Volumen I – DESDE LOS USUARIOS

1. Definiciones básicas e Introducción a Lenguajes Formales *(ver p. 7)*

La sintaxis de un Lenguaje de Programación (LP) describe las combinaciones de símbolos que forman un programa sintácticamente correcto. Los Lenguajes Formales (LFs) están formados por palabras, que son cadenas, y éstas están constituidas por caracteres, que forman parte de un alfabeto.

1.1 Caracteres y Alfabetos *(ver p. 7 y 8)*

Carácter: elemento constructivo básico, indivisible, a partir del cual se forman los alfabetos.

Alfabeto: conjunto finito de caracteres. Con sus caracteres se construyen las cadenas de caracteres de un LF. Se lo suele identificar con la letra griega Σ (sigma).

1.2 Cadenas *(ver p. 8)*

Cadena de caracteres (String): secuencia finita de caracteres, tomados de cierto alfabeto, colocados uno a continuación de otro. Es decir, una cadena se construye concatenando caracteres de un alfabeto dado.

No es lo mismo un carácter de un alfabeto que una cadena compuesto por ese único carácter.

1.2.1 Longitud *(ver p. 8)*

Longitud de una cadena: cantidad de caracteres que la componen.

1.2.2 Cadena Vacía *(ver p. 9)*

Cadena vacía: cadena que no tiene caracteres. Su longitud es 0. Se la simbolizada habitualmente con la letra griega ε, aunque otros autores eligen utilizar la letra griega λ.

El carácter ε no forma parte de ningún alfabeto.

1.2.3 Potenciación de un carácter *(ver p. 9)*

La potenciación de un carácter se utiliza para simplificar la repetición de un carácter. Siendo c un carácter cualquiera, c^n representa la repetición del carácter c, n veces. Por ejemplo: para c^n, siendo n=4, c^4 = cccc.

Para un carácter, no existe la potencia 0.

1.2.4 Concatenación de cadenas *(ver p. 9 y 10)*

La operación de concatenación aplicada a cadenas produce una nueva cadena formada por los caracteres de la primera cadena seguidos inmediatamente por los caracteres de la segunda cadena.

La concatenación se extiende a más de 2 cadenas no siempre es conmutativa y su elemento neutro es la cadena vacía.

1.2.5 Potenciación de una cadena *(ver p. 10 y 11)*

La potenciación de una cadena se utiliza para simplificar la repetición de una cadena. Siendo S es una cadena, S^n representa la cadena que resulta de concatenar la cadena S, consigo misma, n-1 veces. Es decir, la cadena final resulta de repetir la cadena original n veces. Por ejemplo: para S^n, con n=3, S^3 = SSS.

S^0 es ε, la cadena vacía.
$\varepsilon^n = \varepsilon$, siempre que n sea un entero no negativo.

1.3 Lenguajes Naturales (LNs) y Lenguajes Formales (LFs) *(ver p. 11, 12 y 13)*

Un LN es el lenguaje hablado/escrito usado por los seres humanos para comunicarse. Los LNs evolucionan con el paso del tiempo. Sus reglas gramaticales surgen después del desarrollo del lenguaje, para poder explicar su sintaxis. El significado (la semántica) de cada palabra, oración y frase es, en general, más importante que su composición sintáctica.

Un LF es un conjunto de cadenas formadas con caracteres de un alfabeto dado. Las cadenas que lo constituyen no tienen una semántica asociada, sólo tienen una sintaxis dada por la ubicación de los caracteres dentro de una cadena. Los LFs nunca puede evolucionar porque están definidos por reglas gramaticales preestablecidas, debiéndose ajustar rigurosamente a ellas.

1.3.1 Palabra *(ver p. 13 y 14)*

Una cadena es una palabra de un lenguaje formal si esa cadena pertenece a dicho lenguaje.

La pertenencia de una cadena a un LF le da la propiedad de ser palabra de ese lenguaje en particular.

Un LF puede ser descrito por extensión, por comprensión o por una frase comunicada en un LN.
Si al describir un LF no se indica explícitamente sobre qué alfabeto está construido, se considera que el alfabeto está formado por los caracteres que forman las palabras del lenguaje.

1.3.2 Propiedades de las Palabras *(ver p. 14 y 15)*

Todos los conceptos sobre cadena (longitud, palabra vacía, concatenación, potenciación) se aplican a las palabras.

La concatenación de dos palabras produce una cadena que no siempre es una palabra del lenguaje.

1.3.3 Cardinalidad de un LF *(ver p. 15)*

La cardinalidad de un LF es la cantidad de palabras que lo componen.

1.3.4 Sublenguajes *(ver p. 15)*

Un sublenguaje es un subconjunto de un lenguaje formal dado.

El sublenguaje vacío (generalmente representado por φ) es un sublenguaje de cualquier LF.
NO CONFUNDIR sublenguaje vacío (de cardinalidad 0) con el lenguaje que sólo contiene la palabra vacía (de cardinalidad 1).

1.4 LFs Infinitos *(ver p. 16)*

Los LFs infinitos son los LFs que tienen una cantidad infinita de palabras (aunque cada una de ellas debe ser de longitud finita, ya que no existen las palabras de longitud infinita).

1.4.1 Lenguaje Universal (LU) sobre un alfabeto *(ver p. 16 y 17)*

Dado un alfabeto Σ, el LU sobre este alfabeto es un LF infinito que contiene todas las palabras que se pueden formar con los caracteres del alfabeto Σ más la palabra vacía. Se lo representa Σ*. Por ejemplo: si Σ = {a, b}, entonces su LU para este alfabeto estará formado por la palabra vacía, todas las palabras que sólo contengan a, todas las palabras que contengan b y todas las palabras formadas por a y b concatenadas en cualquier orden. La concatenación de palabras de Σ* siempre produce una palabra de este LU sobre el alfabeto {a, b}.

Como cualquier lenguaje L sobre el alfabeto Σ es un sublenguaje de Σ, existen infinitos LFs sobre un alfabeto dado.*

1.4.2 LFs Infinitos más complejos

(Ver próximo capítulo...)

2. Gramáticas Formales y Jerarquía de Chomsky *(ver p. 19)*

Un LF, sea finito o infinito, es un conjunto de palabras con una connotación sintáctica, sin semántica.

Las Gramáticas Formales son las estructuras que generan todas las palabras que forman un LF.

2.1 Gramática Formal (GF) *(ver p. 19 y 20)*

Una GF es un conjunto de producciones, que son reglas de reescritura que se aplican para obtener las palabras del LF que la GF en cuestión genera.

Toda producción está formada por 3 partes: el lado izquierdo, el operador de producción y el lado derecho. Esto quiere decir que el lado izquierdo produce (o genera) al lado derecho.
Una GF puede tener entre sus producciones la producción-épsilon, la cual indica que el lado izquierdo genera a la palabra vacía.

2.1.1 Definición Formal de una GF *(ver p. 20, 21 y 22)*

Toda GF es una 4-upla (V_N, V_T, P, S), donde:

- V_N es el conjunto finito de "productores".
- V_T es el conjunto finito de "producidos".
- P es el conjunto finito de producciones.
- $S \in V_N$ es el axioma, un no terminal a partir del cual siempre deben comenzar a aplicarse las producciones que generan las palabras de un determinado LF.

Un ejemplo de una GF sería: G=({S, T}, {a,b}, {S→aT, T→aT, T→b}, S).

"Una GF genera un LF" significa que esa gramática es capaz de generar todas las palabras del LF y, a su vez, no genera ninguna cadena que esté fuera del lenguaje.

2.2 La Jerarquía de Chomsky *(ver p. 22)*

La Jerarquía de Noam Chomsky establece una clasificación de 4 tipos de GFs que generan 4 tipos diferentes de LFs.

Las GF se clasifican según las restricciones que se imponen a sus producciones, y la Jerarquía de Chomsky define estos 4 niveles:

- Gramáticas Regulares (GRs) o gramáticas tipo 3.
- Gramáticas Independientes del Contexto (GICs) o gramáticas tipo 2.
- Gramáticas Sensibles al Contexto (GSCs) o gramáticas tipo 1.
- Gramáticas Irrestrictas (GIs) gramáticas tipo 0.

Dichos tipos de gramáticas generan los siguientes tipos de LF:

GF	GR	GIC	GSC	GI
LF generado por la GF	LR	LIC	LSC	LI

2.2.1 Gramática Regular (GR) *(ver p. 22 y 23)*

Para que una GF sea una GR, sus producciones deben tener estas restricciones:

- El lado izquierdo debe tener 1 solo no terminal.
- El lado derecho debe estar formado por 1 solo terminal o un terminal seguido de un no terminal o ε.

