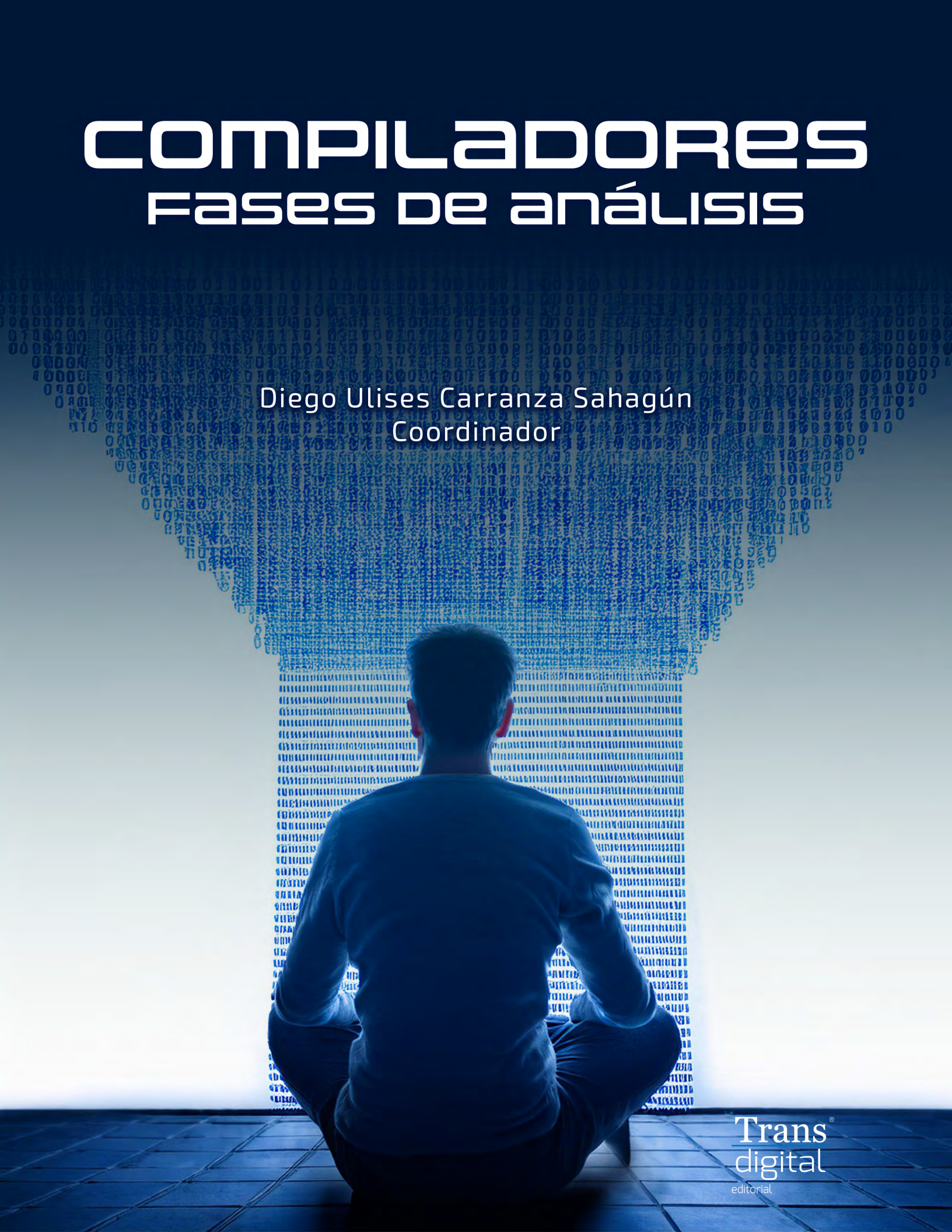
COMPILADORES
Fases de análisis

Diego Ulises Carranza Sahagún
Coordinador



Trans digital
editorial

Compiladores

Fases de análisis

Diego Ulises Carranza Sahagún

Coordinador

Transdigital®
editorial

2024.

Título original: Compiladores: fases de análisis / Diego Ulises Carraza Sahagún — Ciudad de Querétaro, México: Editorial Transdigital, 2024 — 81 páginas.

International Standard Book Number (ISBN): 978-607-26754-0-7.

Digital Object Identifier (DOI): https://doi.org/10.56162/transdigitalb44

Clasificación DEWEY. Materia: 004 - Ciencia de los computadores.

Tipo de Contenido: Libros universitarios.

Clasificación thema: U - Computación y tecnologías de las información.

Tipo de soporte: libro digital descargable. Formato: PDF. Tamaño: 3.9 Mb.

Esta obra ha sido dictaminada por pares académicos expertos con el método de doble ciego.

Trans digital editorial

Sociedad de Investigación sobre Estudios Digitales, S. C. Circuito Altos Juriquilla 1132. Colonia Altos Juriquilla. C. P. 76230, Juriquilla, Querétaro, México. +52 (442) 301 32 38. aescudero@editorial-transdigital.org
www.editorial-transdigital.org

Redes sociales:

https://www.linkedin.com/company/transdigital-mx/
https://twitter.com/TransdigitalMx
https://www.facebook.com/transdigital.mx/
https://www.instagram.com/transdigital.mx
https://www.youtube.com/@transdigitalmx
https://wa.me/message/PFGE567UBNMOE1

CULTURA INDAUTOR

Registro en el Padrón Nacional de Editores como agente editor Sociedad de Investigación sobre Estudios Digitales, S. C., con el Dígito Identificador 978-607-99594.

CONAHCYT

Registro Nacional de Instituciones y Empresas Científicas y Tecnológicas del Consejo Nacional de Humanidades, Ciencias y Tecnologías (CONAHCyT) con el folio: RENIECYT 2400068.

Sugerencia de referencia en APA 7a. edición:

Carraza Sahagún, D. U. (2024) (Coordinador). *Compiladores: fases de análisis.* Editorial Transdigital. https://doi.org/10.56162/transdigitalb44

“Un programador que escriba un código limpio, entiende perfectamente el problema antes de escribir el código”
Robert C. Martin

“Tratar de ser más inteligente que un compilador elimina gran parte del propósito de usar uno”
Kernighan and Plauger

“Un lenguaje que no afecte a la forma en que piensas al programar no merece la pena conocerlo”
Alan J. Peris

Contenido

PREFACIO

Un compilador traduce un programa fuente escrito en un lenguaje de alto nivel, a un programa objetivo en un lenguaje de bajo nivel o a lenguaje máquina. Este proceso se realiza en dos partes: la parte de análisis y la parte de síntesis.

El presente libro se enfoca en la parte de análisis, que permite revisar el programa fuente en sus aspectos léxicos, sintácticos y semánticos. Está orientado para un curso inicial de compiladores, dirigido a estudiantes de Ciencias Computacionales.

Se muestran técnicas para la implementación de los diferentes tipos de análisis de una manera básica, utilizando ejemplos y con una sección de ejercicios y actividades para los estudiantes. Estos ejercicios y actividades pueden ser profundizados con la guía y el asesoramiento del facilitador o profesor de la materia.

Capítulo 1. Introducción y conceptos

Kleophé Alfaro Castellanos
José Ávila Paz

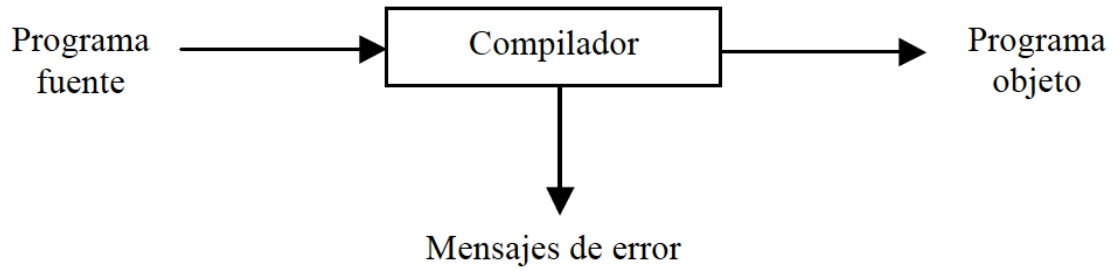
1.1. Introducción

Este libro tiene la intención de presentar conceptos y actividades prácticas relacionados con el proceso de compilación con un enfoque en las fases de análisis que permitan al usuario conocer los conceptos básicos y las funciones de un compilador.

1.2. Compiladores

Un compilador es un programa o software que recibe como entrada un *programa fuente* escrito en un lenguaje de alto nivel (lenguaje fuente), y lo traduce a un *programa objeto* u *objetivo* equivalente en un lenguaje máquina o un lenguaje de bajo nivel (Figura 1.1). Como parte importante de este proceso de traducción, el compilador informa a su usuario de la presencia de errores en el programa fuente.

Figura 1.1
Compilador



En el proceso de compilación hay dos partes: análisis y síntesis. La parte del análisis lee el programa fuente, verifica su estructura, lo divide en sus elementos componentes y crea una representación intermedia del programa fuente. La parte de la síntesis construye el programa objeto deseado a partir de la representación intermedia.

Durante el análisis, se determinan los tipos de componentes léxicos y las operaciones que implica el programa fuente y se registran en una estructura jerárquica llamada *árbol*. A menudo se usa una clase especial de árbol llamada *árbol sintáctico*, donde cada nodo

representa una operación y los hijos de un nodo son los argumentos de la operación. En el presente libro nos enfocamos en esta fase de análisis.

1.3. Fases de un compilador

Conceptualmente, un compilador opera en fases, cada una de las cuales transforma al programa fuente de una representación en otra (Figura 1.2).

Figura 1.2
Fases del proceso de compilación

Análisis léxico

También llamado lineal o lexicográfico es en el que la cadena de caracteres que constituye el programa fuente, se lee de izquierda a derecha y se agrupa en componentes léxicos (*tokens*), que son secuencias de caracteres que tienen un significado colectivo.

El análisis léxico o de exploración tiene como función principal detectar los lexemas del programa fuente y arroja como resultado un listado de componentes léxicos encontrados en el programa fuente. Los errores que se pueden observar en este análisis son de escritura, es decir, que tengamos símbolos o caracteres no válidos para el lenguaje de programación en que está escrito el programa fuente.

Por ejemplo, en el análisis léxico los caracteres de la proposición de asignación

```
p = x + y * 60
```

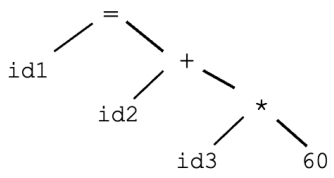
se agruparían en los componentes léxicos siguientes:

1. El identificador *p*.
2. El operador de asignación (=)
3. El identificador *x*
4. El operador de suma (+)
5. El identificador *y*
6. El símbolo de multiplicación (*)
7. La constante numérica *60*

Análisis sintáctico

Denominado también análisis jerárquico, tiene como función agrupar los componentes léxicos del programa fuente en frases gramaticales que el compilador utiliza para sintetizar la salida. Por lo general, las frases gramaticales del programa fuente se representa mediante un árbol de análisis sintáctico (Figura 1.3).

Figura 1.3
*Árbol sintáctico para p = x + y * 60*



Análisis semántico

La fase de análisis semántico revisa el programa fuente para tratar de encontrar errores semánticos y reúne la información sobre los tipos de datos para la fase posterior de generación de código. Se realizan ciertas revisiones para asegurar que los componentes de un programa se ajustan de un modo significativo. En esta fase, se utiliza la estructura jerárquica determinada por la fase de análisis sintáctico para identificar los operadores, operandos de expresiones y proposiciones.

Un componente importante del análisis semántico es la verificación de tipos de datos. En esta parte del análisis, el compilador verifica si cada operador tiene operandos permitidos por la especificación del lenguaje fuente.

Administración de la tabla de símbolos

Una función esencial de un compilador es registrar los identificadores utilizados en el programa fuente y reunir información sobre los distintos atributos de cada identificador como son: tipo, dirección de memoria, bloque en que fue definido, entre otros. Una tabla de símbolos es una estructura de datos que contiene un registro por cada identificador, con los campos para los atributos del identificador. Esta estructura de datos permite encontrar rápidamente el registro de cada identificador y almacenar o consultar rápidamente datos de ese registro.

Manejador de errores

Cada fase puede encontrar errores. Sin embargo, después de detectar un error, cada fase debe tratar de alguna forma ese error, para poder continuar la compilación permitiendo la detección de más errores en el programa fuente.

Las fases de análisis por lo general manejan una gran porción de errores detectables por el compilador. La fase de análisis léxico puede detectar errores donde los caracteres restantes de la entrada no forman ningún componente léxico del lenguaje, es decir, símbolos que no forman parte del alfabeto del lenguaje. Los errores donde las cadenas de componentes léxicos violan las reglas de estructura (sintaxis) del lenguaje, son determinados por la fase de análisis sintáctico. Durante el análisis semántico el compilador intenta detectar construcciones que tengan la estructura sintáctica correcta, pero que no tenga significado para la operación implicada.

Generación de código intermedio

Después de los análisis sintáctico y semántico, algunos compiladores generan una representación intermedia explícita del programa fuente. Esta representación intermedia debe tener dos propiedades importantes: debe ser fácil de producir y fácil de traducir al programa objeto.

Optimización de código

La fase de optimización de código trata de mejorar el código intermedio, de modo que resulte un código de máquina más rápido de ejecutar con la menor cantidad de recursos utilizados.

Generación de código

La fase final de un compilador es la generación de código objeto, que por lo general consiste en código de máquina relocalizable o en una versión de código en un lenguaje de bajo nivel. Las posiciones de memoria se seleccionan para cada una de las variables usadas por el programa. Después, cada una de las instrucciones intermedias se traduce a una secuencia de instrucciones de lenguaje máquina que ejecuta la misma tarea.

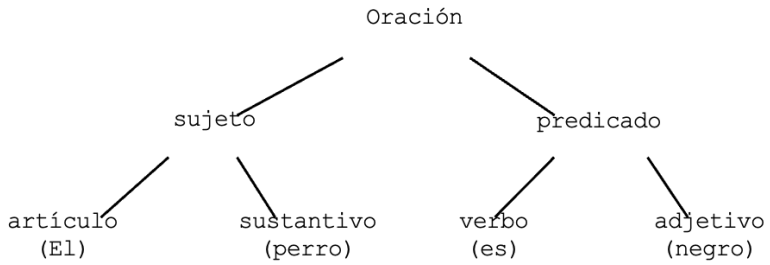
En un ejemplo del proceso de traducción, antes de aplicarlo a una instrucción en un lenguaje de programación en específico, se puede traducir una frase en español al inglés.

Teniendo la frase "El perro es negro" como fuente, al aplicarle la fase de análisis léxico obtenemos los *tokens*: *artículo sustantivo verbo adjetivo*; todas las palabras son válidas en español por lo que la frase es léxicamente correcta. Después, el listado de tokens obtenidos sirve como entrada para el análisis sintáctico. Es en esta fase donde se revisa el orden en que aparecen los componentes y se verifica mediante la gramática del lenguaje. Utilizando la gramática de la Figura 1.4, que s**ó**lo está elaborada para ejemplificar, se puede obtener el árbol sintáctico de la Figura 1.5; a consecuencia de que fue posible obtener el árbol sintáctico, se puede decir que la frase es válida sintácticamente.

