

ASPECTOS FORMALES DE LOS LENGUAJES DE PROGRAMACIÓN

MATERIAL COMPLEMENTARIO

Basado en "Programming Languages. Design and Implementation", T. Pratt - M. Zelkowitz. Edición 4, 2001 y en "Introduction to Automata Theory, Languages and Computation", J. Hopcroft, R. Motwani y J.

Ullman, Edición 3, 2006

1. Introducción

La tarea de proveer descripciones entendibles, pero concisas, de los lenguajes de programación es una tarea compleja, pero esencial para el éxito de los mismos. Uno de los principales problemas para describir lenguajes, es la diversidad de personas que deben entender esa descripción.

Los implementadores de los lenguajes de programación deben ser capaces de determinar cómo las expresiones, las sentencias y unidades de programas de un lenguaje están formadas, como así también cual será su efecto cuando sean ejecutadas.

La implementación de lenguajes de programación se puede realizar a través de la compilación, la interpretación o de sistemas híbridos, como ya hemos estudiado. La compilación utiliza un programa llamado *compilador*, el cual traduce programas escritos en lenguajes de alto nivel en código de máquina, tal es el caso de lenguajes como Pascal, C, Ada (básicamente en sus primeras versiones), etc. La interpretación pura no lleva a cabo traducción, los programas son interpretados en su forma original por un intérprete del software, como sucede en lenguajes como LISP, Prolog. Los sistemas híbridos traducen programas escritos en lenguajes de alto nivel en formas intermedias, que luego se interpretan. Por ejemplo, varias versiones de JAVA, cuentan con un compilador de JAVA, que traduce los fuentes a bytecode, y luego el intérprete de JAVA interpreta bytecode.

En los primeros tiempos del diseño de lenguajes de programación (es decir, en la década del 50 y 60, durante el desarrollo de FORTRAN, ALGOL, COBOL, LISP), se creía que sólo era necesario contar con una sintaxis formal para especificar programas. Pero en la actualidad sabemos que, en realidad, el estudio de los lenguajes de programación, al igual que el estudio de los lenguajes naturales requieren del estudio del análisis de la *sintaxis* y la *semántica*.

- **Sintaxis:** se puede definir como *la disposición de palabras como elementos en una oración para mostrar su relación*. En el contexto de los lenguajes de programación, es la forma que tienen los elementos básicos del programa (identificadores, palabras claves, etc), expresiones, sentencias y unidades del programa (subprogramas, funciones). La sentencia en C, $A=B+C$ representa una serie válida de símbolos, pero $BC+-$ no representa una serie válida de símbolos en este lenguaje.

La sintaxis provee información significativa que se necesita para entender un programa, y además suministra información imprescindible para la traducción del programa fuente a un programa objeto.

- **Semántica:** es el significado que tienen los elementos básicos del programa, expresiones, sentencias y unidades del programa.

Por ejemplo, la sintaxis (es decir como se escribe) de una sentencia **while** en Java es:

```
while (<expr_booleana>) <sentencia>
```

La semántica de esta sentencia es que, mientras el valor de la expresión booleana sea *verdadero*, la sentencia embebida será ejecutada.

Aunque estos conceptos son a menudo separados con la finalidad de definirlos, la sintaxis y la semántica están estrechamente relacionadas. La descripción y entendimiento completo de un lenguaje requiere de ambos aspectos. En un lenguaje bien diseñado, la semántica debería surgir directamente desde la sintaxis.

Describir sintaxis, en general, es una tarea más fácil que describir semántica, particularmente porque se cuenta con una notación concisa y universalmente aceptada para descripción de sintaxis, pero no se ha desarrollado ninguna aún para describir semántica.

2. Criterios Sintácticos Generales

El propósito principal de la sintaxis es proveer una notación para la comunicación entre el programador y el procesador del lenguaje de programación. Los detalles de sintaxis se eligen en general basados en criterios secundarios, con objetivos generales de hacer que los programas sean fáciles de leer, fáciles de escribir, fáciles de traducir y no ambiguos.

- **Legibilidad:** como ya vimos previamente, un programa es legible en la medida que es autodocumentado, es decir que es legible sin documentación complementaria.

Factores que afectan en forma positiva: la legibilidad se mejora a través de características del lenguaje tales como formato de sentencias naturales, sentencias estructuradas, identificadores de longitud sin restricción, uso abundante de palabras claves, palabras opcionales, recursos para uso de comentarios, símbolos nemotécnicos de operadores, etc. La legibilidad mejora a través de una sintaxis de programa en la cual las diferencias sintácticas reflejan diferencias semánticas, de manera que las construcciones sintácticas que hacen cosas similares se ven parecidas.

Factores que afectan en forma negativa: cuando los lenguajes proveen pocas construcciones sintácticas, en general, conducen a programas menos legibles, por ejemplo, APL y SNOBOL son lenguajes que proveen un formato de sentencias específico. Cuando las diferencias entre una sentencia goto, una bifurcación condicional de múltiples líneas y otras estructuras comunes de programas se reflejan sólo por diferencias en unos pocos símbolos, en estos casos suele ser necesario un análisis minucioso del programa para lograr su entendimiento. Además un simple error, como por ejemplo un carácter incorrecto, puede alterar en forma radical el significado del enunciado sin hacerlo sintácticamente incorrecto.

- **Facilidad para escribir programas:** también analizamos, anteriormente, este aspecto, como dijimos se buscan estructuras sintácticas concisas, regulares y poderosas, mientras que en el criterio anterior se buscan construcciones más “verborrágicas”. Pero, en general, las características sintácticas que hacen que un programa sea fácil de escribir suelen encontrarse en conflicto con las características que facilitan su lectura. Por ejemplo, el lenguaje C, provee recursos para programas muy concisos, por ejemplo el uso de la versión reducida de la sentencia condicional if, el operador ++, etc., aunque también cuenta con otras características útiles.

Factores que afectan en forma positiva: convenciones sintácticas implícitas que permiten dejar sin especificar declaraciones y operaciones, hacen a los programas más cortos.

Otros factores que favorecen ambos objetivos, legibilidad y facilidad de escritura, son por ejemplo el uso de sentencias estructuradas, formatos simple de enunciados naturales, símbolos nemotécnicos de operaciones.

- **Facilidad de Traducción e Interpretación:** un aspecto importante es el de hacer que los programas sean fáciles de traducir a una forma ejecutable. La legibilidad y la facilidad de escritura son características dirigidas a las necesidades de los programadores, la facilidad de traducción se relaciona con las necesidades del traductor que procesa el programa escrito en algún lenguaje de alto nivel. La sintaxis del lenguaje LISP proporciona un ejemplo de una estructura de programa que no es particularmente legible ni fácil de escribir, pero que resulta muy fácil de interpretar. Cuanto más regular es la estructura de un programa, más simple es de traducir, pero cuantas más construcciones sintácticas posee el lenguaje, es más difícil de traducir los programas, este es el caso de COBOL.
- **Facilidad de Verificación:** después de muchos años de experiencia, se puede concluir que entender cada sentencia de un lenguaje de programación es relativamente sencillo, el proceso global de crear programas correctos es en muchos casos difícil, por lo tanto se requieren técnicas que permitan determinar que un programa sea matemáticamente correcto.
- **Falta de ambigüedad:** la ambigüedad es un problema importante en todo diseño de lenguajes. La definición de un lenguaje debería proveer un único significado para toda construcción sintáctica. Una construcción ambigua permite dos o más interpretaciones diferentes.

