



**EDUCACIÓN**  
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO  
INSTITUTO TECNOLÓGICO DE IZTAPALAPA



## Instituto tecnológico de Iztapalapa

### Ingeniería en Sistemas Computacionales

#### Lenguajes y automatas 2

Santana González Jesús Salvador: 171080127

Cabrera Ramírez Gerardo: 171080187

Morales Carrillo Gerardo: 171080120

#### Actividad semana 14

16/10/2020

## Procesadores de lenguaje

Los lenguajes de programación son notaciones que describen los cálculos a las personas y las máquinas. Nuestra percepción del mundo en que vivimos depende de los lenguajes de programación, ya que todo el software que se ejecuta en todas las computadoras se escribió en algún lenguaje de programación. Los sistemas de software que se encargan de esta traducción se llaman compiladores. Un compilador es un programa que puede leer un programa en un lenguaje (el lenguaje fuente) y traducirlo en un programa equivalente en otro lenguaje (el lenguaje destino). El programa destino en lenguaje máquina que produce un compilador es, por lo general, más rápido que un intérprete al momento de asignar las entradas a las salidas.

## La estructura de un compilador

La parte del análisis divide el programa fuente en componentes e impone una estructura gramatical sobre ellas. Después utiliza esta estructura para crear una representación intermedia del programa fuente. La parte del análisis también recolecta información sobre el programa fuente y la almacena en una estructura de datos llamada tabla de símbolos, la cual se pasa junto con la representación intermedia a la parte de la síntesis. La parte de la síntesis construye el programa destino deseado a partir de la representación intermedia y de la información en la tabla de símbolos. A la parte del análisis se le llama comúnmente el front-end del compilador; la parte de la síntesis (propriamente la traducción) es el back-end. Algunos compiladores tienen una fase de optimización de código independiente de la máquina, entre el front-end y el back-end. El propósito de esta optimización es realizar transformaciones sobre la representación intermedia, para que el back-end pueda producir un mejor programa destino de lo que hubiera producido con una representación intermedia sin optimizar.

### Análisis de léxico

El analizador de léxico lee el flujo de caracteres que componen el programa fuente y los agrupa en secuencias significativas, conocidas como lexemas.

### Análisis sintáctico

El parser (analizador sintáctico) utiliza los primeros componentes de los tokens producidos por el analizador de léxico para crear una representación intermedia en forma de árbol que describa la estructura gramatical del flujo de tokens. Una representación típica es el árbol sintáctico, en el cual cada nodo interior representa una operación y los hijos del nodo representan los argumentos de la operación.

Las fases siguientes del compilador utilizan la estructura gramatical para ayudar a analizar el programa fuente y generar el programa destino.

### Análisis semántico

Utiliza el árbol sintáctico y la información en la tabla de símbolos para comprobar la consistencia semántica del programa fuente con la definición del lenguaje. También recopila información sobre el tipo y la guarda, ya sea en el árbol sintáctico o en la tabla de símbolos, para usarla más tarde durante la generación de código intermedio. Una parte importante del

análisis semántico es la comprobación (verificación) de tipos, en donde el compilador verifica que cada operador tiene operandos que coincidan.

Cabrera Ramírez Gerardo

("Análisis Léxico (partes 2 y 3)")

### **Traducción binaria**

La tecnología de compiladores puede utilizarse para traducir el código binario para una máquina al código binario para otra máquina distinta, con lo cual se permite a una máquina ejecutar los programas que originalmente eran compilados para otro conjunto de instrucciones. La traducción binaria también puede usarse para ofrecer compatibilidad inversa.

### **Síntesis de hardware**

No sólo la mayoría del software está escrito en lenguajes de alto nivel; incluso hasta la mayoría de los diseños de hardware se describen en lenguajes de descripción de hardware de alto nivel, como Verilog y VHDL (Very High-Speed Integrated Circuit Hardware Description Language, Lenguaje de descripción de hardware de circuitos integrados de muy alta velocidad).

### **Simulación compilada**

La simulación es una técnica general que se utiliza en muchas disciplinas científicas y de ingeniería para comprender un fenómeno, o para validar un diseño. Por lo general, las entradas de los simuladores incluyen la descripción del diseño y los parámetros específicos de entrada para esa ejecución específica de la simulación. Las simulaciones pueden ser muy costosas. Por lo general, necesitamos simular muchas alternativas de diseño posibles en muchos conjuntos distintos de entrada, y cada experimento puede tardar días en completarse, en una máquina de alto rendimiento.

### **Al hablar sobre el análisis léxico, utilizamos tres términos distintos, pero relacionados:**

Un token es un par que consiste en un nombre de token y un valor de atributo opcional. El nombre del token es un símbolo abstracto que representa un tipo de unidad léxica; por ejemplo, una palabra clave específica o una secuencia de caracteres de entrada que denotan un identificador. Los nombres de los tokens son los símbolos de entrada que procesa el analizador sintáctico. A partir de este momento, en general escribiremos el nombre de un token en negrita. Con frecuencia nos referiremos a un token por su nombre.

Un patrón es una descripción de la forma que pueden tomar los lexemas de un token. En el caso de una palabra clave como token, el patrón es sólo la secuencia de caracteres

que forman la palabra clave. Para los identificadores y algunos otros tokens, el patrón es una estructura más compleja que se relaciona mediante muchas cadenas.

Un lexema es una secuencia de caracteres en el programa fuente, que coinciden con el patrón para un token y que el analizador léxico identifica como una instancia de ese token.

### Atributos para los tokens

Cuando más de un lexema puede coincidir con un patrón, el analizador léxico debe proporcionar a las subsiguientes fases del compilador información adicional sobre el lexema específico que coincidió.

Otras funciones que realiza :

- Eliminar los comentarios del programa.
- Eliminar espacios en blanco, tabuladores, retorno de carro, etc, y en general, todo aquello que carezca de significado según la sintaxis del lenguaje.
- Reconocer los identificadores de usuario, números, palabras reservadas del lenguaje, ..., y tratarlos correctamente con respecto a la tabla de símbolos (sólo en los casos que debe de tratar con la tabla de símbolos).
- Llevar la cuenta del número de línea por la que va leyendo, por si se produce algún error, dar información sobre dónde se ha producido.

### Errores léxicos

Sin la ayuda de los demás componentes es difícil para un analizador léxico saber que hay un error en el código fuente. La estrategia de recuperación más simple es la recuperación en “modo de pánico”. Eliminamos caracteres sucesivos del resto de la entrada, hasta que el analizador léxico pueda encontrar un token bien formado al principio de lo que haya quedado de entrada. Esta técnica de recuperación puede confundir al analizador sintáctico, pero en un entorno de computación interactivo, puede ser bastante adecuado. Otras de las posibles acciones de recuperación de errores son:

1. Eliminar un carácter del resto de la entrada.
2. Insertar un carácter faltante en el resto de la entrada.
3. Sustituir un carácter por otro.
4. Transponer dos caracteres adyacentes.

Las transformaciones como éstas pueden probarse en un intento por reparar la entrada. La estrategia más sencilla es ver si un prefijo del resto de la entrada puede transformarse en un lexema válido mediante una transformación simple. Esta estrategia tiene sentido, ya que en la práctica la mayoría de los errores léxicos involucran a un solo carácter. Una estrategia de corrección más general es encontrar el menor número de transformaciones necesarias para convertir el programa fuente en uno que consista

sólo de lexemas válidos, pero este método se considera demasiado costoso en la práctica como para que valga la pena realizarlo.

### Uso de búfer en la entrada

Antes de hablar sobre el problema de reconocer lexemas en la entrada, vamos a examinar algunas formas en las que puede agilizarse la simple pero importante tarea de leer el programa fuente. Esta tarea se dificulta debido a que a menudo tenemos que buscar uno o más caracteres más allá del siguiente lexema para poder estar seguros de que tenemos el lexema correcto.

En C, los operadores de un solo carácter como `-`, `=` o `<` podrían ser también el principio de un operador de dos caracteres, como `->`, `==` o `<=`. Por ende, vamos a presentar un esquema de dos búferes que se encarga de las lecturas por adelantado extensas sin problemas. Después consideraremos una mejora en la que se utilizan “centinelas” para ahorrar tiempo al verificar el final de los búferes.

### Pares de búferes

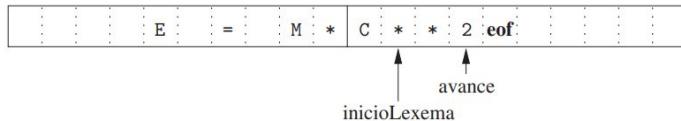
Debido al tiempo requerido para procesar caracteres y al extenso número de caracteres que se deben procesar durante la compilación de un programa fuente extenso, se han desarrollado técnicas especializadas de uso de búferes para reducir la cantidad de sobrecarga requerida en el procesamiento de un solo carácter de entrada.

Cada búfer es del mismo tamaño  $N$ , y por lo general  $N$  es del tamaño de un bloque de disco (es decir, 4 096 bytes). Mediante el uso de un comando de lectura del sistema podemos leer  $N$  caracteres y colocarlos en un búfer, en vez de utilizar una llamada al sistema por cada carácter. Si quedan menos de  $N$  caracteres en el archivo de entrada, entonces un carácter especial, representado por `eof`, marca el final del archivo fuente y es distinto a cualquiera de los posibles caracteres del programa fuente. Se mantienen dos apuntadores a la entrada:

1. El apuntador `inicioLexema` marca el inicio del lexema actual, cuya extensión estamos tratando de determinar.
2. El apuntador `avance` explora por adelantado hasta encontrar una coincidencia en el patrón; durante el resto del capítulo cubriremos la estrategia exacta mediante la cual se realiza esta determinación.

Una vez que se determina el siguiente lexema, `avance` se coloca en el carácter que se encuentra en su extremo derecho. Después, una vez que el lexema se registra como un valor de atributo de un token devuelto al analizador sintáctico, `inicioLexema` se coloca en el carácter que va justo después del lexema que acabamos de encontrar.

### Centinelas



Si utilizamos el esquema en la forma descrita, debemos verificar, cada vez que movemos el apuntador avance, que no nos hayamos salido de uno de los búferes; si esto pasa, entonces también debemos recargar el otro búfer. Así, por cada lectura de caracteres hacemos dos pruebas: una para el final del búfer y la otra para determinar qué carácter se lee (esta última puede ser una bifurcación de varias vías). Podemos combinar la prueba del final del búfer con la prueba del carecer actual si extendemos cada búfer para que contenga un valor centinela al final. El centinela es un carácter especial que no puede formar parte del programa fuente, para lo cual una opción natural es el carácter eof.

### Cadenas y lenguajes

Un alfabeto es un conjunto finito de símbolos. Algunos ejemplos típicos de símbolos son las letras, los dígitos y los signos de puntuación. El conjunto  $\{0, 1\}$  es el alfabeto binario. ASCII es un ejemplo importante de un alfabeto; se utiliza en muchos sistemas de software.

Una cadena sobre un alfabeto es una secuencia finita de símbolos que se extraen de ese alfabeto. En la teoría del lenguaje, los términos “oración” y “palabra” a menudo se utilizan como sinónimos de “cadena”. La longitud de una cadena  $s$ , que por lo general se escribe como  $|s|$ , es el número de ocurrencias de símbolos en  $s$ .

