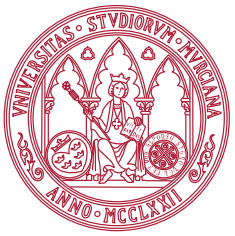

Universidad de Murcia
Grado en Ingeniería Informática

Compiladores
Práctica de traducción de MiniC a código ensamblador de MIPS (MiniC Compiler)
Curso 2018 – 2019
(Convocatoria de junio)

Memoria de la práctica

Juan Francisco Carrión Molina
juanf.carrion@um.es
Grupo 1, subgrupo 1.1

Profesor
Eduardo Martínez Graciá

Mayo de 2019

Índice

1. Introducción

El compilador de MiniC constituye el proyecto de prácticas para la asignatura de Compiladores. Se ponen a prueba y se acompañan los conocimientos adquiridos en la parte teórica de la asignatura, así como otros nuevos.

En los siguientes apartados se especifica el lenguaje MiniC y las etapas de su traducción a código ensamblador de MIPS, así como el proceso de diseño del traductor.

Aunque lo especificaremos en el correspondiente apartado, el lanzamiento y enlace de todas las herramientas se realiza desde el fichero `mccMain.c` y se coordina desde el fichero `makefile`.

2. Lenguaje MiniC

MiniC es un lenguaje parecido a C aunque más reducido en diversos aspectos. Solo maneja constantes y variables enteras. De esta manera, los tipos booleanos se representan con enteros, siendo 0 el valor *falso* y el resto *verdadero*. No existen operadores relacionales ni lógicos y las sentencias de control del flujo de ejecución se reducen a `if`, `if-else`, `while` y `do-while`.

2.1. Símbolos terminales

El analizador sintáctico hace uso de una gramática que veremos más adelante. Esta gramática está compuesta por 23 símbolos terminales que permiten, en primer lugar, el análisis léxico del programa fuente. Son los siguientes.

Los enteros, *num* (token `LITINT` en el código), pueden tener un valor desde -2^31 hasta 2^31. También existen cadenas de texto, *string* (token `LITSTR` en el código), delimitadas por comillas dobles.

Los identificadores, *id* (token `ID` en el código), están formados por secuencias de letras, dígitos y símbolos de subrayado, no comenzando por dígito y no excediendo los 16 caracteres.

Las palabras reservadas son `func` (`T_FUNC`), `var` (`T_VAR`), `const` (`T_CONST`), `if` (`T_IF`), `else` (`T_ELSE`), `while` (`T_WHILE`), `do` (`T_DO`), `print` (`T_PRINT`) y `read` (`T_READ`).

Por último, disponemos de los caracteres especiales de separación: `;` (`T_SMCLN`) y `,` (`T_COMMA`); de operaciones aritméticas: `+` (`T_PLUS`), `-` (`T_SUBS`), `*` (`T_MULT`) y `/` (`T_DIVI`); de asignación: `=` (`T_ASSIGN`); y de control precedencia y bloques: `(` (`T_PARL`), `)` (`T_PARR`), `{` (`T_BRKL`) y `}` (`T_BRKR`).

2.2. Gramática para el análisis sintáctico

Nuestro MiniC está basado en una gramática libre de contexto que permite al analizador sintáctico comprobar la corrección del programa fuente. A continuación se muestra dicha gramática en notación BNF y con los `símbolos terminales` diferenciados, al igual que en el apartado anterior. Algunos de esos *símbolos terminales* están resaltados ya que son especiales, porque necesitamos su lexema para el análisis semántico.

program	→	`func` *id* `(` `)` { declarations statement_list }
declarations	→	declarations `var` identifier_list `;`
	|	declarations `const` identifier_list `;`
	|	λ
identifier_list	→	asig
	|	identifier_list `,` asig
asig	→	*id*
	|	*id* `=` expression
statement_list	→	statement_list statement
	|	λ
statement	→	*id* `=` expression `;`
	|	{ statement_list }
	|	`if` `(` expression `)` statement `else` statement
	|	`if` `(` expression `)` statement
	|	`while` `(` expression `)` statement
	|	`do` statement `while` `(` expression `)` `;`
	|	`print` print_list `;`
	|	`read` read_list `;`
print_list	→	print_item
	|	print_list `,` print_item
print_item	→	expression
	|	*string*
read_list	→	*id*
	|	read_list `,` *id*
expression	→	expression `+` expression
	|	expression `-` expression
	|	expression `*` expression
	|	expression `/` expression
	|	`-` expression
	|	`(` expression `)`
	|	*id*
	|	*num*

3. Análisis léxico (herramienta **Flex**)

En primer lugar, diseñamos nuestro analizador léxico. Utilizamos, para ello, la herramienta Flex, que precisamente genera analizadores léxicos. Todo el desarrollo de esta parte se encuentra en el fichero `mccLexicalAnalyzer.l`.

