**CAPITULO 1**

**Compilador**

Es un programa que lee un programa escrito en lenguaje *fuente,* y lo traduce a un programa equivalente en lenguaje *objeto.*

|  |  |  |
| --- | --- | --- |
| programa | *compilador* | *9* programa |
| fuente | objeto |
|  | mensajes de error |  |

En la compilación hay dos partes: análisis y síntesis.

La parte del **análisis** divide al programa fuente en sus elementos componentes y crea una representación inter­media del programa fuente.

La parte de la **síntesis** construye el programa objeto de­seado a partir de la representación intermedia

análisis lineal o **análisis léxico**

En este el programa fuente se lee de izquierda a derecha y se agrupa en ***componentes léxicos****,* que son secuencias de caracteres que tienen un significado colectivo. La secuencia de caracteres que forma un componente léxico se denomina ***lexema***del componente.

análisis léxico los caracteres de la proposición de asignación

posición := inicial + velocidad \* 60

<id, posición > <:=, > <id, inicial> <+,> <id, velocidad> <\*,> <60,>

se agruparían en los componentes léxicos siguientes:

1. El identificador posición.
2. El símbolo de asignación : =.
3. El identificador inicial.
4. El signo de suma.
5. El identificador velocidad.
6. El signo de multiplicación.
7. El número 60.

**Análisis sintáctico (o jerárquico)**

Este implica agrupar jerárquicamente los com­ponentes léxicos del programa fuente en frases gramaticales (colecciones anidadas) que el compilador uti­liza para sintetizar la salida. Por lo general, las frases gramaticales del programa fuente se representan mediante un árbol de análisis sintáctico

Nota : orden de aparicion de los componentes lexicos

**Análisis semántico**

Ejemplo : El refrigerador verde corría alegremente

Este revisa el programa fuente para tratar de encontrar errores semánticos y reúne la información sobre los tipos para la fase posterior de generación de código. En ella se utiliza la estructura jerárquica determinada por el análisis sintáctico para identificar los operadores y operandos de expresiones y proposiciones.

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

**Generación de código intermedio**

Después de los análisis sintáctico y semántico, algunos compiladores generan una representación intermedia explícita del programa fuente. Se puede considerar esta representación intermedia como un programa para una máquina abstracta. Esta re­presentación intermedia debe tener dos propiedades importantes; debe ser fácil de producir y fácil de traducir al programa objeto.

*El "código de tres direcciones", que es como el lenguaje ensamblador para una máquina en la que cada posición de memoria puede actuar como un registro.* El código de tres direcciones consiste en una secuencia de instrucciones, cada una de las cuales tiene como máximo tres operandos. El pro­grama fuente de (1.1) puede aparecer en código de tres direcciones como

templ := entareal(60)

temp2 := id3 \* templ

temp3 := id2 + temp2

*idl := temp3*

**Optimación de código**

Este trata de mejorar el código intermedio, de modo que resulte un código de máquina más rápido de ejecutar.

templ := id3 \* 60.0 ,. ..

**idl := id2 + templ**

**Generación de código**

Este genera el código objeto, que por lo gral. consiste en código de máquina relocalizable o código ensamblador. Las posiciones de memoria se seleccionan para cada una de las variables usadas por el programa. Después, cada una de las instrucciones intermedias se traduce a una secuencia de instrucciones de máquina que ejecuta la misma tarea. Un aspecto decisivo es la asig­nación de variables a registros.

MOVF id3, R2

MULF #60.0, R2

MOVF id2, Rl

ADDF R2, Rl

MOVF Rl, idl

**CAPITULO 2**

Una *gramática independiente del contexto* tiene cuatro componentes:

1. Un conjunto de componentes léxicos, denominados símbolos *terminales.*

*2.* Un conjunto de no terminales.

3. Un conjunto de producciones, en el que cada producción consta de un no terminal, llamado *lado izquierdo* de la producción, una flecha y una secuencia de componentes léxicos y no terminales, o ambos, llamado *lado derecho* de la producción.

