



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






Compilador: es un programa que puede leer programas en un lenguaje (el lenguaje fuente) y traducirlo en un programa en otro lenguaje (el lenguaje destino).

programa fuente  compilador  programa destino

Si el programa destino es un programa ejecutable en lenguaje máquina, entonces el usuario puede ejecutarlo para procesar las entradas y producir salidas.

entrada  programa destino  salida

Intérprete: Este en vez de producir un programa destino como traducción, este nos da la apariencia de ejecutar directamente las operaciones especificadas en el programa de origen (fuente) con las entradas proporcionadas por el usuario.

programa fuente 
entrada  intérprete  salida

En la estructura de un compilador hay dos procesos:

Análisis: Que divide el programa fuente en componentes e impone una estructura gramatical sobre ellas. Si esta parte detecta un error de sintaxis en el programa fuente, este debe enviar un mensaje de error al usuario para que se corrija.

Síntesis: Construye el programa destino deseado a partir de la representación intermedia y de la información en la tabla de símbolos.

Análisis del léxico: Este lee el flujo de caracteres que componen el programa fuente y los agrupa en secuencias significativas, conocidas como lexemas. Para cada lexema, el analizador léxico produce como salida un token de la forma: nombre-token, valor-atributo

Análisis sintáctico: Utiliza los primeros componentes de los tokens generados por el analizador de léxico para crear una representación que describa la estructura gramatical del flujo de tokens.

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. Lo importante de esto es la comprobación (verificación) de tipos, en donde el compilador verifica que cada operador tenga operandos que coincidan.

Generación de código: recibe como entrada una representación intermedia del programa fuente y la asigna al lenguaje destino. Si el lenguaje destino es código máquina, se seleccionan registros o ubicaciones de memoria para cada variable que utiliza el programa.

y como ya todos sabemos el lenguaje máquina es el que puede leer las computadoras como todos sus componentes ya sea la RAM, procesador, etc. y el de alto nivel es el que el usuario puede ordenar a la máquina que hacer a través de instrucciones en nuestro lenguaje.

Análisis Sintáctico (parte 1)

Es el proceso de determinar cómo puede generarse una cadena de terminales mediante una gramática.

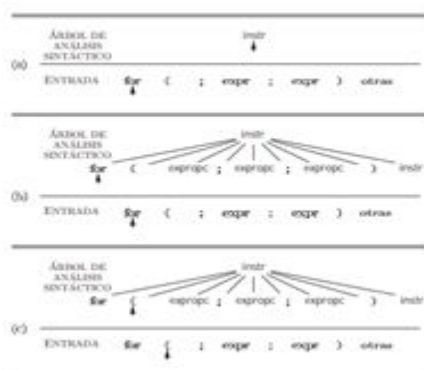
Esta sección introduce un método de análisis sintáctico conocido como “descenso recursivo”, este puede usarse tanto para el análisis sintáctico, como para la implementación de traductores orientados a la sintaxis.

La mayoría de los métodos de análisis sintáctico se adaptan a una de dos clases, llamadas métodos descendente y ascendente. Estos se refieren al orden en el que se construyen los nodos en el árbol de análisis sintáctico. En los analizadores tipo descendente, la construcción empieza en la raíz y procede hacia las hojas, mientras que en los analizadores tipo ascendente, la construcción empieza en las hojas y procede hacia la raíz.

Análisis sintáctico tipo arriba-abajo

La construcción descendente de un árbol de análisis sintáctico como el de la figura 2.17 se realiza empezando con la raíz, etiquetada con el no terminal inicial *instr*

instr → *expr* ;
 | *if* (*expr*) *instr*
 | *for* (*expr* ; *expr* ; *expr*) *instr*
 | *otras*
expr → *ε*
 | *expr*



Análisis sintáctico predictivo

Aquí consideraremos una forma simple de análisis sintáctico de descenso recursivo, conocido como análisis sintáctico predictivo, en el cual el símbolo de preanálisis determina sin ambigüedad el flujo de control a través del cuerpo del procedimiento para cada no terminal.

