Análisis sintáctico descendente recursivo

Existen dos clases de algoritmos para implementar el análisis sintáctico de una frase. Los analizadores sintácticos ascendentes tratan de construir el árbol de análisis sintáctico de la entrada comenzando en las hojas y progresando hacia la raíz mientras que los analizadores sintácticos descendentes construyen el árbol de análisis sintáctico partiendo desde la raíz y avanzando hacia las hojas. Ambos tipos de algoritmos operan de forma determinista, leen las frases de entrada (programas) de izquierda a derecha considerando un símbolo cada vez.

Los analizadores ascendentes comienzan la construcción del árbol sintáctico con los tokens de la frase a analizar y tratan de hallar la estructura sintáctica para la secuencia más larga de la entrada. Esto sucede con las gramáticas independientes del contexto cuando se encuentra el lado derecho de una regla de producción. En ese momento se sustituye la secuencia por la variable sintáctica del lado izquierdo, y esta acción se denomina una reducción. El análisis ascendente alterna acciones de leer el siguiente token de la entrada y la realización de reducciones. El análisis ascendente finaliza cuando se ha leído toda la entrada y se ha reducido por el símbolo de arranque.

Los analizadores sintácticos descendentes comienzan la construcción del árbol de análisis sintáctico partiendo del símbolo de arranque de la gramática, que constituye la raíz del árbol de análisis sintáctico. Este tipo de analizadores predicen cómo es la frase de entrada (programa) o parte de ella y tratan de confirmarlo. Técnicamente esta predicción es una forma sentencial izquierda de la gramática. La primera predicción consta solamente del símbolo de arranque de la gramática. Supongamos que alguna parte de la predicción se confirma por los tokens de la entrada que ya han sido leídos. Si la parte sin confirmar comienza con una variable sintáctica, el analizador descendente escogerá una producción para esa variable utilizando el siguiente token de la entrada sin consumir y genera una nueva predicción. Si la predicción actual comienza con un token, el analizador lo compara con el siguiente token de la entrada. Si coinciden, se confirma otro token de la predicción, en caso contrario se produce un error. El análisis descendente finaliza cuando toda la entrada ha sido confirmada.

En este capítulo nos restringiremos a los analizadores sintácticos descendentes recursivos, que son un tipo particular de analizadores descendentes. La popularidad de los analizadores sintácticos descendentes se debe al hecho de que posibilitan la construcción manual (sin ayuda de herramientas) de analizadores sintácticos eficientes con mayor facilidad. La construcción descendente de un árbol de análisis sintáctico se hace empezando por la raíz, etiquetada con el símbolo de arranque de la gramática y realizando de forma repetida los dos pasos siguientes:

- En el nodo n, etiquetado con el no terminal A, seleccionar una producción para A y construir los hijos de n para los símbolos del lado derecho de la producción.
- Encontrar el siguiente nodo en el que se ha de construir el subárbol.

Para algunas gramáticas, los pasos anteriores se aplican durante un examen sencillo de izquierda a derecha de la cadena de entrada. El componente léxico en curso analizado en la entrada se denomina símbolo de preanálisis (lookahead). Inicialmente el símbolo de preanálisis es el componente léxico (token) situado más a la izquierda en la cadena de entrada.

Los analizadores sintácticos descendentes recursivos utilizan una serie de procedimientos recursivos para modelizar el árbol de análisis sintáctico que han de construir. Se escribirá un procedimiento para cada símbolo no terminal de la gramática. La secuencia de procedimientos llamados en el procesamiento de la entrada define implícitamente un árbol de análisis sintáctico para la entrada. El símbolo de preanálisis determinará sin ambigüedad la producción que ha de seleccionarse para cada no terminal. Para que este tipo de análisis sea posible, la gramática independiente del contexto subyacente ha de cumplir ciertas restricciones.

