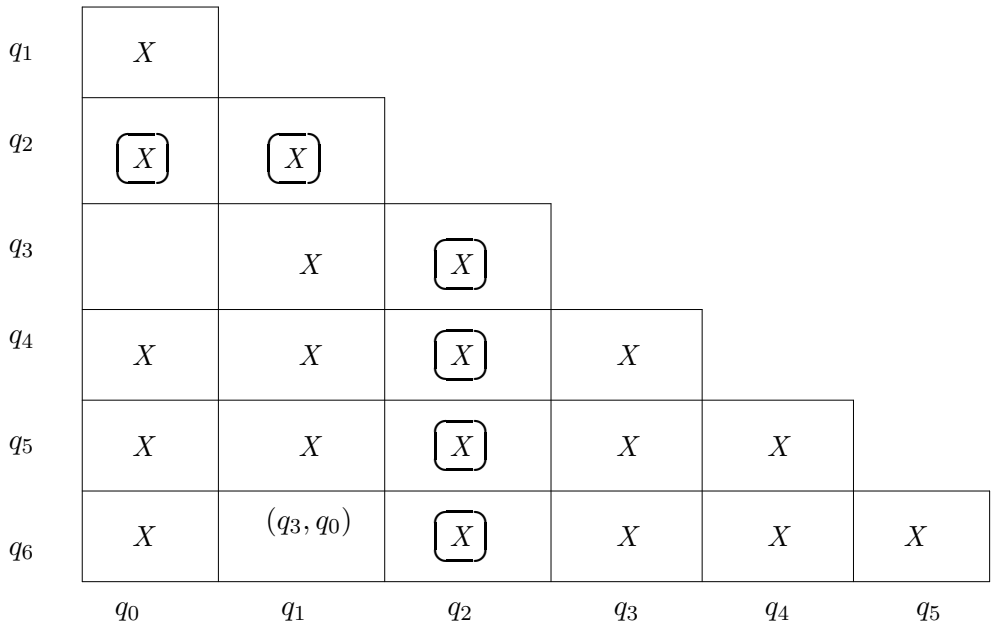


Claramente se ve que el estado q_7 es inaccesible, por tanto, se puede eliminar este estado y sus transiciones. Tenemos que construir ahora una tabla triangular con filas desde q_1 hasta q_6 y columnas desde q_0 hasta q_5. Marcamos la tabla que finalmente queda como sigue:

Entrada: Un AFD $M = (Q, V, \delta, q_0, F)$ con $Q = \{q_0, \ldots, q_n\}$, $V = \{a_1, \ldots, a_m\}$
Salida: AFD mínimo $M_{\approx} = (Q', V, \delta', q_0', F')$

1. Eliminar estados inaccesibles de M;
2. Construir tabla T con filas desde q_1 hasta q_n y columnas desde q_0 hasta q_{n-1};
3. Asociar a PAR (q_i, q_j) una lista de parejas de estados LISTA (q_i, q_j);
4. MARCAR (PAR (q_i, q_j)) donde un estado del par es final y el otro no;
5. for $i = 1$ to n
6. for $j = 0$ to $i - 1$
7. if PAR (q_i, q_j) no marcado
8. for $k = 1$ hasta m
9. $q_r \leftarrow \delta(q_i, a_k)$; $q_s \leftarrow \delta(q_j, a_k)$;
10. if PAR (q_r, q_s) marcado
11. MARCAR (PAR (q_i, q_j))
MARCAR-RECURSIVAMENTE (LISTA (q_i, q_j));
break;
12. else añadir a LISTA (q_r, q_s) el PAR (q_i, q_j);
13. end-if;
14. if PAR (q_i, q_j) no marcado entonces $q_i \approx q_j$;
15. calcular Q', q_0', δ', F' según la definición de autómata cociente;

Figura 3.8: Algoritmo de Minimización de un AFD

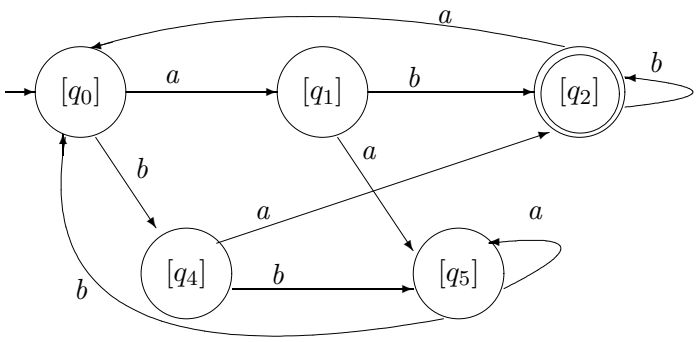


Las X recuadradas corresponden a las posiciones de la tabla que se marcan inicialmente (línea 4 del algoritmo). Al final quedan sin marcar PAR (q_3, q_0) *y* PAR (q_6, q_1) *y por tanto,* $q_0 \approx q_3$ *y*

$q_1 \approx q_6$. *Y el autómata cociente (mínimo) es:*

$$M_{\approx} = \left(\{[q_0], [q_1], [q_2], [q_4], [q_5]\}, \{0,1\}, \delta', [q_0], \{[q_2]\}\right)$$

cuyo diagrama de transición es:



El proceso de minimización de un AFD tiene gran **importancia práctica** ya que muchas aplicaciones de reconocimiento de patrones y de control se basan en la implementación de un AFD. Cuanto menos estados tenga el AFD más eficiente será la implementación.

9. Aplicaciones: análisis léxico

Una de las aplicaciones más importantes de los AFs es la construcción de analizadores léxicos. Como vimos en el tema 1, dentro del contexto general de un compilador, un *analizador léxico* (AL) tiene como principal función generar una lista ordenada de *tokens* a partir de los caracteres de entrada. Esos *tokens*, o componentes léxicos, serán usados por el analizador sintáctico para construir el correspondiente árbol sintáctico. Por otro lado, guarda información sobre algunos tokens, necesaria en el proceso de análisis y síntesis, en forma de atributos de esos componentes léxicos. Así pues, se debería considerar al analizador léxico como un módulo subordinado al analizador sintáctico (AS), tal y como se indica en el esquema de interacción entre ambos, que aparece en la figura 3.9.

El analizador léxico lee caracteres de entrada hasta que detecta que con el último carácter leído se puede formar un token, o un error en su caso, y comunica el evento correspondiente al analizador sintáctico. Si no hubo error, el AS procesa el token, y el AL no vuelve a entrar en juego hasta que el analizador sintáctico vuelva a necesitar otro *token* del flujo de entrada.

Otra de las funciones principales del AL es la detección y reparación de *errores léxicos*, aunque en este nivel la tarea de recuperación de errores no es muy sofisticada. Un error se produce cuando el AL detecta cualquier símbolo que es incapaz de reconocer y/o clasificar. Por ejemplo, la mayoría de versiones de PASCAL requieren que la expresión de un número en punto flotante comience con un 0: el *token* 0,5 pertenecería al lenguaje y ,5 no. Otro error que el analizador léxico podría detectar es el de exceder el número de caracteres máximo para un identificador.

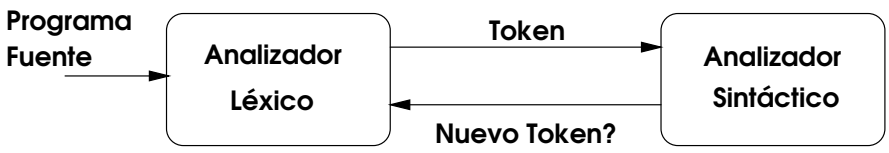


Figura 3.9: Esquema de interacción AL-AS

9.1. Especificación y reconocimiento de componentes léxicos

Los tokens se pueden *describir* mediante expresiones regulares y se pueden *reconocer* mediante autómatas finitos. Una expresión regular para un token, describe todos los *lexemas* que dicho token puede tener asociados. Los autómatas finitos se construyen a partir de las expresiones regulares que describen los tokens.
Para un buen reconocimiento léxico, los posibles tipos de *tokens* se deben diseñar con mucho cuidado. En general, un conjunto de cadenas a la entrada pueden corresponder al mismo componente léxico de salida. Cada conjunto de cadenas se va a definir mediante un patrón, asociado a un token determinado. En la siguiente tabla se dan ejemplos de varios tipos comunes de componentes léxicos, junto con lexemas ejemplo, y patrones que definen esos tokens.

Token	Lexemas ejemplo	Patrón no formal
const	const	const
if	if	if
relación	<,≤,=,<>,≥	< ó ≤ ó = ó <> ó ≥
identificador	pi, cuenta, D2	letra seguida de letras y dígitos
número	3.1416, 0, 6.02E23	cualquier cte. numérica
literal	"vaciado de memoria"	cualquier carácter entre " y " excepto "

Los componentes léxicos serán símbolos terminales de la gramática, y en la mayoría de lenguajes de programación se van a considerar como componentes léxicos las siguientes construcciones:

- **Palabras clave**: son cadenas que forman parte del lenguaje de programación en cuestión.
- **Operadores**.
- **Identificadores**.
- **Constantes** (reales, enteras y de tipo carácter).
- **Cadenas de caracteres**.
- **Signos de puntuación**.

Por simplicidad debe procurarse que los tokens sean sencillos y que los lexemas sean independientes. Aún así, podemos encontrarnos con *problemas a la hora de reconocer tokens*. A continuación analizamos algunos de ellos:

- A veces necesitamos leer uno o más caracteres extra de la entrada, denominados *caracteres de anticipación*, para decidir el código de token reconocido. Por ejemplo, si el lenguaje de programación admite los operadores "<" y "<=", es necesario, una vez que se lee de la entrada el símbolo "<", comprobar que si el siguiente es o no el símbolo "=".
- Las *palabras clave* pueden estar reservadas o no. Si son *reservadas*, su significado está predefinido y el usuario no puede modificarlo usándolas como identificadores, por ejemplo. En este caso, el analizador léxico debe reconocerlas directamente (a través del autómata finito) o bien usando una tabla de palabras reservadas. Si las palabras clave *no* están *reservadas*, entonces el analizador léxico las reconoce como identificadores , y la tarea de distinguirlas de éstos queda relegada al analizador sintáctico. Por ejemplo, en `PL/1` las palabras clave no son reservadas, y por lo tanto una sentencia de este tipo tiene sentido:

```
IF THEN THEN THEN = ELSE; ELSE ELSE = THEN;
```

- Los *comentarios* deben ser reconocidos y eliminados.

- Los *blancos* pueden actuar como delimitadores, en cuyo caso el AL debe eliminarlos sin más, o pueden no tener este papel. En este caso, además de eliminarlos, el AL debe agrupar lexemas. Por otro lado, en algunos lenguajes con formato de línea, como FORTRAN, se exige que ciertas construcciones aparezcan en posiciones fijas de la línea de entrada (por ejemplo, que se comience en la columna 7). Así, la alineación de un lexema puede ser importante para determinar si es correcto o no. En este caso, la definición de un token cuyo lexema está formado por seis blancos, podría facilitar esta labor. Hoy en día, se tiende a diseñar lenguajes de programación independientes del formato.
- La definición del token *EOF* (End Of File) puede facilitar el análisis sintáctico posterior, pues permitiría comprobar si después del final del programa aparecen más símbolos, o bien si el fichero termina antes de que termine la escritura de un programa completo.

9.2. Diseño de un analizador léxico

El proceso que se sigue para la implementación del analizador léxico puede resumirse en los siguientes **pasos**:

- Identificar los *tokens* del lenguaje, y definirlos utilizando ERs como herramientas expresivas de descripción.
- Obtener el AF correspondiente a las ERs que ha de reconocer el AL. Minimizar el número de estados del AF.
- Se ha de programar el autómata, simulando su ejecución con la técnica que se considere oportuna.
- Se ha de diseñar el interface entre la entrada estándar, de donde proviene el programa fuente, y el AL.
- Una vez que se ha conseguido simular el autómata, los tokens reconocidos van a ser utilizados por el analizador sintáctico. Por lo tanto se hace necesario el diseño de una interface adecuada, entre el AS y el autómata que simula el reconocedor. Normalmente se incluye el AL como una subrutina del AS, devolviéndole un token cada vez que el AS lo requiera, así como la información asociada a él. Con respecto a esto, es importante definir el TAD (Tipo Abstracto de Datos) que servirá como soporte a la *tabla de símbolos* necesaria en este proceso de interacción.
- Especificar qué tipo de manejo de errores va a seguir el AL.

Ejemplo 3.13 *Vamos a ver un ejemplo sencillo. El token que podemos llamar* REAL *representa las constantes reales en Fortran y se puede describir mediante la siguiente expresión regular*

$$\text{REAL} = D^+(\lambda + .) + (D^*.D^+)$$

donde D a su vez describe la expresión regular $D = (0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9)$
Para simplificar, vamos a suponer que tenemos un código fuente donde deben aparecer constantes reales separadas por blancos (denotado por B). El $AFND$-λ que reconoce una constante real según la expresión regular anterior e ignora los blancos que pueda haber antes, es el de la figura 3.10. La nueva función de transición Δ' del $AFND$ equivalente es:

$$\begin{array}{ll}
\Delta'(q_0, 0-9) = \{q_1, q_4\} & \\
\Delta'(q_0, \cdot) = \{q_2\} & \Delta'(q_2, 0-9) = \{q_3\} \\
\Delta'(q_0, b) = \{q_0\} & \Delta'(q_3, 0-9) = \{q_3\} \\
\Delta'(q_1, \cdot) = \{q_2\} & \Delta'(q_4, 0-9) = \{q_4\} \\
\Delta'(q_1, 0-9) = \{q_1\} & \Delta'(q_4, \cdot) = \{q_5\}
\end{array}$$

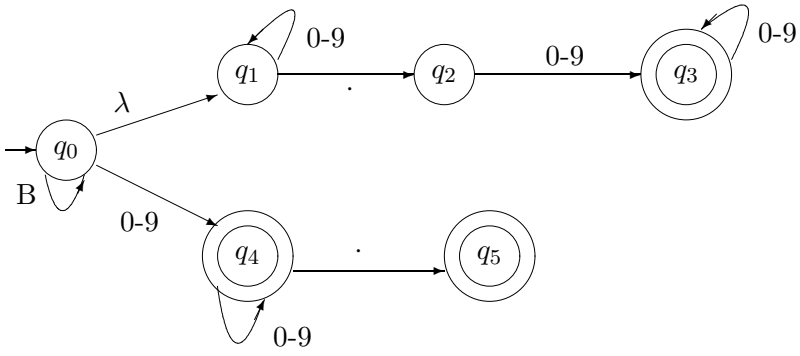


Figura 3.10: $AFND$-λ para REAL

Finalmente se obtiene un AFD cuya función de transición δ se refleja en el diagrama de transición de la figura 3.11

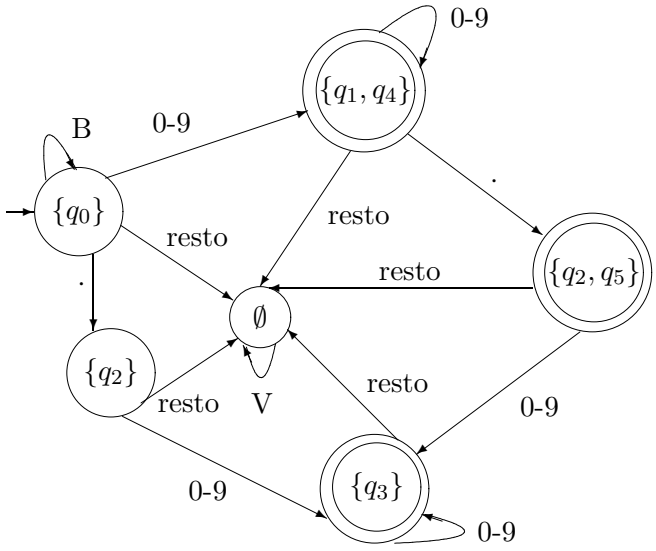


Figura 3.11: AFD para REAL

El AFD no es mínimo. Se puede comprobar que los estados $\{q_3\}$ y $\{q_2, q_5\}$ son equivalentes. Si minimizamos el AFD y renombramos los estados que quedan, tenemos el diagrama de transición del AFD mínimo en la figura 3.12.
*Ahora **simulamos** el AFD mínimo a partir de su **diagrama de transición**, implementando en lenguaje C una función* REAL1 *que al ser llamada nos dice si se ha leído una constante real correcta o no. Se supone que lee los caracteres de la entrada estándar.*

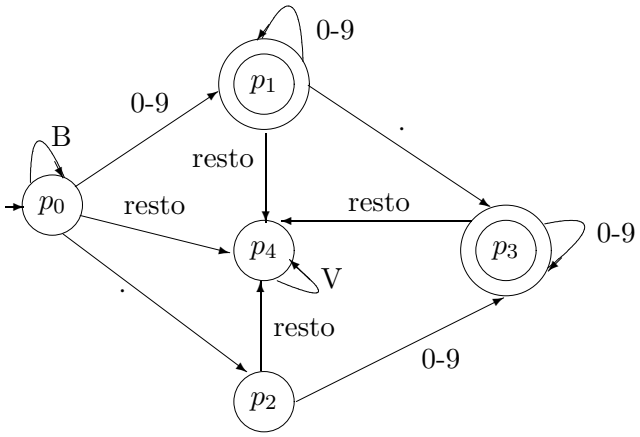


Figura 3.12: AFD mínimo para REAL

```
    REAL1() {
    int car;
1.  while (isspace(car=getchar())); /* se mantiene en p0 */
2.  ungetchar(car);
3.  if (isdigit(car=getchar())) { /* pasa a p1 */
4.    for (car=getchar(); isdigit(car); car=getchar());
5.    if (car=='.') { /* pasa a p3 */
6.      for (car=getchar(); isdigit(car); car=getchar());
7.      comprueba_fin(); } /* puede acabar en p3 */
8.    else comprueba_fin();}
9.  else if (car=='.') /* pasa a p2 */
10.   if (isdigit(car=getchar())) { /* pasa a p3 */
11.     for (car=getchar(); isdigit(car); car=getchar());
12.     comprueba_fin(); }
13.   else puts(''constante real incorrecta'');
14. else puts(''constante real incorrecta'');}

    COMPRUEBA_FIN()
15. if (car=='\ n') /* se ha leido toda la cte sin error*/
16.   puts(''constante real correcta'');
17. else puts(''constante real incorrecta'');}
```

Otra forma de simular el AFD es a partir de su ***tabla de transición****. A continuación mostramos la tabla y en lugar de poner una columna para cada símbolo del alfabeto, utilizamos una de las columnas para todos los dígitos y otra para los demás caracteres que no sean un dígito, el punto decimal o un carácter blanco.*

δ	b	$0-9$	$\cdot$	$V-\{0-9,\cdot,B\}$
p_0	p_0	p_1	p_2	p_4
p_1	p_4	p_1	p_3	p_4
p_2	p_4	p_3	p_4	p_4
p_3	p_4	p_3	p_4	p_4
p_4	p_4	p_4	p_4	p_4

A continuación mostramos el código para simular el AFD a partir de esta tabla, que se corresponde con la función REAL2*.*

```
    #define ERROR 4
    REAL2() {
    int estado, car, colum;
1.  int tabla[5][4]={0,1,2,4,4,1,3,4,4,3,4,4,4,3,4,4,4,4,4,4};
    /* asignación por filas y usamos
    colum 0 ⟶ para carácter blanco
    colum 1 ⟶ para dígitos
    colum 2 ⟶ para el punto decimal
    colum 3 ⟶ para el resto de caracteres */
2.  estado=0;
3.  While ((car=getchar())!=\ n') {
4.    if (isspace(car)) colum=0;
5.    else if (isdigit(car)) colum=1;
6.      else if (car == '.') colum=2 ; else colum=3;
7.    estado=tabla[estado][colum];}
8.  if ((estado==1) || (estado==3))
9.    puts(''constante real correcta'');
10. else puts(''constante real incorrecta'');}
```

9.3. Manejo de errores léxicos

No son muchos los errores que puede detectar un scanner; sin embargo, puede darse la situación de encontrar una cadena que no concuerde con ninguno de los patrones correspondientes a los *tokens* del lenguaje. ¿Qué hacer entonces?
El AL detectará un error cuando no exista transición válida en el autómata finito para el carácter de entrada. En este caso, el AL debe de informar del error, pero además, sería deseable que intentara recuperarse para seguir buscando otros errores. Aunque no existen métodos generales que funcionen bien en todos los casos, algunos de ellos se comportan de forma aceptable y son bastante eficientes. Por ejemplo:

- Recuperación en *modo pánico*: este tipo de estrategia es la más común. Consiste en ignorar caracteres extraños hasta encontrar un carácter válido para un nuevo token.
- Borrar un carácter extraño.
- Insertar un carácter que falta (e.g. reemplazar `2C` por `2*C`).
- Reemplazar un carácter incorrecto por otro correcto (e.g. reemplazar `INTAGER` por `INTEGER` si el lugar en donde aparece el primer lexema no es el indicado para un identificador).
- Intercambiar dos caracteres adyacentes.

Se suele utilizar el *modo pánico*, pues las otras técnicas, aunque más sofisticadas, también son más costosas de implementar. La recuperación de errores durante el AL puede producir otros en las siguientes fases. Por ejemplo, con el siguiente programa

```
var numero : integer;
begin
    num?ero:=10;
end
```

el compilador podría producir los siguientes mensajes de error:

- ERROR LÉXICO: carácter no reconocido (?)

- ERROR SEMÁNTICO: identificador no declarado (num)
- ERROR SINTÁCTICO: falta operador entre identificadores
- ERROR SEMÁNTICO: identificador no declarado (ero)

En otras ocasiones, sin embargo, la recuperación en modo pánico no conlleva efectos en otros ámbitos, como sería el caso del siguiente programa, en el se generaría un sólo aviso de error léxico (semejante al primero del ejemplo anterior):

```
var i,j: integer;
begin
    i:=1;
    ?
    j:=2;
end
```

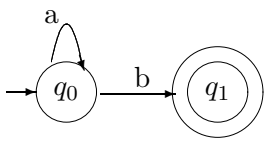
Finalmente, hay que tener en cuenta que existen errores que no son recuperables, como ocurriría por ejemplo, si se nos olvidara cerrar un comentario.

EJERCICIOS RESUELTOS

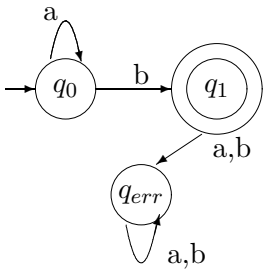
---

1. Dado un $AFND$ que cumple $|\Delta(q,a)| \leq 1$ para todo estado y todo símbolo del alfabeto ¿Cómo se puede pasar de forma sencilla a un AFD equivalente?

Este autómata es "casi determinista", lo único que puede ocurrir es que la función de transición no esté definida para alguna pareja (estado, símbolo). En este caso lo que hacemos es incluir un estado nuevo que actúa como *estado de error*, de forma que a él van a parar las transiciones no definidas y todos los arcos que salen del estado de error vuelven a él. De esta forma, si en un cálculo llegamos a una configuración con el estado de error, podemos estar seguros de que la palabra no será aceptada. Como ejemplo, podemos ver los siguientes diagramas de transición:



$AFND$



AFD equivalente

La palabra $abab$ no es aceptada por el $AFND$ ya que el único cálculo posible es el siguiente:

$$(q_0, abab) \vdash (q_0, bab) \vdash (q_1, ab) \vdash \text{ (no puede seguir)}$$

Con el AFD tenemos el siguiente cálculo, donde se procesa toda la palabra, pero no acaba en configuración de aceptación:

$$(q_0, abab) \vdash (q_0, bab) \vdash (q_1, ab) \vdash (q_{err}, b) \vdash (q_{err}, \lambda)$$

2. ¿Cual es la expresión regular que describe el lenguaje aceptado por los autómatas anteriores?

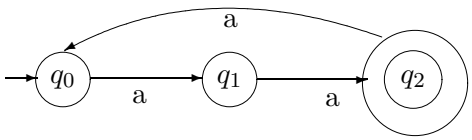
Parece sencillo ver a simple vista que $L(M) = \{a^n b \mid n \geq 0\}$, o lo que es lo mismo, $L(M) = L(a^*b)$. Para estar seguros podemos aplicar el método de las ecuaciones características. Se pueden obtener las ecuaciones para el $AFND$ o el AFD indistintamente. Mejor obtener las correspondientes al $AFND$ que tiene menos estados:

$$\begin{array}{l} q_0 = aq_0 + bq_1 \\ q_1 = \lambda \end{array} \quad \dashrightarrow \quad q_0 = aq_0 + b \quad \dashrightarrow \quad \boxed{q_0 = a^*b}$$

Si obtenemos las ecuaciones para el AFD el resultado es el mismo:

$$\begin{array}{l} q_0 = aq_0 + bq_1 \\ q_1 = (a+b)q_{err} + \lambda \\ q_{err} = (a+b)q_{err} \end{array} \dashrightarrow \begin{array}{l} q_0 = aq_0 + bq_1 \\ q_1 = (a+b)\cdot\varnothing + \lambda = \lambda \\ q_{err} = (a+b)^*\cdot\varnothing = \varnothing \end{array} \dashrightarrow \quad q_0 = aq_0 + b \quad \dashrightarrow \quad \boxed{q_0 = a^*b}$$

3. Demostrar formalmente que $L(M) = \{(aaa)^n aa \mid n \geq 0\}$, donde M es el siguiente autómata finito:



Si probamos que $(q_i, a^m) \vdash^* (q_j, \lambda)$, donde $j = (m+i) \bmod 3$, entonces la palabra a^m será aceptada si $(i = 0)$ y $(j = 2)$, siendo $2 = m \bmod 3$. Luego debe ser $m = 3n+2$, $n \geq 0$, y en este caso $a^m = (aaa)^n aa$, que es lo que se indica en la descripción de $L(M)$ en el enunciado. Por tanto tenemos que probar la siguiente hipótesis:

$$\boxed{HIP. : \forall\, 0 \leq i \leq 2 \text{ se cumple } (q_i, a^m) \vdash^* (q_j, \lambda)\,, \text{ donde } j = (m+i) \bmod 3}$$

Dem.- Lo demostramos por inducción sobre m.

Base.- $(m = 0)$ Entonces $j = i \bmod 3 = i$ y por ser la relación de cálculo $\vdash^*$ reflexiva se tiene trivialmente que $(q_i, \lambda) \vdash^* (q_i, \lambda)$.

Inducción.- Supongamos que se cumple la hipótesis $\forall\, m \leq k$ y vamos a demostrar que se cumple también para $m = k+1$. Como $m > 0$ podemos afirmar que:

$$(q_i, a^m) \vdash \left(q_{i'}, a^{m-1}\right) \vdash^* (q_j, \lambda)$$

para algún i', j. Por definición de la relación de cálculo y según el diagrama de transición, tenemos que $i' = (i+1) \bmod 3$. Por hipótesis de inducción podemos decir que $j = (m - 1 + i') \bmod 3$. Sustituyendo el valor de i' tenemos:

$$j = (m - 1 + (i+1) \bmod 3) \bmod 3$$

y por propiedades del módulo se tiene $j = (m+i) \bmod 3$, como queríamos demostrar. Por tanto el lenguaje acepta las palabras de la forma a^m donde $m = 3n+2$, $\forall\, n \geq 0$, condición que podemos expresar como $L(M) = \{(aaa)^n aa \mid n \geq 0\}$. ■

El ejemplo anterior nos muestra como un AF se puede usar para "contar" símbolos o

verificar si se cumplen algunas propiedades aritméticas, como por ejemplo, un lenguaje formado por palabras que tienen un número par de $a's$. Pero los AFs son bastante limitados en este sentido. Ya veremos en el tema siguiente que no se pueden reconocer con AFs lenguajes más complicados como $L = \{a^n b^n \mid n \geq 0\}$, o el lenguaje $L = \{a^n \mid n \text{ es un cuadrado perfecto}\}$.

---

4. Probar la hipótesis del teorema 3.1 que muestra la validez del método para pasar de un $AFND$ M_N a un AFD M_D equivalente. La hipótesis es:

$$HIP.: \underset{(1)}{(S,w) \vdash^*_{M_D} (S',\lambda)} \quad \Leftrightarrow \quad \underset{(2)}{S' = \left\{p \in Q \mid (q,w) \vdash^*_{M_N} (p,\lambda) \wedge q \in S\right\}}$$

Dem.- Vamos a demostrarlo por inducción sobre $|w|$.

Base.- ($|w| = 0$). Entonces $w = \lambda$. Por (1) tenemos que $(S,\lambda) \vdash^*_{M_D} (S',\lambda)$ sii (por definición de $\vdash^*_{M_D}$) $S = S'$. Pero esto es así ya que:

$$S' = \left\{p \in Q \mid (q,\lambda) \vdash^*_{M_N} (p,\lambda) \wedge q \in S\right\} = S$$

puesto que para que se de $(q,\lambda) \vdash^*_{M_N} (p,\lambda)$ debe ser $p = q$. Queda demostrado pues que (1) sii (2) para $|w| = 0$.