Árboles sintácticos para las instrucciones

<i>programa</i> →	<i>bloque</i>	{ return <i>bloque.n</i> ; }
<i>bloque</i> →	'{ <i>instrs</i> }'	{ <i>bloque.n</i> = <i>instrs.n</i> ; }
<i>instrs</i> →	<i>instrs</i> , <i>instr</i>	{ <i>instrs.n</i> = new <i>Seq</i> (<i>instrs.n</i> , <i>instr.n</i>); }
	<i>*</i>	{ <i>instrs.n</i> = null; }
<i>instr</i> →	<i>expr</i> ;	{ <i>instr.n</i> = new <i>Expr</i> (<i>expr.n</i>); }
	if (<i>expr</i>) <i>instrs</i>	{ <i>instr.n</i> = new <i>If</i> (<i>expr.n</i> , <i>instrs.n</i>); }
	while (<i>expr</i>) <i>instrs</i>	{ <i>instr.n</i> = new <i>While</i> (<i>expr.n</i> , <i>instrs.n</i>); }
	do <i>instrs</i> while (<i>expr</i>)	{ <i>instr.n</i> = new <i>DoWhile</i> (<i>instrs.n</i> , <i>expr.n</i>); }
	<i>bloque</i>	{ <i>instr.n</i> = <i>bloque.n</i> ; }
<i>expr</i> →	<i>rel</i> * <i>expr</i>	{ <i>expr.n</i> = new <i>Assign</i> ('=', <i>rel.n</i> , <i>expr1.n</i>); }
	<i>rel</i>	{ <i>expr.n</i> = <i>rel.n</i> ; }
<i>rel</i> →	<i>rel</i> < <i>adic</i>	{ <i>rel.n</i> = new <i>Rel</i> ('<', <i>rel1.n</i> , <i>adic1.n</i>); }
	<i>rel</i> <= <i>adic</i>	{ <i>rel.n</i> = new <i>Rel</i> ('<=', <i>rel1.n</i> , <i>adic1.n</i>); }
	<i>adic</i>	{ <i>rel.n</i> = <i>adic.n</i> ; }
<i>adic</i> →	<i>adic</i> , * <i>term</i>	{ <i>adic.n</i> = new <i>Op</i> ('*', <i>adic1.n</i> , <i>term.n</i>); }
	<i>term</i>	{ <i>adic.n</i> = <i>term.n</i> ; }
<i>term</i> →	<i>term</i> , * <i>factor</i>	{ <i>term.n</i> = new <i>Op</i> ('*', <i>term1.n</i> , <i>factor.n</i>); }
	<i>factor</i>	{ <i>term.n</i> = <i>factor.n</i> ; }
<i>factor</i> →	(<i>expr</i>)	{ <i>factor.n</i> = <i>expr.n</i> ; }
	<i>num</i>	{ <i>factor.n</i> = new <i>Num</i> (<i>num.value</i>); }

Figura 2.28: Construcción de árboles sintácticos para expresiones e instrucciones

Árboles sintácticos para las expresiones

SINTAXIS CONCRETA	SINTAXIS ABSTRACTA
=	asigna
	cond
&&	cond
== !=	rel
< <= >= >	rel
+ -	op
* / %	op
!	not
- unario	menos
[]	acceso

Análisis Léxico (partes 2 y 3)

Un analizador léxico lee los caracteres de la entrada y los agrupa en “objetos token”. Junto con un símbolo de terminal que se utiliza para las decisiones de análisis sintáctico, un objeto token lleva información adicional en forma de valores de atributos. El analizador léxico en esta sección permite que aparezcan números, identificadores y “espacio en blanco” (espacios, tabuladores y nuevas líneas) dentro de las expresiones.

Lectura adelantada

Si el siguiente carácter es =, entonces > forma parte de la secuencia de caracteres >=, el lexema para el token del operador “mayor o igual que”. En caso contrario, > por sí solo forma el operador “mayor que”, y el analizador léxico ha leído un carácter de más. Un método general para leer por adelantado en la entrada es mantener un búfer de entrada, a partir del cual el analizador léxico puede leer y devolver caracteres.



