PROYECTO COMPILADOR DE MINIC
MAYO 2021/2022

MARIA DEL CARMEN ALCANTUD JUÁREZ,48754759A
mariacarmen.alcantudj@um.es
AURORA HERVÁS LÓPEZ, 20514953B
aurora.hervasl@um.es
SUBGRUPO PCEO(1.4)
Profesor de prácticas : Eduardo Martinez García

1. Introducción

En este proyecto hemos diseñado un compilador de un lenguaje basado en C, al que llamamos miniC, desde cero. Para hacerlo fue necesario apoyarnos en los conocimientos de teoría para hacer un correcto análisis léxico, sintáctico y semántico.

La finalidad última es conseguir un código en ensamblador que sea funcional y coherente con el fichero de entrada que realice las sentencias indicadas, fue necesario el uso tanto de Flex como de Bison. Mostraremos en esta memoria el proceso que hemos seguido para conseguir que funcione, algunos ejemplos de prueba y las mejoras implementadas.

2. Análisis léxico

Para la fase del analizador léxico, hemos utilizado la herramienta Flex, que lo genera automáticamente. En primer lugar, hemos creado un archivo miniC.l y dividido en 3 secciones para la definición del léxico.

En la sección de definiciones, añadimos las definiciones de dígito, letra y pánico, que serán todos los caracteres no válidos:

```
digito [0-9]
letra [a-zA-Z_]
panico [^ \n\r\ta-zA-Z_0-9+\-*/();=,{}]
```

A continuación, en la sección de reglas, utilizamos expresiones regulares para definir los tokens. Definimos los caracteres especiales *;, ,, +, -, *, /, =, (,),{* y *}* y las palabras reservadas *void, var, const, if, else, while, print* y *read*. También añadimos las palabras reservadas *do* y *for* para la implementación del for y el do while. Seguimos el siguiente formato:

```
"void"                          { return VOID; }
"+"                             { return MAS; }
```

Definimos los identificadores como una secuencia de letras, dígitos y símbolos de subrayado, no comenzando por dígito y no excediendo los 16 caracteres, con la expresión regular:

```
{letra}({letra}|{digito})*      { if (yyleng > 16) {
                                      printf("ERROR: identificador demasiado
                                      largo en linea %d: %s\n",yylineno, yytext);
                                      errores_lexicos++;}
                                  else
                                      yylval.cadena = strdup(yytext);
```

Para definir los enteros (desde −2^31 hasta 2^31):

```
{digito}+                   {   yylval.cadena = strdup(yytext);
                                if (atol(yytext) > 2147483648 || atol(yytext) < -2147483648)
                                    {printf("ERROR: entero demasiado grande en linea %d: %s\n", yylineno, yytext);
                                    errores_lexicos++; }
                               return NUM; }
```

Para las cadenas hemos utilizado las siguientes expresiones, una para definirlas como una secuencia de caracteres delimitados por comillas dobles y otra que detecta si una cadena está sin cerrar y devuelve un fallo.

```
\"([^"\n]|\\\")*\"              { yylval.cadena = strdup(yytext); return STR; }
\"([^"\n]|\\\")*                { printf("ERROR: cadena sin cerrar en linea %d\n", yylineno);
                                      errores_lexicos++;}
```

Si detectamos caracteres que no están definidos devolvemos un error:

```
{panico}+                   { printf("ERROR: caracteres no validos en linea %d: %s\n", yylineno, yytext);
                                  errores_lexicos++;}
```

Por último tenemos expresiones regulares para definir los comentarios multilínea (que comienzan por /* y terminan por */) y los comentarios de una línea (que comienzan con //) y dar un error si hay un comentario sin cerrar al igual que con las cadenas.

```
"/*"                            {inicio_comentario = yylineno ;BEGIN(comentario); }
<comentario>"*/"                {BEGIN(INITIAL);    }
<comentario>(.|\n)              { }


<comentario><<EOF>>         { printf("ERROR: comentario no cerrado en línea %d\n", inicio_comentario);
                              errores_lexicos++;
                              return 0;}
```

Para la recuperación de errores en modo pánico hemos definido en la sección de subrutinas de usuario la función `error_lexico` que devuelve un mensaje indicando que se ha producido un error léxico y el número de la línea.

```
void error_lexico() {
    printf("Error lexico en linea %d: %s\n", yylineno, yytext);
}
```

3. Análisis sintáctico

Para realizar el analizador sintáctico hemos utilizado la herramienta Bison. En primer lugar hemos creado el fichero miniC.y para que funcione conjuntamente con el léxico definido en la sección anterior, y así reconocer sintácticamente ficheros generados por la gramática de miniC.