El análisis sintáctico predictivo depende de la información sobre los primeros símbolos que pueden ser generados a partir del lado izquierdo de una producción (los firsts). Se deben considerar

los conjuntos first del siguiente modo: si hay dos producciones A $\rightarrow \alpha$ y A $\rightarrow$ ß el análisis sintáctico descendente recursivo sin retroceso requiere que FIRST(α) y FIRST(ß) sean disjuntos. Si esta condición se cumple, entonces el símbolo de preanálisis se puede usar para decidir la producción a utilizar:

- Si el símbolo de preanálisis está en FIRST(α) entonces se usa
  $A \rightarrow \alpha$.
- De otro modo, si el símbolo de preanálisis está en FIRST(ß), entonces se usa $A \rightarrow$ ß.

Las producciones con el vacío del lado derecho necesitan un tratamiento especial: el analizador sintáctico descendente recursivo usará la producción ε cuando no se pueda aplicar otra producción.

Por ejemplo considérese la siguiente gramática para sentencias:

Stmt $\rightarrow$ **begin** OptStmts **end**
OptStmts $\rightarrow$ StmtList | ε

Durante el análisis sintáctico de OptStmts si el símbolo de preanálisis no está en FIRST(StmtList) entonces se usa una producción ε. Esta elección es justo la correcta si el símbolo de preanálisis es **end**. Cualquier otro símbolo dará como resultado un error, que se detectará durante el análisis sintáctico de Stmt.

Un analizador sintáctico predictivo recursivo es un programa que consiste en un procedimiento para cada símbolo no terminal de la gramática. Cada procedimiento realiza dos acciones:

1. Decide la producción que utilizará (de entre las posibles para el no terminal considerado) analizando el símbolo de preanálisis. Si el símbolo de preanálisis está en FIRST(α), se usa la producción con lado derecho α. Si hay un conflicto entre dos lados derechos de cualquier símbolo de preanálisis, entonces esa gramática no es apta para emplear este método de análisis sintáctico. Si el símbolo de preanálisis no está en el conjunto FIRST de ningún lado derecho, se usa una producción con ε en el lado derecho (si existe una producción así en la gramática).
2. Utiliza una producción imitando al lado derecho. Un no terminal provoca una llamada al procedimiento correspondiente, y un token que coincida con el símbolo de preanálisis da como resultado que se lea el siguiente token. Si el token de la producción no coincide en algún punto con el símbolo de preanálisis se notifica un error y se abandona el análisis de la entrada.

Hay un problema con producciones recursivas por la izquierda del tipo Expr $\rightarrow$ Expr + Term en las que el símbolo situado más a la izquierda del lado derecho es el mismo que el no terminal del lado izquierdo de la producción. Este tipo de producciones provocan que un analizador sintáctico descendente recursivo entre en un bucle infinito pues el procedimiento correspondiente a Expr se llama recursivamente a sí mismo dado que el símbolo de preanálisis cambia sólo cuando coincide con el terminal del lado derecho. Como la producción comienza con el no terminal Expr no se realiza ningún cambio en la entrada entre llamadas recursivas, lo que provoca el bucle infinito. Las gramáticas que contienen producciones como la anterior se dicen recursivas por la izquierda y tampoco son aptas para un análisis descendente recursivo.

Consideremos a modo de ejemplo la gramática de las expresiones (habiendo eliminado la recursividad por la izquierda):

E $\rightarrow$ T E′
E′ $\rightarrow$ **+** T E′ | ε
T $\rightarrow$ F T′
T′ $\rightarrow$ ***** F T′ | ε
F $\rightarrow$ **(** E **)** | **id**

Un pseudocódigo para los procedimientos que realizan el análisis de los no terminales E y E′ tendría la forma que se muestra en la Figura 4.1.

```
procedure E
call T
call E'
print("E")
end

procedure E'
if lookahead = + then
begin
        print("+")
        match(+)
        call T
        call E'
        end;
else
        ;
end;
```

Figura 4.1 El pseudocódigo para los procedimientos E y E'

Los procedimientos para T y T' son similares ya que también lo son las producciones para estos símbolos gramaticales. La Figura 4.2 presenta el pseudocódigo para el procedimiento F.

```
procedure F
case lookahead
        (:
                print("(")
                match(()
                call E
                match())
        id:
                print("id")
                match(id)
        default:
                call Error
end;
```

Figura 4.2 El procedimiento F

En estos pseudocódigos, la variable *lookahead* representa el token producido por el analizador léxico. Normalmente se trata de una constante numérica. Se define una constante diferente para cada uno de los símbolos terminales de la gramática. El analizador léxico se encarga de leer cadenas de caracteres, y cuando detecta un token, lo entrega al analizador sintáctico (devuelve el número asociado al token). Así, por ejemplo, en la gramática de Pascal hay un token que es la palabra reservada *program*. Cuando el analizador léxico detecta en la entrada los caracteres que componen la cadena *program*, devuelve el token (número) asociado.