Un analizador léxico

Hasta ahora en esta sección, los fragmentos de pseudocódigo se juntan para formar una función llamada escanear que devuelve objetos token, de la siguiente manera:

```
Token escanear () {
    omitir espacio en blanco;
    manejar los números;
    manejar las palabras reservadas e identificadores;
    /* Si llegamos aquí, tratar el carácter de lectura de preanálisis vistazo como token */
    Token t = new Token(vistazo);
    vistazo = espacio en blanco /* inicialización, como lo vimos antes */;
    return t;
}
```

Tokens, patrones y lexemas

- 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.

TOKEN	DESCRIPCIÓN INFORMAL	LEXEMAS DE EJEMPLO
if	caracteres i, f	if
else	caracteres e, l, s, e	Else
comparacion	< o > o <= o >= o == o !=	<=, !=
id	letra seguida por letras y dígitos	pi, puntuacion, D2
numero	cualquier constante numérica	3.14159, 0, 6.02e23
literal	cualquier cosa excepto " , rodeada por " 's	"core dumped"

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.

Ejemplo: Los nombres de los tokens y los valores de atributo asociados para la siguiente instrucción en Fortran:

`E = M * C ** 2`

se escriben a continuación como una secuencia de pares.

<id, apuntador a la entrada en la tabla de símbolos para E>

<asigna-op>

<id, apuntador a la entrada en la tabla de símbolos para M>

<mult-op>

<id, apuntador a la entrada en la tabla de símbolos para C>

<exp-op>

<numero, valor entero 2>

Observe que en ciertos pares en especial en los operadores, signos de puntuación y palabras clave, no hay necesidad de un valor de atributo.

Errores léxicos

Sin la ayuda de los demás componentes es difícil para un analizador léxico saber que hay un error en el código fuente.

Ejemplo:

`fi (a == f(x)) ...`

Un analizador léxico no puede saber si `fi` es una palabra clave `if` mal escrita, o un identificador de una función no declarada. Como `fi` es un lexema válido para el token `id`, el analizador léxico debe regresar el token `id` al analizador sintáctico y dejar que alguna otra fase del compilador (quizá el analizador sintáctico en este caso) mande un error debido a la transposición de las letras.



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.

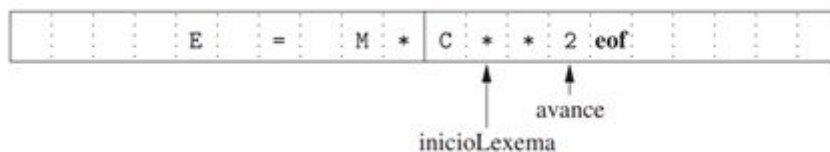
Uso de búfer en la entrada

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.

Pares de búferes

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.

Un esquema importante implica el uso de dos búferes que se recargan en forma alterna:



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.

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.

Especificación de los tokens



Las expresiones regulares son una notación importante para especificar patrones de lexemas. Aunque no pueden expresar todos los patrones posibles, son muy efectivas para especificar los tipos de patrones que en realidad necesitamos para los tokens.

Operaciones en los lenguajes

La concatenación de lenguajes es cuando se concatenan todas las cadenas que se forman al tomar una cadena del primer lenguaje y una cadena del segundo lenguaje, en todas las formas posibles. La cerradura (Kleene) de un lenguaje L , que se denota como L^* , es el conjunto de cadenas que se obtienen al concatenar L cero o más veces. Observe que L^0 , la "concatenación de L cero veces", se define como $\{\epsilon\}$, y por inducción, L^i es $L^{i-1} L$. Por último, la cerradura positiva, denotada como L^+ , es igual que la cerradura de Kleene, pero sin el término L^0 . Es decir, no estará en L^+ a menos que esté en el mismo L .

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

Definiciones regulares

Si Σ es un alfabeto de símbolos básicos, entonces una definición regular es una secuencia de definiciones de la forma:

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

...

$d_n \rightarrow r_n$

En donde:

1. Cada d_i es un nuevo símbolo, que no está en Σ y no es el mismo que cualquier otro d .
2. Cada r_i es una expresión regular sobre el alfabeto $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$.