Por ejemplo la sentencia `if` del lenguaje C admite dos formas distintas:

```
if (<expresion>) <sentencia> else <sentencia>
```

```
if (<expresion>) <sentencia>
```

La interpretación que se debe hacer de cada enunciado está claramente definida, sin embargo, cuando se combinan ambas sentencias se puede dar origen a lo que se denomina *else* ambiguo:

```
if (<expresion>) if (<expresion>) <sentencia> else <sentencia>
```

Esta forma es ambigua, dado que no queda claro, a simple vista, con quien se corresponde el *else*, si con el primero o el segundo *if*. En este caso el usuario del lenguaje deberá tener claro que para evitar dicha ambigüedad el lenguaje establece que el **else** corresponde al **if** más cercano.

3. Elementos sintácticos de un lenguaje

Son los factores sintácticos básicos que afectan el estilo sintáctico general.

- **Conjunto de caracteres:** un programa es esencialmente una secuencia de caracteres, que conforman palabras escritas a partir de un lenguaje, entonces la primera decisión es cómo representar esos caracteres. El conjunto de caracteres más conocido es el ASCII, el UNICODE, que es con el que trabaja el lenguaje Java.
- **Identificadores:** la sintaxis básica para identificadores, una cadena de letras y dígitos comenzando con letra, es ampliamente aceptada. Aunque lenguajes como Fortran, en sus comienzos, restringían a 6 el número máximo de caracteres, esto puede reducir la legibilidad.

- **Símbolos de operadores:** las operaciones primitivas pueden ser representadas por:

Símbolos especiales: +, -, *, /, etc.

Identificadores, para todas las primitivas como sucede en Lisp.

- **Palabras claves y palabras reservadas:** una palabra clave es un identificador que se usa como una parte fija de la sintaxis de una sentencia. Una palabra clave es reservada si, no puede también ser usada como un identificador elegido por el programador.

La mayoría de los lenguajes de programación utilizan palabras reservadas, esto ayuda en el proceso de traducción, especialmente en la detección de errores. El lenguaje Fortran utiliza palabras claves.

- **Palabras opcionales:** son palabras que son insertadas en sentencias para mejorar la legibilidad. Por ejemplo Cobol provee tales opciones, la sentencia `GO TO`, requiere la palabra clave `GO`, pero el `TO` es opcional.
- **Comentarios:** la inclusión de comentarios en un programa es una parte importante de su documentación. Existen diferentes formas de incluir comentarios, por ejemplo, líneas de comentarios separadas en el programa, delimitados por símbolos especiales o delimitadores (`/*` y `*/` en C), comenzando en cualquier lugar en la línea pero que termina en el fin de línea (`//` en C++, `-` en Ada, etc.).
- **Blancos:** las reglas sobre el uso de los blancos varían ampliamente entre los lenguajes. Es decir, los blancos pueden ser significativos o no. En C, por ejemplo los blancos no son significativos (salvo en las cadenas de caracteres), es decir que si están se descartan, no se necesitan para separar unidades sintácticas entre sí, porque el lenguaje provee separadores. En cambio hay lenguajes como Snobol que usan los blancos como separadores, pero esto obviamente conduce a mucha confusión.
- **Delimitadores y corchetes:** los delimitadores marcan el comienzo o fin de alguna unidad sintáctica o expresión. Los corchetes son delimitadores apareados. Tienen como finalidad eliminar ambigüedades, definiendo los límites de una construcción sintáctica particular, por ejemplo: `begin end` de Pascal, las llaves (`{}`), los paréntesis, etc.
- **Formatos de campo fijo o libre:**

Formato libre: las sentencias van en cualquier posición en las líneas de entrada sin importar cortes de línea.

Formato fijo: la posición dentro de la línea de entrada es importante. Por ejemplo Fortran reserva los cinco primeros caracteres de cada línea para un rótulo de enunciado.
- **Expresiones:** construcciones sintácticas que acceden a objetos de datos y retornan un valor (son funciones). En C, por ejemplo, constituyen las operaciones básicas, en Lisp las expresiones forman el control básico de secuencia que controla la ejecución del programa.
- **Sentencias:** componente sintáctica principal en los lenguajes imperativos, la clase dominante de lenguajes en uso en la actualidad.

Sentencias simples, no admiten otros enunciados (sentencias) dentro.

Sentencias estructuradas, pueden estar compuestas de otras sentencias, simples o estructuradas.

4. Estructura Programa-Subprograma Completo

La organización sintáctica completa del programa principal y las definiciones de subprogramas es tan variada como los otros aspectos de la sintaxis de los lenguajes de programación.

- **Definiciones de subprogramas separados:** cada subprograma es tratado como una unidad sintáctica separada. Por ejemplo C, mantiene una organización en la cual cada definición de subprograma es tratada como una unidad sintáctica separada. Cada subprograma se compila de manera separada y los programas compilados son linkeados en tiempo de carga.
- **Definiciones de datos separados:** agrupar todas las operaciones que manipulan un objeto de datos dado. Este es el propósito general del mecanismo de **clases** en lenguajes como Java, C++, Smalltalk.
- **Definiciones de subprogramas anidados:** posibilidad de poder definir subprogramas dentro del programa principal, y que a su vez dentro de éstos se puedan definir otros hasta cualquier profundidad. El objetivo es proveer un ambiente de referenciación no local para subprogramas, tal que permite chequeo de tipo estático y compilación de código ejecutable eficiente para subprogramas que contienen referencias no locales.
- **Definiciones de interfaces separadas:** cómo manejar la invocación a subprogramas para chequeo de tipos sin necesidad de recompilación total, cosa que sucede por ejemplo en Pascal, donde el programa completo debe ser recompilado, aún si tiene cientos o miles de sentencias, cada vez que una sentencia es cambiada. En cambio, C incluye ciertas operaciones de archivos de sistema que permiten al programa incluir archivos que contienen estas definiciones de interface. Los archivos `.h` forman las componentes de especificación y los archivos `.c` las componentes de implementación. Ada con el uso de los `packages` provee esta característica directamente en el lenguaje.
- **Descripciones de datos separados de sentencias ejecutables:** agrupar todos los datos (data divisions) y los subprogramas separados (procedure divisions). Todos los datos son globales a todos los subprogramas, no hay datos locales. Esta característica aparece en el lenguaje Cobol.
- **Definiciones de subprogramas no separados:** un programa es una lista de sentencias, y los puntos donde un subprograma comienza y finaliza no son diferenciados sintácticamente, es decir no presentan construcciones particulares que identifican el comienzo de un subprograma y su invocación. Este es el caso de lenguajes como Basic, Snobol 4.

5. Etapas de la Traducción

El proceso de traducir un programa escrito en su sintaxis original, en la forma ejecutable, es fundamental en toda implementación de lenguajes de programación.

Lógicamente se puede dividir la traducción en dos partes principales: **el análisis** del programa fuente dado como entrada y **la síntesis** del programa objeto ejecutable.

Dentro de cada una de estas partes hay a su vez divisiones, comúnmente denominadas fases, ver la figura 1 que ilustra la estructura de un compilador típico.

