Práctica 4

Análisis LALR para "miLenguaje" y construcción de un traductor de "miLenguaje"

1. Objetivos

1. Hacer una introducción práctica a la traducción dirigida por la sintaxis basada en el análisis LALR
2. Introducirse en aspectos semánticos de la traducción
3. Practicar el uso de Bison

2. Contenidos

Habitualmente, el objetivo final de un compilador (o más genéricamente, un traductor) es la generación de un código equivalente transformado. En esta práctica se propone generar código Ada equivalente a un fuente de "miLenguaje", de manera que, una vez compilado, tendremos un ejecutable para el programa "miLenguaje".

Una forma relativamente sencilla de generar código es que dicho código se encuentre asociado a las producciones que definen la gramática. De ahí el término "traducción dirigida por la sintaxis".

Es posible asociar acciones (fuente C) a cada aparición de un símbolo gramatical en una regla.

Así, la producción Bison

```
Izq: D1 D2 {acciones} ;
```

contiene acciones (instrucciones C) que se ejecutarán justo en el momento anterior a ejecutar la reducción de `D1 D2` por `Izq`, es decir, cuando el analizador detecte que `D1 D2` esté en la parte superior de la pila, ejecutará las acciones y a continuación la reducción.

Es también posible poner acciones entre los símbolos de la derecha de la producción, como en el ejemplo que sigue:

```
Izq: D1 {acciones1}
     D2 {acciones2}
;
```

En este caso, el código anterior es equivalente al siguiente, en el que se ha introducido un *marcador* (un marcador es un no terminal añadido a la gramática. Bison, además del no terminal, introducirá una producción Epsilon a la que se asocian las acciones intercaladas):

```
Izq: D1 MARCADOR
     D2 {acciones2}
;
MARCADOR: {acciones1}
;
```

de manera que las acciones siempre se ejecutan antes de una reducción. También es posible asociar información a un símbolo gramatical (en forma de "atributos") y después acceder a él (para información más detallada, consultar la documentación sobre Bison). Veamos dos ejemplos para ilustrar todo lo anterior.

2.1. Ejemplo 1

Consideremos un analizador léxico capaz de reconocer los tokens `tkCONSTENT` (entero sin signo), `tkESCRIBIR` (lexema "escribir"), `tkOPAS` (:=), `tkIDENTIFICADOR` (letra seguido de letras y/o dígitos) y `tkVAR` (lexema "var"). Asumamos la siguiente gramática, que corresponde a un lenguaje sencillo que maneja variables enteras, asignaciones de expresiones, sumas de expresiones y escritura de variables.

El fuente Bison para el analizador sintáctico puede ser:

```
%{
```

```
#include <stdio.h>
%}
%token tkCONSTENT tkESCRIBIR tkOPAS tkIDENTIFICADOR tkVAR
%left '+'
%%
programa:
   tkVAR declaraciones ';' instrucciones
;
declaraciones:
   declaraciones ',' tkIDENTIFICADOR
|  tkIDENTIFICADOR
;
instrucciones:
   instrucciones instruccion
|  instruccion
;
instruccion:
   instAs  ';'
|  instEs  ';'
;
instAs:
   tkIDENTIFICADOR tkOPAS expr
;
instEs:
   tkESCRIBIR '(' tkIDENTIFICADOR ')'
;
expr:
   expr '+' termino
|  termino
;
termino:
   tkIDENTIFICADOR
|  '(' expr ')'
|  tkCONSTENT
;
%%
int main() {
    yyparse();
```

```
   exit(0);
}
```

Así, por ejemplo, para el siguiente programa

```
var a,b,c;

   a := 25;
   b :=100;
   c := a+b;
   escribir(c);
```

un programa en C, equivalente, podría ser el que sigue (el fuente aparece sin formatear dado que, en un principio, no va a ser leído por un "humano", sino simplemente usado como elemento intermedio para ser compilado con `cc` o `gcc` y poder tener un código ejecutable en una máquina dada):

```
/**************************************
   Autor: Gurb
   Fecha: 1/1/3003
   Versión traductor: 3.141592
   Com.:  Este fuente ha sido generado
          por el compilador de miLenguaje.
**************************************/

#include <stdio.h>
int main(){
int a,b,c;
a=25;
b=100;
c=a+b;
fprintf(stdout,"%d\n",c);
exit(0);
}
```