### Operaciones en los lenguajes

En el análisis léxico, las operaciones más importantes en los lenguajes son la unión, la concatenación y la cerradura

OPERACIÓN	DEFINICIÓN Y NOTACIÓN
Unión de $L$ y $M$	$L \cup M = \{ s \mid s \text{ está en } L \text{ o } s \text{ está en } M \}$
Concatenación de $L$ y $M$	$LM = \{ st \mid s \text{ está en } L \text{ y } t \text{ está en } M \}$
Cerradura de Kleene de $L$	$L^* = \bigcup_{i=0}^{\infty} L^i$
Cerradura positivo de $L$	$L^+ = \bigcup_{i=1}^{\infty} L^i$

Figura 3.6: Definiciones de las operaciones en los lenguajes

### Expresiones regulares

Se usa para describir a todos los lenguajes que puedan construirse a partir de estos operadores, aplicados a los símbolos de cierto alfabeto. En esta notación, si letra\_ se establece de manera que represente a cualquier letra o al guion bajo, y dígito\_ se Figura 3.6: Definiciones de las operaciones en los lenguajes OPERACIÓN DEFINICIÓN Y NOTACIÓN Unión de L y M  $L \cup M = \{ s \mid s \text{ está en } L \text{ o } s \text{ está en } M \}$  Concatenación de L y M  $LM = \{ st \mid s \text{ está en } L \text{ y } t \text{ está en } M \}$  Cerradura de Kleene de L  $L^* = \bigcup_{i=0}^{\infty} L^i$  Cerradura positivo de L  $L^+ = \bigcup_{i=1}^{\infty} L^i$



AM establece de manera que represente a cualquier dígito, entonces podríamos describir el lenguaje de los identificadores de C mediante lo siguiente:

letra\_( letra\_ | dígito )\*

La barra vertical de la expresión anterior significa la unión, los paréntesis se utilizan para agrupar las subexpresiones, el asterisco indica "cero o más ocurrencias de", y la yuxtaposición de letra\_ con el resto de la expresión indica la concatenación. Las expresiones regulares se construyen en forma recursiva a partir de las expresiones regulares más pequeñas, usando las reglas que describiremos a continuación. Cada expresión regular  $r$  denota un lenguaje  $L(r)$ , el cual también se define en forma recursiva, a partir de los lenguajes denotados por las subexpresiones de  $r$ . He aquí las reglas que definen las expresiones regulares sobre cierto alfabeto  $\Sigma$ , y los lenguajes que denotan dichas expresiones.

**BASE:** Hay dos reglas que forman la base:

1.  $a$  es una expresión regular, y  $L(a) = \{a\}$ ; es decir, el lenguaje cuyo único miembro es la cadena vacía.
2. Si  $a$  es un símbolo en  $\Sigma$ , entonces  $a$  es una expresión regular, y  $L(a) = \{a\}$ , es decir, el lenguaje con una cadena, de longitud uno, con  $a$  en su única posición. Tenga en cuenta que por convención usamos cursiva para los símbolos, y negrita para su correspondiente expresión regular.

**INDUCCIÓN:** Hay cuatro partes que constituyen la inducción, mediante la cual las expresiones regulares más grandes se construyen a partir de las más pequeñas. Suponga que  $r$  y  $s$  son expresiones regulares que denotan a los lenguajes  $L(r)$  y  $L(s)$ , respectivamente.

1.  $(r)l(s)$  es una expresión regular que denota el lenguaje  $L(r) \cup L(s)$ .
2.  $(r)(s)$  es una expresión regular que denota el lenguaje  $L(r)L(s)$ .
3.  $(r)^*$  es una expresión regular que denota  $L((r))^*$ .
4.  $(r)$  es una expresión regular que denota a  $L(r)$ . Esta última regla dice que podemos agregar pares adicionales de paréntesis alrededor de las expresiones, sin cambiar el lenguaje que denotan.

Según su definición, las expresiones regulares a menudo contienen pares innecesarios de paréntesis. Tal vez sea necesario eliminar ciertos pares de paréntesis, si adoptamos las siguientes convenciones:

- a) El operador unario  $*$  tiene la precedencia más alta y es asociativo a la izquierda.
- b) La concatenación tiene la segunda precedencia más alta y es asociativa a la izquierda.

c) | tiene la precedencia más baja y es asociativo a la izquierda.

A un lenguaje que puede definirse mediante una expresión regular se le llama conjunto regular. Si dos expresiones regulares  $r$  y  $s$  denotan el mismo conjunto regular, decimos que son equivalentes y escribimos  $r = s$ . Por ejemplo,  $(a|b) = (b|a)$ . Hay una variedad de leyes algebraicas para las expresiones regulares; cada ley afirma que las expresiones de dos formas distintas son equivalentes. La figura 3.7 muestra parte de las leyes algebraicas para las expresiones regulares arbitrarias  $r$ ,  $s$  y  $t$ .

LEY	DESCRIPCIÓN
$r s = s r$	es conmutativo
$r (s t) = (r s) t$	es asociativo
$r(st) = (rs)t$	La concatenación es asociativa
$r(s t) = rs rt; (s t)r = sr tr$	La concatenación se distribuye sobre
$\epsilon r = r\epsilon = r$	$\epsilon$ es la identidad para la concatenación
$r^* = (r \epsilon)^*$	$\epsilon$ se garantiza en un cerradura
$r^{**} = r^*$	* es idempotente

### Extensiones de las expresiones regulares

Desde que Kleene introdujo las expresiones regulares con los operadores básicos para la unión, la concatenación y la cerradura de Kleene en la década de 1950, se han agregado muchas extensiones a las expresiones regulares para mejorar su habilidad al especificar los patrones de cadenas.

1. Una o más instancias. El operador unario postfijo  $+$  representa la cerradura positivo de una expresión regular y su lenguaje. 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  $*$ . Dos leyes algebraicas útiles,  $r^* = r+|$  y  $r+ = rr^* = r^*r$  relacionan la cerradura de Kleene y la cerradura positiva.
2. Cero o una instancia. ¿El operador unario postfijo  $?$  significa “cero o una ocurrencia”. Es decir,  $r?$  es equivalente a  $r|$ , o dicho de otra forma,  $L(r?) = L(r) \cup \{\}$ . El operador  $?$  tiene la misma precedencia y asociatividad que  $*$  y  $+$ .
3. Clases de caracteres. Una expresión regular  $a_1|a_2|\dots|a_n$ , en donde las  $a_i$ s son cada una símbolos del alfabeto, puede sustituirse mediante la abreviación  $[a_1a_2\dots a_n]$ . Lo que es más importante, cuando  $a_1, a_2, \dots, a_n$  forman una secuencia lógica, por ejemplo, letras mayúsculas, minúsculas o dígitos consecutivos, podemos sustituirlos por  $a_1-a_n$ ; es decir, sólo la primera y última separadas por un guion corto. Así,  $[abc]$  es la abreviación para  $a|b|c$ , y  $[a-z]$  lo es para  $a|b|\dots|z$ .

## Reconocimiento de tokens

Para operar, usamos los operadores de comparación de lenguajes como Pascal o SQL, en donde = es “es igual a” y <> es “no es igual a”, ya que presenta una estructura interesante de lexemas. Las terminales de la gramática, que son if, then, else, opel, id y numero, son los nombres de tokens en lo que al analizador léxico respecta.

<i>digito</i>	$\rightarrow [0-9]$
<i>digitos</i>	$\rightarrow digito^+$
<i>numero</i>	$\rightarrow digitos \ ( . \ digitos )? \ ( \ E \ [+-]? \ digitos \ )?$
<i>letra</i>	$\rightarrow [A-Za-z]$
<i>id</i>	$\rightarrow letra \ ( letra \mid digito )^*$
<i>if</i>	$\rightarrow if$
<i>then</i>	$\rightarrow then$
<i>else</i>	$\rightarrow else$
<i>oprel</i>	$\rightarrow < \mid > \mid <= \mid >= \mid = \mid <>$

Para este lenguaje, el analizador léxico reconocerá las palabras clave if, then y else, así como los lexemas que coinciden con los patrones para opel, id y numero. Para simplificar las cosas, vamos a hacer la suposición común de que las palabras clave también son palabras reservadas; es decir, no son identificadores, aun cuando sus lexemas coinciden con el patrón para identificadores.

Además, asignaremos al analizador léxico el trabajo de eliminar el espacio en blanco, reconociendo el “token” ws definido por:

$$ws \rightarrow ( \text{blanco} \mid \text{tab} \mid \text{nuevalinea} )^+$$

Aquí, blanco, tab y nuevalinea son símbolos abstractos que utilizamos para expresar los caracteres ASCII de los mismos nombres. El token ws es distinto de los demás tokens porque cuando lo reconocemos, no lo regresamos al analizador sintáctico, sino que reiniciamos el analizador léxico a partir del carácter que va después del espacio en blanco. El siguiente token es el que se devuelve al analizador sintáctico.

LEXEMAS	NOMBRE DEL TOKEN	VALOR DEL ATRIBUTO
Cualquier <i>ws</i>	—	—
if	if	—
Then	then	—
else	else	—
Cualquier <i>id</i>	id	Apuntador a una entrada en la tabla
Cualquier <i>numero</i>	numero	Apuntador a una entrada en la tabla
<	oprel	LT
<=	oprel	LE
=	oprel	EQ
>	oprel	NE
>=	oprel	GT
>=	oprel	GE

## Diagramas de transición de estados



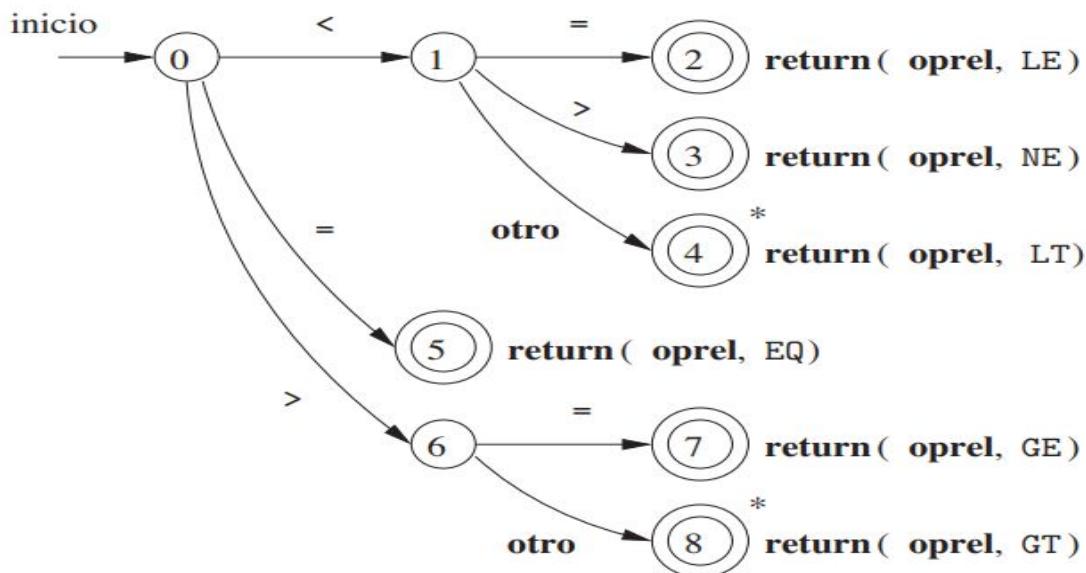
Como paso intermedio en la construcción de un analizador léxico, primero convertimos los patrones en diagramas de flujo estilizados, a los cuales se les llama “diagramas de transición de estados”.

Los diagramas de transición de estados tienen una colección de nodos o círculos, llamados estados. Cada estado representa una condición que podría ocurrir durante el proceso de explorar la entrada, buscando un lexema que coincida con uno de varios patrones. Podemos considerar un estado como un resumen de todo lo que debemos saber acerca de los caracteres que hemos visto entre el apuntador inicioLexema y el apuntador avance.

Las líneas se dirigen de un estado a otro del diagrama de transición de estados. Cada línea se etiqueta mediante un símbolo o conjunto de símbolos. Si nos encontramos en cierto estado  $s$ , y el siguiente símbolo de entrada es  $a$ , buscamos una línea que salga del estado  $s$  y esté etiquetado por  $a$  (y tal vez por otros símbolos también). Si encontramos dicha línea, avanzamos el apuntador avance y entramos al estado del diagrama de transición de estados al que nos lleva esa línea. Asumiremos que todos nuestros diagramas de transición de estados son deterministas, lo que significa que nunca hay más de una línea que sale de un estado dado, con un símbolo dado de entre sus etiquetas.