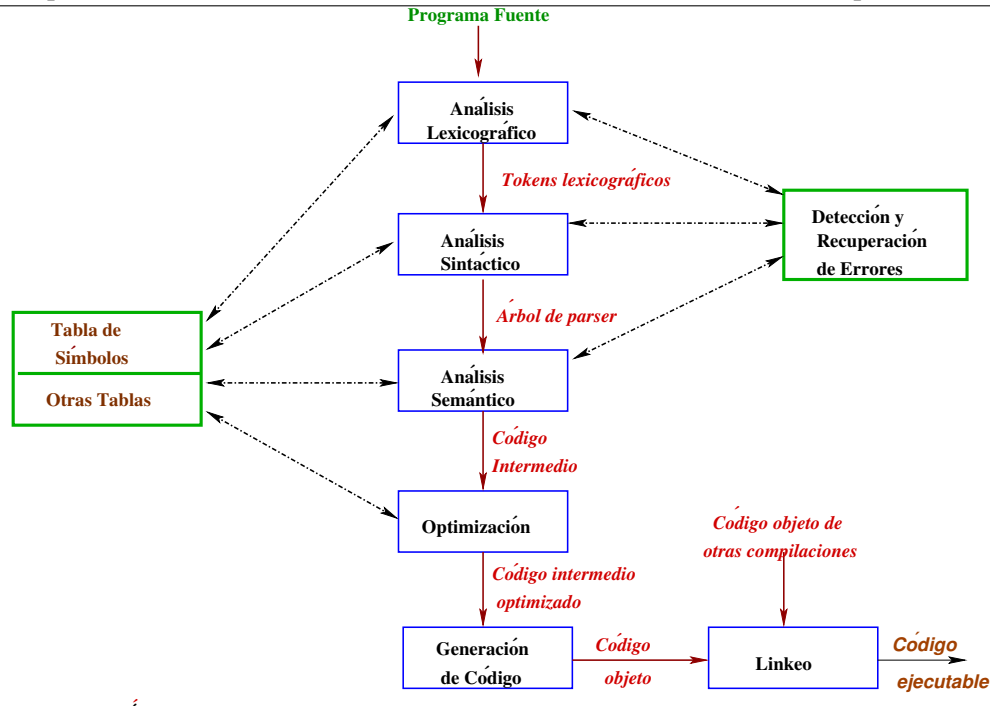


Figura 1: Etapas de la Traducción.

5.1. Módulos de un Compilador

Los traductores generalmente se agrupan de acuerdo al número de **pasadas** que hacen sobre el programa fuente.

El compilador más común hace dos pasadas sobre el programa fuente. La primera fase del análisis descompone el programa en sus componentes constituyentes y obtiene información, como por ejemplo el uso de un nombre de variable en el programa. La segunda fase genera un programa objeto a partir de la información recolectada.

Si la velocidad de compilación es importante, se puede usar la estrategia de una pasada que es lo que hace el compilador Pascal.

5.2. Análisis del Programa Fuente

Para un traductor el programa fuente es una secuencia de cientos, miles o más caracteres no diferenciados.

El análisis de la estructura del programa se construye caracter a caracter durante la traducción.

- **Análisis lexicográfico:** la fase básica del proceso de traducción es agrupar esta secuencia de caracteres en sus componentes elementales: identificadores, delimitadores, símbolos de operadores, números, palabras claves, blancos, comentarios, etc.

Esta fase se denomina análisis lexicográfico (o **scanner**) y los elementos básicos del programa que resultan del análisis se llaman **tokens** (o ítems lexicográficos).

Para diseñar analizadores lexicográficos se utilizan habitualmente **expresiones regulares** o **autómatas finitos**.

- **Análisis Sintáctico:** aquí se identifican las estructuras del programa (sentencias, expresiones, etc.) usando los ítems producidos por el analizador lexicográfico.

El análisis sintáctico interactúa con el análisis semántico, se encarga de identificar una secuencia de ítems lexicográficos formando una unidad sintáctica y luego el analizador semántico es llamado para procesar esta unidad.

- **Análisis Semántico:** aquí las estructuras sintácticas reconocidas por el analizador sintáctico son procesadas, y la estructura del código objeto ejecutable comienza a tomar forma.

El analizador semántico puede producir directamente el código objeto ejecutable, pero lo más común es que la salida de esta fase sea alguna forma intermedia del programa ejecutable final, el cual es manipulado por la fase de optimización del traductor, antes que el código ejecutable sea realmente generado. Completa el análisis sintáctico y es el encargado principal de recolectar información sobre los tipos y de realizar la verificación de tipos.

Las funciones del analizador semántico pueden variar, dependiendo del lenguaje y la organización lógica del traductor. Algunas de las funciones más comunes pueden ser:

Mantenimiento de la Tabla de Símbolos.

Inserción de información implícita.

Detección de Errores.

Operaciones de tiempo de compilación y macro procesamiento.

Las fases finales de la traducción tienen que ver con la construcción del programa ejecutable a partir de la salida producida por el analizador semántico. Esta fase involucra la generación de código y puede también incluir optimización del programa generado. Si los subprogramas son traducidos en forma separada, o si subprogramas de librerías son usados, una fase de linkeo y carga es necesaria para producir el programa completo ejecutable.

6. Modelos de Traducción Formales

Existen dos tipos básicos y reconocidos de lenguajes: los lenguajes naturales y los lenguajes formales. El origen y desarrollo de los primeros, como pueden ser el castellano, el inglés, el portugués, es natural, es decir, sin el control de ninguna teoría. Sin embargo, los lenguajes formales, como las matemáticas y la lógica, se desarrollaron, en general, a través del establecimiento de una teoría.

Las lenguas son sistemas más o menos complejos, que asocian contenidos de pensamiento y significación a manifestaciones simbólicas tanto orales como escritas. Aunque, hablando

estrictamente, el lenguaje representa la capacidad humana para comunicarse mediante lenguas, también se utiliza este término para denotar los mecanismos de comunicación no humanos (el lenguaje de las abejas o el de los delfines), o los creados por los hombres con fines específicos (los lenguajes de programación, los lenguajes de la lógica, los de la aritmética, etc.).

Un lenguaje, ya sea natural (como el castellano) o artificial (como Java), es un conjunto de cadenas de caracteres a partir de alguna colección finita de símbolos. En el caso de cualquier lenguaje natural, la colección finita es el conjunto de letras del alfabeto, junto con los símbolos que se utilizan para construir palabras (tales como el guión, el punto, etc.). De manera similar la representación de enteros, son secuencias de caracteres del conjunto de los dígitos $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

Un lenguaje natural se puede considerar como un conjunto de oraciones, que generalmente es infinito, y se forman con palabras obtenidas a partir del alfabeto. Una oración, en castellano, es una secuencia finita de palabras del castellano, donde sabemos que el conjunto de esas palabras es finito. Sin embargo no todas las combinaciones de palabras son permitidas, es necesario que esas combinaciones sean correctas (con respecto a la sintaxis) y tengan sentido (con respecto a la semántica).

Contrariamente a lo que ciertas teorías lingüísticas formales harían a uno creer, el lenguaje natural no fue fundamentado sobre una verdad racional a priori, pero fue desarrollado y organizado a partir de la experiencia humana en el mismo proceso en que esta experiencia humana fue organizada. En su forma actual, los lenguajes naturales tienen un gran poder expresivo y pueden ser utilizados para analizar situaciones altamente complejas y razonar sutilmente. La riqueza de la semántica y su cerrada relación con los aspectos prácticos de los contextos en los que son usados, dá a los lenguajes naturales, su gran poder expresivo.

Así como la formalización de la semántica de un lenguaje natural, es decir el cómo determinar si las oraciones tienen significado, es bastante difícil, por otro lado, la sintaxis de un lenguaje natural puede ser modelada fácilmente por un lenguaje formal similar a los utilizados en matemáticas, lógica o computación.

La definición de una teoría de un lenguaje formal dado precedió a su definición intensiva, es decir, que se definen unívocamente las oraciones correctas que componen un lenguaje. En computación, por ejemplo, hablamos de programas correctos.

