Compilador Mini-C

Compiladores – Curso 2022/23

Convocatoria de Junio

Jose Luis Galiano Gómez
joseluis.galianog@um.es

Miguel Lucas Serrano
miguel.lucass@um.es

Grupo 1.3

Índice

1. Introducción

El proyecto de prácticas consiste en la elaboración de un compilador para el lenguaje Mini-C, versión más sencilla de C. Utilizaremos para ello las siguientes herramientas: Flex, para el analizador léxico, Bison para el sintáctico y las clases `ListaCodigo` y `ListaSimbolos` proporcionadas por los profesores de la asignatura para el semántico.

2. Estructura del compilador

El compilador se compone de los siguientes ficheros:

· **makefile**: Este es el fichero responsable de compilar correctamente el compilador, haciendo uso del compilador de C gcc y creando los ficheros en el orden correcto. Además, elimina los ficheros innecesarios para el compilador final generados durante la compilación de este (clear).

· **main:** Este fichero contiene la función que se inicia al ejecutar el compilador, por lo tanto declara las variables necesarias, lee el fichero que contiene el código Mini-C a compilar a partir de su nombre, y lanza el analizador léxico con la función de Bison yyparse(). Esta a su vez llama a la función de Flex yylex() para ir obteniendo los tokens, iniciando así la compilación. Finalmente, esta función comprueba cuántos errores se han producido en la compilación, mostrando la cantidad por la salida de error (después de los errores en sí).

· **lexico.l:** El analizador léxico implementado con Flex. Va devolviendo al analizador sintáctico/semántico los tokens conforme los encuentra.

· **sintactico.y:** El analizador sintáctico/semántico implementado con Bison. Define la gramática del lenguaje así como las acciones semánticas a realizar tras reconocer una cierta estructura sintáctica.

3. Análisis Léxico (Flex)

El análisis léxico se basa en el reconocimiento de los **tokens** definidos para la gramática mediante sus expresiones regulares. Una vez reconocido un token mediante su expresión regular, simplemente se pasa al analizador sintáctico con un simple return. Sin embargo, hay algunos tokens que, una vez reconocidos, necesitan de ciertas acciones adicionales.

Los **errores léxicos** se contabilizan en este fichero, declarando una variable numErroresLéxicos que será aumentada cada vez que se encuentre uno.

El primer error léxico que podemos encontrar es a la hora de declarar un identificador, los cuales, en Mini-C, no pueden ser superiores a 16 caracteres. Usamos la variable yyleng para averiguar la longitud del identificador y de superarse el límite, informamos del error.

Otro posible error léxico es a la hora de declarar un entero, tipo que tiene un límite. Si el entero es demasiado grande, se informa del error.

Cuando no se reconocen los token de entrada se entra en **modo pánico**, el cual se implementa de manera sencilla; con una expresión regular que lo acepta todo. Declarando el reconocimiento de esta al final del fichero, nos aseguramos de que esta se acepte sólo si no se ha podido aceptar ninguno de los tokens declarados previamente. De esta manera, en cuanto vuelva a aparecer un token aceptado el analizador se recuperará automáticamente y sin problema.

También destacamos los **comentarios multilínea**, los cuales necesitan de un cambio de contexto. Declaramos el contexto COMMULTI con %x para que sólo las reglas cualificadas con la condición de arranque <COMMULTI> se tengan en cuenta, y estas consisten en ignorar todo lo que se ajuste a la expresión regular proporcionada por los profesores para el comentario multilínea, y en volver al contexto INITIAL una vez reconocida la expresión regular de fin de comentario (también ya proporcionada). Si llegamos al final del fichero en el contexto COMMULTI, estamos ante un comentario multilínea sin cerrar, por lo que aumentamos el número de errores léxicos. Finalmente, para poder proporcionar un mejor mensaje de error, al iniciar el comentario usamos la variable yylineno de Flex para almacenar la línea donde se inicia el comentario, y así poder informar al usuario.

4. Análisis Sintáctico (Bison)

Tanto el análisis sintáctico como el semántico se realizan en el mismo fichero, sintactico.y, aunque la parte correspondiente al análisis sintáctico es fácilmente reconocible.

Primero se declara la variable numErroresSintacticos, que como su nombre indica contabiliza los **errores sintácticos** encontrados, los cuales se manejan mediante la función de Bison yyerror, donde se aumenta la variable además de mostrar el mensaje de error.

El reconocimiento de la gramática se realiza implementando con Bison la gramática en notación BNF proporcionada por los profesores, a la que también añadimos **puntos de sincronización**, más concretamente en las declaraciones (no-terminal declarations) y en las sentencias (no-terminal statement).

Son importantes las precedencias de los operadores, necesarias para evitar **conflictos**. Estas se resuelven en el analizador; simplemente se asocia siempre por la izquierda, dando precedencia máxima al operador unario - (UMINUS) seguido por la multiplicación y división (* y /) y finalmente la menor precedencia para la suma y resta (+ y -). Se usa la opción %expect 1 para aceptar conflictos.