*4. La denominación de uno de los no terminales como símbolo inicial.*

Ejemplo de **GLC**

|  |  |
| --- | --- |
| *lista ->* | *lista + dígito* |
| *lista ->* | *lista - dígito* |
| *lista ->* | *dígito* |
| *dígito->* | *0 | 1 | 2 | 3 |4 | 5 | 6 | 7 |8 | 9* |

o

*lista -> lista + dígito* | *lista* - *dígito* | *dígito*

**Arbol de análisis sintáctico**

Indica gráficamente cómo del símbolo inicial de una gramática deriva una cadena del lenguaje.

Formalmente, dada una gramática independiente del contexto, un ***árbol de análisis sintáctico***es un árbol con las propiedades siguientes:

1. La raíz está etiquetada con el símbolo inicial.
2. Cada hoja está etiquetada con un componente léxico o con **ε**.
3. Cada nodo interior está etiquetado con un no terminal.
4. Si *A* es el no terminal que etiqueta a algún nodo interior y *X1 X2 • • Xn, son las etiquetas de los hijos*

*de ese nodo, de izquierda a derecha, entonces A —> X1 X2 • • Xn, es una producción. Aquí, X1 X2 • • Xn*

*representa un sím­bolo que es un terminal o un no terminal. Como caso especial, si .A —> e. entonces*

*un nodo etiquetado con A tiene sólo un hijo etiquetado con* **ε*.***

**Ambigüedad**

*Para demostrar que una gramática es ambigua, lo único que se requiere es encontrar una cadena de componentes léxicos que tenga más de un árbol de análisis sintáctico.*

*cadena -> cadena + cadena | cadena - cadena |0|l|2|3|4|5|6|7 | 8|9*

**Asociatividad de operadores**

**Definicion dirigidas por la sintaxis**

*Asocia a cada símbolo de la gramática un conjunto de atributos y a cada producción, un conjunto de reglas semánticas para calcular los valores de los atributos asociados con los símbolos que aparecen en esa producción.*

Una traducción es una transformación de una entrada en una salida. La salida para cada entrada *x* se especifica de la forma siguiente. Primero, se construye un árbol de análisis sintáctico para *x.* Supóngase que un nodo *n* del árbol de análisis sintáctico está etiquetado con el símbolo *X* de la gramática. Se escribe *X.a* para indicar el valor del atributo a de X en ese nodo. El valor de *X.a* en *n* se calcula por fa regla semántica para el atributo *a* asociado con la producción de *X* utilizada en el nodo *n. Al árbol de análisis sintáctico que muestre los valores de los atributos en cada nodo se dice que es un árbol de análisis sintáctico con anotaciones.*

**Atributos sintetizados**

*Se dice que un atributo está sintetizado si su valor en un nodo del árbol de análisis sintáctico se determina a partir de los valores de atributos de los hijos del nodo.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Producción | | | Regla Semántica | |
| *expr* -> *expr1 +término*  *térmi* | | | *expr -> expr1.t +término.t*  *= expr¡.t término.t '+'* | |
| *expr -> expr1 - término* | | | *expr -> expr1.t – término.t* | |
| *expr* -> *termino* | | | *expr.t* | = *término.t* |
| *término*-> 0 |  | *término,t* ***= '0'***  **= '0'** | | |
| *término*-> 1 |  | *término.t* ***= '1'***  **= '1'** | | |
| *término ->* 9 |  | *término. t* **= '9'** | | |

|  |  |  |
| --- | --- | --- |
| Producción | Regla Semántica | |
| *sec ->* **comienza** | *sec.x*  *sec. y* | = 0 = 0 |
| *sec -*> *sec instr* | *sec.x sec. y* | = *sec.x + instr.dx = sec.y + instr.dy* |
| *instr -*> **este** | *instr.dx instr.dy* | = 1 = 0 |
| *instr -*> **norte** | *instr.dx instr. dy* | = 0  = 1 |
| *instr -*> **oeste** | *instr.dx i instr.dy* | = -1 = 0 |
| Instr -> **sur** | *instr.dx instr.dy* | = 0  = -1 |

**Recorridos en profundidad**

**procedure** *visita (n:* nodo); **begin**

**for** cada hijo *m* de *n,* de izquierda a derecha **do**

*visita (ni);*

evalúa reglas semánticas en el nodo *n*

**end**

**Fig. 2.10.** Un recorrido en profundidad de un árbol.

Se llama "en profundidad" porque siempre que pueda, visita a un hijo no vi­sitado de un nodo, de modo que intenta visitar los nodos más alejados de la raíz lo antes posible.