Por lo tanto a partir de ahora es nuestro objetivo estudiar métodos que permitan la descripción de lenguajes formales, es decir la sintaxis. En particular nuestro interés es el de estudiar cómo se definen los lenguajes de programación.

6.1. Definiciones previas

Símbolo: un **símbolo** es una entidad abstracta a la que no definiremos formalmente, como ocurre con el punto en geometría. Letras y dígitos son ejemplos de símbolos frecuentemente usados. Por ejemplo 0, 1 y a son símbolos.

Alfabetos: un **alfabeto** es un conjunto finito, no vacío de símbolos. Se utiliza comúnmente el símbolo Σ para denotarlo. Por ejemplo, son alfabetos:

1. $\Sigma = \{0, 1\}$, el alfabeto binario.
2. $\Sigma = \{a, b, c, d, \dots, z\}$, el alfabeto de las letras minúsculas.
3. El conjunto de todos los caracteres ASCII.

Cadenas: una **cadena** o **palabra** es una secuencia finita de símbolos elegida desde un alfabeto. Por ejemplo, 001, 011110, 11111111, etc. son cadenas obtenidas a partir del alfabeto $\Sigma = \{0, 1\}$, concatenando los símbolos.

Palabra vacía: la **cadena vacía**, que se denota con λ , representa la cadena de longitud nula.

Longitud de cadenas: a menudo es útil clasificar las cadenas por su longitud, es decir, el número de posiciones para símbolos en la cadena. Por ejemplo, 11001 tiene longitud 5, aaaaaaaaaa tiene longitud 9, etc.

Si Σ es un alfabeto, se puede expresar el conjunto de todas las cadenas de cierta longitud, a partir de dicho alfabeto, usando notación exponencial. Se define Σ^k , como el conjunto de todas las cadenas de longitud k , cuyos símbolos están en Σ .

Ejemplo: notar que $\Sigma^0 = \{\lambda\}$, independientemente de Σ . Es decir, λ es la única cadena de longitud 0. Si $\Sigma = \{a, b\}$, entonces $\Sigma^1 = \{a, b\}$, $\Sigma^2 = \{aa, ab, ba, bb\}$, y así sucesivamente.

Definición: para cualquier alfabeto Σ , el conjunto de todas las posibles cadenas sobre Σ es denotado por Σ^* (lenguaje referencial).

Ejemplo: Sea $\Sigma = \{a, b\}$, luego

$$\Sigma^* = \{a, b\}^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots = \bigcup_{k=0}^{\infty} \Sigma^k = \{\lambda, a, b, aa, ab, ba, bb, aaa, aab, \dots\}$$

Concatenación de cadenas: La operación de concatenación (\cdot), es una operación binaria entre cadenas de Σ^* , es decir:

$$\cdot : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$$

Ejemplo: sean $x, y \in \Sigma^*$ la concatenación de x e y se denota $x.y$ o simplemente xy . Si $x = a_1a_2..a_n$ y $y = b_1b_2..b_m$ entonces:

$$xy = a_1a_2..a_nb_1b_2..b_m$$

Observación: la cadena vacía λ es el elemento neutro de la operación de concatenación:

$$\forall x \in \Sigma^*, \lambda x = x\lambda = x$$

Potencia de cadenas: se puede definir de manera recursiva de la siguiente manera, si $x \in \Sigma^*$ para $n \in \mathbb{N}$:

$$x^n = \begin{cases} \lambda & \text{si } n = 0, \\ xx^{n-1} & \text{si } n \geq 1 \end{cases}$$

Ejemplo: sea $\Sigma = \{a, b\}$,
 $(aab)^0 = \lambda$, $(aab)^1 = aab$
 $(aab)^5 = aabaabaabaabaab$

Definición de Lenguaje: se define como **lenguaje**, al conjunto de **cadenas** formadas a partir de un determinado **alfabeto**, es decir, un **lenguaje** sobre Σ es un subconjunto de Σ^* .

Importante: Los lenguajes pueden ser expresados como los conjuntos, por extensión (cuando son finitos), por comprensión, la manera de hacerlo es especificando una propiedad que deben satisfacer las cadenas del lenguaje. Es también común utilizar expresiones con parámetros, y describir las cadenas del lenguaje estableciendo condiciones sobre los parámetros.

Ejemplo: sea $\Sigma = \{0, 1\}$ un alfabeto, se pueden definir, como ejemplos, los siguientes lenguajes a partir de él:

1. El lenguaje formado por todas las cadenas de longitud impar. Es decir, que las cadenas 0, 011, 000, 01010, 00111, 10101, 11011, ..., pertenecen a este lenguaje.
2. $L = \{w \in \Sigma^* / w \text{ empieza con un } 1\}$, este lenguaje incluye todas las cadenas (w) que empiezan con un 1. Es decir, que las cadenas 1, 11, 10, 101, 110, 111, 1000, 1101, 1100, ..., pertenecen a este lenguaje.
3. $L_1 = \{0^n 1^{2n} / n \geq 1\}$, este lenguaje incluye todas las cadenas de n 0's seguidos de $2n$ 1's. Es decir, que las cadenas 011, 001111, 000111111, 00001111111, ..., pertenecen a este lenguaje.

7. Dispositivos descriptores de lenguajes

En general, los lenguajes pueden ser definidos formalmente por medio de **dispositivos generadores** o por medio de **dispositivos reconocedores**.

- **Dispositivos generadores - Gramática:** Es un modelo matemático que permite generar a través de reglas sintácticas o gramaticales, cadenas miembros de un lenguaje específico.

Una gramática consiste de un conjunto de reglas (también llamadas producciones) que permiten especificar las secuencias de caracteres válidos en el lenguaje que se está definiendo. En el contexto de los lenguajes de programación estas secuencias de caracteres válidos, corresponden a programas válidos en el lenguaje de programación que se está definiendo.

A mediados de los años 50, un lingüista llamado Noam Chomsky, describió 4 dispositivos generadores o gramáticas que definen cuatro tipos de lenguajes diferentes: *lenguajes regulares*, *lenguajes libres del contexto*, *lenguajes dependientes del contexto*, *lenguajes*

irrestringidos. De estas gramáticas, las libres del contexto y las regulares son las que permiten describir la sintaxis de los lenguajes de programación, abordaremos el estudio de estas gramáticas en la Sección 9 de este apunte,

- **Dispositivos reconocedores - Autómata o Máquina Abstracta:** Es un modelo matemático que representa la idea de computación o manipulación de cadenas vía la aplicación de acciones preestablecidas. Tiene como objetivo, en general, determinar la pertenencia de una cadena a un lenguaje específico.

7.1. Otra forma de describir lenguajes regulares - Expresiones Regulares

Las Expresiones Regulares (ER) son otro tipo de notación simple, que se utiliza para definir lenguajes, en particular ciertos tipos de lenguajes, los **lenguajes regulares**. Ofrecen una forma declarativa de expresar las cadenas que pertenecen a un lenguaje.

Las ER se construyen utilizando los siguientes elementos:

- **operandos:**

los símbolos σ en Σ

la palabra vacía (λ)

el conjunto vacío (\emptyset)

expresiones regulares

- **operadores:**

unión (+)

concatenación (.)

clausura de Kleene ()*

Definición: las expresiones regulares se definen recursivamente como:

- \emptyset es una ER y denota el conjunto vacío, $L(\emptyset) = \emptyset$.
- λ es una ER y denota el conjunto $\{\lambda\}$. $L(\lambda) = \{\lambda\}$.
- $\forall a \in \Sigma$, a es una ER y denota el conjunto $\{a\}$. $L(a) = \{a\}$.
- Si α y β son ER, entonces:
 - $\alpha + \beta$ es una ER y denota el conjunto $L(\alpha + \beta) = L(\alpha) \cup L(\beta)$
 - $\alpha\beta$ es una ER y denota el conjunto $L(\alpha\beta) = L(\alpha)L(\beta)$
 - α^* es una ER y denota el conjunto $L(\alpha^*) = (L(\alpha))^*$
 - (α) es una ER y denota el conjunto $L((\alpha)) = L(\alpha)$
- Ninguna otra cosa es una ER.

