

MEMORIA DE PROYECTO DE COMPILADORES - miniC  
Ángel Ruiz Fernandez <a.ruizfernandez@um.es>  
2º - Grupo 2.2 - Primera Convocatoria

TABLE OF CONTENTS

1. Introducción .....	3
2. Explicación de la práctica .....	3
2.1. Funciones principales .....	3
2.2. Estructuras de datos .....	4
2.3. Manual de usuario .....	5
3. Ejemplos de ejecución explicados .....	5
4. Mejoras .....	7
4.1. do-while .....	7
4.2. for .....	7
4.3. Operadores relacionales .....	8
5. Conclusión .....	8

## 1. Introducción

En esta memoria se describe la práctica de compiladores según los puntos definidos en la tarea.

## 2. Explicación de la práctica

### 2.1. Funciones principales

- void yyerror(): funcion de error semantico
- void print\_code(ListaC code): imprime una lista de código
- void print\_symtable(): imprime la tabla de simbolos
- void setup\_program(): inicializa estructuras de datos al inicio del programa
- void symtable\_push(const char \*id): insertar entrada en tabla de simbolos
- void ds\_push\_word(char \*id): insertar un .word a segmento de datos (const y var)
- const char\* ds\_push\_asciiiz(const char \*lstr): insertar cadena a segmento de datos
- void cl\_program(const char \*id, ListaC decls, ListaC statements): genera código común a todos los programas
- ListaC cl\_push\_const\_list(ListaC constl, const char \*id, ListaC vl): genera código de inicialización de constantes
- ListaC cl\_push\_lint(const char \*lint): genera código de carga de literales enteros
- ListaC cl\_push\_id(const char \*id): genera código de carga de simbolos
- ListaC cl\_push\_ominus\_expr(ListaC l): genera código para negación unaria
- ListaC cl\_push\_condop(ListaC cond, ListaC tl, ListaC fl): genera código para el operador condicional
- ListaC cl\_push\_binop(const char \*inst, ListaC ll, ListaC rl): genera código para operacdores binarios (+, -, \*, /)
- ListaC cl\_push\_rel(const char \*inst, ListaC ll, ListaC rl): genera código para operadores relacionales (<, >, <=, >=, ==, !=)
- ListaC cl\_push\_assign(const char \*id, ListaC l): genera código para la asignación
- ListaC cl\_push\_if\_else(ListaC cond, ListaC ifl, ListaC elsel): genera código para si-sino
- ListaC cl\_push\_if(ListaC cond, ListaC ifl): genera código para condicional si

- ListaC cl\_push\_while(ListaC cond, ListaC statement1): genera código para bucle mientras
- ListaC cl\_push\_do\_while(ListaC statement1, ListaC cond): genera código para bucle hacer mientras
- ListaC cl\_push\_for(const char \*id, const char \*lintinit, ListaC cond, ListaC statement1, int sign, const char \*lintstep): genera código para bucle for
- ListaC cl\_push\_print\_expr(ListaC expr1): genera código para impresión de expresiones
- ListaC cl\_push\_print\_str(const char \*lstr): genera código para impresión de cadenas
- ListaC cl\_push\_read(const char \*id): genera código para lecturas

#### Funciones de utilidad

- const char\* alloc\_reg(): función de asignación de registros
- void free\_reg(const char \*reg): libera un registro asignado
- char\* \*\_label(): genera etiquetas a partir del correspondiente contador (los contadores se incrementan manualmente)
- char\* next\_string\_label(): genera la siguiente etiqueta de cadena
- char\* label\_colon(const char \*label): añade ':' al final de una etiqueta

## 2.2. Estructuras de datos

%union define la colección de todos los posibles tipos que puede adoptar un sintagma. char\* para tokens yylval como IDs, literales, etc; y ListaC para el resultado de statements, etc.

```
%union {  
    char *lex;  
    ListaC code;  
}
```

La lista de simbolos, que es usada solo para comprobar errores de no definición en el analisis sintactico.

```
Lista symtable = NULL;
```

La lista de codigo correspondiente al segmento de datos, que se va rellenando conforme se encuentran variables, constantes y cadenas.

```
ListaC dataseg = NULL;
```

Array de registros y nombres: representa el uso de estos para el asignador de registros, y sus nombres segun el indice.

```
int regs[10] = { 0 };
const char reg_strs[][10] = {
    "$t0", "$t1", "$t2", "$t3", "$t4", "$t5", "$t6", "$t7", "$t8", "$t9"
};
```