También, una GF es una GR si sus producciones tienen las siguientes restricciones:

- El lado izquierdo debe tener 1 solo no terminal.
- El lado derecho debe estar formado por 1 solo terminal o un no terminal seguido de un terminal o ε.

2.2.1.1 GR que genera un LR Infinito *(ver p. 23 y 24)*

Para generar LRs infinitos existen las llamadas producciones recursivas, en las que el no terminal que figura en el lado izquierdo también figura en el lado derecho.

2.2.1.2 Gramática Quasi-Regular (GQR) *(ver p. 24 y 25)*

Son gramáticas en las que un conjunto de terminales es reemplazado por un no terminal en una o varias producciones. Una GQR abrevia la escritura de una GR, y siempre puede ser reescrita como una GR.

Dos GFs son equivalentes si generan el mismo LF.

2.2.2 Gramática Independiente del Contexto (GIC) *(ver p. 25 y 26)*

Para que una GF sea una GIC, sus producciones deben tener en el lado izquierdo un único no terminal.

2.2.2.1 GICs que generan un Lenguaje Infinito *(ver p. 26)*

Las GICs pueden generar LICs utilizando producciones recursivas.

"Independiente del Contexto" refleja que, como el lado izquierdo de cada producción sólo puede contener un único no terminal, toda producción de una GIC puede aplicarse sin importar el contexto donde se encuentre dicho no terminal.
Toda GR también es una GIC. La inversa no es cierta.

2.2.3 Gramática Irrestricta (GI) *(ver p. 27)*

Son gramáticas en las que tanto el lado izquierdo como el lado derecho de sus producciones pueden ser secuencias de terminales y no terminales. Tienen una única restricción: el lado izquierdo de las producciones nunca puede ser ε.

2.2.4 Gramática Sensible al Contexto (GSC) *(ver p. 27)*

Una GSC es una GI cuando la cardinalidad del lado izquierdo es menor o igual a la cardinalidad del lado derecho.

2.3 El Proceso de Derivación *(ver p. 27, 28, 29 y 30)*

El proceso de derivación permite obtener cada una de las palabras de un LF, a partir del axioma de una GF que lo genera, y aplicando sucesivamente las producciones convenientes de esa gramática.

Este proceso permite determinar si una cadena dada es o no una palabra del lenguaje

Existen distintas formas de representar una derivación: en forma horizontal, en forma vertical o en forma de árbol.

La cadena de derivaciones la cadena de terminales y no terminales que se forma en cada paso de la derivación vertical antes de obtener la palabra, si es que existe.
En las derivaciones verticales, en cada paso se reemplaza el no terminal que se encuentra primero en la cadena de derivación. Si la derivación vertical es a izquierda, se reemplaza el primer no terminal leído de izquierda a derecha. Si la derivación vertical es a derecha, se reemplaza el primer no terminal leído de derecha a izquierda.
Si una palabra puede obtenerse por medio de una derivación vertical a izquierda, también se la podrá obtener si se deriva verticalmente a derecha.

2.4 Introducción a las GQRs, las GICs y los Lenguajes de Programación *(ver p. 30 y 31)*

(Ver próximo capítulo...)

3. Sintaxis y BNF *(ver p. 33)*

Un Lenguaje de Programación (LP) es una notación utilizada para describir algoritmos y estructuras de datos que resuelven problemas computacionales. Desde el punto de vista sintáctico, hay una relación muy importante entre los LPs y los LFs, ya que un LP está formado básicamente por un conjunto de LRs y un conjunto de LICs. Las notaciones que describen en forma precisa la sintaxis de un LP se denominan BNF.

3.1 Introducción a la Sintaxis *(ver p. 33 y 34)*

Las categorías léxicas (o tokens) constituyen diferentes LRs. Son ejemplos los identificadores, las palabras reservadas, los números enteros, los números reales, los caracteres, las cadenas constantes, los operadores y los caracteres de puntuación.. Algunos de estos lenguajes son finitos (como los operadores) y otros, infinitos (como los identificadores, los números enteros o los números reales).

Las categorías sintácticas son las expresiones y las sentencias de un LP (que son, en general, LICs). Estos lenguajes requieren ser generados por GICs (no pueden ser generados por GRs ni por GQRs).

Todos estos tipos de gramáticas se caracterizan por tener un único no terminal en el lado izquierdo.
Una GF no sólo genera un LF, sino que también puede describir la sintaxis del lenguaje que genera.

La notación (que no debe dar lugar a ambigüedades) utilizada para describir la sintaxis de un LP es la BNF.

3.2 Identificadores y su Sintaxis *(ver p. 34, 35 y 36)*

Los identificadores son secuencias de uno o más caracteres que nombran diferentes entidades (variables, funciones, procedimientos, constantes, tipos, etc.) de un LP. Los identificadores constituyen un LR infinito. Las palabras reservadas (que son una secuencia de caracteres) habitualmente forman un LR finito, independiente de los identificadores.

Todo no terminal debe aparecer en el lado izquierdo de al menos una producción.
Un terminal jamás puede aparecer en el lado izquierdo de una producción.

3.3 Las expresiones y la Sintaxis *(ver p. 36, 37 y 38)*

Una GIC para expresiones no sólo debe producir todas las expresiones válidas, sino que también debe preocuparse por la precedencia (prioridad) y la asociatividad (evaluación de izquierda a derecha o de derecha a izquierda para operadores con igual precedencia) de cada uno de los operadores existentes en ese LP.

Cuanto más cerca del axioma, menor es la prioridad de un operador.
La cadena de derivación es la secuencia de terminales y no terminales que se obtiene en cada paso al derivar.

3.3.1 La Evaluación de una expresión, Precedencia y Asociatividad *(ver p. 38, 39, 40, 41 y 42)*

La evaluación de una expresión se realiza como un proceso inverso al de la derivación hasta llegar al axioma.

Se debe hacer una reducción de la tabla generada por la derivación, y es acá donde se ve si la precedencia de los operadores está correctamente determinada por la gramática diseñada.

3.4 BNF y ALGOL *(ver p. 42 y 43)*

La sigla BNF corresponde a la frase “Backus Normal Form” o a la frase “Backus-Naur Form”. Con un breve manual de referencia del lenguaje ALGOL (ALGOrithmic Language) se publicó, en 1960, una descripción formal de un LP. Esta descripción, similar a las producciones de las GICs, se llamaría luego BNF.

La notación BNF consiste en un conjunto de reglas que definen la sintaxis de los componentes y de las estructuras de un LP dado utilizando GICs.

Cada regla se forma con elementos providentes de 3 conjuntos disjuntos:

- No terminales (palabras o frases encerradas entre corchetes angulares).
- Terminales (caracteres del alfabeto o palabras del lenguaje sobre los cuales se construye el LP descripto).
- Metasímbolos (caracteres o grupos de caracteres que ayudan a representar estas reglas).

En toda regla BNF hay, como en toda producción de una GIC, 3 componentes:

- El lado izquierdo (formado por un solo no terminal).
- El lado derecho (formado por terminales y no terminales o, vacío).
- Un metasímbolo (el operador que vincula el lado izquierdo con el lado derecho).

3.4.1 Dos ejemplos de BNF en ALGOL *(ver p. 43 y 44)*

3.5 BNF y el lenguaje Pascal *(ver p. 44)*

3.5.1 La Sintaxis del lenguaje Pascal, según Wirth *(ver p. 44, 45, 46 y 47)*

«Una formulación de la sintaxis es la tradicional BNF, en la que las categorías léxicas y categorías sintácticas son denotadas por palabras en inglés (en castellano, para nosotros) encerradas entre < y >» «Las llaves { y } denotan la posible repetición de los símbolos que encierra 0 o más veces (el operador clausura de Kleene)» A raíz de estas extensiones y demás, algunos autores decidieron utilizar la sigla EBNF (Extended BNF), en lugar de BNF.

3.5.2 Expresiones en Pascal *(ver p. 47, 48 y 49)*

3.5.3 Sentencias con Condiciones Booleanas *(ver p. 49 y 50)*

3.6 BNF y el ANSI C *(ver p. 50 y 51)*

El lenguaje de programación C fue desarrollado por Dennis Ritchie entre 1969 y 1972 para poder implementar el sistema operativo UNIX en un lenguaje de alto nivel. A mediados de los ’80, el ANSI (American National Standards Institute) comenzó a desarrollar una estandarización de C, la cual fue aprobada hacia fines de dicha década. Al año siguiente fue publicada como “American National Standard for Information Systems – Programming Language C”, o Manual de Referencia Oficial del ANSI C (MROC). A partir de entonces, este LP es conocido como ANSI C.