Figura 1.4
Gramática informal de una oración en español

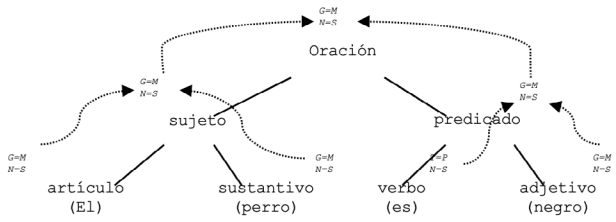
```
Oración     -> <sujeto><predicado>
sujeto      -> <sustantivo>|<artículo><sustantivo>|…
sustantivo  -> perro|gato|…
artículo    -> el|la|los|las|…
predicado   -> <verbo>|<verbo><adjetivo>|…
verbo       -> ser|estar|correr|…
adjetivo    -> blanco|negro|rápido|feo|…
.
.
.
```

Figura 1.5
Árbol sintáctico



Enseguida, se analiza semánticamente la frase, que en el caso del español se revisa género (masculino, femenino) y número (singular, plural). Esta revisión se puede hacer usando atributos en los nodos del árbol y aplicando reglas semánticas del lenguaje. En la Figura 1.6 se muestra el árbol sintáctico adornado con los atributos G (género) y N (número) con sus respectivos valores en los nodos. Las flechas indican la precedencia de obtención de los valores de los atributos, que en este ejemplo nos indica que la oración es género masculino y número singular; cabe mencionar que en el componente verbo no se tiene el atributo género, pero se tiene el atributo T, que indica la conjugación del verbo, que en este caso es *Presente*.

Figura 1.6
Árbol sintáctico con atributos para análisis semántico



Después de las fases de análisis, y una vez verificado que no hay errores, se procede a la fase de generación de código intermedio. En esta fase se puede tomar de la frase, palabra por palabra, y buscar en un diccionario su traducción al inglés y se obtiene como resultado: “The dog to be black”. Evidentemente se está tomando la traducción literal del verbo sin conjugación con la intención de recalcar las siguientes fases. Optimizando esa primer frase resultante, se hace la conjugación del verbo en tercera persona y singular, obteniéndose

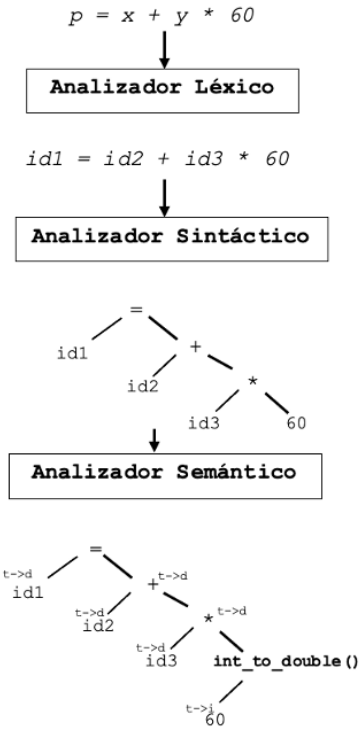
como resultado final de la fase de generación de código la frase “The dog is black”.

Ya se mencionó un ejemplo de traducción con una frase en español a inglés; ahora es el turno de ejemplificar la aplicación de las fases del proceso de compilación con una instrucción escrita, en este caso, en el lenguaje de programación Java.

Suponiendo que se tiene `p = x + y * 60`, en donde todas la variables están declaradas de tipo `double` (Figura 1.7).

Figura 1.7

*Traducción a código en lenguaje de bajo nivel de la instrucción p = x + y * 60*



Generador de código intermedio

```
temp1 = int_to_double(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Optimizador de código

```
temp1 = id3 * 60.0
id1 = id2 + temp1
```

Generador de código

```
MOV  id3, Ax          50   200   01
MULT #60.0, Ax        62   160   01
ADD  id2, Ax          60   210   01
MOV  Ax, id1          50   01    190
STOP                  70
```

Instrucción	Código	Longitud
MOV	50	3
MULT	62	3
ADD	60	3
REST	61	3
DIV	63	3
STOP	70	1

1.4. Actividades

Existen conceptos contextuales a la compilación, así como programas que tienen funciones que sirven al proceso de la compilación como, por ejemplo: preprocesadores, cargadores, enlazadores o ligadores, traductores, intérpretes, autocompiladores, metacompiladores, etc.

Investigue la definición de lo siguiente:

Traductor:

Traductores de lenguaje natural:

Intérprete:

Diferencia entre un compilador y un intérprete:

Ensamblador:

Diferencia entre compilador y un ensamblador:

Preprocesador:

Conversores Fuente-Fuente:

Proceso compilación – montaje – ligado – ejecución:

Compilador cruzado:

Compilación incremental:

Autocompilador:

Metacompilador:

Decompilador:

Bootstrapping:

Capítulo 2. Análisis Léxico

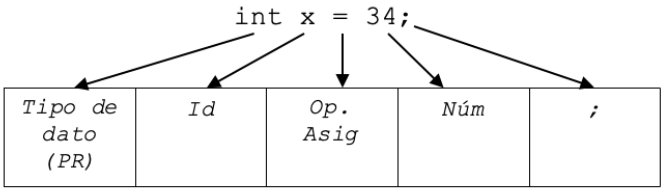
Diego Ulises Carranza Sahagún
Ma. del Carmen Nolasco Salcedo

2.1. Análisis Léxico

La principal tarea de un analizador léxico es leer los caracteres de la entrada del programa fuente, agruparlos en lexemas y producir como salida una secuencia de tokens para cada lexema en el programa fuente (Aho, Lam, Sethi, & Ullman, 2008).

El analizador léxico recibe como entrada el programa fuente, y da como resultado un listado de componentes léxicos (tokens). Cada token es una secuencia de caracteres que representa una unidad de información en el programa fuente.

Por ejemplo:



En este ejemplo el analizador léxico da como resultado un listado de cinco componentes léxicos.

Cabe resaltar que, en esta fase, el nombre de las variables, su valor o su tipo de datos no son relevantes para el proceso; en esta fase sólo se revisa de manera lineal el léxico, los caracteres que se encuentran en el programa fuente.

Podría decirse que los tokens son como el tipo de componente léxico; las palabras o secuencias de caracteres que se encuentran en el programa fuente son los *lexemas* y mediante *patrones,* que son las descripciones que pueden tener los lexemas, se pueden determinar los componentes léxicos que forman las proposiciones en el programa fuente.

Los tokens suelen estar acompañados de atributos que se pueden utilizar para diferenciar los lexemas que pueden coincidir con el mismo patrón, por ejemplo, `x` e `y` coinciden

con el token *id*; pero es necesario más información de estos lexemas que se utilizará en las fases siguientes de la compilación y en las tablas de símbolos.

Entre los componentes léxicos más comunes en un lenguaje de programación encontramos (Tabla 2.1):

- Palabras Reservadas: propias del lenguaje donde el patrón es la misma palabra, se podrían incluir los nombres de los tipos de datos.
- Identificadores: el nombre que se le da a las cosas en un programa (variables, constantes, métodos, funciones, etc.).
- Constantes numéricas: los números representados en sus distintas notaciones (enteros, con parte decimal o con exponentes)
- Constantes literales: las cadenas literales, regularmente delimitadas por comillas "".
- Operadores: símbolos para los distintos tipos de operaciones (asignación, aritméticos, relacionales, lógicos, etc.).
- Símbolos de puntuación: ',', '.', ';', ':', etc.
- Agrupación: (,), [,], {, }

Tabla 2.1
Ejemplos de tokens, patrones y lexemas

Token	Patrón (informal)	Lexemas de ejemplo
PR while	caracteres w,h,i,l,e	while
id	Letra,'_' o '$' seguido de letras, dígitos, '_' o '$'	var_1, PI, esNumero
Op. relacional	**<, >, ≤, ≥, = o ≠**	<, >, <=, >=, ==, !=
núm	constante numérica	6, 1.23, 1e5, 1.2e-4

Como se mencionó previamente, la función principal del analizador léxico es generar un listado de tokens encontrados; pero también hay otras funciones que puede se pueden realizar en esta fase de análisis como son: eliminar los espacios en blanco del programa fuente (se incluyen espaciados, tabulaciones, cambios de línea, retorno de carro, etc.) y eliminar comentarios, los cuales son importantes para la programación; sin embargo, no son instrucciones, por lo que deben eliminarse para hacer más sencillo el proceso de traducción. Evidentemente el analizador léxico interactúa con el administrador de errores, enviándole los

errores encontrados, que en esta fase, están relacionados, en su mayoría, con caracteres o símbolos que no son parte del alfabeto del lenguaje de programación del programa fuente.

2.2. Expresiones Regulares

Una de las formas en la que se pueden determinar los componentes léxicos es mediante el uso de expresiones regulares.

Las *expresiones regulares* representan patrones de cadenas de caracteres. Una expresión regular *r* se encuentra completamente definida mediante el conjunto de cadenas con las que concuerda. Este conjunto se denomina lenguaje generado por la expresión regular y se escribe como *L(r)* (Louden, 2004). Las expresiones regulares nos sirven para validar cadenas de caracteres y asegurar que los datos tengan un formato específico.

El conjunto de símbolos válidos se conoce como **alfabeto** y por lo general se representa mediante el símbolo griego Σ (sigma). También, se necesita especificar una concordancia con la **cadena vacía**, es decir, la cadena que no contiene ningún carácter; para lo cual se utiliza, regularmente, el símbolo ε (épsilon) para denotar la cadena vacía y se establece que L(ε) = {ε}. (El lenguaje de la cadena vacía es la propia cadena vacía). Nótese que la cadena vacía es distinto al conjunto vacío (Φ) que corresponde a la ausencia de cadenas, L(Φ) = { }. Observe la diferencia entre {ε} y { }: el conjunto {ε} contiene la cadena que no se compone de ningún carácter, mientras que el conjunto { } no contiene ninguna cadena.

Las reglas que definen las expresiones regulares sobre cierto alfabeto Σ, y los lenguajes que denotan dichas expresiones son (Aho, Lam, Sethi, & Ullman, 2008):

Reglas base:

1.- ε(cadena vacía) es una expresión regular designada por ***L(ε)***.

2.- Si a es un símbolo en Σ, entonces ***a*** es una expresión regular, y ***L(a)***={a}

Operaciones de expresiones regulares

Existen tres operaciones básicas en las expresiones regulares: 1) selección entre alternativas o unión, la cual se indica mediante el metacarácter | (barra vertical): 2) concatenación, que se indica mediante yuxtaposición (sin un metacarácter), y 3) repetición o “cerradura”, la cual se indica mediante el metacarácter * (asterisco) (Louden, 2004).

Suponiendo que *r* y *s* sean expresiones regulares representadas por lenguaje *L(r)* y *L(s).*

Selección entre alternativas (unión). (*r*)|(*s*) es una expresión regular designada por *L(r)*∪*L(s)*. Por ejemplo: **a|b** que corresponda a los caracteres *a* y *b*, *L*(a|b) = *L*(a)∪*L*(b) = {*a*}∪{*b*} = {*a*, *b*}.

Concatenación. (r)(s) es una expresión regular designada por *L*(r)*L*(s). Por ejemplo: **ab** corresponde sólo a la cadena *ab*.

Repetición o cerradura de Kleene. (*r*)* es una expresión regular designada por *(L(r))**. Por ejemplo: **a*** corresponde a las cadenas { ε, *a*, *aa*, *aaa*, *aaaa*, …

Precedencia de operadores y uso de paréntesis. La cerradura de Kleene * tiene la precedencia más alta, seguida de la concatenación y posteriormente la unión | tiene la prioridad más baja. Evidentemente si se quiere utilizar una precedencia diferente se utilizan los paréntesis (Tabla 2.2).

Ejemplos:

L = (A|B|C|…|Z|a|b|c|…|z)

D = (0|1|2|3|4|5|6|7|8|9)

L | D = {A, B, C, …, Z, a, b, …, z, 0, 1, …, 9}

LD = {A0, A1, … A9, B0, …, Z9, a0, a1, …, z9}

L^4 = LLLL = {AAAA, ABCD, … } L {4}

L* = {ε, A, Z, AA, AAA, abcde, …}

L^+ = {A, Z, AA, AAA, abcde, …}

(L | _ | $)(L | D | _ | $)* = {Aε, AA, A1, Q12345678, qwerty, variable, identificador, …

Tabla 2.2
Leyes algebraicas comunes aplicadas a expresiones regulares

r | s = s | r	| es conmutativo
r | (s | t) = (r | s) | t	| es asociativo
(rs)t = r(st)	Concatenación es asociativa
r(s | t) = rs | rt	Concatenación distribuye sobre |
(s | t)r = sr | tr	
εr = r	ε es el elemento identidad para la concatenación
rε = r	
r* = (r| ε)*	Relación entre * y ε
r** = r*	∗ es idempotente

Ejemplo 2.1: Los identificadores en Java son cadenas de letras, dígitos, guiones bajos y símbolos de dólar. Se muestra a continuación una definición regular para el lenguaje de los identificadores de Java.

```
letras_$          =   A | B | … | Z | a | b | … | z | _ | $
dígito            =   0 | 1 | … | 9
id                =   letras_$ ( letras_$ | dígito )*
id = (A|B| … |Z|a|b| … |z|_|$)(A|B| … |Z|a|b| … |z|_|$|0|1| … |9)*
```

Ejemplo 2.2: Los números sin signo (enteros o de punto flotante) son cadenas con la forma 123, 4.567, 8.91E4 o 2.34E-4. La siguiente es una representación para este conjunto de cadenas.