En este fichero añadimos como símbolos terminales todos los tokens de la gramática que hemos definido anteriormente siguiendo el formato:

```
%token   IF              "if"
%token   LLAVEI          "{"
```

Los símbolos no terminales los declaramos de la siguiente manera:

```
%type <codigo> expression declarations statement statement_list
print_item print_list read_list identifier_list asig
```

Añadimos también en declaraciones la definición de la unión para los tipos de los valores de yylval.

```
%union{
   char *cadena;
   ListaC codigo;  }
```

A continuación, nos fijamos en la sintaxis proporcionada en el apartado 2.1.2 del enunciado de la práctica y vamos introduciendo las reglas en nuestro fichero:

```
program   : VOID ID "(" ")" "{" {l = creaLS();} declarations statement_list "}"
          | %empty                {inicializar_registros();}
          ;

declarations    : declarations VAR {t = VARIABLE;}identifier_list ";"
                | declarations CONST {t = CONSTANTE;} identifier_list ";"
                |%empty
identifier_list : asig
                | identifier_list "," asig
                ;
asig             : ID
                | ID "=" expression
                ;
statement_list : statement_list statement
                |%empty
                ;
statement:       ID "=" expression ";"
                |"{" statement_list "}"
                |"if" "(" expression ")" statement "else" statement
                |"if" "(" expression ")" statement %prec ELSE1
                |"while" "(" expression ")" statement
                |"print" print_list ";"
                |"read" read_list ";"
                ;
print_list:      print_item
                | print_list "," print_item
                ;
print_item:      expression
                | STR
                ;
read_list:       ID
                | read_list "," ID
                ;
expression:      expression "+" expression
                |expression "-" expression
                |expression "*" expression
                |expression "/" expression
                |"-" expression %prec UMINUS
                |"(" expression ")"
                |ID
                |NUM
                ;
```

En cuanto a las precedencias y asociatividades de los operadores aritméticos, establecemos las siguientes reglas para que las expresiones no sean ambiguas:

```
%left "+" "-"
%left "*" "/"
%precedence UMINUS
```

Por último, para evitar los conflictos desplazamiento/reducción, añadimos las siguientes reglas que utilizamos en la declaración del *if:*

```
%nonassoc ELSE1
%nonassoc "else"
```

4. Análisis semántico

La parte de análisis semántico se implementa en el fichero *miniC.l* en el que ya hemos incluido el analizador sintáctico, introduciendo las comprobaciones necesarias en las reglas de derivación que hemos creado.

Utilizamos en esta sección la librería *listaSimbolos.h* con la que fuimos capaces de llevar un control de los identificadores que se declaran a lo largo del código, así como el tipo de variable que son para no poder reescribir una variable que se ha declarado como constante.

Por tanto en la primera sentencia asignamos a una variable *t* donde almacenaremos si es constante o variable, información que guardaremos en la lista de símbolos cuando guardemos la variable.

```
declarations    : declarations VAR {t = VARIABLE;}identifier_list ";"{
                    $$ = $1;
                    concatenaLC($$,$4);
                    liberaLC($4);
                }
                | declarations CONST {t = CONSTANTE;} identifier_list";"{
                    $$ = $1;
                    concatenaLC($$,$4);
                    liberaLC($4);
                }
                |%empty {$$ = creaLC();}
                ;
```

Cuando realizamos una asignación comprobamos gracias a la función booleana *perteneceTablaS(char* n)* si el registro está en la tabla de símbolos, en caso de que ya esté lo añadimos al contador *errores_semanticos* y en caso contrario lo añadimos a la lista y se guarda como variable o como constante según lo que indique el registro t.

```
asig            : ID{
                    if (!perteneceTablaS($1)) {
                        insertaEntrada($1, t); }
                    else{
                        printf("Error en la linea %d: identificador %s
redefinido\n",yylineno, $1);
                        errores_semanticos++;
                    }
                    $$ = creaLC();  }
```

Esta comprobación se realiza más la siguiente regla de producción de asig, al igual que en otras partes del código como es *statement : ID "=" expresion ";"*.