La **Precedencia de los operadores (de mayor a menor)** es: clausura, concatenación y unión.
La **Asociatividad** es: de izquierda a derecha.

Ejemplos:

Considerando $\Sigma = \{a, b\}$

- a es una ER que denota el lenguaje $L = \{a\}$
- b es una ER que denota el lenguaje $L = \{b\}$
- $a + b$ es una ER que denota el lenguaje $L = \{a\} \cup \{b\} = \{a, b\}$
- ab es una ER que denota el lenguaje $L = \{a\} \cdot \{b\} = \{ab\}$
- $(ab)^*$ es una ER que denota el lenguaje $L = \{ab\}^*$
- $(a + b)(ab)^*$ es una ER que denota el lenguaje $L = \{a, b\}\{ab\}^*$

Se deja como ejercicio dar ejemplos de cadenas que pertenecen a este último lenguaje.

Ejercicios:

Considerando $\Sigma = \{a, b\}$, construir una ER para cada uno de los primeros 3 lenguajes:

1. $L_1 = \{\lambda, aab, baa\}$
2. $L_2 = \{x \in \Sigma^* / x \text{ comienza con } b \text{ y termina con } a\}$
3. $L_3 = \{x \in \Sigma^* / x \text{ contiene la subcadena } ab\}$
4. $L_4 = \{x \in \Sigma^* / x \text{ no contiene 3 b's seguidas}\}$

La siguiente podría ser una ER que denota el lenguaje dado, $\Rightarrow (a + ba + bba)^*$, pero ¿es correcta?

Si analizamos en detalle la ER, vemos que no permite obtener cadenas que finalicen con b , por lo tanto debemos obtener otra ER que efectivamente denote todas las cadenas del lenguaje dado, una solución posible puede ser: $\Rightarrow (a + ba + bba)^*(\lambda + b + bb)$.

7.2. Usos de las ER

Las expresiones regulares se utilizan para:

- Intuitivamente: la localización de una serie de caracteres dentro de (por ejemplo) un texto se pueden realizar mediante una búsqueda por unificación basada en patrones.
- Los patrones para realizar una búsqueda por unificación son especificados mediante ER.

Ejemplos:

Extraer e-mails de un informe.

Encontrar una palabra o secuencia de caracteres dentro de un texto

Conocer cuántas páginas de un sitio web incluyen una o varias palabras determinadas en su URL.

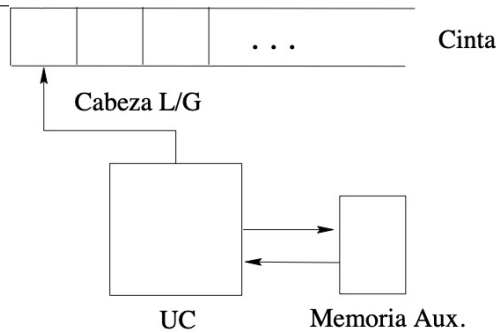


Figura 2: Autómata.

8. Autómatas

En principio, un modelo general de autómata puede ser visualizado como una máquina que recibe como entrada una cadena $x \in \Sigma^*$ y emite como salida una respuesta indicando si la cadena fue reconocida (aceptada) o rechazada.

El esquema general de un autómata puede verse en la figura 2.

Componentes:

- **Cinta de Entrada/Salida:** está dividida en celdas capaces de almacenar un símbolo por vez, la misma puede ser **finita o infinita**. Tales símbolos pertenecen a un conjunto denominado **alfabeto de la cinta**.
- **Cabeza Lectora/Grabadora:** permite leer o grabar símbolos en la cinta. Puede realizar movimientos de a una celda a la derecha o una a la izquierda, desde la posición corriente.
- **Unidad de Control:** Es un conjunto finito de posibles estados de la máquina, que indican en cada momento en qué estado está la máquina. Es una componente importante en la toma de decisiones, pero no la única.
- **Memoria Auxiliar:** Permite extender la capacidad de cierto tipo de máquinas.

8.1. Autómata Finito Determinístico

Un autómata finito es un modelo matemático de un sistema con un número finito de estados.

Un autómata finito (AF) puede ser visto como un dispositivo que lee una cinta de entrada, que comienza en un cierto estado inicial (en general q_0), cada vez que lee un símbolo de la entrada se produce una transición de estado (a algún estado q_i), ver la figura 3. Una cadena es aceptada si el autómata queda en un estado de aceptación o final.

Definición: Un Autómata Finito Determinístico (AFD) es una 5-tupla $M = (Q, \Sigma, \delta, q_0, F)$, donde Q es un conjunto finito de estados, Σ el alfabeto de entrada, $q_0 \in Q$ el estado inicial, $F \subseteq Q$ es el conjunto de estados finales.

La función de transición, $\delta : Q \times \Sigma \rightarrow Q$, es una **función total**. ¿Qué significa?

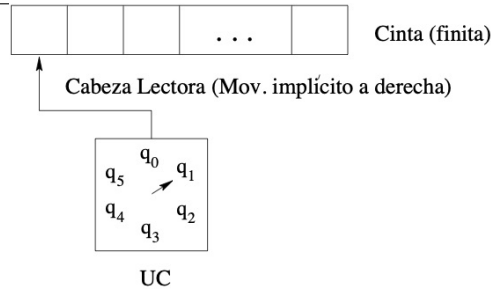


Figura 3: Autómata Finito.

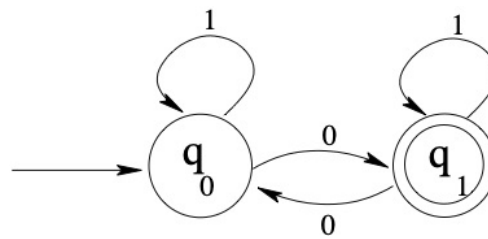


Figura 4: Autómata Finito Determinístico.

Los autómatas se representan mediante un grafo dirigido, o diagrama de transición donde uno de los nodos es el estado inicial, y existen 1 o más estados finales. Los arcos son rotulados con los símbolos del alfabeto.

Ejemplo 1: ver en la figura 4 un AFD que reconoce el siguiente lenguaje:

$$L_1 = \{x \in \{0, 1\}^* | n_0(x) \text{ es impar y mayor que } 0\}$$

donde:

$$n_0(x) \text{ es la cantidad de } 0's \text{ en } x$$

¿Cómo trabaja el autómata para una entrada $w = 101100$. La máquina está inicialmente en q_0 y la cabeza lectora lee el 1, entonces la unidad pasa el control al estado q_0 nuevamente, y la cabeza lectora se mueve al próximo símbolo (0), ahora q_0 por 0 va a q_1 y la cabeza tiene el 1, y así sucesivamente.

Ejemplo 2:

¿Cómo construir un AFD que reconozca números enteros que pueden, o no, estar precedidos por un signo (+, -)?, ver figura 5.

Ejemplo 3:

¿Cómo contruímos un AFD que reconozca las cadenas de a 's y b 's, que nunca contengan tres b 's seguidas?, ver figura 6.