Inducción.- Supongamos que se cumple la hipótesis para palabras de longitud $< n$ y vamos a demostrar que se cumple también para $|w| = n$. Sea $w = az$ con $a \in V$ y $|z| = n - 1$. Por (1) y por definición de la relación de cálculo tenemos que:

$$\underset{(i)}{(S,az) \vdash_{M_D} (S_1,z)} \quad \vdash^*_{M_D} \quad \underset{(ii)}{(S',\lambda)}$$

$$\Leftrightarrow \left\{ \begin{array}{ll} (i) & S_1 = \delta(S,a) = \{r \in Q \mid r \in \Delta(q,a) \wedge q \in S\} \text{ por definición de } \delta \\ (ii) & S' = \left\{p \in Q \mid (r,z) \vdash^*_{M_N} (p,\lambda) \wedge r \in S_1\right\} \text{ por hipótesis} \end{array} \right\}$$

$\Leftrightarrow$ (por pertenecer r a S_1) $S' = \left\{p \in Q \mid (r,z) \vdash^*_{M_N} (p,\lambda) \wedge r \in \Delta(q,a) \wedge q \in S\right\} \Leftrightarrow$

$$S' = \left\{p \in Q \mid (r,z) \vdash^*_{M_N} (p,\lambda) \wedge (q,az) \vdash_{M_N} (r,z) \wedge q \in S\right\}$$

$\Leftrightarrow$ (por ser $\vdash^*_{M_N}$ la clausura reflexiva y transitiva de $\vdash_{M_N}$):

$$S' = \left\{p \in Q \mid (q,w) \vdash^*_{M_N} (p,\lambda) \wedge q \in S\right\} \equiv (2),\ c.q.d$$

Probada ya la hipótesis y en particular para $S = \{q_0\}$ podemos decir que se cumple:

$$(\{q_0\},w) \vdash^*_{M_D} (S',\lambda) \Leftrightarrow S' = \left\{p \in Q \mid (q_0,w) \vdash^*_{M_N} (p,\lambda)\right\}$$

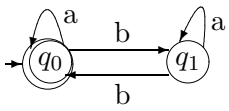
Luego tenemos que $w \in L(M_D) \Leftrightarrow S' \in F' \Leftrightarrow S' \cap F \neq \varnothing \Leftrightarrow \exists\, p_F \in F \wedge p_F \in S' \Leftrightarrow (q_0,w) \vdash^*_{M_N} (p_F,\lambda) \Leftrightarrow w \in L(M_N)$. Así que los dos autómatas aceptan los mismos lenguajes, luego el método es válido. ■

5. Escribir la tabla de transición y el diagrama de transición de un AFD tal que

$$L(M) = \{w \in \{a,b\}^* \mid b's(w) \text{ es par } \}$$

y encontrar un cálculo que acepte la palabra $aabba$

Podemos diseñar el AFD directamente, cuyas transiciones vienen dadas por:



	a	b
$\#q_0$	q_0	q_1
q_1	q_1	q_0

El cálculo para $aabba$ es:

$$(q_0, aabba) \vdash (q_0, abba) \vdash (q_0, bba) \vdash (q_1, ba) \vdash (q_0, a) \vdash (q_0, \lambda)$$

Luego $aabba \in L(M)$.

---

6. Dado el AFD de la figura 3.13 se puede comprobar que los estados q_0 y q_1 son equivalentes y lo mismo pasa con los estados q_2, q_3, q_4. Por tanto el autómata mínimo es el que se refleja

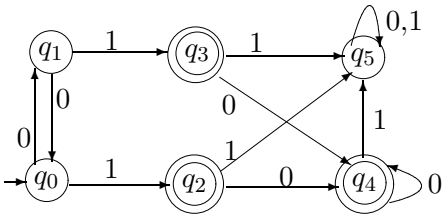
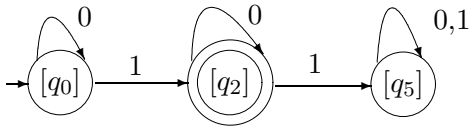


Figura 3.13: AFD que acepta $L(0^*10^*)$

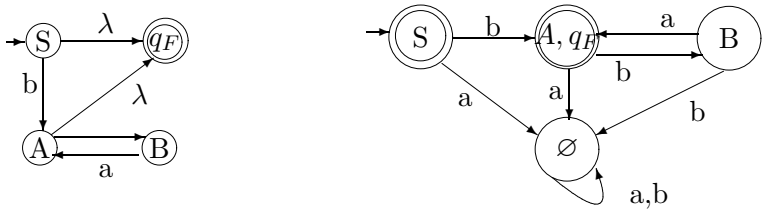
en el siguiente diagrama:



---

7. Dada la gramática con reglas de producción $P = \{S \to bA \mid \lambda,\ A \to bB \mid \lambda,\ B \to aA\}$. Obtener un AFD que acepte el lenguaje generado por la gramática y una expresión regular que describa el lenguaje.

Este ejercicio podemos resolverlo de varias formas. Una de ellas sería obtener la expresión regular a partir de la gramática (según el método del tema 2) y luego obtener el autómata a partir de la expresión regular, según el teorema de síntesis de Kleene. En este caso es más sencillo obtener el autómata a partir de la gramática, según el método que hemos mostrado en este tema. A continuación mostramos los diagramas del $AFND$-λ que se obtiene a partir de la gramática y el AFD equivalente:

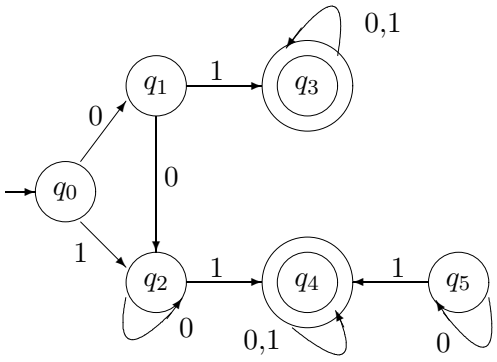


Ahora podemos obtener el sistema de ecuaciones de expresiones regulares a partir de la gramática o del autómata. Lo hacemos a partir de la gramática, ya que se obtienen menos ecuaciones:

$$\begin{array}{l} S = bA + \lambda \\ A = bB + \lambda \\ B = aA \end{array} \dashrightarrow \begin{array}{l} S = bA + \lambda \\ A = baA + \lambda \end{array} \dashrightarrow \begin{array}{l} S = bA + \lambda \\ A = (ba)^* \cdot \lambda \end{array} \dashrightarrow \boxed{S = b(ba)^* + \lambda}$$

---

8. Minimizar el siguiente AFD



Aplicamos el algoritmo de minimización y lo primero que hacemos es eliminar el estado inaccesible q_5. Construimos y marcamos la tabla triangular de la figura 3.14. Podemos observar que $q_1 \approx q_2$ y $q_3 \approx q_4$. Por tanto el autómata mínimo $M_{\approx}$ es el que se muestra en la figura 3.15.

---

9. Vamos a diseñar un analizador léxico para un sencillo lenguaje de programación denominado MICRO, cuyas especificaciones básicas son las siguientes:

   - *Tipo de dato*: entero.
   - No se declaran *variables*.
   - Los *identificadores* empiezan por letra y se componen de letras, dígitos y símbolos de subrayado.

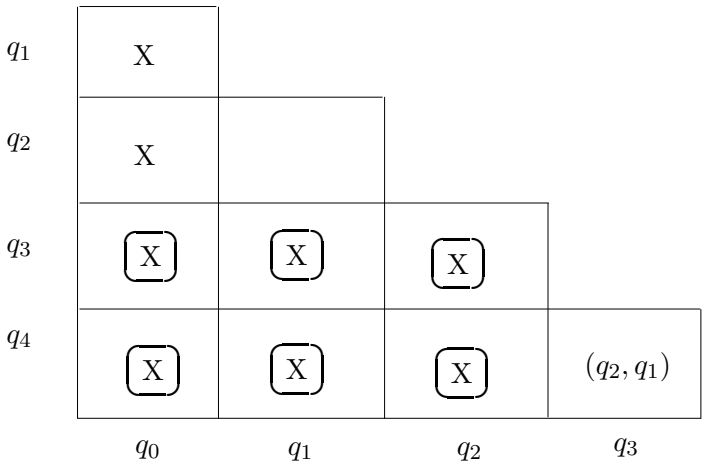


Figura 3.14: Marcado de la tabla

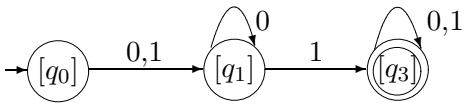


Figura 3.15: Autómata mínimo

- Solo hay *constantes* enteras.
- Los *comentarios* comienzan por $--$ y terminan con EOL.
- *Sentencias*:
  - ID:=expresión;
  - read (lista de ID´s);
  - write (lista de expresiones entre comas);
- *Palabras reservadas*: begin, end, read, write.
- Las *sentencias se separan* con ";".
- El *cuerpo del programa* está *delimitado* por begin/end.
- *Separadores de tokens*: espacios en blanco, tabuladores y new-line.

Expresiones regulares

ID : L(L+D+'-´)*
INTLITERAL : D+
LPAREN : (
RPAREN :)
SEMICOLON : ;
COMMA : ,
ASSIGNOP : :=
PLUSOP : +

MINUSOP : -

SCANEOF : EOF

BLANCOS : (' '+ '\n' + '\t')*

COMMENT : −−C*EOL

Autómata finito

En la figura 3.16 aparece el autómata asociado a las expresiones regulares anteriores.

Implementación

El código que implemente el autómata finito anterior podría ser el que aparece a continuación, contenido en los archivos *lexico.h* y *lexico.c*.

```
/*
 * lexico.h
 */

typedef enum token_types { BEGIN, END, READ, WRITE, ID,
INTLITERAL, LPAREN, RPAREN, SEMICOLON,COMMA, ASSIGNOP, PLUSOP,
MINUSOP, SCANEOF }token;
extern token scanner(void);
extern char
token_buffer[]
```

```
/*
 * lexico.c
 */

#include "lexico.h"
#include <stdio.h>
#include <ctype.h>

void buffer_char(char c);
void clear_buffer(void); token
check_reserved(void); void lexical_error(char c); char
token_buffer[30];

token scanner(void) {   int in_char,c;
    clear_buffer();
    if (feof(stdin))
        return SCANEOF;
    while ((in_char = getchar())!= EOF) {
        if  (isspace(in_char))
            continue; /*do nothing*/
        else if (isalpha(in_char)) {
            /*
             * ID::= LETTER     |ID LETTER
```

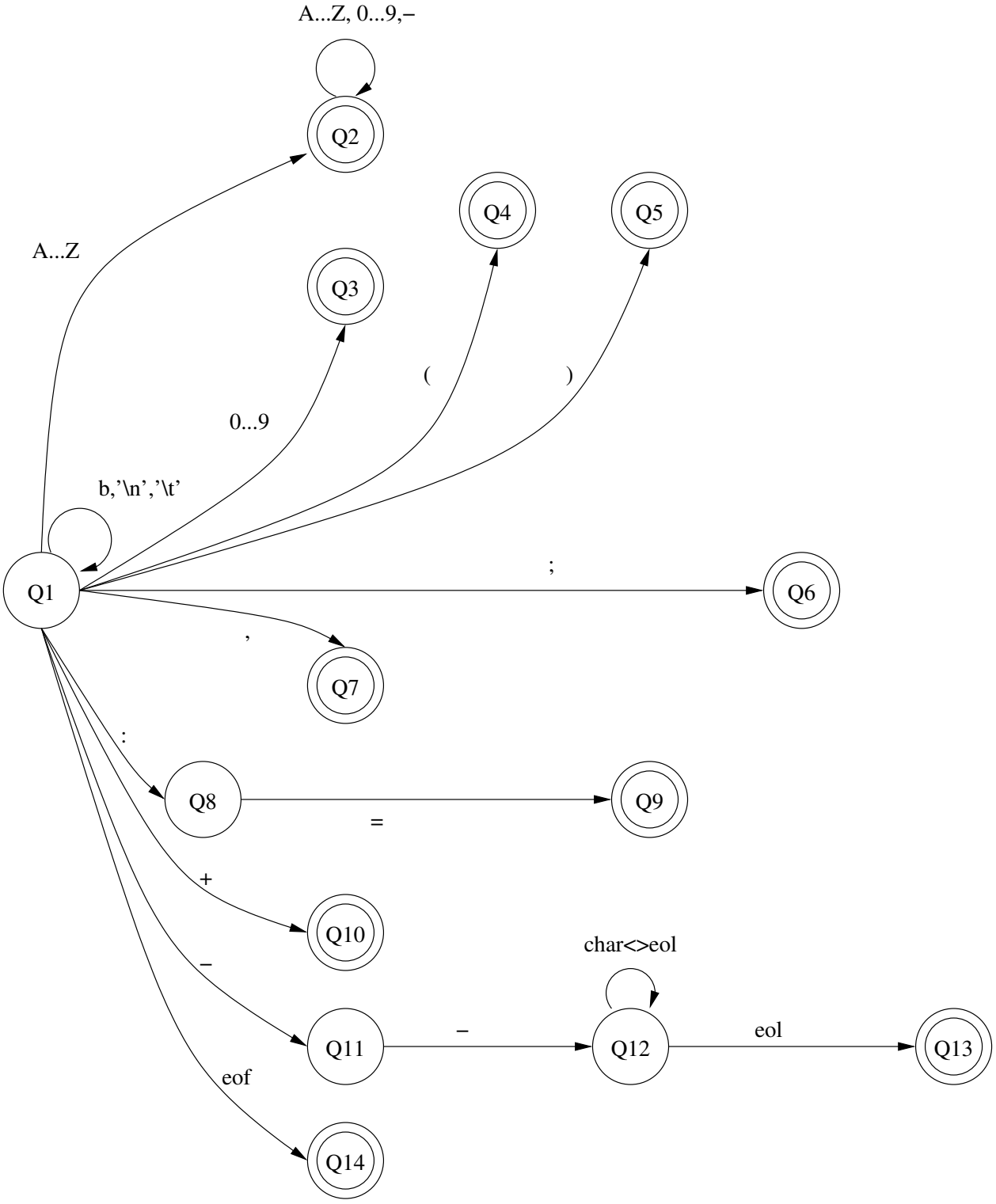


Figura 3.16: Autómata finito para reconocer las ER del lenguaje MICRO

```
         *                  |ID DIGIT
         *                  |ID UNDERSCORE
         */
        buffer_char(in_char);
        for(c=getchar();isalnum(c) || c==‘=‘;c=getchar())
            buffer_char(c);
            ungetc(c,stdin);
            return check_reserved();
        } else if (isdigit(in_char)){
           /*
            * INTLITERAL::= DIGIT
            *             | INTLITERAL DIGIT
            */
            buffer_char(in_char);
            for(c=getchar();isdigit(c);c=getchar())
                buffer_char(c);
            ungetc(c,stdin);
            return INTLITERAL;
        }  else if (in_char == ’(’)
            return LPAREN;
        else if (in_char == ’)’)
            return RPAREN;
            else if (in_char == ’;’)
            return SEMICOLON;
            else if (in_char == ’,’)
            return COMMA;
            else if (in_char == ’+’)
            return PLUSOP;
            else if (in_char == ’:’)
        /*looking for ":="*/
            c = getchar();
            if ( c==‘=´)
                return ASSIGNOP;
            else {
                ungetc(c,stdin);
                lexical_error(in_char);}
        } else if (in_char == ’-’)  {
        /* is it --, comment start */
            c= getchar();
            if ( c == ‘-´) {
                do
                    in_char = getchar();
                while (in_char !=‘\n´);
            } else {
                ungetc(c,stdin);
                return MINUSOP;}
        }else  lexical_error(in_char);
    }}
```

Las *palabras clave* son reservadas, pero no se reconocen en el autómata. La función *check_reserved()* comprueba si el lexema del identificador corresponde a una de ellas.

La *interfaz entre el programa fuente y el AL* se resuelve usando la entrada estándar (stdin) mediante las funciones *getchar()* para leer caracteres y *ungetc(c,stdin)* para devolver los caracteres de anticipación.

La *interfaz entre el AL y el AS* se realiza mediante la devolución de un token cada vez que el AS lo necesita (función *scanner()*). Por otro lado, a través de la variable global *token_buffer*, se pasan al AS los caracteres que forman parte de un identificador o constante entera. La función *buffer_char(c)* se encarga de añadir un carácter al buffer, y *clear_buffer()* borra todos sus caracteres.

El *manejo de errores* lo realiza la función *lexical_error(c)*, que informa de dicho error, continuando el análisis por el siguiente carácter.

EJERCICIOS PROPUESTOS

Se proponen los siguientes ejercicios para resolver en pizarra.

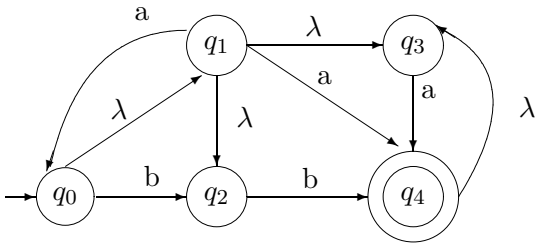
1. Construir AFDs para los siguientes lenguajes:

   $L_1 = \{w \in \{a,b\}^* \mid abab \text{ es subcadena de } w\}$
   $L_2 = \{w \in \{a,b\}^* \mid \text{ ni } aa \text{ ni } bb \text{ es subcadena de } w\}$
   $L_3 = \{w \in \{a,b\}^* \mid ab \text{ y } ba \text{ son subcadenas de } w\}$
   $L_4 = \{w \in \{a,b\}^* \mid bbb \text{ no es subcadena de } w\}$

2. Encontrar un AFD equivalente al $AFND$ $M_N = (\{q_0, q_1\}, \{0,1\}, \Delta, q_0, \{q_1\})$ donde

   $$\Delta(q_0,0) = \{q_0,q_1\},\ \Delta(q_0,1) = \{q_1\},\ \Delta(q_1,0) = \varnothing,\ \Delta(q_1,1) = \{q_0,q_1\}$$

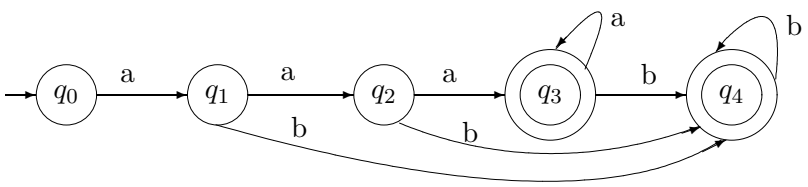
3. Obtener el AFD equivalente al siguiente autómata



4. Construir un AFD y obtener la expresión regular asociada al lenguaje

   $$L = \{w \in \{a,b\}^* \mid \text{cada } a \text{ en } w \text{ está precedida y seguida por una } b\}$$

5. Obtener la expresión regular asociada al lenguaje aceptado por el autómata



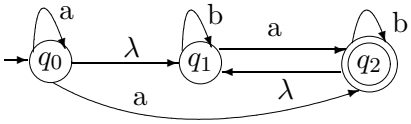
6. Demostrar la hipótesis del teorema 3.2 para probar que $w \in L(M_\lambda) \Leftrightarrow w \in L(M_N)$, para $|w| \geq 1$.

7. Dada la expresión regular $\alpha = a(bc)^*(b + bc) + a$: obtener un $AFND$-λ, un $AFND$ y un AFD que reconozcan $L(\alpha)$ y un cálculo para reconocer la palabra $abcb$ con cada uno de los autómatas.

8. Indicar un método para obtener un AF que reconozca L^R a partir del AF que reconoce L.

9. Obtener un AFD que reconozca el lenguaje formado por palabras sobre el alfabeto $V = \{0, 1\}$ que tienen un número par de ceros o un número de unos que sea múltiplo de tres. Resolver las ecuaciones características para obtener la expresión regular correspondiente.

10. Obtener la expresión regular que describe el lenguaje aceptado por el autómata:



11. Obtener un AFD que reconozca el lenguaje descrito por la expresión regular $(d(ab)^*)^* da(ba)^*$.

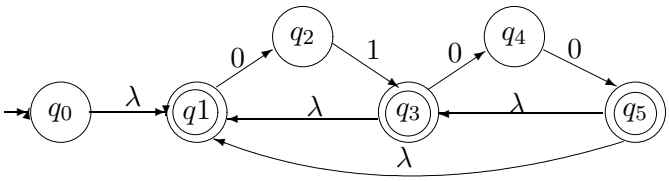
12. Dada la expresión regular $(ab)^*(ba)^* + aa^*$:

    a) Obtener el AFD mínimo equivalente.

    b) Aplicar el método AFD-mínimo $\longrightarrow GR$.

    c) Obtener la gramática G con el método de las derivadas.

13. Dado el siguiente AF que llamaremos M_1:



    a) Mostrar el cálculo por el que M_1 acepta la palabra 0101 y todas las configuraciones de parada posibles ¿Existen cálculos infinitos?

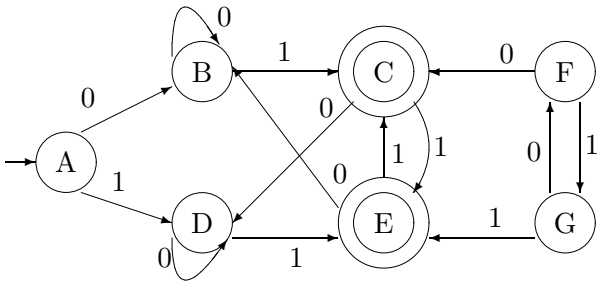
    b) Modificar el diagrama de transición para obtener el $AFND$ equivalente M_2.

    c) ¿Cuantos pasos de cálculo se necesitan en M_2 para aceptar la palabra 0101? En este sentido ¿es más eficiente M_2 que M_1? ¿Por qué?

    d) Mostrar la gramática G_2 a partir de M_2. Obtener una gramática equivalente G_3 sin λ-producciones.

    e) Obtener el AFD mínimo equivalente.

14. Dado el siguiente AFD:



a) Minimizar el autómata.

b) Obtener la gramática regular correspondiente.

15. Dada la expresión regular $0(0+1)^*0 + 10^*(1(0+1)^*0 + \lambda) + \lambda$:

a) Obtener el AFD mínimo M.

b) Obtener la GR que corresponde a la expresión regular por el método de las derivadas.

CUESTIONES BREVES

1. Sea $M_{Q,V}$ el conjunto de todos los $AFDs$ con conjunto de estados Q, alfabeto V, el mismo estado inicial y cada uno de ellos con un sólo estado final. ¿Cuál es el cardinal de $M_{Q,V}$?

2. Sea M un AF (de cualquier tipo) con estado inicial q_0. Si q_0 no es un estado final, ¿podemos asegurar que λ no pertenece a $L(M)$?

3. Si dos AFs tienen distinta función de transición, ¿podemos asegurar que reconocen distintos lenguajes? ¿Y si los dos autómatas fueran deterministas?

4. Dado un $AFND$ M y una palabra $w \in L(M)$, ¿es posible que exista un cálculo para w cuya configuración de parada no sea de aceptación?

5. Explicar cómo se obtendría el AF que reconoce $L(\alpha) \cup L(\beta)$, siendo α y β expresiones regulares.

NOTAS BIBLIOGRÁFICAS

1. Respecto a los distintos tipos de autómatas que hemos visto, nosotros hemos diferenciado entre $AFND$ y $AFND$-λ, siguiendo el ejemplo de autores como [Hop79] (capítulo 2, que recomendamos para las **secciones 1,2,3,4 y 7**).

2. En cuanto a la definición de lenguaje aceptado por un autómata, hemos usado el concepto de configuración y relación de cálculo , como en el libro de [Lew81] (capítulo 2) (consultar para las **secciones 5 y 6**). Hemos preferido usar este enfoque porque consideramos más intuitivo el definir una relación de cálculo, que definir el lenguaje en términos más abstractos mediante el uso de la extensión de la función de transición (como hacen otros autores). Además los conceptos de configuración y cálculo y el uso que se hace de ellos para definir el lenguaje aceptado por una máquina abstracta, seguiremos usándolos para el resto de máquinas que estudiaremos en el curso.

3. Por último, en relación a los teoremas de Kleene, sólo existen diferencias sustanciales en la demostración del teorema de análisis. Nosotros hemos optado por el algoritmo de resolución de sistemas de ecuaciones de expresiones regulares (ver [Isa97], capítulo 3), que se aplica también en el tema anterior para obtener una expresión regular a partir de una gramática regular. Para el teorema de síntesis de Kleene se puede consultar el libro de [Hop79] (capítulo 2).

CAPÍTULO 4:
GRAMÁTICAS LIBRES DEL CONTEXTO

Contenidos Teóricos

1. Definiciones básicas
2. Transformaciones en gramáticas libres del contexto
3. Formas normales

1. Definiciones básicas

1.1. Gramáticas libres del contexto

De entre las cuatro clases de gramáticas de la clasificación de Chomsky, el grupo más importante, desde el punto de vista de la aplicabilidad en teoría de compiladores, es el de las gramáticas independientes o libres del contexto. Las gramáticas de este tipo se pueden usar para expresar la mayoría de estructuras sintácticas de un lenguaje de programación. En este apartado vamos a sentar las bases para el estudio del *parsing*.

Recordemos que las gramáticas libres del contexto tenían la siguiente definición:

Definición 4.1 *Una gramática libre del contexto* $G = (V_N, V_t, S, P)$ *es aquella cuyas producciones tienen la forma* $A \to \alpha$*, siendo* $A \in V_N$ *y* $\alpha \in (V_N \bigcup V_T)^*$.

A continuación, se resumen algunas de las definiciones fundamentales, relacionadas con las gramáticas libres de contexto, que tendrán interés en el estudio de los métodos de análisis sintáctico, hasta llegar a la definición de *árbol de derivación*:

Definición 4.2 *Sea una gramática* $G = (V_N, V_T, P, S)$*. Se dice que la cadena* α *produce directamente a la cadena* β*, denotándolo* $\alpha \Rightarrow \beta$*, si se puede escribir*

$$\alpha = \delta A \mu \ y \ \beta = \delta \gamma \mu$$

para alguna cadena δ *y* $\mu \in (V_T \cup V_N)^*$*, y además existe* $A \to \gamma \in P$.

Si aplicamos repetidamente el concepto de derivación directa, con p.ej:

$$\alpha \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \ldots \Rightarrow \gamma_n = \beta, \ n > 0$$

entonces se dice que la secuencia anterior es una derivación de longitud n. Esta derivación se puede expresar $\alpha \Rightarrow^+ \beta$. Para incluir el caso de la identidad, $\alpha \Rightarrow^* \beta$.

Definición 4.3 *Sea una gramática* $G = (V_N, V_T, P, S)$*. Para cualquier* $A \in V_N$ *y* $\alpha \in (V_N \cup V_T)^*$ *se dice que* $A \Rightarrow^* \alpha$ *si* α *se deriva del axioma, con una cadena de derivaciones de longitud cualquiera, incluso nula.*

Definición 4.4 *El lenguaje definido por una gramática G, denotado $L(G)$ es el conjunto de cadenas de símbolos terminales, que se pueden derivar partiendo del axioma de la gramática, y empleando para las derivaciones las reglas de producción de P. Es decir:*

$$L(G) = \{x / S \Rightarrow^* x \ , \ y \ x \in T^*\}$$

Definición 4.5 *Sea una gramática $G = (V_N, V_T, P, S)$. Las formas sentenciales de G vienen dadas por el conjunto*

$$D(G) = \{\alpha \ / \ S \Rightarrow^* \alpha \ y \ \alpha \in (V_N \cup V_T)^*\}$$

Definición 4.6 *Una forma sentencial x, tal que $x \in V_T^*$ se dice que es una sentencia.*

Definición 4.7 *Sea una gramática $G = (V_N, V_T, P, S)$. Sea una forma sentencial $\alpha\beta\gamma$ en donde $\alpha \in V_T^*$, $\beta \in V_N$ y $\gamma \in (V_T \cup V_N)^*$. Una derivación izquierda se obtiene sustituyendo β por alguna de las partes derechas que la definen.*

Definición 4.8 *Sea una gramática $G = (V_N, V_T, P, S)$. Sea una forma sentencial $\alpha\beta\gamma$ en donde $\alpha \in (V_T \cup V_N)^*$, $\beta \in V_N$ y $\gamma \in V_T^*$. Una derivación derecha se obtiene sustituyendo β por alguna de las partes derechas que la definen.*

Es decir, una derivación más a la izquierda (resp. derecha) para una forma sentencial, es una derivación que, comenzando con el símbolo inicial, acaba en esa forma sentencial, y en cada derivación directa, siempre se aplica una regla de producción correspondiente a la variable más a la izquierda (resp. derecha) de la cadena que se está derivando. Se dice entonces que la forma sentencial es una *forma sentencial izquierda (resp. derecha).*

Un ejemplo de este tipo de derivaciones, para la gramática

$A \rightarrow BF$
$B \rightarrow EC$
$E \rightarrow a$
$C \rightarrow b$
$F \rightarrow c$

puede verse en la figura 4.1.

Definición 4.9 *Sea G una GLC y $\alpha \equiv \gamma_1\beta\gamma_2$ una de sus formas sentenciales. Se dice que β es una frase de la forma sentencial α respecto de la variable A sii:*

$$S \Rightarrow^* \gamma_1 A \gamma_2$$
$$A \Rightarrow^+ \beta$$

Definición 4.10 *Se dice que β es una frase simple sii:*

$$S \Rightarrow^* \gamma_1 A \gamma_2$$

$$A \Rightarrow \beta$$

Definición 4.11 *A una derivación más a la derecha*

$$S \Rightarrow_{md} \gamma_1 \Rightarrow_{md} \gamma_2 \Rightarrow_{md} ...\gamma_{n-1} \Rightarrow_{md} \gamma_n$$

de longitud n, le corresponde una reducción por la izquierda

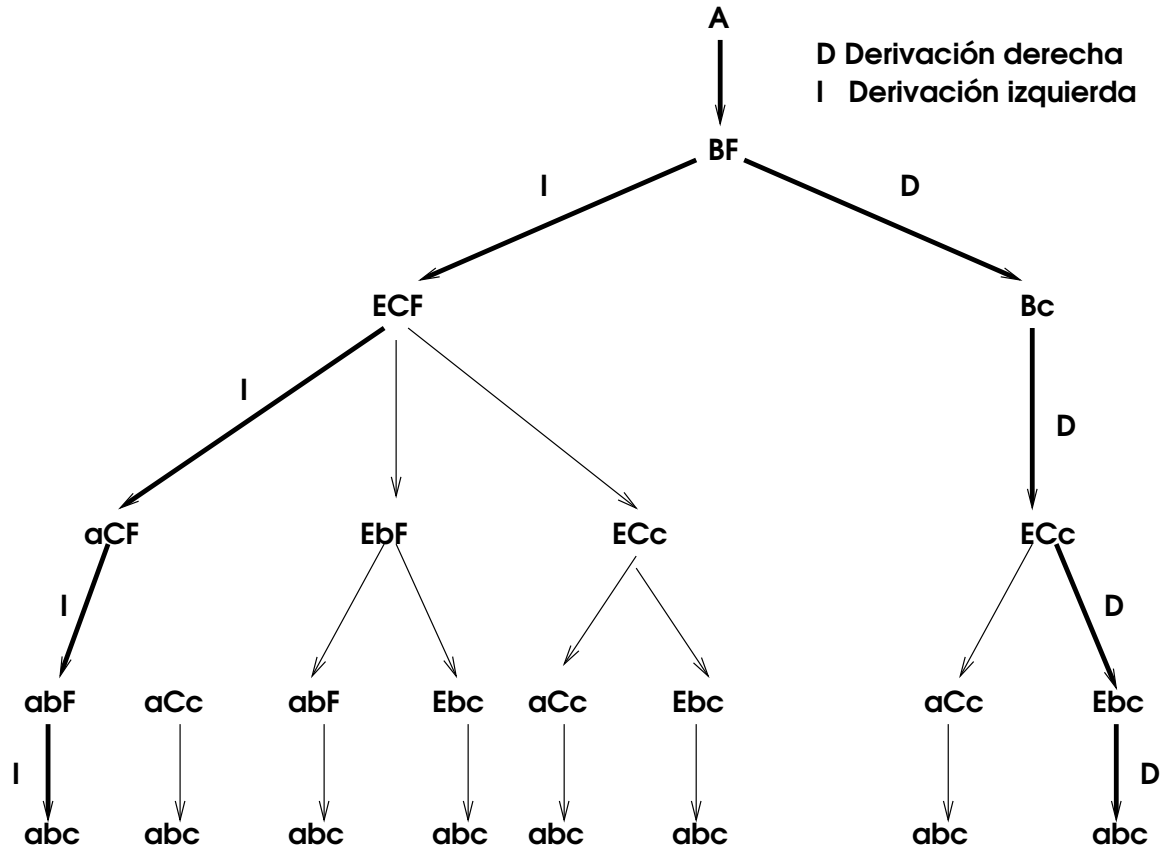


Figura 4.1: Ejemplo de derivaciones izquierda y derecha

$$\gamma_n \Rightarrow^R_{mi} \gamma_{n-1} \Rightarrow^R_{mi} \ldots \Rightarrow^R_{mi} S$$

dónde en cada paso $\gamma_i \Rightarrow^R_{mi} \gamma_{i-1}$ *se sustituye la parte derecha de una regla de producción por su parte izquierda.*
Si $\gamma_i = \alpha_1\beta\alpha_2$ *y* $\gamma_{i-1} = \alpha_1 A\alpha_2$ *entonces* $A \to \beta \in P$.