Cabe destacar que en el caso de que queramos introducir una cadena en la lista de símbolos utilizamos la regla de producción *print_item* , donde comprobamos que esa cadena no esté ya guardada.

Si no lo está, introducimos a mano el valor de cadena en el tipo de símbolo e incrementamos el contador de cadenas, para poder darle un número distinto a cada cadena.

```
| STR {
                    if(!perteneceTablaS($1)){
                        Simbolo aux;
                        aux.nombre = $1;
                        aux.tipo = CADENA;
                        aux.valor = contCadenas;
                        insertaLS(l, finalLS(l), aux);
                        contCadenas++;
                    }
                    /* Instrucciones para traducción a ensamblador*/
```

Para facilitarnos todas estas operaciones con las listas y poder reutilizar el código, hemos definido una serie de funciones que se ven a continuación:

Creamos una función *perteneceTablaS()* que recorre la lista de símbolos para comprobar si un elemento ya se encuentra en nuestra lista. Devuelve verdadero o falso según lo encuentre o no:

```
bool perteneceTablaS(char* n) {
   PosicionLista p = buscaLS(l, n);
   if (p != finalLS(l)) {
       return true;
   }
   return false;
}
```

La función *insertaEntrada()* recibe como parámetros un símbolo y su tipo y lo inserta al final de la lista de símbolos:

```
void insertaEntrada(char *n, Tipo t)
{
   Simbolo aux;
   aux.nombre = n;
   aux.tipo = t;
   insertaLS(l, finalLS(l), aux);
}
```

Por último la función *esConstante*() recibe un símbolo como parámetro, lo busca en la lista de símbolos y si lo encuentra, extrae su tipo y comprueba si se trata de una constante. En ese caso devolverá verdadero, si no devolverá falso:

```
bool esConstante(char* n) {
   PosicionLista p = buscaLS(l, n);
   if(p != finalLS(l)){
       Simbolo s = recuperaLS(l,p);
   if (s.tipo == CONSTANTE) {
       return true;
   }
   return false;
   }
}
```

5. Generación de código

Para la realización de esta parte del proyecto hemos necesitado utilizar la librería lista de código, la cual representa mediante cuádruplas las instrucciones de ensamblador y las almacena en una lista enlazada.

Decidimos ir uniendo listas enlazadas conforme se iban traduciendo las correspondientes instrucciones.

Usamos, salvo en los casos de las llamadas al sistema, los registros temporales desde $t0 hasta $t9, por lo que fue necesario crear una estructura llamada *registros[10]* que utilizamos como booleanos para saber si los registros están libres o no y poder asignarlos a lo largo de la generación de código.

Creamos una función *inicializar_registros()* para marcarlos todos como libres:

```
void inicializar_registros(){
   for(int i = 0; i < 10; i++)
       registros[i] = 0;
}
```

También una función *recuperaReg(char* reg)* que lo que hace es devolver el primer registro temporal que actualmente está libre.

```
char* recuperaReg(){
   for(int i = 0; i< 10; i++)
       if (registros[i] == 0){
           registros[i] = 1;
           char* num = malloc(2*sizeof(char));
           *num = i + '0';
           char* reg = concatena("$t", num);
           return reg;
       }
   return NULL;
}
```

También la función *liberaReg(char* reg)* que se encarga de marcar como libres en la tabla el registro que se le pasa como parámetro.

```
void liberaReg(char* reg){
   int num = reg[2] - '0'
   registros[num] = 0;
}
```

