Contenido

4.1 ANÁLISIS SINTÁCTICO DESCENDENTE MEDIANTE MÉTODO DESCENDENTE RECURSIVO

4.1.1 El método básico descendente recursivo

La idea del análisis sintáctico descendente recursivo es muy simple. Observamos la regla gramatical para un no terminal *A* como una definición para un procedimiento que reconocerá una *A*. El lado derecho de la regla gramatical para *A* especifica la estructura del código para este procedimiento: la secuencia de terminales y no terminales en una selección corresponde a concordancias de la entrada y llamadas a otros procedimientos, mientras que las selecciones corresponden a las alternativas (sentencias case o if) dentro del código.

Como primer ejemplo consideremos la gramática de expresión del capítulo anterior:

exp → *exp opsuma term* | *term*
opsuma → + | -
term → *term opmult factor* | *factor*
opmult → *
factor → (*exp*) | **número**

y consideremos la regla gramatical para un *factor*. Un procedimiento descendente recursivo que reconoce un *factor* (y al cual llamaremos por el mismo nombre) se puede escribir en pseudocódigo de la manera siguiente:

```
procedure factor ;
begin
  case token of
  ( :   match( ( ) ;
        exp ;
        match( ) ) ;
  number :
        match(number) ;
  else error ;
  end case ;
end factor ;
```

En este pseudocódigo partimos del supuesto de que existe una variable *token* que mantiene el siguiente token actual en la entrada (de manera que este ejemplo utiliza un símbolo de búsqueda hacia delante). También dimos por hecho que existe un procedimiento *match* que compara el token siguiente actual con su parámetro, avanza la entrada si tiene éxito y declara un error si no lo tiene:

---

1. Estos conjuntos también son necesarios en algunos de los algoritmos para el análisis sintáctico ascendente que se estudian en el capítulo siguiente.

```
procedure match ( expectedToken ) ;
begin
  if token = expectedToken then
    getToken ;
  else
    error ;
  end if ;
end match ;
```

Por el momento dejaremos sin especificar el procedimiento *error* que es llamado tanto por *match* como por *factor*. Se puede suponer que se imprime un mensaje de error y se sale del programa.

Advierta que en las llamadas *match*( ) y *match*(`number`) en *factor*, se sabe que las variables *expectedToken* y *token* son las mismas. Sin embargo, en la llamada *match*( ), no se puede suponer que *token* sea un paréntesis derecho, de manera que es necesaria una prueba. El código para *factor* también supone que se ha definido un procedimiento *exp* que puede llamarlo. En un analizador sintáctico descendente recursivo para la gramática de expresión el procedimiento *exp* llamará a *term*, el procedimiento term llamará a *factor* y el procedimiento *factor* llamará a *exp*, de manera que todos estos procedimientos deben ser capaces de llamarse entre sí. Desgraciadamente, escribir procedimientos descendentes recursivos para las reglas restantes en la gramática de expresión no es tan fácil como para *factor* y requiere el uso de EBNF, como veremos a continuación.

4.1.2 Repetición y selección: el uso de EBNF

Considere como un segundo ejemplo la regla gramatical (simplificada) para una sentencia if:

sent-if → `if` (*exp*) *sentencia*
| `if` (*exp*) *sentencia* `else` *sentencia*

Esto se puede traducir en el procedimiento

```
procedure ifStmt ;
begin
   match (if) ;
   match ( ( ) ;
   exp ;
   match ( ) ) ;
   statement ;
   if token = else then
      match (else) ;
      statement ;
   end if ;
end ifStmt ;
```

En este ejemplo podríamos no distinguir de inmediato las dos selecciones en el lado derecho de la regla gramatical (ambas comienzan con el token `if`). En su lugar, debemos aplazar la decisión acerca de reconocer la parte-else opcional hasta que veamos el token `else` en la entrada. De este modo, el código para la sentencia if corresponde más a la EBNF

sent-if → `if` (*exp*) *sentencia* [`else` *sentencia*]

que a la BNF, donde los corchetes de la EBNF se traducen en una prueba en el código para *Sent-if*. De hecho, la notación EBNF está diseñada para reflejar muy de cerca el código real de un analizador sintáctico descendente recursivo, de modo que una gramática debería siempre traducirse en EBNF si se está utilizando el modo descendente recursivo. Advierta también que, aun cuando esta gramática es ambigua (véase el capítulo anterior), es natural escribir un analizador sintáctico que haga concordar cada token `else` tan pronto como se encuentre en la entrada. Esto corresponde precisamente a la regla de eliminación de ambigüedad más cercanamente anidada.