β es una frase simple, respecto de A, de γ_i, y además es el *pivote* de la forma sentencial.

Definición 4.12 *Se llama pivote de una forma sentencial* α *a la frase simple situada más a la izquierda de* α.

El pivote de una forma sentencial derecha α se calcula obteniendo una derivación más a la derecha hasta llegar a α, y luego, se observa qué frase simple de α se corresponde con la parte derecha de una regla de producción, tal que al aplicarla, se reduzca a la forma sentencial anterior.

Por ejemplo, consideremos la gramática generada por las siguientes producciones:

$S \to zABz$
$B \to CD$
$C \to c$
$D \to d$
$A \to a$

$\alpha \equiv zAcdz$ es forma sentencial derecha, porque

$$S \Rightarrow zABz \Rightarrow zACDz \Rightarrow zACdz \Rightarrow zAcdz$$

c es frase simple de α respecto de C.
d es frase simple de α respecto de D.
El pivote es c, pues es la frase simple situada más a la izquierda.

Definición 4.13 *Un árbol ordenado y etiquetado D es un árbol de derivación para una gramática libre de contexto $G(A) = (V_N, V_T, P, A)$ si:*

1. *La raíz de D está etiquetada con A.*
2. *Si $D_1, \ldots, D_k$ son los subárboles de los descendientes directos de la raíz, y la raíz de cada D_i está etiquetada con X_i, entonces $A \rightarrow X_1 \cdots X_2 \in P$. Además D_i debe ser un árbol de derivación en $G(X_i) = (V_N, V_T, P, X_i)$ si $X_i \in N$, o bien un nodo hoja con etiqueta X_i si $X_i \in T$.*
3. *Alternativamente, si D_1 es el único subárbol de la raíz de D, y la raíz de D_1 tiene como etiqueta λ, entonces $A \rightarrow \lambda \in P$.*

Los árboles que aparecen en la figura 4.2 son árboles de derivación para la gramática G siguiente:

$$S \rightarrow aSbS|bSaS|\lambda$$

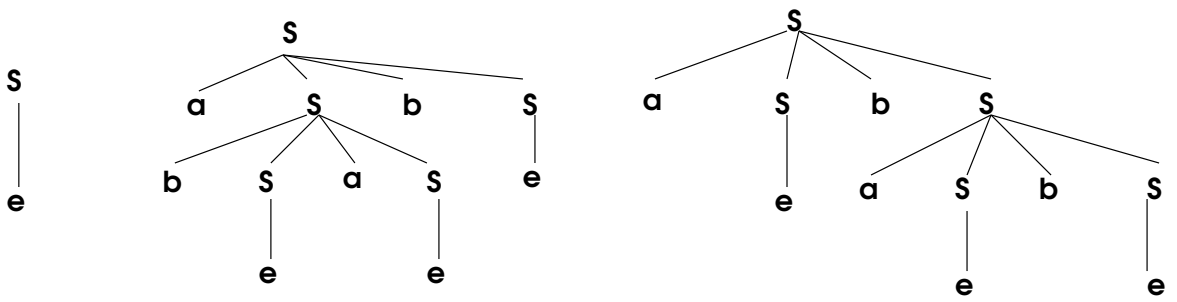


Figura 4.2: Árboles de derivación

En una gramática libre de contexto, una secuencia de derivaciones directas

$$S \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots\gamma_{n-1} \Rightarrow \gamma_n \equiv w$$

que conduzcan del símbolo inicial a una sentencia de la gramática, puede representarse mediante un *árbol de derivación*.
A un mismo árbol pueden corresponderle derivaciones distintas, pero a una derivación le corresponde un único árbol.

1.2. Autómatas de pila (*pushdown automata*)

Si para las expresiones regulares teníamos a nuestra disposición ciertas máquinas abstractas, autómatas finitos, que las reconocían, para las CFG vamos a usar otro tipo de máquina reconocedora denominada autómata de pila. Estas se diferencian de los autómatas finitos en que se ayudan para sus transiciones de una memoria con estructura de pila. Como en los anteriores, la transición entre estados depende del símbolo leído y del estado actual. Cada transición implica la modificación de la pila.

Definición 4.14 *Un autómata de pila se define como una 7-tupla* $AP = (Q, V, \Sigma, \delta, q_0, z_0, F)$ *en donde:*

- Q *es un conjunto finito de estados.*
- V *es el alfabeto de entrada.*
- Σ *es el alfabeto de la pila.*
- q_0 *es el estado inicial.*
- z_0 *es el símbolo inicial de la pila.*
- $F \subseteq Q$ *es el conjunto de estados finales.*
- δ *es la función de transición:*

$$\delta \colon Q \times (V \cup \{\lambda\}) \times \Sigma \to 2^{Q \times \Sigma^*}$$

Observar que el autómata es *no determinista* ya que dado un estado, un símbolo del alfabeto de entrada y otro del alfabeto de la pila, puede pasar a distintos estados y reemplazar el tope de la pila por distintas cadenas γ_i, avanzando o no la cabeza lectora una posición:

$$\delta(q, a, z) = \{(q_1, \gamma_1), (q_2, \gamma_2), \ldots, (q_m, \gamma_m)\}$$

Definición 4.15 *Se entiende por configuración de un autómata con pila a su situación en un instante considerado expresada formalmente por medio de una tripla* $(q, w, \alpha) \in (Q \times V^* \times \Sigma^*)$ *en dónde:*

- $q \in Q$ *es el estado actual del autómata.*
- $w \in V^*$ *es la subcadena de entrada que aun no se ha analizado.*
- $\alpha \in \Sigma^*$ *es el contenido actual de la pila.*

Si $w = \lambda$ no queda nada por analizar. Si $\alpha = \lambda$ se ha reconocido la cadena.

Definición 4.16 *Un movimiento de un AP es una transición entre configuraciones.*

Por ej. el movimiento $(q, aw, Z\alpha) \vdash (q', w, \beta\alpha)$ es un movimiento válido siempre y cuando $(q', \beta) \in \delta(q, a, Z)$ con $q' \in Q$, $a \in (V \cup \lambda)$, $w \in V^*$, α, $\beta \in \Sigma^*$.
Se debe señalar que un AP no puede realizar ningún movimiento si la pila está vacía.

Entonces, un autómata de pila reconocerá una cadena de entrada *por estado final* sii partiendo de su configuración inicial, (q_0, t, Z_0), llega a una configuración final (q_f, λ, α) empleando movimientos válidos y lo expresamos:

$$(q_0, t, Z_0) \vdash^* (q_f, \lambda, \alpha), q_f \in F, \alpha \in \Sigma^*$$

La cadena será aceptada *por vaciado de pila* si después de leerse toda la cadena se llega a un estado con la pila vacía, independientemente del tipo de estado en el que se encuentre el AP.
Veamos un ejemplo. A continuación se presenta la gramática clásica de expresiones aritméticas de suma y producto:

$S \rightarrow S + A$
$S \rightarrow A$
$A \rightarrow A * B$
$A \rightarrow B$
$B \rightarrow (S)$
$B \rightarrow a$

Sea el autómata $AP = (Q, V, \Sigma, \delta, q, s, \emptyset)$ en donde $Q = \{q\}$. y δ se define como:

$\delta(q, \lambda, S) = \{(q, S + A), (q, A)\}$
$\delta(q, \lambda, A) = \{(q, A * B), (q, B)\}$
$\delta(q, \lambda, B) = \{(q, (S)), (q, a)\}$
$\delta(q, OP, OP) = \{(q, \lambda)\}$

siendo $OP = \{a, +, *, (,)\}$.

Una parte del árbol generado para la sentencia $a + a * a$ aparece en la figura 4.3.
Vamos a presentar otro ejemplo: el del reconocimiento de un palíndromo impar restringido. Sea el lenguaje formado por las cadenas del conjunto

$$L = \{tct^r | siendo\ t = (a + b)^+\}$$

Con t^r indicamos la cadena inversa a t. La estrategia a seguir es la de ir apilando la cadena t conforme se va leyendo hasta que aparezca el símbolo c, y después se va comparando símbolo a símbolo, el de entrada con la cabeza de la pila. Si la cadena se reconoce, se agotarán a la vez la entrada y la pila.

El autómata será: $AP = \{\{q_0, q_1, q_2.q_3\}, \{a, b, c\}, \{a, b, c\}, \delta, q_0, z_0, \{q_3\}\}$ La función δ será:

$\delta(q_0, a, z_0) = (q_1, az_0)$
$\delta(q_0, b, z_0) = (q_1, bz_0)$
$\delta(q_1, a, \lambda) = (q_1, a)$
$\delta(q_1, b, \lambda) = (q_1, b)$
$\delta(q_1, c, \lambda) = (q_2, \lambda)$
$\delta(q_2, a, a) = (q_2, \lambda)$
$\delta(q_2, b, b) = (q_2, \lambda)$
$\delta(q_2, \$, z_0) = (q_3, \lambda)$

La representación gráfica es más compleja, y más aún si el autómata es no determinista, que la de los AF, pero también se puede construir una representación basada en tablas, solo que ahora habrá una tabla que determine la transición de los estados, y otra tabla que determine la evolución en la pila. En las siguientes tablas aparece especificado el ejemplo anterior:

En la tabla 4.1 se muestran los cambios de estado, y en la tabla 4.2 se muestran los cambios de pila.

Vamos a verlo con un ejemplo. Veamos como el AP reconoce la cadena *abcba* que efectivamente es un palíndromo.
La secuencia de movimientos sería:

$(q_0, abcba, z_0) \vdash (q_1, bcba, az_0)$
$(q_1, bcba, az_0) \vdash (q_1, cba, baz_0)$
$(q_1, cba, baz_0) \vdash (q_2, ba, baz_0)$

Q	P	a	b	c	$
q_0	z_0	q_1	q_1		
q_1	a	q_1	q_1	q_2	
q_1	b	q_1	q_1	q_2	
q_2	a	q_2			
q_2	b		q_2		
q_2	z_0				q_3

Cuadro 4.1: Cambios de estados

Q	P	a	b	c	$
q_0	z_0	az_0	bz_0		
q_1	α	$a\alpha$	$b\alpha$	α	
q_2	$a\alpha$	α			
q_2	$b\alpha$		α		
q_2	z_0				λ

Cuadro 4.2: Cambios de pila

$(q_2, ba, baz_0) \vdash (q_2, a, az_0)$
$(q_2, a, az_0) \vdash (q_2, \lambda, z_0)$
$(q_2, \lambda, z_0) \vdash (q_3, \lambda, \lambda)$

Un ejemplo de una cadena que no es un palíndromo podría ser el de reconocimiento de la cadena *abcca*, con el que la secuencia de movimientos sería la siguiente:

$(q_0, abcca, z_0) \vdash (q_1, bcca, az_0)$
$(q_1, bcca, az_0) \vdash (q_1, cca, baz_0)$
$(q_1, cca, baz_0) \vdash (q_2, ca, baz_0)$
$(q_2, ca, baz_0) \vdash (!error, ca, baz_0)$

2. Transformaciones en gramáticas libres del contexto

2.1. Factorización

A veces no está claro qué dos producciones alternativas utilizar para derivar con un no-terminal A, puesto que ambas comienzan por la misma cadena de símbolos. En algunos métodos de análisis sintáctico esto supone un problema. En estos casos, se debería reescribir la gramática para retrasar lo más posible esa decisión. Esto lo hacemos factorizando la gramática.

El algoritmo formal para realizar la operación anterior genéricamente es el que se presenta a continuación.

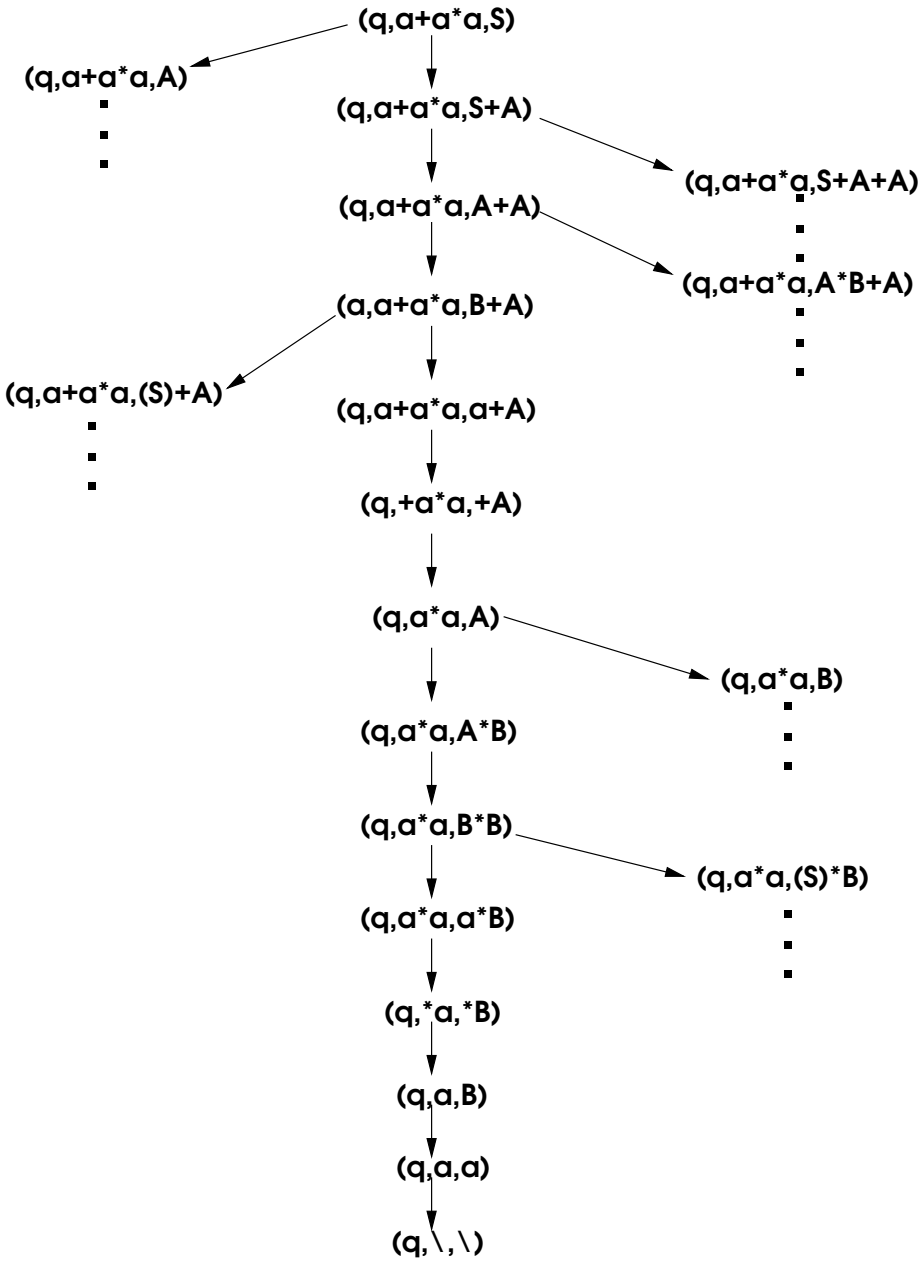


Figura 4.3: Árbol resultado del reconocimiento de $a + a * a$

Algoritmo 4.1

Factorización por la izquierda de una gramática.

Entrada: La gramática G.
Salida: Una gramática equivalente y factorizada por la izquierda.
Método: Para cada no-terminal A, sea α el prefijo más largo común a dos o más de sus alternativas. Si $\alpha \neq \lambda$, o lo que es lo mismo, existe un prefijo común no trivial, se han de sustituir todas las producciones de A,

$$A \rightarrow \alpha\beta_1|\alpha\beta_2|\cdots|\alpha\beta_n|\gamma$$

donde γ representa a todas las partes derechas que no comienzan con α, por

$A \rightarrow \alpha A'|\gamma$
$A' \rightarrow \beta_1|\beta_2|\cdots|\beta_n$

Aplicar la transformación hasta que no haya dos alternativas para un no-terminal con un prefijo común no trivial.

2.2. Eliminación de Símbolos inútiles

Definición 4.17 *Sea $G = (V_N, V_T, S, P)$ una gramática libre de contexto. Decimos que un símbolo $X \in V_N \bigcup V_T$ es útil si existe una derivación de la forma:*

$$S \Rightarrow^* \alpha X \beta \Rightarrow^* w$$

dónde $w \in V_T^*, \alpha, \beta \in (V_N \bigcup V_T)^+$.

Pasos para eliminar los símbolos no útiles de una gramática

1. Eliminación de variables improductivas.
2. Eliminación de símbolos inaccesibles.

Para que el proceso sea correcto, el orden debe ser el anterior.

Definición 4.18 *Una variable $A \in V_N$ es improductiva si no existe ninguna derivación tal que $A \Rightarrow^* w$ con $w \in V_T^*$.*

Definición 4.19 *Un símbolo X es inaccesible si no aparece en ninguna forma sentencial de la gramática, es decir, $\neg\exists\alpha, \beta \in (V_N \bigcup V_T)^*$ tal que $S \Rightarrow^* \alpha X \beta$.*

Eliminación de variables improductivas

Teorema **4.1** *Dada una g.l.c. $G = (V_N, V_T, S, P)$, con $L(G) \neq \emptyset$, existe una g.l.c. equivalente $G' = (V_N', V_T, S, P')$ tal que $\forall A \in V_N'$ se cumple que existe una serie de derivaciones tal que $A \Rightarrow^* w$, $w \in V_T^*$, es decir, existe una gramática equivalente sin variables improductivas.*

El algoritmo para el cálculo de G' (V_N' y P') es el siguiente:

Algoritmo 4.2

```
begin
   OLDV := ∅
   NEWV := {A ∈ V_N | A → w ∈ P, w ∈ V_T^*}
   while OLDV ≠ NEWV do
   begin
      OLDV := NEWV
      NEWV := OLDV ⋃{A ∈ V_N | A → α, α ∈ (V_T ⋃ OLDV)^*}
   end
   V'_N := NEWV
   P' = {A → α ∈ P | A ∈ V'_N, α ∈ (V'_N ⋃ V_T)^*}
end
```

Eliminación de símbolos inaccesibles

Teorema **4.2** *Dada una g.l.c. $G = (V_N, V_T, S, P)$, con $L(G) \neq \emptyset$, existe una g.l.c. equivalente $G' = (V'_N, V_T, S, P')$ sin símbolos inaccesibles.*

El algoritmo para el cálculo de G' (V'_N, V'_T y P') es el siguiente:

Algoritmo 4.3

```
begin
   V'_N := {S}; V'_T := ∅; P' := ∅;
   repeat
      for A ∈ V'_N, A → α_1|α_2|···|α_n, no procesada aún
         {añadir todas las variables de α_i a V'_N
         añadir todos los terminales de α_i a V'_T}
      until V'_N no varíe
   P' = {A → α ∈ P | A ∈ V'_N ∧ α ∈ (V'_N ⋃ V'_T)^*}
end
```

Teorema **4.3** *Dada una gramática libre de contexto G, con L(G)≠ ∅, existe una GLC G' equivalente sin símbolos inútiles.*

Los pasos a seguir serían:

- Pasamos de G a G_1 según el algoritmo 4.2.
- Pasamos de G_1 a G' según el algoritmo 4.3.

G' no contiene símbolo inútiles, es decir, todo símbolo $X \in (V_N \cup V_T)$ es tal que $S \Rightarrow^* \alpha X \beta \Rightarrow^* w$.

2.3. Conversión de una grámatica a λ-libre

Definición 4.20 *Decimos que una gramática l.c. $G = (V_N, V_T, S, P)$ es λ-libre si cumple que en sus reglas de producción no aparece ninguna de la forma $A \to \lambda$, excepto a los sumo $S \to \lambda$, con la condición de que S no aparezca en la parte derecha de ninguna otra regla de producción.*

Teorema **4.4** *Dada una g.l.c. $G = (V_N, V_T, S, P)$, existe una g.l.c. equivalente $G' = (V'_N, V_T, S', P')$ que es λ-libre.*

El algoritmo para calcular G' (V'_N, S' y P') es el siguiente:

Algoritmo 4.4

1. Obtenemos $V_\lambda = \{A \in V_N | A \Rightarrow^* \lambda\}$: Conjunto de variables anulables
   Inicialmente V_λ contiene A si $A \to \lambda$.
   Luego, si tenemos $B \to x_1 x_2 \dots x_n$ y $x_i \in V_\lambda \ \forall i$, añadir B.
2. Obtenemos P' del siguiente modo:
   Por cada producción $A \to x_1 x_2 \dots x_k$ $(k > 0)$ añadimos:
   $A \to Y_1 Y_2 \dots Y_n$, dónde cada Y_i es:
   a) Si x_i no es anulable entonces $Y_i = x_i$
   b) Si $x \in V_\lambda$, entonces se toma Y_i como x_i y como λ
   c) No añadir ninguna producción $A \to \lambda$
3. Si $\lambda \notin L(G)$ entonces $V'_N = V_N$ y $S' = S$.
   En otro caso,
   si S no aparece en la parte derecha
   - Añadir la producción $S \to \lambda$
   - $V'_N = V_N$ y $S' = S$

   en otro caso
   - $V'_N = V_N \bigcup \{S'\}$, siendo S' el nuevo símbolo inicial
   - Añadir a P' $S' \to S|\lambda$

Es importante observar que G' podría quedar con símbolos inútiles.

2.4. Eliminación de producciones unitarias

Definición 4.21 *Llamamos producciones unitarias a las que tienen la forma $A \to B$, con $A, B \in V_N$.*

Teorema 4.5 *Dada una g.l.c. $G = (V_N, V_T, S, P)$ existe una g.l.c. equivalente $G' = (V_N, V_T, S, P')$ que no contiene producciones unitarias.*

El algoritmo para calcular G' es el siguiente:

Algoritmo 4.5

1. Suponemos que G es λ-libre; si no es así, se transforma según el algoritmo 4.4.
2. Para cada $A \in V_N$ se calcula $V_V = \{B \in V_N | A \Rightarrow^+ B\}$.
3. P' = Producciones no unitarias de P.
4. Para cada $A \in V_N$ tal que $V_V(A) \neq \emptyset$.
   Para cada $B \in V_V(A)$
   Para cada $B \to \beta \in P'$ (no unitaria)
   Añadir $A \to \beta$ a P'

En este caso también pueden aparecer símbolos inútiles.

Definición 4.22 *Una gramática libre de ciclos es aquella que no contiene derivaciones de la forma* $A \Rightarrow^* A$.

Definición 4.23 *Una gramática es propia si no tiene símbolos inútiles, es* λ*-libre y libre de ciclos.*

Para convertir una gramática en otra equivalente propia, podemos seguir los siguientes pasos:

1. Pasar la gramática a una equivalente λ-libre.
2. Eliminar las producciones unitarias (no hay ciclos).
3. Eliminar símbolos inútiles.

No debemos olvidar que una gramática puede tener producciones unitarias y ser propia.

2.5. Eliminación de la recursividad por la izquierda

Definición 4.24 *Una g.l.c.* $G = (V_N, V_T, S, P)$ *se dice que es:*

a) Recursiva por la izquierda *si existe* $A \in V_N$ *tal que* $A \Rightarrow^+ A\alpha$.
En este caso A *es una* variable recursiva por la izquierda.

b) Recursiva por la derecha *si existe* $A \in V_N$ *tal que* $A \Rightarrow^+ \alpha A$.
En este caso A *es una* variable recursiva por la derecha.

c) Recursiva *si existe* $A \in V_N$ *tal que* $A \Rightarrow^+ \alpha A \beta$.

Definición 4.25 *Una regla de producción es*

a) Recursiva por la izquierda *si es de la forma* $A \to A\alpha$.

b) Recursiva por la derecha *si es de la forma* $A \to \alpha A$.

c) Recursiva *si es de la forma* $A \to \alpha A \beta$.

Algunos métodos de análisis sintáctico (LL) no pueden manejar gramáticas recursivas por la izquierda. A continuación veremos cómo eliminarla.

El método, con algunas restricciones, puede usarse para eliminar la recursividad por la derecha.

Estudiaremos la eliminación de la recursividad por la izquierda en dos pasos:

(1) Eliminación de reglas recursivas por la izquierda.

(2) Eliminación de la recursividad por la izquierda de la gramática.

(1) Eliminación de la recursividad inmediata por la izquierda

El algoritmo es el siguiente:

Algoritmo 4.6

Eliminación de la recursividad inmediata por la izquierda.
Entrada: Un conjunto de producciones $\{p_i/p_i \in P\}$ con el no terminal $A \in V_N$ como parte derecha de una gramática G CFG sin λ-producciones.
Salida: Un nuevo conjunto de producciones sin recursividad inmediata por la izquierda.

1. Ordénense las A-producciones en la forma

$$A \to A\alpha_1|A\alpha_2|\cdots|A\alpha_m|\beta_1|\beta_2|\cdots|\beta_n$$

   en donde ninguna β_i comienza con A.
2. Sustituir todas las producciones de A por
   $A \to \beta_1 A'|\beta_2 A'|\cdots|\beta_n A'$
   $A' \to \alpha_1 A'|\alpha_2 A'|\cdots|\alpha_m A'|\lambda$
3. La salida es el conjunto de nuevas producciones obtenidas en el paso anterior.

(2) Eliminación de la recursividad por la izquierda de la gramática

El algoritmo anterior elimina la recursividad de un paso, pero no la de dos pasos o más. Esta se elimina con el siguiente algoritmo:

Algoritmo 4.7

Eliminación de la recursividad por la izquierda.
Entrada: Una gramática propia (si no lo es, se transforma).
Salida: Una gramática equivalente, sin recursividad por la izquierda.

1. Ordénense los $A_i \in V_N$ en un orden $A_1, A_2, \ldots, A_n$.
2. **for** i:=1 **to** n **do**
   begin
   a) **for** j:= 1 **to** $i-1$ **do**
      sustituir cada producción de la forma $A_i \to A_j\gamma$ por las producciones $A_i \to \delta_1\gamma|\delta_2\gamma|\cdots|\delta_k\gamma$, en donde $A_j \to \delta_1|\delta_2|\cdots|\delta_k$ es el conjunto de producciones actuales del no terminal A_j;
   b) Además, eliminar la recursividad inmediata por la izquierda de las producciones de A_i.
   end

3. Formas Normales

3.1. Forma Normal de Greibach

Definición 4.26 *Una g.l.c. $G = (V_N, V_T, S, P)$ está en FNG sii*

1. *Es λ-libre.*
2. *Todas las producciones (excepto a lo sumo $S \to \lambda$) son de la forma: $A \to a\Omega$ con $\Omega \in (V_N)^*$ y $a \in V_T$.*

Teorema **4.6** *Dada una g.l.c. $G = (V_N, V_T, S, P)$ existe otra g.l.c. $G' = (V'_N, V_T, S, P')$ que está en FNG.*

El algoritmo para construir la gramática G' (V'_N y P') es el siguiente:

Algoritmo 4.8

1. Suponemos que G es propia y no es recursiva por la izquierda. Si no, se transforma.
2. Si $V_N = \{A_1, A_2, \ldots, A_n\}$ con $A_1 = S$, establecemos un orden total en el conjunto V_N, de forma que $A_1 < A_2 < \ldots < A_n$ y si $A_i \to A_j\alpha \in P$, debe ser $A_i < A_j$ (esto es posible, pues no es recursiva por la izquierda).
3. $V'_N \leftarrow V_N$; $P' \leftarrow P$.
4. `for i=n-1 to 1`
   `for j=n to i+1`
   `Para cada` $A_i \to A_j\alpha \in P'$`, sustituirla por`
   $A_i \to \gamma_1\alpha|\gamma_2\alpha|\cdots|\gamma_p\alpha \in P'$`, dónde` $A_j \to \gamma_1|\gamma_2|\cdots|\gamma_p \in P'$
5. Para cada producción $A \to aX_1X_2\ldots X_k$, $X_i \in (V_N \bigcup V_T)$, $k \geq 1$, sustituirla por $A \to aX'_1X'_2\ldots X'_k$, dónde:
   - Si $X_i \in V_T$, entonces X'_i es una variable nueva, $V'_N = V_N \bigcup \{X'_i\}$
   - Si $X_i \in V_N$, entonces $X'_i = X_i$.
6. Para cada variable nueva X'_i (añadida en 5), se añade a P' la producción $X'_i \to X_i$.