Las llamadas a *print* que aparecen en los códigos anteriores por supuesto son opcionales e imprimirán (al revés) una derivación a izquierdas de la frase (programa) analizada.

El procedimiento *error* tendría el efecto de finalizar el análisis de la entrada, notificando que se ha producido un error e indicando que el programa no es correcto. Por último el procedimiento *match()* se encarga de comprobar que el token que se está procesando (*lookahead*) coincide con el que se esperaba. La Figura 4.3 presenta un código en Pascal para esta función. En ese código, la función *yylex()* corresponde al analizador léxico.

```
procedure match(t: tokens);
begin
if lookahead = t then
   lookahead := yylex;
else
   error;
end;
```

Figura 4.3 El procedimiento match()

Práctica 4.2. Un Analizador sintáctico para programas Batch de DOS

El objetivo de la práctica consiste en implementar en C un analizador sintáctico predictivo recursivo que reconozca programas de ejecución por lotes (*.BAT) del sistema operativo MS-DOS, indicando si son sintácticamente correctos o no.

El analizador léxico a utilizar con la práctica debe ser el proporcionado por la función *yylex()* que se encuentra contenida en la librería *batch.a*. El prototipo de la función yylex es:

int yylex(void);

Esta función devuelve un entero correspondiente al token que ha leído. La relación entre dicho número y el componente léxico de que se trata se halla descrita en el fichero

ftp://ftp.csi.ull.es/pub/asignas/AUTOMALF/p04_batch/batch.h

que se puede obtener en el servidor ftp del CSI. Para facilitar la legibilidad en este fichero se declaran constantes (#define) para la mayoría de los tokens, con lo que incluyendo este fichero en su práctica podrá hacer uso de ellas.

La Figura 4.4 muestra el contenido del fichero batch.h Aparte de los tokens que aparecen listados en la Figura 4.4 (y que se definen en el fichero batch.h), la función *yylex()* también reconoce los caracteres paréntesis (abierto y cerrado) y arroba: '(', ')' y '@', para los cuales devuelve el entero correspondiente a su código ASCII. También cuando *yylex()* lee un indicador de final de fichero devuelve el número 0.

Si no se está seguro de qué tokens devuelve un determinado fichero de entrada batch, puede utilizarse (una vez compilado) el programa

ftp://ftp.csi.ull.es/pub/asignas/AUTOMALF/p04_batch/batchl.c

que permite introducir por teclado líneas e indica por pantalla los tokens que devuelve *yylex()*.

```
/* Tokens del lenguaje: */
#define ECHO_ 257      /* Instrucción ECHO junto al mensaje asociado */
#define PAUSE_ 258     /* Instrucción PAUSE junto al mensaje asociado */
#define REM_ 259       /* Instrucción REM y comentario asociado */
#define GOTO_ 260      /* Instrucción GOTO junto a la etiqueta */
#define IF 261         /* Palabra clave IF */
#define NOT 262        /* Palabra clave NOT */
#define ERRORLEVEL 263 /* Palabra clave ERRORLEVEL */
#define EXIST 264      /* Palabra clave EXIST */
#define FOR_VAR 265    /* Palabra clave FOR más la variable */
                       /* Asociada a éste (p. ej. FOR %%A)    */
#define IN 266         /* Palabra clave IN */
#define DO 267         /* Palabra clave DO */
#define CALL 268       /* Palabra clave CALL */
#define SHIFT 269      /* Palabra clave SHIFT */
#define IGUALIGUAL 270 /* Dos iguales seguidos (==) */
#define NUM 271        /* Un número formado por uno o varios dígitos */
#define UNIDAD 272     /* El nombre de una unidad de disco (A:) */
#define ACC_FICH 273   /* Nombre de fichero, incluyendo o no su ruta */
                       /* de acceso y unidad (ej: A:\DOS\SORT.EXE) */
                       /* o variables de entorno (ej: %PATH%) */
#define ACC_FICH_COM 274       /* Ficheros igual al anterior, pero */
                               /* pueden tener caracteres comodín /*
                               /* (por ej.: A:*.D?T) */
#define CADENA_SIMPLE 275      /* Una cadena de caracteres que no */
                               /* constituye un nombre de fichero */
                               /* (ej: no/soy/fich) */
#define CADENA_COMILLAS 276   /* Una cadena de caracteres entre */
                               /* comillas dobles (ej: "mas o menos”) */
#define ETIQUETA_ 277          /* Etiquetas asociadas a los saltos, */
                               /* es decir, ':' más una cadena */
                               /* (ej: :final) */
#define EOL 278                /* Un final de linea (\n)*/
#define DESCONOCIDO 279/* Carácter desconocido */

#ifndef YYSTYPE
        #define YYSTYPE int
#endif

extern YYSTYPE yyval, yylval;
```