3.6.1 Un programa en ANSI C *(ver p. 51 y 52)*

3.6.2 Sintaxis de un subconjunto de ANSI C *(ver p. 53)*

En el MROC, cada una de las secciones están divididas en tres partes: sintaxis (con una BNF que define la sintaxis del constructo analizado), restricciones (en donde figuran ciertas restricciones sobre el constructo definido en sintaxis y que la BNF no expresa totalmente) y semántica (en donde se explica el significado de lo definido en sintaxis).

3.6.2.1 Las categorías léxicas (tokens) *(ver p. 53 y 54)*

Las categorías léxicas (o tokens), que son LRs, son las palabras reservadas, los identificadores, las constantes, las cadenas literales, los operadores y los caracteres de puntuación. Los elementos que componen un LR se denominan, en el área de los LFs, palabras. Acá, los llamaremos lexemas.

3.6.2.1.1 Las palabras reservadas *(ver p. 55)*

El ANSI C tiene 32 **palabras reservadas** que constituyen los elementos de un LR finito. Algunas de ellas son: char, do, double, else, float, for, if, int, long, return, sizeof, struct, typedef, void, while, etcétera.

3.6.2.1.2 Los identificadores *(ver p. 55)*

Los identificadores (LR infinito) deben empezar con un carácter no dígito, seguido de un dígito o de un no dígito.

3.6.2.1.3 Las constantes *(ver p. 55, 56 y 57)*

3.6.2.1.4 Los operadores y los caracteres de puntuación *(ver p. 58)*

Algunos de los operadores, que constituyen un LR finito, son: ++ * + & ¡ sizeof / % < <= == ¡= && || ?: = +=
Algunos de los caracteres de puntuación, que también forman un LR finito, son: ( ) { } , ;

3.6.2.2 Las categorías gramaticales *(ver p. 58)*

Las categorías gramaticales se dividen en: expresiones, declaraciones y sentencias.

3.6.2.2.1 Las expresiones *(ver p. 58, 59, 60, 61 y 62)*

Sintácticamente, una expresión es una secuencia de operandos y operadores, más el posible uso de paréntesis.

Toda constante, variable o invocación a función son casos particulares y mínimos de una expresión.

3.6.2.2.2 Declaraciones y Definiciones *(ver p. 63 y 64)*

3.6.2.2.3 Sentencias *(ver p. 64, 65 y 66)*

Las sentencias especifican las acciones que llevará a cabo la computadora en tiempo de ejecución.

3.7 Otros usos de BNF *(ver p. 66)*

4. Semántica *(ver p. 67)*

4.1 Definición de un Lenguaje de Programación *(ver p. 67)*

Sintácticamente, el LP se define con reglas que se representan en BNF.

Semánticamente, existen varios métodos para definir un LP (muchas veces se usa el lenguaje natural).

4.2 El ANSI C y su Semántica *(ver p. 67 y 68)*

Objeto: zona de la memoria cuyo contenido puede representar valores durante la ejecución de un programa. *Tipo: brinda el significado de un valor almacenado en un objeto o retornado por una función.*
Los tipos básicos de datos son el char (con y sin signo), los tipos enteros (con y sin signo) y los tipos reales *El tipo de dato void equivale a un conjunto vacío de valores.*

4.2.1 El identificador *(ver p. 68)*

Un identificador denota un objeto, una función y otras entidades.

4.2.2 Semántica de las constantes *(ver p. 68)*

Cada constante tiene un tipo, determinado por su forma y por su valor.

4.2.2.1 Constantes enteras *(ver p. 68 y 69)*

Dada una lista de tipos predeterminada, el tipo de una constante entera es el primero en el que su valor puede ser representado. Por ejemplo, para una constante entera decimal sin sufijo, dicha lista es: int, long int, unsigned long int.

4.2.2.2 Constantes reales *(ver p. 69 y 70)*

Las constantes reales pueden ser en punto fijo o en punto flotante. En general, una constante real tiene una parte significativa (interpretada como un número racional en base 10), que puede estar seguida de una parte exponente (interpretada como un entero en base 10) y de un sufijo real (que especifica su tipo).

La representación de un número real no es exacta, sino que es la representación más cercana a la exacta.

Una constante real sin sufijo es de tipo **double**. Si el sufijo es f o F, la constante real es de tipo **float**. Si el sufijo es l o L, **long double**. La diferencia entre estos tres tipos de constantes reales radica en el rango y en la precisión: las **float** son las que tienen menor rango y menor precisión; y las **long double**, las que tienen mayor rango y mayor precisión.

4.2.2.3 Constantes carácter *(ver p. 70)*

Una constante carácter se delimita por apóstrofos, y su valor es de tipo int. Una constante carácter es en realidad la representación numérica de ese carácter en la tabla de caracteres ASCII.

Existen algunas constantes carácter (el apóstrofo, el apóstrofo doble, la barra invertida, el carácter nueva línea, el carácter fin de cadena...) que requieren un formato de escritura especial, las cuales comienzan con una barra invertida.

4.2.2.4 Literal cadena *(ver p. 70 y 71)*

Un literal cadena es una secuencia de 0 o más caracteres delimitada por apóstrofos dobles.

4.2.3 Semántica de los operadores *(ver p. 71)*

Un operador especifica la realización de una operación que producirá un valor.

Un operando es una entidad sobre la que actúa el operador.

4.2.4 Semántica de los caracteres de puntuación *(ver p. 71)*

Un carácter de puntuación es un símbolo que tiene un significado semántico determinado, ya que depende del contexto (el mismo símbolo, en distintos contextos, pueden representar un carácter de puntuación o un operador, o parte de él), no especifica una operación y no produce un valor.

4.2.5 Semántica de las expresiones *(ver p. 71 y 72)*

Una expresión puede tener varias aplicaciones; la más común es que especifique el cálculo de un valor.

4.2.6 Semántica de las declaraciones y definiciones *(ver p. 73)*

Las declaraciones y las definiciones básicas especifican la interpretación de un conjunto de identificadores.

4.2.7 Semántica de las sentencias *(ver p. 73 y 74)*

Una sentencia especifica la realización de una acción.

5. Lenguajes Regulares y Expresiones Regulares *(ver p. 75)*

Un LF es un conjunto de palabras que sólo tienen una sintaxis, no tienen semántica. Los LFs más simples son los LRs.

Los LRs tienen gran importancia en el diseño de los LPs, ya que sus componentes básicos (tokens) constituyen diferentes LRs.
Un lenguaje es un LR si puede ser generado por una GR.

5.1 Determinación de LRs *(ver p. 75 y 76)*

Si un LF es finito, entonces siempre es un LR.

5.2 Las Expresiones Regulares (ERs) *(ver p. 76 y 77)*

La forma más precisa de representar a los LRs es mediante las ERs.

Una ER es una expresión que describe un conjunto de cadenas: todas aquellas que son palabras del LR que la ER representa. Las ERs se construyen utilizando los caracteres del alfabeto sobre el cual se define el LR.

5.2.1 ERs para LRs Finitos *(ver p. 77, 78 y 79)*

Una ER para un LR finito se obtiene usando los caracteres del alfabeto, el símbolo ε y los operadores de concatenación y de unión.

Un carácter puede representar 3 elementos distintos: un símbolo de un alfabeto, una palabra de longitud 1 o una ER. Esta ambigüedad es resuelta por el contexto en el cual se encuentra el carácter en cuestión.
El operador concatenación tiene mayor prioridad que el operador unión.
La operación concatenación, en general, no es conmutativa (existen la concatenación a izquierda y a derecha).
La operación unión sí es conmutativa.
Dos ERs son equivalentes si representan el mismo LR.

5.2.1.1 El operador potencia *(ver p. 79 y 80)*

El operador potencia es utilizado para simplificar la escritura, la lectura y la comprensión de ciertas ERs.

5.2.2 ERs para LRs Infinitos *(ver p. 80 y 81)*

Los LRs infinitos tienen al menos 1 carácter o un grupo de caracteres que se repite cierto número de veces.

Para representar estas repeticiones indeterminadas, se utiliza el operador unario Clausura de Kleene (con un asterisco), que tiene mayor prioridad que los operadores concatenación y unión. Este operador representa a la palabra vacía y a todas las palabras que se forman con la repetición de su operando un número indeterminado de veces.

