MEMORIA COMPILADOR DEL LENGUAJE MINIC A ENSAMBLADOR MIPS

Compiladores, 2º Ingeniería en Informática, Universidad de Murcia

Autores

Alberto Gálvez Gálvez – alberto.galvezg@um.es – albertogg023@gmail.com
Jaime Sánchez Sánchez – jaime.sanchezs@um.es – jaimesan701@gmail.com

ÍNDICE

Introducción

El compilador de *MiniC* se trata del proyecto de prácticas de la asignatura de Compiladores, perteneciente al segundo cuatrimestre del segundo curso del Grado en Ingeniería en Informática de la Universidad de Murcia.

En los siguientes apartados se explica la sintaxis del lenguaje *MiniC*, las etapas del desarrollo del compilador y como usar este.

Lenguaje MiniC

MiniC es un lenguaje semejante a *C,* aunque muchísimo más simple en todos sus aspectos. En él, tan solo se tiene una función principal y se pueden declarar y usar constantes y variables enteras. Por esto último, los tipos booleanos se representan con enteros, siendo *0* el valor falso y cualquier otro valor 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*.

```
void ejemploMiniC() {
    var a, b, c;
    const X = 3;

    print "Introduce el valor de 'a'\n";
    read  a;
    print "Introduce el valor de 'b'\n";
    read  b;
    print "Introduce el valor de 'c'\n";
    read  c;

    do {
        print a, "\n";
        a = a - 1;
    } while(a)

    if(b){
        print "Hola, soy el cuerpo del if\n";
    }
    print "Hola, soy la salida del if\n";

    if(c){
        print "Hola, soy el cuerpo del if-else\n";
    }else {
        print "Hola, soy el else del if-else\n";
    }
    print "Hola, soy la salida del if-else\n";

    print a*X;

    /*
     ESTO ES UN COMENTARIO MULTILÍNEA
    */
    // ESTO ES UN COMENTARIO EN LÍNEA
}
```

Símbolos terminales

Los analizadores léxico y sintáctico hacen uso de la gramática del siguiente apartado. Esta gramática se compone de 23 símbolos terminales:

- Números enteros: *num* pueden tener un valor desde -2^{31} hasta 2^{31}.
- Cadenas de texto: *string* delimitadas por comillas dobles
- Identificadores: *id*, formados por secuencias de letras, dígitos y símbolos de subrayado, no comenzando por dígito y no excediendo los 16 caracteres
- Palabras reservadas: *void*, *var*, *const*, *if*, *else*, *while*, *do*, *print* y *read*
- Separación: *;* y *,*
- Operadores aritméticos: *+*, *-*, *** y */*
- Asignación: *=*
- Control de precedencia y bloques: *(*, *)*, *{* y *}*

Gramática para el análisis sintáctico

El lenguaje *MiniC* se basa en una gramática libre de contexto que permite al analizador sintáctico comprobar la corrección del programa fuente. A continuación, se muestra esta gramática en notación *BNF*:

program	→	`void` *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
	|	`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*

Análisis léxico

En primer lugar, diseñamos nuestro analizador léxico con la potente herramienta Flex. Todo el desarrollo de esta parte se encuentra en el fichero *analizadorLexico.l*.

El análisis léxico consiste en identificar y presentar al analizador sintáctico los tokens de la gramática mediante el uso de expresiones regulares (*Regex*). Cabe destacar, la omisión de la presentación al analizador sintáctico de los separadores, los comentarios en línea y multilínea. Así, cabe mencionar también, la comprobación de la corrección de ciertos tokens (desbordamiento de los enteros, longitud máxima de los identificadores) y de no serlo, informar al usuario. Además, si se encuentra un comentario o un *string* sin cerrar, se avisa también al usuario.

También, en caso de detectar algo que no sea un terminal de la gramática, implementamos recuperación de errores mediante el modo pánico, el cual informa al usuario del error en el código y localiza el siguiente comienzo de un token correcto.

Análisis sintáctico, análisis semántico y generación del código del programa en lenguaje ensamblador MIPS

La siguiente de las fases sería la del análisis sintáctico. Sin embargo, combinamos esta y las dos últimas fases del desarrollo del compilador en este apartado, ya que estas vienen implementadas en el fichero *analSintSemGenCodigo.y*.

Análisis sintáctico

En cuanto al analizador sintáctico, su tarea es reconocer las agrupaciones entre los distintos tokens (proporcionados por el analizador semántico) de acuerdo con la gramática, formando operaciones aritméticas, sentencias de control, etc.