**Esquema de traducción**

Es una GLC en la que se encuentran intercalados, en los lados derechos de las producciones, fragmentos de programa llamados *acciones semánticas. E*s como una definición dirigida por la sintaxis, con la excep­ción de que el orden de evaluación de las reglas semánticas se muestra explícita­mente.

*expr -*>*expr* + *término { print* ('+') }

*expr* -> *expr* + *término*  { *print {'-') }*

*expr* -> *término*

*término →* 0 { *print* ('0') }

*término →* 1 { *print ('1')}*

*...*

*término →*  9 { *print* ('9') }

**Análisis sintáctico descendente**

En este la construcción se inicia en la raíz y avanza hacia las hojas

1. En el nodo *n,* etiquetado con el no terminal *A,* selecciónese una de las produc­  
   ciones para *A* y construyase los hijos de *n* para los símbolos del lado derecho de  
   la producción.
2. Encuéntrese el siguiente nodo en el que ha de construirse un subárbol.

*tipo* -> *simple*

| ^ id

| **array** [ *simple ]* of *tipo  
simple* -> **integer** (2.8)

**| char**

**| núm puntopunto núm**

**Análisis sintáctico predictivo**

El *análisis sintáctico descendente recursivo* es un método descendente en el que se ejecuta un conjunto de procedimientos recursivos para procesar la entrada. A cada no terminal de una gramática se asocia un procedimiento.

Un *analizador sintáctico predictivo* es un programa que consiste en un procedi­miento para cada no terminal.

**análisis sintác­tico predictivo**, en el que el símbolo de preanálisis determina sin ambigüedad el pro­cedimiento seleccionado para cada no terminal. La secuencia de procedimientos lla­mados en el procesamiento de la entrada define implícitamente un árbol de análisis sintáctico para la entrada.

**void** *parea* ***(complex t*) {**

**if (***preanálisis* == *t* ***)***

*preanálisis= sigcomplex();*

**else** *error();*

**}**

**void** *tipo()* ***{***

**if (***preanálisis==* ***INTEGER* ||** *preanálisis==* **CHAR** || *preanálisis ==* **NUM** ) {

*simple();*

**else if (***preanálisis* = '^' **)**{

*parea* ('^'); *parea* **(id);**

**}**

**else if (***preanálisis ==* ***ARRAY* ) {**

*parea* **(*ARRAY* );** *parea* ('['); *simple(); parea* (']'); parea **(OF);** tipo();

**}**

**else** error()**;**

**}**

**void** *simple()* ***{***

**if (***preanálisis* == **INTEGER)**

*parea* **(INTEGER ) ;**

**else if (***preanálisis* == **CHAR)**

*parea* **(CHAR);**

**else if (***preanálisis* == **NUM ){**

*parea* **(NUM);** *parea* **(PUNTOPUNTO );** *parea* **(NUM)**

**}**

**else** *error()* **;**

**}**

Seudocódigo de **un analizador sintáctico predictivo.**

El análisis sintáctico predictivo depende de la información sobre los primeros símbolos que pueden ser generados por el lado derecho de una producción. Para precisar, sea a el lado derecho de una producción para el no terminal *A.* Se define PRIMERO(a) como el conjunto de componentes léxicos que opere como los pri­meros símbolos de una o más cadenas generadas a partir de a. Si a es **c** o puede generar e, entonces e también está en PRIMERO(a)2. Por ejemplo,

PRIMERO *(simple) =* {**integer, char, núm** }

PRIMERO (^ **id)** = {**^**}

PRIMERO **(array** [ *simple* ] of *tipo) =* { **array** }

En la práctica, muchos lados derechos de una producción comienzan con compo­nentes léxicos, lo que simplifica la construcción de conjuntos PRIMERO.

Se deben considerar los conjuntos PRIMERO si hay dos producciones *A-> alfa A* -> beta*.* El análisis sintáctico descendente recursivo sin retroceso requiere que PRIMERO(alfa) y PRIMERO(beta) sean disjuntos. El símbolo de preanálisis se puede usar entonces para decidir el tipo de producción a utilizar; si el símbolo de preanálisis está en PRIMERO(alfa), entonces se usa alfa. De otro modo, si el símbolo de pre­análisis está en PRIMERO(beta), entonces se usa beta.