3.2. Forma Normal de Chomsky

Definición 4.27 *Sea una CFG $G = (V_N, V_T, P, S)$. Se dice que G está en Forma Normal de Chomsky (FNC), si es λ-libre y toda producción de P tiene una de las formas siguientes:*

1. *$A \to BC$, en donde A, B y C están en V_N,*
2. *$A \to a$, en donde $A \in V_N$ y $a \in V_T$,*

Teorema **4.7** *Toda gramática libre de contexto puede transformarse en una gramática equivalente en FNC.*

Veamos el algoritmo:

Algoritmo 4.9

Conversión a Forma Normal de Chomsky.
Entrada: Una gramática CFG $G = (V_N, V_T, P, S)$ λ-libre y sin producciones unitarias[1].
Salida: Una gramática G' en forma FNC, tal que $L(G) = L(G')$.
Método: Sea el conjunto P' formado por las producciones siguientes:

[1] single productions

1. Añadir toda producción en la forma $A \to a \in P$
2. Añadir toda producción en la forma $A \to AB \in P$
3. Si $S \to \lambda \in P$ entonces añadirla también.
4. Para cada producción en la forma $A \to X_1 \cdots X_k$ con $k > 2$, añadir las producciones resultantes a continuación: asumimos que X_i' representa a X_i si $X_i \in V_N$, y X_i' es un nuevo no-terminal si $X_i \in V_T$. Las nuevas producciones serán las siguientes:

   $$\begin{aligned}
   &A \to X_1' < X_2 \cdots X_k > \\
   &< X_2 \cdots X_k > \to X_2' < X_3 \cdots X_k > \\
   &\cdots \\
   &< X_{k-2} \cdots X_k > \to X_{k-2}' < X_{k-1} X_k > \\
   &< X_{k-1} X_k > \to X_{k-1}' X_k'
   \end{aligned}$$

   en donde cada $< X_i \cdots X_k >$ es un nuevo símbolo no-terminal.
5. Para cada producción de la forma $A \to X_1 X_2$ en donde bien X_1 o X_2 ó los dos están en V_T, añadir $A \to X_1' X_2'$ a P'
6. Para cada $X_i' \in V_N$ añadido en los pasos 4 y 5, añadir $X_i' \to X_i$. Sea V_N' igual a V_N más todos los nuevos no-terminales introducidos en los pasos anteriores.

La nueva gramática $G = (V_N', V_T, P', S)$ es la deseada.

CAPÍTULO 5:
INTRODUCCIÓN AL ANÁLISIS SINTÁCTICO

Contenidos Teóricos

1. Objetivo del analizador sintáctico
2. Problema de la ambigüedad en el análisis sintáctico
3. Análisis sintáctico ascendente y descendente
4. Método de análisis CYK
5. Análisis sintáctico determinista

1. Objetivo del analizador sintáctico

El papel principal del analizador sintáctico es el de producir una salida, a partir de una cadena de componentes léxicos, que consistirá en:

- Un árbol sintáctico con el que se continuará la siguiente etapa, si la cadena es una frase o conjunto de frases correspondiente a la gramática reconocida por el analizador.
- Un informe con la lista de errores detectados, si la cadena contiene errores sintácticos, e.d. alguna frase o frases que no se ajustan a la estructura sintáctica definida por la gramática correspondiente. Este informe deberá ser lo más claro y exacto posible.

Otras tareas, complementarias a las de generar bien un árbol sintáctico, bien un informe de error, pueden ser las de completar la tabla de símbolos, con información sobre los tokens, tareas correspondientes al análisis semántico como la verificación de tipos, e incluso generación de código.

Manejo de errores en un analizador sintáctico

Los objetivos en el manejo de errores para el AS están bien definidos:

- Debe informar de una forma clara y exacta acerca de los errores sintácticos producidos en el programa fuente.
- Se debe de recuperar de un error, con la suficiente habilidad como para poder seguir detectando en una forma precisa errores posteriores.
- La gestión de errores sintácticos no debe significar un retraso notable en el análisis de programas sintácticamente correctos.

Para seguir profundizando en el tema, veamos la clasificación de respuestas de error que puede encontrarse en [Tre85], debida a Horning (en [Hor76]):

1. Respuestas inaceptables:

a) Respuestas incorrectas (los errores no son comunicados):
   1) El compilador se 'cuelga'.
   2) El compilador cae en un bucle infinito en el análisis.
   3) El compilador continúa con el análisis, produciendo código objeto incorrecto.

b) Respuestas correctas, pero poco útiles:
   1) El compilador informa del primer error y luego termina su ejecución.

2. Respuestas válidas:

a) Respuestas factibles:
   1) El compilador informa de un error, se recupera del mismo y continúa intentando encontrar errores posteriores, si existen.
   2) El compilador informa de un error, lo repara y continúa la traducción, produciendo al final un programa objeto válido.

b) Respuestas no factibles en la actualidad:
   1) El compilador informa de un error, lo repara y continúa la traducción, produciendo al final un programa objeto que hace exactamente lo que el programador quería.

El caso de las respuestas 1.(a) corresponde a compiladores en cuyo diseño nunca se tuvo en cuenta como entrada al compilador programas fuente incorrectos. Por lo tanto puede haber situaciones como la de ausencia de respuesta ó código objeto en apariencia correcto pero que en realidad no se comporta como se esperaba.

El caso de las respuestas 1.(b) corresponde a compiladores en los que se considera que la probabilidad de aparición de un error es ínfima, y por lo tanto únicamente es necesario la detección de un error cada vez.

Dentro de las respuestas válidas y factibles, la menos ambiciosa es la que representa la técnica de `recuperación de errores` (*error recovery*). Cuando el compilador encuentra un error sintáctico se ajusta de tal forma que puede seguir con el análisis, como si nada incorrecto hubiera ocurrido. A partir de aquí pueden pasar dos cosas:

- Idealmente, el compilador puede haberse recuperado totalmente, de tal forma que el error reconocido no va a afectar a la aparición de errores subsiguientes.
- Si se tiene una estrategia de recuperación de errores pobre, se producirá una avalancha de mensajes de error que tienen su origen en un error anterior, y que fácilmente se podía haber aprovechado para descartar muchos errores posteriores. Piénsese en el ejemplo del uso de una variable *i* como índice de varios bucles `for` en un mismo programa `C`. Si esa variable no se ha declarado, o si su declaración se ha hecho de forma incorrecta, todas las apariciones de la misma en el resto del programa generarán el mismo mensaje de error, algo así como `Undefined variable: i`.

El otro caso dentro de las respuestas del tipo 2.(a) es el correspondiente a la técnica de `corrección de errores` (*error repair*) en la que el contenido del programa fuente se modifica con el objeto de hacerlo sintácticamente correcto. En estos casos resulta útil como salida el programa fuente reparado, como apoyo para el diagnóstico posterior. A lo peor la corrección no es válida semánticamente (el programa no hace lo que el programador esperaba) pero sugiere una forma de corrección de error desde un punto de vista sintáctico.

El caso ideal es el del compilador que realiza una corrección de errores correcta. Aquí existe una paradoja y es que si somos capaces de realizar un programa traductor que entiende qué es lo que queremos realizar, entonces, ¿para qué usar un lenguaje de alto nivel para volver a decírselo?

¿Cómo facilitar la detección de errores?

Podemos facilitar la detección de errores cuando estamos diseñando la gramática. Para manejar errores adecuadamente primero hay que detectarlos. Si diferenciamos los compiladores actuales en dos tipos:

- aquellos que se han producido a partir de una rigurosa especificación de su gramática (libre de contexto), y usan análisis dirigido por la sintaxis y
- aquellos que se han producido *ad-hoc* o lo que es lo mismo, sin seguir un método formal de especificación del lenguaje y de desarrollo del programa. Estos compiladores existen por razones de eficiencia,

podemos asegurar que los errores en una gramática libre de contexto se detectan en una forma más efectiva y metodológica. Por ejemplo, los métodos de análisis **LL** y **LR** detectan un error lo antes posible, basándose en la propiedad denominada de *prefijo viable*, es decir, que detectan un error en cuanto se encuentra una cadena que no es prefijo de ninguna cadena del lenguaje.

Algunas técnicas concretas de recuperación de errores

- La técnica de recuperación de errores más sencilla y conocida es la de **recuperación en modo pánico**. Cuando se descubre un *token* que no concuerda con la especificación sintáctica del lenguaje, se siguen admitiendo tokens hasta que se encuentra uno que es de un conjunto especial denominado *de sincronización* como por ejemplo un ';' ó un `end`. El inconveniente principal que presenta es que pasa por alto una gran cantidad de tokens en la entrada, de los cuales no comprueba más errores, antes de encontrar uno de sincronización; por contra es muy sencillo de implementar y está libre de bucles infinitos.

- Otra técnica es la de **recuperación a nivel de frase**. En esta, cuando se descubre el error, se realiza una corrección local insertando una cadena que permita continuar con el análisis sintáctico. Por ejemplo, podría sustituir un ';' cuando encuentra un '.'. Sin embargo, tiene como desventaja su dificultad para afrontar situaciones en las que el error se produjo antes del punto de detección. Por otro lado, se corre el riesgo de caer en bucles infinitos.

- Una técnica muy atractiva desde el punto de vista formal es la de **producciones de error**. Si la especificación de la gramática se ha hecho de forma correcta, y se conoce en qué puntos suelen estar los errores más frecuentes, se puede ampliar la gramática con reglas de producción que simulen la producción de errores. Así, si se produce un error contemplado por la gramática ampliada, el análisis podría seguir y el diagnóstico producido sería el adecuado.

- La técnica de **corrección global** se asienta en algoritmos que calculan la distancia mínima de una cadena incorrecta a una cadena correcta en términos de cambios en la primera para convertirla en la segunda. Estos métodos son demasiado costosos aunque tienen mucho interés teórico. Por ejemplo, pueden utilizarse para realizar una evaluación de otras técnicas de recuperación de errores, o bien pueden ser usados localmente para encontrar cadenas de sustitución óptimas en una recuperación a nivel de frase.

2. Problema de la ambigüedad en el análisis sintáctico

Definición 5.1 *Decimos que una sentencia w de una GLC es ambigua , si podemos encontrar para ella dos o más árboles de derivación distintos, o bien, si podemos derivar la sentencia*

mediante dos o más derivaciones más a la izquierda (o más a la derecha) distintas.

Definición 5.2 *Una gramática es ambigua si tiene al menos una sentencia ambigua.*

El hecho de que una gramática sea ambigua es una situación indeseable. Si la gramática se usa para definir la sintaxis de un lenguaje de programación, la ambigüedad de una sentencia puede implicar significados distintos para ella. Esto implicaría una ejecución distinta de la misma sentencia, y, por lo tanto, la posibilidad de producir resultados distintos.

En el ejemplo siguiente, no está clara la precedencia de los operadores + y $*$, y por lo tanto es posible generar, a partir de la sentencia $a+a*a$, dos árboles de derivación distintos, dependiendo de si se calcula antes la suma ó el producto. La gramática posee un conjunto P con las reglas de producción $E \rightarrow E + E|E * E|a$. Para esa cadena se generarán dos árboles (figura 5.1).

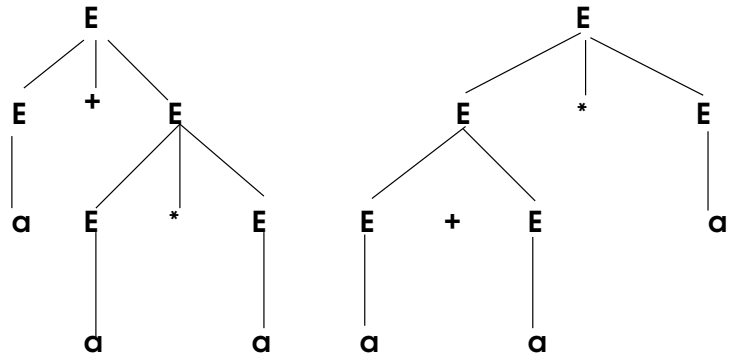


Figura 5.1: Ejemplo de distintas precedencias

El árbol de la izquierda representa un análisis en el que se ha concedido más precedencia al producto que a la suma. En el árbol de la derecha se concede una precedencia más alta a la suma, que se calcula antes.

Por otro lado, la sentencia $a + a + a$ también sería ambigua, pues a partir de ella podrían generarse dos árboles distintos, según que se considerara la asociatividad por la izquierda o por la derecha.

El proceso de decidir si una gramática es ambigua no es algorítmico.

Por otro lado, es importante hacer notar que cuando una gramática es ambigua, el pivote de una forma sentencial derecha no tiene por qué ser único.

Por ejemplo, si consideramos la forma sentencial $E + E * a$, generada por la gramática ambigua anterior, podemos considerar los pivotes a y $E + E$, puesto que podemos considerar las dos sigientes derivaciones más a la derecha:

$$E \Rightarrow_{md} E + E \Rightarrow_{md} E + E * E \Rightarrow_{md} E + E * a$$

$$E \Rightarrow_{md} E * E \Rightarrow_{md} E * a \Rightarrow_{md} E + E * a$$

En algunos casos la ambigüedad puede eliminarse encontrando una gramática equivalente no ambigua. Es decir, el lenguaje sería no ambiguo.
Por ejemplo, si transformamos la gramática anterior en esta otra:

$E \rightarrow E + T|T$
$T \rightarrow T * F|F$
$F \rightarrow a$

se *retrasa* una derivación con la regla de producción que produce expresiones de producto. Lo que hacemos con esto es conseguir dotarla de más precedencia. Con esto resolvemos el problema de la precedencia, e incluimos asociatividad por la izquierda.

En otros casos, sin embargo, podemos encontrarnos con un *lenguaje inherentemente ambiguo*, como el siguiente:

$$L = \{a^n b^n c^m d^m / m, n \geq 1\} \cup \{a^n b^m c^m d^n / n, m \geq 1\}$$

Este lenguaje es generado por la gramática con las siguientes producciones:

$S \rightarrow XY|V$
$X \rightarrow aXb|ab$
$Y \rightarrow cYd|cd$
$V \rightarrow aVd|aZd$
$Z \rightarrow bZc|bc$

Las sentencias de la forma $a^i b^i c^i d^i$ son ambiguas. En la página 100 de [Hop79] puede encontrarse la demostración de que L es inherentemente ambiguo.

El problema del *else* ambiguo

Otro ejemplo clásico de este tipo de problemas es el de las gramáticas que incluyen sentencias del tipo `if-then/if-then-else`. Supóngase la gramática

prop	$\rightarrow$	**if** expr **then** prop
	|	**if** expr **then** prop **else** prop
	|	$S_1 \mid S_2$
expr	$\rightarrow$	$E_1 \mid E_2$

De acuerdo con ella, la sentencia

if E_1 **then** S_1 **else if** E_2 **then** S_2 **else** S_3

no es ambigua, ya que el árbol de derivación correspondiente sería el de la figura 5.2.

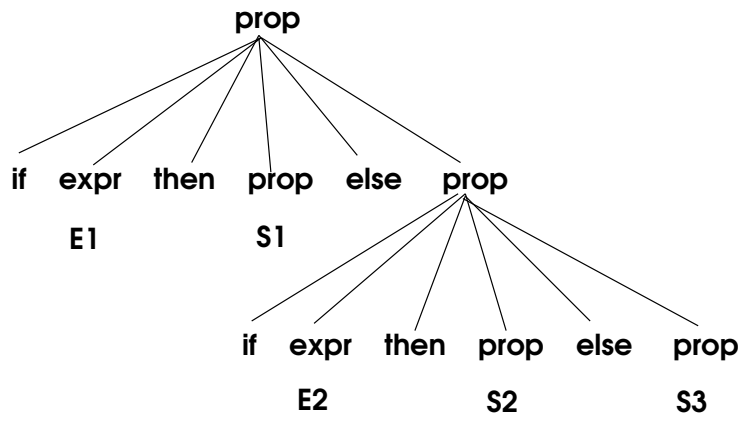


Figura 5.2: Ejemplo de derivación

Sin embargo, la sentencia

if E_1 **then if** E_2 **then** S_1 **else** S_2

si lo sería, ya que daría lugar a la pareja de árboles de derivación distintos de la figura 5.3.

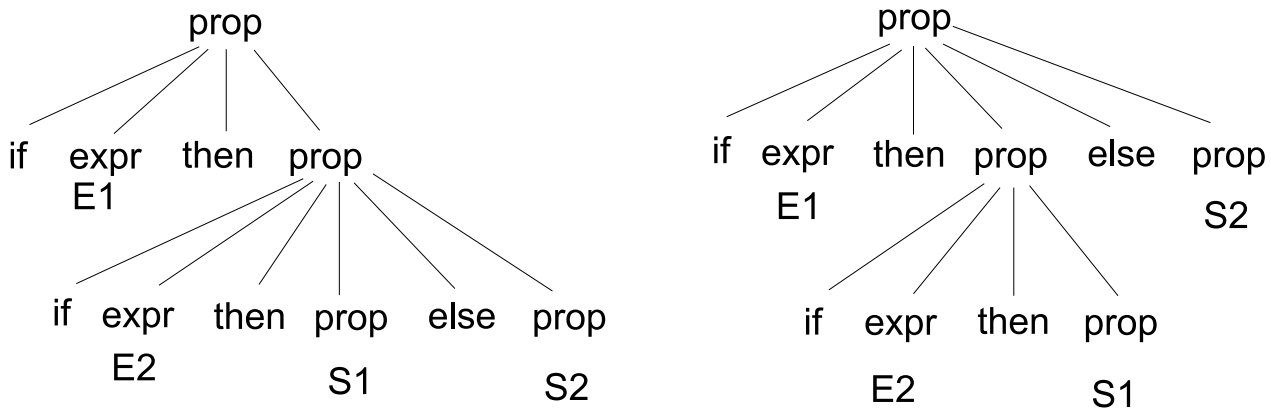


Figura 5.3: Ejemplo de derivación

En el árbol de la derecha, el **else** se asocia al primer **if**. En el de la izquierda la asociación del **else** se hace con el segundo. Esta suele ser la interpretación válida, por convenio, en la mayoría de los lenguajes de programación: asociar el *else* al *if* más próximo. Hay dos enfoques distintos usados para solucionar este problema. El primero es el de transformar la gramática, y la definición del lenguaje mismo para que las construcciones `if-then-else` tengan delimitadores de bloque, y los **else** se asocian con los **if** explícitamente. La segunda es transformar la gramática en otra equivalente (e.d. que reconozca exactamente el mismo lenguaje) y que no sea ambigua. Esto resulta particularmente útil para el análisis predictivo.
Resulta evidente que la más drástica, en términos de influir incluso en el diseño del propio lenguaje de programación es la primera solución. Por ejemplo, si usamos la nueva gramática

$$\begin{array}{lcl} \text{prop} & \rightarrow & \textbf{if}\ \text{expr}\ \textbf{then}\ \text{prop}\ \textbf{endif} \\ & | & \textbf{if}\ \text{expr}\ \textbf{then}\ \text{prop}\ \textbf{else}\ \text{prop}\ \textbf{endif} \\ & | & S_1 \mid S_2 \\ \text{expr} & \rightarrow & E_1 \mid E_2 \end{array}$$

entonces, para escribir una sentencia como la del ejemplo, y en la que se asocie el **else** al segundo **if** quedaría

if E_1 **then if** E_2 **then** S_1 **else** S_2 **endif endif**

Una sentencia que ahora asociara el **else** con el primer **if** sería

if E_1 **then if** E_2 **then** S_1 **endif else** S_2 **endif**

La solución menos drástica es la segunda. Para acometerla es necesario decir que se prefiere el árbol de la izquierda, o lo que es lo mismo `emparejar el else con el then anterior y sin emparejar más cercano`. La idea es dividir las proposiciones entre emparejadas y no emparejadas, y toda proposición que aparezca entre un **then** y un **else** debe estar emparejada, e.d. no debe terminar con un *then* sin emparejar porque entonces el **else** estaría obligado a concordar con ella. Una proposición emparejada es o una proposición **if-then-else** que no contenga proposiciones sin emparejar o cualquier otra clase de proposición no condicional.

prop	$\rightarrow$	prop_emparejada
	|	prop_no_emparejada
prop_emparejada	$\rightarrow$	**if** expr **then** prop_emparejada **else** prop_emparejada
	|	$S_1 \mid S_2$
prop_no_emparejada	$\rightarrow$	**if** expr **then** prop
	|	**if** expr **then** prop_emparejada **else** prop_no_emparejada
expr	$\rightarrow$	$E_1 \mid E_2$

3. Análisis sintáctico ascendente y descendente

Primero comenzaremos con una definición del término *parsing* (análisis sintáctico).

Definición 5.3 *Una sentencia $w \in L(G)$, para alguna CFG (*Context Free Grammar*) ha sido reconocida cuando conocemos alguno de (o quizá todos) sus árboles de derivación.*

Podríamos pensar que en un traductor, ese árbol podría estar almacenado físicamente en memoria. Sin embargo, por lo general, la representación del árbol es más sutil.

La mayoría de los compiladores realizan el *parsing* simulando un AP que reconoce la entrada, bien usando un enfoque *top-down* ó *bottom-up*.
Por lo tanto, existen dos grandes grupos de métodos de análisis sintáctico, dependiendo de la dirección en la que se vaya creando el árbol sintáctico.

- Descendente: en este tipo de análisis, se va recorriendo el árbol sintáctico desde la raíz hasta las hojas, llegando a generar la sentencia que se está analizando. La raíz representa al símbolo inicial de la gramática.
- Ascendente: se parte de las hojas y se intenta construir el árbol hacia arriba, hasta llegar al símbolo inicial de la gramática.

Se puede clarificar un poco el concepto añadiendo que la habilidad de un AP para realizar un análisis *top-down* está asociado con la habilidad de un traductor basado en ese tipo de autómata para hacer corresponder cadenas de entrada con sus derivaciones izquierdas. Asímismo, la habilidad de un AP para realizar un análisis sintáctico *bottom-up* está asociada con la habilidad de un traductor de este tipo para hacer corresponder cadenas de entrada con las inversas de sus derivaciones derechas.

Entre los métodos generales, aparte de la simulación con retroceso (tanto ascendente como descendente) que veremos en los próximos temas, los algoritmos de *Cocke-Younger-Kasami* (CYK) y el método de *Early* son los más conocidos. Sin embargo, estos métodos son bastante ineficientes desde un punto de vista computacional. Los estudiaremos en este tema.

Afortunadamente, para la mayoría de lenguajes de programación es suficiente con trabajar un subconjunto de CFG, como las LL y LR que permiten algoritmos de *parsing* más eficientes.

En el caso de las gramáticas LL, el método que se aplica es descendente, mientras que en las LR se trata de un método ascendente. Estudiaremos estos métodos en los temas dedicados al análisis descendente y ascendente, respectivamente.

De igual forma, también es ascendente un método quizá más restrictivo (aplicable sólo a las llamadas gramáticas de operador), pero extremadamente simple, denominado *método por precedencia de operadores*, que estudiaremos en el tema dedicado al análisis ascendente.

4. Método de análisis CYK

El método que se va a ver en esta sección, CYK (Cocke-Younger-Kasami), es aplicable a cualquier gramática libre de contexto. El algoritmo requiere una complejidad espacial de n^2 y de tiempo de n^3.
Supongamos que la cadena a reconocer es $w = a_1a_2 \dots a_n$. El algoritmo trabaja construyendo una tabla T que va a ser triangular, y que va a estar formada por elementos t_{ij}, con $1 \leq i \leq n$ y $1 \leq j \leq n-i+1$. Cada t_{ij} va a contener un subconjunto de V_N. Así, el no-terminal A va a estar en t_{ij} sii $A \Rightarrow^+ a_ia_{i+1} \dots a_{i+j-1}$, o lo que es lo mismo, si de A se derivan los correspondientes j símbolos de w comenzando en la i-ésima posición. Por lo tanto, la cadena w será reconocida por la gramática si en t_{1n} está el símbolo inicial S.

Veamos el algoritmo que construye la tabla T.

Algoritmo 5.1

Algoritmo de análisis sintáctico de Cocke-Younger-Kasami.
Entrada: Una gramática $G = (V_N, V_T, P, S)$ en CNF y sin λ-producciones, junto con una cadena de entrada $w = a_1a_2 \cdots a_n \in V_T^*$.
Salida: La tabla T en la que cada $t_{i,j}$ contiene a $A \in V_N$ sii $A \Rightarrow^+ a_ia_{i+1} \cdots a_{i+j-1}$.
Método:

1. Hacer $t_{i,j} = \{A | A \to a_i \in P\}$ para todo i.
2. Una vez que $t_{i,j}$ se ha calculado para todo i, $1 \leq i \leq n$, y para todo j', $1 \leq j' < j$. Hágase $t_{i,j} = \{A |$ para algún k $, 1 \leq k < j,\ A \to BC \in P,\ B \in t_{i,k},\ y\ C \in t_{i+k,j-k}\}$ Dado que $i \leq k < j$, tanto k como $j-k$ son menores que j. Por lo tanto, $t_{i,k}$ y $t_{i+k,j-k}$ han sido calculados antes de $t_{i,j}$. Después de este paso, si $t_{i,j}$ contiene a A, entonces
$$A \Rightarrow BC \Rightarrow^+ a_i \cdots a_{i+k-1}C \Rightarrow^+ a_i \cdots a_{i+k-1}a_{i+k} \cdots a_{i+j-1}$$
3. Realizar el paso anterior, hasta que $t_{i,j}$ haya quedado calculado para todo $1 \leq i \leq n$ y $1 \leq j \leq n-i+1$.

Después de haber ejecutado el paso 1, en $t_{1,1}$ vamos a tener los no-terminales que producen directamente a_1. En $t_{2,1}$ vamos a tener los no-terminales que producen directamente a_2, y así sucesivamente. Ahora, se deben ir rellenando columnas, y sus correspondientes filas de forma incremental.

Sea $j = 2$. Entramos en el paso 2, y ahora calculamos todos los $t_{i,2}$ en base a los $t_{i,1}$. Por lo tanto, después de haber ejecutado este paso en la primera iteración, en cada $t_{k,2}$, con $1 \leq k \leq i$ tendremos el no-terminal A tal que $A \to a_ka_{k+1}$. Debido a que la gramática está en formato CNF esto solo podemos hacerlo con las producciones del tipo $A \to BC$, porque no existen producciones del tipo $A \to ab$, con $a, b \in V_T$.

Sea ahora $j = 3$. Volvemos a ejecutar el paso 2. Ahora se van a calcular todos los $t_{i,3}$. Es decir, aquellos no-terminales que produzcan cadenas de longitud 3, $a_ka_{k+1}a_{k+2}$ con $1 \leq k \leq n-2$.

Vamos a calcular $t_{1,3}$. O dicho de otro modo, vamos a calcular el conjunto de no-terminales a partir de los cuales podemos obtener la cadena $a_1a_2a_3$. Tendremos que encontrar producciones del tipo $A \to BC$ tales que:

- o bien B está en $t_{1,1}$, y por lo tanto tendría producciones de la forma $B \to a_{11}$, y por lo tanto C debe estar forzosamente en $t_{2,2}$, con lo que tendría una producción $C \to DE$ y a su vez D estaría en $t_{2,1}$, con una producción $D \to a_2$ y E en $t_{3,1}$ con una producción $D \to a_3$,

- o bien B está en $t_{1,2}$, y por lo tanto tendría una producción del tipo $B \to DE$, con D en $t_{1,1}$, y $D \to a_1 \in P$ y E en $t_{2,1}$ con $E \to a_2 \in P$, y finalmente C estaría en $t_{3,1}$ con una producción $C \to a_3 \in P$.

Calculemos ahora $t_{2,3}$. O dicho de otro modo, el conjunto de no-terminales a partir de los cuales podemos obtener la cadena $a_2a_3a_4$. Volvemos a ejecutar el paso 2. Ahora se han de calcular todos los $t_{i,3}$. Es decir, aquellos no-terminales que produzcan cadenas de longitud 3, y que comiencen a partir de a_2. Tendremos que encontrar producciones del tipo $A \to BC$ tales que:

- o bien B está en $t_{2,1}$, y por lo tanto tendría producciones de la forma $B \to a_2$, y C está en $t_{3,2}$, con lo que tendría una producción $C \to DE$, y a su vez, D estaría en $t_{3,1}$, con una producción $D \to a_3$ y E en $t_{4,1}$ con una producción $D \to a_4$,
- o bien B está en $t_{2,2}$, con una producción de la forma $B \to DE$, con D en $t_{2,1}$ y $D \to a_2$, y $E \in t_{3,1}$ con una producción $E \to a_3$, y C estaría en $t_{4,1}$, con una producción $C \to a_4$.

El mismo razonamiento seguiría desde $t_{3,3}$, y hasta $t_{n-3+1,3}$. Después volveríamos a variar j para hacerla $j = 4$, y así hasta que todos los t_{ij} con $1 \leq i \leq n$, y $1 \leq j \leq n - i + 1$ se hayan calculado.

Un ejemplo práctico

Veámos la aplicación del algoritmo con un ejemplo en el que el conjunto P de nuestra gramática viene dado por el conjunto de producciones

$S \to AA|AS|b$
$A \to SA|AS|a$

Sea $w = abaab$ la cadena de entrada. Aplicando el algoritmo, la tabla resultante será una triangular de 5×5. Aplicando el paso 1, para

- $t_{11} = \{A\}$, ya que $A \to a \in P$.
- $t_{21} = \{S\}$, ya que $S \to b \in P$.
- $t_{31} = \{A\}$, ya que $A \to a \in P$.
- $t_{41} = \{A\}$, ya que $S \to a \in P$.
- $t_{51} = \{S\}$, ya que $A \to b \in P$.

Ahora, hacemos $j = 2$. Tenemos que $t_{i,j'}$ se ha calculado para $j' = 1$. Tenemos que encontrar no-terminales que produzcan subcadenas de w, longitud 2.

- $t_{1,2} = \{S, A\}$, ya que $1 \leq k < 2$, y la regla ha de ser tal que el primer no-terminal de la parte derecha esté en $t_{1,1}$ y el segundo no-terminal en $t_{2,1}$. Por lo tanto la parte derecha ha de ser AS. Tanto S como A tienen reglas con esa parte derecha.
- $t_{22} = \{A\}$, ya que $1 \leq k < 2$, y la regla ha de ser tal que el primer no-terminal de la parte derecha esté en $t_{2,1}$ y el segundo no-terminal en $t_{3,1}$. La parte derecha sería SA. Únicamente A tiene partes derechas de ese tipo.
- $t_{3,2} = \{S\}$, ya que $1 \leq k < 2$, y la regla ha de ser tal que el primer no-terminal de la parte derecha esté en $t_{3,1}$, y el segundo en $t_{4,1}$. Por lo tanto, la parte derecha sería AA. Solamente S tiene reglas de producción con esa parte derecha.