Cabe destacar que, para poder establecer las prioridades necesarias entre los distintos operadores, hacemos uso de la directiva *%left operador*. Otra directiva usada es *%expect 1* para que el analizador acepte (es decir, no informe al usuario sobre el conflicto) un único conflicto *desplazamiento/reducción* de la gramática, el de las sentencias *if-else*. Por defecto, *Bison*, frente a este tipo de conflictos, elige desplazar, lo cual es correcto en este tipo de conflicto, al considerar que le gramática debe asociar el *else* con el último *if*, no con el primero (que sería lo que pasaría si se optase por reducir en este conflicto).

También, implementamos recuperación de errores en ciertas sentencias, informando al usuario del error y avanzando en la entrada hasta encontrar cierto token (dependiendo del caso). Por ejemplo, en caso de que se detecte un error en *statement*, se avanza hasta encontrar el siguiente punto y coma. Esta técnica la realizamos con *identifier_list*, *asig*, *statement*, *print_list*, *read_list* y *expression*.

Análisis semántico

Con respecto al análisis semántico, su tarea es por una parte la de evitar inconsistencias en el código, tales como usar variables o constantes sin declarar, redeclararlas o intentar cambiar el valor de una constante. Por otra parte, otra tarea de este tipo de análisis es la de generar la cabecera *.data* que tiene un programa en *MIPS*.

Para este análisis, hacemos uso del fichero *listaSimbolos.h*, el cual nos proporciona una tabla de símbolos (*Lista*) y diversas funciones y procedimientos. Esta lista se trata de una lista simplemente enlazada, cuyos nodos almacenan el nombre del símbolo, su tipo y su valor.

El uso que hacemos de este fichero en el analizador sintáctico como tal, es el de:

- Insertar los identificadores declarados, comprobando que no se redeclaren también. Estos se insertan con su nombre de identificador, el tipo de identificador (variable o constante) y un valor arbitrario (no es significativo). Esto se realiza mediante el método *void anadeEntradaLS*.

- Cuando se hace uso de un identificador, evitar que se haga de uno que no exista, así como evitar que se cambie el valor de una constante. Esto se hace mediante el método *int perteneceTablaLS*, el cual, si el identificador que se consulta no existe, devuelve *-1*; si existe y no coinciden los tipos, devuelve *0*; y si existe y además coinciden los tipos, devuelve *1*. El motivo por el que usamos esta función (que a priori puede parecer extraña), es el de evitar usar dos listas, una para variables y otras para constantes, o tener funciones auxiliares que nos digan si lo que nos devuelve *busquedaLS* es una variable o una constante. Dicho esto, las consultas las hacemos como si lo que consultásemos fuese una variable. Así, por ejemplo, al intentar cambiar el valor de una constante, se vería que el identificador existe, pero no coinciden los tipos, así avisando al usuario.

- Cuando se hace un reconoce un *print*, insertar el *string* en la tabla de símbolos. En este caso, el campo valor ya es significativo, teniendo que asignarle el contador de cadenas en tal instante.

La impresión de la tabla de símbolos la mencionamos en el siguiente apartado

Generación de código

En cuanto a la generación de código, su tarea es la generar el código del programa *.text*.

Para esta generación de código, hacemos uso del fichero *listaCodigo.h*, el cual nos proporciona una lista (*listaC*) de operaciones (*Operacion*) y diversas funciones y procedimientos. Esta lista se trata de una lista simplemente enlazada, cuyos nodos almacenan cuatro cadenas de texto, pudiendo representar todas las instrucciones de *MIPS*. Estos campos son *op* (nombre de la instrucción), res (resultado de la instrucción), *arg1* (primer argumento de la instrucción) y *arg2* (segundo argumento de la instrucción). Si la instrucción traducida no utiliza alguno de sus operandos, se le asigna el valor *NULL*.

En cuanto al control sobre los registros del procesador, recordemos que disponemos de 10 registros *($t0..$t9*). Por ello, hay que llevar un control sobre cuales están siendo usados y cuales están libres. Para ello, se han implementado: un procedimiento *void inicializarRegistros()*, el cual inicializa una array de registros libres; y una función *char * getReg(),* que pone el primer registro que haya libre como ocupado y devuelve una cadena de texto con el registro libre. Esta última función, en caso de que no haya un registro libre, aborta la ejecución del programa indicando que no había registros suficientes para poder traducir el programa.