Al restringir r_i a Σ y a las d s definidas con anterioridad, evitamos las definiciones recursivas y podemos construir una expresión regular sobre Σ solamente, para cada r_i .

Reconocimiento de tokens

Ahora debemos estudiar cómo tomar todos los patrones para todos los tokens necesarios y construir una pieza de código para examinar la cadena de entrada y buscar un prefijo que sea un lexema que coincida con uno de esos patrones.

$\text{instr} \rightarrow \text{if expr then instr}$

$| \text{if expr then instr else instr}$

$| \epsilon$

$\text{expr} \rightarrow \text{term opre term}$



| term

term → id

| numero

Para oprel, 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, oprel, id y numero, son los nombres de tokens en lo que al analizador léxico respecta.

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

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.

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).

Ejemplo:

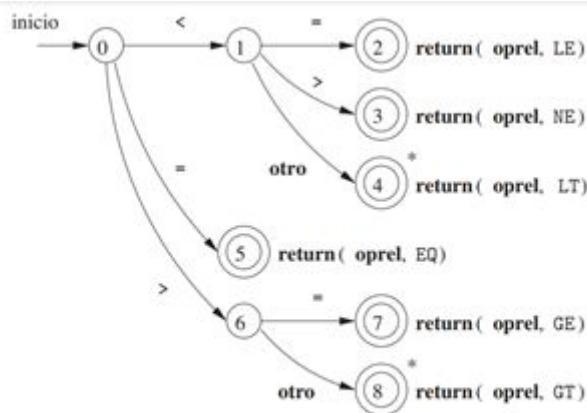


Figura 3.13: Diagrama de transición de estados para **oprel**

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. Para buscar lexemas de identificadores, este diagrama también reconocerá las palabras clave `if`, `then` y `else` de nuestro bosquejo.



Figura 3.14: Un diagrama de transición de estados para identificadores (**id**) y palabras clave

Finalización del bosquejo

El diagrama de transición para los tokens `id` que vimos en la figura tiene una estructura simple. Empezando en el estado 9, comprueba que el lexema empiece con una letra y que pase al estado 10 en caso de ser así. Permanecemos en el estado 10 siempre y cuando la entrada contenga letras y dígitos. Al momento de encontrar el primer carácter que no sea letra ni dígito, pasamos al estado 11 y aceptamos el lexema encontrado.

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

Hay varias formas en las que pueden utilizarse los diagramas de transición de estados para construir un analizador léxico. Cada estado se representa mediante una pieza de código. 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.

Ejemplo:



En la figura 3.18 vemos un bosquejo de obtenerOpRel(), una función en C++ cuyo trabajo es simular el diagrama de transición de la figura 3.13 y devolver un objeto de tipo TOKEN

```
TOKEN obtenerOpRel()
{
    TOKEN tokenRet = new (OPREL);
    while(1) { /* repite el procesamiento de caracteres hasta que
                ocurre un retorno o un fallo */
        switch(estado) {
            case 0: c = sigCar();
                    if ( c == '<' ) estado = 1;
                    else if ( c == '=' ) estado = 5;
                    else if ( c == '>' ) estado = 6;
                    else fallo(); /* el lexema no es un oprel */
                    break;
            case 1: ...
            ...
            case 8: retractar();
                    tokenRet.atributo = GT;
                    return(tokenRet);
        }
    }
}
```