	1	2	3	4	5
1	$\{A\}$	$\{S, A\}$			
2	$\{S\}$	$\{A\}$			
3	$\{A\}$	$\{S\}$			
4	$\{A\}$	$\{A, S\}$			
5	$\{S\}$				

Cuadro 5.1: T después de hacer el paso 2, con $j = 2$

	1	2	3	4	5
1	$\{A\}$	$\{S, A\}$	$\{A, S\}$		
2	$\{S\}$	$\{A\}$	$\{S\}$		
3	$\{A\}$	$\{S\}$	$\{A, S\}$		
4	$\{A\}$	$\{A, S\}$			
5	$\{S\}$				

Cuadro 5.2: T después de hacer el paso 2, con $j = 3$

- $t_{4,2} = \{A, S\}$, ya que $1 \leq k < 2$, y la regla ha de ser tal que el primer no-terminal de la parte derecha esté en $t_{4,1}$, y el segundo en $t_{5,1}$. Por lo tanto, la parte derecha sería AS.

Después de hacer el paso 2, con $j = 2$ queda la tabla que se muestra en 5.1
Ahora, hacemos $j = 3$. Tenemos que $t_{i,j'}$ se ha calculado para $1 \leq j' < 3$. Tenemos que encontrar no-terminales que produzcan subcadenas de w, de longitud 3.

- $t_{1,3} = \{A, S\}$, ya que $1 \leq k < 3$, y la regla ha de ser tal que el primer no-terminal de la parte derecha esté en $t_{1,1}$ (o en $t_{1,2}$) y el segundo no-terminal en $t_{2,2}$ (o en $t_{3,1}$). Por lo tanto la parte derecha ha de ser AA (o SA).

- $t_{2,3} = \{S\}$, ya que $1 \leq k < 3$, y la regla ha de ser tal que el primer no-terminal de la parte derecha esté en $t_{2,1}$ (o en $t_{2,2}$) y el segundo no-terminal en $t_{3,2}$ (o en $t_{4,1}$). La parte derecha sería SS (o AA). Únicamente S tiene una producción $S \rightarrow AA$.

- $t_{3,3} = \{A, S\}$, ya que $1 \leq k < 3$, y la regla ha de ser tal que el primer no-terminal de la parte derecha esté en $t_{3,1}$ (o en $t_{3,2}$), y el segundo en $t_{4,2}$ (o en $t_{5,1}$). Con $t_{3,1}$ y $t_{4,2}$ tenemos dos posibles partes derechas que son AA y AS, y por ello tanto S como A deben estar en $t_{3,3}$. Con $t_{3,2}$ y $t_{5,1}$ tenemos como parte derecha SS, que no es generada por ningún no-terminal.

Después de hacer el paso 2, con $j = 3$ queda la tabla que se muestra en 5.2
Ahora, hacemos $j = 4$. Tenemos que $t_{i,j'}$ se ha calculado para $1 \leq j' < 4$. Tenemos que encontrar no-terminales que produzcan subcadenas de w, de longitud 4.

- $t_{1,4} = \{A, S\}$, ya que $1 \leq k < 4$, y la regla ha de ser tal que el primer no-terminal de la parte derecha esté en $t_{1,1}, k = 1$, o en $t_{1,2}, k = 2$, o en $t_{1,3}, k = 3$ y el segundo no-terminal en $t_{2,3}, k = 1$, o en $t_{3,2}, k = 2$ o en $t_{4,1}$. El conjunto de no-terminales que podrían formar el primer no-terminal de la parte derecha es $\{A, S\}$ y el de no-terminales que podrían formar el segundo no-terminal de la parte derecha es $\{A, S\}$. Por lo tanto la parte derecha va a estar en $\{AA, AS, SA, SS\}$.

- $t_{2,4} = \{A, S\}$, ya que $1 \leq k < 4$, y la regla ha de ser tal que el primer no-terminal de la parte derecha esté en $t_{2,1}, k = 1$, o en $t_{2,2}, k = 2$ o en $t_{2,3}, k = 3$ y el segundo no-terminal

	1	2	3	4	5
1	$\{A\}$	$\{S, A\}$	$\{A, S\}$	{A,S}	
2	$\{S\}$	$\{A\}$	$\{S\}$	$\{A, S\}$	
3	$\{A\}$	$\{S\}$	$\{A, S\}$		
4	$\{A\}$	$\{A, S\}$			
5	$\{S\}$				

Cuadro 5.3: T después de hacer el paso 2, con $j = 4$

en $t_{3,3}, k = 1$, o en $t_{4,2}, k = 2$, o en $t_{5,1}$. El conjunto de no-terminales que podrían formar el primer no-terminal de la parte derecha es $\{A, S\}$, y el de no-terminales que podrían formar el segundo no-terminal de la parte derecha es $\{A, S\}$. Por lo tanto la parte derecha va a estar en $\{AA, AS, SA, SS\}$.

Después de hacer el paso 2, con $j = 4$ queda la tabla que se muestra en 5.3
Hacer el paso 2, con $j = 5$ en clase, y completar la tabla T con $t_{1,5}$.

Obtención de una derivación más a la izquierda

Una vez comprobado que la cadena w pertenece a $L(G)$, ¿cómo podemos obtener una secuencia de derivaciones que nos digan como producir la sentencia a partir de las producciones de P? Para ello está el algoritmo 5.2.

Algoritmo 5.2

Derivación más a la izquierda a partir de la tabla T de parsing.
Entrada: una gramática $G = (V_N, V_T, P, S)$ en formato CNF, y en la que las producciones de P están numeradas de 1 a n, una cadena de entrada $w = a_1 a_2 \cdots a_n$, y la tabla T generada por el algoritmo CYK.
Salida: una derivación izquierda de w o un error.
Método: se va a basar en el uso de una rutina recursiva $gen(i, j, A)$ que va a generar la derivación $A \Rightarrow^+ a_i a_{i+1} \cdots a_{i+j-1}$.

Se comienza ejecutando $gen(1, n, S)$ si $S \in t_{1n}$. En otro caso, se da salida de error.

$gen(i, j, A)$ se define como sigue:

1. Si $j = 1$ y la producción m-ésima es $A \to a_i$, entonces la salida de $gen(i, 1, A)$ es m.
2. Si $j > 1$, sea k el entero más pequeño, $1 \leq k < j$, tal que para algún $B \in t_{i,k}$ y $C \in t_{i+k,j-k}$ se tiene que $A \to BC \in P$. Si hay varias, elegimos la que tenga el índice más pequeño, digamos m. La salida de $gen(i, j, A)$ es m, más las salidas de $gen(i, k, B)$ y $gen(i + k, j - k, C)$ (en este orden).

Un ejemplo

Tomemos la gramática del ejemplo anterior y dispongámosla en el orden siguiente:

$(1) S \to AA$
$(2) S \to AS$

$(3) S \rightarrow b$
$(4) A \rightarrow SA$
$(5) A \rightarrow AS$
$(6) A \rightarrow a$

Sea la cadena de entrada $w = abaab$, la misma que en el ejemplo del cálculo de T. Por lo tanto esa tabla nos va a servir. Para obtener una derivación más a la izquierda, llamamos a $gen(1, 5, S)$, habiendo comprobado que $S \in t_{1,5}$ y que por lo tanto, $w \in L(G)$. Ahora la evolución, con el k y m correspondientes puede verse en la figura 5.4.

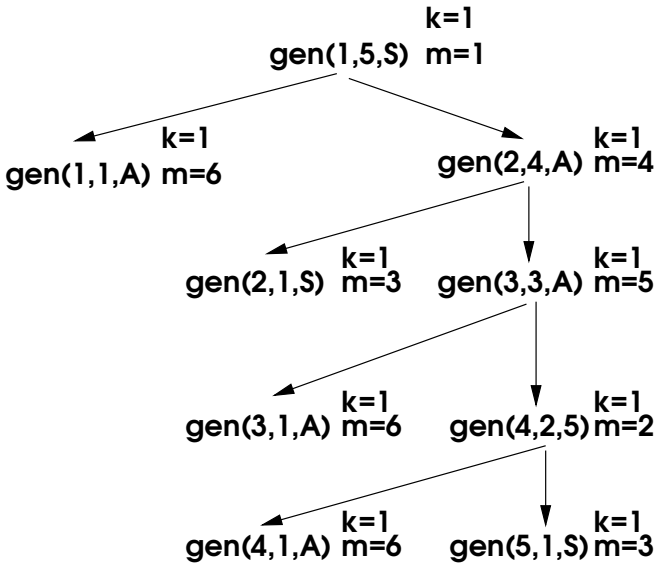


Figura 5.4: Ejemplo de generación de una derivación izquierda

En esa figura puede verse que la secuencia de derivaciones obtenida es la siguiente $S \Rightarrow^1 AA \Rightarrow^6 aA \Rightarrow^4 aSA \Rightarrow^3 abA \Rightarrow^5 abAS \Rightarrow^6 abaS \Rightarrow^2 abaAS \Rightarrow^6 abaaS \Rightarrow^3 abaab$.

5. Análisis sintáctico determinista

5.1. Introducción

A partir de ahora vamos a estudiar un conjunto especial de gramáticas libres de contexto a las cuales se les puede aplicar un análisis con complejidad espacial y temporal $c_1 n$ y $c_2 n$, respectivamente.

Para conseguir estos términos de eficiencia, un gran número de gramáticas han de quedarse en el camino, ante la imposibilidad de aplicarles el tipo de análisis que vamos a ver; sin embargo esto no resulta una restricción muy importante si nos ceñimos a los lenguajes de programación.

Los algoritmos de análisis que vamos a estudiar se caracterizan por ser completamente deterministas. Esto quiere decir que únicamente es necesaria una pasada, de izquierda a derecha, a través de la cadena de entrada w, para encontrar una árbol de derivación que represente su análisis. Con las gramáticas que nos ocupan, (i.e. de tipo $LL(k)$, concretamente las $LL(1)$), va a ser suficiente mirar el siguiente token en la cadena de entrada para determinar la regla de

producción a aplicar en la construcción del árbol de derivación. Por esto, este tipo de análisis también se denomina de una pasada. En él se incluyen las gramáticas:

- **Tipo LL(k)** Aquellas para las que el algoritmo de análisis descendente puede trabajar determinísticamente si se le permite mirar hasta k símbolos por delante de la posición de entrada actual.
- **Tipo LR(k)** Aquellas para las que el algoritmo de análisis ascendente puede trabajar determinísticamente si se le permite mirar hasta k símbolos por delante de la posición de entrada actual.
- **Gramáticas de Precedencia** Aquellas para las que el algoritmo de análisis ascendente puede encontrar la siguiente producción a aplicar en una forma sentencial derecha observando ciertas relaciones entre pares de símbolos adyacentes de esa forma sentencial.

5.2. Análisis LL (recursivo y no-recursivo)

5.2.1. *Gramáticas* $LL(1)$

Para introducir formalmente el concepto de gramática $LL(1)$ primero necesitamos definir el concepto de $FIRST_k(\alpha)$.

Definición 5.4 *Sea una CFG $G = (V_N, V_T, S, P)$. Se define el conjunto*

$$FIRST_k(\alpha) = \{x | \alpha \Rightarrow^*_{lm} x\beta \text{ y } |x| = k \text{ o bien } \alpha \stackrel{*}{\Rightarrow} x \text{ y } |x| < k\}$$

en donde $k \in N$ y $\alpha \in (V_N \cup V_T)^*$.

O, lo que es lo mismo, $FIRST_k(\alpha)$ consiste en todos los prefijos terminales de longitud k (o menores si α deriva una cadena de terminales de longitud menor que k) de las cadenas terminales que se pueden derivar de α.

Ahora podemos definir el concepto de gramática $LL(k)$.

Definición 5.5 *Sea una CFG $G = (V_N, V_T, S, P)$. Decimos que G es $LL(k)$ para algún entero fijo k, cuando siempre que existen dos derivaciones más a la izquierda*

1. $S \Rightarrow^*_{lm} wA\alpha \Rightarrow_{lm} w\beta\alpha \stackrel{*}{\Rightarrow} wx$, *y*
2. $S \Rightarrow^*_{lm} wA\alpha \Rightarrow_{lm} w\gamma\alpha \stackrel{*}{\Rightarrow} wy$

tales que $FIRST_k(x) = FIRST_k(y)$, entonces se tiene que $\beta = \gamma$.

Si lo decimos de una manera informal, G es $LL(k)$ si dada una cadena $wA\alpha \in (V_N \cup V_T)^*$ y los primeros k símbolos que se van a derivar a partir de $A\alpha$, existe a lo sumo una producción que se pueda aplicar a A y que lleve a la derivación de cualquier cadena de terminales que comience con w seguida de esos k símbolos.

Lo vemos con un ejemplo. Sea G_1 la gramática con conjunto $P = \{S \to aAS|b, A \to a|bSA\}$. Vamos a ver que esta gramática es $LL(1)$. Entonces, si

$$S \Rightarrow^*_{lm} wS\alpha \Rightarrow_{lm} w\beta\alpha \Rightarrow^*_{lm} wx$$

y

$$S \Rightarrow^*_{lm} wS\alpha \Rightarrow_{lm} w\gamma\alpha \Rightarrow^*_{lm} wy$$

Si x e y comienzan con el mismo símbolo, se tiene que dar $\beta = \gamma$. Por casos, si $x = y = a$, entonces se ha usado la producción $S \to aAS$. Como únicamente se ha usado una producción, entonces $\beta = \gamma = aAS$. Si $x = y = b$, se ha usado $S \to b$, y entonces $\beta = \gamma = b$.
Si se consideran las derivaciones

$$S \Rightarrow^*_{lm} wA\alpha \Rightarrow_{lm} w\beta\alpha \Rightarrow^*_{lm} wx$$

y

$$S \Rightarrow^*_{lm} wA\alpha \Rightarrow_{lm} w\gamma\alpha \Rightarrow^*_{lm} wy$$

se produce el mismo razonamiento. Sin embargo, determinar si un lenguaje es $LL(1)$ es un problema indecidible.

Definición 5.6 *Sea una CFG $G = (V_N, V_T, S, P)$. Decimos que G es $LL(1)$ cuando siempre que existen dos derivaciones más a la izquierda*

1. $S \Rightarrow^*_{lm} wA\alpha \Rightarrow_{lm} w\beta\alpha \stackrel{*}{\Rightarrow} wx\ y$
2. $S \Rightarrow^*_{lm} wA\alpha \Rightarrow_{lm} w\gamma\alpha \stackrel{*}{\Rightarrow} wy$

tales que $FIRST_1(x) = FIRST_1(y)$, entonces se tiene que $\beta = \gamma$.

De manera informal, G es $LL(1)$ si dada una cadena $wA\alpha \in (V_N \cup V_T)^*$ y un símbolo terminal $b \in FIRST_1(A\alpha)$, existe a lo sumo una producción aplicable a A que conduzca a la derivación de la cadena $wb\beta$, para algún $\beta \in (V_N \cup V_T)^*$.

Entonces, para poder construir un analizador sintáctico predictivo, con $k = 1$, se debe conocer, dado el símbolo de entrada actual a_i y el no terminal A a expandir, cuál de las alternativas de la producción $A \to \alpha_1 | \cdots | \alpha_n$ es la única que va a dar lugar a una subcadena que comience con a_i. Piénsese, por ejemplo, en el conjunto de producciones siguiente:

```
prop  →  if expr then prop else prop
      |  while expr do prop
      |  begin lista_props end
```

Las palabras clave `if`, `while` y `begin` indican la alternativa única con posibilidad de éxito para encontrar una proposición.

Si se tiene cuidado al escribir la gramática, eliminando la ambigüedad, la recursión por la izquierda, y factorizándola por la izquierda, es posible, aunque no seguro, que se obtenga una gramática LL(1).

5.2.2. *Construcción de los Conjuntos $FIRST$ y $FOLLOW$*

Estos conjuntos van a servir de apoyo para la construcción del analizador sintáctico descendente predictivo. Como veremos, son necesarios para completar, posteriormente, la tabla que va a guiar el análisis. Esta tabla indicará, para un símbolo de entrada, y un no-terminal a reducir, la alternativa derecha que se ha de aplicar.

Sea α una forma sentencial de una gramática determinada. Pues bien, se considera el conjunto $FIRST(\alpha)$ como el conjunto de terminales que inician las cadenas derivadas de α. Por supuesto, si $\alpha \stackrel{*}{\Rightarrow} \lambda$, entonces $\lambda \in FIRST(\alpha)$. Este conjunto ya se definió formalmente para gramáticas de tipo $LL(k)$.

Ahora, vamos a introducir el conjunto $FOLLOW_k(\beta)$ formalmente, y luego lo particularizaremos para las gramáticas $LL(1)$.

Definición 5.7 *Sea $G = (V_N, V_T, S, P)$ una gramática CFG. Definimos $FOLLOW_k^G(\beta)$, en donde k es un entero, $\beta \in (V_N \cup V_T)^*$, como el conjunto*

$$\{w | S \stackrel{*}{\Rightarrow} \alpha\beta\gamma \text{ junto con } w \in FIRST_k^G(\gamma)\}$$

Dicho de otro modo, y particularizandolo para $FOLLOW_1 \equiv FOLLOW$, sea A un no terminal de una gramática determinada. Definimos $FOLLOW(A)$ como el conjunto de terminales a que pueden aparecer inmediatamente a la derecha de A en alguna forma sentencial de la gramática. Es decir, el conjunto de terminales a tal que haya una derivación de la forma $S \stackrel{*}{\Rightarrow} \alpha A a \beta$, para algún α y β. Si A es el símbolo más a la derecha en determinada forma sentencial de la gramática, entonces el símbolo $\$ \in FOLLOW(A)$.

Algoritmo para el cálculo del conjunto $FIRST$

Algoritmo 5.3

Cálculo del conjunto $FIRST$ para todos los símbolos no terminales y terminales de la gramática de entrada.

- **Entrada:** Una gramática $G = (V_N, V_T, S, P)$ de tipo CFG.
- **Salida:** Los conjuntos $FIRST(X)$ para todo $X \in (V_N \cup V_T)$.
- **Método:** Ejecutar el siguiente método para todo $X \in (V_N \cup V_T)$.
  1. Si $X \in V_T$, entonces $FIRST(X) = \{X\}$.
  2. Si no, si $X \in V_N$ y $X \to \lambda \in P$, entonces añadir λ a $FIRST(X)$.
  3. Si no, si $X \in V_N$ y $X \to Y_1 Y_2 \cdots Y_k \in P$ añadir todo $a \in V_T$ tal que para algún i, con $1 \leq i \leq k$, $\lambda \in FIRST(Y_1), \lambda \in FIRST(Y_2), \ldots, \lambda \in FIRST(Y_{i-1})$, o lo que es lo mismo, $Y_1 Y_2 \ldots Y_{i-1} \stackrel{*}{\Rightarrow} \lambda$ y $a \in FIRST(Y_i)$. Además, si $\lambda \in FIRST(Y_j)$ para todo $j = 1, 2, \ldots, k$, añadir λ a $FIRST(X)$.

Observar que se puede calcular $FIRST$ para cualquier cadena $X_1 X_2 \cdots X_n$, añadiendo todo símbolo $Y \in FIRST(X_1)$ con $Y \neq \lambda$ a $FIRST(X_1 X_2 \cdots X_n)$. Además, si $\lambda \in FIRST(X_1)$ añadir también $Y \in FIRST(X_2)$ con $Y \neq \lambda$ y así sucesivamente. Se añadirá λ si esta estaba en todos los conjuntos $FIRST(X_i)$, $i = 1, \ldots, n$.

Algoritmo para el cálculo del conjunto $FOLLOW$

Algoritmo 5.4

Cálculo del conjunto $FOLLOW$ para todos los símbolos no terminales de la gramática de entrada.

- **Entrada:** Una gramática $G = (V_N, V_T, S, P)$ de tipo CFG.
- **Salida:** Los conjuntos $FOLLOW(X)$ para todo $X \in V_N$.
- **Método:** Ejecutar el siguiente método para todo $X \in V_N$ hasta que no se pueda añadir nada más a ningún conjunto FOLLOW.

1. Añadir \$ a $FOLLOW(S)$, en donde \$ es el delimitador derecho de la entrada.
2. Si existe una producción $A \rightarrow \alpha B\beta \in P$ añadir todo $FIRST(\beta) - \{\lambda\}$ a $FOLLOW(B)$.
3. Si existen una producción $A \rightarrow \alpha B \in P$, ó $A \rightarrow \alpha B\beta \in P$ tal que $\lambda \in FIRST(\beta)$, entonces añadir $FOLLOW(A)$ a $FOLLOW(B)$.

5.2.3. *Ejemplo de construcción de $FIRST$ y $FOLLOW$.*

Vamos a plantear un ejemplo de construcción de los conjuntos $FIRST$ y $FOLLOW$, con la siguiente gramática:

Gramática 5.1

$$\begin{array}{lcl} E & \rightarrow & TE' \\ E' & \rightarrow & +TE'|\lambda \\ T & \rightarrow & FT' \\ T' & \rightarrow & *FT'|\lambda \\ F & \rightarrow & (E)|id \end{array}$$

- Los conjuntos $FIRST$ para todos los símbolos terminales de $V_T = \{(,), +, *\}$ son ellos mismos.
- Para el no terminal F, aplicando el paso 3 introducimos al conjunto $FIRST$ los símbolos (y id.
- Para el no terminal T', aplicando el paso 2 introducimos a $FIRST$ λ, y por el paso 3, el símbolo λ.
- Para el no terminal T, por el paso tres, con la regla de producción $T \rightarrow FT'$, añadimos $FIRST(F)$ a $FIRST(T)$.
- Para E', con el paso 2 se añade λ y con el tres se añade $+$.
- Para E, $FIRST(E)$ queda con el contenido $\{(, id\}$ al darse la producción $E \rightarrow TE'$, aplicando el paso 3.

Los conjuntos $FIRST$ quedan como sigue:

$$\begin{array}{l} FIRST(F) = \{(, id\} \\ FIRST(T') = \{*, \lambda\} \\ FIRST(T) = \{(, id\} \\ FIRST(E') = \{+, \lambda\} \\ FIRST(E) = \{(, id\} \end{array}$$

Pasamos ahora a calcular los conjuntos $FOLLOW$.

- Para el símbolo E, el conjunto $FOLLOW(E) = \{\$,)\}$, añadiendo el \$ por el paso 1, y el paréntesis derecho por el paso 2 y la producción $F \rightarrow (E)$.
- Al conjunto $FOLLOW(E')$ añadimos el contenido de $FOLLOW(E)$ por el paso 3, y la producción $E \rightarrow TE'$.

- Al conjunto $FOLLOW(T)$ se añade + por el paso 2 y la producción $E \rightarrow TE'$. Además, como $E' \rightarrow \lambda \in P$, añadimos el contenido de $FOLLOW(E')$.
- Como tenemos que $T \rightarrow FT' \in P$, añadimos $FOLLOW(T)$ a $FOLLOW(T')$.
- Por el paso 2, y las producciones $T \rightarrow FT'$ y $T' \rightarrow *FT'$ añadimos el contenido de $FIRST(T') - \lambda$ a $FOLLOW(F)$. Además, como $T' \rightarrow \lambda$ añadimos $FOLLOW(T')$.

Y obtenemos los conjuntos $FOLLOW$ siguientes:

$FOLLOW(E) = \{\$,)\}$
$FOLLOW(E') = \{\$,)\}$
$FOLLOW(T) = \{+, \$,)\}$
$FOLLOW(T') = \{+, \$,)\}$
$FOLLOW(F) = \{*, +, \$,)\}$

5.2.4. *Construcción de la tabla de análisis sintáctico*

Ahora ya tenemos todo lo necesario para construir una tabla de análisis sintáctico que dos diga en todo momento las posibles producciones a aplicar, dado un no-terminal a reducir y un símbolo de la entrada a_i. Esta tabla de análisis va a venir definida, algebraicamente, como:

$$M : V_N \times V_T \cup \{\$\} \rightarrow 2^P$$

El contenido de la tabla se obtiene con el algoritmo que aparece a continuación.

Algoritmo 5.5

Construcción de una tabla de análisis sintáctico predictivo.

- **Entrada:** Una gramática $G = (V_N, V_T, S, P)$, CFG.
- **Salida:** La tabla de análisis sintáctico M.
- **Método:**
  1. Créese una tabla $M_{|V_N| \times (|V_T|+1)}$, con una fila para cada no-terminal y una columna para cada terminal más el \$.
  2. Para cada $A \rightarrow \alpha \in P$, ejecutar los pasos 3 y 4.
  3. Para cada $a \in FIRST(\alpha)$, añadir $A \rightarrow \alpha$ a $M[A, a]$.
  4. Si $\lambda \in FIRST(\alpha)$, añadir $A \rightarrow \alpha$ a $M[A, b]$, para cada terminal $b \in FOLLOW(A)$. Si además, $\$ \in FOLLOW(A)$, añadir $A \rightarrow \alpha$ a $M[A, \$]$.
  5. Introducir, en cada entrada de M vacía un identificador de error.

La tabla nos va a indicar la producción que debe usarse en una paso de derivación en el que tiene que expandirse el símbolo no terminal A, y el token de entrada actual es a. Una observación importante es que si alguna casilla de M contiene más de una producción de P, la gramática no es $LL(1)$, ya que no es suficiente con observar el siguiente token para decidir qué producción coger, al encontrar más de una. Esta condición podemos expresarla algebraicamente, usando una función $Predict$, la cual, aplicada a una producción de la gramática, nos dirá el conjunto de terminales que predicen su uso.

$$\begin{aligned} Predict(A \rightarrow \alpha) \quad = \quad & \text{if } \lambda \in FIRST(\alpha) \\ & \text{then } (FIRST(\alpha) - \{\lambda\} \cup FOLLOW(A)) \\ & \text{else } FIRST(\alpha) \end{aligned}$$

	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \lambda$	$E' \to \lambda$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \lambda$	$T' \to *FT'$		$T' \to \lambda$	$T' \to \lambda$
F	$F \to id$			$F \to (E)$		

Cuadro 5.4: Tabla de análisis sintáctico para la gramática del ejemplo.

Por lo tanto, cada casilla de la tabla podría formarse a partir de esta función, tal que

$$M[A, a] = \{A \to \alpha / a \in Predict(A \to \alpha)\} \forall A \in V_N, a \in V_T\}$$

Teorema **5.1** *Una GLC, $G = (V_N, V_T, S, P)$ es de tipo $LL(1)$ si, y solo si, en caso de que existan producciones $A \to \alpha$ y $A \to \gamma$, entonces $Predict(A \to \alpha) \cap Predict(A \to \gamma) = \emptyset$.*

Para la gramática 5.1, la tabla de análisis sintáctico predictivo queda como se ve en la tabla 5.4.

Ahora vamos a ver un ejemplo, en el que alguna de las celdas de la tabla M contiene más de una producción.

prop → **if** exp **then** prop
| **if** exp **then** prop **else** prop
| a
| b
exp → p
| q

Si eliminamos la ambigüedad, como ya habíamos visto en otro tema, la gramática queda:

prop → prop1
| prop2
prop1 → **if** exp **then** prop1 **else** prop1
| a
| b
prop2 → **if** exp **then** prop
| **if** exp **then** prop1 **else** prop2
exp → p
| q

Si factorizamos la gramática por la izquierda, tenemos

prop → prop1
| prop2
prop1 → **if** exp **then** prop1 **else** prop1
| a
| b
prop2 → **if** exp **then** prop2'
prop2' → prop
| prop1 **else** prop2
exp → p
| q

El alumno debería comprobar que se obtiene la tabla de análisis siguiente:

	if	**then**	**else**	a	b	p	q	$
p	$p \rightarrow p_1 \vert p_2$			$p \rightarrow p_1$				
p_1	$p_1 \rightarrow$ **if** exp **then** p_1 **else** p_1			$p_1 \rightarrow a$	$p_1 \rightarrow b$			
p_2	$p_2 \rightarrow$ **if** exp **then** p_2'							
p_2'	$p_2' \rightarrow p$ $p_2' \rightarrow p_1$ **else** p_2			$p_2' \rightarrow p$ $p_2' \rightarrow p_1$ **else** p_2	$p_2' \rightarrow p$ $p_2' \rightarrow p_1$ **else** p_2			
exp						$exp \rightarrow p$	$exp \rightarrow q$	

Como se puede ver, no es $LL(1)$. Comprobar que modificando el lenguaje, añadiendo delimitadores de bloque (e.g. `endif`) la gramática producida es $LL(1)$.

Una manera *ad-hoc* de solucionar el problema es adoptando la convención de determinar, de antemano, la producción a elegir de entre las disponibles en una celda determinada de M. Si en el ejemplo de la gramática anterior, factorizamos la gramática original, sin eliminar la ambigüedad tenemos:

prop → **if** exp **then** prop prop'
| a
| b
prop' → **else** prop
| λ
exp → p
| q

Si construimos la tabla de análisis para esta gramática, nos queda:

	if	**then**	**else**	a	b	p	q	$
p	$p \rightarrow if\ exp\ then\ p\ p'$			$p \rightarrow a$	$p \rightarrow b$			
p'			$p' \rightarrow else\ p$ $p' \rightarrow \lambda$					$p' \rightarrow \lambda$
exp						$exp \rightarrow p$	$exp \rightarrow q$	

Se observa que en $M[p', else]$ hay dos producciones. Si, por convenio, determinamos elegir siempre $p' \rightarrow else\ p$, lo que estamos haciendo el escoger el árbol de derivación que asociaba el $else$ con el if más próximo.

En cualquier caso, no existe un criterio general para elegir una sola regla de producción cuando hay varias en una misma casilla.

5.2.5. *Análisis Descendente Predictivo No Recursivo*

Para el diseño de un analizador sintáctico, descendente y no recursivo es claro que necesitamos una estructura de pila. Además vamos a usar la tabla que se ha estudiado anteriormente, para determinar qué producción aplicar en cada momento, junto con la cadena de entrada para el análisis. El modelo de parser de este tipo aparece en la figura 5.5.