Ahora bien, insertamos las instrucciones y etiquetas (correspondientes en *MIPS*) en esta lista cuando el analizador sintáctico aplique una regla.

Cabe mencionar que para la generación de las etiquetas de las sentencias condicionales y saltos, hacemos uso de la función *char * getEtiq(),* la cual devuelve una cadena con la etiqueta generada concatenando los caracteres *$et* con el contador de etiquetas en tal instante. En cuanto al almacenamiento de estas, usamos también la lista, para así evitar incompatibilidades, trabajo extra que pueda complicar más el proyecto y reescribir partes de este. Por último, las ponemos en la parte del código donde corresponda, concatenándole ‘:’ a estas mediante la función *char * concatDosPuntos(char * etiq)*.

Hay que destacar también, que a los identificadores en *MIPS* se les concatena una barra baja ‘_’, por ello, cuando se va a generar una instrucción que hace uso de una variable, le anteponemos al nombre de la variable la barra baja mediante la función *char * anteponBarraBaja(char * var)*.

A continuación, comienza el segmento de texto que contiene las instrucciones del código ensamblador y se imprime la lista de código generada.

Una vez generado el código del programa *.text*, procedemos a imprimir todo el código *MIPS* en el fichero ensamblador final (siempre que no haya habido errores de compilación). Primero imprimimos la cabecera *.data* con el procedimiento *void imprimeTablaLS()*, a continuación el punto de entrada al programa *.globl main* y por último el código *.text* con el procedimiento *void imprimeCodigoLC()*.

Guía de uso

Compilación del compilador

Para generar el compilador desde Linux, haremos uso del *Bash*. Primeramente, nos colocamos en el directorio donde se encuentra el código fuente. Ahora, ejecutamos la orden *make* para compilar nuestro compilador gracias a las directivas del archivo makefile.

Adicionalmente, podemos ejecutar *make* con el parámetro *clean* para limpiar los restos de la compilación, o *run* para que, 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.

Uso del compilador

Una vez disponemos del programa objeto *MC_Compiler*, podemos ejecutarlo para compilar el programa que deseemos escrito en *MiniC* y almacenar el resultado en un fichero de código ensamblador MIPS con *./MC_Compiler [codigo.mc] > [ensamblador.s]*

Ejecución del código generado

Para probar nuestro fichero resultado en ensamblador de *MIPS*, podemos utilizar los simuladores *SPIM* o *MARS*, tanto en sus versiones gráficas como de consola. Si usamos *SPIM*, podemos ejecutar con *./spim -file [ficheroCodigoEnsamblador.s].* Si usamos *MARS*, desde la ventana gráfica podemos abrir el fichero ensamblador y ejecutarlo sin problemas.

Ejemplos de compilación y ejecución

Ejemplo de compilación de un programa con errores

```
void ejemploMiniC() {
    var a, b;
    const X = 3;
    read  a, b;

    do {
        a = a - 1;
    } while(a)

    if(b){
        print "Hola, soy el cuerpo del segundo if\n";
    }else {
        print "Hola, soy el else del segundo if\n";
    }
    print "Hola, estoy en la salida del segundo if\n";

    print X;

    /*
     ESTO ES UN COMENTARIO MULTILÍNEA
    */
    // ESTO ES UN COMENTARIO EN LÍNEA
}
```

```
albertogg023@albertogg023-UX430UAR:~/Escritorio/compilador$ make
bison -d -v analSintSemGenCodigo.y
flex analizadorLexico.l
gcc main.c lex.yy.c analSintSemGenCodigo.tab.c listaSimbolos.c listaCodigo.c -o MC_Compiler
albertogg023@albertogg023-UX430UAR:~/Escritorio/compilador$ ./MC_Compiler test.mc
ERROR SEMANTICO en la línea 2: variable o constante a ya declarada
ERROR SINTACTICO en la linea 3: syntax error, unexpected num, expecting id
ERROR SINTÁCTICO en la linea 3
ERROR LÉXICO en la línea 3: carácter no válido en la cadena ??
ERROR SINTÁCTICO en la linea 3
ERROR SINTÁCTICO en la linea 3
ERROR SINTÁCTICO en la linea 4
ERROR SINTÁCTICO en la linea 4
ERROR SINTÁCTICO en la linea 4
ERROR SINTÁCTICO en la linea 7
ERROR SINTÁCTICO en la linea 9
ERROR SINTÁCTICO en la linea 12
ERROR SINTÁCTICO en la linea 14
ERROR SINTÁCTICO en la linea 16
ERROR SINTÁCTICO en la linea 18
ERROR LÉXICO en la línea 19: cadena sin comillas de cierre
ERROR LÉXICO en la linea 26: comentario sin cerrar
```