Considere ahora el caso de una *exp* en la gramática para expresiones aritméticas simples en BNF:

$$exp \rightarrow exp\ opsuma\ term \mid term$$

Si estamos intentando convertir esto en un procedimiento *exp* recursivo de acuerdo con nuestro plan, lo primero que podemos intentar hacer es llamar a *exp* mismo, y esto conduciría a un inmediato ciclo recursivo infinito. Tratar de probar cuál selección usar (*exp* → *exp opsuma term*, o bien, *exp* → *term*) es realmente problemático, ya que tanto *exp* como *term* pueden comenzar con los mismos tokens (un número o paréntesis izquierdo).

La solución es utilizar la regla de EBNF

$$exp \rightarrow term\ \{\ opsuma\ term\ \}$$

en su lugar. Las llaves expresan la repetición que se puede traducir al código para un ciclo o bucle de la manera siguiente:

```
procedure exp ;
begin
  term ;
  while token = + or token = - do
    match (token) ;
    term ;
  end while ;
end exp ;
```

De manera similar, la regla EBNF para *term*:

$$term \rightarrow factor\ \{\ opmult\ factor\ \}$$

se convierte en el código

```
procedure term ;
begin
  factor ;
  while token = * do
    match (token) ;
    factor ;
  end while ;
end term ;
```

Aquí eliminamos los no terminales *opsuma* y *opmult* como procedimientos separados, ya que su única función es igualar a los operadores:

$$opsuma \rightarrow + \mid -$$
$$opmult \rightarrow *$$

En su lugar haremos esta concordancia dentro de *exp* y *term*.

Una cuestión que surge con este código es si la asociatividad izquierda implicada por las llaves (y explícita en la BNF original) todavía puede mantenerse. Suponga, por ejemplo, que deseamos escribir una calculadora descendente recursiva para la aritmética entera simple de nuestra gramática. Podemos asegurarnos de que las operaciones son asociativas por la izquierda realizando las operaciones a medida que circulamos a través del ciclo o bucle (suponemos ahora que los procedimientos de análisis sintáctico son funciones que devuelven un resultado entero):

```
function exp : integer ;
var temp : integer ;
begin
  temp := term ;
  while token = + or token = - do
    case token of
      + : match (+) ;
          temp := temp + term ;
      - : match (-) ;
          temp := temp - term ;
    end case ;
  end while ;
  return temp ;
end exp ;
```

y de manera similar para *term*. Empleamos estas ideas para crear una calculadora simple funcionando, cuyo código en C está dado en la figura 4.1 (páginas 148-149), donde en lugar de escribir un analizador léxico completo, optamos por utilizar llamadas a `getchar` y `scanf` en lugar de un procedimiento `getToken`.

Este método de convertir reglas gramaticales en EBNF a código es muy poderoso, y lo utilizaremos para proporcionar un analizador sintáctico completo para el lenguaje TINY de la sección 4.4. Sin embargo, existen algunas dificultades y debemos tener cuidado al programar las acciones dentro del código. Un ejemplo de esto está en el pseudocódigo anterior para *exp*, donde tiene que ocurrir una concordancia de las operaciones antes de las llamadas repetidas a *term* (de otro modo *term* vería una operación como su primer token, lo que generaría un error). En realidad, el protocolo siguiente para conservar la variable global *token* actual debe estar rígidamente adherido: *token* debe estar establecido antes que comience el análisis sintáctico y *getToken* (o su equivalente) debe ser llamado precisamente después de haber probado con éxito un token (ponemos esto en el procedimiento *match* en el pseudocódigo).