Para comenzar, identificamos los tokens que hemos definido en la gramática del lenguaje. Este proceso se basa en las expresiones regulares correspondientes a cada token. Reconocemos y eliminamos, además, comentarios de una o varias líneas y separadores (espacios en blanco, tabuladores y retornos de carro.

También comprobamos en este apartado la corrección de identificadores y literales enteros, según

lo especificado, e implementamos la detección de errores en modo pánico. Si existe algún problema en el análisis, se informará de un error léxico. Además, pasamos a la siguiente fase del análisis los lexemas de los símbolos terminales *id* (`T_ID`), *str* (`T_LITSTR`) y *num* (`T_LITINT`).

4. Análisis sintáctico y semántico y generación de código (herramienta **Bison**)

La siguiente fase sería el analizador sintáctico. Sin embargo, combinamos esta y las dos últimas en una misma sección ya que las especificamos dentro de un mismo fichero `mccSyntacticAnalyzer.y`. Este fichero pertenece a la herramienta Bison, que nos permite generar el analizador sintáctico y, además, nos incluir lo necesario para guiar las tareas de análisis semántico y generación de código.

4.1. Análisis sintáctico

Por la parte del analizador sintáctico, la función llevada a cabo es reconocer sintácticamente ficheros generados por la gramática que hemos mostrado anteriormente. Se establecen las precedencias necesarias en los operadores y, además, se inserta una opción para que el analizador acepte un único conflicto de reducción, el de las sentencias `if-else`.

También se realizan algunas recuperaciones de errores mediante puntos de sincronización para facilitar la corrección de errores en un programa escrito en MiniC. Si existe algún problema en el análisis, se informará de un error sintáctico.

4.2. Análisis semántico (contenedor **SymbolTable**)

Para realizar el análisis semántico, definimos una tabla de símbolos mediante la estructura contenedora `SymbolTable`. Esta se trata de una lista simplemente enlazada que contiene instancias del tipo `Symbol`. Este tipo almacena el nombre, el tipo y el valor de un símbolo, pudiendo almacenar variables (`SYMVAR`), constantes (`SYMCONST`) y, como veremos ahora, cadenas de texto (`SYMSTR`).

En cuanto a manipulación de la tabla de símbolos, hemos reducido los métodos a algunos básicos, de manera que podemos crearla (`SymbolTableCreate()`), insertar elementos (`SymbolTableInsert()`) y liberarla (`SymbolTableFree()`). También podemos imprimirla (`SymbolTablePrint()`) y comprobar si un símbolo existe en ella (`SymbolTableContains()`) y si es constante (`SymbolTableCheckConstant()`).

Como hemos explicado en el apartado anterior, desde el analizador léxico (referencia al campo `str` de la `union`) llegan, además de los tokens, algunos atributos de símbolos terminales. Estos atributos constituyen el nombre del símbolo a insertar.

Reutilizamos el tipo `Symbol` para almacenar cadenas de texto (en el campo nombre) que luego imprimiremos, junto a los símbolos de tipos variable y constante, en el segmento de datos de nuestro resultado en ensamblador de MIPS. Para la inserción de cadenas, disponemos de un método especial `SymbolTableInsertString()` que nos permite implementar la mejora de no volver a insertar dos cadenas iguales y que, además, se encarga de controlar los identificadores de las cadenas.

Bison nos permite realizar acciones dentro de la gramática que se ejecutarán a medida que se produzcan las correspondientes reducciones. Así, para implementar las acciones de manipulación de la tabla de símbolos, entre llaves en cada regla de producción, insertamos la funcionalidad necesaria.

En las secciones de declaración, el analizador semántico se encargará de comprobar si un elemento ya estaba insertado en la tabla y lanzar un error, o no e insertarlo correctamente. Para saber el tipo que hay que insertar, modificamos las acciones para que cuando se lee un símbolo se ejecuta la función `SymbolTableSetCurrentType()`, que luego utiliza la función de inserción.

En las secciones de asignación, el analizador semántico comprueba si el símbolo al que se asigna existe y es variable.