Figura 4.4 El contenido del fichero batch.h

El analizador léxico que se ha descrito ha sido construido mediante una herramienta para generar analizadores léxicos (lex). La función *yylex()* debe declararse como externa:

extern int yylex();

La función *yylex()* acepta caracteres desde la entrada estándar (stdin), por lo que si se desea probar la implementación que se realice con algún programa batch se deberá redireccionar la entrada hacia nuestro programa (< *fich*). En el servidor ftp del CSI se pueden obtener varios programas batch para comprobar el funcionamiento de la aplicación que se diseñe:

ftp://ftp.csi.ull.es/pub/asignas/AUTOMALF/p04_batch/*.bat

La Figura 4.5 muestra en notación E-BNF la gramática de los programas batch que se deben reconocer. En la gramática se ha utilizado letra negrita para los símbolos terminales (tokens).

Prog_Batch →	Linea Prog_Batch | **-**
Linea →	**'@'** Instruccion | Instruccion
Instruccion →	Ins_Echo | Ins_Pause | Ins_Rem | Ins_Goto | Ins_If | Ins_For | Ins_Call | Ins_Shift | Comando_DOS | Ins_Etiqueta
Ins_Echo →	**ECHO_ EOL**
Ins_Pause →	**PAUSE_ EOL**
Ins_Rem →	**REM_ EOL**
Ins_Goto →	**GOTO_ EOL**
Ins_If →	**IF** Negacion Condicion Instruccion
Negacion →	**NOT** | **-**
Condicion →	Cadena **IGUALIGUAL** Cadena | **ERRORLEVEL NUM** | **EXIST ACC_FICH**
Ins_For →	**FOR_VAR IN '('** Conj_Fich List_Conj_Fich **')' DO** Instruccion
List_Conj_Fich →	Conj_Fich List_Conj_Fich | **-**
Conj_Fich →	**ACC_FICH** | **ACC_FICH_COM**
Ins_Call →	**CALL** Comando_DOS
Ins_Shift →	**SHIFT EOL**
Comando_DOS →	**ACC_FICH** List_Cadena **EOL** | **UNIDAD EOL**
List_Cadena →	Cadena List_Cadena | **-**
Cadena →	**CADENA_SIMPLE** | **CADENA_COMILLAS** | **ACC_FICH** | **ACC_FICH_COM** | **UNIDAD** | **NUM**
Ins_Etiqueta →	**ETIQUETA_ EOL**

Figura 4.5 Una gramática para el lenguaje de comandos del DOS

Si el analizador consigue reconocer la entrada deberá imprimir en pantalla un mensaje que así lo indique, mientras que si por el contrario se produce algún error durante el análisis de la entrada, también se emitirá un mensaje, indicando el número de línea donde se produjo el error.

En el servidor ftp del CSI se encuentra el programa para Unix

ftp://ftp.csi.ull.es/pub/asignas/AUTOMALF/p04_batch/batch

que tiene el comportamiento que se pretende. Como referencia de consulta para el desarrollo de esta práctica se recomienda [Aho90].