2018/2019

MEMORIA

COMPILADORES

FRANCISCO LÓPEZ TOLEDO
Profesora: MARIA ANTONIA CARDENAS VIEDMA
Francisco.lopezt@um.es Convocatoria: Junio. Grupo 2.1

ÍNDICE

1. Introducción.

2. Análisis léxico.

3. Análisis sintáctico.

4. Análisis semántico y generación de código.

5. Instrucciones de ejecución.

6. Ejemplos de funcionamiento.

1.INTRODUCCIÓN

Esta versión de un compilador realizado para el lenguaje miniC con alguna extensión ha sido realizada apoyándose en Flex y Bison. Más concretamente, la herramienta Flex ha permitido realizar el análisis léxico, mientras que Bison ha permitido la realización automática del análisis sintáctico y ha servido como apoyo a la gestión de las tareas del análisis semántico y generación de código apoyándose en algunas estructuras de datos que mas tarde describiré.

Además, ha sido clave para la finalización de la practica tener en cuenta aspectos teóricos básicos a la hora de resolver errores en el análisis sintáctico, conflictos y ambigüedades que aparecen durante la realización de la practica.

2.ANÁLISIS LÉXICO

En primer lugar, para realizar el análisis léxico en Flex se necesitó de una identificación de los tokens necesarios de nuestra gramatica y su reconocimiento mediante expresiones regulares (tipo flex).

Las palabras reservadas son:

FUNC	→	*func*
CONST	→	*const*
VAR	→	*var*
IF	→	*if*
ELSE	→	*else*
DO	→	*do*
WHILE	→	*while*
READ	→	*read*
PRINT	→	*print*

Los caracteres especiales son:

LPAREN	→	(
RPAREN	→	)
SEMICOLON	→	;
COMMA	→	,
ASSIGNOP	→	=
PLUSOP	→	+
MULTOP	→	*
DIVOP	→	/
MINUSOP	→	-
LLLAVE	→	{
RLLAVE	→	}

Los identificadores, números y cadenas son:

ID	→	*identificadores*
INT	→	*números*
STRING	→	*cadenas*

En cuanto al desarrollo de las expresiones regulares, la mayoría son triviales excepto la del comentario multilínea y la de la cadena. Sin embargo, gracias a los recursos en el aula virtual el comentario multilinea y el reconocimiento de un comentario sin cerrar no fue una tarea ardua.

Por otro lado, la expresión regular para reconocer la cadena me dio mas quebraderos de cabeza ya que fallaba al reconocer una cadena con la peculiaridad de que contuviera dentro de las dobles comillas a otras dobles comillas **pero** con un carácter de escape (contrabarra) delante, tuviendo que desarrollarla mejor contemplando ese caso.

\"[^"\n]*\" → \"([^"\n]|\\\")*\"

Además, durante la prueba del analizador semántico surgió un error que no se tuvo en cuenta en el desarrollo de la expresión regular que reconoce los espacios en blanco al no tener \r.

[\n\t]+ → [\n\t\r]+

En cuanto al reconocimiento de errores especificos de las limitaciones del lenguaje, podemos encontrar la identificación de que los enteros no se pasen de los 32 bits y que los identificadores no excedan los 16 caracteres. Para ello, definimos dos funciones en C:

```
void overflow_checker(){

    if(atoll(yytext) > pow(2,31)){
       printf("Error : Entero '%s' fuera del rango permitido en
linea %d \n",yytext,yylineno);
    }
```

```
}

void id_length_checker(){

    if(yyleng > 16){
        printf("Error : Identificador '%s' excede los 16 caracteres
en linea %d \n",yytext,yylineno);
    }
}
```

En cuanto a la recuperación de errores, utilizamos la recuperación en modo pánico que consiste en ignorar caracteres extraños hasta encontrar un carácter valido para un nuevo token. Para esta tarea se ha utilizado la siguiente expresión regular:

[^ a-zA-Z_0-9();\",=+{}\-/*\n\t\r]+

Para poder devolver los atributos de los identificadores, cadenas y números a Bison necesitamos asignar a la variable global yyval (con tipos iguales a la unión declarada en bison) las cadenas reconocidas por su correspondiente expresión regular.