Ejemplo de una compilación sin errores y posterior ejecución en MARS

```
albertogg023@albertogg023-UX430UAR:~/Escritorio/compilador$ make
bison -d -v analSintSemGenCodigo.y
flex analizadorLexico.l
gcc main.c lex.yy.c analSintSemGenCodigo.tab.c listaSimbolos.c listaCodigo.c -o MC_Com
piler
albertogg023@albertogg023-UX430UAR:~/Escritorio/compilador$ make run
./MC_Compiler ./test.mc > test.s
albertogg023@albertogg023-UX430UAR:~/Escritorio/compilador$
```

```
void prueba() {
    var a, b;
    read  a, b;

    do {
        print "Hola, soy el cuerpo del bucle do-while: ";
        print a , "\n";
        a = a - 1;
    } while(a)
    print "Hola, soy la salida del bucle do-while\n";

    if(b){
        print "Hola, soy el cuerpo del if-else\n";
    }else {
        print "Hola, soy el else del if-else\n";
    }
    print "Hola, soy en la salida del if-else\n";

    /*
    print "ERROR: Hola, si aparezco problema\n";
    aaaa
    bbbb
    cccc
    dddd
    */
    //print "ERROR: Hola, si aparezco problema\n";
}
```

```
3
0
Hola, soy el cuerpo del bucle do-while: 3
Hola, soy el cuerpo del bucle do-while: 2
Hola, soy el cuerpo del bucle do-while: 1
Hola, soy la salida del bucle do-while
Hola, soy el else del if-else
Hola, soy en la salida del if-else

-- program is finished running --
```

```
##################
 # Seccion de datos
        .data

_a:
        .word 0
_b:
        .word 0
$str1:
        .asciiz "Hola, soy el cuerpo del bucle do-while: "
$str2:
        .asciiz "\n"
$str3:
        .asciiz "Hola, soy la salida del bucle do-while\n"
$str4:
        .asciiz "Hola, soy el cuerpo del if-else\n"
$str5:
        .asciiz "Hola, soy el else del if-else\n"
$str6:
        .asciiz "Hola, soy en la salida del if-else\n"

##################
# Sección de codigo
        .text
        .globl main
main:
li $v0 5
syscall
sw $v0 _a
li $v0 5
syscall
sw $v0 _b
$l1:
la $a0 $str1
li $v0 4
syscall
lw $t0 _a
move $a0 $t0
li $v0 1
syscall
la $a0 $str2
li $v0 4
syscall
lw $t1 _a
li $t2 1
sub $t1 $t1 $t2
sw $t1 _a
lw $t3 _a
bnez $t3 $l1
la $a0 $str3
li $v0 4
syscall
lw $t4 _b
beqz $t4 $l2
la $a0 $str4
li $v0 4
syscall
b $l3
$l2:
la $a0 $str5
li $v0 4
syscall
$l3:
la $a0 $str6
li $v0 4
syscall


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

Las dos primeras líneas de la ejecución en MARS son la lectura por pantalla de las variables a y b

Conclusiones

En líneas generales, la realización de este proyecto ha sido de gran satisfacción para nosotros. Creemos que ha asentado y complementado muy bien los conocimientos de teoría. Además, nos ha parecido que las sesiones de prácticas han estado muy bien planteadas, teniendo en la pestaña *contenidos* todo lo necesario de una forma clara y concisa. Además, el formato de clase invertida nos ha parecido todo un acierto, ya que así, teníamos una atención dedicada por parte de la profesora para resolver nuestras dudas y problemas.

El proceso de compilación, en nuestra opinión, es una de las mayores incógnitas que surgen a los alumnos cuando empiezan a estudiar programación y estructura de computadores. Por ello, el proyecto ayuda con éxito a arrojar luz sobre esta cuestión, que a priori y sobre todo en los comienzos de su estudio, parece tan compleja.

En resumen, estamos muy contentos con la asignatura, la cual nos ha parecido muy teórica e interesante; el proyecto de prácticas y el equipo docente.