5.2.2.1 El operador clausura positiva *(ver p. 81, 82 y 83)*

El operador clausura positiva se utiliza para simplificar la escritura de la ER. La clausura positiva elimina la existencia de la palabra vacía (a menos que la palabra vacía pertenezca al lenguaje).

5.2.2.2 La Expresión Regular Universal y su aplicación *(ver p. 83, 84 y 85)*

La ERU es la ER que representa al lenguaje universal sobre un alfabeto dado. La ERU representa al LR que contiene la palabra vacía y todas las palabras que se pueden formar con caracteres del alfabeto dado.

A cada ER le corresponde un único LR. Sin embargo, un LR puede ser representado por varias ERs.

5.3 Definición formal de las ERs *(ver p. 85 y 86)*

La ER original se construye utilizando sólo los operadores básicos (unión, concatenación y clausura de Kleene) y se define formalmente de la siguiente manera recursiva:

- φ, que es una ER que representa al LR vacío.
- ε, que es una ER que representa al LR que sólo contiene la palabra vacía.
- Todo carácter x de un alfabeto corresponde a una ER x que representa a un LR que sólo tiene una palabra formada por ese carácter x.
- Una cadena s es una ER s que representa a un LR que sólo contiene la palabra s.

A partir de las cuatro reglas básicas citadas arriba, toda ER más compleja se construye así (definición formal):

- Si R_1 y R_2 son ERs, R_1+R_2 es una ER.
- Si R_1 y R_2 son ERs, $R_1 \cdot R_2$ es una ER.
- Si R_1 es una ER, R_1^* es una ER.
- Si R_1 es una ER, (R_1) es una ER.

Esta definición se puede ampliar agregando los dos operadores:

- Si R_1 es una ER, R_1^+ es una ER.
- Si R_1 es una ER, R_1^n (con $n \geq 0$ y entero) es una ER.

Los tres operadores unarios (clausura de Kleene, clausura positiva y potencia) tienen prioridad máxima; el operador concatenación tiene prioridad media; y el operador unión tiene prioridad mínima.

5.4 Operaciones sobre LRs y las ERs correspondientes

Un LF es un LR si puede ser representado mediante una ER.

5.4.1 Generalidades *(ver p. 86)*

Los LRs son cerrados bajo la unión, la concatenación, la clausura de Kleene, la clausura positiva, el complemento (con respecto al LU) y la intersección.

5.4.2 La unión *(ver p. 86 y 87)*

La unión de dos LRs es un LR que contiene todas las palabras que pertenecen a cualquiera de los dos LRs.

5.4.3 La concatenación *(ver p. 87)*

La concatenación de dos LRs es un LR en el que cada palabra está formada por la concatenación de una palabra del primer LR con una palabra del segundo LR.

5.4.4 La clausura de Kleene *(ver p. 87 y 88)*

La clausura de Kleene de un LR, es un LR infinito (salvo una única excepción, que se produce cuando el LR está formado sólo por la palabra vacía) formado por la palabra vacía, las palabras de dicho lenguaje y todas aquellas palabras que se obtienen concatenando palabras de tal lenguaje un número arbitrario de veces.

5.4.5 La clausura positiva *(ver p. 88)*

La clausura positiva de un LR es un LR formado por las palabras de dicho lenguaje y todas aquellas palabras que se obtienen concatenando palabras de tal lenguaje un número arbitrario de veces.

La clausura positiva de un LR únicamente contiene la vacía si ésta pertenece al LR original.

5.4.6 El complemento *(ver p. 88 y 89)*

El complemento de un LR es un LR que está formado por todas las palabras que no pertenecen al LR original.

5.4.7 La intersección *(ver p. 89)*

La intersección de dos LRs es un LR constituido por todas las palabras que pertenecen a los dos lenguajes dados.

5.5 ERs y LPs *(ver p. 89, 90, 91 y 92)*

Los componentes léxicos de un LP (identificadores, palabras reservadas, constantes, literales cadena, operadores, símbolos de puntuación) constituyen diferentes LRs y, como tales, los podemos representar mediante LRs.

5.6 ERs Extendidas *(ver p. 92)*

Las ERs también se emplean para representar datos que serán procesados por diversas herramientas de software.

Lex es una herramienta diseñada para ayudar a construir un analizador léxico, un módulo fundamental que tiene todo compilador. Lex utiliza ERs para describir los patrones que debe encontrar y procesar.

5.6.1 Un metalenguaje para ERs *(ver p. 92)*

Un metalenguaje es un lenguaje que se usa para describir otro lenguaje. El lenguaje que queremos describir es el de las ERs, y el metalenguaje que se usará para describirlo tiene ciertos símbolos, además de los operandos y de los operadores, llamados metacaracteres. El lenguaje de las ERs está formado por: operandos (caracteres del alfabeto que intervienen en la construcción de las ERs), operadores (clausura de Kleene, concatenación, unión), metacaracteres (caracteres especiales que colaboran en la descripción de cada ER) y paréntesis (utilizados para agrupar).

5.6.1.1 El metalenguaje, sus metacaracteres y sus operadores *(ver p. 93, 94 y 95)*

Todo el texto debe escribirse en la línea sin subíndices ni supraíndices. Los metacaracteres y los operadores deben representar las operaciones básicas para la escritura de ERs más otras operaciones para facilitar la escritura de las ERs.

Metacaracteres operadores	Explicación	Ejemplo
.	Representa cualquier carácter, excepto el caracter nueva línea.	**a.a** representa toda cadena de 3 caracteres en la que el primer y el tercer carácter son **a**.
|	Operador unión de ERs.	**ab|b** representa la ER **ab+b**.
[ ]	Describe un conjunto de caracteres, simplifica el uso del operador unión.	**[abx]** representa la ER **a+b+x**.
[–]	Describe una clase de caracteres en 1 o más intervalos.	**[a–d]** representa la ER **a+b+c+d**.
{ }	Operador potencia.	**a{3}** representa la ER **aaa**.
{ , }	Operador potencia, extendido a un intervalo.	**a{1,3}** representa la ER **a+aa+aaa**.
?	Operador que indica 0 o 1 ocurrencias de la ER que lo precede.	**a?** representa la ER **a+ε**.
*****	Operador clausura de Kleene: cero o más ocurrencias de la ER que lo precede.	**a*** representa la ER a^*.
+	Operador clausura positiva: una o más ocurrencias de la ER que lo precede.	**a+** representa la ER a^+.
( )	Se utilizan para agrupar una ER.	**((ab)?|b)+** representa la ER **(ab+ ε +b)⁺)**

5.6.1.2 Ejercicios para escribir metaERs *(son sólo consignas de ejercicios)*

5.6.1.3 Una aplicación: Lex y Ejemplos *(ver p. 95, 96 y 97)*

El **lex** es una herramienta que permite detectar y procesar palabras de uno o varios LRs. Para detectar esas palabras utiliza ERs escritas en un metalenguaje, y para procesarlas utiliza el lenguaje de programación ANSI C.

El programa lex está compuesto por tres secciones: la primera sección (la llamada "sección de definiciones", que introduce codificación en ANSI C que será copiada en el programa final), la segunda sección ("sección de las reglas", en la que cada regla está formada por partes: un patrón –descripto mediante una metaER– y una acción –descripta mediante un bloque de código en ANSI C–) y la tercera sección (la llamada "sección de subrutinas del usuario", que puede contener cualquier codificación en ANSI C).

Volumen II – DESDE EL COMPILADOR

1. Introducción a Autómatas Finitos y Aplicaciones *(ver p. 7 y 8)*

Una GF genera un determinado LF y un autómata reconoce ese LF.

Un autómata reconoce a un determinado LF si tiene la capacidad de reconocer cada palabra del lenguaje y de rechazar toda cadena que no es una palabra de tal lenguaje.

Al surgir estos mecanismos abstractos llamados autómatas, se completa la jerarquía de Chomsky de esta manera:

Gramática Formal	Lenguaje Formal	Autómata
GR	LR	Autómata Finito
GIC	LIC	Autómata Finito con Pila
GSC	LSC	Máquina de Turing
GI	LI	Máquina de Turing

1.1 ¿Qué es un Autómata Finito (AF)? *(ver p. 8, 9 y 10)*

Un AF es una herramienta abstracta que se utiliza para reconocer un determinado LR.

Un AF es un modelo matemático que recibe una cadena constituida por caracteres de cierto alfabeto Σ y tiene la capacidad de determinar si esa cadena pertenece al LR que el AF reconoce.

Reconocer a un LR significa aceptar cada cadena que es una palabra del LR y rechazar las que no pertenecen a dicho lenguaje.