```
dígito            =   0 | 1 | … | 9
dígitos           =   dígito dígito*
frac_opc          =   .dígitos | ε
exp_opc           =   ((E|e)(+|-|ε)dígitos)|ε
número            =   dígitos frac_opc exp_opc
```

Extensiones de las expresiones regulares

Desde la introducción de las expresiones regulares se han agregado extensiones para simplificar y facilitar la especificación de patrones de cadenas, entre las cuales se encuentran algunas útiles para su uso en analizadores léxicos:

Una o más instancias. El operador unario postfijo + representa la cerradura positiva de una expresión regular. Es decir, si *r* es una expresión regular, entonces (*r*)+ denota el lenguaje (*L*(*r*))+. El operador + tiene la misma precedencia y asociatividad que el operador *. El conjunto resultante no incluye la cadena vacía. *r*+ = *rr** = *r***r*

Cero o una instancia. El operador unario postfijo *?* significa “cero o una ocurrencia”. Es decir, *r?* es equivalente a *r* |ε . El operador *?* tiene la misma precedencia y asociatividad que * y + .

Clases de caracteres. Una expresión del tipo *a* | *b* | *c* puede sustituirse por la abreviación [*abc*]. También se emplea esta notación para escribir un intervalo de caracteres, por ejemplo: *a* | *b* | *c ... z* se puede escribir como [*a-z*]. Nótese que cada par de corchetes [ ] representa un solo carácter y que no hay espacios en blanco entre los símbolos, dado que estos, también son caracteres (Tabla 2.3).

Tabla 2.3

Clases de caracteres

`[abc]`	a, b, o c (clase simple)
`[^abc]`	Cualquier carácter excepto a, b, c (negación)
`[a-zA-Z]`	De a hasta z, or A hasta Z, inclusivo (rango)
`[a-d[m-p]]`	De a hasta d, o de m hasta p: `[a-dm-p]` (unión)
`[a-z&&[def]]`	d, e, o f (intersección)
`[a-z&&[^bc]]`	De a hasta z, excepto para b y c: `[ad-z]` (sustracción)
`[a-z&&[^m-p]]`	De a hasta z, y no de m hasta p: `[a-lq-z]`(sustracción)

Mediante el uso de las abreviaciones anteriores, se puede escribir la definición regular del ejemplo 2.1 como:

```
letras_$    =    [A-Za-z_$]
  dígito    =    [0-9]
      id    =    letras_$ ( letras_$ | dígito )*
```

La definición regular del ejemplo 2.2 también puede simplificarse con el uso de las abreviaciones, quedando como:

```
 dígito    =     [0-9]
dígitos    =     dígito+
número     =     dígitos (.dígitos)? ([Ee][+-]? dígitos)?
```

Ejemplo 2.3: Las expresiones regulares no son exclusivas de los lenguajes de programación o del proceso de compilación; por definición, son utilizadas en la validación de cadenas de caracteres, determinando si cumplen con un patrón establecido.

Las expresiones regulares se pueden utilizar para validar direcciones IP que se componen de cuatro octetos, es decir, tienen la forma: ###.###.###.### , en donde cada ### representa a un octeto (8 dígitos binarios), que en su valor decimal puede ser un número de 0 hasta 255 (ej. 148.202.148.1).

Para obtener la expresión regular de un octeto, hay que verificar ¿cuáles son los posibles caracteres que pueden aparecer en cada posición? Para valores de 000 a 255, se tienen dos casos: para los valores de 000 a 199, el primer carácter puede ser 0 o 1, el segundo carácter puede ser cualquier dígito (0-9) y el tercer carácter también cualquier dígito; sin embargo, para el caso cuando el primer carácter es 2, el segundo carácter puede ser de 0 a 5 solamente y aquí se tienen otras dos situaciones: si el segundo carácter es de 0 a 4, entonces el tercer carácter puede ser cualquier dígito de 0 a 9 y el caso cuando el segundo carácter es 5, entonces el tercer carácter puede ser un carácter entre 0 y 5.

Por lo que queda una expresión para un octeto de la siguiente forma:

```
"([01][0-9][0-9] | 2([0-4][0-9]|5[0-5])).
([01][0-9][0-9] | 2([0-4][0-9]|5[0-5])).
([01][0-9][0-9] | 2([0-4][0-9]|5[0-5])).

([01][0-9][0-9] | 2([0-4][0-9]|5[0-5]))"
```

Cabe señalar, que en este ejemplo se puso el carácter '.' como el símbolo . (punto) del albabeto literal, que en algunos lenguajes de programación que implementan expresiones regulares, el punto (.) es cualquier carácter y habría que especificar con la notación del lenguaje que se hace referencia precisamente al símbolo punto (.). También, se hace notar que la expresión propuesta siempre maneja octetos de tres caracteres y las direcciones IP, pueden tener desde uno hasta tres caracteres para representar el octeto en numeración decimal.

2.3. Expresiones regulares en Java

El lenguaje de programación Java incluye una buena cantidad de clases que nos permiten implementar analizadores léxicos. Existen clases en Java como String, StringBuilder, Character, StringTokenizer que nos permiten trabajar con cadenas de caracteres. Se tienen una gran cantidad de métodos con los que podemos, por ejemplo, buscar subcadenas dentro de otras cadenas, verificar si coinciden con cierto formato, recortar o copiar subcadenas, separar por componentes alguna cadena, etc. En Java, también se incluyen clases para manejar patrones que definen expresiones regulares para la validación de cadenas de caracteres como son la clase Pattern y la clase Matcher, de las que hablaremos más adelante.

Una de las formas más simples de validar si una cadena de caracteres cumple con una expresión regular es utilizar el método *matches* de la clase *String,* el cual, recibe una cadena que especifica la expresión regular, e iguala el contenido del objeto String que lo llama con la expresión regular. Este método devuelve un valor de tipo boolean indicando si hubo concordancia o no.

Por ejemplo: La expresión regular `"[A-Z][a-zA-Z]*",` que valida el primer nombre o nombre de pila, con el formato de la primer letra en mayúscula y el resto puede estar en mayúsculas o minúsculas, se puede validar mediante el método *matches* de la clase *String*. Este método nos regresa un valor booleano, verdadero o falso, si coincide o no coincide la cadena con el patrón proporcionado como parámetro al método: suponiendo que se tiene un objeto de la clase *String* llamado *primerNombre* (`String primerNombre;`), si `primerNombre = "Homero";` entonces el método `primerNombre.matches("[A-Z][a-zA-Z]*");` dará como resultado verdadero (true) y si tuviéramos por ejemplo la cadena "Nombre01" nos dará false (false) (Tabla 2.4).

Tabla 2.4

Clases predefinidas de caracteres en Java

.	Cualquier carácter (puede o no coincidir con los delimitadores de línea)
\d	Un dígito: `[0-9]`
\D	Cualquier carácter que no sea dígito: `[^0-9]`
\s	Cualquier espacio en blanco: `[ \t\n\x0B\f\r]`
\S	Cualquier carácter que no sea de espacio en blanco: `[^\s]`
\w	Cualquier carácter de palabra: `[a-zA-Z_0-9]`
\W	Cualquier carácter que no sea de palabra: `[^\w]`

Una expresión regular consiste de caracteres literales y símbolos especiales. La Tabla 2.4 especifica algunas clases predefinidas de caracteres que pueden usarse con las expresiones regulares. Una clase de carácter es una secuencia de escape que representa a un grupo de caracteres. Un dígito es cualquier carácter numérico. Un carácter de espacio en blanco es un espacio, tabulador, retorno de carro, nueva línea o avance de página. Un carácter de palabra es cualquier letra (mayúscula o minúscula), cualquier dígito o el carácter de guión bajo. Cada clase de carácter se iguala con un solo carácter en la cadena que intentamos hacer concordar con la expresión regular.

Java también utiliza las clases de caracteres mostradas en la Tabla 2.3. Para hacer que concuerde un conjunto de caracteres que no tiene una clase predefinida de carácter, puede utilizar los corchetes [ ]. Por ejemplo, el patrón "[*aeiou*]" puede usarse para concordar con una sola vocal. Los rangos de caracteres pueden representarse colocando un guión corto (-) entre dos caracteres. En otro ejemplo, "[*A-Z*]" concuerda con una sola letra mayúscula. Si el primer carácter entre corchetes es "^", la expresión acepta cualquier carácter distinto a los que se indiquen. Sin embargo, es importante observar que "[^*Z*]" no es lo mismo que "[*A-Y*]", la cual concuerda con las letras mayúsculas *A-Y*; "[^*Z*]" concuerda con cualquier carácter distinto de la letra *Z* mayúscula, incluyendo las letras minúsculas y los caracteres que no son letras, como el carácter de nueva línea. Los rangos de las clases de caracteres se determinan mediante los valores enteros de las letras (verifique el código ASCII para referencia), por ejemplo, "[*A-Za-z*]" concuerda con todas las letras mayúsculas y minúsculas. Las clases de caracteres delimitadas entre corchetes concuerdan con un solo carácter en el objeto de búsqueda y observar que en "[*A-Za-z*]" no hay espacios en blanco entre los caracteres y los corchetes, ni entre los guiones cortos y los caracteres, ya que los espacios también son caracteres (Tabla 2.5).

Tabla 2.5

Cuantificadores usados en expresiones regulares

Cuantificador	Concuerda con
`X?`	`X`, una o ninguna vez
`X*`	`X`, cero o más veces
`X+`	`X`, una o más veces
`X{n}`	`X`, exactamente `n` veces
`X{n,}`	`X`, al menos `n` veces
`X{n,m}`	`X`, al menos `n` pero no más de `m` veces

En la Tabla 2.5 se muestran los cuantificadores usados en expresiones regulares en Java. El cuantificador asterisco (*) y el cuantificador signo de suma (+), son los operadores de a cerradura de Kleene y la cerradura positiva que ya se había mencionado su función en la sección de las operaciones de expresiones regulares. Todos los cuantificadores afectan solamente a la subexpresión que va inmediatamente antes del cuantificador, es decir, son asociativos por la izquierda. El cuantificador signo de interrogación (?) concuerda con cero

o una ocurrencia de la expresión que cuantifica. Un conjunto de llaves que contienen un número ({*n*}) concuerda exactamente con *n* ocurrencias de la expresión que cuantifica. Si se incluye una coma después del número encerrado entre llaves, el cuantificador concordará al menos con *n* ocurrencias de la expresión cuantificada. El conjunto de llaves que contienen dos números ({*n,m*}) concuerda entre *n* y *m* ocurrencias de la expresión que califica. Los cuantificadores pueden aplicarse a patrones encerrados entre paréntesis para crear expresiones regulares más complejas.

Clases Pattern y Matcher

Java proporciona otras clases en el paquete *java.util.regex* que ayudan a los desarrolladores a manipular expresiones regulares. La clase **`Pattern`** representa una expresión regular. La clase **`Matcher`** contiene tanto un patrón de expresión regular como un objeto CharSequence en el que se va a buscar ese patrón.

La clase `Pattern` tiene el método `static` **`matches`** que se usa cuando se va a utilizar una expresión regular una sola vez. Este método toma una cadena que especifica la expresión regular y un objeto CharSequence en la que se va a realizar la prueba de concordancia. Este método devuelve un valor de tipo `boolean`, el cual indica si el objeto de búsqueda (el segundo argumento) concuerda con la expresión regular.

Si una expresión regular se va a utilizar más de una vez, es recomendable utilizar por eficiencia el método `static` **`compile`** de la clase `Pattern` para crear un objeto `Pattern` específico de esa expresión regular. Este método recibe una cadena que representa el patrón y devuelve un nuevo objeto `Pattern`, el cual puede utilizarse para llamar al método **`matcher`**.

La clase `Matcher` cuenta con el método **`matches`**, el cual realiza la misma tarea que el método **`matches`** de `Pattern`, es decir, devuelve un valor de tipo `boolean` que indica si el objeto de búsqueda concuerda con la expresión regular; pero no recibe argumentos; el patrón y el objeto de búsqueda están encapsulados en el objeto `Matcher`. La clase `Matcher` proporciona otros métodos, incluyendo `find`, `lookingAt`, `replaceFirst` y `replaceAll`.

En la Figura 2.1 se tienen un ejemplo del uso de las clases **`Pattern`** y **`Matcher`** para la verificación de cadenas. Se observa en la línea 11 el uso del método `compile` para ingresar una expresión regular al objeto *`patron`* de la clase `Pattern`, en la línea 18 se genera un

objeto de la clase `Matcher` resultado del método `patron.matcher(s)` del objeto *`patron`* que compara la cadena de caracteres `s`, que fue ingresada en la línea 16 del código. Este objeto *`matcher`* encapsula, como ya se comentó previamente, el patrón y el objeto de búsqueda (la cadena de caracteres a comparar). La línea 21 del código muestra en la salida el resultado del método `matches()` del objeto *`matcher`*, que regresa verdadero (*true*) en caso de que la cadena coincida con el patrón o falso (*false*) en caso de que no. La línea 24 muestra en la salida el resultado del método `pattern()` del objeto *`matcher`* la expresión regular con la que se está haciendo la validación de la cadena de caracteres, en este ejemplo solo se muestra con la intención de monitorizar la expresión.

Figura 2.1
Ejemplo en Java del uso de las clases Pattern y Matcher

```
 1 package regexp;
 2 import java.util.Scanner;
 3 import java.util.regex.Pattern;
 4 import java.util.regex.Matcher;
 5
 6 public class RegExp {
 7
 8     public static void main (String[] args) {
 9         Scanner in = new Scanner(System.in);
10
11         Pattern patron = Pattern.compile("[A-Za-z_$][A-Za-z_$0-9]*");
12
13         /*Lectura de la cadena a comparar con el patrón*/
14         String s;  //cadena a evaluar
15         System.out.print("Ingresa la cadena a evaluar: ");
16         s=in.nextLine();
17
18         Matcher matcher = patron.matcher(s);
19
20
21         System.out.println("El valor de matcher "+ matcher.matches());
22
23         System.out.println("--------------------");
24         System.out.println("Patron   "+matcher.pattern());
25
26         System.out.println("--------------------");
27         System.out.println("s.matches(\"[A-Za-z_$][A-Za-z_$0-9]*\")= "
28                            +s.matches("[A-Za-z_$][A-Za-z_$0-9]*"));
29         System.out.println();
30
31
32     }
33
34 }
```

Finalmente, en el código en la línea 27 se muestra el uso del método `matches` de la clase **`String`**, el cual funciona de manera similar al método `matches` de la clase **`Matcher`**, con la diferencia de que cada vez que se requiera hacer una comparación, se tienen que crear los objetos **`String`** e ingresar la expresión regular; y con el método de la clase **`Matcher`** se ingresa la expresión una sola vez y se puede utilizar las veces que sea necesario.

En este ejemplo de uso de las clases Pattern y Matcher se utilizó la expresión regular `"[A-Za-z_$][A-Za-z_$0-9]*"` que valida si una cadena de caracteres es un identificador en el lenguaje Java.

En la Figura 2.2 se muestra una salida de la ejecución del código de la Figura 2.1.

Figura 2.2
Ejemplo de ejecución de la clase RegExp de la figura 2.1