Estos autómatas son determinísticos, es decir que la función de transición es total, o sea que para cada estado y para cada símbolo del alfabeto hay una y sólo una transición definida. Analice cuál es el alfabeto para cada uno de los AFD's de los ejemplos 2 y 3.

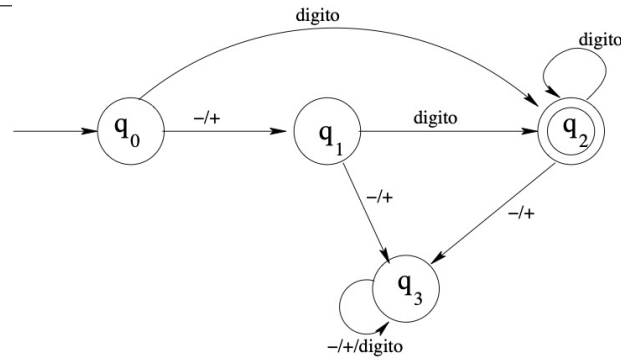


Figura 5: Autómata Finito Determinístico, donde digito = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.

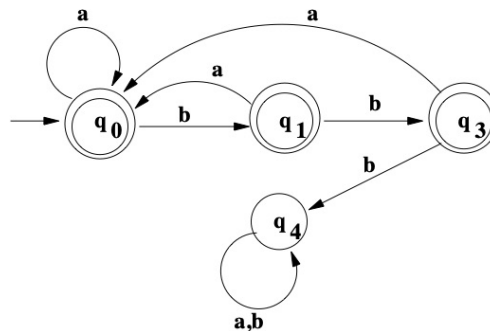


Figura 6: Autómata Finito Determinístico.

8.2. Autómata Finito No-Determinístico (AFND)

Un autómata finito es no determinístico si la función de transición no es total. Es decir que pueden haber múltiples arcos desde un estado con el mismo símbolo del alfabeto, o puede que para algunos símbolos del alfabeto y algunos estados no haya definida ninguna transición. La definición formal es igual a la de AFD, salvo que la función de transición se define de la siguiente manera: $\delta : Q \times \Sigma \rightarrow 2^Q$.

¿Cuál será el lenguaje que reconoce el AFND de la figura 7? En primer lugar recordemos que, una cadena es aceptada si hay algún paso desde el estado inicial a un estado final.

Por ejemplo la cadena 01 es aceptada por este autómata porque existe un paso desde el estado inicial al estado final, es decir el paso $q_0 \xrightarrow{0} q_1$ y $q_1 \xrightarrow{1} q_2$ y dado que q_2 es un estado final, entonces se puede asegurar que la cadena es aceptada. Pero cuando trabajamos con AFND se deben analizar todos los pasos posibles, por ejemplo considerando la cadena 01, también existe el paso $q_0 \xrightarrow{0} q_0$ y $q_0 \xrightarrow{1} q_0$, pero q_0 no es final. Es decir que, para asegurar que una cadena es aceptada por un AFND es necesario analizar todos los pasos posibles, si por al menos 1 de estos pasos se arriba a un estado final, entonces la cadena es aceptada. Y si una cadena no pertenece al lenguaje aceptado por el AFND, entonces no debe existir ningún camino desde el estado inicial a algún estado final, para dicha cadena.

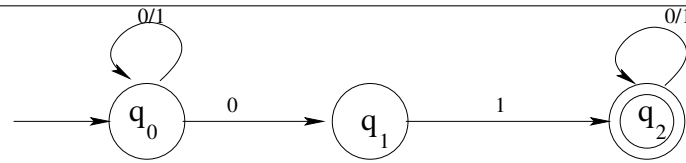


Figura 7: Autómata Finito No Determinístico.

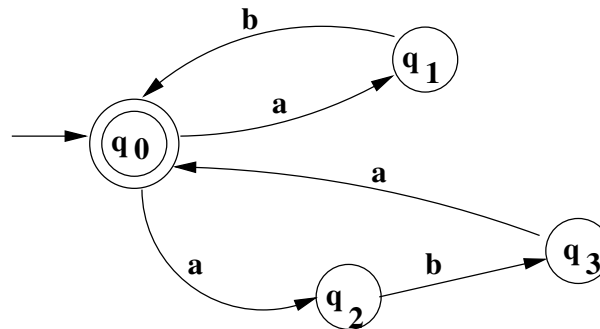


Figura 8: Autómata Finito No Determinístico.

Ejemplo: Un AFND que reconoce el lenguaje cuyas cadenas se pueden descomponer en secuencias de *ab* o *aba*, ver figura 8.

Importante: existe un teorema que asegura que *para todo AFND existe un AFD que reconoce el mismo lenguaje*.

9. Gramáticas

Son el método más popular utilizado para describir la sintaxis de los lenguajes de programación.

Chomsky definió cuatro tipos:

- 1- Gramáticas regulares que permiten generar lenguajes regulares.
- 2- Gramáticas libres del contexto que permiten generar lenguajes libres del contexto.
- 3- Gramáticas dependientes del contexto que permiten generar lenguajes dependientes del contexto.
- 4- Gramáticas irrestrictas que permiten generar lenguajes irrestrictos.

Las más usadas en el área de compiladores son las **gramáticas libres del contexto** y las **gramáticas regulares**.

Definición: formalmente una gramática G es una 4-upla $G = (N, \Sigma, P, S)$ donde:

N = es el conjunto finito de símbolos no terminales.

Σ = es el conjunto finito de símbolos terminales o alfabeto.

P = es el conjunto finito de reglas o producciones,

$$P \subseteq (N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$$

$S \in N$ = es un símbolo especial llamado símbolo distinguido.

Las cadenas o sentencias que forman un lenguaje se obtienen por las sucesivas aplicaciones de las producciones, comenzando desde el símbolo distinguido.

Ejemplo:

$$G = (\{S, T\}, \{0, 1\}, P, S)$$

$P :$

$$S \rightarrow 1|1T$$

$$T \rightarrow 0|1|0T|1T$$

Las gramáticas contienen un conjunto de símbolos denominados no terminales, en este caso $\{S, T\}$, un conjunto de símbolos terminales, $\{0, 1\}$, es decir los símbolos que constituirán las cadenas válidas, el símbolo distinguido S y el conjunto de reglas que van a permitir generar cadenas.

La $|$ se utiliza para **separar las distintas alternativas (reglas)**, por ejemplo: $S \rightarrow 1|1T$, es lo mismo que escribir $S \rightarrow 1$ y $S \rightarrow 1T$

¿Cómo se usan las reglas o producciones para generar cadenas?

En primer lugar hay que comenzar por una regla que tenga en su parte izquierda el símbolo distinguido, y de ahí se aplican las posibles reglas reemplazando cada no terminal, de a uno por vez, por su respectiva parte derecha, utilizando una producción que contenga en su parte izquierda dicho no terminal, y el reemplazo se hace por su respectiva parte derecha. Para esto se utiliza una relación que se denomina **deriva** y para la cual se utiliza el símbolo \Rightarrow .

$S \Rightarrow 1$ se deriva la cadena 1

$S \Rightarrow 1T \Rightarrow 11T \Rightarrow 110$ se deriva la cadena 110

$S \Rightarrow 1T \Rightarrow 10T \Rightarrow 101T \Rightarrow 1010T \Rightarrow 10100$ se deriva la cadena 10100

Una **forma sentencial** es una cadena que contiene símbolos terminales y no terminales, para el ejemplo anterior tenemos, entre otras, $1T$, $11T$, $10T$, $101T$, etc.

Una **sentencia** es una cadena formada sólo de símbolos terminales, por ejemplo 1, 110 y 10100 son sentencias.