Ejemplo:

```
{yylval.cadena = strdup(yytext);return STRING;}
```

3.ANÁLISIS SINTÁCTICO

En primer lugar, es necesario definir la gramática que se utiliza en Bison para construir nuestro análisis sintáctico cumpliendo la especificación de la práctica con la adición de la funcionalidad del bucle DO-WHILE.

```
program: FUNC ID LPAREN RPAREN LLLAVE declarations statements_list
RLLAVE;

declarations : declarations VAR  identifier_list SEMICOLON
             | declarations CONST identifier_list SEMICOLON
             | declarations VAR error SEMICOLON
             | declarations CONST error SEMICOLON
             |  λ
             ;

identifier_list : asig
                | identifier_list COMMA asig
                ;


asig : ID
      | ID ASSIGNOP expression
      ;

statements_list : statements_list statement
                | λ
                  ;


statement : ID ASSIGNOP expression SEMICOLON
           | LLLAVE statements_list RLLAVE
           | IF LPAREN expression RPAREN statement ELSE statement
           | IF LPAREN expression RPAREN statement
           | DO statement WHILE LPAREN expression RPAREN
           | WHILE LPAREN expression RPAREN statement
           | PRINT print_list SEMICOLON
           | READ read_list SEMICOLON
           ;

print_list : print_item
            | print_list COMMA print_item
            ;
```

```
print_item : expression
           | STRING
           ;

read_list : ID
          | read_list COMMA ID
          ;

expression : expression PLUSOP expression
           | expression MINUSOP expression
           | expression MULTOP expression
           | expression DIVOP expression
           | MINUSOP expression %prec UMINUS
           | LPAREN expression RPAREN
           | ID
           | INT
           ;
```

En cuanto a la forma de solucionar ambigüedades de las operaciones aritmeticas asignaremos precedencias y asociatividades con el fin de eliminarlas. Para esta tarea, utilizaremos **%left** para asignar asociatividad por la izquierda. Sin embargo, al menos unario hay que tratarlo de forma distinta dándole la mayor precedencia.

%left MINUSOP PLUSOP
%left MULTOP DIVOP
%left UMINUS

`MINUSOP expression %prec UMINUS`

Para solucionar el conflicto de desplazamiento/reducción que ocurre en el if else, dejaremos a bison que lo solucione por defecto, desplazando. Además, para evitar el *warning* que produce bison para el conflicto podemos utilizar **%expect 1** pero no cambiara el funcionamiento del analizador sintáctico.

Ademas, he realizado una recuperación de errores en Bison añadiendo nuevas reglas de producción. Sin embargo, mi primera propuesta producia un nuevo conflicto desplazamiento/reducción en **DECLARATIONS** añadiendo la regla de producción: **declarations: error SEMICOLON.** La cuestión es que, tal y como había puesto la regla en declarations, se produce un conflicto desplazamiento/reducción porque **error** es un símbolo que pertenece a SIGUIENTE(declarations) porque está en PRIMERO(statement_list). Y como declarations tiene una regla que deriva en lambda, esa regla se puede usar para reducir si se detecta un **error**. Pero también se puede desplazar para usar la regla de declarations que comienza con **error**.

Cambiando esa primera propuesta por una que evite que declarations comience por **error** en una regla. La idea es que necesitas haber metido en la pila un no terminal declarations, luego tienes que haber visto un token VAR ó CONST para poder tratar el error sintáctico por aquí, saltando hasta el siguiente punto y coma.

declarations VAR error SEMICOLON
declarations CONST error SEMICOLON

Lo mismo hago en **STATEMENT**, añadiendo esta regla de producción: **error SEMICOLON**.

4.ANÁLISIS SEMÁNTICO Y GENERACIÓN DE CÓDIGO

Para realizar la fase del compilador de análisis semántico y de generación de código nos apoyaremos en varias estructuras de datos.