```
run:
Ingresa la cadena a evaluar: id
El valor de matcher true
--------------------
Patron   [A-Za-z_$][A-Za-z_$0-9]*
--------------------
s.matches("[A-Za-z_$][A-Za-z_$0-9]*")= true
```

2.4. Ejercicios de expresiones regulares

1. Elaborar una expresión regular que valide un octeto (8 dígitos binarios) que se expresa en decimal y que se usan en las direccione IP. Tómese en cuenta el ejemplo 2.3; pero en ese ejemplo siempre los octetos son de 3 caracteres (000 a 255), la expresión propuesta debe validar cadenas de una, dos o tres cifras. (0-255)

2. Elabore una expresión regular que valide si una cadena de caracteres corresponde a una dirección de correo electrónico.

3. Elabore una expresión regular que valide si una cadena de caracteres es una dirección de un sitio web con el formato www.cualquierletra.com.mx

4. Elabore una expresión regular que valide si una cadena tiene el formato de un número telefónico.

5. Elabore una expresión regular que determine si un apellido de una persona apellido tiene el formato adecuado a que sólo debe contener letras, espacios, apóstrofes y guiones cortos.

6. Escriba una expresión regular que valide un número par de letras x. (xx, xxxx, xxxxxx, …)

7. Escriba una expresión regular que valide un número impar de letras y. (y, yyy, yyyyy, …)

8. Escriba definiciones regulares para el lenguaje de todas las cadenas de letras en minúsculas que contengan las cinco vocales en orden.

9. Escriba una expresión regular para el lenguaje de todas las cadenas de dígitos que representen números pares.

10. Elabore una expresión regular que valide una cadena de caracteres que represente una fecha con el formato AAMMDD (año, mes día). Para el mes 02 valide para 29 días.

Describa los lenguajes que denotan las siguientes expresiones regulares:

1. [+-]?[0-9]+(\\.[0-9]+)?([Ee][+-]?[0-9]+)?

2. [A-Za-z0-9_]+@[A-Za-z0-9_]+\\.[A-Za-z]{3}(\\.[A-Za-z]{2})?

3. [0-9A-Za-z]\\d{8}

4. a*ba*ba*ba*

5. (\\w)+@(\\w)+\\.[a-zA-Z]{3}(\\.[A-Za-z]{2})?

6. (xx)*y(yy)*

7. [1-9]\\d{2} - [1-9]\\d{2} - \\d{4}

8. (0 | 1 | ... | 9 | A | B | C | D | E | F) +(x | X)

9. (y+ | x+y+ | x+y+x+y+)

10. [0-9]*[13579]

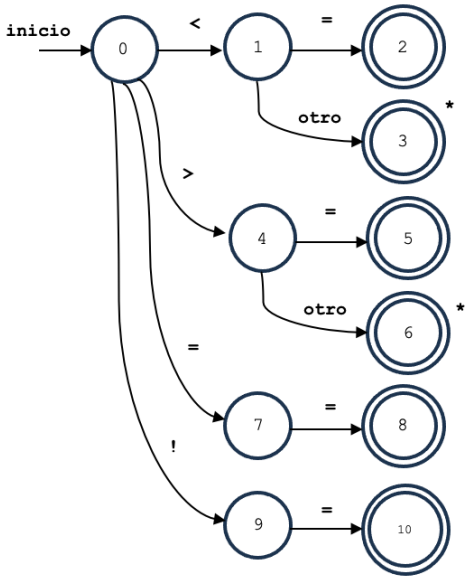
2.5 Diagramas de transición de estados

Otra de las formas de elaborar analizadores léxicos, es mediante la implementación de diagramas de transición de estados.

Los *diagramas de transición de estado* sirven como una representación visual de cómo un analizador puede transitar entre diferentes estados mientras procesa una cadena de entrada.

Los diagramas de transición de estados tienen una colección de nodos, llamados *estados* representados círculos. Cada estado representa una condición que podría ocurrir durante la lectura de la entrada, buscando su coincidencia con los patrones, es decir, representan los diferentes puntos en los que se puede encontrar el analizador durante el proceso. Los estados se conectan con *flechas* que indican la transición entre estados a través de la coincidencia con la *etiqueta* de la flecha, es decir, representan los cambios de estado que ocurren cuando se lee un determinado símbolo de entrada. Un estado se designa como *estado inicial* y se distingue por una flecha que llega a este estado pero no proviene de ningún otro estado. Ciertos estados se consideran *finales* o estados *de aceptación* e indican que se encontró una coincidencia con el patrón que representa el diagrama; estos estados de aceptación se distinguen por un círculo doble (Figura 2.3).

Figura 2.3
Diagrama de transición de estados para operadores relacionales



En la Figura 2.3 se presenta un diagrama de transición de estados que reconoce lexemas que coinciden con los operadores relacionales. Se empieza en el estado inicial 0, se obtiene el siguiente carácter a evaluar. Si el carácter es < se hace la transición al estado 1, en este estado se obtiene el siguiente carácter en la entrada; si el carácter es = se hace la transición al estado 2 que es un estado de aceptación, lo que significa que se tiene en la entrada el operador relacional <= (menor o igual que).

Estando en el estado 1 y como entrada se tiene otro carácter que no sea =, se hace la transición al estado de aceptación 3, lo que significa que coincide con el operador < (menor que). Sin embargo, se observa que el estado 3 tiene un * (asterisco) que indica que se debe retroceder una posición en la entrada.

En general, el comportamiento del diagrama es similar a lo mencionado, cada vez que se llega a un estado, se obtiene el siguiente carácter de la entrada y se verifica si hay una transición a otro estado con la coincidencia de ese símbolo. Posteriormente, se verifica si se está en un estado de aceptación para dar por válido el operador. En muchas ocasiones los estados de aceptación van acompañados de funciones de retorno, que devuelven el token encontrado y pueden regresar también valores de atributos que tengan asociados.

En la Figura 2.6 se puede observar un bosquejo de la implementación de una función `obtenerOpRel()` en el que se simula el diagrama de transiciones de la Figura 2.3. En esta función, que retorna el operador relacional con el que concuerda una secuencia de entrada, evidentemente, falta especificar los casos de todos los estados del diagrama, once en total; pero sí se especifica el funcionamiento en el estado inicial, `case 0`:, y el `case 3`: que es un estado de aceptación que implica que se tiene que hacer un retroceso en la secuencia de caracteres de la entrada y retorna el tipo de operador relacional concordante, en este caso el `menor que MQ`. Los demás casos trabajan de forma similar a los casos especificados.

Figura 2.6

Bosquejo de la implementación del diagrama para operadores relacionales de la figura 2.3

```
FUNCIÓN obtenerOpRel()
{
    while(1){ /* repite el procesamiento hasta que
                 ocurre un retorno o un fallo */
        switch(estado){
            case 0: c = obtenerSigCar();
                    switch(c){
                        case '<': estado = 1;
                                  break;
                        case '>': estado = 4;
                                  break;
                        case '=': estado = 7;
                                  break;
                        case '!': estado = 9;
                                  break;
                        default fallo();
                    }
            case 1: ...
            ...
            case 3: retroceder();
                    retornar(MQ) /*Menor que*/
        }
    }
}
```

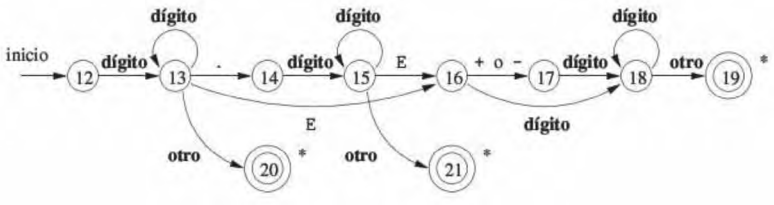
Las expresiones regulares son una forma concisa de describir patrones de texto. Los diagramas de transición de estados, por su parte, ofrecen una representación gráfica de esos mismos patrones. Existen herramientas y algoritmos para hacer conversiones entre diagramas de transición de estados y expresiones regulares. Sin embargo, también se puede hacer la conversión de forma manual.

En la Figura 2.5 se muestra un diagrama de transición de estados para números sin signo, que aparece en el libro Compiladores de Aho. Este diagrama corresponde a la expresión regular:

```
dígito   =    [0-9]
dígitos  =    dígito+
número   =    dígitos (.dígitos)? ([Ee][+-]? dígitos)?
```

Figura 2.5

Diagrama de transición de estados para números



2.6. Ejercicios de diagramas de transición de estados

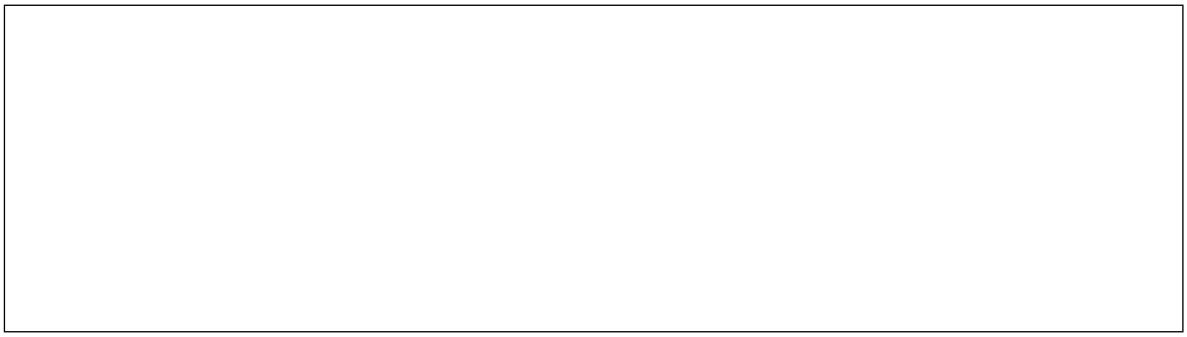
1. Dibuje un diagrama de transiciones de estado que valide si una cadena de caracteres corresponde a una dirección de correo electrónico.

2. Elabore un diagrama de transición de estados que valide un número par de letras x. (xx, xxxx, xxxxxx, …)

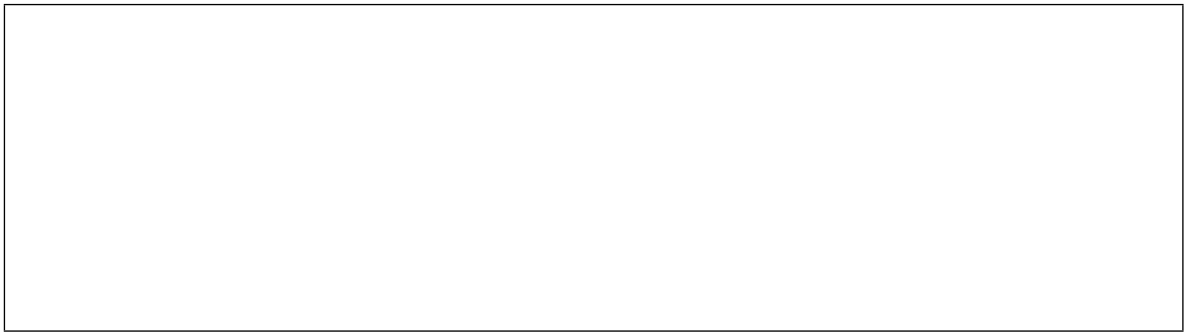
3. Elabore un diagrama de transición de estados que valide un número impar de letras y. (y, yyy, yyyyy, …)

4. Elabore un diagrama de transición de estados para el lenguaje de todas las cadenas de letras en minúsculas que contengan las cinco vocales en orden.

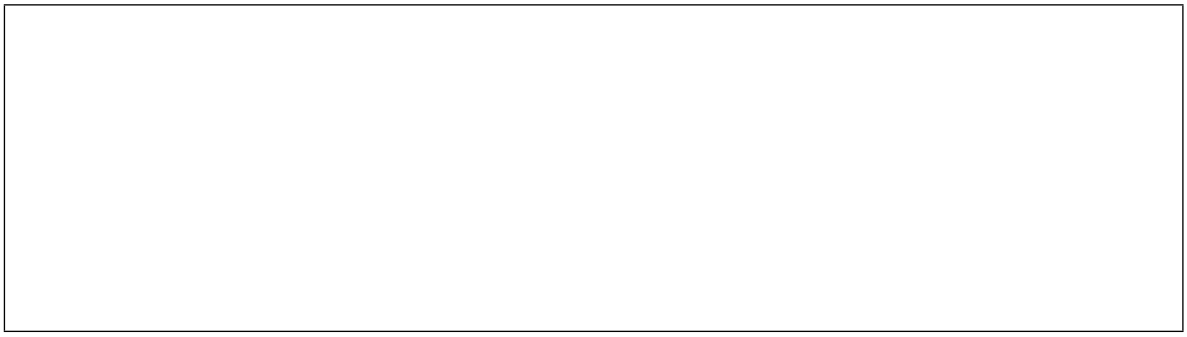
5. Elabore un diagrama de transición de estados para el lenguaje de todas las cadenas de dígitos que representen números pares.



6. Elabore un diagrama de transición de estados que valide una cadena de caracteres que represente una fecha con el formato AAMMDD (año, mes día). Para el mes 02 valide para 29 días.



7. Diseñe un diagrama de transiciones para un autómata finito que acepte el lenguaje consistente en las cadenas del alfabeto {w, x, y, z} en donde el patrón xy siempre es seguido por una w y donde al patrón yx siempre siga una z. (p.e. xyw, yxz, xywxyw, yxzyxz, xywyxz, yxzxyw, xxxxyw, yyyxz, xxxywyyyxz, yyxzxxxyw, etc.).

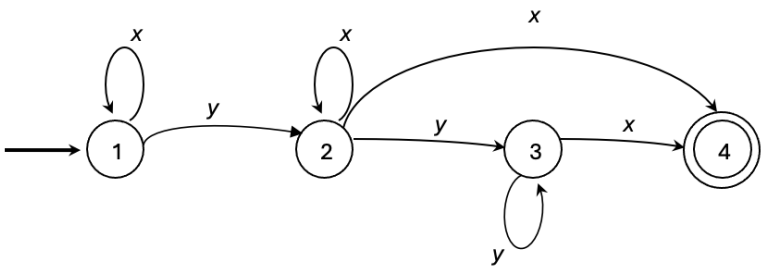


8. Elabore un diagrama de transiciones que acepte el mismo lenguaje que la siguiente expresión regular:

(*y*+ | *x*+*y*+ | *x*+*y*+*x*+*y*+)

9. Elabore un diagrama de transiciones que valide cadenas de caracteres que representen números hexadecimales.

10. Determine el lenguaje que valida el siguiente diagrama:



2.7. Actividades

1. Defina expresiones regulares para los principales componentes de un lenguaje de programación: Palabras reservadas, identificadores, números, operadores aritméticos (+ , - , * , /), operadores relacionales (< , <=, >, >=, ==, !=), operadores lógicos (&&, ||), signos de puntuación (‘.’, ‘,’, ‘;’, ‘:’), símbolos de agrupación ((,), [,], {, }).

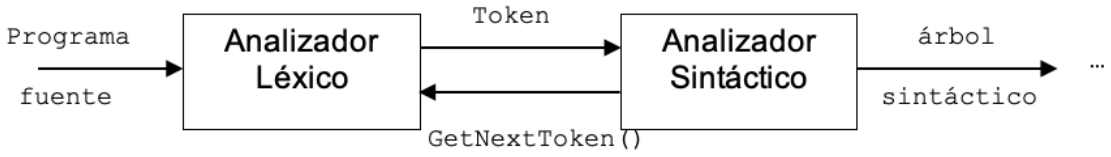
Capítulo 3. Análisis Sintáctico

Diego Ulises Carranza Sahagún
Óscar Antonio Zárate Águila

3.1. Análisis Sintáctico

La principal tarea de un analizador sintáctico es indicar si la secuencia de componentes léxicos, encontrados en el análisis léxico, están en el orden correspondiente a las reglas gramaticales del lenguaje (Figura 3.1).

Figura 3.1
Esquema de la relación entre analizadores léxico y sintáctico



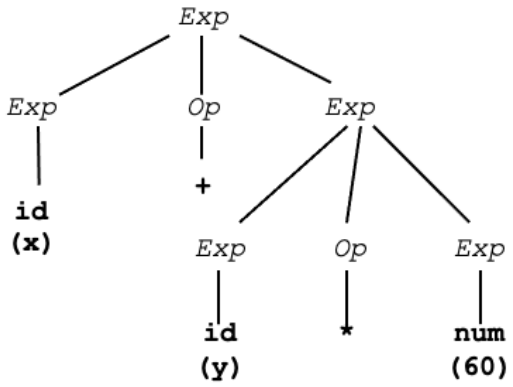
El analizador sintáctico recibe como entrada los componentes léxicos, puede ser que desde un inicio de la fase se pasen todos los componentes y que el analizador sintáctico ya no tenga más interacción con el análisis léxico; o puede ser que el analizador sintáctico solicite el siguiente componente cuando lo vaya necesitando mediante una función, que en la Figura 3.1 es la función `GetNextToken()`. Lo que se obtiene como resultado de esta fase de análisis sintáctico, regularmente es una estructura que se conoce como árbol sintáctico o árbol de análisis gramatical. Sin embargo, dependiendo de la forma como se implementa el analizador sintáctico podría no construirse en forma explícita.

$$
\begin{aligned}
&Exp \rightarrow Exp\ Op\ Exp \\
&Exp \rightarrow \mathbf{id} \\
&Exp \rightarrow \mathbf{num} \\
&Op \rightarrow + \mid - \mid * \mid /
\end{aligned}
\qquad (3.1)
$$

Por ejemplo, tomando en cuenta la gramática 3.1, la cual, es una gramática sencilla para expresiones aritméticas, y la expresión `x + y * 60`, se puede obtener como resultado el

árbol sintáctico de la Figura 3.2; lo cual nos indica que la expresión es válida sintácticamente, dado que se pudo generar un árbol sintáctico a partir de la secuencia de componentes léxicos siguiendo la gramática 3.1 (Figura 3.2).

Figura 3.2
Árbol sintáctico

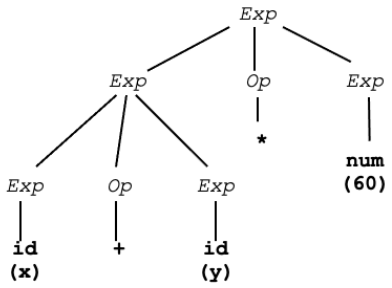


Sin embargo, también se puede generar como resultado, para la misma expresión y con la misma gramática, el árbol de la Figura 3.3. Ambos árboles son correctos, si se hace un recorrido de los árboles de las Figuras 3.2 y 3.3 se llega a la misma secuencia de componentes: `id + id * num`. Cabe resaltar, que en esta fase no se toman en cuenta las prioridades de ejecución de las operaciones aritméticas, el analizador sintáctico sólo verifica el orden de los componentes léxicos.

En este caso, la gramática utilizada tiene la característica de ser ambigua. Una **gramática** es **ambigua** cuando se pueden generar más de un árbol sintáctico para la misma secuencia de componentes con la misma gramática.

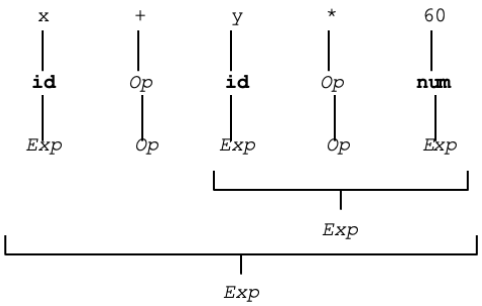
En general, hay dos tipos de métodos que se utilizan para los analizadores sintácticos: descendentes o ascendentes. En los analizadores sintácticos descendentes, para generar el árbol sintáctico, se parte de la raíz y se hacen las derivaciones hasta llegar a las hojas. Y en los analizadores sintácticos ascendentes, se parte de las hojas (terminales) y se van haciendo reducciones hasta llegar al nodo raíz. En el ejemplo que se generaron los árboles de las Figuras 3.2 y 3.3 se utilizó un algoritmo de forma descendente y en la Figura 3.4 se muestra una representación de un árbol sintáctico de la misma expresión; pero generado de forma ascendente

Figura 3.3
Árbol sintáctico (alternativo)



En general, hay dos tipos de métodos que se utilizan para los analizadores sintácticos: descendentes o ascendentes. En los analizadores sintácticos descendentes, para generar el árbol sintáctico, se parte de la raíz y se hacen las derivaciones hasta llegar a las hojas. Y en los analizadores sintácticos ascendentes, se parte de las hojas (terminales) y se van haciendo reducciones hasta llegar al nodo raíz. En el ejemplo que se generaron los árboles de las Figuras 3.2 y 3.3 se utilizó un algoritmo de forma descendente y en la Figura 3.4 se muestra una representación de un árbol sintáctico de la misma expresión; pero generado de forma ascendente.

Figura 3.4
Árbol sintáctico ascendente



3.2. Gramáticas libres de contexto

Una de las maneras de implementar analizadores léxicos es utilizando gramáticas libres de contexto. Una gramática libre de contexto es una especificación para la estructura sintáctica de un lenguaje de programación (Louden, 2004).

Aho (2008) menciona en su libro de compiladores algunos beneficios que tiene el uso de gramáticas, como son:

• Una gramática proporciona una especificación sintáctica precisa y fácil de entender, de un lenguaje de programación.

• A partir de ciertas clases de gramáticas, se puede construir de manera automática un analizador sintáctico eficiente que determine la estructura sintáctica de un programa fuente.

• La estructura impartida a un lenguaje mediante una gramática diseñada en forma apropiada es útil para traducir los programas fuente en código objeto correcto, y para detectar errores.

• Una gramática permite que un lenguaje evolucione o se desarrolle en forma iterativa, agregando nuevas construcciones para realizar nuevas tareas. Estas nuevas construcciones pueden integrarse con más facilidad en una implementación que siga la estructura gramatical del lenguaje.

Una gramática libre de contexto consta de cuatro componentes:

1. Un conjunto de símbolos terminales. Estos terminales, son los símbolos básicos del lenguaje definido por la gramática

2. Un conjunto de símbolos *no terminales*. Donde cada no terminal representa un conjunto de terminales. Como su nombre lo indica, no terminan ahí, sino que tienen más producciones con conjuntos de terminales.

3. Un conjunto de *producciones*, en donde cada producción consiste en un no terminal llamado *lado izquierdo* o *encabezado* de la producción, una flecha y una secuencia de terminales y no terminales llamada *cuerpo* o *lado derecho* de la producción.

4. Uno de los símbolos no terminales designado como *símbolo inicial*.

Por ejemplo, en la gramática 3.1 se tiene:

Terminales: { **id**, **num**, +, -, *, / }

No terminales: { *Exp, Op* }

Producciones: { $Exp \rightarrow Exp\ Op\ Exp$; $Exp \rightarrow$ **id**; $Exp \rightarrow$ **num**; $Op \rightarrow$ + | - | * | / }

Símbolo inicial: { Exp }

Las reglas gramaticales determinan las secuencias válidas de componentes por medio de derivaciones. Una *derivación* es una secuencia de reemplazos, por el cuerpo de una producción para ese no terminal.

Por ejemplo, en la Figura 3.5 se proporciona una derivación para la expresión: `x + y * 60`, utilizando la gramática 3.1. En cada paso se proporciona a la derecha la regla gramatical utilizada en el reemplazo.

Figura 3.5
*Una derivación para la expresión: x + y * 60*

1)	$Exp \Rightarrow Exp\ Op\ Exp$	$[Exp \rightarrow Exp\ Op\ Exp]$
2)	$\Rightarrow Exp\ Op\ Exp\ Op\ Exp$	$[Exp \rightarrow Exp\ Op\ Exp]$
3)	$\Rightarrow Exp\ Op\ Exp\ Op$ **num**	$[Exp \rightarrow$ **num**$]$
4)	$\Rightarrow Exp\ Op\ Exp$ * **num**	$[Op \rightarrow$ *$]$
5)	$\Rightarrow Exp\ Op$ **id** * **num**	$[Exp \rightarrow$ **id**$]$
6)	$\Rightarrow Exp$ + **id** * **num**	$[Op \rightarrow$ +$]$
7)	$\Rightarrow$ **id** + **id** * **num**	$[Exp \rightarrow$ **id**$]$

3.3. Árboles de análisis sintáctico

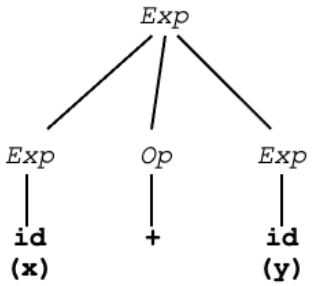
Un árbol de análisis sintáctico correspondiente a una derivación, es un árbol etiquetado en el cual los nodos interiores están etiquetados por no terminales, es decir, un árbol sintáctico muestra de forma gráfica la manera en que el símbolo inicial de una gramática deriva a una secuencia de componentes.

En un ejemplo, la derivación:

$$\begin{aligned} Exp &\Rightarrow Exp\ Op\ Exp \\ &\Rightarrow \textbf{id}\ Op\ Exp \\ &\Rightarrow \textbf{id} + Exp \\ &\Rightarrow \textbf{id} + \textbf{id} \end{aligned}$$