Figura 3.18: Bosquejo de la implementación del diagrama de transición **oprel**

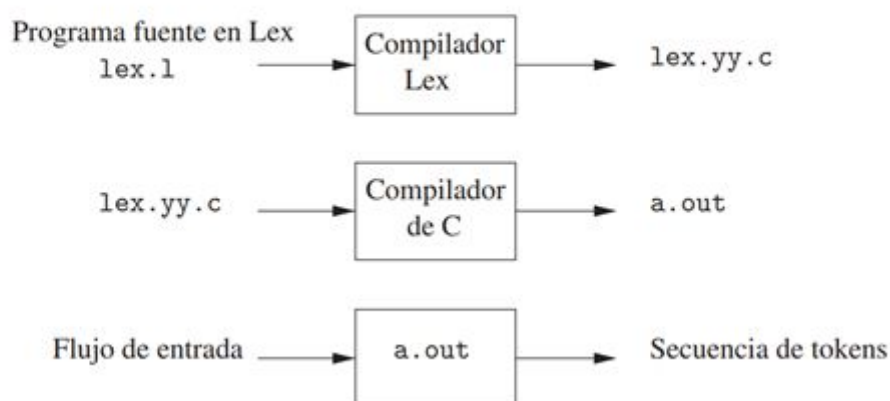
El generador de analizadores léxicos Lex

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.

Uso de lex

Un archivo de entrada, al que llamaremos lex.1, está escrito en el lenguaje Lex y describe el analizador léxico que se va a generar. El compilador Lex transforma a lex.1 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.

El generador de analizadores léxicos Lex





Estructura de los programas en Lex

Un programa en Lex tiene la siguiente forma:

declaraciones

%%

reglas de traducción

%%

funciones auxiliares

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.

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.

Autómatas finitos

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.

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 Σ , el alfabeto de entrada. Suponemos que , que representa a la cadena vacía, nunca será miembro de Σ .
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).

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 ϵ . 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.

Ejemplo:

Autómatas finitos

ESTADO	a	b	ϵ
0	$\{0, 1\}$	$\{0\}$	\emptyset
1	\emptyset	$\{2\}$	\emptyset
2	\emptyset	$\{3\}$	\emptyset
3	\emptyset	\emptyset	\emptyset

Autómatas finitos deterministas

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

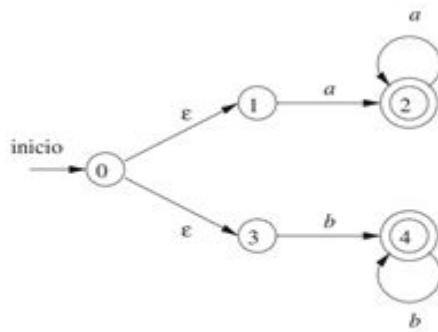


Figura 3.26: AFN que acepta a $aa^*|bb^*$

1. No hay movimientos en la entrada .
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.

Ejemplo AFN:

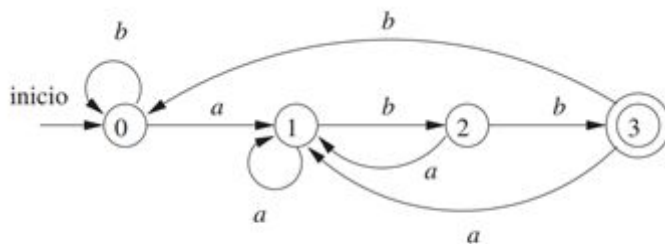


Figura 3.28: AFD que acepta a $(a|b)^*abb$

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.

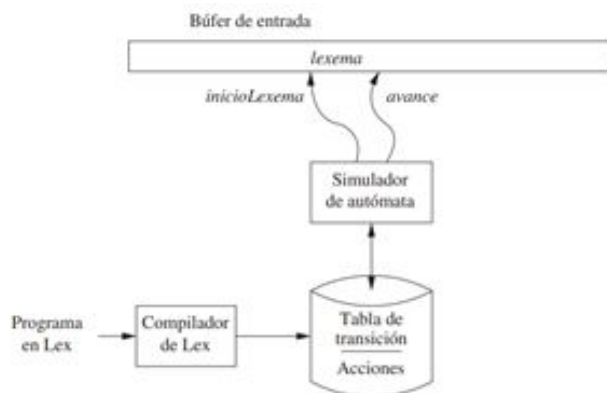


Figura 3.49: Un programa en Lex se convierte en una tabla de transición y en acciones, para que las utilice un simulador de autómatas finitos

Estos componentes son:

1. Una tabla de transición para el autómata.
2. Las funciones que se pasan directamente a través de Lex a la salida.
3. Las acciones del programa de entrada, que aparece como fragmentos de código que el simulador del autómata debe invocar en el momento apropiado.