**Cuándo se usan las producciones ε**

El ana­lizador sintáctico descendente recursivo usará una producción epsilon por defecto cuando no se pueda aplicar otra producción. Por ejemplo, considérese:

*prop* -> begin *props-opc* end

*props-opc* —> *lista-props* | **ε**

Durante el análisis sintáctico de *props-opc,* si el símbolo de preanálisis no está en *PRlMERO(lista-props),* entonces se usa la producción epsilon. Esta elección es justo la correcta si el símbolo de preanálisis es **end.** Cualquier símbolo de preanálisis di­ferente de **end** dará como resultado un error, que se detectará durante el análisis sin­táctico *de prop.*

**Diseño de un analizador sintáctico predictivo**

Cada procedimiento hace dos cosas:

1. Decide la producción que utilizará analizando el símbolo de preanálisis. Si el símbolo de preanálisis está en PRIMERO(a), se usa la producción con lado de­recho a. Si hay un conflicto entre dos lados derechos de cualquier símbolo de preanálisis, entonces en esa gramática no se puede emplear este método de aná-lisis sintáctico. Si el símbolo de preanálisis no está en el conjunto PRIMERO de ningún otro lado derecho, se usa una producción con € en el lado derecho.

2. El procedimiento usa una producción imitando al lado derecho. Un no terminal da como resultado una llamada al procedimiento del no terminal, y un compo­nente léxico que coincida con el símbolo de preanálisis da como resultado que se lea el siguiente componente léxico. Si el componente léxico de la producción no coincide en algún punto con el símbolo de preanálisis, se declara un error. La figura 2.17 es el resultado de aplicar estas reglas a la gramática (2.8).

Al igual que se crea un esquema de traducción extendiendo una gramática, se puede crear un traductor dirigido por la sintaxis extendiendo un analizador sintác­tico predictivo. En la sección 5.5 se da un algoritmo para este propósito. La si­guiente construcción limitada es suficiente por el momento, pues los esquemas de traducción que se implantan en este capítulo no asocian atributos a los no termi­nales:

1. Construyase un analizador sintáctico predictivo, ignorando las acciones en las  
   producciones.
2. Copíense las acciones del esquema de traducción en el analizador sintáctico. Si  
   una acción aparece después del símbolo gramatical *X* en la producción *p,* enton­  
   ces se copia después del código que implanta *X.* De otro modo, si aparece á"l prin­  
   cipio de la producción, entonces se copia justo antes del código que implanta la  
   producción. " ,

En la siguiente sección se construirá un traductor de este tipo.

**Recursividad por la izquierda**

Es posible que un analizador sintáctico descendente recursivo entre en un lazo (bu­cle) indefinido.

Hay un problema con producciones recursivas por la izquierda del tipo

*expr* -» *expr + término*

en la que el símbolo situado más a la izquierda del lado derecho es el mismo que el no terminal del lado izquierdo de la producción.

Para eliminar la recursividad por la izquierda las producciones

A → Aa | b(donde la a representa alfa y la b beta)

se transforman en las producciones

A → b | bR

R → aR | **ε**

**CAPITULO 3**

**AUTOMATA FINITO**

**Autómata finito no determinista**

Un autómata finito no determinista NFA consiste en :

1. Un conjunto de estados finito S .

2. Un conjunto de símbolos de entrada sigma ( el alfabeto de entrada ).

3. Una función de transición mueve que transforma pares estado-símbolo en

conjuntos de estados.

4. Un estado de s0 que se considera el estado de inicio (o inicial) .

5. Un conjunto de estados F considerados como estados de aceptación (o finales).

|  |  |  |
| --- | --- | --- |
| estado | símbolos de entrada | |
| a | b |
| 0 | { 0 , 1 } | { 0 } |
| 1 |  | { 2 } |
| 2 |  | { 3 } |
| 3 |  |  |

tabla de transición del NFA

**Autómatas finitos deterministas**

Un autómata finito determinista DFA es un caso especial de un NFA donde

1. ningún estado tiene una transición epsilon, es decir una transición con la entrada epsilon.