Para conseguir lo anterior, es suficiente con insertar en la propia definición de la gramática las acciones adecuadas. Para la gramática considerada, una solución podría ser:

```
%{
#include <stdio.h>

void ponCabecera(FILE *f){
    fprintf(f,"/*************************************\n");
    fprintf(f,"   Autor: Gurb                         \n");
    fprintf(f,"   Fecha: 1/1/3003                     \n");
    fprintf(f,"   Versión traductor: 3.141592         \n");
    fprintf(f,"   Com.:  Este fuente ha sido generado  \n");
    fprintf(f,"          por el compilador de ULBS.    \n");
    fprintf(f,"*************************************/\n");
    fprintf(f,"\n\n");
}
%}
%union{
   char texto[255]; /*lexema de tkIDENTIFICADOR y de
                      tkCONSTENT. El analizador léxico
                      asignará el valor adecuado al atributo*/
}
%token tkCONSTENT tkESCRIBIR tkOPAS tkIDENTIFICADOR tkVAR
%type<texto> tkCONSTENT tkIDENTIFICADOR
%left '+'
%%
programa:
      {ponCabecera(stdout);
       fprintf(stdout,"#include<stdio.h>\n");
       fprintf(stdout,"int main(){\n");}
   tkVAR declaraciones  ';' instrucciones
      {fprintf(stdout,"exit(0);\n}\n");}
;
declaraciones:
   declaraciones ',' tkIDENTIFICADOR
      {fprintf(stdout,"int %s;\n",$3);}
| tkIDENTIFICADOR
      {fprintf(stdout,"int %s;\n",$1);}
;
instrucciones:
   instrucciones instruccion
```

```
|  instruccion
;
instruccion:
   instAs ';'
      {fprintf(stdout,";\n");}
|  instEs ';'
      {fprintf(stdout,";\n");}
;
instAs:
   tkIDENTIFICADOR
      {fprintf(stdout,"%s",$1);}
   tkOPAS
      {fprintf(stdout,"=");}
   expr
;
instEs:
   tkESCRIBIR
      {fprintf(stdout, "fprintf(stdout,\"%d\\n\",");}
   '('
   tkIDENTIFICADOR
      {fprintf(stdout,"%s)",$1);}
   ')'
;
expr:
   expr
   '+'
      {fprintf(stdout,"+");}
   termino
|  termino
;
termino:
   tkIDENTIFICADOR
      {fprintf(stdout,"%s",$1);}
|  '('
      {fprintf(stdout,"(");}
   expr
   ')'
      {fprintf(stdout,")");}
```

```
|  tkCONSTENT
      {fprintf(stdout,"%s",$1);}
;
%%
int main(){
    yyparse();
   exit(0);
}
```

2.2. Ejemplo 2

Se pretende escribir una calculadora, de manera que tomando varias expresiones (una en cada línea) va escribiendo los resultados. Así, para una entrada

```
3+5-2
6+7-5-1
```

generará la salida

```
->6
->7
```

El analizador léxico puede ser

```
%{
#include "y.tab.h"
%}
%%
[\t ]+          {/* tab, blanco, no hace nada */};
[0-9]+          {sscanf(yytext,"%d",&(yylval.valorEntero));
                 return(tkCONSTENT);}
.|\n            {return(yytext[0]);}
%%
```

y el analizador sintáctico (con las acciones que realizan los cálculos) puede ser

```
%{
#include <stdio.h>
%}
%union{
    int valorEntero;
}
%token tkCONSTENT
%type <valorEntero> expresion
%type <valorEntero> tkCONSTENT
%%
lineas:
   lineas
   expresion
   '\n'
      {fprintf(stdout,"->%d\n",$2);}
|  expresion
   '\n'
      {fprintf(stdout,"->%d\n",$1);}
;

expresion:
   expresion
   '+'
   tkCONSTENT
      {$$=$1+$3;}
|  expresion
   '-'
   tkCONSTENT
      {$$=$1-$3;}
|  tkCONSTENT
      {$$=$1;}
;
%%
int yyerror(){
    return 0;
}

int main(){
```

```
   yyparse();
   exit(0);
}
```

El objetivo final de esta práctica es implementar, utilizando Bison, un analizador sintáctico para "miLenguaje" que además genere la traducción del programa fuente analizado al lenguaje Ada.