En primer lugar, para el análisis semántico utilizaremos la lista de símbolos proporcionada en la que almacenaremos las variables, constantes y cadenas siguiendo un orden para la posterior generación de código en la que insertaremos por el principio si es una cadena y por el final si es constante o variable. Además, he modificado para permitir este convenio de inserción el archivo ***listaSimbolos.h*** añadiendo ***CADENA*** al enumerado de los tipos y cambiando de un array de caracteres con longitud fija a un ***char* nombre*** en la estructura de **Simbolo.**

Posteriormente, utilizaremos una lista que almacenará instacias de una estructura llamada **Operación** que nos permite representar las operaciones MIPS.

Se declarará en Bison también una unión para que Bison elija el tipo adecuado automáticamente ya que por defecto los valores devueltos por las acciones y el analizador léxico son enteros. Para posteriormente asociar los nombres de los miembros de la unión a cada token y a cada no terminal.

```
%union {char* cadena; ListaC codigo;}
%token LPAREN RPAREN LLLAVE RLLAVE VAR CONST SEMICOLON
        COMMA ASSIGNOP ELSE IF WHILE FUNC PRINT READ DO
%token <cadena>STRING INT ID
%type <codigo> expression statement statements_list print_list print_item read_list
declarations identifier_list asig
```

En cuanto a la inserción de símbolos en la lista, crearemos una función en C para añadir constantes y variables (***añadeEntradaVar(char * valor,Tipo tipoId)***) y otra para añadir las cadenas (***anadeEntradaCadena(char* s1)***).

```
void anadeEntradaCadena(char* s1){
        Simbolo s = {s1,CADENA,contCadenas};
        insertaLS(l,inicioLS(l),s);
}
void anadeEntradaVar(char *valor, Tipo tipoId){
        Simbolo s = {strdup(valor),tipoId,0};
        insertaLS(l, finalLS(l), s);
}
```

Con esta lista de Simbolos podemos controlar que no se declare una variable mas de una vez, que no volvamos a asignar valores a una constante y que no se use una variable sin haberse declarado antes mostrando el mensaje de error correcto en cada caso.

```
int perteneceTablaS(char* valor){

        return buscaLS(l,valor) != finalLS(l);
}
```

En cuanto a la generación de código, usaremos el análisis sintáctico ascendente de Bison (LALR) para ir subiendo las listas de código de esta forma **$$ = creaLC()** y según que acción estemos haciendo concatenar a **$$** o insertar al final la operación.

Para traducir las expresiones aritméticas suma, resta, multiplicación, división y gestionar el menos unario. Para todas excepto el menos unario son todas muy similares simplemente cambiando en el campo operación.op que depende de cada caso siguiendo este "esquema":

```
$$=creaLC();
concatenaLC($$, $1);
concatenaLC($$, $3);
        Operacion operacion =
    {"add/sub/mul/div",nuevoRegistro(),recuperaResLC($1),recuperaResLC
    ($3)};

insertaLC($$, finalLC($$), operacion);
guardaResLC($$, operacion.res);
marcarRegistroLibre(operacion.arg1);
marcarRegistroLibre(operacion.arg2);
liberaLC($1);
liberaLC($3);
```

Mientras que para el menos unario se utiliza en el campo operación.op "**neg"** y que no necesita argumento 2 asignadole a se campo **NULL**.

En cuanto a las sentecias de control ya no es tan fácil su traducción ya que tienes que controlar bien la sentencia que quieres traducir en ensamblador.

IF LPAREN expression RPAREN statement

```
$$=$3;

Operacion operacion =
{"beqz",recuperaResLC($3),generarEtiqueta(),NULL};
insertaLC($$, finalLC($$), operacion);
marcarRegistroLibre(operacion.res);
concatenaLC($$, $5);

char* etiqueta = operacion.arg1;
operacion.op= "\n";
operacion.res= concatenaCadenas(etiqueta,":");
operacion.arg1=NULL;
operacion.arg2=NULL;
insertaLC($$, finalLC($$), operacion);
liberaLC($5);
```