2. Para cada estado s y cada símbolo de entrada a, hay a lo sumo una arista etiquetada a que sale de s.

**Simulación de un DFA .**

ENTRADA : Una cadena de entrada x que termina con un carácter de fin de archivo. Un DFA D con el estado de inicio s0, y un conjunto de estados de aceptación F y una función de transición.

SALIDA : respuesta “sí”, si D acepta x, “no" de otro modo .

MÉTODO : Aplicar el algoritmo de la figura . 3,27 para la cadena de entrada x . La función mueve(s,c) da el estado al cual hay una arista desde el estado s en la entrada c . La función sigtecar devuelve el siguiente carácter de la cadena de entrada x .

s = s0 ;

c = sigtecar();

while (c != eof ) {

s = mueve(s, c); c = sigtecar() ;

}

if (s is in F)

return "yes" ;

else return "no" ;

Algoritmo 2: Construcción de subconjuntos. Construcción de un AFD a partir de un AFN.

Entrada. Un AFN N.

Salida. Un AFD D que acepta el mismo lenguaje.

Método. El algoritmo construye una tabla de transiciones tranD para D. Cada estado del AFD es un conjunto de estados del AFN y se construye tranD de modo que D simulará “en paralelo” todos los posibles movimientos que N puede realizar con una determinada cadena de entrada.

Operaciones sobre los estados de un AFN

|  |  |
| --- | --- |
| OPERACION | DESCRIPCIÓN |
| cerradura- epsilon (s) | Conjunto de estados del AFN alcanzables desde el estado s del AFN con transiciones epsilon solamente |
| cerradura- epsilon (T) | Conjunto de estados del AFN alcanzables desde algún estado s en T con transiciones epsilon solamente. |
| mover(T, a) | Conjunto de estados del AFN hacia los cuales hay una transición con el símbolo de entrada a desde algún estado s en T del AFN. |

al inicio cerradura- epsilon(s0) es el único estado dentro de estadosD y no esta marcado;

while (haya un estado no marcado T en estadosD){

marcar T;

for(cada símbolo de entrada a){

U:= cerradura-epsilon(mueve(T, a));

if U no esta en estadosD then

añadir U como estado no marcado a estadosD;

tranD(T, a):= U

}

}

Figura 4. La construcción de subconjuntos

meter todos estados de T en pila;

inicializar cerradura-epsilon(T) a T;

while (pila no este vacía) {

sacar t , el elemento tope de pila ;

for(cada estado u con una arista desde t a u etiquetada con epsilon)

if (u no esta en cerradura-epsilon (T)){

añadir u a cerradura-epsilon(T) ;

meter u en pila ;

}

}

Calculando la cerradura-epsilon

Algoritmo:construcción de un AFN a partir de una expresión regular

Entrada: Una expresión regular r en un alfabeto S

Salida: Un AFN N que acepte L(r)

Para vacío, construir el AFN

Para a del alfabeto, construir el AFN

Supóngase que N(s) y N(t) son AFN para las expresiones regulares s y t

Para las expresiones regular s | t, constrúyase el siguiente AFN compuesto N(s | t )

Para la expresión regular st, constrúyase el AFN, compuesto N(st)

Para la expresión regular s\*, constrúyase el AFN compuesto N(s\*)

Para la expresión regular entre paréntesis (s), utilícese N(s) como AFN.

Cada vez que se construye un nuevo estado, se le da un nombre distinto. De esta forma, no puede haber dos estados de un AFN componentes con el mismo nombre. Aunque aparezcan los mismos símbolos varias veces en r, se crea para cada ejemplo ese símbolo un AFN aparte con sus propios estados.

Algoritmo 3.22 : Simulación de un AFN .

ENTRADA : Una cadena de entrada x terminada por un carácter de final de archivo eof. un AFN N con un estado de inicio s0 , y un conjunto de estados de aceptancion F.

SALIDA : respuesta “sí" si N acepta x ; “no" de otro modo .

MÉTODO :

S := cerradura-epsilon({s0});

a = sigtecar();

while( a != eof ) {

S := cerradura-epsilon(mueve(S,a));

a = sigtecar();

}

if ( S intserseccion F != vacio ) return "si" ;

else return "no" ;

Figure 3.37: Simulando un AFN