Ahora se puede definir el lenguaje generado por una gramática G como: **el conjunto de sentencias que pueden ser derivadas desde el símbolo distinguido de una gramática.**

Formalmente:

$L(G) = \{w \in \Sigma^* / S \xRightarrow{*} w\}$, o sea las cadenas w que pueden ser derivadas a partir de S , usando las producciones de G .

Recordar: en general cuando se construye una gramática se da directamente el conjunto de producciones o reglas.

9.1. Gramáticas libres del contexto (GLC)

Su principal característica es la de permitir generar lenguajes cuyas cadenas presentan modalidad espejo. ¿Qué significa esto? que permiten generar cadenas del tipo: $(())$, `begin begin end end`, $\{\{\{\}\}\}$, etc.

Su importancia radica en que permiten describir los aspectos sintácticos de los lenguajes de programación. Es decir, cómo debe escribirse correctamente un programa. Sin embargo tienen sus limitaciones desde el punto de vista semántico (por ejemplo, declaración y uso de identificadores en Pascal).

Importante: Las restricciones impuestas sobre el conjunto P de producciones determinan el tipo de gramática y por ende el tipo de lenguaje que dicha gramática puede generar.

Definición: una gramática $G = (N, \Sigma, P, S)$ es libre de contexto, si sus producciones en P tienen la siguiente forma: $A \rightarrow \alpha$, con $A \in N$ y $\alpha \in (N \cup \Sigma)^*$.

El nombre “*libre de contexto*” se debe a que cada una de las producciones puede ser aplicada independientemente del contexto en donde aparezca un no terminal en una *forma sentencial*.

Ejemplo: construir una gramática que genere secuencias de paréntesis anidados, es decir, cadenas del tipo $()$, $(())$, $((()))$, etc:

¿Cómo se puede comenzar a construir una GLC?

En primer lugar podemos generar $()$, entonces con la regla $S \rightarrow ()$, se puede obtener, pero ahora se necesita repetir de manera anidada los paréntesis, entonces se puede tener una regla que repita S , $S \rightarrow S$, junto con la anterior, entonces:

$$P : \\ S \rightarrow () | S$$

Entonces, con estas reglas se pueden generar cadenas como estas:

$$S \Rightarrow () \\ S \Rightarrow S \Rightarrow S \Rightarrow \dots()$$

Se ve que se obtiene sólo la cadena $()$, entonces esta gramática **no genera** las cadenas que se necesitan, porque en realidad se necesita tener un no terminal junto con los paréntesis, es decir que se podría agregar la producción $S \rightarrow (S)$, la gramática completa sería:

$$P : \\ S \rightarrow () | (S)$$

Algunos ejemplos de cadenas que se pueden generar son:

$$S \Rightarrow () \\ S \Rightarrow (S) \Rightarrow ((S)) \Rightarrow ((()))$$

Esta gramática sí genera las secuencias de paréntesis anidados, que era el objetivo planteado. Se puede describir el lenguaje generado por esta gramática de la siguiente manera $L_1 = \{()^n / n \geq 1\}$.

Otra manera de construir esta gramática, usando la **palabra vacía** (λ), para poder observar su utilidad en la construcción de gramáticas:

$$P : \\ S \rightarrow \lambda|(S)$$

Algunos ejemplos de cadenas que se pueden generar:

$$S \Rightarrow \lambda$$

$$S \Rightarrow (S) \Rightarrow (), \text{ esta última derivación es porque } S \rightarrow \lambda.$$

$$S \Rightarrow (S) \Rightarrow ((S)) \Rightarrow (((S))) \Rightarrow ((())), \text{ esta última derivación es porque } S \rightarrow \lambda.$$

El lenguaje generado por esta gramática es $L_2 = \{(^n)^n/n \geq 0\}$. ¿Cuál es la diferencia con L_1 ?

9.1.1. Criterios de selección del no terminal a expandir en una secuencia de derivaciones.

En el contexto de las gramáticas libres del contexto, cuando se aplican las reglas con el objetivo de generar una sentencia, se suele trabajar seleccionando un criterio con respecto al no terminal que se va a reemplazar, básicamente existen dos criterios:

- Elegir siempre el no terminal de más a la izquierda. La forma sentencial así obtenida se denomina **“forma sentencial izquierda.”**
- Elegir siempre el no terminal de más a la derecha. La forma sentencial así obtenida se denomina **“forma sentencial derecha.”**

Ejemplo: construir una gramática que permita generar expresiones aritméticas, es decir, expresiones que contengan los operadores $+$, $*$, que permitan el uso de $(,)$ y los posibles identificadores a, b, c . Las expresiones pueden ser: $b, a + b * c, a + a + a, b * c * b + a, (c * a) + b$, etc.

El símbolo E puede ser el símbolo distinguido y el que permita generar las expresiones, pero puede suceder que se necesiten más de un no terminal. Por ejemplo, un no terminal, que con sus reglas, permita generar los identificadores, llamémosle I , entonces ¿cómo podrían ser las reglas?

$$P : \\ E \rightarrow E + E | E * E | (E) | I \\ I \rightarrow a | b | c$$

falta agregar que $N = \{E, I\}, \Sigma = \{+, *, (,), a, b, c\}$.

A continuación se muestran ejemplos de cadenas que pueden ser generadas, utilizando los distintos criterios de selección del no terminal a expandir. Los símbolos subrayados son los escogidos para realizar la expansión en cada paso de la derivación.

- $E \Rightarrow \underline{E} + E \Rightarrow \underline{I} + E \Rightarrow a + \underline{E} \Rightarrow a + \underline{E} * E \Rightarrow a + \underline{I} * E \Rightarrow a + b * \underline{E} \Rightarrow a + b * \underline{I} \Rightarrow a + b * c$
- $E \Rightarrow E + \underline{E} \Rightarrow E + E * \underline{E} \Rightarrow E + E * \underline{I} \Rightarrow E + \underline{E} * c \Rightarrow E + \underline{I} * c \Rightarrow \underline{E} + b * c \Rightarrow \underline{I} + b * c \Rightarrow a + b * c$

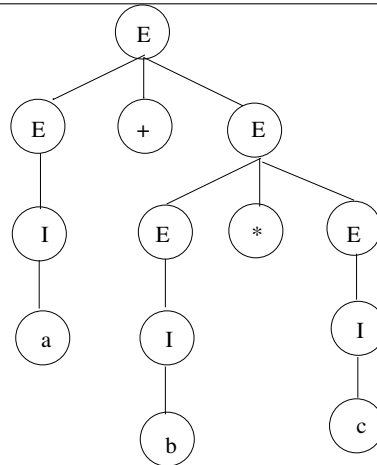


Figura 9: Árbol de Derivación para la cadena $w = a + b * c$

9.2. Árboles de Derivación

Una de las características distintivas de las gramáticas libres del contexto es que describen la estructura sintáctica jerárquica de las sentencias del lenguaje. Esta estructura puede ser expresada mediante un árbol, denominado árbol de derivación o de parsing.

Definición: Sea $G = (N, \Sigma, P, S)$ una GLC, luego un árbol es un Árbol de Derivación para G si:

1. Cada vértice está rotulado por un símbolo de $(N \cup \Sigma) \cup \{\lambda\}$.
2. El rótulo de la raíz del árbol es S , el símbolo distinguido.
3. Si un vértice es un nodo interior con rótulo A , luego $A \in N$.
4. Si n tiene rótulo A y los vértices n_1, n_2, \dots, n_k son los descendientes de n , de izquierda a derecha, con rótulos X_1, X_2, \dots, X_k respectivamente, luego $A \rightarrow X_1 X_2 \dots X_k \in P$.
5. Si el vértice n tiene rótulo λ , luego n es una hoja y es el único descendiente de su padre. (Para las producciones del tipo $A \rightarrow \lambda$).
6. Las hojas son rotuladas con símbolos de Σ .