Para realizar esta tarea de reconocimiento, el AF recorre, uno por uno, los caracteres que constituyen la cadena que ha sido leída. Cada carácter procesado produce un cambio de estado en el AF. Si al terminar de analizar todos los caracteres de la cadena, el AF se encuentra en un estado especial llamado estado final (o estado de aceptación), entonces se afirma que el AF ha reconocido a la cadena y que ésta es una palabra del lenguaje. En caso contrario, la cadena es rechazada porque no pertenece al LR tratado. Un AF puede tener varios estados finales.

El estado inicial (único en todo AF) es el estado en el que el AF se encuentra antes de comenzar su actividad.

Todo AF tiene asociado un dígrafo, llamado diagrama de transiciones (DT), que permite visualizar con mayor claridad el funcionamiento del AF. En este diagrama, los nodos del dígrafo se representan los diferentes estados del AF, y los arcos, que representan las transiciones entre los estados, informan cada cambio de estado que ocurre en el AF según el carácter leído de la cadena en estudio. Es por eso que las transiciones estarán etiquetadas con símbolos del alfabeto.

El estado inicial se distingue con un '–' en su superíndice. El estado final, con un '+' en su superíndice.

Cada AF reconoce un único LR.
Para cada LR, pueden existir muchos AFs que lo reconozcan.

1.2 AFs que reconocen LR Finitos *(ver p. 10 y 11)*

Los LRs finitos que se representan mediante ERs utilizan 3 operadores: concatenación, unión y potencia.

1.3 AFs que reconocen LR Infinitos *(ver p. 12, 13, 14, 15 y 16)*

Las ERs que representan LRs infinitos requieren la utilización del operador estrella. En un AF, la implementación del operador estrella aplicado a un sólo carácter consiste en un bucle sobre un solo estado. Cada transición tiene un estado de partida y un estado de llegada. En un bucle, el estado de partida coincide con el de llegada.

Un AF tiene un número finito de estados.
Si un AF reconoce a un LU sobre cierto alfabeto, ninguna cadena podrá ser rechazada.

1.4 AFs Determinísticos (AFDs) *(ver p. 16 y 17)*

La característica funcional de los AFDs es que, para cualquier estado en que se encuentre el autómata en un momento dado, la lectura de un carácter determina sin ambigüedades cuál será el estado de llegada en la próxima transición. Un AF es AFD cuando, dado un carácter leído por el autómata, éste transita desde un estado de partida a un estado de llegada preciso.

Un AFD es una colección de 3 conjuntos: un conjunto finito de estados (uno de los cuales se designa como el estado inicial y otros son designados como estados finales), un alfabeto de caracteres (con el que se forman las cadenas que serán procesadas por el AFD) y un conjunto finito de transiciones (que indican a qué estado debe moverse el AFD, para cada estado y para cada carácter del alfabeto).

1.4.1 Definición formal de un AFD *(ver p. 17 y 18)*

Formalmente, un AFD es una 5-upla (Q, Σ, T, q_0, F), donde:

- Q es un conjunto no finito de estados.
- Σ es el alfabeto de caracteres reconocidos por el autómata.
- $q_0 \in Q$ es el estado inicial único.
- $F \subseteq Q$ es el conjunto no vacío de estados finales.
- $T: Q \times \Sigma \rightarrow Q$ es la función de transiciones, representada por medio de la tabla de transiciones.

1.4.2 AFD Completo *(ver p. 18, 19 y 20)*

Un AFD es completo si cada estado tiene exactamente una transición por cada carácter del alfabeto. Es decir, un AFD es completo cuando su tabla de transiciones (TT) no tiene "huecos" o espacios vacíos.

Para completar los AFDs, eliminando los huecos de su TT, hay que llevar a cabo los siguientes pasos:

(1) Agregar un nuevo estado (denominado estado de rechazo).
(2) Reemplazar cada hueco que hay en la TT por una transición a este nuevo estado.
(3) Incorporar una nueva entrada en la TT para el estado de rechazo, en la que se representarán bucles para todos los caracteres del alfabeto. Con estos bucles se informa que una vez que el AF se sitúa en el estado de rechazo, no puede salir de él.

Si dos AFDs reconocen el mismo LR, entonces son equivalentes.

1.5 Una aplicación: validación de cadenas *(ver p. 20, 21 y 22)*

1.6 Una extensión de la aplicación: una secuencia de cadenas *(ver p. 22, 23, 24 y 25)*

2. AFs con Pila (AFP) *(ver p. 27)*

Los AFP son más poderosos que los AFs porque reconocen a los LICs, además de reconocer a los LRs.

El AFP tiene una memoria en forma de pila que permite almacenar, retirar y consultar cierta información que será útil para reconocer a los LICs. Es decir, mientras que un AF sólo tiene control sobre los caracteres que forman la cadena a analizar, un AFP controla los caracteres a analizar y la pila.

2.1 Definición formal de un AFP *(ver p. 27 y 28)*

Un AFP está constituido por:

- Un flujo de entrada, infinito en una dirección (contiene la secuencia de caracteres que debe analizar).
- Un control finito (formado por estados y transiciones etiquetadas).
- Una pila abstracta (representada como una secuencia de caracteres tomados de cierto alfabeto, diferente al alfabeto sobre el que se construye un LIC reconocido por un AFP).

Un AFP se define formalmente como la 7-upla M = (E, A, A', T, e0, p0, F), donde:

- E es un conjunto finito de estados.
- A es el alfabeto de entrada (cuyos caracteres se utilizan para formar la cadena a analizar).
- A' es el alfabeto de la pila.
- T es la función de transiciones entre los estados
- e0 es el estado inicial.
- p0 es el símbolo inicial de la pila (el que indica que la pila no tiene símbolos).
- F es el conjunto de estados finales.

Un AFP puede reconocer un LIC por estado final (como en los AFs) o por pila vacía.

2.2 AFP Determinístico (AFPD) *(ver p. 29 y 30)*

Un AFPD está formado por una colección de estos 8 elementos:

- Un alfabeto de entrada A (son los símbolos con los que se forman las palabras del LIC a reconocer; hay un símbolo especial que sólo puede aparecer al final de la cadena en estudio para indicar su terminación: fdc, fin de cadena).
- Un flujo de entrada infinito en una dirección (contiene la cadena a analizar).
- Un alfabeto A' de símbolos de la pila (hay un símbolo especial que indica que la pila está lógicamente vacía: $).

- Una pila infinita en una dirección (inicialmente está vacía).
- Un conjunto finito y no vacío de estados E.
- Un único estado inicial.
- Un conjunto de estados finales.
- Una función de transiciones entre los estados.

2.2.1 La tabla de movimientos (TM) *(ver p. 30 y 31)*

La TM en un AFP o en un AFPD es parecida a la TT en un AF, sólo que en la TM, en lugar de hablar de "estado", debemos hablar del par "estado y símbolo en el tope de la pila".

2.3 AFPDs y expresiones aritméticas *(ver p. 32)*

Las expresiones aritméticas que usen paréntesis para determinar un orden de evaluación constituyen un LIC existente en la mayoría de los LPs. Por lo tanto, se podrá construir un AFPD que las reconozca, que determine si una secuencia de caracteres (que constituye una supuesta expresión matemática) es sintácticamente correcta o no.

3. Introducción al proceso de compilación *(ver p. 33)*

3.1 Conceptos básicos *(ver p. 33)*

Un compilador es un programa que lee un programa escrito en un lenguaje fuente (programa fuente) y lo traduce a un programa equivalente en un lenguaje objeto (programa objeto; normalmente más cercano al lenguaje de máquina).

El proceso de compilación está formado por 2 partes:

- El análisis: a partir del programa fuente crea una representación inmediata del mismo.
- La síntesis: a partir de esta representación intermedia, construye el programa objeto.