DO statement WHILE LPAREN expression RPAREN

```
$$=creaLC();

char* etiq1= generarEtiqueta();
 Operacion operacion;
operacion.op="\n";
operacion.res=concatenaCadenas(etiq1,":");
operacion.arg1=NULL;
operacion.arg2=NULL;
insertaLC($$, finalLC($$), operacion);
concatenaLC($$,$2);
concatenaLC($$,$5);

operacion.op = "bnez";
operacion.res= recuperaResLC($5);
operacion.arg1= etiq1;
operacion.arg2=NULL;
insertaLC($$, finalLC($$), operacion);


marcarRegistroLibre(operacion.res);
liberaLC($2);
liberaLC($5);
```

WHILE LPAREN expression RPAREN statement

```
        $$= creaLC();

    char* etiq1= generarEtiqueta();
    Operacion operacion;
    operacion.op="\n";
    operacion.res=concatenaCadenas(etiq1,":");
    operacion.arg1=NULL;
    operacion.arg2=NULL;
    insertaLC($$, finalLC($$), operacion);


    concatenaLC($$,$3);

    operacion.op = "beqz";
    operacion.res= recuperaResLC($3);
    operacion.arg1=generarEtiqueta();
    operacion.arg2=NULL;
    insertaLC($$, finalLC($$), operacion);
    marcarRegistroLibre(operacion.res);

    concatenaLC($$, $5);
    char* etiq2= operacion.arg1;

    operacion.op= "b";

    operacion.res=etiq1;
    operacion.arg1=NULL;
    operacion.arg2=NULL;
    insertaLC($$, finalLC($$), operacion);

    operacion.op="\n";
    operacion.res= concatenaCadenas(etiq2,":");
    operacion.arg1=NULL;
    operacion.arg2=NULL;
    insertaLC($$, finalLC($$), operacion);
    liberaLC($3);
    liberaLC($5);}
```

IF LPAREN expression RPAREN statement ELSE statement

```
                $$=$3;

                Operacion operacion =
{"beqz",recuperaResLC($3),generarEtiqueta(),NULL};
        insertaLC($$, finalLC($$), operacion);
                marcarRegistroLibre(operacion.res);
                concatenaLC($$, $5);



        char* etiqueta1= operacion.arg1;

        operacion.op= "b";
        operacion.res=generarEtiqueta();
        operacion.arg1=NULL;
        operacion.arg2=NULL;
        insertaLC($$, finalLC($$), operacion);


        char* etiqueta2= operacion.res;

        operacion.op = "\n";
        operacion.res= concatenaCadenas(etiqueta1,":");
        operacion.arg1= NULL;
        operacion.arg2=NULL;
        insertaLC($$, finalLC($$), operacion);

        concatenaLC($$, $7);

        operacion.op = "\n";
        operacion.res= concatenaCadenas(etiqueta2,":");
        operacion.arg1= NULL;
        operacion.arg2=NULL;
        insertaLC($$, finalLC($$), operacion);
        liberaLC($5);
        liberaLC($7);
```

Para la traducción de estas sentencias se han seguido las guías proporcionadas en el aula virtual.

En cuanto la generación de código de **print_item expression**, debemos crear la operación **move $a0, recuperaResLC($1),** posteriormente **li $v0,1** y finalizar con un **syscall** siguiendo la directivas de MIPS.

Mientras que en **print_item STRING** debemos hacer un **la $a0,$strX** donde X se consigue recuperando de la lista de símbolos y concatenándolo a str el valor, posteriormente **li $v0,4** y **syscall**.

Para traducir **read_list** se haría un li $v0,5, syscall y luego guardamos el resultado con **sw $v0,_x**.

Debo de marcar que no he terminado el código con un **jr $ra** ya que asi no terminaba de ejecutar el programa, por lo que opté por poner un **li $v0,10** y un **syscall.**

Una de las dificultades de la generación de código fueron las violaciones del segmento por olvidar subir los atributos en el control de errores semánticos.