Como se ve en la figura, el analizador usa un buffer de entrada, una pila, una tabla de análisis sintáctico y genera una cadena de salida. El final del buffer de entrada está delimitado con el signo $, así como el fondo de la pila. Esta podrá albergar tanto símbolos terminales como no-terminales. Y estará vacía cuando el elemento que aparezca en la cabeza de la misma sea $.

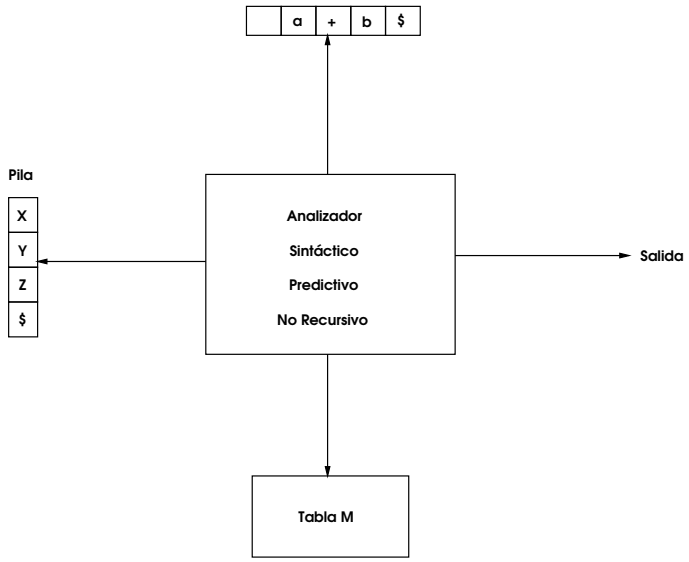


Figura 5.5: Modelo de analizador sintáctico predictivo, no recursivo

El control de la manipulación de todos esos elementos se describe fácilmente. Siempre se tiene en cuenta la cabeza de la pila, en la figura 5.5 el símbolo X, y el siguiente carácter a la entrada, llamémosle a, el símbolo + en la figura 5.5. Dependiendo de si X es no-terminal ó terminal tendremos:

- Si $X = a = \$$ el análisis finaliza con éxito.
- Si $a \in V_T$ y $X = a$, el analizador sintáctico saca X de la pila, y desplaza el apuntador de la entrada un lugar a la derecha. No hay mensaje de salida.
- Si $X \in V_N$, es hora de usar M. Para ello, el control del análisis consulta la entrada $M[X, a]$.
  - Si $M[X, a] = \{X \to UVW\}$, por ejemplo, se realiza una operación pop, con lo que sacamos X de la cima, y una operación $push(UVW)$, estando U en la cima. La salida, tras esa operación, es precisamente la producción utilizada, $X \to UVW$.
  - Si $M[X, a] = \emptyset$, el análisis es incorrecto, y la cadena de entrada no pertenece al lenguaje generado por la gramática. La salida es **error**. Posiblemente se llame a una rutina de recuperación de errores.

Pasamos ahora a especificar el algoritmo formalmente.

Algoritmo 5.6

Análisis Sintáctico Predictivo No Recursivo.

- **Entrada:** Una tabla de análisis sintáctico M para una gramática $G = (V_N, V_T, S, P)$, CFG y una cadena de entrada w.
- **Salida:** Si $w \in L(G)$, una derivación por la izquierda de w; si no una indicación de error.
- **Método:** Sea la configuración inicial de la pila, $\$S$. Sea $w\$$ el buffer de entrada.
  - Hacer que ap(apuntador) apunte al primer símbolo de $w\$$.
  - Repetir
    - Sea X el símbolo a la cabeza de la pila, y a el símbolo apuntado por ap.

- Si $X \in V_T$ o $X = \$$ Entonces
  - Si $X = a$ Entonces extraer X de la pila y avanzar *ap*.
  - Si no error();

  Si No
  - Si $M[X, a] = X \to Y_1 Y_2 \cdots Y_k$ entonces
  - Begin
    1. Extraer X de la pila
    2. Meter $Y_k Y_{k-1} \cdots Y_1$ en la pila, con Y_1 en la cima
    3. Emitir a la salida la producción $X \to Y_1 Y_2 \cdots Y_k$
  - End
  - Si no error()
- Hasta que $(X = \$)$.

Para hacer un seguimiento de las sucesivas configuraciones que va adquiriendo el algoritmo, se usa una tabla de tres columnas: en la primera se muestra, para cada movimiento el contenido de la pila, en la segunda la entrada que aun queda por analizar, y en la tercera la salida que va emitiendo el algoritmo. Veamos un ejemplo, con la gramática 5.1 y la correspondiente tabla 5.4. La evolución es la que aparece en la tabla 5.5:

Pila	Entrada	Salida
$\$E$	$id + id * id\$$	
$\$E'T$	$id + id * id\$$	$E \to TE'$
$\$E'T'F$	$id + id * id\$$	$T \to FT'$
$\$E'T'id$	$id + id * id\$$	$F \to id$
$\$E'T'$	$+id * id\$$	
$\$E'$	$+id * id\$$	$T' \to \lambda$
$\$E'T+$	$+id * id\$$	$E' \to +TE'$
$\$E'T$	$id * id\$$	
$\$E'T'F$	$id * id\$$	$T \to FT'$
$\$E'T'id$	$id * id\$$	$F \to id$
$\$E'T'$	$*id\$$	
$\$E'T'F*$	$*id\$$	$T' \to *FT'$
$\$E'T'F$	$id\$$	
$\$E'T'id$	$id\$$	$F \to id$
$\$E'T'$	$\$$	
$\$E'$	$\$$	$T' \to \lambda$
$\$$	$\$$	$E' \to \lambda$

Cuadro 5.5: Evolución de la pila para la gramática 5.1 con la palabra $id + id * id\$$

5.2.6. *Recuperación de Errores en el análisis descendente predictivo*

En el contexto del análisis sintáctico predictivo, los errores pueden darse por dos situaciones bien diferentes:

- Cuando el terminal de la cabeza de la pila no concuerda con el siguiente terminal a la entrada.

- Cuando se tiene un no-terminal A en la cima de la pila, y un símbolo a a la entrada, y el contenido de $M[A, a] = \emptyset$.

Recuperación en Modo Pánico

Como ya sabemos, la recuperación de errores en modo pánico consiste, *grosso modo*, en que cuando se detecta un token no esperado, se siguen consumiendo tokens, procedentes del análisis léxico, hasta que llega un determinado token denominado **de sincronización**. Los tokens de sincronización forman un conjunto que debe ser elegido cuidadosamente pues la eficiencia del manejo de errores en modo pánico va a depender de cómo de bien se elijan esos tokens. Además se deberá prestar más atención a aquellos errores que ocurren con más frecuencia en la práctica.

Las siguientes son algunas heurísticas que nos van a ayudar a decidir cuales van a ser los tokens de sincronización para nuestra gramática:

a) Para cada símbolo $A \in V_N$, los tokens de sincronización podrían ser aquellos pertenecientes a $FOLLOW(A)$. Con esto estamos atacando aquellos errores que se cometen en la porción de forma sentencial producida por ese A. Así, cuando la cabeza de la pila es A y $M[A, a] = \emptyset$, podemos extraer A de la cima de la pila una vez que hayamos encontrado un token perteneciente a este conjunto de sincronización. A partir de ahí se continúa el análisis.

b) Sin embargo ese conjunto de sincronización resulta insuficiente. Piénsese en un lenguaje cuyas sentencias terminen por el carácter ';', por ejemplo. Si el error ha sido omitir ese carácter, a continuación, seguramente, encontraremos una palabra clave que inicia una sentencia. Esta palabra clave no pertenecerá a $FOLLOW(A)$, obviamente. Por lo tanto toda la sentencia siguiente quedará invalidada. Una solución para eso sería incluir las palabras claves en el conjunto de sincronización para A. Esto nos lleva a un esquema más general. Si el lenguaje está formado por construcciones organizadas en una forma jerárquica, e.g. en donde los bloques contienen otros bloques y sentencias, y a su vez estas contienen expresiones, ... una buena aproximación es incluir en los conjuntos de sincronización de no-terminales inferiores, los terminales que inician las construcciones superiores.

c) Si consideramos un tipo de error muy común, que consiste en colocar caracteres extraños, con estructura de token (e.g. identificador), en una sentencia, es claro que la estructura de frase se ve alterada. Podemos evitar ese tipo de tokens incluyendo, en el conjunto de sincronización de los correspondientes A, el contenido de $FIRST(A)$. En este caso se continuaría el análisis al encontrar el token de sincronización, sin sacar A de la pila.

d) Una solución, poco elegante, aunque definitiva podría ser esta: si no se puede emparejar un terminal en la cabeza de la pila, extraerlo de la pila, emitir un mensaje que indique que se insertó un terminal en la entrada, y continuar el análisis (i.e. es equivalente a considerar como componentes de sincronización el resto de componentes léxicos).

En resumen, y como *estrategia general*, podemos actuar de la siguiente forma:

- Si el *teminal de la pila no coincide con el de la entrada*, se actúa como en el caso d) anterior.
- Si $M[A, a] = \emptyset$ se saltan tokens de la entrada hasta encontrar un token de sincronización que cumpla una de estas condiciones y en este orden:
  - Que pertenezca al conjunto $FIRST(A)$ y se actúa como en el caso c).
  - Que pertenezca al conjunto $FOLLOW(A)$ y se actúa como en a), o que pertenezca al conjunto de sincronización definido según b) ajustando la pila de forma adecuada.

Lo vemos mejor con un ejemplo.

Observemos la tabla 5.4 y supongamos que estamos utilizando los tokens de sincronización de los conjuntos $FIRST(A)$ y $FOLLOW(A)$ para cada no terminal A.

Según la estrategia general, el algoritmo se comportaría de la siguiente forma, para la entrada $+id*id+$:

Pila	Entrada	Comentario
$\$E$	$+id*id+\$$	**Error**: ignorar + al no ser t.sincr. Como $id \in FIRST(E)$, continuar
$\$E$	$id*id+\$$	
$\$E'T$	$id*id+\$$	
$\$E'T'F$	$id*id+\$$	
$\$E'T'id$	$id*id+\$$	
$\$E'T'$	$*id+\$$	
$\$E'T'F*$	$*id+\$$	
$\$E'T'F$	$id+\$$	
$\$E'T'id$	$id+\$$	
$\$E'T'$	$+\$$	
$\$E'$	$+\$$	
$\$E'T+$	$+\$$	
$\$E'T$	$\$$	**Error**: se extrae T de la pila, pues $\$ \in FOLLOW(T)$
$\$E'$	$\$$	
$\$$	$\$$	

Observar que hay una primera secuencia de derivaciones más a la izquierda, antes de la detección del segundo error:

$$E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow idT'E' \Rightarrow id*FT'E' \Rightarrow id*idT'E' \Rightarrow id*idE' \Rightarrow id*id+TE'$$

A partir de ahí, no podríamos seguir generando la cadena. Si eliminamos T de la cima de la pila podemos continuar aplicando la regla $E' \Rightarrow \lambda$ obteniendo la última derivación:

$$id*id+E' \Rightarrow id*id+$$

Con lo que, al final somos capaces de *simular* la producción de la cadena errónea.

Otro ejemplo puede ser el de la entrada $(id\$$, para la gramática del ejemplo previo. La evolución del algoritmo será:

Pila	Entrada	Acción
$\$E$	$(id\$$	
$\$E'T$	$(id\$$	$E \rightarrow TE'$
$\$E'T'F$	$(id\$$	$T \rightarrow FT'$
$\$E'T')E(F$	$(id\$$	$F \rightarrow (E)$
$\$E'T')E$	$id\$$	
$\$E'T')E'T$	$id\$$	$E \rightarrow TE'$
$\$E'T')E'T'F$	$id\$$	$T \rightarrow FT'$
$\$E'T')E'T'id$	$id\$$	$F \rightarrow id$
$\$E'T')E'T'$	$\$$	
$\$E'T')E'$	$\$$	$T' \rightarrow \lambda$
$\$E'T')$	$\$$	$E' \rightarrow \lambda$
$\$E'T'$	$\$$	Error. Sacamos ')' de la pila.
$\$E'$	$\$$	$T' \rightarrow \lambda$
$\$$	$\$$	$E' \rightarrow \lambda$
$\$$	$\$$	

Lo que se ha interpretado, con este error, es que se había omitido, por equivocación el paréntesis derecho. Esa interpretación va a dar la derivación izquierda siguiente:

$$E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow (E)T'E' \Rightarrow (TE')T'E' \Rightarrow (FT'E')T'E' \Rightarrow (idT'E')T'E'$$
$$\Rightarrow (idE')T'E' \Rightarrow (id)T'E' \Rightarrow (id)E' \Rightarrow (id)$$

Recuperación a Nivel de Frase
El esquema general de tratamiento de errores con esta técnica consiste en introducir apuntadores a rutinas de error en las casillas en blanco de la tabla M. Dependiendo de cual sea la casilla de error, la rutina de tratamiento ejecutará un tipo de operación u otro. Las operaciones habituales son las de cambiar, eliminar ó añadir caracteres a la entrada emitiendo los pertinentes mensajes de error. Este enfoque puede resultar bastante complicado, pues habría que considerar los posibles símbolos de entrada que pueden causar error, y luego dar un mensaje además de un tratamiento adecuado para cada tipo de error.

Si consideramos de nuevo la tabla 5.4, y suponemos que en la pila aparece E y en la entrada), esto puede deberse a dos situaciones diferentes, de las que debería informarnos e intentar recuperarse una rutina a la que se llamara cuando se intentara acceder a la casilla correspondiente en la tabla. Los mensajes y actuaciones correspondientes a cada una de estas situaciones podrían ser:

- "Se colocó) al principio del programa". Saltar) de la entrada.
- "Falta expresión entre paréntesis". Sacar de la pila E) y eliminar) de la entrada.

5.2.7. *Análisis Descendente Predictivo Recursivo*

Este método descendente de análisis se basa en la ejecución, en forma recursiva, de un conjunto de procedimientos que se encargan de procesar la entrada. Se asocia un procedimiento a cada no-terminal de la gramática, con lo que se tiene que codificar cada uno de ellos según sus características. Estas van a estar condicionadas por el hecho de usar el tipo de análisis predictivo, y para gramáticas de tipo $LL(1)$. Por lo tanto, los símbolos de los respectivos conjuntos $FIRST$ van a determinar, de forma no ambigua, el siguiente procedimiento que se deberá invocar. Precisamente esta secuencia es la que va a definir la derivación izquierda que se está aplicando de forma implícita.

Vamos a introducir este análisis usando como gramática de referencia una CFG, $G = (V_N, V_T, S, P)$ con el siguiente conjunto de producciones en P:

Gramática 5.2

```
  tipo  →  simple
        |  ↑ id
        |  array [simple] of tipo
simple  →  integer
        |  char
        |  num puntopunto num
```

Observar que la gramática 5.2 es de tipo $LL(1)$, ya que los respectivos conjuntos $FIRST(tipo)$ y $FIRST(simple)$ son disjuntos. Por lo tanto, el primer símbolo de entrada va a determinar qué producción aplicar para obtener toda la cadena. Con sus producciones se definen tipos

compuestos y tipos simples. Los compuestos son punteros a identificadores y arrays de tipos compuestos. Los simples son los enteros, caracteres simples y números reales.

Volviendo al análisis recursivo de la gramática 5.2 , vamos primero a introducir los tipos de procedimientos de los que se hablaba arriba. Vamos a tener dos procedimientos similares, uno para cada símbolo perteneciente a V_N. Cada uno de los procedimientos, correspondientes a los no terminales `tipo` y `simple`, junto con un procedimiento `empareja` para simplificar el código de los dos anteriores aparecen en la figura del pseudocódigo 5.1

Pseudo código

Pseudo código 5.1

```
procedure empareja(t:complex);
begin
    if (preanalisis == t) then
        preanalisis := sigcomplex
    else error
end;
procedure tipo;
begin
    if preanalisis is in {integer, char, num} then
        simple
    else if preanalisis == '↑' then begin
        empareja('↑'); empareja(id)
    end
    else if preanalisis == array then begin
        empareja(array); empareja('['); simple; empareja(']');
        empareja(of); tipo
    end
    else error
end;
procedure simple;
begin
    if preanalisis == integer then
        empareja(integer)
    else if preanalisis == char then
        empareja(char)
    else if preanalisis == num then begin
        empareja(num); empareja(puntopunto); empareja(numero);
    end
    else error
end;
```

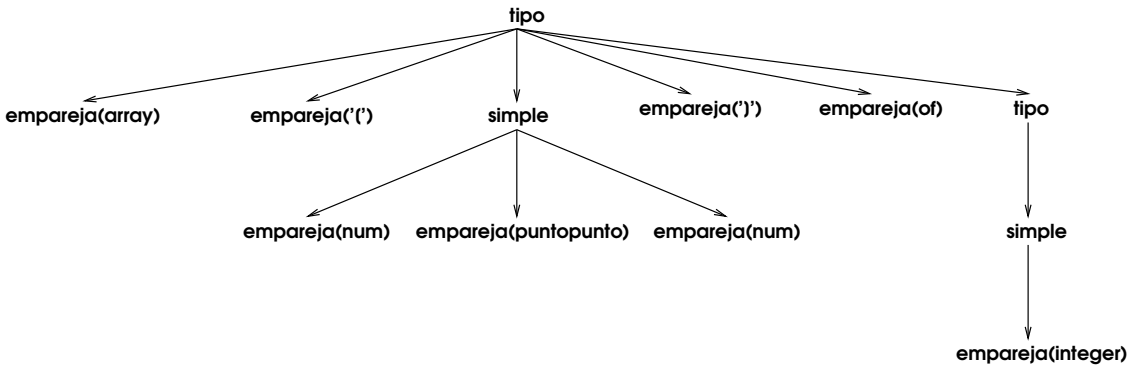


Figura 5.6: Ejemplo de árbol de llamadas del análisis descendente recursivo predictivo

Nótese que el análisis sintáctico debe comenzar con una llamada al no-terminal inicial, `tipo`. En este, se testea el contenido de la variable global `preanalisis` que contiene el carácter de anticipación de la cadena de entrada, que posibilita el análisis predictivo. Si tomamos como ejemplo la entrada que aparece en el pseudocódigo siguiente:

```
array [num puntopunto num] of integer;
```

el contenido de preanalisis es, inicialmente, **array**. Por lo tanto, se generan las llamadas

```
empareja(array); empareja('['); simple; empareja(']'); empareja(of); tipo
```

que precisamente corresponde con la producción

```
tipo  → array [simple] of tipo
```

de la gramática del ejemplo. Lo que hacemos es, simplemente, invocar al procedimiento `empareja` para cada símbolo terminal, y a los correspondientes `simple` y `tipo` para el tamaño y el tipo base del array, respectivamente. El orden de la invocación es importante, al estar realizando un análisis descendente y, por lo tanto, obteniendo una derivación más a la izquierda.

Observar que el símbolo de anticipación inicial (i.e. `array`) coincide con el argumento de `empareja(array)`. Por lo tanto, se actualiza la variable `preanalisis` al siguiente carácter a la entrada, que es '['. La llamada `empareja('[')` también la actualiza la variable `preanalisis` pasando a ser ahora `num`. Ahora se invoca a `simple`, que compara el contenido de esta variable con todos los símbolos terminales que forman su correspondiente conjunto $FIRST$. Coincide con `num` y por lo tanto se hace la siguiente serie de invocaciones:

```
empareja(num); empareja(puntopunto); empareja(num)
```

que resultan exitosas. Después de su ejecución, el contenido de `preanalisis` es `of`, y estamos en la llamada `empareja(of)`. Resulta exitosa y nuevamente se actualiza el contenido de preanalisis a `integer`. Se llama ahora a `tipo` que genera su correspondiente llamada `simple` según dicta el símbolo de preanalisis y el conjunto $FIRST(tipo)$. Finalmente se genera la llamada `empareja(integer)`, y como el siguiente símbolo es $, finaliza con éxito. La secuencia de llamadas puede seguirse con ayuda de la figura 5.6.

Otro ejemplo podemos verlo con la gramática 5.1, cuyas producciones volvemos a incluir a continuación:

$$
\begin{array}{lcl}
E & \rightarrow & TE' \\
E' & \rightarrow & +TE'|\lambda \\
T & \rightarrow & FT' \\
T' & \rightarrow & *FT'|\lambda \\
F & \rightarrow & (E)|id
\end{array}
$$

Vamos a escribir los procedimientos necesarios para el análisis recursivo descendente predicitivo, para esta gramática $LL(1)$. Como ya hemos mencionado, se debe escribir un procedimiento para cada símbolo no-terminal, que se encargue de analizar sus correspondientes partes derechas. El listado completo puede encontrarse en la figura de pseudocódigo 5.2.

Pseudo código

Pseudo código 5.2

```
procedure empareja(t:simbolo);
begin
    if (preanalisis == t) then
        preanalisis := sigsimbolo
    else error
end;
procedure No_terminal_E;
begin
    No_terminal_T; No_terminal_E'
end;
procedure No_terminal_E';
begin
    if preanalisis == '+' then
        empareja('+'); No_terminal_T; No_terminal_E'
    else
        begin
        end
end;
procedure No_terminal_T;
begin
    No_terminal_F; No_terminal_T'
end;
procedure No_terminal_T';
begin
    if preanalisis == '*' then begin
        empareja('*'); No_terminal_F; No_terminal_T'
    end
end procedure No_terminal_F;
begin
    if preanalisis == '(' then begin
```

```
        empareja('('); No_terminal_E; empareja(')')
    else if preanalisis == id then
        empareja('id');
end
```

Se ha de tener en cuenta que, en el caso especial de las $\lambda-$producciones, como ocurre para los no-terminales E' y T', si la variable *preanalisis* no coincide con $+$ ó $*$, respectivamente, se interpreta que el correspondiente símbolo no-terminal se ha reducido a la palabra vacía y se continua el análisis.

5.3. Análisis LR

5.3.1. *Introducción*

Las técnicas que utilizan el análisis sintáctico ascendente pueden ser vistas como totalmente opuestas a las técnicas de la sección anterior. En aquéllas se partía del símbolo inicial de la gramática, hasta conseguir una derivación izquierda que produjera la cadena de entrada, si esta pertenecía al lenguaje generado por la gramática. Por el contrario, el análisis ascendente parte de las hojas del correspondiente árbol de derivación derecho, o lo que es lo mismo, de la propia cadena de entrada, tratando de construir el árbol desde éstas hasta la raíz. Se dice que el árbol se construye por desplazamiento-reducción (o *shift-reduce*).

En este apartado vamos a estudiar el análisis ascendente predictivo, en el cual se busca una derivación derecha de la cadena de entrada de forma determinista. Este se sustenta en su aplicación a un grupo determinado de gramáticas: las gramáticas $LR(k)$. La L viene del hecho de que la lectura de la cadena de entrada se realiza de izquierda a derecha. La R significa que se produce un árbol de derivación derecho. Finalmente, k indica el número de símbolos que es necesario leer a la entrada para tomar la decisión de qué producción emplear.

Veamos cómo funciona. Un parser del tipo shift-reduce predictivo, puede verse como un autómata de pila determinista, extendido, que realiza el análisis de abajo hacia arriba. Dada una cadena de entrada w, obtiene una derivación más a la derecha, como

$$S \Rightarrow_{rm} \alpha_0 \Rightarrow_{rm} \alpha_1 \Rightarrow_{rm} \cdots \Rightarrow_{rm} \alpha_m \equiv w$$

Para seguir profundizando es necesaria una definición previa.

Definición 5.8 *Sea $G = (V_N, V_T, S, P)$ una CFG, y supóngase que*

$$S \overset{*}{\Rightarrow}_{rm} \alpha Aw \Rightarrow_{rm} \alpha\beta w \overset{*}{\Rightarrow}_{rm} xw$$

es una derivación más a la derecha. Podemos decir entonces que la forma sentencial derecha $\alpha\beta w$ puede ser reducida por la izquierda, mediante la producción $A \to \beta$ a la forma sentencial derecha αAw. Además, la subcadena β, en la posición en la que aparece se denomina manejador (o mango) de $\alpha\beta w$.

El concepto de mango hace referencia a la porción de la forma sentencial derecha considerada, que puede ser reducida por un no-terminal determinado, y que además conduce a otra forma sentencial derecha en la cual se pueden seguir aplicando reducciones para llegar a S. Obsérvese

que si la reducción de β mediante un no-terminal no llevara a otra forma sentencial derecha que pudiera conducir al símbolo inicial de la gramática, no sería un mango de $\alpha\beta w$.

Estudiemos la definición de mango con una gramática que consta de las siguientes producciones:

Gramática 5.3

$$\begin{array}{lcl} S & \rightarrow & Ac|Bd \\ A & \rightarrow & aAb|ab \\ B & \rightarrow & aBbb|abb \end{array}$$

Esta gramática genera el lenguaje $\{a^n b^n c | n \geq 1\} \cup \{a^n b^{2n} d | n \geq 1\}$. Sea la forma sentencial derecha *aabbbbd*. El único mango de esa cadena es *abb*, ya que *aBbbd* sigue siendo una forma sentencial derecha (i.e. una cadena que se puede obtener a partir del símbolo inicial de la gramática, mediante derivaciones más a la derecha de cierta longitud). Observar que *ab* no lo es, ya que aunque se tiene $A \rightarrow ab$, sin embargo *aAbbbd* no es una forma sentencial derecha para esa gramática.

Ahondando más en el tema, sea ahora αx una forma sentencial derecha tal que α es λ o termina con un símbolo no-terminal. Además, $x \in V_T^*$. Entonces denominamos a α como la porción abierta de αx, y a x como su porción cerrada.

El autómata de pila determinista guarda cada forma sentencial α_i con la porción abierta aún en la pila, y la porción cerrada en el resto de la cadena que queda por leer.

Por ejemplo, si $\alpha_i = \alpha A x$, entonces αA está, en ese momento, en la pila, y x aun no se ha leído.

Supongamos que $\alpha_{i-1} = \gamma B z$, y que se usa la producción $B \rightarrow \beta y$ en $\alpha_{i-1} \Rightarrow_{rm} \alpha_i$, en donde $\gamma\beta = \alpha A$ es la porción abierta de $\gamma\beta yz$, e $yz = x$ la porción cerrada de α_i. Por lo tanto $\gamma\beta$ está en la pila del autómata. Lo que hará el autómata es desplazar hacia la derecha, sobre algunos símbolos de yz (posiblemente ninguno, si $y = \lambda$) hasta encontrar el mango de α_i. Así, y también pasará a la cabeza de la pila.

Una vez que se ha delimitado el mango por la derecha, debe localizarse su límite izquierdo. Cuando se haya localizado se sustituye todo el mango, βy, por B, emitiendo como salida $B \rightarrow \beta y$. Ahora, en la cima de la pila está γB, y la entrada es z. Estas son, respectivamente, la porción abierta y cerrada de α_{i-1}. Recapitulando, un algoritmo de parsing *shift-reduce* debe tomar, a lo largo de su ejecución, tres tipos de decisiones:

1. Antes de cada movimiento debe elegir entre desplazar un símbolo de entrada, o reducir. O lo que es lo mismo, descubrir si ha encontrado el límite derecho de un mango.
2. Una vez que se ha determinado el límite derecho del mango, se ha de encontrar el límite izquierdo.
3. Después se ha de elegir qué no-terminal debe reemplazar a éste.

Las gramáticas de tipo $LR(k)$ definen un conjunto muy extenso de gramáticas para las cuales siempre podemos encontrar, mediante un algoritmo de análisis determinista, árboles de derivación derechos.

De manera informal, decimos que una gramática es $LR(k)$ si, dada una derivación más a la derecha como esta, $S = \alpha_0 \Rightarrow \alpha_1 \cdots \Rightarrow \alpha_m = z$, podemos determinar el mango de cada forma sentencial derecha, y determinar también qué no-terminal va a reemplazar en la pila al mango, examinando α_i de izquierda a derecha, pero no más de k símbolos, a partir del final del mango.

Veámoslo más en profundidad. Sea $\alpha_{i-1} = \alpha A w$ y $\alpha_i = \alpha\beta w$, en donde β es el mango de α_i. Sea además, $\beta = X_1 X_2 \ldots X_r$. Si la gramática en cuestión es $LR(k)$, podemos asegurar los siguientes tres hechos:

1. Si conocemos $\alpha X_1 X_2 \dots X_j$ y los primeros k símbolos de $X_{j+1} \dots X_r w$, podemos estar seguros de que el final derecho del mango no se alcanzará hasta que $j = r$.

2. Conociendo $\alpha\beta$, y como mucho los primeros k símbolos de w, podemos afirmar que β es el mango, y que β se va a reducir por A.

3. Si $\alpha_{i-1} = S$, podemos afirmar que la cadena de entrada va a ser aceptada.

Vamos a estudiar la comparación entre una gramática $LL(k)$ y una gramática $LR(k)$. Para que una gramática sea $LR(k)$, debe ser posible reconocer la parte derecha de una producción, habiendo visto todo lo que se deriva de la misma, además de k símbolos de anticipación de la entrada. Por otro lado, para que una gramática sea $LL(k)$, debe ser posible elegir una producción a aplicar únicamente mirando k símbolos derivados de su correspondiente parte derecha. Como puede verse, esta última condición es más restrictiva. La condición de las gramáticas LR permite disponer de más información; esto es, toda la parte derecha de la producción correspondiente, y además k símbolos a la derecha de la misma. Por lo tanto, **el conjunto de gramáticas LR es más amplio que el de las LL**. De hecho, el conjunto de las gramáticas LL es un subconjunto propio de las LR.

Analizadores LR

Vamos a comprobar, en esta sección, las bondades de los analizadores LR. Estas son:

- Como hemos visto antes, el conjunto de gramáticas LL es un subconjunto propio de las gramáticas LR. No solo eso, sino que es posible construir un analizador sintáctico LR para reconocer prácticamente la totalidad de las construcciones de los lenguajes de programación que se pueden definir mediante gramáticas CFG.

- El método de análisis LR es el método del tipo *shift-reduce*, sin retroceso, más general que se conoce pero, además, su aplicación es tan eficiente como la de otros métodos *shift-reduce* menos generales.