También consideramos que era de bastante utilidad una función que se encarga de crear e insertar las cuadruplas que representan las instrucciones en la lista, para ganar así legibilidad en el código. Por lo que creamos la función *creaInsertaLC*():

```
void creaInsertaLC(char* op, char* res, char* arg1, char* arg2, ListaC l ){
   Operacion oper;
   oper.op = op;
   oper.res = res;
   oper.arg1 = arg1;
   oper.arg2 = arg2;
   insertaLC(l, finalLC(l), oper);
}
```

Al ser una función costosa decidimos añadir una función llamada *analisisOk()* que comprueba si hay algún error de algún tipo antes de añadir código en la lista, ya que si hay algún fallo no será necesario hacer la traducción.

```
int analisisOk(){
   if (errores_semanticos + errores_sintacticos + errores_lexicos == 0)
           return true;
   return false;
}
```

Y la última función para la generación de código que implementamos es *concatena(char* c1, char* c2)*, que nos fue muy útil a la hora de darle nombre a los string (*$strx*) así como para concatenar la cadena necesaria para llamar a los registros en instrucciones como *sw*.

```
char* concatena(char* c1, char* c2){
   char* nueva = malloc((strlen(c1)+strlen(c2)+1)*sizeof(char));
   strcpy(nueva, c1);
   strcat(nueva, c2);
   return nueva;
}
```

Una vez se comprueba que el programa a compilar no tiene ningún fallo procedemos a generar un fichero con el código ensamblador. Para ello es necesario una función que se encarga de volcar en la salida el código en ensamblador del programa por lo que diseñamos la función *imprimirLS()* que se encarga de la parte *.data* del programa y la función *imprimirCodigo(ListaC codigo)* que revisa si el primer campo de la cuádrupla es el tipo *"etiq"* ya que en ese caso debe mostrar únicamente la etiqueta, en caso de que no lo sea imprime los argumento que tenga dicha instrucción.

6. Manual del usuario

Para que el usuario final pueda utilizar este compilador debemos fijarnos en el fichero *makefile* en él aparecen todas las acciones que puede realizar, donde podemos observar que hemos añadido una regla para que aparezcan dos ejecutables.

Decidimos poner dos ejecutables para poder realizar las pruebas del léxico de manera independiente y tener así la salida esperada en las pruebas.

Lo único que debe saber el usuario es que puede ejecutar el comando *make run* que volcará en el fichero *salida.s* el código en ensamblador del fichero *prueba.txt* y para ejecutarlo basta con hacer *spim -file salida.s*.

```
all: miniC lexico

miniC :     main.c lex.yy.c miniC.tab.c listaSimbolos.c listaCodigo.c
         gcc -g main.c lex.yy.c miniC.tab.c listaSimbolos.c listaCodigo.c -lfl -o miniC

lexico :main_lexico.c lex.yy.c miniC.tab.c listaSimbolos.c listaCodigo.c
         gcc -g main_lexico.c lex.yy.c miniC.tab.c listaSimbolos.c listaCodigo.c -lfl -o
lexico

lex.yy.c : miniC.l miniC.tab.h
   flex miniC.l

miniC.tab.c miniC.tab.h : miniC.y
   bison -d -v miniC.y

clean :
   rm -f miniC.tab.* lex.yy.c miniC

lex: lexico test_lex.mc.txt
   ./lexico test_lex.mc.txt > salida_lexico.txt

sintactico: miniC test_lex.mc.txt
   ./miniC prueba_sintactico.mc > salida_sintactico.txt


run: miniC prueba.txt
   ./miniC prueba.txt > salida.s
```

Al ejecutar el comando *make run* se generará en código ensamblador del fichero *prueba.txt* y se volcará en *salida.s.*

7. Ejemplos de funcionamiento

Lo primero que haremos será un *make clean* para borrar todos los ficheros que hayan sido generados y seguidamente un *make* para asegurarnos que estamos trabajando con la última versión del código. Y ahora procedemos a ejecutar los ficheros de prueba que nos han sido proporcionados por los profesores de la asignatura.