También debe tenerse el mismo cuidado al programar las acciones durante la construcción de un árbol sintáctico. Vimos que la asociatividad por la izquierda en una EBNF con repetición puede mantenerse para propósitos de cálculo al realizar éste a medida que se ejecuta el ciclo.

Figura 4.1 (inicio)
Una calculadora descendente recursiva para aritmética entera simple

```
/* Calculadora de aritmética entera simple según el EBNF:

   <exp> -> <term> { <opsuma> <term> }
   <opsuma> -> + | -
   <term> -> <factor> { <opmult> <factor> }
   <opmult> -> *
   <factor> -> ( <exp> ) | Número

   Introduce una línea de texto desde stdin
   Sale "Error" o el resultado.
*/

#include <stdio.h>
#include <stdlib.h>

char token; /* variable token global */

/* prototipos de función para llamadas recursivas */
int exp(void);
int term(void);
int factor(void);

void error(void)
{ fprintf(stderr,"Error\n");
  exit(1);
}

void match( char expectedToken)
{ if (token==expectedToken) token = getchar();
  else error();
}

main()
{ int result;
  token = getchar(); /* carga token con el primer carácter
                        para búsqueda hacia delante */
  result = exp();
  if (token=='\n') /* verifica el fin de línea */
    printf("Result = %d\n",result);
  else error(); /* caracteres extraños en línea */
  return 0;
}
```

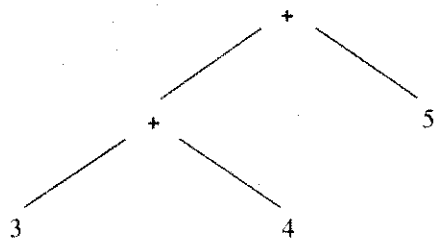
Figura 4.1 (conclusión)
Una calentadora descendente recursiva para aritmética entera simple

```
int exp(void)
{ int temp = term();
  while ((token=='+')||(token=='-'))
    switch (token) {
    case '+': match('+');
              temp+=term();
              break;
    case '-': match('-');
              temp-=term();
              break;
    }
  return temp;
}

int term(void)
{ int temp = factor();
  while (token=='*') {
    match('*');
    temp*=factor();
  }
  return temp;
}

int factor(void)
{ int temp;
  if (token=='(') {
    match('(');
    temp = exp();
    match(')');
  }
  else if (isdigit(token)) {
    ungetc(token,stdin);
    scanf("%d",&temp);
    token = getchar();
  }
  else error();
  return temp;
}
```

Sin embargo, esto ya no corresponde a una construcción descendente del árbol gramatical o sintáctico. En realidad, si consideramos la expresión `3+4+5`, con árbol sintáctico



el nodo que representa la suma de 3 y 4 debe ser creado (o procesado) antes del nodo raíz (el nodo que representa su suma con 5). Al traducir esto a la construcción de árbol sintáctico real, tenemos el siguiente pseudocódigo para el procedimiento *exp*:

```
function exp : syntaxTree ;
var temp, newtemp : syntaxTree ;
begin
  temp := term ;
  while token = + or token = - do
    case token of
     + : match (+) ;
         newtemp := makeOpNode(+) ;
         leftChild(newtemp) := temp ;
         rightChild(newtemp) := term ;
         temp := newtemp ;
     - : match (-) ;
         newtemp := makeOpNode(-) ;
         leftChild(newtemp) := temp ;
         rightChild(newtemp) := term ;
         temp := newtemp ;
    end case ;
  end while ;
  return temp ;
end exp ;
```

o la versión más simple

```
function exp : syntaxTree ;
var temp, newtemp : syntaxTree ;
begin
  temp := term ;
  while token = + or token = - do
    newtemp := makeOpNode(token) ;
    match (token) ;
    leftChild(newtemp) := temp ;
    rightChild(newtemp) := term ;
    temp := newtemp ;
  end while ;
  return temp ;
end exp ;
```

En este código utilizamos una nueva función *makeOpNode*, que recibe un token de operador como un parámetro y devuelve un nodo de árbol sintáctico recién construido. También indicamos la asignación de un árbol sintáctico *p* como un hijo a la izquierda o a la derecha de un árbol sintáctico *t* al escribir *leftChild*(*t*) := *p* o *rightChild*(*t*) := *p*. Con este pseudocódigo el procedimiento *exp* en realidad construye el árbol sintáctico y no el árbol de análisis gramatical. Esto se debe a que una llamada a *exp* no construye invariablemente un nuevo nodo del árbol; si no hay operadores, *exp* simplemente pasa de regreso al árbol recibido desde la llamada inicial a *term* como su propio valor. También se puede escribir el pseudocódigo correspondiente para *term* y *factor* (véanse los ejercicios).