3.1.1 El análisis del programa fuente *(ver p. 33, 34, 35 y 36)*

En la compilación, el análisis está formado por 3 fases:

- El análisis léxico (realizado por el scanner): detecta los diferentes elementos básicos (lexemas) que constituyen un programa fuente (identificadores, palabras reservadas, constantes, operadores y caracteres de puntuación). El análisis léxico sólo se ocupa de los LRs que forman parte del LP. Los LRs a los que pertenecen estos lexemas se denominan tokens o categorías léxicas. Para el análisis léxico, entran caracteres y salen tokens.
- El análisis sintáctico (realizado por el parser): trabaja con los tokens detectados durante el análisis léxico. Esta etapa de análisis (como sí conoce la sintaxis del LP) tiene la capacidad de determinar si las construcciones que componen el programa son sintácticamente correctas, sin poder determinar si el programa es sintácticamente correcto en su totalidad. Dada una definición sintáctica formal, como la que brinda una GIC, el parser recibe tokens del scanner y los agrupa en unidades, tal como está especificado por las producciones de la GIC utilizada. Mientras se realiza el análisis sintáctico, el parser verifica la corrección de la sintaxis y, si encuentra un error sintáctico, el parser emite el diagnóstico correspondiente. En la medida que las estructuras semánticas son reconocidas, el parser llama a las correspondientes rutinas semánticas que realizarán el análisis semántico.
- El análisis semántico realiza diferentes tareas, completando lo que hizo el análisis sintáctico. Una de estas importantes tareas es la de "verificación de tipos", para que cada operador trabaje sobre operandos permitidos según la especificación del LP. Las rutinas semánticas llevan a cabo 2 funciones: chequear la semántica estática de cada construcción (verificar que la construcción analizada sea legal y que tenga un significado) y traducir, si es que la construcción es semánticamente correcta (generar el código para una máquina virtual que, a través de sus instrucciones, implementa correctamente la construcción analizada). Cuando una producción es reconocida por el parser, éste llama a las rutinas semánticas asociadas para chequear la semántica estática (compatibilidad de tipos) y luego realizar las operaciones de traducción, generando las instrucciones para la máquina virtual.

La tabla de símbolos es una estructura de datos compleja que se usa para el almacenamiento de todos los identificadores del programa a compilar. Cada elemento de la TS está formado por una cadena y sus atributos. En general, la TS es muy utilizada durante toda la compilación y, específicamente, en la etapa de análisis por las rutinas semánticas.

Una definición completa de un LP debe incluir las especificaciones de su sintaxis y de su semántica. La sintaxis se divide, normalmente, en componentes independientes del contexto y en componentes sensibles al contexto. Las restricciones sensibles al contexto son manejadas como situaciones semánticas. En consecuencia, el componente semántico de un LP se divide en 2 clases: semántica estática y semántica en tiempo de ejecución. La semántica estática de un LP define las restricciones sensibles al contexto que deben cumplirse para que el programa sea considerado correcto (algunas son, por ejemplo: que todos los identificadores estén declarados, que los operadores y los operandos tengan tipos compatibles, que las rutinas –funciones y procedimientos– deban ser llamadas con el número apropiado de argumentos).

4. Análisis Léxico, Análisis Sintáctico y Análisis Semántico *(ver p. 57)*

4.1 Introducción a la Tabla de Símbolos (TS) *(ver p. 57)*

La tabla de símbolos es un conjunto de estructuras de datos que se utiliza para contener todos los identificadores del programa fuente que se está compilando, junto con los atributos que posee cada identificador. En el caso que estos identificadores sean nombres de variables, sus atributos son: tipo y ámbito (la parte del programa donde tiene validez). En el caso que los identificadores sean nombres de rutinas (funciones o procedimientos), algunos atributos son cantidad de parámetros y tipo de cada uno, métodos de transferencia, tipo del valor que retorna. Estos atributos se codifican mediante números enteros o valores por enumeración.

4.2 El Análisis Léxico *(ver p. 58, 59, 60 y 61)*

El análisis léxico, realizado por el scanner, es el proceso que consiste en recorrer el flujo de caracteres que forman el programa fuente, detectar los lexemas que componen este programa, y traducir la secuencia de estos lexemas en una secuencia de tokens cuya representación es más útil para el resto del compilador.

El análisis léxico es un proceso que se realiza sobre palabras (los lexemas) de LRs (los tokens). Cuando estos LRs son infinitos, sus lexemas son obtenidos recién cuando se detecta un carácter espurio llamado centinela. En cambio, si los LRs son finitos, pueden o no requerir un centinela para obtener el lexema correspondiente.

4.2.1 Recuperación de errores *(ver p. 61 y 62)*

Ciertos caracteres son ilegales en ANSI C porque no pertenecen a los alfabetos de ninguno de los LRs que forman este LP. El scanner se puede recuperar de estos errores de varias maneras. La más simple es descartar el carácter inválido, mostrar un mensaje de error adecuado, pasar un código de error al parser y continuar el análisis léxico a partir del siguiente carácter. Obviamente, el resultado final de todo el proceso ya no será una compilación correcta.

4.3 El Análisis Sintáctico *(ver p. 62)*

El análisis sintáctico es llevado a cabo por el parser, quien verifica si los tokens que recibe del scanner forman secuencias válidas, según la gramática sintáctica del LP correspondiente. El parser está formado por un grupo de rutinas que convierte el flujo de tokens en un árbol de análisis sintáctico, generalmente implícito, que representa de modo jerárquico a la secuencia analizada, donde los tokens que forman la construcción son las hojas del árbol.

4.3.1 Tipos de Análisis Sintácticos y de GICs *(ver p. 63, 64 y 65)*

Al derivar una secuencia de tokens, si existe más de un no terminal en una cadena de derivación, debemos elegir cuál es el próximo no terminal que se va a expandir. Es decir, cuál será reemplazado por su lado derecho de la producción. Para eso se utilizan 2 tipos de derivaciones que determinan con precisión cuál será el no terminar a tratar: la derivación a izquierda y la derivación a derecha.

Una derivación es una forma de mostrar cómo una construcción del flujo de tokens (una declaración, una expresión, una sentencia, el programa completo...) se puede obtener a partir de una gramática sintáctica dada. Este proceso puede ser: derivación a izquierda (ocurre cuando siempre se reemplaza el primer no terminal que se encuentre en una cadena de derivación leída de izquierda a derecha) o derivación a derecha (ocurre cuando siempre reemplaza el último no terminal de la cadena de derivación leída de izquierda a derecha).

Es importante destacar la relación que existe entre el tipo de análisis sintáctico y el tipo de derivación. El análisis sintáctico descendente produce una derivación por izquierda, que comienza en el no terminal, llamado axioma, y finaliza con los terminales que forman la construcción analizada. La derivación por derecha se utiliza en el análisis sintáctico ascendente, pero de una manera especial (hay que tener en cuenta que la derivación, sea por izquierda o por derecha, siempre comienza en el axioma de la GIC y finaliza con una secuencia de terminales). Un parser ascendente utiliza una derivación a derecha, pero en orden inverso: la última producción aplicada en la derivación a derecha es la primera producción que es descubierta, mientras que la primera producción utilizada, la que involucra al axioma, es la última producción en ser descubierta. Podríamos decir que reduce el árbol de análisis sintáctico hasta llegar al axioma.

4.3.3.1 Gramáticas LL y LR *(ver p. 65 y 66)*

Un análisis sintáctico LL consiste en analizar el flujo de tokens de izquierda a derecha, por medio de una derivación por izquierda. Una gramática LL es una GIC que puede ser utilizada en un análisis sintáctico LL. No todas las gramáticas son LL, pero afortunadamente la mayoría de los LPs pueden describir sus respectivas sintaxis por medio de gramáticas LL. Una gramática especial es la llamada LL(1), que puede ser utilizada por un parser LL con un solo símbolo de preanálisis. Esto es: si un no terminal tiene varias producciones, el parser puede decidir cuál lado derecho debe aplicar con solo conocer el próximo token del flujo de entrada.

Otro tipo de análisis sintáctico es el análisis sintáctico LR, que consiste en analizar el flujo de tokens de izquierda a derecha, pero por medio de una derivación a derecha, aunque utilizada a la inversa: la última producción expandida en la derivación será la primera en ser reducida.

En general, un parser LL(1) puede ser construido por un programador a partir de una GIC adecuada. En cambio, un parser LR(1) requiere la utilización de un programa especial tipo yacc.

4.3.2 Gramáticas LL(1) y aplicaciones *(ver p. 66, 67 y 68)*

4.3.3 Obtención de gramáticas LL(1) *(ver p. 68)*

4.3.3.1 Factorización a izquierda *(ver p. 68 y 69)*

4.3.3.2 Eliminación de la recursividad a izquierda *(ver p. 69 y 70)*

4.3.3.3 Símbolos de preanálisis y el conjunto Primero *(ver p. 70)*

4.3.3.3.1 Obtención del conjunto Primero *(ver p. 71)*

4.3.3.3.2 Obtención del conjunto Siguiente *(ver p. 71 y 72)*

4.3.3.4 La función predice y el ASDR *(ver p. 73)*

4.3.4 Usando una pila para implementar un parser predictivo *(ver p. 73, 74, 75 y 76)*

Un análisis sintáctico predictivo puede ser implementado usando una GIC, una pila y el siguiente algoritmo, en el que la pila es inicializada con el axioma de la GIC.

