UNIVERSIDAD DE MURCIA

Práctica compiladores

Integrantes:
Federico Marquez Marconetto - federico.marquez@um.es
JunPeng Jin - junpeng.jin@um.es

Grupo: 2.1

Índice

Manual de usuario

El lenguaje de implementación es C y se usan herramientas como Flex y Bison. Para poder generar el archivo ejecutable se utiliza un archivo makefile el cual contiene qué hacer con cada uno de los archivos de código para compilarlos.
Para ello, abrimos una terminal, situándonos en la ruta donde se encuentran todos los archivos necesarios para el compilador, y ejecutamos el comando *make*, el cual ejecutará todos los comandos que hay definido en dicho archivo.

```
alumno@FIUM:~/Escritorio/Comp/COMPILADORES 2022$ make
bison -d sintactico.y
sintactico.y: aviso: 1 conflicto desplazamiento/reducción [-Wconflicts-sr]
flex lexico.l
gcc lex.yy.c main.c sintactico.tab.c listaSimbolos.c listaCodigo.c -lfl -o salid
a
alumno@FIUM:~/Escritorio/Comp/COMPILADORES 2022$
```

Para ejecutar el compilador con un archivo de entrada es necesario tener el archivo .exe que se genera mediante la ejecución del makefile explicado anteriormente y un archivo .mc el cual será el archivo de entrada con el código en miniC para el compilador.
Para esto, es necesario ejecutar el comando *./salida.exe entrada.mc* siendo salida.exe el archivo generado con el makefile y entrada.mc el archivo que contiene el código en miniC.

```
alumno@FIUM:~/Escritorio/Comp/COMPILADORES 2022$ ./salida prueba.mc
Compilacion realizada con exito.
alumno@FIUM:~/Escritorio/Comp/COMPILADORES 2022$
```

Ejemplo de funcionamiento:

El siguiente fichero de entrada en miniC:

```
1.  void prueba() {
2.    const uno = 1;
3.    var variable = 5;
4.    var c = 4 * 2 - 18 / 3 * 3;
5.    print "Resultado de 4 * 2 - 18 / 3 * 3 = ", c, "\n";
6.    print "Comienzo del programa\n";
7.    while (variable) {
8.              variable = variable - uno;
9.              if (variable) print "no se ha llegado a cero, variable: " , variable, "\n";
10.             else print "Se ha llegado a cero\n";
11.   }
12.
13.   print "Final del programa\n";
14. }
15.
```

Se traduce a una salida en ensamblador:

```
1.  ##################
2.  # Seccion de datos
3.    .data
4.  $str1:
5.    .asciiz "Resultado de 4 * 2 - 18 / 3 * 3 = "
6.  $str2:
7.    .asciiz "\n"
8.  $str3:
9.    .asciiz "Comienzo del programa\n"
10. $str4:
11.   .asciiz "no se ha llegado a cero, variable: "
12. $str5:
13.   .asciiz "\n"
14. $str6:
15.   .asciiz "Se ha llegado a cero\n"
16. $str7:
17.   .asciiz "Final del programa\n"
18. _uno:
19.   .word 0
20. _variable:
21.   .word 0
22. _c:
23.   .word 0
24.
25. ##################
26. # Seccion de codigo
27.   .text
28.   .globl main
29. main:
30.   li $t0, 1
31.   sw $t0, _uno
32.   li $t0, 5
33.   sw $t0, _variable
34.   li $t0, 4
35.   li $t1, 2
36.   mul $t2, $t0, $t1
37.   li $t0, 18
38.   li $t1, 3
39.   div $t3, $t0, $t1
40.   li $t0, 3
41.   mul $t1, $t3, $t0
42.   sub $t0, $t2, $t1
43.   sw $t0, _c
44.   la $a0, $str1
45.   li $v0, 4
46.   syscall
47.   lw $t0, _c
48.   move $a0, $t0
49.   li $v0, 1
50.   syscall
51.   la $a0, $str2
52.   li $v0, 4
53.   syscall
54.   la $a0, $str3
55.   li $v0, 4
56.   syscall
57.   $l3:
58.   lw $t0, _variable
59.   beqz $t0, $l4
60.   lw $t1, _variable
61.   lw $t2, _uno
62.   sub $t3, $t1, $t2
63.   sw $t3, _variable
64.   lw $t1, _variable
65.   beqz $t1, $l1
66.   la $a0, $str4
```

```
67.   li $v0, 4
68.   syscall
69.   lw $t2, _variable
70.   move $a0, $t2
71.   li $v0, 1
72.   syscall
73.   la $a0, $str5
74.   li $v0, 4
75.   syscall
76.   b $l2
77.   $l1:
78.   la $a0, $str6
79.   li $v0, 4
80.   syscall
81.   $l2:
82.   b $l3
83.   $l4:
84.   la $a0, $str7
85.   li $v0, 4
86.   syscall
87.
88. ##############
89. # Fin
90.   li $v0, 10
91.   syscall
92.
```

Salida de la ejecución del código ensamblador:

Resultado de 4 * 2 - 18 / 3 * 3 = -10
Comienzo del programa
no se ha llegado a cero, variable: 4
no se ha llegado a cero, variable: 3
no se ha llegado a cero, variable: 2
no se ha llegado a cero, variable: 1
Se ha llegado a cero
Final del programa