Al igual que el código final, la tabla de símbolos solo se imprimirá si no se ha generado ningún error de ningún tipo. Esta comprobación se realiza en el programa principal `mccMain.c`.

4.3. Generación de código (contenedor **CodeList**)

Para implementar el ensamblaje de MiniC, utilizamos un subconjunto del código ensamblador de MIPS. En el fichero ensamblador final, se vuelca, en primer lugar, una representación de la tabla de símbolos en el segmento de datos (`.data`). Aquí se declaran las cadenas de texto (`.asciiz`) y las variables enteras globales (`.word` de 32 bits inicializada a 0 y con el identificador precedido del carácter _ para diferenciarlo de las instrucciones de MIPS).

A continuación, comienza el segmento de texto que contiene las instrucciones del código ensamblador. Se define el punto de entrada al programa (`.globl main`) y se imprime la lista de código generada.

Para almacenar las instrucciones de MIPS generadas en la última etapa del compilador, utilizamos un tipo `Instruction` y un contenedor de tipo lista `CodeList`.

El tipo `Instruction` está simplemente compuesto por cuatro cadenas de texto, pudiendo representar todas las instrucciones de MIPS. Estos campos son `op` (código de la operación), `res` (resultado de la operación), `arg1` (primer argumento de la operación) y `arg2` (segundo argumento de la operación). Si alguno de ellos no se usa, se marca como `NULL`.

Aprovechamos el tipo `Instruction` para almacenar las etiquetas generadas en las listas de código, asignando *etiq* el campo `op` para que, a la hora de imprimir la lista de código general, podamos tratarlas adecuadamente.

En cuanto a la lista de código, cada símbolo no terminal de la gramática dispone de su propia `CodeList`, almacenada en su correspondiente atributo `$$` (referencia al campo `codigo` de la `union`). Mediante las reglas de producción, en las reducciones, nos encargamos de crear las listas de código (`CodeListCreate()`), liberarlas (`CodeListFree()`), añadirles operaciones (`CodeListInsert()`) y concatenarlas con las listas hijas (`CodeListJoin()`), según las necesidades de cada caso.

El contenedor dispone también de las funciones necesarias para realizar el control de las etiquetas de las secciones de código (`CodeListGenerateLabel()`), así como de los registros temporales (`CodeListGetAvailableTemporaryRegister()` y `CodeListReleaseTemporaryRegister()`), incluidos los de resultados de expresiones a utilizar en las siguientes líneas de código, por ejemplo en comprobaciones de `if` (`CodeListSetResultRegister()` y `CodeListGetResultRegister()`).

5. Manual de uso

5.1. Compilación inicial de MiniC Compiler

Para generar el compilador desde Ubuntu, utilizamos una shell. Primeramente, nos colocamos en el directorio del proyecto. Ahora, ejecutamos la orden `make`. Este programa se encargará ahora de seguir las directivas establecidas en el fichero `makefile` a través de un autómata para generar el programa objeto de MiniC Compiler, `mcc`.

Adicionalmente, podemos ejecutar `make` con los parámetros `clear` para limpiar los restos de la compilación o `run` para, una vez generado el programa objeto, ejecutarlo con un fichero de entrada `test.mc` y generar un fichero de salida `test.s` en el mismo directorio.

5.2. Uso básico de MiniC Compiler

Una vez disponemos del prorgama objeto `mcc`, podemos ejecutarlo para compilar algún programa escrito en MiniC y almacenar el resultado en un fichero de código ensamblador MIPS de la siguiente manera.

```
./mcc source.mc > target.s
```

5.3. Prueba del código final

Para probar nuestro fichero resultado en esamblador de MIPS, podemos utilizar los simuladores SPIM o MARS, tanto en sus versiones gráficas como de consola.

```
./spim -file target.s
```

Es necesario destacar que el compilador está implementado para generar una terminación de programa con llamada al *caller*, es decir, utilizando el salto de contador de progama `jr $ra`. Esto generará un error si se utiliza MARS, ya que el código ejecutado por este no tiene un *caller*, siendo nula la dirección de retorno. Si quisiéramos arreglar esto, tendríamos que sustituir esa instrucción por la `syscall` número 10, que termina la ejecución.