Esta práctica consta de dos partes, la primera obligatoria, que valdrá un 30 % de la nota de esta práctica, y la segunda optativa, que valdrá el 70 % de la nota.

1. Parte 1 (obligatoria, 30 %): escribir en el fichero
   `AnSem_miLenguaje.y`
   la gramática LALR del lenguaje que habéis definido. Este fuente debe compilarse sin problemas mediante bison, y no es admisible más de un conflicto.

2. Parte 2 (optativa, 70 %): completar el fichero anterior (llamar
   `GenCod_miLenguaje.y`
   al nuevo fichero) con las acciones semánticas necesarias para que traduzca el código fuente a uno equivalente en Ada.

3. Resultados

Como resultado de esta práctica hay que **entregar un único** fichero denominado `pract4.tar`, de manera que la ejecución de la instrucción
`tar -xvf pract4.tar`
genere un directorio denominado `pract4` con los siguientes ficheros:

3.1. Parte 1 obligatoria

- `AL_miLenguaje_3.l` que contiene el fuente Flex del analizador léxico usado (es posible que la inserción de las acciones semánticas requiera modificar el analizador léxico usado en la práctica anterior).

- `AnSem_miLenguaje.y` que contiene el fuente Bison con la gramática, tal y como se pide para la parte obligatoria de la práctica.

- `AnSem_miLenguajeMake` que contiene el fuente para la utilidad `make` de manera que la ejecución de

  ```
  make -f AnSem_miLenguajeMake
  ```

  genere el ejecutable `AnSem_miLenguaje` que llevaría a cabo el análisis sintáctico correspondiente a la parte obligatoria de la práctica. La invocación a este analizador debe ser

  ```
  AnSem_miLenguaje miPrograma.ml
  ```

- Deben también incluirse todos aquellos ficheros que sean necesarios (conteniendo, por ejemplo, implementaciones de tipos de datos, funciones para el tratamiento de errores, etc.).

3.2. Parte 2 optativa

Los ficheros de esta parte optativa deberán estar contenidos también en el fichero `pract4.tar`, pero solo deberán aparecer en aquellos casos en los que se opte por realizar la parte opcional.

- `GenCod_miLenguaje.y` que contiene el fuente Bison correspondiente a la parte optativa de la práctica.

- `GenCod_miLenguajeMake` que contiene el fuente para la utilidad `make` de manera que la ejecución de

  ```
  make -f GenCod_miLenguajeMake
  ```

  genere el ejecutable `Gencod_miLenguaje` y este a su vez al ejecutarlo de la siguiente forma

  ```
  GenCod_miLenguaje miPrograma.ml miProgramaTraducido.adb
  ```

  traduce el fuente en "miLenguaje" al lenguaje Ada.

- Deben también incluirse todos aquellos ficheros que sean necesarios (conteniendo, por ejemplo, implementaciones de tipos de datos, funciones para el tratamiento de errores, etc.).

4. Notas

- Con el objetivo de simplificar **la parte optativa** de la práctica, vamos a asumir que el fuente "miLenguaje" que se va a traducir es completamente correcto.

- Como paso previo a la corrección de las prácticas se ejecutará de manera automática un script que comprobará si lo entregado por el alumno como resultado de la práctica es exactamente lo que se pide (ejecución de la descompresión mediante `tar` y comprobación de que se han generado los ficheros pedidos). Si esto no es así, la práctica aparecerá como no presentada. En caso de dudas respecto a lo que se tiene que generar, consultar con el profesor.

- En el directorio de salidas de la asignatura en Merlin hay un ejemplo completo de fichero Makefile para la compilación de conjunto de fuentes flex y bison. Por otro lado, también hay ejemplos ilustrativos de cómo pueden formatearse fuentes tanto flex como bison.

5. Fecha límite para la entrega de resultados

Obligatoriamente, cada grupo debe entregar la cuarta práctica antes del día 12 de enero de 2008.