5. Análisis Semántico (Tabla de símbolos)

Para llevar a cabo el análisis semántico utilizamos la clase ListaSimbolos proporcionada por los profesores, así como una serie de funciones adicionales definidas en sintactico.y, las cuales hacen uso de las que ya aparecen en la clase. Estas funciones son las siguientes:

establecerTipoLS es una función muy sencilla que únicamente establece el tipo del siguiente símbolo a insertar. Es necesario declarar una variable global TipoLS que indique el tipo del siguiente elemento a insertar en la lista, ya que la determinación del tipo del símbolo y la inserción se hacen en puntos distintos de la gramática (declarations y asig, respectivamente).

imprimeLS imprime la lista de símbolos, formateándola ya como la sección de datos del programa. Esta es usada al final del reconocimiento de la gramática, es decir, al final de las acciones asociadas al no-terminal program (que abarca todo el programa). También cabe destacar ahora que se crea la lista de símbolos, declarada como variable global, nada más reconocer el no-terminal program, es decir, al iniciar el análisis sintáctico. Tras imprimir la lista de símbolos, se libera la memoria ocupada.

contieneLS busca si un cierto símbolo está ya en la lista, y esta función se usa para detectar variables declaradas o, por el contrario sin declarar, lo cual constituye un error si se trata de una variable declarada anteriormente o si se está intentando utilizar una variable, respectivamente. **comprobarConstanteLS** sirve también para detectar errores semánticos, más concretamente asignaciones a constantes fuera de su declaración, lo cual no está permitido.

insertaEntradaLS simplemente inserta los símbolos en la tabla. La función hace uso de TipoLS, variable global manejada por la función establecerTipoLS, y es importante diferenciar entre el caso de que se inserte una variable o constante (en cuyo caso hay que usar TipoLS), y el caso de que se inserte una cadena, ya que esto implica, además de insertar el símbolo, quitar las comillas de la cadena y devolver el nombre de la cadena, necesario para la sección de datos. Estas se identifican sencillamente por el orden en el que fueron insertadas, por lo que se tiene una variable global contadorCadenas que se aumenta cada vez que se llama a esta función.

Antes de continuar con la generación de código, cabe destacar la defincón de la **unión** que nos permite soportar los tipos str y codigo, necesarios para aquellos tokens que van a la tabla de símbolos (id, int y string) y las listas de código, respectivamente. Además declara la variable externa yylval usada en el analizador léxico.

6. Generación de código (Listas de código)

El uso de las listas de código para la generación de la sección de código en la salida final consiste en lo siguiente: cada símbolo no-terminal de la gramática cuenta con una lista de código asociada, almacenada en el atributo $$. En las reglas de producción, unimos las listas de código asociadas a los no-terminales que participan en la producción y las concatenamos, para después asociarlas al no-terminal de la izquierda de la regla de producción, efectuando así las **reducciones**. Según la regla concreta, tendremos que añadir (o no) ciertas operaciones.

Para un uso más cómodo de las **operaciones**, se han declarado unas funciones que nos permiten crearlas de manera sencilla, especificando el tipo de operación y sus campos (el tipo viene definido por la cantidad de campos).

Al igual que en el análisis semántico, definimos una serie de funciones que usaremos para manejar el correcto funcionamiento de la generación de código:

obtenRegTemp y **liberaRegTemp** son funciones necesarias para no exceder el número máximo de registros temporales disponibles, así como para correctamente asociar a cada operación qué registros temporales va a usar exactamente, si los necesita. Simplemente se declara un array de enteros (que usaremos como booleanos) para indicar si cada registro está ocupado o no.

obtenEtiqueta devuelve el nombre de la nueva etiqueta creada que, de manera similar a las cadenas, es el número de etiquetas en el programa hasta ese punto, el cual se maneja con la variable contadorEtiquetas. Al obtener una etiqueta esta se devuelve para poder almacenarla en el campo res de la operación (importante a la hora de imprimir).

Finalmente, **imprimeLC** imprime la lista de código final (asociada a program) formateada como sección de código. Cabe destacar el tratamiento especial de las etiquetas, de manera que si en el campo op de una operación encontramos “etiq”, significa que estamos ante una etiqueta y por lo tanto no hay que imprimir los campos de la operación como de costumbre, sólo la etiqueta en sí (almacenada en res).

Un último detalle a mencionar; la directiva %code es utilizada para resolver la dependencia del analizador sintáctico del tipo ListaC debida a la unión.

7. Manual de uso del usario

El uso es sencillo, para compilar el compilador basta con lanzar el Makefile con la instrucción `make` desde una terminal. Posteriormente, para compilar un fichero dado hay que ejecutar el compilador desde la terminal también. Seguidamente se esperará a que especifiques la ruta del fichero a compilar y su versión en ensamblador se mostrará por pantalla. Para poder probar el código, es recomendable redirigir la salida a un fichero de código ensamblador que luego podrá ser usado en cualquier programa que permita la ejecución del mismo para comprobar su correcto funcionamiento (para realizar la práctica se usó spim). Si además redirigimos la salida de error a otro fichero distinto, tendremos en él los errores detectados durante la compilación. Con todo esto, un uso estándar del compilador se vería así:

```
> make

> ./compilador > prueba.s 2> errores

> prueba.mc

> usr/bin/spim -file prueba.s
```

8. Ejemplo de funcionamiento

Procedemos a mostrar un ejemplo de programa en Mini-C proporcionado por los profesores. Este es el código original en Mini-C:

```
void prueba() {
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 while (c)
            {
                  print "c = ",c,"\n";
                  c = c-2+1;
            }
  print "Final","\n";
}
```

Esta la salida del compilador:

```
##################
# 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, $l5
      la $a0, $str2
      li $v0, 4
      syscall
      la $a0, $str3
      li $v0, 4
      syscall
      b $l6
$l5:
      lw $t1, _b
      beqz $t1, $l3
      la $a0, $str4
      li $v0, 4
      syscall
      b $l4
$l3:
$l1:
      lw $t2, _c
      beqz $t2, $l2
      la $a0, $str5
      li $v0, 4
      syscall
```

```
      lw $t3, _c
      move $a0, $t3
      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
      b $l1
$l2:
$l4:
$l6:
      la $a0, $str7
      li $v0, 4
      syscall
      la $a0, $str8
      li $v0, 4
      syscall

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

Y esta la salida de spim, la cual muestra que el programa se comporta como se espera a partir del código original en Mini-C:

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