Desarrollo del compilador:

Para la correcta implementación del compilador, se desarrolló en 4 fases:

El analizador léxico:

El analizador léxico agrupa los caracteres del programa fuente en lexemas y genera una lista ordenada de tokens a partir de los caracteres de entrada. Si el analizador léxico encuentra algún lexema que no corresponde a la expresión regular de los tokens del lenguaje, debe informar un mensaje de error léxico.
También el analizador léxico debe encargarse de eliminar símbolos insignificantes para el programa, como los espacios en blanco, tabuladores, comentarios, etc.
Para la implementación de este analizador se utiliza la herramienta Flex donde se encuentran las expresiones regulares asociadas a un bloque de sentencias en C utilizando el archivo `lexico.l`.
Para poder implementar la recuperación de errores se agregó una expresión regular la cual implementa la recuperación de errores en modo pánico que consiste en ignorar los caracteres extraños que puede producir un error hasta encontrar un carácter válido para un nuevo token.
La expresión regular es: `[^[:alnum:]()/\*\\{};,+="_\- \n\t\r]*`
Para la comprobación del límite del número entero, hemos implementado la función `maxnumero()` donde comprobará si un entero introducido es mayor que el máximo entero de 32 bits (2.147.483.647) notificando un mensaje de error léxico.

El analizador sintáctico:

Para la implementación de este analizador se utiliza la herramienta Bison la cual permite definir las reglas de producción de la gramática, asociada a una acción semántica.
En este caso el analizador sintáctico hace un análisis ascendente traduciendo la gramática y además aplicando la asociatividad por la izquierda con la opción %left para los operadores.
Si el programa no contiene una estructura sintáctica correcta, el analizador sintáctico debe notificarlo mediante un mensaje de error sintáctico. Que en este caso se utiliza la función yyerror() que imprime un mensaje de error cuando ocurre un error sintáctico.
Por otra parte, Bison detecta e informa las ambigüedades cuando hay conflictos desplazamiento/reducción o reducción/reducción generando el árbol de todas formas. Lo que se hace es que en un conflicto shift/reduce, se hace un desplazamiento y en un conflicto reduce/reduce, se reduce con la regla de la gramática que aparezca primero en la especificación de Bison.

Como se puede ver, al ejecutar el makefile, sale un aviso de que hay un conflicto desplazamiento/reducción. Esto se produce porque hay un conflicto en la gramática con las secuencias *IF-ELSE*.
Cuando compilamos Bison con la opción -v, se genera un archivo *sintactico.output* el cual tiene más información sobre los conflictos encontrados.
Al comienzo del programa tenemos un aviso de que en state 71 se encuentra un conflicto shift/reduce.
En state 71 nos encontramos con lo siguiente:

```
state 71

   16 state: IF LPAR expr RPAR state . ELSE state
   17      | IF LPAR expr RPAR state .

    ELSE  shift, and go to state 73

    ELSE      [reduce using rule 17 (state)]
    $default  reduce using rule 17 (state)
```

Como se puede ver, Bison lo que aplica para este tipo de conflictos es un desplazamiento.

El analizador semántico:

En el lenguaje miniC que utiliza este compilador, la sintaxis es sensible al contexto y el analizador semántico se encarga de la comprobación de la declaración de variables, donde no puede haber variables declaradas más de una vez, usar variables no declaradas, asignar más de una vez las variables constantes.
Para esto se utiliza una lista enlazada definida en los archivos *listaSimbolos.h* y *listaSimbolos.c* la cual va generando la tabla de símbolos mediante la función *anadeEntrada()* que luego es utilizada para hacer las verificaciones correspondientes mediante las funciones *perteneceTablaS()* y *esConstante()*.

Generación de código:

Esta es la última fase del analizador, donde se genera el código ensamblador. En este caso se utiliza ensamblador MIPS el cual puede ser ejecutado por cualquier compilador de dicho lenguaje.
Para esta parte hemos usado los archivos *listaCodigo.c* y *listaCodigo.h* donde tenemos funciones tanto para crear una lista codigo(*creaLC()*) como para guardar el código ensamblador en un archivo salida llamado *MiPrograma.s.*
También hemos definido algunas funciones:

- Para el control de registros temporales, la función `getRegistroTemporal()` comprueba si hay un registro temporal libre y nos la devuelve como resultado, si no tenemos ningún registro libre finaliza el programa, por último, tenemos también la funciona `liberarRegistro()` donde tomará como entrada un registro y dejará lo dejara libre para ser utilizado.
- Para la creación de operación, usamos la función `creaOperación()` donde le damos los parámetros operación, el registro resultado, el primer argumento y el segundo argumento. Nos devolverá la operación para poder ser insertado.
- Para la impresión en pantalla/Creación del fichero salida, hemos creado las funciones `LCimprimir()` y `guardarArchivoLC()` ambos reciben una lista código de entrada pero uno en vez de crear un archivo nuevo y guardar ahí las operaciones en en `Miprograma.s` lo imprime en pantalla.

Por fines prácticos y por la simplicidad del lenguaje nosotros utilizamos un solo archivo para hacer las últimas 3 fases del compilador haciendo uso del archivo `sintactico.y`.