Como funciones auxiliares he creado:

char generarEtiqueta();*
char nuevoRegistro();*
void anadeEntradaCadena(char s1);*
void marcarRegistroLibre(char reg);*
char concatenaCadenas(char* s1, char* s2);*
*void anadeEntradaVar(char *valor, Tipo tipoId);*
int perteneceTablaS(char valor);*
*int esConstante(char *valor);*
void imprimirListaC(ListaC lista);
void imprimirTablaS();
void imprimirTablaSCadenas(PosicionLista p,int contador);

5.INSTRUCCIONES DE EJECUCIÓN

1. make
2. ./analizador
3.Introducir el nombre del archivo
4. Se generará un archivo con la salida llamado **salida.s**

Observación: He optado por leer el archivo de la entrada pero si se quisiera se podría ejecutar recibiendo de la terminal como argumento el nombre del archivo (implementado pero comentado).

6.EJEMPLOS DE FUNCIONAMIENTO

Fichero entrada.txt

```
func main ()
{
  const a=0, b=0;
  var c=5+2-2;
  print "Inicio del programa\n";
  if (a) print "a","\n";
  else if (b) print "No a y b\n";
       else do
         {
              print "c = ",c,"\n";
              c = c-2+1;
         }while(c)
 print "Final","\n";
}
```

salida.s

```
################
# Seccion de datos
   .data
$str1:
   .asciiz  "Inicio del programa\n"
$str2:
   .asciiz  "a"
$str3:
   .asciiz  "\n"
$str4:
   .asciiz  "No a y b\n"
$str5:
   .asciiz  "c = "
$str6:
   .asciiz  "\n"
$str7:
   .asciiz  "Final"
$str8:
   .asciiz  "\n"
_a :
 .word 0
_b :
 .word 0
_c :
 .word 0
```

```
####################
# Seccion de codigo
  .text
  .globl main
main:
   li $t0,0
   sw $t0,_a
   li $t0,0
   sw $t0,_b
   li $t0,5
   li $t1,2
   add $t2,$t0,$t1
   li $t0,2
   sub $t1,$t2,$t0
   sw $t1,_c
   la $a0,$str1
   li $v0,4
   syscall
   lw $t0,_a
   beqz $t0,$et4
   la $a0,$str2
   li $v0,4
   syscall
   la $a0,$str3
   li $v0,4
   syscall
   b $et5

 $et4:
   lw $t1,_b
   beqz $t1,$et2
   la $a0,$str4
   li $v0,4
   syscall
   b $et3

 $et2:

 $et1:
   la $a0,$str5
   li $v0,4
   syscall
   lw $t2,_c
   move $a0,$t2
   li $v0,1
   syscall
   la $a0,$str6
```

```
   li $v0,4
   syscall
   lw $t3,_c
   li $t4,2
   sub $t5,$t3,$t4
   li $t3,1
   add $t4,$t5,$t3
   sw $t4,_c
   lw $t3,_c
   bnez $t3,$et1

 $et3:

 $et5:
   la $a0,$str7
   li $v0,4
   syscall
   la $a0,$str8
   li $v0,4
   syscall

#################
# Fin
   li $v0, 10
   syscall
```

Salida tras ejecutar el código ensamblador.

```
Inicio del programa
c = 5
c = 4
c = 3
c = 2
c = 1
Final

-- program is finished running --
```

Fichero entrada1.txt

```
func main ()
{
```

```
const a=6,b=2;
var x,y=7*b-(a*4)/2,z; //y=2.
/* Comprueba sentencias
y expresiones */
while (y)
          {
            x=a+2*b;
            if (x+1) print "x vale ", x," e y ",y,"\n"; // x vale 10.
            if ((x-b*5)/100) print "Esto no va bien","\n";
            else { print "Introduce valores de x y z:\n"; read x,z; print 3-y,"; Debe salir
(x+z)*100:", (x+z)*100,"\n";}
            y=y-1;
      }
}
```