que corresponde a la expresión x + y genera el árbol de análisis sintáctico de la Figura 3.6.

Figura 3.6
Árbol sintáctico para la expresión: `x + y`



Las gramáticas tienen la capacidad de describir la mayoría de la sintaxis de los lenguajes de programación. Para la implementación de un analizador sintáctico descendente, como se mostrará más adelante, se pueden utilizar gramáticas con tres características:

- No deben ser gramáticas *ambiguas*. Una gramática es ambigua si para una secuencia de componentes se generan más de un árbol sintáctico. Existen, en la literatura de compiladores, algoritmos que permiten eliminar la ambigüedad de una gramática; pero esos temas están fuera del alcance de este libro.
- No deben tener *recursividad por la izquierda*. Si la gramática tiene producciones recursivas por la izquierda, el analizador sintáctico puede entrar en un ciclo infinito.
- Deben estar *factorizadas por la izquierda.* La factorización nos evita retrocesos en la validación de las secuencias y permite que la gramática sea adecuada para un analizador sintáctico predictivo.

Ambigüedad

Cuando se tiene una gramática que genera más de un árbol de análisis sintáctico para cierta secuencia de tokens, se dice que la gramática es ambigua. Por ejemplo, la gramática 3.1, previamente utilizada, y evaluando la expresión `x + y * 60`, que genera la secuencia de componentes léxicos `id + id * num`, se pueden obtener los árboles sintácticos de la Figura 3.7.

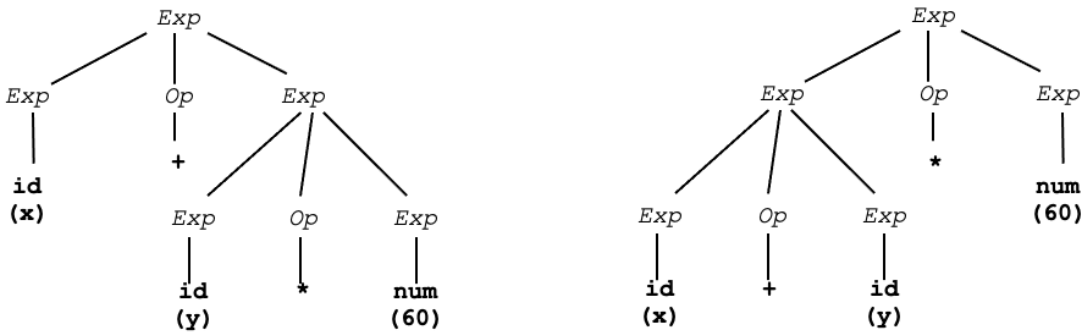
Exp → Exp Op Exp

Exp → id

Exp → num (3.1)

Op → + | - | * | /

Figura 3.7
Dos árboles sintácticos distintos para la misma expresión



Recursividad por la izquierda

Una gramática es *recursiva por la izquierda* si el primer símbolo del lado derecho de una producción es el mismo símbolo no terminal del lado izquierdo de la producción, es decir, una gramática es recursiva por la izquierda si se tiene una producción de la forma $A \rightarrow A\alpha$.

Los métodos de análisis sintácticos descendentes no permiten el uso de estas gramáticas recursivas, dado que se puede entrar a un ciclo infinito, por lo que es necesario hacer una transformación de la gramática para eliminar la recursividad por la izquierda.

La recursividad por la izquierda se puede eliminar reescribiendo la producción, esto es, si tiene un terminal A con dos producciones:

$$A \rightarrow A\alpha \mid \beta$$

donde α *y* β son secuencias de terminales y no terminales que no empiezan con A, se puede sustituir por las siguientes producciones que no son recursivas por la izquierda:

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

Por ejemplo, de la producción

$$Exp \rightarrow Exp + Term \mid Term$$

se tiene que el no terminal $A = Exp$, la cadena α = + *Term,* y la cadena β = *Term*, se obtiene:

$$Exp \rightarrow Term\ Exp'$$
$$Exp' \rightarrow + Term\ Exp' \mid \varepsilon$$

En la literatura de compiladores aparecen algoritmos para la implementación del proceso de eliminación de la recursividad por la izquierda, sin embargo, para este libro solamente ejemplificaremos cómo hacerlo.

Si se tiene la gramática 3.2 para validar expresiones aritméticas con términos y factores,

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned} \qquad (3.2)$$

se observa que la gramática es recursiva por la izquierda en dos producciones: *E* y *T*, y eliminando la recursividad por la izquierda se obtiene la gramática 3.3. Nótese que la produción $F \rightarrow (E) \mid \mathbf{id}$, no es recursiva por la izquierda por lo que no se modifica.

$$\begin{aligned} E &\rightarrow T\,E' \\ E' &\rightarrow + T\,E' \mid \varepsilon \\ T &\rightarrow F\,T' \\ T' &\rightarrow * F\,T' \mid \varepsilon \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned} \qquad (3.3)$$

Para verificar la equivalencia de la gramática 3.2 recursiva por la izquierda con la gramática 3.3 que no es recursiva por la izquierda, se generaron los árboles de análisis sintáctico, utilizando la secuencia de componentes `id + id`.

Figura 3.8
Árbol de análisis sintáctico utilizando gramática 3.2

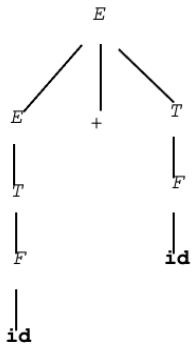
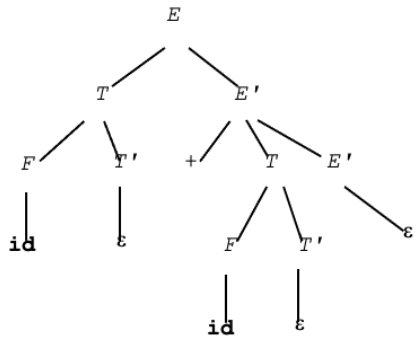


Figura 3.9
Árbol de análisis sintáctico utilizando gramática 3.3



Si se hace el recorrido en preorden del árbol de análisis sintáctico de la Figura 3.8, se obtiene la secuencia **`id`** + **`id`**, que es la que se está evaluando. Y si se hace el recorrido del árbol de análisis sintáctico de la Figura 3.9, se obtiene la secuencia **`id`**ε + **`id`**εε, que tomando en cuenta que la cadena vacía es el valor identidad de la concatenación, es decir si concatenamos la cadena vacía con otra cadena, es equivalente a tener solo la otra cadena; por lo que se obtiene, también, la secuencia **`id`** + **`id`**.

Evidentemente, al comparar los árboles de análisis sintáctico de las Figuras 3.8 y 3.9, es notable que el árbol de la gramática 3.3, sin recursión por la izquierda, es más frondoso que el obtenido con la gramática 3.2 que es recursiva por la izquierda; pero también es notable que la gramática 3.3 se puede implementar sin problema en un analizador sintáctico descendente predictivo.

Factorización por la izquierda

La factorización por la izquierda es bastante útil para producir gramáticas adecuadas para un analizador sintáctico descendente. El tener gramáticas factorizadas ayuda a clarificar que alternativa seguir cuando se tiene más de una. Generalmente, si se tiene un par de producciones para A del tipo $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ y la entrada comienza con una secuencia derivada de α, no queda claro si se debe expandir A a $\alpha\beta_1$ o a $\alpha\beta_2$. No obstante, si se factoriza por la izquierda, ya que tienen símbolos iguales α al inicio del cuerpo, las producciones originales se convierten en:

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$

Por ejemplo, si se tiene la siguiente gramática de una instrucción *if* con un *else* opcional:

$$\begin{aligned} S &\rightarrow i\,E\,t\,S \mid i\,E\,t\,S\,e\,S \mid a \\ E &\rightarrow b \end{aligned} \qquad (3.4)$$

Aquí, i, t y e representan las palabras reservadas **if**, **then** y **else**; E representa una expresión condicional y S representa una instrucción. Si se factoriza por la izquierda, esta gramática se convierte en:

$$\begin{aligned} S &\rightarrow i\,E\,t\,S\,S' \mid a \\ S' &\rightarrow e\,S \mid \varepsilon \\ E &\rightarrow b \end{aligned} \qquad (3.5)$$

En este caso, en la producción $S' \rightarrow e\,S \mid \varepsilon$, si llegase a haber un *else*, se sigue la derivación $S' \rightarrow e\,S$ y en caso de que no exista un *else*, se sigue la derivación $S' \rightarrow \varepsilon$. Lo anterior permite que no se tenga que recurrir a un retroceso en la secuencia de entrada a evaluar, lo que sí ocurriría si se sigue la gramática 3.4 que no está factorizada, en la que si tuvieramos un *else* a evaluar y se hiciera la derivación usando $S \rightarrow i\,E\,t\,S$, se tendría que retroceder hasta el inicio en la secuencia de entrada de evaluación para después hacer la derivación utilizando $S \rightarrow i\,E\,t\,S\,e\,S$.

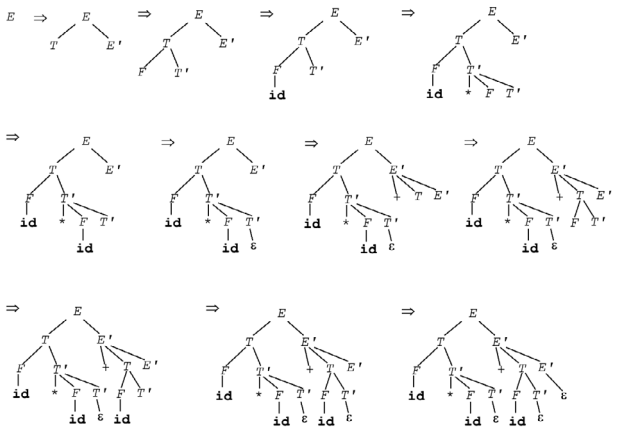
3.4. Análisis sintáctico descendente

El análisis sintáctico descendente analiza una cadena de componentes léxicos de entrada mediante la búsqueda de los pasos para una derivación por la izquierda (Louden, 2004). Se denomina descendente porque para generar el árbol de análisis sintáctico se realiza de la raíz a las hojas.

Por ejemplo, la secuencia de árboles de análisis sintáctico en la Figura 3.10 para la entrada **id*****id**+**id** es un análisis sintáctico descendente, de acuerdo a la gramática 3.3, que se repite a continuación:

$$
\begin{aligned}
&E \rightarrow T\,E' \\
&E' \rightarrow +\,T\,E' \mid \varepsilon \\
&T \rightarrow F\,T' \\
&T' \rightarrow *\,F\,T' \mid \varepsilon \\
&F \rightarrow (\,E\,) \mid \mathbf{id}
\end{aligned}
\qquad (3.3)
$$

Figura 3.10
*Análisis sintáctico descendente para id*id+id*



Considerando el análisis sintáctico descendente de la Figura 3.10, en la cual se construye un árbol con tres nodos T'. En el primer nodo T' (en preorden) se elige la producción $T' \rightarrow *\,F\,T'$; y en el segundo y tercer nodo se elige la producción $T' \rightarrow \varepsilon$. Un analizador sintáctico predictivo puede elegir una de las producciones T' mediante el análisis del siguiente símbolo de secuencia de entrada.

3.5. Análisis sintáctico descendente recursivo

Una manera de implementar un análisis sintáctico descendente es mediante el método de análisis sintáctico descendente recursivo. La idea en este análisis, es definir un procedimiento para cada no terminal de la gramática. Cada procedimiento sigue la secuencia del lado derecho de la producción, si se evalúa un no terminal, se llama al procedimiento correspondiente; y cuando se evalúa un terminal, se verifica que sea el terminal buscado en la producción, en caso de que sí lo sea, se continúa con la secuencia de la producción, de lo contrario se llama a una rutina de error. Cabe resaltar que en este método, cuando no se coincide con el terminal a evaluar o cuando se llame a la rutina de error, es muy probable que se tenga que hacer un retroceso o rastreo hacia atrás en la secuencia de entrada evaluada.

La ejecución comienza con el símbolo inicial de la gramática, esto es, para iniciar se hace una llamada al procedimiento del no terminal inicial; y de ahí se van derivando las demás llamadas a los procedimientos y la evaluación de los terminales. Finalmente, al explorar toda la cadena completa de entrada se define, si no hubo error, que la cadena de componentes es válida sintácticamente (Figura 3.11).

Figura 3.11

Bosquejo de un procedimiento para un no terminal en un analizador sintáctico descendente