- Los errores pueden detectarse tan pronto como sea posible hacerlo, en un examen de la entrada de izquierda a derecha.

Sin embargo, como desventaja principal podemos indicar que, a veces, construir un analizador sintáctico de este tipo para una gramática dada es demasiado complejo como para intentar hacerlo a mano. Para ello se deben usar generadores de analizadores automáticos como `YACC`.

Vamos a ver, en este capítulo, tres técnicas diferentes para construir analizadores LR. Un primer método es el SLR (*Simple LR*), el más sencillo de construir pero es el menos potente de los tres. Esta potencia ha de ser entendida en términos de la cantidad de gramáticas que puede abarcar. Por otro lado tenemos el método LR canónico, que es el más costoso y potente de los tres. Por último, el método $LALR$ (*Look-Ahead LR*) está entre los otros dos, en términos de su complejidad y potencia.

El algoritmo genérico de análisis LR

Cada uno de los tres métodos se basa en un determinado tipo de tabla de análisis, cuya construcción depende totalmente del tipo de método. Sin embargo, en los tres casos, el algoritmo de análisis LR es siempre el mismo. Un diagrama estructural de lo que podría ser un sistema implementador del algoritmo aparece en la figura 5.7.

En esta figura puede verse el buffer de entrada, y la pila. En ella se almacena una cadena en la forma $s_0 X_1 s_1 \cdots X_m s_m$. El estado s_m está en la cima de la pila. Los símbolos X_i son símbolos

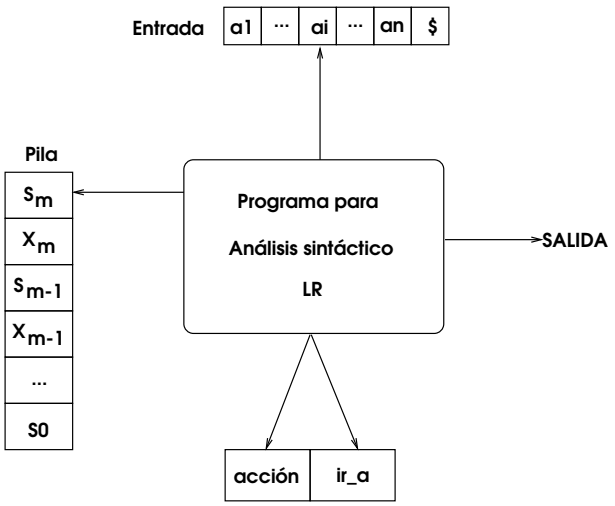


Figura 5.7: Diagrama estructural de un analizador sintáctico LR

gramaticales (i.e. $X_i \in V_T \cup V_N$), y los s_i son estados. Para acceder a la tabla de análisis se usa, precisamente, el símbolo de estado en la cima de la pila y el siguiente carácter a la entrada. Como puede verse, en la parte inferior de la figura 5.7, el algoritmo puede realizar dos tipos de movimientos (que dictará la tabla de análisis). El movimiento de tipo `acción` puede ser, a su vez, uno entre cuatro posibles acciones, que son **reducir**, **desplazar**, **aceptar** y **error**. El otro movimiento, `ir_a`, representa una transición entre estados del autómata de pila.

Nuevamente utilizaremos el concepto de *configuración*. En el analizador LR, una configuración constará de un par formado por el contenido de la pila y la entrada aun sin procesar:

$$(s_0 X_1 s_1 X_2 s_2 \cdots X_m s_m, a_i a_{i+1} \cdots a_n \$)$$

Observar que una configuración corresponde, si ignoramos los símbolos de estado, con la forma sentencial derecha que se va construyendo, a medida que avanza el análisis sintáctico. Si la cadena es reconocida llegaremos a la forma sentencial \$. La configuración anterior corresponderá a la forma sentencial $X_1 X_2 \cdots X_m a_i a_{i+1} \cdots a_n$.

Pasamos ahora a detallar los cuatro tipos de acciones. Para realizar un movimiento, se lee a_i de la cadena de entrada, y s_m en la cima de la pila. Nos vamos después a consultar la parte de acciones de la tabla de análisis, con $accion[s_m, a_i]$ y dependiendo del movimiento:

- Si $accion[s_m, a_i] = desplazar\ s$, se ha de desplazar a la pila el símbolo a_i, junto con el siguiente estado, dado en la tabla, s, pasándose ahora a la configuración

$$(s_0 X_1 s_1 X_2 s_2 \cdots X_m s_m a_i s, a_{i+1} \cdots a_n \$)$$

- Si $accion[s_m, a_i] = reducir A \rightarrow \beta$, entonces, si $|\beta| = r$, se han de reducir los primeros r símbolos X_k de la pila, junto con los correspondientes r estados, por el no terminal A. Obsérvese que $\beta = X_{m-r+1} X_{m-r+2} \cdots X_m$. Ahora la nueva configuración es

$$(s_0 X_1 s_1 X_2 s_2 \cdots X_{m-r} s_{m-r} A s, a_i a_{i+1} \cdots a_n \$)$$

  en donde el nuevo estado s se obtiene observando en la tabla el contenido de $ir\_a[s_{m-r}, A]$.

- Si $accion[s_m, a_i] = aceptar$, el análisis sintáctico termina con éxito.

- Si $accion[s_m, a_i] = error$, hay un error en la cadena de entrada, y se llama a una rutina de recuperación de errores.

La especificación del algoritmo correspondiente a la figura 5.7, es la siguiente:

Algoritmo 5.7

Algoritmo de análisis sintáctico LR

- **Entrada:** Una cadena de entrada w y una tabla de análisis sintáctico LR, con las funciones *accion* e ir_a para una gramática $G = (V_N, V_T, S, P)$ de tipo CFG.
- **Salida:** si $w \in L(G)$ entonces, una derivación derecha; si no una salida de error.
- **Método:** la pila del analizador contiene, inicialmente, a s_0, en donde s_0 es el estado inicial del autómata correspondiente. $w\$$ estará completa en el buffer de entrada. Sea ap el apuntador al símbolo de entrada actual. A partir de aquí, ejecutar:

```
Hacer que ap apunte al primer símbolo de w$.
Repeat forever
Begin
    Sea s el estado en la cima de la pila, y a el símbolo apuntado
    por ap
    if accion[s, a] = desplazar s' then begin
       Introducir a, y después s' en la cima de la pila
       Hacer que ap apunte al siguiente símbolo a la entrada
    End
    else if accion[s, a] = reducir A → β then
    Begin
       Extraer 2 × |β| símbolos de la pila
       Sea s' el estado que ahora está en la cima de la pila
       Introducir A y después introducir el estado resultante de
       ir_a[s', A]
       Emitir la producción A → β
    End
    else if accion[s, a] = aceptar then return
    else error()
End
```

5.3.2. *Tabla de Análisis SLR*

Como ya se ha mencionado en este capítulo, la construcción de una tabla SLR es relativamente sencilla, si la comparamos con la construcción de una LR canónica, ó una LALR. Sin embargo, el conjunto de gramáticas para las cuales se puede construir este tipo de tablas es el más pequeño de los tres.

La base para construir analizadores SLR la forman un grupo de conjuntos de items $LR(0)$. Estos items $LR(0)$, para una gramática dada G, están formados por una de las producciones de G, con un punto en cualquier posición de su correspondiente parte derecha. Por ejemplo, a partir de la producción $A \rightarrow XYZ$ se pueden construir los siguientes items $LR(0)$:

$$
\begin{array}{lcl}
A & \rightarrow & \bullet XYZ \\
A & \rightarrow & X \bullet YZ \\
A & \rightarrow & XY \bullet Z \\
A & \rightarrow & XYZ\bullet
\end{array}
$$

$A \rightarrow \bullet$ sería el único item asociado a la producción $A \rightarrow \lambda$. Los conjuntos formados por estos items, denominados **colección canónica** $LR(0)$, van a constituir los estados de un autómata de pila que se encargará de reconocer los *prefijos viables* de la gramática.

Definición 5.9 *Sea* $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta w$ *una derivación derecha para la gramática G. Decimos que una cadena* γ *es un prefijo viable de G si* γ *es un prefijo de* $\alpha\beta$*. Esto es,* γ *es una cadena que es un prefijo de alguna forma sentencial derecha, pero que no abarca más allá del límite derecho del mango de esa forma sentencial.*

Para construir la colección canónica previamente mencionada, es necesario definir la gramática aumentada, y dos funciones: *cerradura* e *ir_a*.

Definición 5.10 *Sea* $G = (V_N, V_T, P, S)$ *una CFG. Definimos la gramática aumentada* G' *derivada de G como* $G' = (V_N \cup \{S'\}, V_T, P \cup \{S' \rightarrow S\}, S')$.

Este concepto de gramática aumentada es necesario para indicar mejor al analizador cuando determinar la aceptación de la cadena.

Definición 5.11 *Sea I un conjunto de elementos del análisis sintáctico* $LR(0)$ *para G. Entonces* $cerradura(I)$ *es el conjunto de items construido a partir de I siguiendo las dos reglas siguientes:*

1. *Añadir todo item de I a* $cerradura(I)$.
2. *Si* $A \rightarrow \alpha \bullet B\beta \in cerradura(I)$, *y* $B \rightarrow \gamma \in P$, *entonces añadir* $B \rightarrow \bullet\gamma$ *a* $cerradura(I)$, *si aún no se ha añadido. Repetir esta regla hasta que no se puedan añadir más items.*

La función *ir_a* está definida en el dominio del producto cartesiano formado por los conjuntos de items I, y el conjunto de símbolos de la gramática en cuestión, $(\{V_N - S'\} \cup \{V_T - \$\})$. Dicho de otra forma, el conjunto formado por parejas de la forma (*estado_actual*,*nueva_entrada*). El codominio es el de las funciones de cerradura.

Definición 5.12 *Sea* $I_X = \{[A \rightarrow \alpha X \bullet \beta]$ *tal que* $[A \rightarrow \alpha \bullet X\beta] \in I\}$. *Entonces,* $ir\_a(I, X) = cerradura(I_X)$.

Esto es, si I es el conjunto de items válidos para algún prefijo viable γ, la función $ir\_a(I, X)$ nos dará el conjunto de items válidos para el prefijo viable γX. Por lo tanto, dado un estado actual del autómata, y un símbolo de la gramática a procesar, la función *ir_a* nos llevará a un estado formado por los items en los que el meta-símbolo aparece a la derecha del mismo, indicando que lo hemos reconocido.

Ahora ya podemos construir la colección canónica de conjuntos de items $LR(0)$, para una gramática aumentada G'.

Algoritmo 5.8

Generación de la colección canónica de conjuntos de items para una gramática aumentada G'.

- **Entrada:** una gramática aumentada G', a partir de una CFG, G.
- **Salida:** C, la colección canónica de conjuntos de items.
- **Método:**

  Begin

Inicializar $C := \{cerradura(\{[S' \to \bullet S]\})\}$
Repetir
Para cada conjunto de items I en C, y cada símbolo gramatical X tal que $ir\_a(I, X) \neq \emptyset$ e $ir\_a(I, X) \notin C$ hacer
Añadir $ir\_a(I, X)$ a C.
Hasta que no se puedan añadir más conjuntos de items a C.
End

Ejemplo

Vamos a ver todos estos conceptos con un ejemplo. Lo introduciremos con la gramática 5.4. Si aumentamos esta gramática, obtenemos la gramática 5.5.

Gramática 5.4

$$\begin{array}{lcl} E & \to & E+T|T \\ T & \to & T*F|F \\ F & \to & (E)|id \end{array}$$

Gramática 5.5

$$\begin{array}{lcl} E' & \to & E \\ E & \to & E+T|T \\ T & \to & T*F|F \\ F & \to & (E)|id \end{array}$$

Si empezamos aplicando el algoritmo 5.8, debemos inicializar C con la cerradura del conjunto formado por el item $[E' \to \bullet E]$. Si le aplicamos la definición 5.11, obtenemos el siguiente conjunto I_0:

$$\begin{array}{lcl} E' & \to & \bullet E \\ E & \to & \bullet E+T \\ E & \to & \bullet T \\ T & \to & \bullet T*F \\ T & \to & \bullet F \\ F & \to & \bullet(E) \\ F & \to & \bullet id \end{array}$$

Ahora entramos en el bucle del algoritmo 5.8, de tal manera que para las combinaciones siguientes $(I, simbolo)$, obtenemos los conjuntos de items indicados:

- Para (I_0, E) obtenemos un I_1:

$$\begin{array}{lcl} E' & \to & E\bullet \\ E & \to & E \bullet +T \end{array}$$

- Para (I_0, T) obtenemos un I_2:

$$\begin{array}{lcl} E & \to & T\bullet \\ T & \to & T \bullet *F \end{array}$$

- Para (I_0, F) obtenemos un I_3:

$$T \to F\bullet$$

- Para $(I_0, ()$ obtenemos un I_4:

$$
\begin{array}{lcl}
F & \rightarrow & (\bullet E) \\
E & \rightarrow & \bullet E + T \\
E & \rightarrow & \bullet T \\
T & \rightarrow & \bullet T * F \\
T & \rightarrow & \bullet F \\
F & \rightarrow & \bullet (E) \\
F & \rightarrow & \bullet id
\end{array}
$$

- Para (I_0, id) obtenemos un I_5:

$$
\begin{array}{lcl}
F & \rightarrow & id\bullet
\end{array}
$$

- Para $(I_1, +)$ obtenemos un I_6:

$$
\begin{array}{lcl}
E & \rightarrow & E + \bullet T \\
T & \rightarrow & \bullet T * F \\
T & \rightarrow & \bullet F \\
F & \rightarrow & \bullet (E) \\
F & \rightarrow & \bullet id
\end{array}
$$

- Para $(I_2, *)$ obtenemos un I_7:

$$
\begin{array}{lcl}
T & \rightarrow & T * \bullet F \\
F & \rightarrow & \bullet (E) \\
F & \rightarrow & \bullet id
\end{array}
$$

- Para (I_4, E) obtenemos un I_8:

$$
\begin{array}{lcl}
F & \rightarrow & (E\bullet) \\
E & \rightarrow & E \bullet + T
\end{array}
$$

- Para (I_6, T) obtenemos un I_9:

$$
\begin{array}{lcl}
E & \rightarrow & E + T\bullet \\
T & \rightarrow & T \bullet * F
\end{array}
$$

- Para (I_7, F) obtenemos un I_{10}:

$$
\begin{array}{lcl}
T & \rightarrow & T * F\bullet
\end{array}
$$

- Para $(I_8,))$ obtenemos un I_{11}:

$$
\begin{array}{lcl}
F & \rightarrow & (E)\bullet
\end{array}
$$

Observar que podemos diseñar un AFD, si hacemos que cada estado i se corresponda con cada I_i. Las transiciones serían de dos tipos:

- De $A \rightarrow \alpha \bullet X\beta$ a $A \rightarrow \alpha X \bullet \beta$, etiquetada con X,
- De $A \rightarrow \alpha \bullet B\beta$ a $B \rightarrow \bullet\gamma$ etiquetada con λ.

Una representación gráfica puede verse en la figura 5.8.

Una vez que podemos calcular la colección canónica de items $LR(0)$ de una gramática, ya estamos listos para calcular la tabla de análisis sintáctico SLR. Como ya se ha visto, en el análisis LR genérico hay cuatro tipos de acciones, para aceptar y rechazar una cadena o para reducir un no terminal ó avanzar en la lectura de w. El algoritmo 5.9 puede utilizarse para rellenar las partes *accion* e ir_a de la tabla de análisis.

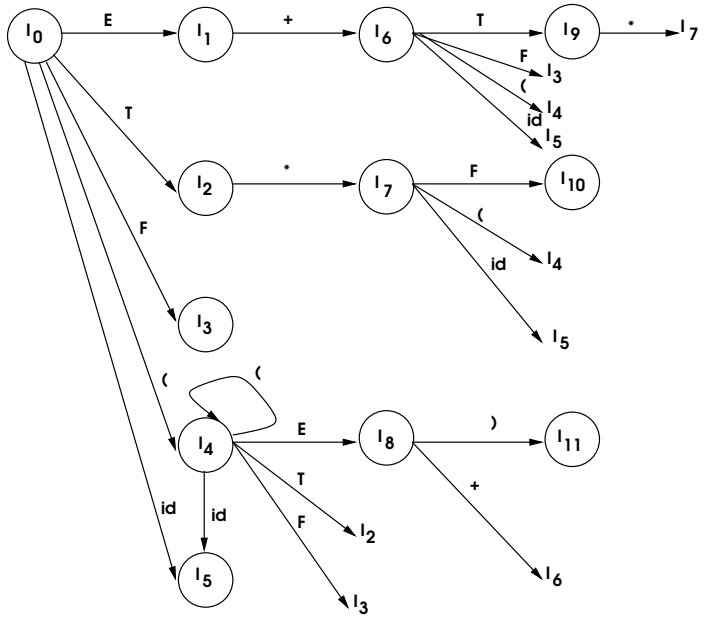


Figura 5.8: AFD formado a partir de la colección canónica de items $LR(0)$ de la gramática 5.5.

Algoritmo 5.9

Tabla de análisis sintáctico SLR

- **Entrada:** Una gramática aumentada G'.
- **Salida:** Las funciones *accion* e ir_a de la tabla de análisis sintáctico SLR para G'.
- **Método:**
  1. Constrúyase $C = \{I_0, I_1, \ldots, I_n\}$, la colección de conjuntos de items $LR(0)$ para G'.
  2. Para el AFD, el estado i se construye a partir del conjunto I_i. Las acciones de análisis sintáctico para el estado i se construyen según:
     a) Si $[A \to \alpha \bullet a\beta]$ está en I_i, y además $ir\_a(I_i, a) = I_j$, asignar $shift\ j$ a $accion[i, a]$, siendo $a \in V_T$.
     b) Si $[A \to \alpha\bullet] \in I_i$, entonces asignar $reduce\ A \to \alpha$ a $accion[i, a]$, para todo $a \in FOLLOW(A)$.
     c) Si $[S' \to S\bullet] \in I_i$, entonces asignar *aceptar* a $accion[i, \$]$.
  3. Las transiciones ir_a para el estado i se construyen, para todos los $A \in V_N$ utilizando la regla

$$\text{si } ir\_a(I_i, A) = I_j, \text{ entonces } ir\_a[i, A] = j$$

  4. Todas las entradas de la tabla no actualizadas por 2 y 3 son consideradas *error*.
  5. El estado inicial es el construido a partir del conjunto I_0.

Observar que si los pasos 2.a. y 2.b. generan acciones contradictorias, para una misma celda $accion[i, a]$, se dice entonces que la gramática no es $SLR(1)$, y por lo tanto no es posible construir un analizador sintáctico para la gramática. Una tabla de análisis generada mediante este algoritmo se denomina **tabla de análisis sintáctico SLR(1)**.

Estado	acción						*ir_a*		
	id	+	*	(	)	$	E	T	F
0	d5			d4			1	2	3
1		d6				*aceptar*			
2		r2	d7		r2	r2			
3		r4	r4		r4	r4			
4	d5			d4			8	2	3
5		r6	r6		r6	r6			
6	d5			d4				9	3
7	d5			d4					10
8		d6			d11				
9		r1	d7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Figura 5.9: Tabla de análisis sintáctico SLR(1) de la gramática 5.4

Vamos a utilizar nuevamente la gramática 5.5 para desarrollar un ejemplo en el que construir la tabla de análisis sintáctico $SLR(1)$. Los conjuntos $FOLLOW$ son

$$
\begin{aligned}
&FOLLOW(E') = \{\$\} \\
&FOLLOW(E) = \{\$,), +\} \\
&FOLLOW(T) = \{\$,), +, *\} \\
&FOLLOW(F) = \{\$,), +, *\}
\end{aligned}
$$

Comencemos:

- Para el estado I_0, por el paso 2.a. consideramos las producciones:
  - $F \rightarrow \bullet(E)$ que da lugar a $shift\ I_4$
  - $F \rightarrow \bullet id$ que da lugar a $shift\ I_5$
- Para el estado I_1 se tienen en cuenta, por 2.a. y 2.b. respectivamente,
  - $E \rightarrow E \bullet +T$ que da lugar a $shift\ I_6$
  - $E' \rightarrow E\bullet$, que da lugar a $aceptar$, en $accion[I_1, \$]$
- Para I_2, por 2.a. y 2.b. respectivamente,
  - $T \rightarrow T \bullet *F$ da lugar a $shift\ I_7$
  - $E \rightarrow T\bullet$ da lugar a reduce $E \rightarrow T$ para $accion[I_2, +]$, $accion[I_2,)]$, $accion[I_2, \$]$.
- Para I_3, por 2.b. $T \rightarrow F\bullet$ da lugar a reduce $T \rightarrow F$ en $accion[I_3, \$]$, $accion[I_3, +]$, $accion[I_3,)]$, $accion[I_3, *]$.
- ...

En la figura 5.9 se muestra como queda la tabla de análisis.

5.3.3. *Tabla de análisis LR-canónica*

El método SLR no es lo suficientemente bueno como para reconocer lenguajes generados por ciertas gramáticas, que aun sin ser ambiguas pueden producir resultados ambiguos en la tabla de análisis sintáctico $SLR(1)$. Es el caso del grupo de producciones relativas al manejo de asignaciones siguiente:

$$
\begin{array}{lcl}
S & \rightarrow & L = R \\
S & \rightarrow & R \\
L & \rightarrow & *R \\
L & \rightarrow & id \\
R & \rightarrow & L
\end{array}
$$

Al aplicar el algoritmo 5.8 conseguimos la siguiente colección de conjuntos de items:

$$
\begin{array}{lcl}
I_0 & & \\
S' & \rightarrow & \bullet S \\
S & \rightarrow & \bullet L = R \\
S & \rightarrow & \bullet R \\
L & \rightarrow & \bullet * R \\
L & \rightarrow & \bullet id \\
R & \rightarrow & \bullet L \\
 & & \\
I_1 & & \\
S' & \rightarrow & S\bullet \\
 & & \\
I_2 & & \\
S & \rightarrow & L\bullet = R \\
R & \rightarrow & L\bullet \\
 & & \\
I_3 & & \\
S & \rightarrow & \bullet R \\
 & & \\
I_4 & & \\
L & \rightarrow & * \bullet R \\
R & \rightarrow & \bullet L \\
L & \rightarrow & \bullet * R \\
L & \rightarrow & \bullet id
\end{array}
\qquad
\begin{array}{lcl}
I_5 & & \\
L & \rightarrow & id\bullet \\
 & & \\
I_6 & & \\
S & \rightarrow & L = \bullet R \\
R & \rightarrow & \bullet L \\
L & \rightarrow & \bullet * R \\
L & \rightarrow & \bullet id \\
 & & \\
I_7 & & \\
L & \rightarrow & *R\bullet \\
 & & \\
I_8 & & \\
R & \rightarrow & L\bullet \\
 & & \\
I_9 & & \\
S & \rightarrow & L = R\bullet
\end{array}
$$

Si se observa el conjunto de items I_2, al estar en ese estado, y recibir un signo $=$, por el primer item se debería desplazar, e ir a I_6. Por el segundo, y dado que $= \in FOLLOW(R)$, se debería reducir por $R \rightarrow L$. Se produce una situación de conflicto $shift/reduce$ cuando en ese estado llega el signo $=$.

Como ya se puede suponer, la gramática ejemplo anterior no es ambigua y el conflicto surge por las propias limitaciones del método $SLR(1)$ para construir la tabla de análisis. Obsérvese que no existe en el lenguaje generado por la gramática, una sentencia que comience con $R = \cdots$ y por lo tanto la acción reduce no debería considerarse ahí. Es, por tanto, una reducción no válida.

El problema es que puede haber ciertos prefijos viables, para los cuales no sea correcta una reducción. El método $SLR(1)$ carece de la potencia necesaria para recordar el contexto a la izquierda. Para posibilitar esto, se debe dotar a los estados de información adicional que permita detectar y prohibir esas reducciones no válidas. Esto es posible en el análisis LR canónico.

Incorporando información adicional al método de análisis

Para añadir más información al análisis se debe redefinir el concepto de item, añadiendo un segundo componente al mismo. Ahora la forma general de un item va a ser $[A \rightarrow \alpha \bullet \beta, a]$ en donde $A \rightarrow \alpha\beta$ es una producción de la gramática, y a un terminal, ó \$.

A este tipo de item se le denomina item del análisis sintáctico $LR(1)$. El número 1 denota la longitud del símbolo a, o símbolo de anticipación del item. Este símbolo no va a tener efecto en aquellos items en los cuales la cadena $\beta \neq \lambda$ ya que no van a dictar una reducción. Sin embargo, un item $[A \rightarrow \alpha\bullet, a]$ indica que se aplique una reducción siempre que el siguiente símbolo a la entrada sea a. Obviamente, las a's van a formar un subconjunto de $FOLLOW(A)$, y pueden formar un subconjunto propio de este, como debería ser en el ejemplo anterior.

Definición 5.13 *Sea una gramática $G = (V_N, V_T, S, P)$. Sea γ un prefijo viable de G. Sean $A \rightarrow \alpha\beta \in P$ y $a \in V_T \cup \{\$\}$. Se dice que el item $LR(1)$ $[A \rightarrow \alpha \bullet \beta, a]$ es válido para γ si existe una derivación $S \Rightarrow^*_{rm} \delta Aw \Rightarrow_{rm} \delta\alpha\beta w$, en donde*

1. *$\gamma = \delta\alpha$ y*
2. *a es el primer símbolo de w, ó $w = \lambda$ y $a = \$$.*

Construcción de la colección de conjuntos de elementos $LR(1)$ válidos

El procedimiento de construcción de la colección de conjuntos de elementos $LR(1)$ válidos es muy similar al que hace lo propio con la colección de conjuntos de elementos $LR(0)$. Se trata de modificar los procedimientos *cerradura* e *ir_a*.

Definición 5.14 *Sea I un conjunto de items. Se define la función cerradura(I) como el conjunto resultante de aplicar el siguiente procedimiento:*

begin

repeat

for *cada elemento $[A \rightarrow \alpha \bullet B\beta, a] \in I$, cada producción $B \rightarrow \gamma \in G'$ y cada $b \in V_T$ tal que $b \in FIRST(\beta a)$ y $[B \rightarrow \bullet\gamma, b] \notin I$, entonces*

Añadir $[B \rightarrow \gamma, b]$ a I

until *no se puedan añadir más elementos a I;*

end;

Esta definición ha cambiado sensiblemente respecto de la definición 5.11 en la que se describía la función *cerradura* para la colección de conjuntos de items $SLR(1)$. Ahora hay que tener en cuenta los símbolos terminales que pueden aparecer, en formas sentenciales derechas, inmediatamente a continuación del no terminal mediante el cual se podría aplicar una reducción.

La forma de trabajar del procedimiento es la siguiente: partiendo de un conjunto I con unos determinados items iniciales, se toma en consideración el item $[A \rightarrow \alpha \bullet B\beta, a]$. Se supone que este item es válido para un prefijo viable δ concreto, en el cual se tiene como sufijo α, o dicho de otra forma $\delta = \mu\alpha$. Por lo tanto, existe una derivación derecha

$$S \Rightarrow^*_{rm} \mu Aay \Rightarrow_{rm} \mu\alpha B\beta ay$$

Ahora supongamos que, a partir de βay se deriva la cadena de terminales bt en donde $t \in V_T^*$. Entonces, para cada producción $B \rightarrow \varphi$, se va a tener una derivación más a la derecha

$$S \Rightarrow^*_{rm} \mu\alpha Bbt \Rightarrow_{rm} \mu\alpha\varphi bt$$

por lo tanto, $[B \to \bullet\varphi, b]$ será un item válido para el mismo prefijo viable δ. Lo que se hace, al fin, es incluir todos los items formados por las producciones $B \to \varphi$ y los símbolos terminales que están en el conjunto $FIRST(\beta a) \equiv FIRST(\beta a y)$.

Definición 5.15 *Sea I un conjunto de items, y $X \in V_N \cup V_T \cup \{\$\}$. Se define la función $ir\_a(I, X)$ como el conjunto de items resultante de aplicar el siguiente procedimiento:*

begin

Sea J el conjunto de items $[A \to \alpha X \bullet \beta, a]$ tal que $[A \to \alpha \bullet X\beta, a]$ está en I. El estado resultante es el que viene dado por $cerradura(J)$.

end;

La función ir_a es idéntica a la anterior.

Ahora la colección de conjuntos de items $LR(1)$ viene descrita en la siguiente definición.

Definición 5.16 *Sea $G = (V_N, V_T, S, P)$ una CFG. Sea G' la gramática aumentada de G, en donde el símbolo inicial es ahora S'. La colección de conjuntos de items $LR(1)$ es la resultante de aplicar el siguiente procedimiento sobre G'.*

begin

$C = \{cerradura(\{[S' \to \bullet S, \$]\})\}$;
repeat
for *cada $I \in C$, y cada $X \in V_N \cup V_T \cup \{\$\}$ tal que $ir\_a(I, X) \neq \emptyset$ y $I \notin C$* **do**
añadir $ir\_a(I, X)$ a C
until *no se puedan añadir más conjuntos a C*

end;

Idéntica al algoritmo 5.8.

Ejemplo: construyendo la colección de items $LR(1)$

Vamos a utilizar la gramática aumentada 5.6 para calcular la colección de conjuntos de items $LR(1)$.

Gramática 5.6

$$\begin{array}{lcl} S' & \to & S \\ S & \to & CC \\ C & \to & cC \\ C & \to & d \end{array}$$

Primero se debe calcular $cerradura(\{[S' \to \bullet S, \$]\})$. Para ello, a partir del item $[S' \to \bullet S, \$]$, como $FIRST(\$) = \{\$\}$, introducimos en I_0 el nuevo item $[S \to \bullet CC, \$]$. A partir de este, con el conjunto $FIRST(C\$) \equiv FIRST(C)$ ya que C no genera la palabra vacía, y las producciones $C \to cC$ y $C \to d$ generan los nuevos items

$[C \to \bullet cC, c]$
$[C \to \bullet cC, d]$
$[C \to \bullet d, c]$
$[C \to \bullet d, d]$

Ya tenemos el conjunto I_0:

$[S' \rightarrow \bullet S, \$]$
$[S \rightarrow \bullet CC, \$]$
$[C \rightarrow \bullet cC, c/d]$
$[C \rightarrow \bullet d, c/d]$

Observar que se ha tratado de simplificar la aparatosidad de los nuevos conjuntos, utilizando una notación reducida en la que se representan varios items por uno sólo cuando lo único que no coincide es el símbolo terminal a la derecha.