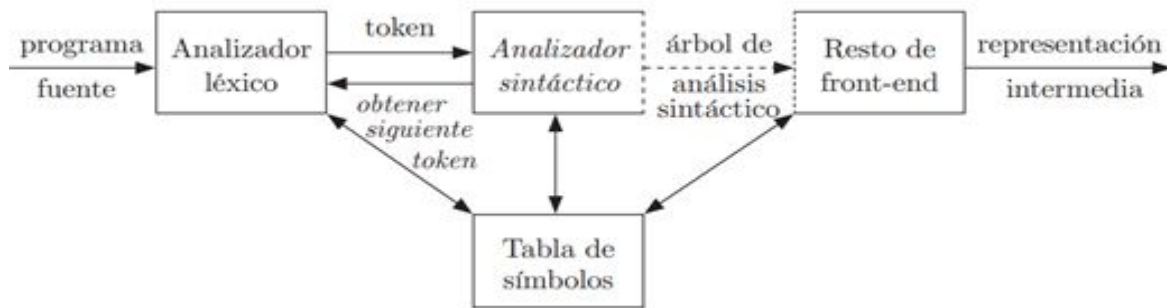
Optimización de los buscadores por concordancia de patrones basados en AFD

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.

Análisis sintactico

La función del analizador sintáctico

el analizador sintáctico obtiene una cadena de tokens del analizador léxico, como se muestra en la imagen:



Y verifica que la cadena de nombres de los tokens pueda generarse mediante la gramática para el lenguaje fuente. Esperamos que el analizador sintáctico reporte cualquier error sintáctico en forma inteligible y que se recupere de los errores que ocurren con frecuencia para seguir procesando el resto del programa.

Para los programas bien formados, el analizador sintáctico construye un árbol de análisis sintáctico y lo pasa al resto del compilador para que lo siga procesando.

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. 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.

Métodos descendentes: Construyen árboles de análisis sintáctico de la parte superior (raíz) a la parte inferior (hojas).

Métodos ascendentes: Empiezan de las hojas y avanzan hasta la raíz.

En cualquier caso, la entrada al analizador se explora de izquierda a derecha, un símbolo a la vez.

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.

La asociatividad y la precedencia se resuelvan en la siguiente gramática:

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:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \end{aligned} \quad (4.1)$$



$F \rightarrow (E) \mid id$

La gramática para expresiones (4.1) pertenece a la clase de gramáticas LR que son adecuadas para el análisis sintáctico ascendentes.

Manejo de los errores sintácticos

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 tamañoElipce en vez de tamañoElipse, 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.

El manejo 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.

Estrategias para recuperarse de los errores

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

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.

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. 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.

La sustitución a nivel de frase se ha utilizado en varios compiladores que reparan los errores, ya que puede corregir cualquier cadena de entrada.

Producciones de errores

Al anticipar los errores comunes que podríamos encontrar, podemos aumentar la gramática para el lenguaje, con producciones que generen 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.

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 a x en y sea lo más pequeño posible.

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:

$$\text{instr} \rightarrow \text{if (expr) instr else instr} \quad (4.4)$$

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 .

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 los terminales son los primeros componentes de los tokens que produce el analizador léxico. En (4.4), 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. En (4.4), *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 \rightarrow . 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.

En esta gramática, los símbolos de los terminales son:

id + - * / ()

Los símbolos de los no terminales son *expresión*, *term* y *factor*, y *expresión* es el símbolo inicial.

expresión \rightarrow *expresión* + *term*

expresión \rightarrow *expresión* - *term*

expresión \rightarrow *term*

term \rightarrow *term* * *factor*

term \rightarrow *term* / *factor*

term \rightarrow *factor*

factor \rightarrow (*expresión*)

factor \rightarrow *id*

Convenciones de notación



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 α , β , γ , 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 α 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 $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_k$. A α_1 , α_2 , ..., α_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.

$$\begin{array}{lcl}
 E & \rightarrow & E \quad \mid \quad E - \quad \mid \\
 & \rightarrow & * \quad \mid \quad \blacksquare \quad \mid \\
 & \rightarrow & (E) \quad \mid \quad \textbf{id}
 \end{array}$$

Las convenciones de notación nos indican que E, T y F son no terminales, y E es el símbolo inicial. El resto de los símbolos son terminales.