Algunas convenciones importantes de los diagramas de transición de estados son:

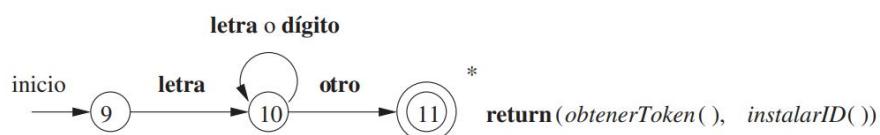
1. Se dice que ciertos estados son de aceptación, o finales. Estos estados indican que se ha encontrado un lexema, aunque el lexema actual tal vez no consista de todas las posiciones entre los apuntadores inicioLexema y avance. Siempre indicamos un estado de aceptación mediante un círculo doble, y si hay que realizar una acción (por lo general, devolver un token y un valor de atributo al analizador sintáctico), la adjuntaremos al estado de aceptación.
2. Además, si es necesario retroceder el apuntador avance una posición (es decir, si el lexema no incluye el símbolo que nos llevó al estado de aceptación), entonces deberemos colocar de manera adicional un \* cerca del estado de aceptación.
3. Un estado se designa como el estado inicial; esto se indica mediante una línea etiquetada como “inicio”, que no proviene de ninguna parte. El diagrama de transición siempre empieza en el estado inicial, antes de leer cualquier símbolo de entrada.



Por otro lado, si en el estado 0 el primer carácter que vemos es =, entonces este carácter debe ser el lexema. De inmediato devolvemos ese hecho desde el estado 5. La posibilidad restante es que el primer carácter sea >. Entonces, debemos pasar al estado 6 y decidir, en base al siguiente carácter, si el lexema es  $\geq$  (si vemos a continuación el signo =), o sólo > (con cualquier otro carácter). Observe que, si en el estado 0 vemos cualquier carácter además de <, = o >, no es posible que estemos viendo un lexema oprel, por lo que no utilizaremos este diagrama de transición de estados.

### Reconocimiento de las palabras reservadas y los identificadores

El reconocimiento de las palabras reservadas y los identificadores presenta un problema. Por lo general, las palabras clave como if o then son reservadas (como en nuestro bosquejo), por lo que no son identificadores, aun cuando lo parecen. Así, aunque por lo general usamos un diagrama de transición de estados, para buscar lexemas de identificadores, este diagrama también reconocerá las palabras clave if, then y else de nuestro bosquejo.

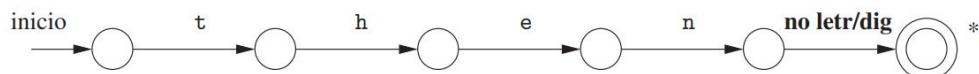


Hay dos formas en las que podemos manejar las palabras reservadas que parecen identificadores:

1. Instalar las palabras reservadas en la tabla de símbolos desde el principio. Un campo de la entrada en la tabla de símbolos indica que estas cadenas nunca serán identificadores ordinarios, y nos dice qué token representan. Al encontrar un identificador, una llamada a instalarID lo coloca en la tabla de símbolos, si no se encuentra ahí todavía, y devuelve un apuntador a la entrada en la tabla de símbolos para

el lexema que se encontró. La función obtenerToken examina la entrada en la tabla de símbolos para el lexema encontrado, y devuelve el nombre de token que la tabla de símbolos indique que representa este lexema; ya sea id o uno de los tokens de palabra clave que se instaló en un principio en la tabla.

2. Crear diagramas de transición de estados separados para cada palabra clave. Si adoptamos este método, entonces debemos dar prioridad a los tokens, para que los tokens de palabra reservada se reconozcan de preferencia en vez de id, cuando el lexema coincide con ambos patrones.



### Finalización del bosquejo

El diagrama de transición de estados para el token numero se muestra en la figura 3.16, y es hasta ahora el diagrama más complejo que hemos visto. Empezando en el estado 12, si vemos un dígito pasamos al estado 13. En ese estado podemos leer cualquier número de dígitos adicionales. No obstante, si vemos algo que no sea un dígito o un punto, hemos visto un número en forma de entero; 123 es un ejemplo. Para manejar ese caso pasamos al estado 20, en donde devolvemos el token numero y un apuntador a una tabla de constantes en donde se introduce el lexema encontrado. Esta mecánica no se muestra en el diagrama, pero es análoga a la forma en la que manejamos los identificadores.

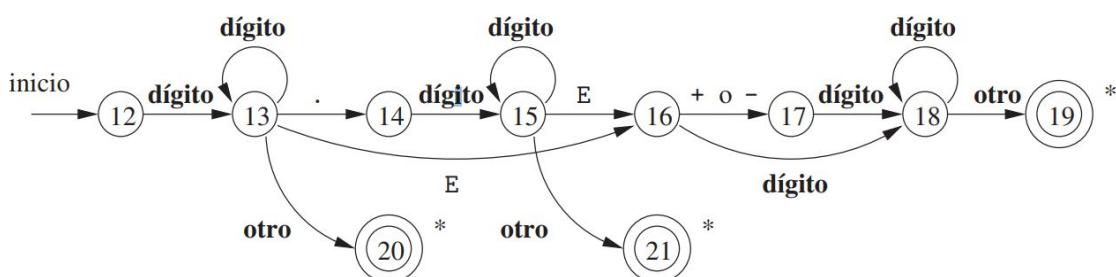


Figura 3.16: Un diagrama de transición para los números sin signo

Si en vez de ello vemos un punto en el estado 13, entonces tenemos una “fracción opcional”. Pasamos al estado 14, y buscamos uno o más dígitos adicionales; el estado 15 se utiliza para este fin. Si vemos una E, entonces tenemos un “exponente opcional”, cuyo reconocimiento es trabajo de los estados 16 a 19. Si en el estado 15 vemos algo que no sea una E o un dígito, entonces hemos llegado al final de la fracción, no hay exponente y devolvemos el lexema encontrado, mediante el estado 21.

El diagrama de transición de estados final, que se muestra en la figura 3.17, es para el espacio en blanco. En ese diagrama buscamos uno o más caracteres de “espacio en blanco”, representados por delim en ese diagrama; por lo general estos caracteres son los espacios, tabuladores, caracteres de nueva línea y tal vez otros caracteres que el diseño del lenguaje no considere como parte de algún token.

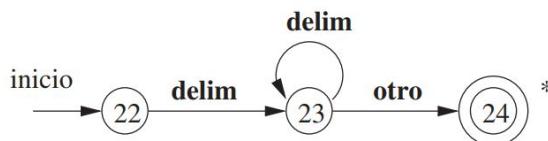


Figura 3.17: Un diagrama de transición para el espacio en blanco

Observe que, en el estado 24, hemos encontrado un bloque de caracteres de espacio en blanco consecutivos, seguidos de un carácter que no es espacio en blanco. Regresemos la entrada para que empiece en el carácter que no es espacio en blanco, pero no regresamos nada al analizador sintáctico, sino que debemos reiniciar el proceso del análisis léxico después del espacio en blanco.

### Arquitectura de un analizador léxico basado en diagramas de transición de estados

Una instrucción switch con base en el valor de estado nos lleva al código para cada uno de los posibles estados, en donde encontramos la acción de ese estado. A menudo, el código para un estado es en sí una instrucción switch o una bifurcación de varias vías que determina el siguiente estado mediante el proceso de leer y examinar el siguiente carácter de entrada.

### El generador de analizadores léxicos Lex

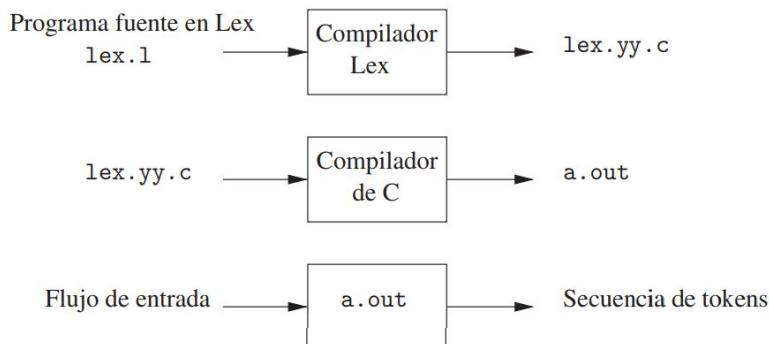
Nos permite especificar un analizador léxico mediante la especificación de expresiones regulares para describir patrones de los tokens. La notación de entrada para la herramienta Lex se conoce como el lenguaje Lex, y la herramienta en sí es el compilador Lex. El compilador Lex transforma los patrones de entrada en un diagrama de transición y genera código, en un archivo llamado lex.yy.c, que simula este diagrama de transición. La mecánica de cómo ocurre esta traducción de expresiones regulares a diagramas de transición es el tema de las siguientes secciones; aquí sólo aprenderemos acerca del lenguaje Lex.

### Uso de Lex

Un archivo de entrada, al que llamaremos lex.l, está escrito en el lenguaje Lex y describe el analizador léxico que se va a generar. El compilador Lex transforma a lex.l en un programa en C, en un archivo que siempre se llama lex.yy.c. El compilador de C compila este archivo en un archivo llamado a.out, como de costumbre. La salida del compilador



de C es un analizador léxico funcional, que puede recibir un flujo de caracteres de entrada y producir una cadena de tokens



### Estructura de los programas en Lex

Un programa en Lex tiene la siguiente forma:

```
declaraciones
%%
reglas de traducción
%%
funciones auxiliares
```

La sección de declaraciones incluye las declaraciones de variables, constantes de manifiesto (identificadores que se declaran para representar a una constante; por ejemplo, el nombre de un token) y definiciones regulares.

Cada una de las reglas de traducción tiene la siguiente forma

:

```
Patrón { Acción }
```

Cada patrón es una expresión regular, la cual puede usar las definiciones regulares de la sección de declaraciones. Las acciones son fragmentos de código, por lo general, escritos en C, aunque se han creado muchas variantes de Lex que utilizan otros lenguajes. La tercera sección contiene las funciones adicionales que se utilizan en las acciones. De manera alternativa, estas funciones pueden compilarse por separado y cargarse con el analizador léxico. El analizador léxico que crea Lex trabaja en conjunto con el analizador sintáctico de la siguiente manera. Cuando el analizador sintáctico llama al analizador léxico, éste empieza a leer el resto de su entrada, un carácter a la vez, hasta que encuentra el prefijo más largo de la entrada que coincide con uno de los patrones  $P_i$ . Después ejecuta la acción asociada  $A_i$ . Por lo general,  $A_i$  regresará al analizador sintáctico, pero si no lo hace (tal vez debido a que  $P_i$  describe espacio en blanco o comentarios), entonces el analizador léxico procede a buscar lexemas adicionales, hasta que una de las acciones correspondientes provoque un retorno al analizador sintáctico. El analizador léxico devuelve un solo valor, el nombre del token, al

analizador sintáctico, pero utiliza la variable entera compartida `yylval` para pasarle información adicional sobre el lexema encontrado, si es necesario.

### Resolución de conflictos en Lex

Nos hemos referido a las dos reglas que utiliza Lex para decidir acerca del lexema apropiado a seleccionar, cuando varios prefijos de la entrada coinciden con uno o más patrones:

1. Preferir siempre un prefijo más largo a uno más corto.
2. Si el prefijo más largo posible coincide con dos o más patrones, preferir el patrón que se lista primero en el programa en Lex.