Ahora se ha de calcular la función $ir\_a(I_0, S)$ que nos da un nuevo estado I_1 con el item inicial $[S' \rightarrow S\bullet, \$]$. Al aplicar la cerradura a este nuevo item, vemos que ya no es posible generar más, así que I_1 queda:

$[S' \rightarrow S\bullet, \$]$

Cuando se calcula $ir\_a(I_0, C)$, se ha de aplicar la cerradura a $[S \rightarrow C \bullet C, \$]$. Se obtiene un I_2:

$[S \rightarrow C \bullet C, \$]$
$[C \rightarrow \bullet cC, \$]$
$[C \rightarrow \bullet d, \$]$

ya que ahora $\$ \in FIRST(\$)$. No se pueden añadir más items.

Al calcular $ir\_a(I_0, c)$, se cierra $\{[C \rightarrow c \bullet C, c/d]\}$. Se tienen que añadir nuevos items para $C \rightarrow cC$ y $C \rightarrow d$, siendo el segundo componente c/d ya que $c \in FIRST(c)$ y $d \in FIRST(d)$. I_3 queda:

$[C \rightarrow C \bullet C, c/d]$
$[C \rightarrow \bullet cC, c/d]$
$[C \rightarrow \bullet d, c/d]$

El último ir_a para I_0 se hace con d, cerrando $\{[C \rightarrow d\bullet, c/d]\}$. Se obtiene un I_4:

$[C \rightarrow d\bullet, c/d]$

Ya no quedan más elementos gramaticales que considerar, a partir de I_0. Pasamos a I_1. De este estado no salen transiciones. Pasamos a I_2. En este se van a dar transiciones con $\{C, c, d\}$. Para C, $ir\_a(I_2, C)$ se obtiene cerrando $\{[S \rightarrow CC\bullet, \$]\}$. En el cierre no se añade ningún elemento más, con lo que I_5 es

$[S \rightarrow CC\bullet, \$]$

Para obtener $ir\_a(I_2, c)$, se debe hacer la cerradura de $\{[C \rightarrow c \bullet C, \$]\}$. Se obtiene, ya que $FIRST(\$) = \{\$\}$, los elementos para I_6:

$[C \rightarrow c \bullet C, \$]$
$[C \rightarrow \bullet cC, \$]$
$[C \rightarrow \bullet d, \$]$

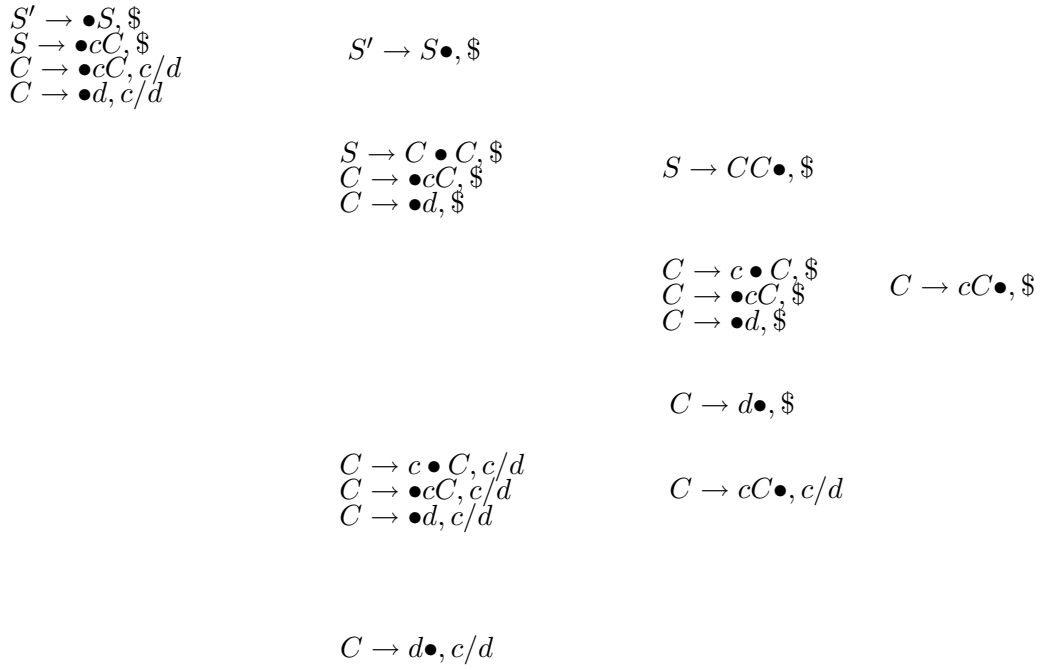


Figura 5.10: Autómata generado a partir de la colección de items $LR(1)$, y la función $ir\_a()$ del ejemplo para la gramática 5.6

.

Para obtener $ir\_a(I_2, d)$, se hace la cerradura de $\{[C \to \bullet d, \$]\}$ que no añade más elementos a I_7:

$$[C \to d\bullet, \$]$$

Pasamos al estado I_3. En este va a haber tres transiciones, $ir\_a(I_3, C)$, $ir\_a(I_3, c)$ e $ir\_a(I_3, d)$. La primera da lugar a un nuevo estado, I_8:

$$[C \to cC\bullet, c/d]$$

Las otras dos van a parar a los estados I_3 e I_4, respectivamente. Ahora, de I_4 e I_5 no salen transiciones posibles. Sin embargo, para I_6 hay transiciones para C, c y d. La función $ir\_a(I_6, C)$ nos lleva a un nuevo estado, I_9:

$$[C \to cC\bullet, \$]$$

La transición $ir\_a(I_6, c)$ nos lleva al propio I_6 e $ir\_a(I_6, d)$ a I_7. Ya no se generan nuevos estados ni transiciones.
El autómata así generado puede verse en la figura 5.10.

La tabla de análisis LR canónico

El algoritmo 5.10 construye las funciones *accion* e Ir_a del análisis LR-canónico.

Algoritmo 5.10

Construcción de la tabla de análisis sintáctico $LR(1)$-canónico.

- **Entrada:** una gramática aumentada G', a partir de una CFG, G.
- **Salida:** Las funciones *accion* e ir_a de la tabla de análisis sintáctico LR-canónico para G'.
- **Método:**
  1. Constrúyase, mediante la función de la definición 5.16, la colección de conjuntos $LR(1)$ para G'.
  2. Los nuevos estados, i, del analizador sintáctico se construyen a partir de los correspondientes I_i. Además, las acciones de análisis sintáctico se construyen siguiendo los siguientes tres puntos:
     a) Si $[A \to \alpha \bullet a\beta, b] \in I_i$, e $ir\_a(I_i, a) = I_j$ introducir $shift\ j$ en $accion[i, a]$, siendo $a \in V_T$.
     b) Si $[A \to \alpha\bullet, a] \in I_i$, y $A \neq S'$, entonces introducir $reduce\ A \to \alpha$ en $accion[i, a]$.
     c) Si $[S' \to S\bullet, \$] \in I_i$, entonces introducir *aceptar* en $accion[i, \$]$.
  3. Las transiciones, para el estado i se determinan según la siguiente regla:
     Si $ir\_a(I_i, A) = I_j$, entonces $ir\_a[i, A] = j$
  4. Todas las celdas vacías de la tabla se consideran *error*
  5. El estado inicial del analizador sintáctico es el correspondiente i construido a partir del conjunto que contiene el item $[S' \to \bullet S, \$]$.

Al igual que ocurría en el algoritmo 5.9, si las reglas del paso 2 produjeran movimientos conflictivos en una misma celda de la tabla, esto significaría que la gramática no es $LR(1)$, y por lo tanto el algoritmo no va a funcionar correctamente. La tabla construida por este algoritmo se denomina tabla de análisis sintáctico $LR(1)$-canónico. Asimismo, un analizador que utilice esta tabla se denomina analizador sintáctico $LR(1)$-canónico.

Ejemplo de construcción de una tabla de análisis $LR(1)$-canónico

Si aplicamos el algoritmo 5.10 al conjunto de items obtenidos a partir de la gramática 5.6, obtendremos la tabla de análisis sintáctico $LR(1)$ canónico que se muestra en la figura 5.11

5.3.4. *Tabla de análisis LALR*

Ya hemos visto cómo construir una tabla de análisis SLR, con un método relativamente sencillo, y que generaba un número de estados más o menos manejable. Después hemos visto el método de construcción de una tabla de análisis LR-canónico. Se ha podido comprobar que el método era más potente (i.e. abarcaba un número de gramáticas mayor) pero a cambio, también ganaba en complejidad, tanto en operaciones como en estados para el AFD generado.

Esto es así porque incorpora información adicional en los items: el segundo componente que condiciona el poder realizar o no una reducción.

Ahora vamos a ver el método $LALR$ (Look-Ahead LR). Su funcionamiento se basa en la idea de fusionar, de alguna forma, algunos de los estados que se producen al crear el conjunto de items $LR(1)$ para el análisis sintáctico LR-canónico.

Estado	acción			*ir_a*	
	c	d	$	S	C
0	d3	d4		1	2
1			*aceptar*		
2	d6	d7			5
3	d3	d4			8
4	r3	r3			
5			r1		
6	d6	d7			9
7			r3		
8	r2	r2			
9			r2		

Figura 5.11: Tabla de análisis sintáctico LR(1)

Si echamos un vistazo a los diferentes estados que aparecen en la figura 5.10, podemos ver que algunos estados se parecen. Es más, son idénticos en los primeros componentes de los items correspondientes. Esto se explica si se observan los items de cada conjunto, sin los segundos componentes y se interpretan como conjuntos generados por el método $LR(0)$. Por ejemplo, en ese mismo autómata de la figura 5.10, los estados I_4 e I_7 son muy similares. Estos dos estados habrían sido uno sólo en la generación $LR(0)$ de la colección de conjuntos de items. El estado correspondiente no habría distinguido entre los terminales c y d frente a \$ y en ciertas cadenas se habría cometido un error, como el que se vio en el apartado 5.3.3.

La gramática 5.6 genera el mismo lenguaje que la expresión regular $c * dc * d$. Por lo tanto, tiene sentido distinguir entre los estados I_4 e I_7. Al estado I_4 se pasaría si, habiendo leído en la entrada una d (y sólo una desde el principio), viniera después una c, u otra d (correspondería a la subexpresión $c * d$ izquierda de la expresión regular). Sin embargo, al estado I_7 se pasaría desde el I_2, o desde el I_6, siempre que, leyendo una d, el siguiente símbolo a la entrada fuera el delimitador de la cadena.

Si fusionamos los estados I_4 e I_7 en el estado $I_{4,7}$, su contenido sería ahora $\{[C \rightarrow d\bullet, c/d/\$]\}$. Pero, ¿qué pasaría con las transiciones que van a parar a esos estados y las que salen de ellos? Si se comparan las funciones *cerradura* para los métodos SLR y LR-canónico que aparecen en las definiciones 5.12 y 5.15 respectivamente, se puede observar que el resultado de ésta depende únicamente del primer componente de los items correspondientes. Por lo tanto, las transiciones de los estados originales pueden fundirse en el nuevo estado $I_{4,7}$.

No debemos olvidar, sin embargo, que en algunos casos se pierde información en la fusión, cuando el segundo componente del item determina la reducción.

Después de hacer todas las posibles fusiones, el nuevo autómata queda como el que aparece en la figura 5.12.

A continuación se introduce el algoritmo que construye la tabla $LALR$.

Algoritmo 5.11

Construcción de la tabla de análisis sintáctico $LALR$.

- **Entrada:** una gramática aumentada G', a partir de una CFG, G.
- **Salida:** Las funciones *accion* e *ir_a* de la tabla de análisis sintáctico $LALR$ para G'.
- **Método:**

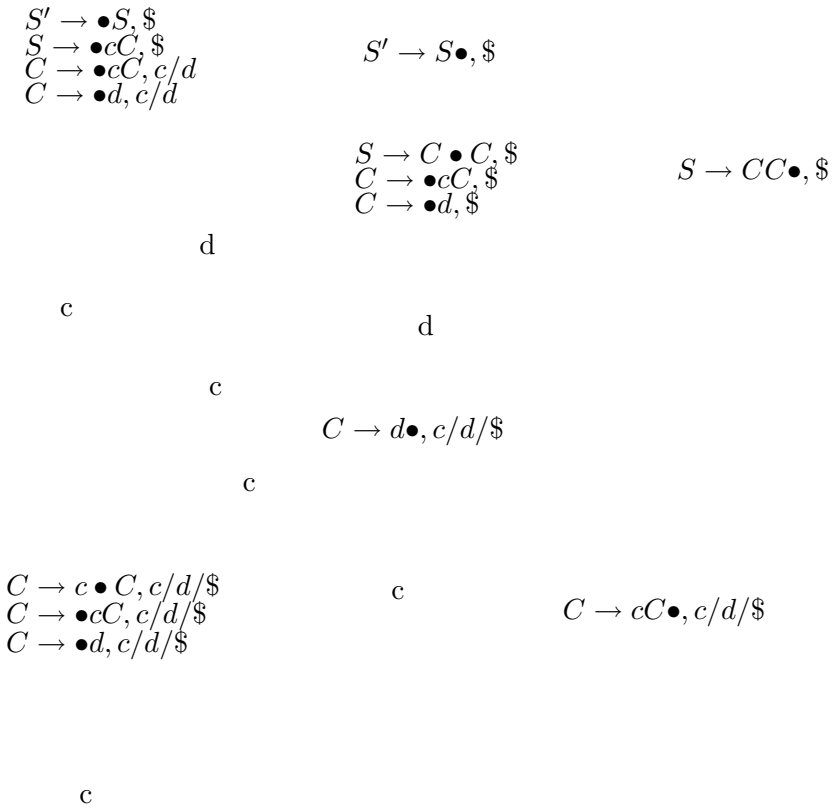


Figura 5.12: Autómata generado a partir de la colección de items $LR(1)$, y la función $ir\_a()$ del ejemplo para la gramática 5.6

.

1. Constrúyase, mediante la función de la definición 5.16, la colección de conjuntos $LR(1)$ para G'.
2. Encuéntrense los conjuntos de items, en los que los primeros componentes de todos sus items coinciden y fúndanse en un único estado nuevo.
3. Sea $C' = \{J_0, J_1, \ldots, J_m\}$ los nuevos conjuntos de items $LR(1)$ obtenidos. Construir las acciones para el estado i, a partir de el conjunto J_i, como en el algoritmo 5.10. Si existe algún conflicto en las acciones correspondientes al análisis sintáctico, el algoritmo resulta no eficaz y se dice que la gramática no es $LALR(1)$.
4. Las transiciones, para el estado i se determinan según la siguiente regla:

   Si J es la unión de uno o más conjuntos de elementos $LR(1)$, digamos $J = I_1 \cup I_2 \cup \cdots \cup I_k$, entonces $ir\_a(I_1, X), ir\_a(I_2 X), \ldots$ e $ir\_a(I_k, X)$, van a estados $LR(1)$ que coinciden en el conjunto formado por el primer componente de todos sus items respectivos. Sea J_t el estado resultante de la fusión de todos esos estados. Entonces $ir\_a(J, X) = J_t$.

La tabla resultante de la aplicación del algoritmo 5.11 se denomina tabla de análisis sintáctico $LALR$ para G. Si después de haber construido la tabla no existen conflictos en ella, entonces se puede asegurar que la gramática es de tipo $LALR(1)$. Obsérvese que en el paso 3 del algoritmo se construye una nueva colección de conjuntos de items. Esta se denomina colección de conjuntos de items $LALR(1)$.

Ejemplo

Si aplicamos este algoritmo a la gramática 5.6, la colección de items $LALR(1)$ obtenida es la que forma los estados del autómata que aparece en la figura 5.12. La tabla de análisis $LALR(1)$ que se obtiene se muestra en la figura 5.13.

Estado	acción			*ir_a*	
	c	d	$	S	C
0	d36	d47		1	2
1			*aceptar*		
2	d36	d47			5
36	d36	d47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Figura 5.13: Tabla de análisis sintáctico LALR(1) para gramática 5.6

Obsérvese que el tamaño de la tabla ha disminuido considerablemente, al condensar, dos a dos, cada pareja de estados en los cuales había coincidencia del primer componente en todos sus items, en uno solo.

Conflictos LALR

Ya vimos como una gramática podía no ser SLR y por lo tanto el método de construcción de la tabla correspondiente producía conflictos. Ahora vamos a ver que, **si la gramática es LR-canónica, en el método $LALR$ no pueden originarse conflictos del tipo $shift/reduce$ pero si del tipo $reduce/reduce$.**
Supongamos que, para una gramática G, después de la construcción de la colección de conjuntos de items $LR(1)$ para el análisis $LALR$ se tiene un estado I_i con el siguiente contenido:

$[A \to \alpha\bullet, a]$
$[B \to \beta \bullet a\gamma, b]$

Como se ve, el primer item va a generar una entrada reduce, en la columna para a de la tabla de análisis, y el segundo item va a generar un movimiento de desplazamiento para esa misma columna. Tenemos por tanto un conflicto $shift/reduce$. Pues bien, si esta situación se diera, implicaría que algún conjunto $LR(1)$ del análisis LR-canónico de entre los cuales procede el anterior tendría el contenido

$[A \to \alpha\bullet, a]$
$[B \to \beta \bullet a\gamma, c]$

Con b no necesariamente igual que c. Por lo tanto la gramática no sería LR-canóniga.
Si, por otro lado, tuviéramos que

$$[A \to \alpha\bullet, a], [B \to \beta\bullet, b] \in I_i$$

$$[A \to \alpha\bullet, c], [B \to \beta\bullet, a] \in I_j$$

en el análisis LR-canónico, esto implicaría

$$[A \to \alpha\bullet, c/a], [B \to \beta\bullet, a/b] \in I_{i,j}$$

en el análisis $LALR$, por lo tanto pueden originarse conflictos $reduce/reduce$.

Ambigüedad en el análisis LR

Vamos a volver a estudiar el caso del `else` ambiguo. Aparecía en la gramática

prop	→	**if** expr **then** prop
	|	**if** expr **then** prop **else** prop
	|	**otra**

Al estudiar las gramáticas $LL(1)$ se pudo comprobar que esta gramática era ambigua y por lo tanto no era $LL(1)$. Tampoco va a ser LR. De hecho, ninguna gramática ambigua puede ser LR. Para hacer menos engorroso el manejo de los conjuntos de items vamos a cambiar la representación de la gramática. Ahora i va a representar a if expr $then$ y e a `else`. El símbolo a representará al resto de producciones. La gramática queda:

Gramática 5.7

Estado	acción				*ir_a*
	i	e	a	$	S
0	d2		d3		1
1				*aceptar*	
2	d2		d3		4
3		r3		r3	
4		d5		r2	
5	d2		d3		6
6		r1		r1	

Figura 5.14: Tabla de análisis SLR de la gramática 5.7

I_0
$S' \rightarrow \bullet S$
$S \rightarrow \bullet iSeS$
$S \rightarrow \bullet iS$
$S \rightarrow \bullet a$

I_1
$S' \rightarrow S\bullet$

I_2
$S \rightarrow i \bullet SeS$
$S \rightarrow i \bullet SeS$
$S \rightarrow \bullet iSeS$
$S \rightarrow \bullet iS$
$S \rightarrow \bullet a$

I_3
$S \rightarrow \bullet a$

I_4
$S \rightarrow iS \bullet eS$
$S \rightarrow iS\bullet$

I_5
$S \rightarrow iSe \bullet S$
$S \rightarrow \bullet iSeS$
$S \rightarrow \bullet iS$
$S \rightarrow \bullet a$

I_6
$S \rightarrow iSeS\bullet$

Figura 5.15: Conjunto de Items $LR(0)$ para la gramática 5.7

$S' \rightarrow S$
$S \rightarrow iSeS \mid iS \mid a$

Si calculamos la correspondiente colección de items $LR(0)$ para la gramática 5.7, estos aparecen en la figura 5.15. En ella puede verse que en el estado I_4 se produce un conflicto $shift/reduce$. Por lo tanto, cuando en la cabeza de la pila del analizador hay una iS (if expr $then$ prop) y el siguiente símbolo en la entrada es un `else`, ¿reducimos por $S \rightarrow iS\bullet$ ó desplazamos por $S \rightarrow iS \bullet eS$?. Teniendo en mente la interpretación semántica usual, se debería desplazar el `else`, puesto que por convenio se considera cada `else` asociado al `if` más cercano. Por lo tanto, la tabla de análisis sintáctico SLR correspondiente queda como se muestra en la figura 5.14. Para la entrada $w = iiaea$, los movimientos se muestran en la figura 5.16

Recuperación de errores en el análisis LR

Si atendemos al algoritmo 5.7, y a las distintas tablas de análisis generadas según las técnicas SLR, LR-canónico y $LALR$ mediante los algoritmos 5.9, 5.10 y 5.11 respectivamente, los errores en el análisis sintáctico solo se van a detectar cuando se acceda a la parte de *accion* de la correspondiente tabla de análisis sintáctico. La parte de ir_a para los no terminales de la

Pila	Entrada	Acción
0	iiaea$	$shift$
$0i2$	iaea$	$shift$
$0i2i2$	aea$	$shift$
$0i2i2a3$	ea$	reducir $S \rightarrow a$
$0i2i2S4$	ea$	$shift$
$0i2i2S4e5$	a$	$shift$
$0i2i2S4e5a3$	$	reducir $S \rightarrow a$
$0i2i2S4e5S6$	$	reducir $S \rightarrow iSeS$
$0i2S4$	$	reducir $S \rightarrow iS$
$0S1$	$	*aceptar*

Figura 5.16: Movimientos realizados por la máquina SLR para la gramática 5.7 con la entrada $iiaea$

gramática nunca producirá errores.

Dicho de otra forma, nunca se accederá a una celda vacía en las columnas correspondientes a los símbolos no terminales.

Esto es así porque cuando se aplica una reducción mediante una producción (e.g. $A \rightarrow \beta$), tenemos garantía de que el nuevo conjunto de símbolos de la pila (i.e. los anteriores y el nuevo símbolo no terminal, A) forman un prefijo viable. Por tanto, mediante reducciones, podremos obtener el símbolo inicial de la gramática.

En otras palabras, cuando se accede a la casilla $ir\_a[s, A]$ de tabla, después de haber aplicado una reducción, y siendo s el estado que queda en el tope de la pila, encontraremos siempre una transición válida, y no una celda vacía.

Modo Pánico

En modo pánico, la situación es ahora diferente a la que se presentaba en el análisis descendente. En este caso, sabíamos en todo momento cuál era el símbolo no terminal A que había que derivar. Se necesitaba determinar, por tanto, la A-producción a aplicar.

Ahora la situación es distinta; de alguna forma, lo único que conocemos es una parte derecha, y tenemos que encontrar un símbolo no terminal que sea adecuado, para aplicar una reducción y sustituirla por él.

Por lo tanto, es, en cierto modo, arbitrario el símbolo no terminal, digamos A a elegir para simular una reducción.

Pues bien, una vez detectado un error, se van extrayendo símbolos de la pila hasta encontrar un estado s que tenga un valor $ir\_[s, A] = s'$ en la tabla de análisis. Entonces, se comienzan a ignorar símbolos a la entrada hasta que llegue uno considerado de sincronización para A. A continuación, se introducen en la pila los símbolos As' y a partir de aquí se sigue el análisis normalmente.

Nótese que se está suponiendo un intento de A-reducción. Por lo tanto, cuando no se reciben los símbolos que se esperan para formar el prefijo viable adecuado, se intenta simular que ha sido formando un prefijo viable incorrecto (i.e. *falsamente* correcto) que llegará hasta el símbolo situado inmediatamente a la izquierda de A.

Por ejemplo, si retomamos la gramática 5.5, observando su tabla SLR (figura 5.9), la simulación del algoritmo de análisis LR, para la cadena de entrada $id * id($, con la recuperación de errores

en modo pánico, sería como se muestra en la figura 5.17

Pila	Entrada	Acción
0	id*id($	$shift\ 5$
$0id5$	*id($	reducir $F \rightarrow id$
$0F3$	*id($	reducir $T \rightarrow F$
$0T2$	*id($	$shift\ 7$
$0T2*7$	id($	$shift\ 5$
$0T2*7id5$	($	ERROR: Actualizamos pila (eliminar id5) y entrada (eliminar ()
$0T2*7F10$	$	reducir $T \rightarrow T*F$
$0T2$	$	reducir $E \rightarrow T$
$0E1$	$	*aceptar*

Figura 5.17: Movimientos realizados por el analizador SLR (figura 5.9) con la entrada $id*id($

A Nivel de Frase

En la recuperación de errores a nivel de frase se modifica la secuencia de tokens que se reciben desde la entrada para intentar convertirla en legal.

Para esto resulta útil basarse en los errores cometidos por el programador más frecuentemente; además hay que tener en cuenta las particularidades del lenguaje de programación.

Una buena estrategia para llevar a cabo este tipo de recuperación puede ser la siguiente: para cada entrada en blanco de la tabla de análisis, introducir una rutina de manejo del error que realice la acción apropiada para el estado en el que estamos, y el símbolo que se está leyendo. Dentro de las posibles acciones que se pueden realizar, tenemos:

- Inserción, ó eliminación de símbolos en la pila,
- Inserción, o eliminación de símbolos a la entrada y
- Alteración ó transposición de símbolos a la entrada.

Debe evitarse, al extraer símbolos de la pila, sacar de la misma un símbolo no terminal ya que su presencia en ella indica que se había reconocido una estructura con éxito, y en general no interesa deshacer eso.

Sea la gramática cuyo conjunto P está compuesto de las siguientes producciones:

E → E+E
| E*E
| (E)
| id

Sea, para ella la siguiente tabla de análisis sintáctico LR:

Estado	acción						*ir_a*
	i	+	*	(	)	$	S
0	d3	e1	e1	d2	e2	e1	1
1	e3	d4	d5	e3	e2	*aceptar*	
2	d3	e1	e1	d2	e2	e1	6
3	r4	r4	r4	r4	r4	r4	
4	d3	e1	e1	d2	e2	e1	7
5	d3	e1	e1	d2	e2	e1	8
6	e3	d4	d5	e3	d9	e4	
7	r1	r1	d5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

Veamos un ejemplo de comportamiento de una de estas rutinas de error, por ejemplo, la llamada con $e1$:

> En los estados 0, 2, 4 y 5 se espera un operando, es decir, un token del tipo *id* o (. Si en lugar de eso, llega un operador o el símbolo $, estando en uno de esos estados, se llama a la rutina de error $e1$. Esta se encargará de introducir un *id* imaginario en la pila y cubrirlo con el estado 3. Además, se emite el mensaje de error "falta operando".

Bibliografía

Consideramos de interés general los libros que a continuación se detallan, aunque hemos de decir que el temario propuesto no sigue ‘ ‘al pie de la letra” ningún texto en concreto. Pero digamos que con estos libros se cubren todos los conceptos, definiciones, algoritmos, teoremas y demostraciones que se exponen en los contenidos teóricos. Destacamos con una **B** los libros básicos para el alumno y con una **C** los libros complementarios de interés para ciertas cuestiones teóricas y por los ejercicios propuestos. Los demás son más bien de interés para el profesor.

C **[Aho72]** A. Aho, J. Ullman. *The Theory of Parsing, Translation and Compiling, Vol. I.* Prentice-Hall, 1972.

C **[Alf97]** M. Alfonseca, J. Sancho, M. Martínez. *Teoría de Lenguajes, Gramáticas y Autómatas.* Publicaciones R.A.E.C., 1997.

[Bro93] J. Brookshear. *Teoría de la Computación.* Addison-Wesley, 1993.

[Car89] J. Carroll, D. Long. *Theory of Finite Automata with an Introduction to Formal Languages.* Prentice Hall, 1989.

[Coh86] D.I.A. Cohen. *Introduction to Computer Theory.* John Wiley & Sons, 1991.

[Dav94] M.D. Davis, R. Sigal, E.J. Weyuker. *Computability, Complexity and Languages: Fundamentals of Theoretical Computer Science.* Academic Press, 1994.

[Flo94] R. Floyd, R. Beigel. *The Language of Machines.* Computer Science Press, 1994.

[Gar79] M. Garey, D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1979.

[Her84] H. Hermes. *Introducción a la Teoría de la Computabilidad.* Tecnos, 1984.

[Hop79] J.E. Hopcroft, D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

B **[Hop02]** J.E. Hopcroft, R. Motwani, D. Ullman. *Introducción a la Teoría de Autómatas, Lenguajes y Computación.* Addison-Wesley, 2002.

[Hor76] J.J. Horning, *What The Compiler Should Tell the User, Compiler Construction: An Advanced Course, 2d ed.*, New York: Springer-Verlag, 1976.

B **[Isa97]** P. Isasi, P. Martínez, D. Borrajo *Lenguajes, Gramáticas y Autómatas. Un enfoque práctico.* Addison-Wesley, 1997.

B **[Kel95]** D. Kelley. *Teoría de Autómatas y Lenguajes Formales.* Prentice Hall, 1995.

C **[Koz97]** D.C. Kozen. *Automata and Computability.* Springer, 1997.

C **[Lew81]** H. Lewis, C. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, 1981.

B **[Lin97]** P. Linz. *An Introduction to Formal Languages and Automata*. Jones and Barlett Publishers, 1997.

[Min67] M. Minsky. *Computation: Finite and Infinite Machines*. Prentice Hall, 1967.

[Mol88] R.N. Moll, M.A. Arbib. *An Introduction to Formal Language Theory*. Springer-Verlag, 1988.

[Rev83] G.E. Révész. *Introduction to Formal Languages*. Dover Publications, 1983.

[Sal73] A. Salomaa. *Formal Languages*. Academic Press, 1973.

[Sal85] A. Salomaa. *Computation and Automata*. Cambridge University Press, 1985.

[Sud91] T.A. Sudkamp. *Languages and Machines*. Addison-Wesley, 1988.

[Tre85] J. P. Tremblay, P. G. Sorenson, *The theory and practice of compiler writing, McGraw-Hill International*, 1985.

[Woo87] D. Wood. *Theory of Computation*. John Wiley & Sons, 1987.

Índice general