7.1 Ejemplos léxico

Para comprobar el analizador léxico usaremos el fichero *test_lex.mc* de los recursos y comprobamos que coincide con la salida esperada. Para que la salida sea correcta debemos probar el analizador léxico de forma independiente, por lo que haremos uso del ejecutable *lexico*, por lo que desde la terminal hacemos *make lex*. Que se encarga de ejecutar el léxico con el fichero t*est_lex_salida.txt* y lo volcamos directamente en *salida_lexico.txt.*

Mostramos aquí la salida y también está en el fichero *salida_lexico.txt*

```
ERROR: entero demasiado grande en linea 8: 123456789012
ERROR: identificador demasiado largo en linea 11: _123456789012345678
ERROR: caracteres no validos en linea 14: #@$%&
ERROR: cadena sin cerrar en linea 17
ERROR: comentario no cerrado en línea 21
----------------------
Errores lexicos: 5
```

7.2 Ejemplos sintáctico

En este apartado tenemos un fichero de prueba que está en recursos que se llama *prueba_sintactico1.mc*, pero al no tener ninguna errata aparece lo esperado, que es el código en ensamblador. Por lo que tenemos dicha prueba en el fichero *salida_semantico.txt*

Ejecutamos desde la terminal *make sintactico* que le pasará al compilador el fichero de prueba y lo vuelca en *salida_sintactico.txt*.

7.3 Ejemplos semántico

Este apartado es en el que más ficheros de prueba teníamos así que lo que haremos será usar el comando ./miniC fichero.txt con cada uno de los ficheros de prueba.

Comenzamos con el primero que es *test_sem1.mc* y lo volcamos en *salida_sem1.txt*. la parte *.data* coincide con la salida pero no cuenta con la generación de código, por lo que nuestra salida si lo incluye al tener el compilador completo.

```
############################
.data

# STRINGS ##################
$str1:          .asciiz "Test 1\n"
$str2:          .asciiz "Fin test1\n"

# IDENTIFIERS ##############
_a:     .word 0
_b:     .word 0
_c:     .word 0
####################
# Seccion de codigo
        .text
        .globl main
main:
        li $t0,3
        sw $t0,_b
        li $t0,0
        sw $t0,_c
        li $v0,4
        la $a0,$str1
        syscall
        li $t0,1
        lw $t1,_b
        add $t0,$t0,$t1
        sw $t0,_a
        li $v0,5
        syscall
        sw $v0,_a
        li $v0,4
        la $a0,$str2
        syscall
####################
#FIN
        jr $ra
```

El segundo fichero de prueba lo probaremos con el comando *./miniC test_sem2.mc > salida_sem2.txt*

```
Error en la linea 5: identificador a redefinido
Error en la linea 6: identificador a redefinido
----------------------
Errores lexicos: 0
Errores sintacticos: 0
Errores semanticos: 2
```

La tercera prueba la haremos de la misma forma usando el comando *./miniC test_sem3.mc > salida_sem3.txt*

```
Error en la linea 5: variable x no declarada
Error en la linea 6: variable y no declarada
Error en la linea 7: variable z no declarada
----------------------
Errores lexicos: 0
Errores sintacticos: 0
Errores semanticos: 3
```

Por último lanzamos el comando *./miniC test_sem4.mc > salida_sem4.txt*

```
Error en la linea 5: identificador b es constante
Error en la linea 6: identificador b es constante
----------------------
Errores lexicos: 0
Errores sintacticos: 0
Errores semanticos: 2
```

La última prueba que haremos será la del fichero *prueba.txt* que es la indicada en la memoria para ello desde la terminal ponemos