### El operador adelantado

Lex lee de manera automática un carácter adelante del último carácter que forma el lexema seleccionado, y después regresa la entrada para que sólo se consuma el propio lexema de la entrada. No obstante, algunas veces puede ser conveniente que cierto patrón coincida con la entrada, sólo cuando vaya seguido de ciertos caracteres más. De ser así, tal vez podamos utilizar la barra diagonal en un patrón para indicar el final de la parte del patrón que coincide con el lexema. Lo que va después de / es un patrón adicional que se debe relacionar antes de poder decidir que vimos el token en cuestión, pero que lo que coincide con este segundo patrón no forma parte del lexema.

### Autómatas finitos

En el corazón de la transición se encuentra el formalismo conocido como autómatas finitos. En esencia, estos consisten en gráficos como los diagramas de transición de estados, con algunas diferencias:

1. Los autómatas finitos son reconocedores; sólo dicen "sí" o "no" en relación con cada posible cadena de entrada.
2. Los autómatas finitos pueden ser de dos tipos:
  - (a) Los autómatas finitos no deterministas (AFN) no tienen restricciones en cuanto a las etiquetas de sus líneas. Un símbolo puede etiquetar a varias líneas que surgen del mismo estado, y , la cadena vacía, es una posible etiqueta.
  - (b) Los autómatas finitos deterministas (AFD) tienen, para cada estado, y para cada símbolo de su alfabeto de entrada, exactamente una línea con ese símbolo que sale de ese estado.

Tanto los autómatas finitos deterministas como los no deterministas son capaces de reconocer los mismos lenguajes. De hecho, estos lenguajes son exactamente los mismos lenguajes, conocidos como lenguajes regulares, que pueden describir las expresiones regulares.

### **Autómatas finitos no deterministas**

Un autómata finito no determinista (AFN) consiste en:

1. Un conjunto finito de estados  $S$ .
2. Un conjunto de símbolos de entrada  $\Sigma$ , el alfabeto de entrada. Suponemos que  $\epsilon$ , que representa a la cadena vacía, nunca será miembro de  $\Sigma$ .
3. Una función de transición que proporciona, para cada estado y para cada símbolo en  $\Sigma \cup \{\epsilon\}$ , un conjunto de estados siguientes.
4. Un estado  $s_0$  de  $S$ , que se distingue como el estado inicial.
5. Un conjunto de estados  $F$ , un subconjunto de  $S$ , que se distinguen como los estados aceptantes (o estados finales).

Podemos representar un AFN o AFD mediante un gráfico de transición, en donde los nodos son estados y los flancos Indecidibles representan a la función de transición. Hay un flanco Indecidable  $a$ , que va del estado  $s$  al estado  $t$  si, y sólo si  $t$  es uno de los estados siguientes para el estado  $s$  y la entrada  $a$ . Este gráfico es muy parecido a un diagrama de transición, excepto que:

- a) El mismo símbolo puede etiquetar flancos de un estado hacia varios estados distintos.
- b) Un flanco puede etiquetarse por  $\epsilon$ , la cadena vacía, en vez de, o además de, los símbolos del alfabeto de entrada.

### **Tablas de transición**

También podemos representar a un AFN mediante una tabla de transición, cuyas filas corresponden a los estados, y cuyas columnas corresponden a los símbolos de entrada y a  $\epsilon$ . La entrada para un estado dado y la entrada es el valor de la función de transición que se aplica a esos argumentos. Si la función de transición no tiene información acerca de ese par estado-entrada, colocamos  $\emptyset$  en la tabla para ese estado.

### **Aceptación de las cadenas de entrada mediante los autómatas**

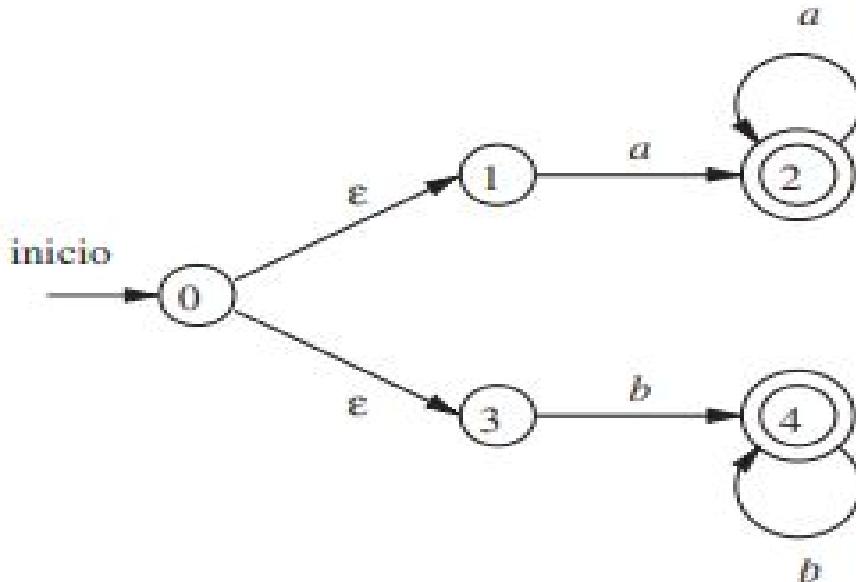
Un AFN acepta la cadena de entrada  $x$  si, y sólo si hay algún camino en el grafo de transición, desde el estado inicial hasta uno de los estados de aceptación, de forma que los



símbolos a lo largo del camino deletreen a x. Observe que las etiquetas  $\epsilon$  a lo largo del camino se ignoran, ya que la cadena vacía no contribuye a la cadena que se construye a lo largo del camino.

### Autómatas finitos deterministas

Un autómata finito determinista (AFD) es un caso especial de un AFN, en donde:



1. No hay movimientos en la entrada  $\epsilon$ .
2. Para cada estado s y cada símbolo de entrada a, hay exactamente una línea que surge de s. Indecidible como a.

Si utilizamos una tabla de transición para representar a un AFD, entonces cada entrada es un solo estado. Por ende, podemos representar a este estado sin las llaves que usamos para formar los conjuntos.

Mientras que el AFN es una representación abstracta de un algoritmo para reconocer las cadenas de cierto lenguaje, el AFD es un algoritmo simple y concreto para reconocer cadenas. Sin duda es afortunado que cada expresión regular y cada AFN puedan convertirse en un AFD que acepte el mismo lenguaje, ya que es el AFD el que en realidad implementamos o simulamos al construir analizadores léxicos. El siguiente algoritmo muestra cómo aplicar un AFD a una cadena.

### Conversión de un AFN a AFD

La idea general de la construcción de subconjuntos es que cada estado del AFD construido corresponde a un conjunto de estados del AFN. Después de leer la entrada  $a_1a_2 \dots a_n$ , el AFD se encuentra en el estado que corresponde al conjunto de estados que el AFN puede alcanzar, desde su estado inicial, siguiendo los caminos etiquetados como  $a_1a_2 \dots a_n$ . Es posible que el número de estados del AFD sea exponencial en el número de estados del AFN, lo cual podría provocar dificultades al tratar de implementar este AFD. No obstante,



parte del poder del método basado en autómatas para el análisis léxico es que para los lenguajes reales, el AFN y el AFD tienen aproximadamente el mismo número de estados, y no se ve el comportamiento exponencial.

Algoritmo 3.20: La construcción de subconjuntos de un AFD, a partir de un AFN.

ENTRADA: Un AFN N.

SALIDA: Un AFD D que acepta el mismo lenguaje que N.

MÉTODO: Nuestro algoritmo construye una tabla de transición Dtran para D. Cada estado de D es un conjunto de estados del AFN, y construimos Dtran para que D pueda simular “en paralelo” todos los posibles movimientos que N puede realizar sobre una cadena de entrada dada. Nuestro primer problema es manejar las transiciones de N en forma apropiada. En la figura 3.31 vemos las definiciones de varias funciones que describen cálculos básicos en los estados de N que son necesarios en el algoritmo. Observe que s es un estado individual de N, mientras que T es un conjunto de estados de N.

OPERACIÓN	DESCRIPCIÓN
$\epsilon$ -cerradura(s)	Conjunto de estados del AFN a los que se puede llegar desde el estado s del AFN, sólo en las transiciones $\epsilon$ .
$\epsilon$ -cerradura(T)	Conjunto de estados del AFN a los que se puede llegar desde cierto estado s del AFN en el conjunto T, sólo en las transiciones $\epsilon$ ; = $\bigcup_{s \in T} \epsilon\text{-cerradura}(s)$ .
mover(T, a)	Conjunto de estados del AFN para los cuales hay una transición sobre el símbolo de entrada a, a partir de cierto estado s en T.

Figura 3.31: Operaciones sobre los estados del AFN

Debemos explorar esos conjuntos de estados en los que puede estar N después de ver cierta cadena de entrada. Como base, antes de leer el primer símbolo de entrada, N puede estar en cualquiera de los estados de  $\epsilon$ -cerradura(s0), en donde s0 es su estado inicial. Para la inducción, suponga que N puede estar en el conjunto de estados T después de leer la cadena de entrada x. Si a continuación lee la entrada a, entonces N puede pasar de inmediato a cualquiera de los estados en mover(T, a). No obstante, después de leer a también podría realizar varias transiciones ; por lo tanto, N podría estar en cualquier estado de  $\epsilon$ -cerradura(mover(T, a)) después de leer la entrada xa. Siguiendo estas ideas, la construcción del conjunto de estados de D, Destados, y su función de transición Dtran.

### La estructura del analizador generado

La figura 3.49 presenta las generalidades acerca de la arquitectura de un analizador léxico generado por Lex. El programa que sirve como analizador léxico incluye un programa fijo que simula a un autómata; en este punto dejamos abierta la decisión de si el autómata es determinista o no. El resto del analizador léxico consiste en componentes que se crean a partir del programa Lex, por el mismo Lex.

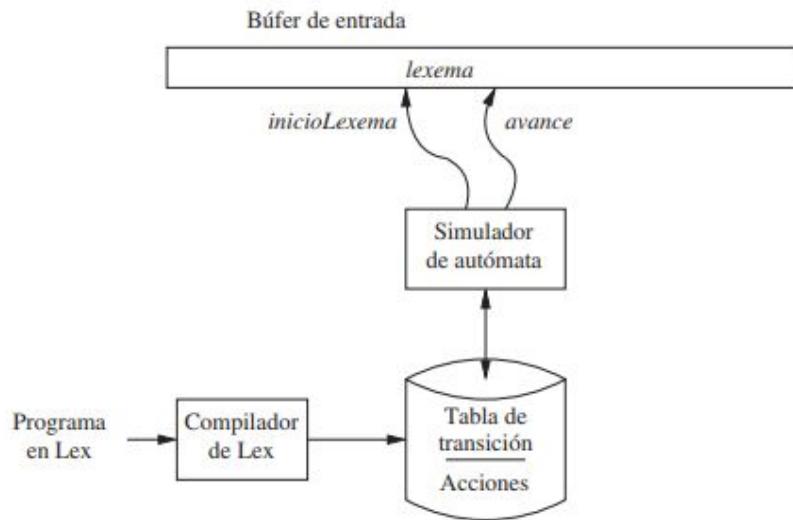


Figura 3.49: Un programa en Lex se convierte en una tabla de transición y en acciones, para que las utilice un simulador de autómatas finitos

### Coincidencia de patrones con base en los AFNs

Si el analizador léxico simula un AFN como el de la figura 3.52, entonces debe leer la entrada que empieza en el punto de su entrada, al cual nos hemos referido como *inicioLexema*. A medida que el apuntador llamado *avance* avanza hacia delante en la entrada, calcula el conjunto de estados en los que se encuentra en cada punto.