```
void A() {
      Elegir una producción A, A → X1X2…Xk ;
      for( i = 1 a k ){
      if( Xi es un no terminal )
                  llamar al procedimiento Xi();
            else if( Xi es igual al símbolo de entrada actual a )
                    avanzar la entrada hasta el siguiente símbolo;
                  else /* rutina de error y retroceso()*/;
      }
}
```

Nota. Tomada de Aho, Lam, Sethi, & Ullman (2008, p. 219).

Por ejemplo: considerando la regla gramatical $F \rightarrow (E) \mid \mathbf{id}$, de la gramática 3.3, se puede diseñar un algoritmo como el de la Figura 3.12 con un método con el mismo nombre (F), el cual retorna un número entero que representa un código de error, que en el caso de no haberlo, retorna un 0. En este método se utiliza como entrada, un arreglo `componentes`, que proporciona el componente léxico a evaluar y que es resultado de la fase anterior de análisis léxico. Se tiene también un índice `i` que apunta al componente a evaluar en la

secuencia de entrada, el cual cada vez que se evalúa un token, avanza al siguiente componente en la secuencia de entrada `i++`; pero cuando se llega a una rutina de error se regresa al componente anterior `i--`, esto con la intención de evitar el saltarse algún componente en la revisión o dar por terminada la revisión sin haber completado la revisión. En el procedimiento primero se busca un '*(*', si se encuentra, entonces se llama a la función E, la cual también regresa un código de error, 0 si no lo hay. En caso de no haber error, entonces se continúa con la verificación de '*)*'. Esto sucede cuando se utiliza la producción $F \rightarrow (\ E\ )$. Por otro lado, se evalúa la producción $F \rightarrow \mathbf{id}$ que verifica si el componente léxico a evaluar es un **id**, en caso de no ser un **id**, entonces se llama a la rutina de error con el código de error correspondiente. Finalmente se retorna el código de error para su respectivo manejo por medio de un procedimiento que administre los errores, que es otro proceso importante en la compilación.

Figura 3.12

Código en Java para la regla gramatical F → (E) | id en un analizador sintáctico descendente

```
1     public static int F()
2     {
3         int error=0;
4         complex = componentes[i];
5         i++;
6         if (complex=='(') // F -> (E)
7         {
8             error = E();
9             if (error==0) //si no hay error; continua verificando
10            {
11                complex = componentes[i];
12                i++;
13                if (complex!=')')
14                {
15                    error = 1;    //No se encontró ')'
16                    i--; /*manejar retroceso*/
17                }
18            }
19        }
20        else
21        {
22            if (complex!=id) //si no es id
23            {
24                error = 2;
25                i--; /*manejar retroceso*/
26            }
27        }
28        return error;
```

3.6. Conjuntos Primero y Siguiente

La construcción de analizadores sintácticos es auxiliada por dos funciones, Primero y Siguiente, asociadas con una gramática *G*. Durante el análisis sintáctico descendente, Primero y Siguiente nos permiten elegir la producción que se va a aplicar, con base en el siguiente símbolo de entrada.

Si *X* es un símbolo de la gramática (un terminal o no tenninal) o ε, entonces el conjunto Primero(*X*), compuesto de terminales, y posiblemente de ε, se calcula aplicando para todos los símbolos gramaticales de *X* las siguientes reglas hasta que no puedan agregarse más terminales o ε a ningún conjunto Primero.

1. Si *X* es un terminal, entonces Primero(*X*) = {*X*}.

2. Si *X* es un no terminal y $X \rightarrow Y_1 Y_2 \ldots Y_k$ es una producción para cierta $k \geq 1$, entonces se coloca a en Primero(*X*) si para cierta *i*, a está en Primero(Y_i), y ε está en todas las funciones Primero(Y_1), ..., Primero(Y_{i-1}); es decir, $Y_1 \ldots Y_{i-1} \Rightarrow^* \varepsilon$. Si ε está en Primero($Y_j$) para todas las $j = 1, 2, \ldots, k$, entonces se agrega ε a Primero(*X*). Por ejemplo, todo lo que hay en Primero(Y_1) se encuentra sin duda en Primero(*X*). Si Y_i no deriva a ε, entonces no se agrega nada más a Primero(*X*), pero si $Y_1 \Rightarrow^* \varepsilon$, entonces se agrega Primero(Y_2), y así sucesivamente.

3. Si $X \rightarrow \varepsilon$ es una producción, entonces se agrega ε a Primero(*X*).

Para calcular Siguiente(*A*) para todos los no terminales *A*,se aplican las siguientes reglas hasta que no pueda agregarse nada a cualquier conjunto Siguiente.

1. Colocar *$* en Siguiente(*S*), en donde *S* es el símbolo inicial y *$* es el delimitador derecho de la entrada.

2. Si hay una producción $A \rightarrow \alpha B \beta$, entonces todo lo que hay en Primero(β) excepto ε está en Siguiente(*B*).

3. Si hay una producción $A \rightarrow \alpha B$, *o* una producción $A \rightarrow \alpha B \beta$, en donde Primero(β) contiene a ε,entonces todo lo que hay en Siguiente(*A*) está en Siguiente(*B*).

Por ejemplo, considerando la gramática 3.3, se tiene que:

Primero(*E*) = Primero(*T*) = Primero(*F*) = { '(', **id** }. Esto dado que *E* comienza con *T*, los primeros elementos de *E* son los mismos primeros de *T*, y debido a que *T* comienza con

F, los primeros elementos de *T* son los primeros elementos de *F*. *F* tiene dos opciones que comienzan con terminales por lo que sus primeros elementos son Primero(‘*(*‘) y Primero(**id).**

Primero(*E'*) = { +, ε } . Debido a que *E'* tiene dos producciones, una comienza con el terminal + y la otra es ε.

Primero(*T'*) = { *, ε } . Debido a que *T'* tiene dos producciones, una comienza con el terminal * y la otra es ε.

Siguiente(*E*) = Siguiente (*E'*) = { ‘)’, $}. Esto debido a que *E* es el símbolo inicial *$* debe estar en Siguiente(*E*), y la producción *F* → *(E)* hace que ‘)’ esté en Siguiente(*E*) por ser un terminal que apare después de *E.* Y para Siguiente(*E'*) se observa que la producción *E*→*TE'*, tiene la forma de la producción *A* → α*B*, por lo que aplicando la regla 3: todo lo que hay en Siguiente(*E*) está en Siguiente*(E').*

Siguiente(*T*) = Siguiente(*T'*) ={+, ‘)’, $}. Se Observa que *T* aparece en las producciones sólo seguido por *E'.* Por lo tanto, todo lo que esté en Primero(*E'*), excepto ε, debe estar en Siguiente(*T*); por eso el símbolo +. No obstante, como Primero(*E'*) contiene a ε, y *E'* es la cadena completa que va después de *T* en los cuerpos de las producciones *E*, todo lo que hay en Siguiente(*E*) también debe estar en Siguiente(*T*). Eso explica los símbolos *$* y el ‘)’. En cuanto a *T'*, como aparece sólo en los extremos de las producciones *T*, debe ser que Siguiente(*T')* = Siguiente(*T).*

Siguiente(*F*) = {+, *,), $}. El razonamiento es análogo al de *T* en el párrafo anterior.

3.7. Gramáticas *LL(1)*

Otra manera de implementar analizadores sintácticos descendentes en por medio de las gramáticas $\mathrm{LL}(1)$. La primera La primera “L” se refiere al hecho de que se procesa la entrada de izquierda a derecha. (del inglés “Left-right”). La segunda “L” hace referencia al hecho de que rastrea una derivación por la izquierda para la cadena de entrada. El número 1 entre paréntesis significa que se utiliza sólo un símbolo de entrada para predecir la dirección del análisis sintáctico.

Una gramática G es $\mathrm{LL}(1)$ si, y sólo si cada vez que $A \to \alpha \mid \beta$, son dos producciones distintas de G, se aplican las siguientes condiciones:

1. Para el no terminal a, tanto α como β derivan cadenas que empiecen con a.

2. A lo mucho, sólo α *o* β puede derivar la cadena vacía.

3. Si $\beta \Rightarrow^* \varepsilon$, entonces α no deriva a ninguna cadena que empiece con una terminal en Siguiente(A). De igual forma, si $\alpha \Rightarrow^* \varepsilon$, entonces β no deriva a ninguna cadena que comience con un terminal en Siguiente(A).

Pueden construirse analizadores sintácticos predictivos para las gramáticas $\mathrm{LL}(1)$, ya que puede seleccionarse la producción apropiada a aplicar para un no terminal con sólo analizar el símbolo de entrada actual. Para esto se pueden expresar las selecciones posibles construyendo una tabla de análisis sintáctico.

Una **tabla de análisis sintáctico** es esencialmente un arreglo bidimensional indizado por no terminales y terminales que contienen opciones de producción a emplear en el paso apropiado del análisis sintáctico (incluyendo el signo monetario $ para representar el final de la entrada) (Louden, 2004).

El siguiente algoritmo utiliza la información de los conjuntos Primero y Siguiente en una tabla de análisis predictivo $M[A, a]$, un arreglo bidimensional, en donde A es un no terminal y a es un terminal o el símbolo $\$$, el marcador de fin de la entrada.

Algoritmo 3.1: Construcción de una tabla de análisis sintáctico predictivo (Aho, Lam, Sethi, & Ullman, 2008, p. 224).

Entrada: La gramática G.

Salida: La tabla de análisis sintáctico M.

Método: Para cada producción $A \rightarrow \alpha$ de la gramática, hacer lo siguiente:

1. Para cada terminal a en Primero(A), agregar $A \rightarrow \alpha$ a $M[A, a]$.
2. Si ε está en Primero(α), entonces para cada terminal b en Siguiente(A), se agrega $A \rightarrow \alpha$ a $M[A, b]$. Si ε está en Primero(α) y $ se encuentra en Siguiente(A), se agrega $A \rightarrow \alpha$ a $M[A,\$]$ también.

La Figura 3.13 presenta la tabla de análisis sintáctico (Tabla 3.1) generada para la gramática 3.3. Los espacios en blanco en la matriz, son entradas de error; los espacios que no están en blanco indican una producción con la cual se expande un no terminal.

Tabla 3.1

Tabla de análisis sintáctico para la gramática 3.3

No terminal	Símbolo de entrada					
	id	**+**	*****	**(**	**)**	**$**
E	E → TE ‘			E → TE ‘		
E ‘		E ‘→ +TE ‘			E ‘ → ε	E ‘ → ε
T	T → FT ‘			T → FT ‘		
T ‘		T ‘ → ε	T ‘→ *FT ‘		T ‘ → ε	T ‘ → ε
F	*F* → **id**			*F* → (*E*)		

3.8. Análisis sintáctico predictivo no recursivo

Se puede construir un analizador sintáctico predictivo no recursivo a partir de una cadena w y una tabla de análisis sintáctico M para la gramática G, mediante el uso de una pila. Este analizador controlado por una tabla, tiene un búfer de entrada, una pila que contiene una secuencia de símbolos gramaticales, una tabla de análisis sintáctico construida por el Algoritmo 3.1, y un flujo de salida. El búfer de entrada contiene la cadena que se va a analizar, seguida por el marcador final $. Se utiliza también el símbolo $ para marcar la parte inferior de la pila, que al principio contiene el símbolo inicial de la gramática encima de $.

Algoritmo 3.2: Análisis sintáctico predictivo, controlado por una tabla. (Aho, Lam, Sethi, & Ullman, 2008, pág. 226)

Entrada: Una cadena *w* y una tabla de análisis sintáctico *M* para la gramática *G*.

Salida: Si *w* está en el lenguaje de la gramática *L*(*G*), una derivación por la izquierda de *w*; de lo contrario, una indicación de error.

Método: Al principio, el analizador sintáctico se encuentra en una configuración con *w*$ en el búfer de entrada, y el símbolo inicial *S* de *G* en la parte superior de la pila, por encima de $.