5.4. Ejemplo

A continuación se plantea un ejemplo de código origen en MiniC (fichero adjunto `test.mc`).

```
func prueba () {
    const a = 1;
    const b = 2 * 3;
    var c;
    var d = 5 + 2, e = 9 / 3;

    print "Inicio del programa\n";

    print "Introduce el valor de \"c\":\n";
    read c;

    if (c) print "\"c\" no era nulo.", "\n";
    else print "\"c\" si era nulo.", "\n";

    /* Imprimir d */
    while (d) {
        print "\"d\" vale", d, "\n";
        d = d - 1;
    }

    /* Imprimir e */
    do {
        print "\"e\" vale", e, "\n";
        e = e - 1;
    } while(e);

  print "Final","\n";
}
```

Y su correspondiente fichero resultado de la compilación en ensamblador de MIPS (fichero adjunto `test.s`).

```
  .data
$str1:
  .asciiz "Inicio del programa\n"
$str2:
  .asciiz "Introduce el valor de \"c\":\n"
$str3:
  .asciiz "\"c\" no era nulo."
$str4:
  .asciiz "\n"
$str5:
  .asciiz "\"c\" si era nulo."
$str6:
  .asciiz "\"d\" vale"
$str7:
  .asciiz "\"e\" vale"
$str8:
  .asciiz "Final"
_a:
  .word 0
_b:
  .word 0
_c:
  .word 0
_d:
  .word 0
_e:
  .word 0

  .text
  .globl main

main:
  li $t0, 1
  sw $t0, _a
  li $t0, 2
  li $t1, 3
  mul $t2, $t0, $t1
  sw $t2, _b
  li $t0, 5
  li $t1, 2
  add $t2, $t0, $t1
  sw $t2, _d
  li $t0, 9
  li $t1, 3
  div $t2, $t0, $t1
  sw $t2, _e
  la $a0, $str1
  li $v0, 4
  syscall
  la $a0, $str2
  li $v0, 4
  syscall
  li $v0, 5
  syscall
  sw $v0, _c
  lw $t0, _c
  beqz $t0, $l1
  la $a0, $str3
  li $v0, 4
  syscall
  la $a0, $str4
  li $v0, 4
  syscall
```

```
  b $l2
$l1:
  la $a0, $str5
  li $v0, 4
  syscall
  la $a0, $str4
  li $v0, 4
  syscall
$l2:
$l3:
  lw $t0, _d
  beqz $t0, $l4
  la $a0, $str6
  li $v0, 4
  syscall
  lw $t1, _d
  move $a0, $t1
  li $v0, 1
  syscall
  la $a0, $str4
  li $v0, 4
  syscall
  lw $t1, _d
  li $t2, 1
  sub $t3, $t1, $t2
  sw $t3, _d
  b $l3
$l4:
$l5:
  la $a0, $str7
  li $v0, 4
  syscall
  lw $t0, _e
  move $a0, $t0
  li $v0, 1
  syscall
  la $a0, $str4
  li $v0, 4
  syscall
  lw $t0, _e
  li $t1, 1
  sub $t2, $t0, $t1
  sw $t2, _e
  lw $t0, _e
  bnez $t0, $l5
  la $a0, $str8
  li $v0, 4
  syscall
  la $a0, $str4
  li $v0, 4
  syscall

  jr $ra
```

6. Conclusiones

La asignatura de Compiladores recorre de forma básica la construcción de traductores de código para lenguajes de programación. Se aprenden, así, las fases del proceso de traducción, la organización de los programas y las técnicas necesarias para resolver los posibles problemas derivados de esta actividad.

A mi parecer, este proyecto de prácticas tiene la importante finalidad de generar, desde mi posición

como estudiante, un conocimiento propio, para así poder desarrollar y asimilar lo estudiado de forma teórica en esta asignatura y en las de su mismo área.

Creo, además, que los contenidos que se cubren en la asignatura de Compiladores son de gran utilidad para el desarrollo de un ingeniero informático, siendo este un potencial programador y necesitando conocer cómo se realiza la construcción de los lenguajes y su ensamblado a la máquina que ejecutará los programas diseñados.

Con respecto al desarrollo de las sesiones de laboratorio, me gustaría destacar la comodidad con la que he podido trabajar, tanto en referencia al profesorado como a los recursos docentes. La documentación sobre las herramientas utilizadas (presentaciones y ejemplos) es muy ilustrativa y está muy bien organizada. También la planificación me ha parecido correcta ya que ha permitido avanzar entre los apartados de la práctica de forma coherente y fluida.

Personalmente, hubo algunos momentos en los que perdí el hilo de la realización del proyecto, pero de nuevo destaco mi agradecimiento al profesorado por mostrarse completamente disponible para resolver cualquier duda.