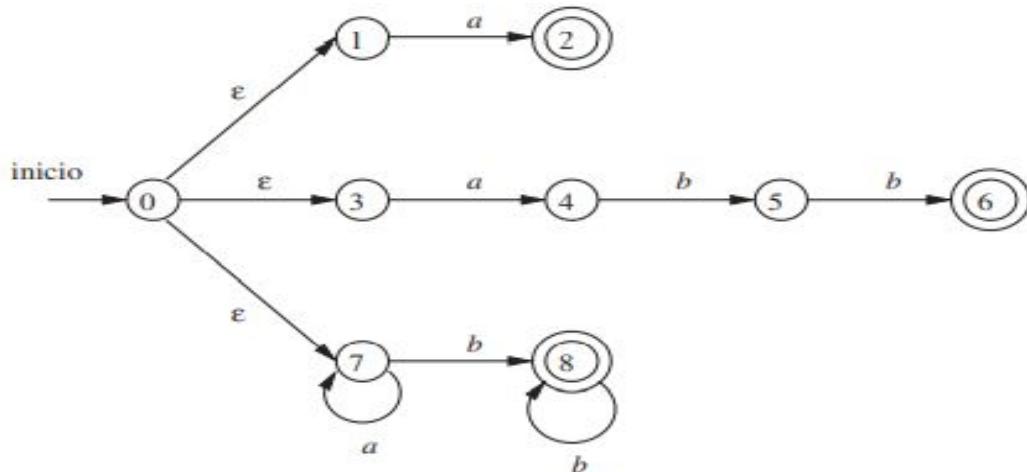


Figura 3.52: AFN combinado

En algún momento, la simulación del AFN llega a un punto en la entrada en donde no hay siguientes estados. En ese punto, no hay esperanza de que cualquier prefijo más largo de la entrada haga que el AFN llegue a un estado de aceptación; en vez de ello, el conjunto de estados siempre estará vacío. Por ende, estamos listos para decidir sobre el prefijo más largo que sea un lexema que coincide con cierto patrón.



Buscamos hacia atrás en la secuencia de conjuntos de estados, hasta encontrar un conjunto que incluya uno o más estados de aceptación. Si hay varios estados de aceptación en ese conjunto, elegimos el que esté asociado con el primer patrón  $\pi$  en la lista del programa en Lex. Retrocedemos el apuntador avance hacia el final del lexema, y realizamos la acción  $A_i$  asociada con el patrón  $\pi$ .

### **AFDs para analizadores léxicos**

Dentro de cada estado del AFD, si hay uno o más estados aceptantes del AFN, se determina el primer patrón cuyo estado aceptante se representa, y ese patrón se convierte en la salida del estado AFD.

### **Implementación del operador de preanálisis**

Al convertir el patrón  $r_1/r_2$  en un AFN, tratamos al / como si fuera , por lo que en realidad no buscamos un / en la entrada. No obstante, si el AFN reconoce un prefijo  $xy$  del búfer de entrada, de forma que coincide con esta expresión regular, el final del lexema no es en donde el AFN entró a su estado de aceptación. En vez de ello, el final ocurre cuando el AFN entra a un estado  $s$  tal que:

1.  $s$  tenga una transición en el / imaginario.
2. Hay un camino del estado inicial del AFN hasta el estado  $s$ , que deletrea a  $x$ .
3. Hay un camino del estado  $s$  al estado de aceptación que deletrea a  $y$ .
4.  $x$  es lo más largo posible para cualquier  $xy$  que cumpla con las condiciones 1-3.

Si sólo hay un estado de transición en el / imaginario en el AFN, entonces el final del lexema ocurre cuando se entra a este estado por última vez, como se ilustra en el siguiente ejemplo. Si el AFN tiene más de un estado de transición en el / imaginario, entonces el problema general de encontrar el estado  $s$  actual se dificulta mucho más.

### **Optimización de los buscadores por concordancia de patrones basados en AFD**

En esa sección presentaremos tres algoritmos que se utilizan para implementar y optimizar buscadores por concordancia de patrones, construidos a partir de expresiones regulares.

1. El primer algoritmo es útil en un compilador de Lex, ya que construye un AFD directamente a partir de una expresión regular, sin construir un AFN intermedio. Además, el AFD resultante puede tener menos estados que el AFD que se construye mediante un AFN.
2. El segundo algoritmo disminuye al mínimo el número de estados de cualquier AFD, mediante la combinación de los estados que tienen el mismo comportamiento a futuro.



El algoritmo en sí es bastante eficiente, pues se ejecuta en un tiempo  $O(n \log n)$ , en donde  $n$  es el número de estados del AFD.

3. El tercer algoritmo produce representaciones más compactas de las tablas de transición que la tabla estándar bidimensional.

### Estados significativos de un AFN

Durante la construcción de subconjuntos, pueden identificarse dos conjuntos de estados del AFN (que se tratan como si fueran el mismo conjunto) si:

1. Tienen los mismos estados significativos.
2. Ya sea que ambos tengan estados de aceptación, o ninguno.

El AFN construido sólo tiene un estado de aceptación, pero éste, que no tiene transiciones de salida, no es un estado significativo. Al concatenar un único marcador final # derecho con una expresión regular  $r$ , proporcionamos al estado de aceptación para  $r$  una transición sobre #, con lo cual lo marcamos como un estado significativo del AFN para  $(r)\#$ . En otras palabras, al usar la expresión regular aumentada  $(r)\#$ , podemos olvidarnos de los estados de aceptación a medida que procede la construcción de subconjuntos; cuando se completa la construcción, cualquier estado con una transición sobre # debe ser un estado de aceptación.

Los estados significativos del AFN corresponden directamente a las posiciones en la expresión regular que contienen símbolos del alfabeto. Como pronto veremos, es conveniente presentar la expresión regular mediante su árbol sintáctico, en donde las hojas corresponden a los operandos y los nodos interiores corresponden a los operadores. A un nodo interior se le llama nodo-concat, nodo-o o nodo-asterisco si se etiqueta mediante el operador de concatenación (punto), el operador de unión |, o el operador \*, respectivamente.

### Funciones calculadas a partir del árbol sintáctico

Para construir un AFD directamente a partir de una expresión regular, construimos su árbol sintáctico y después calculamos cuatro funciones: anulable, primerapos, ultimapos y siguientepos, las cuales se definen a continuación. Cada definición se refiere al árbol sintáctico para una expresión regular aumentada  $(r)\#$  específica.

1. anulable( $n$ ) es verdadera para un nodo  $n$  del árbol sintáctico si, y sólo si, la subexpresión representada por  $n$  tiene a en su lenguaje. Es decir, la subexpresión puede "hacerse nula" o puede ser la cadena vacía, aun cuando pueda representar también a otras cadenas.

2.  $\text{primerapos}(n)$  es el conjunto de posiciones en el subárbol con raíz en  $n$ , que corresponde al primer símbolo de por lo menos una cadena en el lenguaje de la subexpresión con raíz en  $n$ .

3.  $\text{ultimapos}(n)$  es el conjunto de posiciones en el subárbol con raíz en  $n$ , que corresponde al último símbolo de por lo menos una cadena en el lenguaje de la subexpresión con raíz en  $n$ .

4.  $\text{siguientenpos}(p)$ , para una posición  $p$ , es el conjunto de posiciones  $q$  en todo el árbol sintáctico, de tal forma que haya cierta cadena  $x = a_1a_2 \dots a_n$  en una  $L((r)\#)$  tal que para cierta  $i$ , haya una forma de explicar la membresía de  $x$  en  $L((r)\#)$ , haciendo que  $a_i$  coincida con la posición  $p$  del árbol sintáctico y  $a_{i+1}$  con la posición  $q$

### Cálculo de anulable, primerapos y ultimapos

Por último, necesitamos ver cómo calcular  $\text{siguientenpos}$ . Sólo hay dos formas en que podemos hacer que la posición de una expresión regular siga a otra.

1. Si  $n$  es un nodo-concat con el hijo izquierdo  $c_1$  y con el hijo derecho  $c_2$ , entonces para cada posición  $i$  en  $\text{ultimapos}(c_1)$ , todas las posiciones en  $\text{primerapos}(c_2)$  se encuentran en  $\text{siguientenpos}(i)$ .

2. Si  $n$  es un nodo-asterisco  $e$   $i$  es una posición en  $\text{ultimapos}(n)$ , entonces todas las posiciones en  $\text{primerapos}(n)$  se encuentran en  $\text{siguientenpos}(i)$ .

### Cálculo de siguientepos

Por último, necesitamos ver cómo calcular  $\text{siguientepos}$ . Sólo hay dos formas en que podemos hacer que la posición de una expresión regular siga a otra. 1. Si  $n$  es un nodo-concat con el hijo izquierdo  $c_1$  y con el hijo derecho  $c_2$ , entonces para cada posición  $i$  en  $\text{ultimapos}(c_1)$ , todas las posiciones en  $\text{primerapos}(c_2)$  se encuentran en  $\text{siguientepos}(i)$ . 2. Si  $n$  es un nodo-asterisco  $e$   $i$  es una posición en  $\text{ultimapos}(n)$ , entonces todas las posiciones en  $\text{primerapos}(n)$  se encuentran en  $\text{siguientepos}(i)$ .

### Conversión directa de una expresión regular a un AFD

Algoritmo 3.36: Construcción de un AFD a partir de una expresión regular  $r$ .

ENTRADA: Una expresión regular  $r$ .

SALIDA: Un AFD  $D$  que reconoce a  $L(r)$ .

MÉTODO:



1. Construir un árbol sintáctico T a partir de la expresión regular aumentada  $(r)^*$ .
2. Calcular anulable, primerapos, ultimapos y siguientepos para T, mediante los métodos de las secciones 3.9.3 y 3.9.4.
3. Construir Destados, el conjunto de estados del AFD D, y Dtran, la función de transición para D, mediante el procedimiento de la figura 3.62. Los estados de D son estados de posiciones en T. Al principio, cada estado está “sin marca”, y un estado se “marca” justo antes de que consideremos sus transiciones de salida. El estado inicial de D es primerapos( $n_0$ ), en donde el nodo  $n_0$  es la raíz de T. Los estados de aceptación son los que contienen la posición para el símbolo de marcador final #.

### **Intercambio de tiempo por espacio en la simulación de un AFD**

La manera más simple y rápida de representar la función de transición de un AFD es una tabla bidimensional indexada por estados y caracteres. Dado un estado y el siguiente carácter de entrada, accedemos al arreglo para encontrar el siguiente estado y cualquier acción especial que debemos tomar; por ejemplo, devolver un token al analizador sintáctico. Como un analizador léxico ordinario tiene varios cientos de estados en su AFD e involucra al alfabeto ASCII de 128 caracteres de entrada, el arreglo consume menos de un megabyte.

No obstante, los compiladores también aparecen en dispositivos muy pequeños, en donde hasta un megabyte de memoria podría ser demasiado. Para tales situaciones, existen muchos métodos que podemos usar para compactar la tabla de transición. Por ejemplo, podemos representar cada estado mediante una lista de transiciones (es decir, pares carácter-estado) que se terminen mediante un estado predeterminado, el cual debe elegirse para cualquier carácter de entrada que no se encuentre en la lista. Si elegimos como predeterminado el siguiente estado que ocurra con más frecuencia, a menudo podemos reducir la cantidad de almacenamiento necesario por un factor extenso.

Hay una estructura de datos más sutil que nos permite combinar la velocidad del acceso a los arreglos con la compresión de listas con valores predeterminados. Podemos considerar esta estructura como cuatro arreglos. El arreglo base se utiliza para determinar la ubicación base de las entradas para el estado s, que se encuentran en los arreglos siguiente y comprobacion. El arreglo predeterminado se utiliza para determinar una ubicación base alternativa, si el arreglo comprobacion nos indica que el que proporciona base[s] es inválido.