(1) Si la pila está vacía y el flujo de tokens de entrada llega al fin de texto, el análisis sintáctico se completó.
(2) Si el símbolo que está en el tope de la pila es un no terminal, hay que reemplazarlo por su lado derecho pero invirtiendo el orden de los símbolos. Volver a (1).
(3) Si el símbolo en el tope de la pila es un terminal, debe coincidir con el símbolo de preanálisis (token): si no lo es, ha ocurrido un error sintáctico; si lo es, hay que eliminar ese terminal y avanzar al próximo token del flujo de entrada. Volver a (1).

Un parser que puede implementarse tan sencillamente se lo conoce como parser predictivo, porque puede predecir cuál será el próximo token y, si la gramática es adecuada, puede expandir inmediatamente el no terminal que está en tope de la pila y así seguir avanzando con este proceso. Para poder construir un parser predictivo, la GIC debe ser LL(1).

4.3.5 Lenguaje de los Paréntesis Anidados - AFP - Parser dirigido por una tabla *(ver p. 76 y 77)*

Las expresiones con operadores infijos, como tienen la mayoría de los LPs, tienen cierta complejidad para ser descriptas sintácticamente mediante una GIC. Esto se debe a que la GIC también debe representar la precedencia y asociatividad de cada operador y, además, debe mostrar el correcto uso de los paréntesis.

Un parser dirigido por una tabla está basado en un AFP. Este parser es dirigido por una máquina de estados y utiliza una pila para mantener un registro del progreso de esta máquina de estados.

4.3.6 El problema del If-Then-Else *(ver p. 77 y 78)*

Casi todas las construcciones de un LP pueden especificarse con una GIC LL(1). Sin embargo, hay una importante excepción: la construcción **if-then-else**. El problema es que como la cláusula **else** es opcional, puede haber más partes **then** que partes **else** y, por lo tanto, la correspondencia entre ambas partes no es única.

4.3.7 Análisis Sintáctico Ascendente (ASA) *(ver p. 78)*

4.3.7.1 Cómo funciona *(ver p. 78, 79 y 80)*

El ASA es realizado por un parser ascendente, que requiere un conjunto de estados para guiarlo y una pila para recordar el estado actual. El parser construye el árbol sintáctico desde las hojas hacia la raíz. El ASA trabaja así:

(1) Si los elementos que están en la parte superior de la pila forman el lado derecho de una producción, se quitan (*pop*) y se reemplazan por el no terminal del lado izquierdo (*push*). A este proceso se lo llama proceso de reducción (reemplaza el lado derecho de una producción en la pila con el correspondiente no terminal del lado izquierdo).
(2) Caso contrario, *push* el símbolo de entrada actual (un token) en la pila y avance en el flujo de tokens de entrada (*input*). Esta operación se conoce como desplazamiento (proceso en el que un token se mueve desde el *input* a la pila) porque desplaza el siguiente token desde el flujo de entrada a la pila.
(3) Si la operación previa consistió en una reducción que resultó con el no terminal objetivo (el axioma) en el tope de la pila, el *input* es aceptado y el ASA se ha completado. De lo contrario, se vuelve al paso (1).

4.3.7.2 Recursividad *(ver p. 80)*

Para realizar el ASD, la gramática no puede ser recursiva a izquierda. Esta situación se invierte al realizar un ASA: la recursividad a izquierda siempre debe utilizarse para aquellas listas en las que la asociatividad no es un tema importante o debe ser a izquierda. Por otro lado, la recursividad a derecha debe ser usada sólo cuando se requiere la asociatividad a derecha.

4.3.7.3 Otra visión: la implementación de un parser ascendente como un AFP *(ver p. 80, 81)*

4.3.7.4 El uso de yacc *(ver p. 81 y 82)*

4.4 Análisis Semántico *(ver p. 83 y 84)*

Desde el punto de vista de etapa de análisis del compilador, el análisis semántico consiste básicamente en la realización de tareas de verificación y de conversión que no puede realizar el análisis sintáctico. El análisis semántico debe verificar que se cumplan todas las reglas sensibles al contexto y, para ello, utiliza mucho la tabla de símbolos.

4.4.1 En ANSI C: Derivable vs Sintácticamente correcto *(ver p. 84)*

Volumen III – ALGORITMOS

1. AFDs y AFNs *(ver p. 7)*

1.1 Autómatas Finitos Determinísticos (AFDs) (Vol. II – 1.4) *(ver p. 7 y 8)*

1.1.1 Definición formal de un AFD (Vol. II – 1.4.1) *(ver p. 8 y 9)*

1.1.2 AFD completo (Vol. II – 1.4.2) *(ver p. 9 y 10)*

1.2 Autómata Finito No Determinístico (AFN) *(ver p. 11 y 12)*

En un AFN cualquier estado del autómata tiene 0, 1 o más transiciones, por el mismo carácter del alfabeto.

El AFN tiene este nombre porque habrá por lo menos un estado y un carácter para los que el autómata deberá elegir, entre dos o más transiciones, cuál es el camino a seguir.

1.2.1 Definición formal de un AFN *(ver p. 12)*

La definición formal de un AFN se similar a la definición formal de un AFD, pero cambia la función de transiciones.

1.2.2 AF con Transiciones-ε *(ver p. 12, 13 y 14)*

El AF con transiciones-ε es un segundo modelo de AFN y se caracteriza por la existencia de 1 o más transiciones que ocurren sin que el autómata lea el próximo carácter de la cadena que está analizando. Una transición-ε representa un cambio de estado repentino, sin que intervenga ningún carácter del alfabeto. Su definición formal es similar a la de un AFN, con otra diferencia en la función de transición.

La TT de un AFN con transiciones-ε debe tener una columna por cada carácter del alfabeto, más una columna por el símbolo ε.

2. De la ER al AF *(ver p. 15 y 16)*

2.1 Algoritmo de Thompson *(ver p. 16)*

El algoritmo desarrollado por Ken Thompson consiste en:

(1) Desmembrar la ER de partida en sus componentes básicos (en caracteres, operadores y ε).
(2) Generar un AF básico por cada carácter o símbolo ε que forme parte de la ER.
(3) Componer estos autómatas básicos según los operadores existentes en la ER, hasta lograr el AF que reconoce a la ER dada.

2.1.1 Autómata para cada carácter y para el símbolo ε *(ver p. 16 y 17)*

2.1.2 Autómata para la unión *(ver p. 17 y 18)*

2.1.3 Autómata para la concatenación *(ver p. 18 y 19)*

2.1.4 Autómata para la clausura de Kleene *(ver p. 19 y 20)*

2.1.5 Características generales de un AFN "por Thompson" *(ver p. 20 y 21)*

2.2 Utilización de Thompson en resolución no algorítmica *(ver p. 21, 22, 23 y 24)*

3. Del AFN al AFD *(ver p. 25)*

Un AF obtenido a partir de la descripción de un LR, mediante una frase o desde una ER, puede ser un AFD o un AFN. Si el AF se construye utilizando el algoritmo de Thompson, será un AFN con transiciones-ε.

Un autómata es un AFN porque: hay un estado del que parten 2 o más transiciones etiquetadas con un mismo carácter del alfabeto, o tiene transiciones-ε, o ambas cosas.

Para todo AFN existe un AFD equivalente.

3.1 Definiciones preliminares *(ver p. 25)*

El algoritmo para construir un AFD a partir de un AFN se basa en los conocimientos del conjunto clausura-ε y el conjunto Hacia.

3.1.1 Clausura-ε de un estado *(ver p. 25 y 26)*

Sea **q** un estado de un AFN, entonces la **clausura-ε (q)** es el conjunto de estados formado por **q** y todos aquellos estados a los cuales se llega, desde **q**, utilizando únicamente transiciones-ε.

El conjunto clausura-ε de un estado nunca puede ser vacío porque contiene, como mínimo a su propio estado.

3.1.2 Clausura-ε de un conjunto de estados *(ver p. 27)*

Sea **R** un conjunto de estados. Entonces la **clausura-ε (R)** es la unión de las clausuras-ε de los estados que componen el conjunto **R**.

3.1.3 El conjunto "hacia" *(ver p. 27)*

Es el conjunto de estados de llegada. Siendo **R** un conjunto de estados y **x** un símbolo del alfabeto, **hacia(R, x)** es el conjunto de estados a los cuales se transita por el símbolo **x**, desde los estados de **R** que tengan esa transición.