En contraste, el árbol sintáctico para una sentencia if se puede construir de manera estrictamente descendente por medio de un analizador sintáctico descendente recursivo:

```
function ifStatement : syntaxTree ;
var temp : syntaxTree ;
begin
   match (if) ;
   match ( ( ) ;
   temp := makeStmtNode(if) ;
   testChild(temp) := exp ;
   match ( ) ) ;
   thenChild(temp) := statement ;
   if token = else then
      match (else) ;
      elseChild(temp) := statement ;
   else
      elseChild(temp) := nil ;
   end if ;
end ifStatement ;
```

La flexibilidad del análisis sintáctico descendente recursivo, al permitir al programador ajustar la programación de las acciones, lo hace el método de elección para analizadores sintácticos generados a mano.

4.3 Otros problemas de decisión

El método descendente recursivo que describimos es muy poderoso, pero todavía es de propósito específico. Con un lenguaje pequeño cuidadosamente diseñado (tal como TINY o incluso C), estos métodos son adecuados para construir un analizador sintáctico completo. Deberíamos estar conscientes de que pueden ser necesarios más métodos formales en situaciones complejas. Pueden surgir diversos problemas. En primer lugar, puede ser difícil convertir una gramática originalmente escrita en BNF en forma EBNF. Una alternativa para el uso de la EBNF se estudia en la siguiente sección, donde se construye una BNF transformada que en esencia es equivalente a la EBNF. En segundo lugar, cuando se formula una prueba para distinguir dos o más opciones de regla gramatical

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

puede ser difícil decidir cuándo utilizar la selección $A \rightarrow \alpha$ y cuándo emplear la selección $A \rightarrow \beta$, si tanto α como β comienzan con no terminales. Un problema de decisión de esta naturaleza requiere el cálculo de los conjuntos **Primero** de α y β: el conjunto de tokens que pueden iniciar legalmente cada cadena. Formalizaremos los detalles de este cálculo en la sección 4.3. En tercer lugar, al escribir el código para una producción ε

$$A \rightarrow \varepsilon$$

puede ser necesario conocer cuáles tokens pueden aparecer legalmente después del no terminal A, puesto que tales tokens indican que A puede desaparecer de manera apropiada en este punto del análisis sintáctico. Este conjunto se llama conjunto **Siguiente** de A. El cálculo de este conjunto también se precisará en la sección 4.3.

Una inquietud final que puede requerir el cálculo de los conjuntos Primero y Siguiente es la detección temprana de errores. Considere, por ejemplo, el programa calculadora de la figura 4.1. Dada la entrada `)3-2)`, el analizador sintáctico descenderá desde **`exp`** hasta **`term`** hasta **`factor`** antes de que se informe de un error, mientras que se podría declarar el error ya en **`exp`**, puesto que un paréntesis derecho no es un primer carácter legal en una expresión. El conjunto Primero de *exp* nos diría esto, lo que permitiría una detección más temprana del error. (La detección y recuperación de errores se analiza con más detalle al final del capítulo.)

Tabla 4.1
Acciones de análisis sintáctico de un analizador sintáctico descendente

	Pila de análisis sintáctico	Entrada	Acción
1	\$ S	( ) \$	$S \rightarrow$ (S) S
2	\$ S) S (	( ) \$	match (concordar)
3	\$ S) S	) \$	$S \rightarrow \varepsilon$
4	\$ S)	) \$	match (concordar)
5	\$ S	\$	$S \rightarrow \varepsilon$
6	\$	\$	aceptar