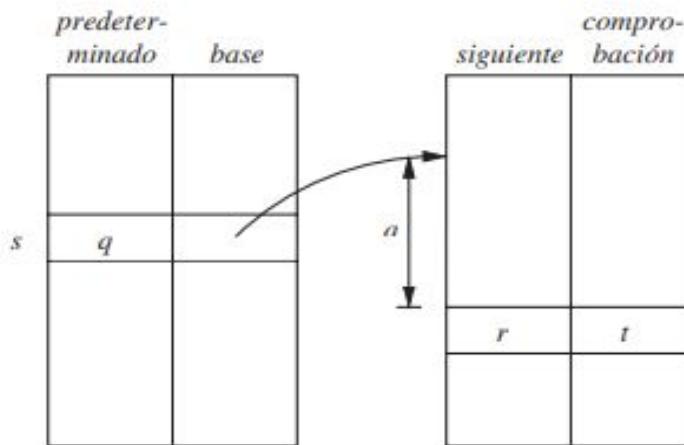


Figura 3.66: Estructura de datos para representar tablas de transición

Para calcular siguienteEstado( $s, a$ ), la transición para el estado  $s$  con la entrada  $a$ , examinamos las entradas siguiente y comprobacion en la ubicación  $l = base[s]+a$ , en donde el carácter  $a$  se trata como entero, supuestamente en el rango de 0 a 127. Si  $comprobacion[l] = s$ , entonces esta entrada es válida y el siguiente estado para el estado  $s$  con la entrada  $a$  es  $siguiente[l]$ . Si  $comprobacion[l] \neq s$ , entonces determinamos otro estado  $t = predeterminado[s]$  y repetimos el proceso, como si  $t$  fuera el estado actual. De manera más formal, la función siguienteEstado se define así:

```
int siguienteEstado(s, a) {
    if ( comprobacion[base[s]+a] == s ) return siguiente[base[s] + a];
    else return siguienteEstado(predeterminado[s], a);
}
```

### (“Análisis Sintáctico (parte 1)”)

De hecho, el árbol de análisis sintáctico no necesita construirse en forma explícita, ya que las acciones de comprobación y traducción pueden intercalarse con el análisis sintáctico, como veremos más adelante. Por ende, el analizador sintáctico y el resto de la interfaz de usuario podrían implementarse sin problemas mediante un solo módulo.

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



A partir de ciertas clases de gramáticas, podemos construir de manera automática un analizador sintáctico eficiente que determine la estructura sintáctica de un programa fuente. Como beneficio colateral, el proceso de construcción del analizador sintáctico puede revelar ambigüedades sintácticas y puntos problemáticos que podrían haberse pasado por alto durante la fase inicial del diseño del lenguaje.

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.

Existen tres tipos generales de analizadores para las gramáticas: universales, descendentes y ascendentes. Los métodos universales de análisis sintáctico como el algoritmo de Cocke-Younger-Kasami y el algoritmo de Earley pueden analizar cualquier gramática (vea las notas bibliográficas). Sin embargo, estos métodos generales son demasiado ineficientes como para usarse en la producción de compiladores.

Los métodos que se utilizan, por lo regular, en los compiladores pueden clasificarse como descendentes o ascendentes. Los métodos descendentes y ascendentes más eficientes sólo funcionan para subclases de gramáticas, pero varias de estas clases, en especial las gramáticas LL y LR, son lo bastante expresivas como para describir la mayoría de las construcciones sintácticas en los lenguajes de programación modernos. Los errores léxicos incluyen la escritura incorrecta de los identificadores, las palabras clave o los operadores; por ejemplo, el uso de un identificador `tamanioElipce` en vez de `tamanioElipse`, y la omisión de comillas alrededor del texto que se debe interpretar como una cadena.

Los errores sintácticos incluyen la colocación incorrecta de los signos de punto y coma, además de llaves adicionales o faltantes; es decir, “{” o “}”. Como otro ejemplo, en C o Java, la aparición de una instrucción `case` sin una instrucción `switch` que la encierre es un error sintáctico (sin embargo, por lo general, esta situación la acepta el analizador sintáctico y se atrapa más adelante en el procesamiento, cuando el compilador intenta generar código).

Otra de las razones para enfatizar la recuperación de los errores durante el análisis sintáctico es que muchos errores parecen ser sintácticos, sea cual fuere su causa, y se exponen cuando el análisis sintáctico no puede continuar. Algunos errores semánticos, como los conflictos entre los tipos, también pueden detectarse con eficiencia; sin



embargo, la detección precisa de los errores semánticos y lógicos en tiempo de compilación es, por lo general, una tarea difícil.

- Reportar la presencia de errores con claridad y precisión.
- Recuperarse de cada error lo bastante rápido como para poder detectar los errores siguientes.
- Agregar una sobrecarga mínima al procesamiento de los programas correctos.

### Representación de gramáticas

Las construcciones que empiezan con palabras clave como while o int son muy fáciles de analizar, ya que la palabra clave guía la elección de la producción gramatical que debe aplicarse para hacer que coincida con la entrada. Por lo tanto, nos concentraremos en las expresiones, que representan un reto debido a la asociatividad y la precedencia de operadores. La asociatividad y la precedencia se resuelvan en la siguiente gramática, que es similar a las que utilizamos en el capítulo 2 para describir expresiones, términos y factores. E representa a las expresiones que consisten en términos separados por los signos +, T representa a los términos que consisten en factores separados por los signos \*, y F representa a los factores que pueden ser expresiones entre paréntesis o identificadores:

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & ( E ) \mid \text{id} \end{array}$$

### Manejo de los errores sintácticos

El resto de esta sección considera la naturaleza de los errores sintácticos y las estrategias generales para recuperarse de ellos. Dos de estas estrategias, conocidas como recuperaciones en modo de pánico y a nivel de frase, se describirán con más detalle junto con los métodos específicos de análisis sintáctico. Si un compilador tuviera que procesar sólo programas correctos, su diseño e implementación se simplificaría en forma considerable. No obstante, se espera que un compilador ayude al programador a localizar y rastrear los errores que, de manera inevitable, se infiltran en los programas, a pesar de los mejores esfuerzos del programador. Aunque parezca increíble, son pocos los lenguajes que se diseñan teniendo en mente el manejo de errores, aun cuando éstos son tan comunes. Nuestra civilización sería radicalmente distinta si los lenguajes hablados tuvieran los mismos requerimientos en cuanto a precisión sintáctica que los lenguajes de computadora. La mayoría de las especificaciones de los lenguajes de programación no describen la forma en que un compilador debe responder a los errores; el manejo de los mismos es responsabilidad del diseñador del compilador. La planeación del manejo de los errores desde el principio

puede simplificar la estructura de un compilador y mejorar su capacidad para manejar los errores. Los errores de programación comunes pueden ocurrir en muchos niveles distintos.

- Los errores léxicos incluyen la escritura incorrecta de los identificadores, las palabras clave o los operadores; por ejemplo, el uso de un identificador tamanioElipce en vez de tamanioElipse, y la omisión de comillas alrededor del texto que se debe interpretar como una cadena.
- Los errores sintácticos incluyen la colocación incorrecta de los signos de punto y coma, además de llaves adicionales o faltantes; es decir, “{” o “}”. Como otro ejemplo, en C o Java, la aparición de una instrucción case sin una instrucción switch que la encierre es un error sintáctico (sin embargo, por lo general, esta situación la acepta el analizador sintáctico y se atrapa más adelante en el procesamiento, cuando el compilador intenta generar código).
- Los errores semánticos incluyen los conflictos de tipos entre los operadores y los operandos. Un ejemplo es una instrucción return en un método de Java, con el tipo de resultado void.
- Los errores lógicos pueden ser cualquier cosa, desde un razonamiento incorrecto del programador en el uso (en un programa en C) del operador de asignación =, en vez del operador de comparación ==. El programa que contenga = puede estar bien formado; sin embargo, tal vez no refleje la intención del programador.

La precisión de los métodos de análisis sintáctico permite detectar los errores sintácticos con mucha eficiencia. Varios métodos de análisis sintáctico, como los métodos LL y LR, detectan un error lo más pronto posible; es decir, cuando el flujo de tokens que proviene del analizador léxico no puede seguirse analizando de acuerdo con la gramática para el lenguaje. Dicho en forma más precisa, tienen la propiedad de prefijo viable, lo cual significa que detectan la ocurrencia de un error tan pronto como ven un prefijo de la entrada que no puede completarse para formar una cadena válida en el lenguaje. Otra de las razones para enfatizar la recuperación de los errores durante el análisis sintáctico es que muchos errores parecen ser sintácticos, sea cual fuere su causa, y se exponen cuando el análisis sintáctico no puede continuar. Algunos errores semánticos, como los conflictos entre los tipos, también pueden detectarse con eficiencia; sin embargo, la detección precisa de los errores semánticos y lógicos en tiempo de compilación es, por lo general, una tarea difícil. El mango de errores en un analizador sintáctico tiene objetivos que son simples de declarar, pero difíciles de llevar a cabo:

- Reportar la presencia de errores con claridad y precisión.
- Recuperarse de cada error lo bastante rápido como para poder detectar los errores siguientes.
- Agregar una sobrecarga mínima al procesamiento de los programas correctos.

Por fortuna, los errores comunes son simples, y a menudo basta con un mecanismo simple para su manejo. ¿De qué manera un mango de errores debe reportar la presencia de un error? Como mínimo, debe reportar el lugar en el programa fuente en donde se detectó un error, ya que hay una buena probabilidad de que éste en sí haya ocurrido en uno de los pocos tokens anteriores. Una estrategia común es imprimir la línea del problema con un apuntador a la posición en la que se detectó el error.

### **Estrategias para recuperarse de los errores**

Una vez que se detecta un error, ¿cómo debe recuperarse el analizador sintáctico? Aunque no hay una estrategia que haya demostrado ser aceptable en forma universal, algunos métodos pueden aplicarse en muchas situaciones. El método más simple es que el analizador sintáctico termine con un mensaje de error informativo cuando detecte el primer error. A menudo se descubren errores adicionales si el analizador sintáctico puede restaurarse a sí mismo, a un estado en el que pueda continuar el procesamiento de la entrada, con esperanzas razonables de que un mayor procesamiento proporcione información útil para el diagnóstico. Si los errores se apilan, es mejor para el compilador desistir después de exceder cierto límite de errores, que producir una molesta avalancha de errores “falsos”. El resto de esta sección se dedica a las siguientes estrategias de recuperación de los errores: modo de pánico, nivel de frase, producciones de errores y corrección global.

### **Recuperación en modo de pánico**

Con este método, al describir un error el analizador sintáctico descarta los símbolos de entrada, uno a la vez, hasta encontrar un conjunto designado de tokens de sincronización. Por lo general, los tokens de sincronización son delimitadores como el punto y coma o }, cuya función en el programa fuente es clara y sin ambigüedades. El diseñador del compilador debe seleccionar los tokens de sincronización apropiados para el lenguaje fuente. Aunque la corrección en modo de pánico a menudo omite una cantidad considerable de entrada sin verificar errores adicionales, tiene la ventaja de ser simple y, a diferencia de ciertos métodos que consideraremos más adelante, se garantiza que no entrará en un ciclo infinito.

### **Recuperación a nivel de frase**

Al descubrir un error, un analizador sintáctico puede realizar una corrección local sobre la entrada restante; es decir, puede sustituir un prefijo de la entrada restante por alguna cadena que le permita continuar. Una corrección local común es sustituir una coma por un punto y coma, eliminar un punto y coma extraño o insertar un punto y coma faltante. La elección de la corrección local se deja al diseñador del compilador. Desde luego que debemos tener cuidado de elegir sustituciones que no nos lleven hacia ciclos infinitos, como sería, por ejemplo, si siempre insertáramos algo en la entrada adelante del símbolo de entrada actual. La sustitución a nivel de frase se ha utilizado en varios



compiladores que reparan los errores, ya que puede corregir cualquier cadena de entrada. Su desventaja principal es la dificultad que tiene para arreglárselas con situaciones en las que el error actual ocurre antes del punto de detección.