```
establecer ip para que apunte al primer símbolo de w;
establecer X con el símbolo de la parte superior de la pila;
while ( X ≠ $ ) { /* mientras la pila no está vacía*/
    if ( X es a ) extraer de la pila y avanzar ip; /* a=símbolo al que apunta ip */
    else if ( X es un terminal ) error( )
        else if ( M[X, a] es una entrada de error ) error( )
            else if ( M[X, a] = X→ Y1Y2... Yk ) {
                enviar de salida la producción X→ Y1Y2... Yk;
                extraer de la pila;
                meter Yk, Yk-1, ..., Y1 en la pila, con Y1 en la parte superior;
                }
    establecer X con el símbolo de la cima de la pila;
}
```

El analizador sintáctico se controla mediante un programa que considera a X, el símbolo en la parte superior de la pila, y a a, el símbolo de entrada actual. Si X es un no terminal, el analizador sintáctico elige una producción X mediante una consulta a la entrada $M[X, a]$ de la tabla de análisis sintáctico M. En cualquier otro caso, verifica si hay una coincidencia entre el terminal X y el símbolo de entrada actual a.

Figura 3.13
*Movimientos que realiza un analizador sintáctico predictivo con la entrada id * id + id*

Pila	Entrada	Acción
$*E*	**id*id+id$**	E → TE'
$*E' T*	**id*id+id$**	T → FT'
$*E' T' F*	**id*id+id$**	$F \rightarrow$ **id**
$*E' T'* id	**id*id+id$**	concuerda(id)
$*E' T'*	***id+id$**	T' → *FT'
$*E' T' F**	***id+id$**	concuerda(*)
$*E' T' F*	**id+id$**	$F \rightarrow$ **id**
$*E' T'* id	**id+id$**	concuerda(id)
$*E' T'*	**+id$**	T' → ε
$*E'*	**+id$**	E' → +TE'
$*E' T +*	**+id$**	concuerda(+)
$*E' T*	**id$**	T → FT'
$*E' T' F*	**id$**	$F \rightarrow$ **id**
$*E' T'* id	**id$**	concuerda(id)
$*E' T'*	$	T' → ε
$*E'*	$	E' → ε
$	$	aceptar( )

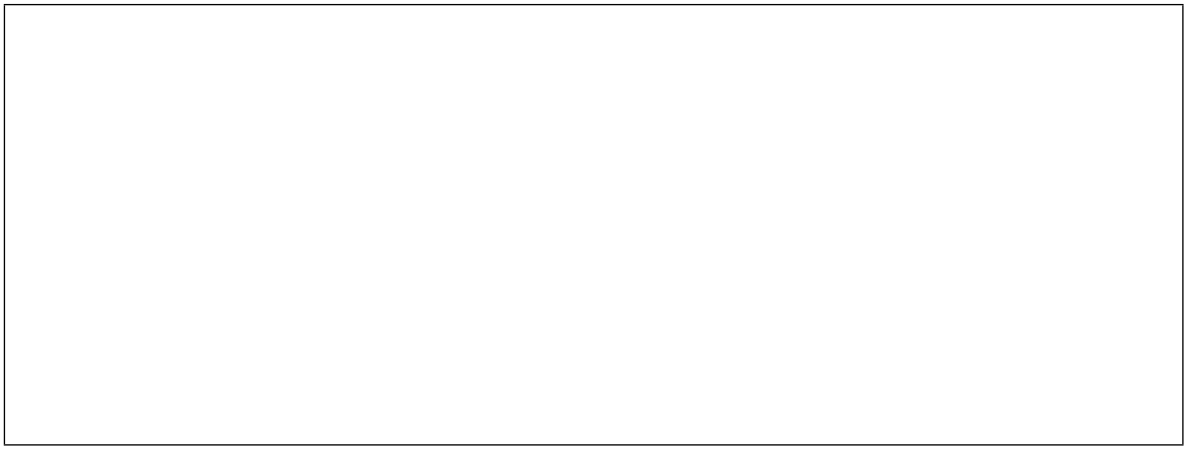
En el algoritmo 3.2 se observa el funcionamiento de un analizador sintáctico predictivo donde se utiliza una tabla de análisis sintáctico y una pila para validar la cadena de entrada. La Figura 3.13 nos presenta los movimientos que realiza un analizador sintáctico predictivo para la gramática 3.3 con la entrada **id * id + id**., utilizando la tabla de análisis sintáctico de la tabla 3.1. En la Figura se observa en la columna **ACCIÓN** la secuencia de producciones que se siguen para determinar que la secuencia de entrada es válida sintácticamente.

3.9. Ejercicios y actividades

1. a) Escriba una gramática que genere el conjunto de cadenas

{*s;* , *s;s;* , *s;s;s;* , … }.

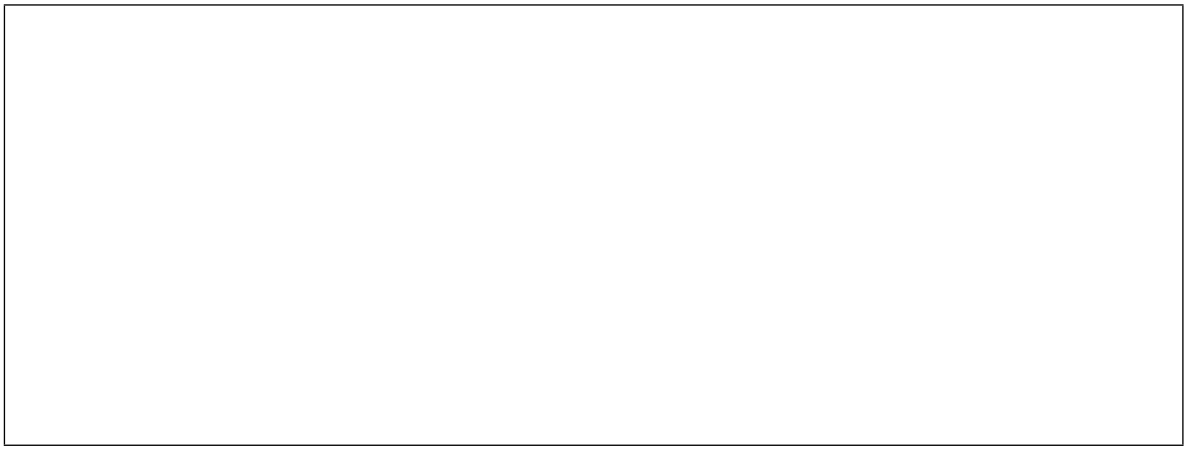
b) Genere un árbol sintáctico para la cadena *s;s;*



2. Considere la siguiente gramática:

rexp → *rexp* "|" *rexp*
| *rexp rexp*
| *rexp* "*"
| "(" *rexp* ")"
| **letra**

a) Genere un árbol sintáctico para la expresión regular (ab|b)*.



3. De las siguientes gramáticas, describa el lenguaje generado por la gramática y genere árboles sintácticos con las respectivas cadenas.

a) $S \rightarrow S\,S\,+ \mid S\,S\,* \mid a$ con la cadena $aa+a*$.

b) $S \rightarrow 0\,S\,1 \mid 0\,1$ con la cadena *000111.*

c) $S \rightarrow +\,S\,S \mid *\,S\,S \mid a$ con la cadena $+\,*\,aaa$.

4. ¿Cuál es el lenguaje generado por la siguiente gramática?

$S \rightarrow xSy \mid \varepsilon$

5. Genere el árbol sintáctico para la cadena ***zazabzbz*** utilizando la siguiente gramática:

$S \rightarrow zMNz$

$M \rightarrow aNa$

$N \rightarrow bNb$

$N \rightarrow z$

6. Demuestre que la gramática que se presenta a continuación es ambigua, mostrando que la cadena *ictictses* tiene derivaciones que producen distintos árboles de análisis sintáctico.

$S \rightarrow ictS$

$S \rightarrow ictSeS$

$S \rightarrow s$

7. Considere la siguiente gramática

$S \rightarrow (L) \mid a$

$L \rightarrow L , S \mid S$

Encuéntrense árboles de análisis sintáctico para las siguientes frases:

a) (a, a)

b) (a, (a, a))

c) (a, ((a, a), (a, a)))

8. Constrúyase un árbol sintáctico para la frase *not* ***(true or false)*** y la gramática:

bexpr → *bexpr* **or** *bterm* | *bterm*

bterm → *bterm* **and** *bfactor* | *bfactor*

bfactor → **not** *bfactor* | *(bexpr)* | **true** | **false**

9. Diseñe una gramática para el lenguaje del conjunto de todas las cadenas de símbolos 0 y 1 tales que todo 0 va inmediatamente seguido de al menos un 1.

10.Elimine la recursividad por la izquierda de la siguiente gramática:

$$S \rightarrow (L) \mid a$$

$$L \rightarrow L\ ,\ S \mid S$$

11. Dada la gramática $S \rightarrow (S) \mid \mathbf{x}$, escriba un pseudocódigo para el análisis sintáctico de esta gramática mediante el método descendente recursivo.

12. Qué movimientos realiza un analizador sintáctico predictivo con la entrada **(id+id)*id**, mediante el algoritmo 3.2, y utilizándose la tabla de análisis sintáctico de la tabla 3.1. (Tómese como ejemplo la Figura 3.13).

13. La gramática 3.2, sólo maneja las operaciones de suma y multiplicación, modifique esa gramática para que acepte, también, la resta y la división; Posteriormente, elimine la recursividad por la izquierda de la gramática completa y agregue la opción de que F, también pueda derivar en **num**, es decir, $F \rightarrow (E) \mid \mathbf{id} \mid \mathbf{num}$

14. Escriba un pseudocódigo (e implemente en Java) utilizando el método descendente recursivo para la gramática resultante del ejercicio anterior (ejercicio 13).

Capítulo 4. Análisis Semántico

Ma. del Carmen Nolasco Salcedo
Angélica Patricia Ávila Paz

4.1. Análisis Semántico

La semántica de un programa es su significado, en oposición a su sintaxis o estructura. La semántica determina el comportamiento del programa durante la ejecución, pero la mayoría de los lenguajes de programación tienen características que se pueden determinar antes de la ejecución e incluso no se pueden expresar de manera adecuada como sintaxis y analizarse por medio del analizador sintáctico (Louden, 2004).

Un analizador semántico utiliza el árbol de análisis sintáctico y la información de las tablas de símbolos para comprobar la consistencia semántica del programa fuente con la definición del lenguaje.

Una de las funciones importantes del análisis semántico es la comprobación o verificación de tipos, incluye las declaraciones y se verifica que cada operador tenga los operandos correspondientes.

El implementar un analizador semántico implica agregar las reglas semánticas características de cada lenguaje de programación al proceso de traducción. Sin embargo, para la implementación de analizadores semánticos no hay una forma específica de generarlos, es decir, para los analizadores léxicos se pueden utilizar expresiones regulares o diagramas de transición de estados, para los analizadores sintácticos se pueden utilizar gramáticas o autómatas; pero no hay un manera única respaldada por formalismos matemáticos para generarlos, dado a que cada lenguaje puede tener reglas semánticas muy diferentes y existe gran variedad en la implementación.

Un método para describir el análisis semántico es la identificación de atributos, o propiedades, de entidades del lenguaje que deben calcularse y escribir reglas semánticas, que expresan cómo el cálculo de tales atributos está relacionado con las reglas gramaticales del lenguaje. Un conjunto así de atributos y ecuaciones se denomina gramática con atributos. Estas gramáticas con atributos son útiles en los lenguajes que manejan semántica dirigida por sintaxis, la cual asegura que el contenido semántico se encuentra relacionado son su sintaxis.

4.2. Definiciones dirigidas por la sintaxis

Un esquema de traducción orientado a la sintaxis es una notación para especificar una traducción, uniendo los fragmentos de un programa a las producciones en una gramática. Un esquema de traducción es como una definición orientada a la sintaxis, sólo que el orden de evaluación de las reglas semánticas se especifica en forma explícita.

Una *definición dirigida por la sintaxis* es una gramática libre de contexto, junto con atributos y reglas (Aho, Lam, Sethi, & Ullman, 2008, p. 304). El método más general para la traducción orientada por la sintaxis es construir un árbol de análisis sintáctico y después calcular los valores de los atributos en los nodos del árbol. En muchos casos, la traducción puede realizarse durante el análisis sintáctico sin construir un árbol explícito. Para los no terminales se pueden manejar dos tipos de atributos: atributos sintetizados, se definen por reglas semánticas asociadas las producciones; y los atributos heredados, se definen por reglas semánticas asociadas con la producción en el padre del nodo.

La definición orientada por la sintaxis de la Figura 4.1 se basa en una gramática de expresiones aritméticas con los operadores + y *. En la definición orientada por la sintaxis, cada uno de los no terminales tiene dos atributos sintetizados: el atributo val, que tiene el valor numérico y se puede usar para determinar el valor de la expresión; y el atributo t que contiene el tipo de dato y se puede utilizar para comprobar la compatibilidad entre los tipos de datos del lenguaje.

Figura 4.1
Definición orientada por la sintaxis de expresiones aritméticas

Producción	Reglas semánticas
1) $E \rightarrow E_1 + T$	*E.val=E$_1$.val + T.val;* {*si E$_1$.t y T.t son* **int**,*entonces E.t*=**int**}
2) $E \rightarrow T$	E.val=T.val; E.t=T.t;
3) $t \rightarrow t_1 * f$	*T.val=T$_1$.val * F.val;* {*si T$_1$.t y F.t son* **int**,*entonces T.t*=**int**}
4) $t \rightarrow f$	T.val=F.val; T.t=F.t;

Figura 4.1
Definición orientada por la sintaxis de expresiones aritméticas

Producción	Reglas semánticas
5) $f \rightarrow (E)$	F.val=E.val; F.t=E.t;
6) $f \rightarrow$ dígito	*F.val=* **dígito***.valex;* {*F.t = tipo de datos de* **dígito***, en este caso* **int**}

La producción 1) $E \rightarrow E_1 + T$ establece el valor de toda la expresión, calcula *val* como la suma aritmética de los valores en E_1 y T. Para el atributo t se aplica la regla semántica de que si los dos operandos, en la expresión aritmética, son enteros, entonces la expresión es *int*.

La producción 2) $E \rightarrow T$ se establecen los atributos *val* y t de E, a partir de los valores de los respectivos atributos de T.

En la producción 3) $T \rightarrow T_1 * F$ se asemeja a la producción 1) calcula *val* como la multiplicación aritmética de los valores en E_1 y T. Para el atributo t se aplica la regla semántica de que si los dos operandos, en la expresión aritmética, son enteros, entonces la expresión es *int*.

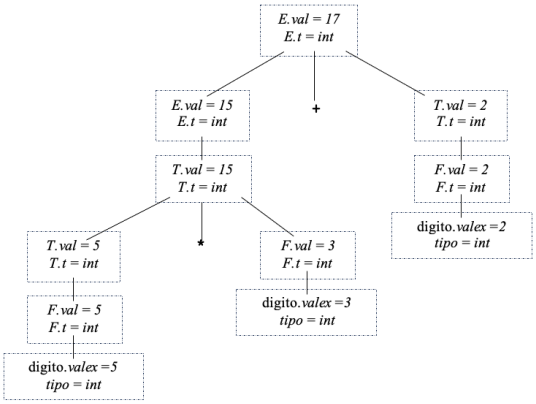
Para la producción 4) $T \rightarrow F$, se establecen los atributos *val* y t de T, a partir de los valores de los respectivos atributos de F.

La producción 5) $F \rightarrow (E)$, de forma similar a las producciones 2) y 4), se establecen los atributos *val* y t de T, a partir de los valores de los respectivos atributos de F.

Finalmente, en la producción 6) $F \rightarrow$ **dígito**, el atributo *val* de F se establece a partir del valor numérico del dígito; y en el atributo t de F se establece el tipo de dato del número, que en este caso es un **int**. Cabe hacer notar, que se hace interacción con la tabla de símbolos.

En la Figura 4.2 muestra un árbol de análisis sintáctico anotado para la expresión 5*3+2 construida mediante el uso de la gramática y las reglas semánticas de se despliegan en la Figura 4.1. Cada uno de los nodos tiene los atributos val y t que se van generando aplicando las reglas semánticas correspondientes a la definición orientada por la sintaxis mostrada en la Figura 4.1.

Figura 4.2
*Árbol de análisis sintáctico anotado para 5*3+2*



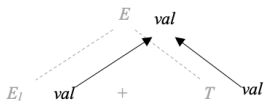
4.3. Grafos de dependencias

Los grafos de dependencias son una herramienta útil en la determinación de un orden de evaluación para las instancias de los atributos en un árbol de análisis sintáctico dado. Mientras que un árbol de análisis sintáctico anotado muestra los valores de los atributos, un grafo de dependencias puede colaborar a determinar cómo pueden calcularse esos valores.

Un *grafo de dependencias* describe el flujo de información entre las instancias de atributos en un árbol de análisis sintáctico específico; una flecha de una instancia de atributo a otra significa que el valor de la primera se necesita para calcular la segunda (Aho, Lam, Sethi, & Ullman, 2008).

Por ejemplo, considerando la producción $E \rightarrow E_1 + T$ con la regla semántica $E.val = E_1.val + T.val$. En cada nodo etiquetado como E, con hijos correspondientes al cuerpo de esta producción, el atributo val en el nodo padre se calcula usando los valores del atributo val de los dos hijos. La Figura 4.3 muestra el grafo de dependencias para cuando se utilice esta producción. En este caso el valor de $E.val$ depende de $E_1.val$ y de $T.val$.

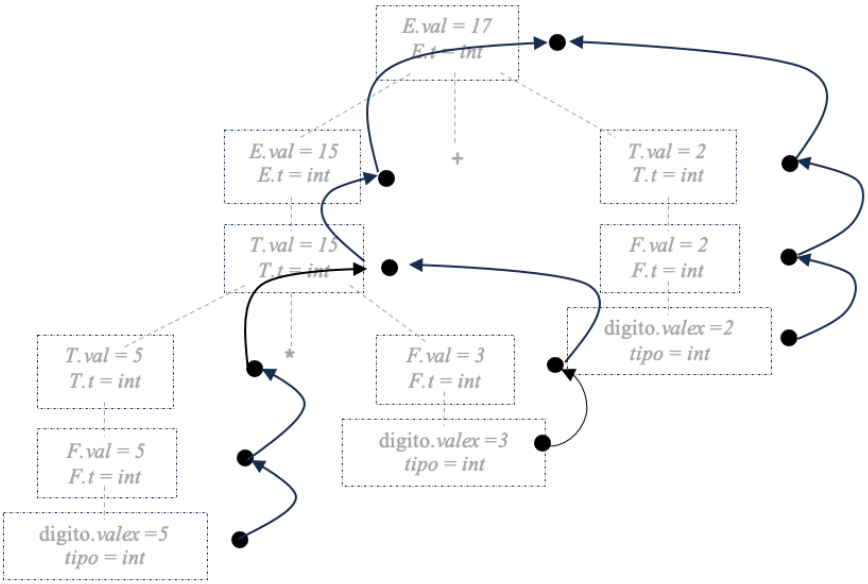
Figura 4.3
E.val se sintetiza a partir de E1.val + T.val



En la Figura 4.4 se muestra un grafo de dependencias sobre el árbol de análisis sintáctico anotado de la Figura 4.2. Nótese que en este ejemplo los nodos tienen los atributos val y t, lo que implica generar dos grafos, uno para cada atributo; sin embargo, se obtienen árboles similares por el valor de atributos sintetizados, por lo que sólo se muestra un grafo.

Figura 4.4

*Grafo de dependencias sobre árbol de análisis sintáctico anotado para 5*3+2*



4.4. Ejercicios y actividades

1. Para la siguiente gramática, elimine la recursión por la izquierda y defina reglas semánticas adecuadas para cada producción de la gramática que maneje los tipos de datos en las declaciones. Posteriormente elabore el grafo de dependencias para la declaración **int** $\mathbf{id}_1$, $\mathbf{id}_2$.

$D \rightarrow T\ L$

$T \rightarrow$ **int**

$T \rightarrow$ **float**

$L \rightarrow L_1$, **id**

$L \rightarrow$ **id**

2. Considere la siguiente gramática; defina atributos y reglas semánticas adecuadas para la gramática que generen el resultado de la expresión aritmética a validar, así como su verificación de que los tipos de datos son compatibles.

3. Utilizando la definición orientada por la sintaxis de la Figura 4.1, elabore un árbol sintáctico anotado y un grafo de precedencias para la expresión $(6 + 2)$

Bibliografía

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2008). *Compiladores principios, técnicas y herramientas*. Pearson.

Aho, A. V., Sethi, R., & Ullman, J. D. (1990). *Compiladores Principios, técnicas y herramientas*. Addison-Wesley Iberoamericana.

Deitel, P. J., & Deitel, H. M. (2008). *Java, Cómo programar*. Pearson.

Louden, K. C. (2004). *Construcción de Compiladores*. Thompson.

Ruiz Catalán, J. (2010). *Compiladores, Teoría e implementación*. Alfaomega.

Zakhour, S., Hommel, S., Royal, J., Rabinovitch, I., Risser, T., & Hoeber, M. (2006). *The Java Tutorial Fourth Edition: A Short Course on the Basics*. Addison Wesley Professional.

Semblanzas de autores y autoras

Diego Ulises Carranza Sahagún

diego.carranza@academicos.udg.mx

Doctor en Ciencias con Orientación en Computación y Automatización, Universidad de Guadalajara, México. Profesor Titular del Departamento de Ciencias Básicas del Centro Universitario de la Ciénega de la Universidad de Guadalajara, México. Temas de Investigación: Observadores en eventos discretos, Simuladores y Laboratorios Virtuales en la Educación, Desarrollo y uso de herramientas de software para objetos de aprendizaje.

Kleophé Alfaro Castellanos

kleophe.alfaro@academicos.udg.mx

Doctora en Ciencias de la Educación, Universidad de Santander. Institución de adscripción Universidad de Guadalajara, Centro Universitario de la Ciénega, Departamento de Ciencias Básicas. Temas de investigación: En educación: Objetos de aprendizajes, Aprendizaje activo, Integración de metodologías innovadoras con pertinencia cultural y sus efectos en el proceso educativo.

Ma. del Carmen Nolasco Salcedo

ma.nolasco@academicos.udg.mx

Profesor, investigador, adscrito al Departamento de ciencias Básicas en el Centro Universitario de la Ciénega, Universidad de Guadalajara, México. La Doctora Carmen Nolasco es Informática, tiene el Doctorado en Educación por parte de la Universidad Santander en Tamaulipas y la maestría en especialidad de Programación por parte Universidad central “ Marta Abreu” de las Villas, Cuba. En el campo de la Programación, ha trabajado en proyectos tales como Sistema de Control de Usuarios del Centro de Computo(SisCUC) y Sistema de Olimpiadas del Saber(SIOS).

José Ávila Paz

jose.apaz@academicos.udg.mx

Profesor e investigador del Centro Universitario Ciénega de la Universidad de Guadalajara, México. Ingeniero en Comunicaciones y Electrónica. Doctor en Ciencias con mención en Física por la Universidad de Guadalajara y Maestro en Ciencias con especialidad en Programación por la Universidad Central “Marta Abreu” de las Villas, Cuba.

Angélica Patricia Ávila Paz

angelica.apaz@academicos.udg.mx

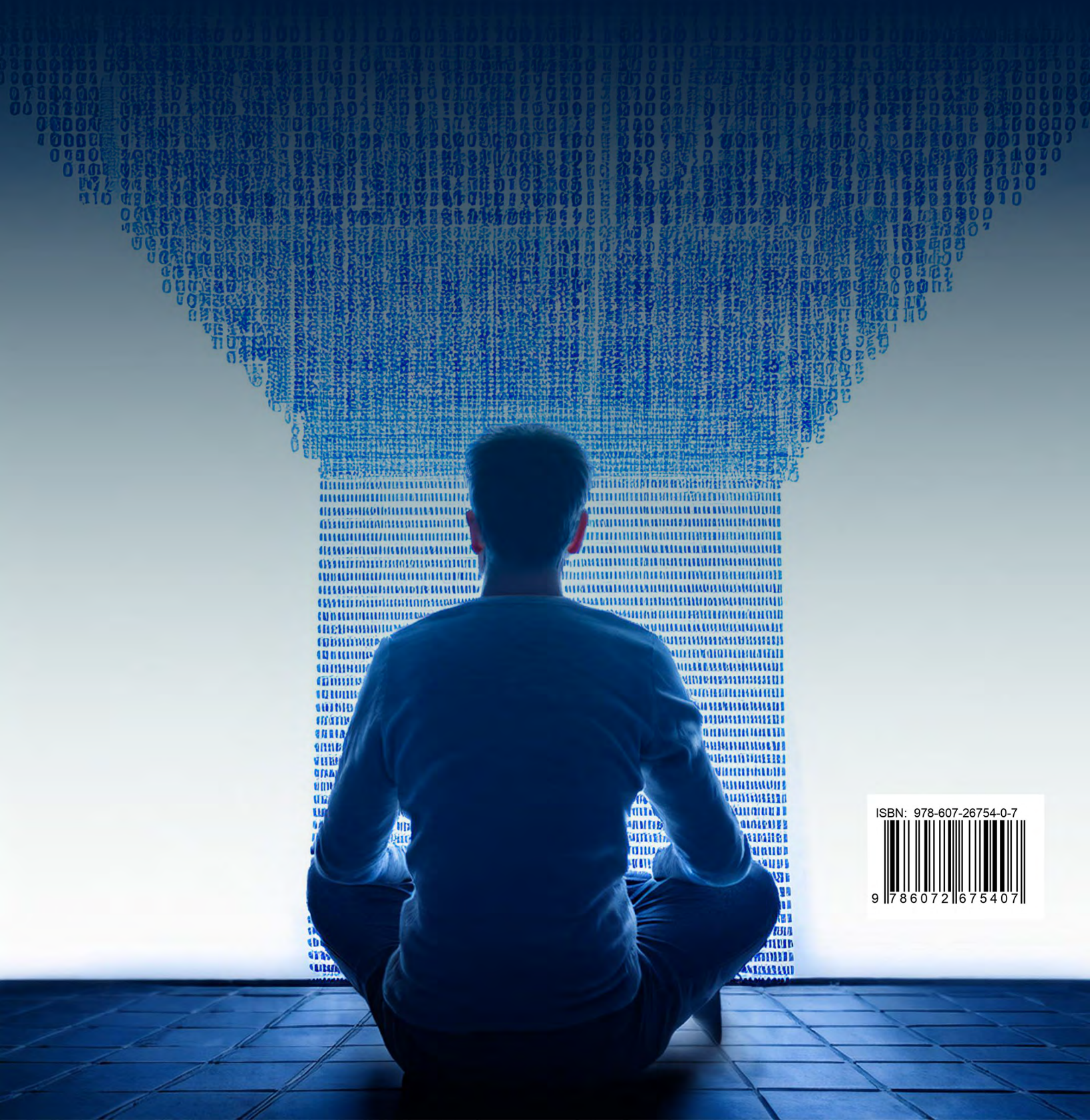
Profesora e Investigadora de la Escuela Regional de Educación Media Superior Ocotlán, perteneciente al Sistema de Educación Media Superior de la Universidad de Guadalajara, México. Licenciada en Geografía por la Universidad de Guadalajara. Maestra en Educación Superior por el Centro Universitario de la Ciénega de la Universidad de Guadalajara y Doctora en Metodología de la Enseñanza por el Instituto de Estudios Pedagógicos de Zapopan, Jalisco, México. Formadora y promotora en lectura y escritura, obteniendo el Premio FIL 2020 en la categoría de audiocuento.

Oscar Antonio Zárate Águila

oscar.zarate@academicos.udg.mx

Doctor en Computación egresado de la Universidad Técnica de Clausthal (Alemania). Imparte Modelado y Animación 3D y programación de videojuegos en C#. Labora como profesor de tiempo completo en el Departamento de Ciencias Tecnológicas del Centro Universitario de la Ciénega.

COMPILADORES
Fases de análisis



ISBN: 978-607-26754-0-7