MEMORIA DE PROYECTO DE COMPILADORES - miniC

Ángel Ruiz Fernandez <a.ruizfernandez@um.es>
2º - Grupo 2.2 - Primera Convocatoria

TABLE OF CONTENTS

1. Explicación de la práctica

1.1. 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_asciiz(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 statementl): genera código para bucle mientras
- ListaC cl_push_do_while(ListaC statementl, ListaC cond): genera código para bucle hacer mientras

- ListaC cl_push_for(const char *id, const char *lintinit, ListaC cond, ListaC statementl, int sign, const char *lintstep): genera código para bucle for

- ListaC cl_push_print_expr(ListaC exprl): 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

1.2. 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;
```

1.3. Manual de usuario

minicc acepta dos flags:

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

2. 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, if1else
        li $v0, 4
        la $a0, $str1
        syscall
        li $v0, 4
        la $a0, $str2
        syscall
        j if1end
    if1else:
        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:
    if1end:
        li $v0, 4
        la $a0, $str6
        syscall
        li $v0, 4
        la $a0, $str7
        syscall
    # program end
        li $v0, 10
        syscall
```

Fig 4. Código generado

3. Mejoras

3.1. 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’.

3.2. 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 codigo 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

3.3. 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 codigo 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.