### Producciones de errores

Al anticipar los errores comunes que podríamos encontrar, podemos aumentar la gramática para el lenguaje, con producciones que generen las construcciones erróneas. Un analizador sintáctico construido a partir de una gramática aumentada por estas producciones de errores detecta los errores anticipados cuando se utiliza una producción de error durante el análisis sintáctico. Así, el analizador sintáctico puede generar diagnósticos de error apropiados sobre la construcción errónea que se haya reconocido en la entrada.

### Corrección global

Lo ideal sería que un compilador hiciera la menor cantidad de cambios en el procesamiento de una cadena de entrada incorrecta. Hay algoritmos para elegir una secuencia mínima de cambios, para obtener una corrección con el menor costo a nivel global. Dada una cadena de entrada incorrecta  $x$  y una gramática  $G$ , estos algoritmos buscarán un árbol de análisis sintáctico para una cadena y relacionada, de tal forma que el número de inserciones, eliminaciones y modificaciones de los tokens requeridos para transformar  $x$  en  $y$  sea lo más pequeño posible. Por desgracia, estos métodos son en general demasiado costosos para implementarlos en términos de tiempo y espacio, por lo cual estas técnicas sólo son de interés teórico en estos momentos. Hay que observar que un programa casi correcto tal vez no sea lo que el programador tenía en mente. Sin embargo, la noción de la corrección con el menor costo proporciona una norma para evaluar las técnicas de recuperación de los errores, la cual se ha utilizado para buscar cadenas de sustitución óptimas para la recuperación a nivel de frase.

### Gramáticas libres de contexto

Si utilizamos una variable sintáctica  $instr$  para denotar las instrucciones, y una variable  $expr$  para denotar las expresiones, la siguiente producción:

$$instr \rightarrow if ( expr ) instr \text{ else } instr$$

especifica la estructura de esta forma de instrucción condicional. Entonces, otras producciones definen con precisión lo que es una  $expr$  y qué más puede ser una  $instr$ . En esta sección repasaremos la definición de una gramática libre de contexto y presentaremos la terminología para hablar acerca del análisis sintáctico. En especial, la noción de derivaciones es muy útil para hablar sobre el orden en el que se aplican las producciones durante el análisis sintáctico.

## La definición formal de una gramática libre de contexto

1. Los terminales son los símbolos básicos a partir de los cuales se forman las cadenas. El término “nombre de token” es un sinónimo de “terminal”; con frecuencia usaremos la palabra “token” en vez de terminal, cuando esté claro que estamos hablando sólo sobre el nombre del token. Asumimos que las terminales son los primeros componentes de los tokens que produce el analizador léxico. Los terminales son las palabras reservadas if y else, y los símbolos (“” y ””).
2. Los no terminales son variables sintácticas que denotan conjuntos de cadenas. Instr y expr son no terminales. Los conjuntos de cadenas denotados por los no terminales ayudan a definir el lenguaje generado por la gramática. Los no terminales imponen una estructura jerárquica sobre el lenguaje, que representa la clave para el análisis sintáctico y la traducción.
3. En una gramática, un no terminal se distingue como el símbolo inicial, y el conjunto de cadenas que denota es el lenguaje generado por la gramática. Por convención, las producciones para el símbolo inicial se listan primero.
4. Las producciones de una gramática especifican la forma en que pueden combinarse los terminales y los no terminales para formar cadenas. Cada producción consiste en:  
(a) Un no terminal, conocido como encabezado o lado izquierdo de la producción; esta producción define algunas de las cadenas denotadas por el encabezado. (b) El símbolo →. Algunas veces se ha utilizado ::= en vez de la flecha. (c) Un cuerpo o lado derecho, que consiste en cero o más terminales y no terminales. Los componentes del cuerpo describen una forma en que pueden construirse las cadenas del no terminal en el encabezado.

## Convenciones de notación

Para evitar siempre tener que decir que “éstos son los terminales”, “éstos son los no terminales”, etcétera, utilizaremos las siguientes convenciones de notación para las gramáticas durante el resto de este libro:

1. Estos símbolos son terminales:
  - (a) Las primeras letras minúsculas del alfabeto, como a, b, c.
  - (b) Los símbolos de operadores como +, \*, etcétera.
  - (c) Los símbolos de puntuación como paréntesis, coma, etcétera.
  - (d) Los dígitos 0, 1, ..., 9. (e) Las cadenas en negrita como id o if, cada una de las cuales representa un solo símbolo terminal.
2. Estos símbolos son no terminales:



- (a) Las primeras letras mayúsculas del alfabeto, como A, B, C.
- (b) La letra S que, al aparecer es, por lo general, el símbolo inicial.
- (c) Los nombres en cursiva y minúsculas, como expr o instr.
- (d) Al hablar sobre las construcciones de programación, las letras mayúsculas pueden utilizarse para representar no terminales. Por ejemplo, los no terminales para las expresiones, los términos y los factores se representan a menudo mediante E, T y F, respectivamente.
3. Las últimas letras mayúsculas del alfabeto, como X, Y, Z, representan símbolos gramaticales; es decir, pueden ser no terminales o terminales.
  4. Las últimas letras minúsculas del alfabeto, como u, v, ..., z, representan cadenas de terminales (posiblemente vacías).
  5. Las letras griegas minúsculas  $\alpha$ ,  $\beta$ ,  $\gamma$ , por ejemplo, representan cadenas (posiblemente vacías) de símbolos gramaticales. Por ende, una producción genérica puede escribirse como  $A \rightarrow \alpha$ , en donde A es el encabezado y  $\alpha$  el cuerpo.
  6. Un conjunto de producciones  $A \rightarrow \alpha_1$ ,  $A \rightarrow \alpha_2$ , ...,  $A \rightarrow \alpha_k$  con un encabezado común A (las llamaremos producciones A), puede escribirse como  $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_k$ . A  $\alpha_1$ ,  $\alpha_2$ , ...,  $\alpha_k$  les llamamos las alternativas para A.
  7. A menos que se indique lo contrario, el encabezado de la primera producción es el símbolo inicial.

## Derivaciones

La construcción de un árbol de análisis sintáctico puede hacerse precisa si tomamos una vista derivacional, en la cual las producciones se tratan como reglas de rescritura. Empezando con el símbolo inicial, cada paso de rescritura sustituye a un no terminal por el cuerpo de una de sus producciones. Esta vista derivacional corresponde a la construcción descendente de un árbol de análisis sintáctico, pero la precisión que ofrecen las derivaciones será muy útil cuando hablemos del análisis sintáctico ascendente. Como veremos, el análisis sintáctico ascendente se relaciona con una clase de derivaciones conocidas como derivaciones de “más a la derecha”, en donde el no terminal por la derecha se reescribe en cada paso.

Por ejemplo, considere la siguiente gramática, con un solo no terminal E, la cual agrega una producción  $E \rightarrow -E$  a la gramática:

$$E \rightarrow E + E \mid E * E \mid - E \mid ( E ) \mid id$$



La producción  $E \rightarrow -E$  significa que si  $E$  denota una expresión, entonces  $-E$  debe también denotar una expresión. La sustitución de una sola  $E$  por  $-E$  se describirá escribiendo lo siguiente:

$$E \Rightarrow -E$$

lo cual se lee como “ $E$  deriva a  $-E$ ”. La producción  $E \rightarrow (E)$  puede aplicarse para sustituir cualquier instancia de  $E$  en cualquier cadena de símbolos gramaticales por  $(E)$ ; por ejemplo,  $E * E \Rightarrow (E) * E$  o  $E * E \Rightarrow E * (E)$ . Podemos tomar una sola  $E$  y aplicar producciones en forma repetida y en cualquier orden para obtener una secuencia de sustituciones. Por ejemplo,

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$$

A dicha secuencia de sustituciones la llamamos una derivación de  $-(\mathbf{id})$  a partir de  $E$ . Esta derivación proporciona la prueba de que la cadena  $-(\mathbf{id})$  es una instancia específica de una expresión. Para una definición general de la derivación, considere un no terminal  $A$  en la mitad de una secuencia de símbolos gramaticales, como en  $\alpha A \beta$ , en donde  $\alpha$  y  $\beta$  son cadenas arbitrarias de símbolos gramaticales. Suponga que  $A \rightarrow \gamma$  es una producción. Entonces, escribimos  $\alpha A \beta \Rightarrow \alpha \gamma \beta$ . El símbolo  $\Rightarrow$  significa, “se deriva en un paso”. Cuando una secuencia de pasos de derivación  $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$  se rescribe como  $\alpha_1$  a  $\alpha_n$ , decimos que  $\alpha_1$  deriva a  $\alpha_n$ . Con frecuencia es conveniente poder decir, “deriva en cero o más pasos”. Para este fin, podemos usar el símbolo  $\Rightarrow^*$ . Así,

1.  $\alpha \xrightarrow{*} \alpha$ , para cualquier cadena  $\alpha$ .
2. Si  $\alpha \xrightarrow{*} \beta$  y  $\beta \Rightarrow \gamma$ , entonces  $\alpha \xrightarrow{*} \gamma$ .

De igual forma,  $\Rightarrow$  significa “deriva en uno o más pasos”. Si  $S \Rightarrow^* \alpha$ , en donde  $S$  es el símbolo inicial de una gramática  $G$ , decimos que  $\alpha$  es una forma de frase de  $G$ . Observe que una forma de frase puede contener tanto terminales como no terminales, y puede estar vacía. Un enunciado de  $G$  es una forma de frase sin símbolos no terminales. El lenguaje generado por una gramática es su conjunto de



oraciones. Por ende, una cadena de terminales  $w$  está en  $L(G)$ , el lenguaje generado por  $G$ , si y sólo si  $w$  es un enunciado de  $G$  ( $\text{o } S \Rightarrow^* w$ ). Un lenguaje que puede generarse mediante una gramática se considera un lenguaje libre de contexto. Si dos gramáticas generan el mismo lenguaje, se consideran como equivalentes.

Cada no terminal se sustituye por el mismo cuerpo en las dos derivaciones, pero el orden de las sustituciones es distinto. Para comprender la forma en que trabajan los analizadores sintácticos, debemos considerar las derivaciones en las que el no terminal que se va a sustituir en cada paso se elige de la siguiente manera:

1. En las derivaciones por la izquierda, siempre se elige el no terminal por la izquierda en cada de frase. Si  $\alpha \Rightarrow \beta$  es un paso en el que se sustituye el no terminal por la izquierda en  $\alpha$ , escribimos  $\alpha \Rightarrow lm \beta$ .
2. En las derivaciones por la derecha, siempre se elige el no terminal por la derecha; en este caso escribimos  $\alpha \Rightarrow rm \beta$ .

La derivación (4.8) es por la izquierda, por lo que puede rescribirse de la siguiente manera:

$$E \xrightarrow{lm} -E \xrightarrow{lm} -(E) \xrightarrow{lm} -(E+E) \xrightarrow{lm} -(\text{id}+E) \xrightarrow{lm} -(\text{id}+\text{id})$$

Si utilizamos nuestras convenciones de notación, cada paso por la izquierda puede escribirse como  $wAy \Rightarrow lm w\delta y$ , en donde  $w$  consiste sólo de terminales,  $A \rightarrow \delta$  es la producción que se aplica, y  $y$  es una cadena de símbolos gramaticales. Para enfatizar que  $\alpha$  deriva a  $\beta$  mediante una derivación por la izquierda, escribimos  $\alpha \Rightarrow^* lm \beta$ . Si  $S \Rightarrow^* lm \alpha$ , decimos que  $\alpha$  es una forma de frase izquierda de la gramática en cuestión. Las análogas definiciones son válidas para las derivaciones por la derecha. A estas derivaciones se les conoce algunas veces como derivaciones canónicas.