Ejemplo: a partir de la gramática que genera expresiones aritméticas dada previamente y la cadena $w = a + b * c$ obtenemos el árbol de derivación de la figura 9:

9.3. BNF

Existe también otra forma de describir un lenguaje de programación que se denomina BNF (Backus Naur Form). BNF es un **metalenguaje**, o sea un lenguaje que se usa para describir otro lenguaje.

La BNF es una notación natural para describir sintaxis. Al igual que con gramáticas se utilizan las mismas componentes, pero se especifican, en algunos casos de otra forma, a saber:

$\langle \dots \rangle$ representa un no terminal, dentro de los corchetes angulares puede haber una secuencia de símbolos.

$::=$ se lee produce o se define como, se usa en lugar de \rightarrow .

Los símbolos terminales se representan con una o más letras en minúscula.

Ejemplo:

La sentencia condicional if en Pascal se puede definir utilizando dos reglas:

$$\begin{aligned} \langle \text{sentencia} - \text{condicional} \rangle &::= \text{if } \langle \text{expresion} - \text{booleana} \rangle \\ &\quad \text{then } \langle \text{sentencia} \rangle \\ &\quad \text{else } \langle \text{sentencia} \rangle \end{aligned}$$

$$\langle \text{sentencia} - \text{condicional} \rangle ::= \text{if } \langle \text{expresion} - \text{booleana} \rangle$$

$$\quad \text{then } \langle \text{sentencia} \rangle$$

Ejercicio: completar las reglas para $\langle \text{expresion} - \text{booleana} \rangle$ y para $\langle \text{sentencia} \rangle$.

9.4. BNF extendida (BNFE)

Con el objetivo de incrementar la legibilidad y la facilidad de escritura a la BNF se hicieron algunas extensiones, que no afectan su poder descriptivo. Dichas extensiones son las siguientes:

- Si un elemento es opcional se encierra entre $[]$.

Ejemplo:

$$\langle \text{selec} \rangle ::= \text{if}(\langle \text{exp} \rangle) \langle \text{sent} \rangle [\text{else } \langle \text{sent} \rangle]$$

- Para la elección de alternativas se usa $|$ y opcionalmente también se pueden usar $()$.

Ejemplo:

$$\langle \text{for} \rangle ::= \text{for } \langle \text{var} \rangle := \langle \text{expresion} \rangle (\text{to}|\text{downto}) \langle \text{expresion} \rangle \text{ do } \langle \text{sentencia} \rangle$$

- Para repetir un elemento un número arbitrario de veces, se encierra al mismo entre $\{ \}$.

Ejemplo:

$$\langle \text{lista} - \text{ident} \rangle ::= \langle \text{identificador} \rangle \{ , \langle \text{identificador} \rangle \}$$

Ejemplo:

BNF

$$\langle \text{entero} - \text{con} - \text{signo} \rangle ::= \langle \text{entero} \rangle | \langle \text{signo} \rangle \langle \text{entero} \rangle$$

$$\langle \text{entero} \rangle ::= \langle \text{digito} \rangle | \langle \text{digito} \rangle \langle \text{entero} \rangle$$

$$\langle \text{signo} \rangle ::= + | -$$

$$\langle \text{digito} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$$

BNFE

$$\langle \text{entero} - \text{con} - \text{signo} \rangle ::= [+|-](0|1|2|3|4|5|6|7|8|9)\{(0|1|2|3|4|5|6|7|8|9)\}$$

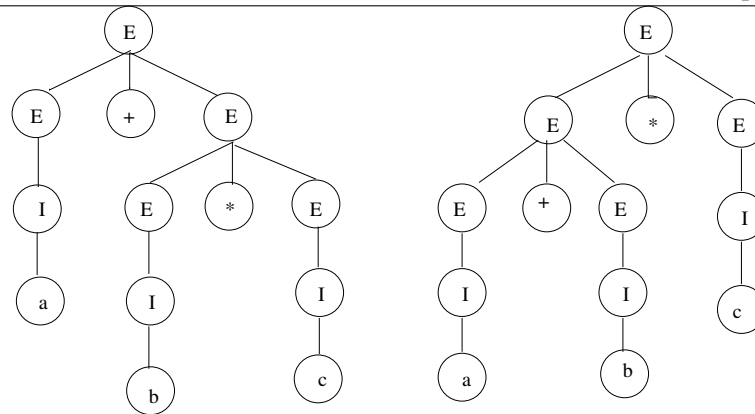


Figura 10: Árboles de Derivación para $w = a + b * c$.

9.5. Precedencia y asociatividad de operadores en una gramática

Las reglas de las gramáticas permiten reflejar la precedencia y asociatividad de operadores (en el caso que se construya una gramática que genere expresiones). A continuación se describe cada uno de estos conceptos:

Precedencia de Operadores:

El hecho de que un operador en una expresión aritmética, tenga mayor profundidad en el árbol de derivación, obtenido para ella, puede ser utilizado para indicar que tiene precedencia sobre uno que tenga menor profundidad.

Por ejemplo si se observa la gramática que genera expresiones aritméticas dada en el comienzo, y se analizan los dos posibles árboles de derivación para la expresión $a + b * c$, ver figura 10.

En el primer árbol el operador de multiplicación tiene profundidad mayor que el operador de suma.

En el segundo árbol el operador de suma es el que tiene mayor profundidad que el de multiplicación.

Es posible construir una gramática de manera de separar los operadores y reflejar la precedencia que es habitual para ellos, es decir que el operador de multiplicación tenga mayor precedencia que el de suma. Esto por lo tanto se verá reflejado en el árbol de derivación. Una gramática tal es justamente la gramática no ambigua, construida previamente, que genera expresiones aritméticas.

Asociatividad de Operadores

Otro aspecto importante concerniente a las gramáticas para expresiones, es si la asociatividad de los operadores se describe correctamente.

Es decir, al construir los árboles de parsing para expresiones con dos o más ocurrencias de operadores adyacentes con igual precedencia, ¿es posible que aquellas ocurrencias tengan un orden jerárquico propio?

Un ejemplo de una expresión tal es $a + b + c$, ver en la figura 11 el árbol de derivación para esta cadena, utilizando la gramática no ambigua para expresiones aritméticas.

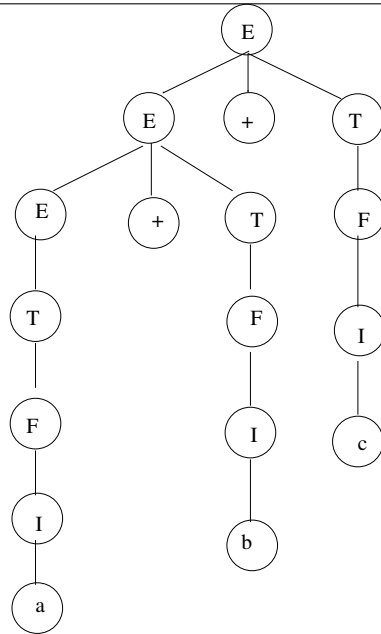


Figura 11: Árbol de Derivación para $w = a + b + c$.

Se puede observar que el operador de suma de más a la izquierda, se encuentra más abajo en el árbol, que el operador de suma de más a la derecha. Esto es correcto si se trabaja con asociatividad a izquierda, que de hecho es la manera usual de asociatividad.