Contadores de etiquetas por cada tipo de sentencia.

```
int string_counter = 0, cond_counter = 0, if_counter = 0, while_counter = 0,
    dowhile_counter = 0, for_counter = 0;
```

### 2.3. Manual de usuario

minicc acepta dos flags:

- --help: para imprimir el uso
- --debug: para activar la salida de depuración

Y la entrada y la salida se definen con respectivos argumentos posicionales, primero la entrada.

```
usage: minicc [--help|--debug] [input] [output]
--help:      display this
--debug:     enable debugging output
```

Fig 1. Ayuda incluida

### 3. Ejemplos de ejecución explicados

```
# compilar usando stdin y stdout
./minicc < prueba.mc > prueba.S
# compilar usando archivos
./minicc prueba.mc prueba.S
```

Fig 2. Llamada

```
prueba() {
const int a=0, b=0;
var int c;
print ("Inicio del programa\n");
c = 5+2-2;
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");
}
```

Fig 3. Entrada de prueba

```
.data
_a:
.word 0
_b:
.word 0
_c:
.word 0
$str0:
.asciiz "Inicio del programa\n"
$str1:
.asciiz "a"
$str2:
.asciiz "\n"
$str3:
.asciiz "No a y b\n"
$str4:
.asciiz "c = "
$str5:
.asciiz "\n"
$str6:
.asciiz "Final"
$str7:
.asciiz "\n"
.text
.globl main
main:
li $t0, 0
sw $t0, _a
li $t0, 0
sw $t0, _b
li $v0, 4
la $a0, $str0
syscall
li $t0, 5
li $t1, 2
add $t2, $t0, $t1
li $t0, 2
sub $t1, $t2, $t0
sw $t1, _c
lw $t0, _a
beqz $t0, iflelse
li $v0, 4
la $a0, $str1
syscall
li $v0, 4
la $a0, $str2
syscall
j iflend
iflelse:
lw $t1, _b
beqz $t1, if0else
li $v0, 4
la $a0, $str3
syscall
j if0end
if0else:
```

```
while0:
    lw $t2, _c
    beqz $t2, while0end
    li $v0, 4
    la $a0, $str4
    syscall
    lw $t3, _c
    li $v0, 1
    move $a0, $t3
    syscall
    li $v0, 4
    la $a0, $str5
    syscall
    lw $t3, _c
    li $t4, 2
    sub $t5, $t3, $t4
    li $t3, 1
    add $t5, $t5, $t3
    sw $t5, _c
    j while0

while0end:
if0end:
iflend:
    li $v0, 4
    la $a0, $str6
    syscall
    li $v0, 4
    la $a0, $str7
    syscall
# program end
    li $v0, 10
    syscall
```

Fig 4. Código generado

## 4. Mejoras

### 4.1. do-while

El bucle do-while se implementó definiendo el lexema "do", la sintaxis de statement "do statement while (expr);" y generando la etiqueta de loop primero, seguido de el cuerpo, y al final unicamente la expresión de condición y una instrucción de 'bnez'.

### 4.2. for

El bucle for se implementó definiendo el lexema "for" y las sintaxis en statement:

- for (id = lint; expr) statement
- for (id = lint; expr; [-]lint) statement

Las sintaxis admiten el id del iterador, el valor inicial, la expresión de condición (asumiendo uso del mismo iterador) y opcionalmente el paso con o sin signo que sumar al iterador, seguido del cuerpo del bucle.

El código de este bucle se genera concatenando las siguientes listas en este orden:

- etiqueta de inicio de bucle
- lista de expresión condición
- instrucción de branch a la terminación de bucle
- lista de cuerpo de bucle
- instrucciones para cargar, incrementar y guardar iterador
- instrucción de salto al inicio (bucle)
- etiqueta de terminación de bucle

#### 4.3. Operadores relacionales

Definidos los siguientes lexemas:

- <
- >
- <=
- >=
- ==
- !=

Definidas las siguientes sintaxis en expr:

- expr < expr
- expr > expr
- expr <= expr
- expr >= expr
- expr == expr
- expr != expr

Y una función que toma las listas de las dos expresiones y la instrucción de comparación correspondiente al operador, y se genera el código concatenando las dos listas de expresiones y a continuación la instrucción de comparación que toma como operandos los resultados de cada expresión, y retorna la nueva lista cuyo resultado es un nuevo registro con el resultado de la comparación.

#### 5. Conclusión

Esta práctica ha sido muy interesante para descubrir como funciona un compilador basado en GNU flex y bison como lo es GNU gcc, y para entender como se define la gramática de un lenguaje en este.