### Árboles de análisis sintáctico y derivaciones

Un árbol de análisis sintáctico es una representación gráfica de una derivación que filtra el orden en el que se aplican las producciones para sustituir los no terminales. Cada nodo interior de un árbol de análisis sintáctico representa la aplicación de una producción. El nodo interior se etiqueta con el no terminal  $A$  en el encabezado de la producción; los hijos del nodo se etiquetan, de izquierda a derecha, mediante los símbolos en el cuerpo de la producción por la que se sustituyó esta  $A$  durante la derivación.



Las hojas de un árbol de análisis sintáctico se etiquetan mediante no terminales o terminales y, leídas de izquierda a derecha, constituyen una forma de frase, a la cual se le llama producto o frontera del árbol. Para ver la relación entre las derivaciones y los árboles de análisis sintáctico, considere cualquier derivación  $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$ , en donde  $\alpha_1$  es un sólo no terminal A. Para cada forma de frase  $\alpha_i$  en la derivación, podemos construir un árbol de análisis sintáctico cuyo producto sea  $\alpha_i$ . El proceso es una inducción sobre i.

BASE: El árbol para  $\alpha_1 = A$  es un solo nodo, etiquetado como A.

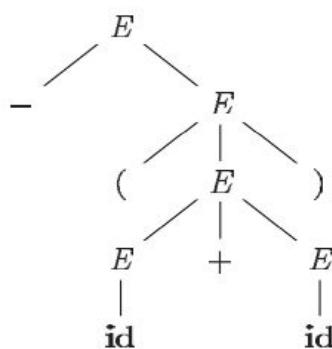
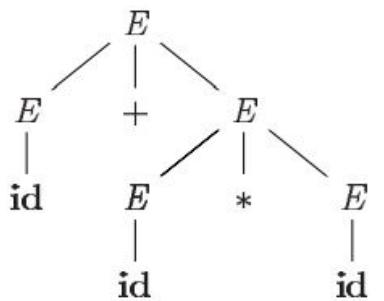


Figura 4.3: Árbol de análisis sintáctico para  $-(\text{id} + \text{id})$

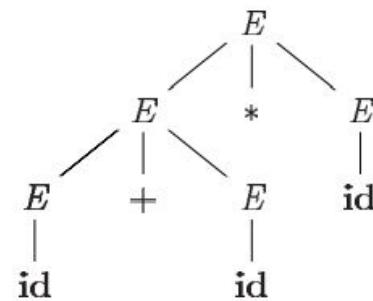
INDUCCIÓN: Suponga que ya hemos construido un árbol de análisis sintáctico con el producto  $\alpha_{i-1} = X_1 X_2 \dots X_k$  (tenga en cuenta que, de acuerdo a nuestras convenciones de notación, cada símbolo gramatical  $X_i$  es un no terminal o un terminal). Suponga que  $\alpha_i$  se deriva de  $\alpha_{i-1}$  al sustituir  $X_j$ , un no terminal, por  $\beta = Y_1 Y_2 \dots Y_m$ . Es decir, en el  $i$ -ésimo paso de la derivación, la producción  $X_j \rightarrow \beta$  se aplica a  $\alpha_{i-1}$  para derivar  $\alpha_i = X_1 X_2 \dots X_{j-1} \beta X_{j+1} \dots X_k$ . Para modelar este paso de la derivación, buscamos la  $j$ -ésima hoja, partiendo de la izquierda en el árbol de análisis sintáctico actual. Esta hoja se etiqueta como  $X_j$ . A esta hoja le damos  $m$  hijos, etiquetados  $Y_1, Y_2, \dots, Y_m$ , partiendo de la izquierda. Como caso especial, si  $m = 0$  entonces  $\beta = \epsilon$ , y proporcionamos a la  $j$ -ésima hoja un hijo etiquetado como  $\epsilon$ .

### Ambigüedad

Para la mayoría de los analizadores sintácticos, es conveniente que la gramática no tenga ambigüedades, ya que de lo contrario, no podemos determinar en forma única qué árbol de análisis sintáctico seleccionar para un enunciado. En otros casos, es conveniente usar gramáticas ambiguas elegidas con cuidado, junto con reglas para eliminar la ambigüedad, las cuales “descartan” los árboles sintácticos no deseados, dejando sólo un árbol para cada enunciado.



(a)



(b)

Figura 4.5: Dos árboles de análisis sintáctico para **id+id\*id**

#### Verificación del lenguaje generado por una gramática

Aunque los diseñadores de compiladores muy raras veces lo hacen para una gramática de lenguaje de programación completa, es útil poder razonar que un conjunto dado de producciones genera un lenguaje específico. Las construcciones problemáticas pueden estudiarse mediante la escritura de una gramática abstracta y concisa, y estudiando el lenguaje que genera. A continuación vamos a construir una gramática de este tipo, para instrucciones condicionales. Una prueba de que una gramática  $G$  genera un lenguaje  $L$  consta de dos partes: mostrar que todas las cadenas generadas por  $G$  están en  $L$  y, de manera inversa, que todas las cadenas en  $L$  pueden generarse sin duda mediante  $G$ .

#### Comparación entre gramáticas libres de contexto y expresiones regulares

Antes de dejar esta sección sobre las gramáticas y sus propiedades, establecemos que las gramáticas son una notación más poderosa que las expresiones regulares. Cada construcción que puede describirse mediante una expresión regular puede describirse mediante una gramática, pero no al revés. De manera alternativa, cada lenguaje regular es un lenguaje libre de contexto, pero no al revés. Por ejemplo, la expresión regular  $(a|b)^*$  abb y la siguiente gramática:

$$\begin{array}{lcl} A_0 & \rightarrow & aA_0 \mid bA_0 \mid aA_1 \\ A_1 & \rightarrow & bA_2 \\ A_2 & \rightarrow & bA_3 \\ A_3 & \rightarrow & \epsilon \end{array}$$

describen el mismo lenguaje, el conjunto de cadenas de as y bs que terminan en abb.

Podemos construir de manera mecánica una gramática para reconocer el mismo lenguaje que un autómata finito no determinista (AFN). La gramática anterior se construyó a partir del AFN de la figura 3.24, mediante la siguiente construcción:

1. Para cada estado  $i$  del AFN, crear un no terminal  $A_i$ .
2. Si el estado  $i$  tiene una transición al estado  $j$  con la entrada  $a$ , agregar la producción  $A_i \rightarrow a A_j$ . Si el estado  $i$  pasa al estado  $j$  con la entrada  $\epsilon$ , agregar la producción  $A_i \rightarrow A_j$ .
3. Si  $i$  es un estado de aceptación, agregar  $A_i \rightarrow \cdot$ .
4. Si  $i$  es el estado inicial, hacer que  $A_i$  sea el símbolo inicial de la gramática.

### **Escritura de una gramática**

Las gramáticas son capaces de describir casi la mayoría de la sintaxis de los lenguajes de programación. Por ejemplo, el requerimiento de que los identificadores deben declararse antes de usarse, no puede describirse mediante una gramática libre de contexto. Por lo tanto, las secuencias de los tokens que acepta un analizador sintáctico forman un superconjunto del lenguaje de programación; las fases siguientes del compilador deben analizar la salida del analizador sintáctico, para asegurar que cumpla con las reglas que no verifica el analizador sintáctico.

### **Comparación entre análisis léxico y análisis sintáctico**

sería razonable preguntar: “¿Por qué usar expresiones regulares para definir la sintaxis léxica de un lenguaje?” Existen varias razones.

1. Al separar la estructura sintáctica de un lenguaje en partes léxicas y no léxicas, se proporciona una manera conveniente de colocar en módulos la interfaz de usuario de un compilador en dos componentes de un tamaño manejable.
2. Las reglas léxicas de un lenguaje son con frecuencia bastante simples, y para describirlas no necesitamos una notación tan poderosa como las gramáticas.
3. Por lo general, las expresiones regulares proporcionan una notación más concisa y fácil de entender para los tokens, en comparación con las gramáticas.
4. Pueden construirse analizadores léxicos más eficientes en forma automática a partir de expresiones regulares, en comparación con las gramáticas arbitrarias.

No hay lineamientos firmes en lo que se debe poner en las reglas léxicas, en contraste a las reglas sintácticas. Las expresiones regulares son muy útiles para describir la estructura de las construcciones como los identificadores, las constantes, las palabras reservadas y el espacio en blanco. Por otro lado, las gramáticas son muy útiles para describir estructuras anidadas, como los paréntesis balanceados, las instrucciones begin-end relacionadas, las instrucciones if-then-else correspondientes, etcétera. Estas estructuras anidadas no pueden describirse mediante las expresiones regulares.

### **Eliminación de la ambigüedad**

Algunas veces, una gramática ambigua puede rescribirse para eliminar la ambigüedad. Como ejemplo, vamos a eliminar la ambigüedad de la siguiente gramática del “else colgante”:



*instr* → **if** *expr* **then** *instr*  
|      **if** *expr* **then** *instr* **else** *instr*  
|      **otra**

Aquí, “otra” representa a cualquier otra instrucción. De acuerdo con esta gramática, la siguiente instrucción condicional compuesta:

**if** *E*<sub>1</sub> **then** *S*<sub>1</sub> **else** **if** *E*<sub>2</sub> **then** *S*<sub>2</sub> **else** *S*<sub>3</sub>

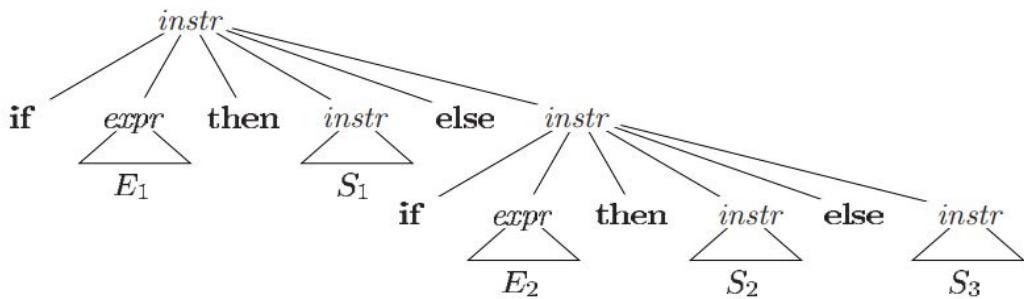


Figura 4.8: Árbol de análisis sintáctico para una instrucción condicional

#### Eliminación de la recursividad por la izquierda

Una gramática es recursiva por la izquierda si tiene una terminal A tal que haya una derivación  $A \Rightarrow A\alpha$  para cierta cadena  $\alpha$ . Los métodos de análisis sintáctico descendentes no pueden manejar las gramáticas recursivas por la izquierda, por lo que se necesita una transformación para eliminar la recursividad por la izquierda.

La recursividad inmediata por la izquierda puede eliminarse mediante la siguiente técnica, que funciona para cualquier número de producciones A. En primer lugar, se agrupan las producciones de la siguiente manera:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

en donde ninguna  $\beta_i$  termina con una A. Después, se sustituyen las producciones A mediante lo siguiente:



$$\begin{array}{l} A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A' \\ A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon \end{array}$$

El no terminal A genera las mismas cadenas que antes, pero ya no es recursiva por la izquierda. Este procedimiento elimina toda la recursividad por la izquierda de las producciones A y A' (siempre y cuando ninguna  $\alpha_i$  sea  $\epsilon$ ), pero no elimina la recursividad por la izquierda que incluye a las derivaciones de dos o más pasos. Por ejemplo, considere la siguiente gramática:

$$\begin{array}{l} S \rightarrow A \ a \mid b \\ A \rightarrow A \ c \mid S \ d \mid \epsilon \end{array}$$

El no terminal S es recursiva por la izquierda, ya que  $S \Rightarrow Aa \Rightarrow Sda$ , pero no es inmediatamente recursiva por la izquierda.

Cabrera Ramírez Gerardo