3.2 El algoritmo *(ver p. 27, 28, 29, 30 y 31)*

El algoritmo por el cual se construye un AFD a partir de un AFN recibe el nombre de algoritmo de clausura-ε. El AFD simulará, "en paralelo", todas las transiciones que realiza el AFN del cual se parte, para cualquier cadena de entrada que el autómata deba procesar.

Para obtener el AFD se necesitan determinar tres cosas:

- Su estado inicial.
- Sus estados finales.
- Su tabla de transiciones.

El algoritmo comienza con la obtención del estado inicial del AFD, a partir del cual, determinará los demás estados de dicho AFD y, con ellos, construirá la tabla de transiciones del nuevo autómata y también resolverá qué estados son finales. Por definición, el estado inicial del AFD es la clausura-ε del estado inicial del AFN dato, mientras que los estados finales del AFD son todos aquellos conjuntos de estados del AFN que contienen, por lo menos, un estado final.

Finalmente, la tabla de transiciones se construye así:

(1) Se obtiene el estado inicial del AFD.
(2) Se agrega este estado a la primera columna de la tabla.
(3) Para cada símbolo del alfabeto, se calcula el **conjunto Hacia** del estado que se acaba de agregar a la primera columna de la tabla.
(4) Se determinan nuevos estados del AFD, por medio de la clausura-ε de cada *conjunto Hacia* recién obtenido, y son incorporados a la tabla, en las columnas que les corresponden.
(5) Si un nuevo estado obtenido en el punto anterior no existe todavía en la primera columna de la tabla transiciones, se lo agrega.
(6) Se repiten los pasos (3) al (5) hasta que no surjan nuevos estados.

4. Dos operaciones fundamentales con AFs *(ver p. 33)*

Un lenguaje es regular si es reconocido por algún AF.

4.1 Operaciones con AFDs *(ver p. 33 y 34)*

4.1.1 Complemento de un AFD *(ver p. 34 y 35)*

El complemento de un AFD ya es un AFD que se obtiene invirtiendo los estados finales y no finales:

- Todo estado no final del AFD dato será un estado final del AFD complemento.
- Todo estado final del AFD dato será un estado no final del AFD complemento.

Sea un AFD $\mathbf{M_1 = (Q_1, \Sigma, T_1, q_1, F_1)}$, el autómata $\mathbf{M = M_1^C}$ se define formalmente así: $\mathbf{M = (Q_1, \Sigma, T_1, q_1, Q_1 - F_1)}$. La única diferencia radica en el conjunto de estados finales.

4.1.2 Intersección de dos AFDs *(ver p. 36 y 37)*

El autómata intersección de dos AFDs también es un AFD, y reconoce las palabras comunes a los LRs reconocidos por los dos AFDs datos. Los estados del AFD intersección son pares ordenados de estados, uno de cada AFD dato.

Sean dos AFDs: $\mathbf{M_1 = (Q_1, \Sigma, T_1, q_1, F_1)}$ y $\mathbf{M_2 = (Q_2, \Sigma, T_2, q_2, F_2)}$. Entonces, el autómata $\mathbf{M = M_1 \cap M_2 = (Q, \Sigma, T, q_0, F)}$ tiene estos componentes: $\mathbf{Q = Q_1 \times Q_2}$; $\mathbf{q_0 = (q_1, q_2)}$; $\mathbf{F = F_1 \times F_2}$; **T** es: $\mathbf{T((p, q), x) = (T_1(p, x), T_2(q, x)) \; \forall p \in Q_1 \wedge \forall p \in Q_2}$.

En la construcción del AFD intersección hay que tener en cuenta que todos los estados son pares ordenados y que, normalmente, muchos posibles estados del AFD intersección serán estados erróneos.

Consecuentemente, una buena técnica para hallar el AFD intersección es:

(1) Obtener el estado inicial y colocarlo en la primera final de la tabla de transiciones.
(2) Determinar las transiciones desde el estado inicial y agregar nuevas entradas en la tabla a medida que surjan nuevos estados, siempre que éstos no sean estados erróneos.
(3) Repetir este proceso para cada nuevo estado, hasta que no se puedan crear más estados.
(4) Recordar que los únicos estados finales son aquellos en los que ambos estados del par ordenado son finales.

5. Del AF a la ER *(ver p. 39)*

5.1 Depuración del AF *(ver p. 39 y 40)*

Hay ciertas operaciones que producen autómatas con 2 tipos de estados erróneos:

- Estados inalcanzables (los que no pueden ser alcanzados desde el estado inicial).
- Estados de rechazo (los que no conducen a un estado final).

La depuración del AF es el primer paso del algoritmo que consiste en detectar y eliminar todos los estados erróneos. Una vez depurado el AF, el segundo paso del algoritmo consiste en establecer un sistema de ecuaciones (donde cada ecuación describe el comportamiento de un estado del AF) que tendrá tantas ecuaciones como estados haya en el AF ya depurado. Cuando este sistema de ecuaciones sea resuelto, su resultado será la ER buscada.

5.2 Resolución del sistema de ecuaciones *(ver p. 40, 41 y 42)*

Se parte de la tabla de transición depurada y habrá tantas ecuaciones como estados tenga el AF.

En términos generales, cada ecuación tiene:

- En su lado izquierdo, el estado cuyo comportamiento es descripto por una ecuación.
- En su lado derecho, la unión de términos que representan las transiciones que parten del estado mencionado en el lado izquierdo (cada término se forma, en general, con el carácter que etiqueta a la transición concatenado con el estado de llegada; excepto cuando se desarrolla la ecuación de un estado final).

5.3 Reducciones *(ver p. 42, 43 y 44)*

El proceso de reducción se debe realizar en una ecuación que tiene 1 o más bucles. En estas ecuaciones, el estado que figura en el lado izquierdo de la ecuación también se encuentra en el lado derecho. A estas ecuaciones se las llaman ecuaciones recursivas. El formato general de una ecuación recursiva es $\mathbf{e = \alpha e + \beta}$, donde: **e** representa un estado; **α** es una ER; y **β** es una expresión que puede estar formada por caracteres del alfabeto, el símbolo **ε** y estados.

6. Obtención del AFD Mínimo *(ver p. 45)*

El AFD mínimo es el AFD con la mínima cantidad de estados que reconoce a un LR. Es por eso que se lo considera como el único autómata óptimo asociado a la aceptación de determinado LR. La importancia de obtener el AFD mínimo tiene tres aplicaciones inmediatas:

- Determinar si 2 o más AFDs son equivalentes.
- Probar la equivalencia de 2 o más ERs.
- El AFD mínimo es el que tiene la tabla de transiciones más reducida (con menor cantidad de filas), hecho que beneficia la implementación del AFD mediante un programa de computadora.

Si de 2 o más AFDs se obtienen el mismo AFD mínimo, entonces esos AFDs son equivalentes. *Si 2 o más AFDs son reconocidos por el mismo AFD mínimo, entonces esos AFDs son equivalentes.*
Si dos estados pertenecen a la misma clase y tienen el mismo comportamiento, entonces esos estados son equivalentes.

6.1 El Algoritmo *(ver p. 45, 46, 47, 48, 49, 50, 51, 52 y 53)*

7. Máquina de Turing (MT) *(ver p. 55 y 56)*

Una máquina de Turing es un autómata determinístico con la capacidad de reconocer cualquier lenguaje formal y está formada por:

- Un alfabeto **A** de símbolos o caracteres del lenguaje a reconocer.
- Una cinta infinita dividida en una secuencia de celdas, cada una de las cuales contiene un carácter o un blanco (acá va la cadena a analizar, el resto de las celdas contiene blancos).
- Una cabeza de cinta que puede, en un solo paso, leer el contenido de una celda de la cinta, reemplazarlo con otro carácter o dejar el mismo, y reposicionarse para apuntar a la celda contigua (ya sea la de la derecha, avanzando, o la de la izquierda, retrocediendo).
- Un alfabeto **A'** de símbolos o caracteres que pueden ser escritos en la cinta por la cabeza de la cinta.
- Un conjunto finito de estados (que incluye un estado inicial, donde comienza la ejecución, y un conjunto de estados finales posiblemente vacío que producen la terminación de la ejecución cuando se llega a alguno de ellos).
- Un programa (un conjunto de reglas que indican, en función del estado en que se encuentra la MT y del carácter leído por la cabeza de la cinta, qué carácter escribir en la cinta en la misma posición, en qué dirección se debe mover la cabeza de la cinta y a qué estado debe realizar la transición).