salida.s

```
################
# Seccion de datos
   .data
$str1:
   .asciiz  "x vale "
$str2:
   .asciiz  " e y "
$str3:
   .asciiz  "\n"
$str4:
   .asciiz  "Esto no va bien"
$str5:
   .asciiz  "\n"
$str6:
   .asciiz  "Introduce valores de x y z:\n"
$str7:
   .asciiz  "; Debe salir (x+z)*100:"
$str8:
   .asciiz  "\n"
_a :
 .word 0
_b :
 .word 0
_x :
 .word 0
_y :
 .word 0
_z :
```

```
 .word 0

####################
# Seccion de codigo
  .text
  .globl main
main:
   li $t0,6
   sw $t0,_a
   li $t0,2
   sw $t0,_b
   li $t0,7
   lw $t1,_b
   mul $t2,$t0,$t1
   lw $t0,_a
   li $t1,4
   mul $t3,$t0,$t1
   li $t0,2
   div $t1,$t3,$t0
   sub $t0,$t2,$t1
   sw $t0,_y

 $et4:
   lw $t0,_y
   beqz $t0,$et5
   lw $t1,_a
   li $t2,2
   lw $t3,_b
   mul $t4,$t2,$t3
   add $t2,$t1,$t4
   sw $t2,_x
   lw $t1,_x
   li $t2,1
   add $t3,$t1,$t2
   beqz $t3,$et1
   la $a0,$str1
   li $v0,4
   syscall
   lw $t1,_x
   move $a0,$t1
   li $v0,1
   syscall
   la $a0,$str2
   li $v0,4
   syscall
   lw $t2,_y
   move $a0,$t2
```

```
  li $v0,1
  syscall
  la $a0,$str3
  li $v0,4
  syscall

$et1:
  lw $t3,_x
  lw $t4,_b
  li $t5,5
  mul $t6,$t4,$t5
  sub $t4,$t3,$t6
  li $t3,100
  div $t5,$t4,$t3
  beqz $t5,$et2
  la $a0,$str4
  li $v0,4
  syscall
  la $a0,$str5
  li $v0,4
  syscall
  b $et3

$et2:
  la $a0,$str6
  li $v0,4
  syscall
  li $v0,5
  syscall
  sw $v0,_x
  li $v0,5
  syscall
  sw $v0,_z
  li $t3,3
  lw $t4,_y
  sub $t6,$t3,$t4
  move $a0,$t6
  li $v0,1
  syscall
  la $a0,$str7
  li $v0,4
  syscall
  lw $t3,_x
  lw $t4,_z
  add $t7,$t3,$t4
  li $t3,100
  mul $t4,$t7,$t3
```

```
   move $a0,$t4
   li $v0,1
   syscall
   la $a0,$str8
   li $v0,4
   syscall

 $et3:
   lw $t3,_y
   li $t5,1
   sub $t7,$t3,$t5
   sw $t7,_y
   b $et4

 $et5:

#################
# Fin
   li $v0, 10
   syscall
```

Salida tras ejecutar el código ensamblador.

```
x vale 10 e y 2
Introduce valores de x y z:
1
2
1; Debe salir (x+z)*100:300
x vale 10 e y 1
Introduce valores de x y z:
2
2
2; Debe salir (x+z)*100:400

-- program is finished running --
```

Fichero entrada2.txt

```
func main(){

var a = 20 , b=10;
while (a){
        a=a-b;
        print a,"\n";
}
if(a)
        print "a es 0";
else a = 0;

print "Esto es el Final\n";


}
```

salida.s

```
################
# Seccion de datos
   .data
$str1:
   .asciiz  "\n"
$str2:
   .asciiz  "a es 0"
$str3:
   .asciiz  "Esto es el Final\n"
_a :
 .word 0
_b :
 .word 0

####################
# Seccion de codigo
  .text
  .globl main
main:
   li $t0,20
   sw $t0,_a
   li $t0,10
```

```
  sw $t0,_b

$et1:
  lw $t0,_a
  beqz $t0,$et2
  lw $t1,_a
  lw $t2,_b
  sub $t3,$t1,$t2
  sw $t3,_a
  lw $t1,_a
  move $a0,$t1
  li $v0,1
  syscall
  la $a0,$str1
  li $v0,4
  syscall
  b $et1

$et2:
  lw $t0,_a
  beqz $t0,$et3
  la $a0,$str2
  li $v0,4
  syscall
  b $et4

$et3:
  li $t2,0
  sw $t2,_a

$et4:
  la $a0,$str3
  li $v0,4
  syscall

#################
# Fin
  li $v0, 10
  syscall
```

Salida tras ejecutar el código ensamblador.

```
10
0
Esto es el Final
```