make run

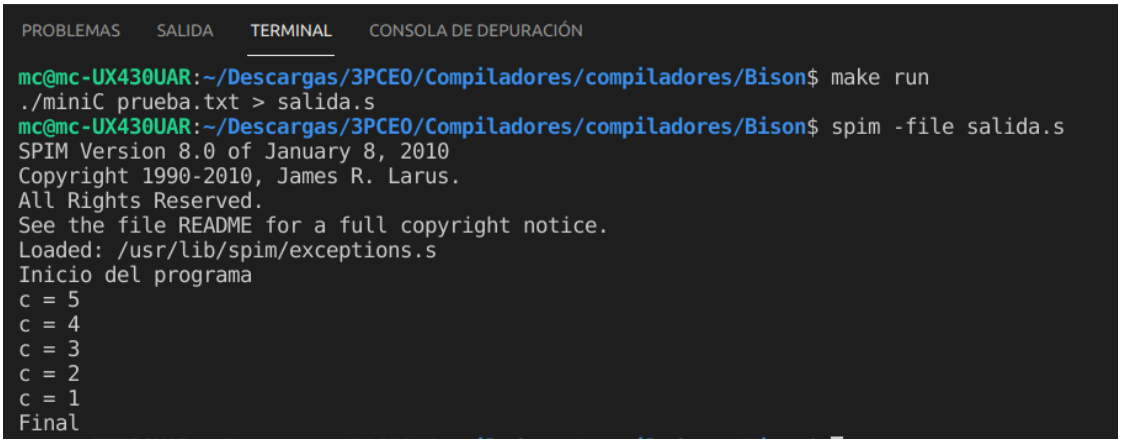
spim -file salida.s

```
#############################
.data

# STRINGS ##################
$str1:          .asciiz "Inicio del programa\n"
$str2:          .asciiz "\n"
$str3:          .asciiz "a"
$str4:          .asciiz "No a y b\n"
$str5:          .asciiz "c = "
$str6:          .asciiz "Final"

# IDENTIFIERS ###############
_a:     .word 0
_b:     .word 0
_c:     .word 0
####################
# Seccion de codigo
        .text
        .globl main
main:
        li $t0,3
        sw $t0,_a
        li $t0,0
        sw $t0,_b
        li $t0,5
        li $t1,2
        add $t0,$t0,$t1
        li $t1,2
```

```
        sub $t0,$t0,$t1
        sw $t0,_c
        li $v0,4
        la $a0,$str1
        syscall
        li $t0,1
        neg $t0,$t0
$l1:
        blez $t0,$l2
        beqz $t0,$l2
        lw $t1,_a
        li $v0,1
        move $a0,$t1
        syscall
        li $v0,4
        la $a0,$str2
        syscall
        li $t2,1
        sub $t0,$t0,$t2
        b $l1
$l2:
        lw $t0,_a
        beqz $t0,$l7
        li $v0,4
        la $a0,$str3
        syscall
        li $v0,4
        la $a0,$str2
        syscall
        b $l8
$l7:
        lw $t2,_b
        beqz $t2,$l5
        li $v0,4
        la $a0,$str4
        syscall
        b $l6
$l5:
$l3:
        lw $t3,_c
        beqz $t3,$l4
        li $v0,4
        la $a0,$str5
        syscall
        lw $t4,_c
        li $v0,1
        move $a0,$t4
        syscall
        li $v0,4
        la $a0,$str2
        syscall
        lw $t5,_c
        li $t6,2
        sub $t5,$t5,$t6
```

```
        li $t6,1
        add $t5,$t5,$t6
        sw $t5,_c
        b $l3
$l4:
$l6:
$l8:
        li $v0,4
        la $a0,$str6
        syscall
        li $v0,4
        la $a0,$str2
        syscall
####################
#FIN
        jr $ra
```

El contenido de *salida.s* es el código mostrado en ensamblador y al usar spim vemos que la salida es la esperada del código en C.



8. Conclusión

La realización de este proyecto nos ha resultado muy interesante ya que nos hemos acercado mucho más al funcionamiento de un compilador que nos resultaba tan abstracto. Además, como en la mayoría de las prácticas, nos ha resultado de gran ayuda para afianzar los conceptos estudiados en la teoría de la asignatura, como el análisis léxico, sintáctico y semántico.

El proyecto en general es extenso por lo que al principio estuvimos un poco perdidas, hasta que trabajando semana a semana fuimos entendiendo mejor las tareas que había que realizar y conseguimos coger soltura con los contenidos. Por ello nos ha ayudado bastante que el trabajo se divida en partes y hayamos podido ir realizando y comprobando cada parte poco a poco. Esto ha hecho que no se nos haya acumulado el trabajo para el final.

En general el proyecto nos ha resultado bastante entretenido y diferente a todo lo que habíamos hecho y nos ha gustado ir creando un compilador parte por parte hasta poder comprobarlo al final y que funcionara tal y como se esperaba.