Derivaciones



Considere la siguiente gramática, con un solo no terminal E, la cual agrega una producción $E \rightarrow - E$

$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \text{id} \quad (4.7)$$

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).

Podemos tomar una sola E y aplicar producciones en forma repetida y en cualquier orden para obtener una secuencia de sustituciones. Por ejemplo:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$$

A dicha secuencia de sustituciones la llamamos una derivación de $-(id)$ a partir de E.

La cadena $-(id + id)$ es un enunciado de la gramática (4.7), ya que hay una derivación

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(id + E) \Rightarrow -(id + id) \quad (4.8)$$

Las cadenas E, $-E$, $-(E)$, ..., $-(id + id)$ son todas formas de frases de esta gramática.

Escribimos $E \Rightarrow - * (id + id)$ para indicar que $-(id + id)$ puede derivarse de E.

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.

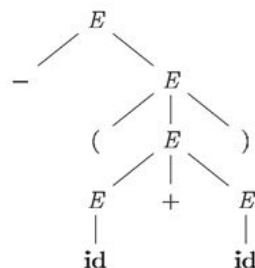


Figura 4.3: Árbol de análisis sintáctico para $-(id + id)$

Ambigüedad



Una gramática ambigua es aquella que produce más de una derivación por la izquierda, o más de una derivación por la derecha para el mismo enunciado.

Ejemplo 4.11:

La gramática de expresiones aritméticas (4.3) permite dos derivaciones por la izquierda distintas para el enunciado $\text{id} + \text{id} * \text{id}$:

$$\begin{array}{ll}
 E \Rightarrow E & E \\
 \Rightarrow \text{id} & E \\
 \Rightarrow \text{id} & E * E \\
 \Rightarrow \text{id} & \text{id} * E \\
 \Rightarrow \text{id} & \text{id} * \text{id}
 \end{array}
 \qquad
 \begin{array}{ll}
 E \Rightarrow E * E \\
 \Rightarrow E & E * E \\
 \Rightarrow \text{id} & E * E \\
 \Rightarrow \text{id} & \text{id} * E \\
 \Rightarrow \text{id} & \text{id} * \text{id}
 \end{array}$$

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.

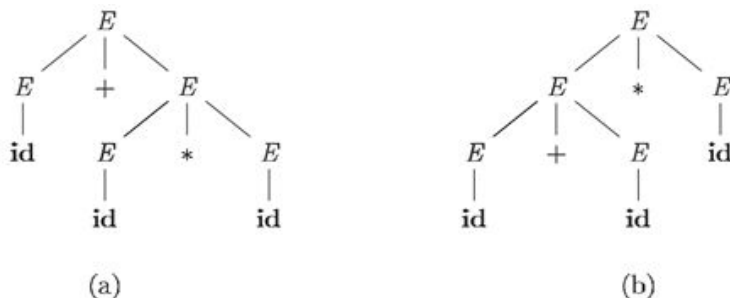


Figura 4.5: Dos árboles de análisis sintáctico para $\text{id} + \text{id} * \text{id}$

Comparación entre gramáticas libres de contexto y expresiones regulares

Toda estructura que se pueda describir mediante expresiones regulares se puede describir mediante sintaxis, pero no de forma opuesta. O, todo lenguaje convencional es un lenguaje sin contexto, pero no al revés.

Por ejemplo:

La expresión regular $(\text{alb})^* \text{abb}$ y la siguiente gramática:



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

Podemos construir de manera mecánica una gramática para reconocer el mismo lenguaje que un autómata finito no determinista (AFN), 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 $A_i \rightarrow aA_j$. Si el estado i pasa al estado j con la entrada ϵ , agregar la producción $A_i \rightarrow A_j$.
3. Si i es un estado de aceptación, agregar $A_i \rightarrow \epsilon$.
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.

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

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.

Morales Carrillo Gerardo