PROYECTO MINIC
COMPILADORES

Pablo Belchí Corredor()
Daniel Cascales Marcos
(d.cascalesmarcos@um.es)
(pablo.belchic@um.es)

Convocatoria mayo 2025

ÍNDICE

1. Explicación de la práctica……..3
   1.1. Funciones principales…….3
   1.2. Estructuras de datos……...5
   1.3. Manual de usuario……...5
2. Ejemplos de funcionamiento explicados……..6
3. Mejoras……..9
   3.1. Do-while…….9
   3.2. For……..9
   3.3. Operadores relacionales…….9

1. Explicación de la práctica

1.1. Funciones principales

- void yyerror():
Función de error semántico.

- void print_code(ListaC code):
Imprime una lista de código

- void print_symtable():
Imprime la tabla de símbolos.

- void setup_program():
Inicializa estructuras de datos al inicio del programa.

- void symtable_push(const char *id):
Insertar entrada en tabla de símbolos.

- void agregar_variable_datos(char *id):
Insertar un .word al segmento de datos (const y var).

- const char* agregar_cadena_datos(const char *lstr):
Insertar cadena a segmento de datos.

- void generar_programa_base(const char *id, ListaC decls, ListaC statements):
Genera código común a todos los programas.

- ListaC generar_constantes(ListaC constl, const char *id, ListaC vl):
Genera código de inicialización de constantes.

- ListaC generar_literal_entero(const char *lint):
Genera código de carga de literales enteros.

- ListaC generar_carga_identificador(const char *id):
Genera código de carga de símbolos.

- ListaC generar_negacion(ListaC l):
Genera código para negación unaria.

- ListaC generar_operador_condicional(ListaC cond, ListaC tl, ListaC fl):
Genera código para el operador condicional.

- ListaC generar_operacion_binaria(const char *inst, ListaC ll, ListaC rl):
Genera código para operadores binarios (+, -, *, /).

- ListaC generar_operacion_relacional(const char *inst, ListaC ll, ListaC rl):
Genera código para operadores relacionales (<, >, <=, >=, ==, !=).

- ListaC generar_asignacion(const char *id, ListaC l):
Genera código para la asignación.

- ListaC generar_if_else(ListaC cond, ListaC ifl, ListaC elsel):
Genera código para si-sino.

- ListaC generar_if(ListaC cond, ListaC ifl):
Genera código para condicional si.

- ListaC generar_bucle_while(ListaC cond, ListaC statementl):
Genera código para bucle mientras.

- ListaC generar_bucle_do_while(ListaC statementl, ListaC cond):
Genera código para bucle hacer mientras.

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

- ListaC generar_impresion_expresion(ListaC exprl):
Genera código para impresión de expresiones.

- ListaC generar_impresion_cadena(const char *lstr):
Genera código para impresión de cadenas.

- ListaC generar_lectura(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 símbolos, que es usada sólo para comprobar errores de no definición en el análisis sintáctico.

```
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 según el índice.

```
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
```

2. Ejemplos de funcionamiento explicados

Llamada:

Compilación con stdin y stdout
./minicc < prueba.mc > prueba.S
Compilación con archivos
./minicc prueba.mc prueba.S

Entrada de prueba:

```
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");
  }
```

Código generado:

```
      .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
```

3. Mejoras

3.1. do-while

El bucle do-while se construyó añadiendo el lexema "do" y permitiendo la estructura de control con la forma do statement while (expr);. La generación de código comienza colocando una etiqueta al inicio del bucle, después se añade el bloque de instrucciones correspondiente, y finalmente se evalúa la condición junto con una instrucción bnez que permite repetir el ciclo si se cumple.

3.2. for

La implementación del bucle for se realizó introduciendo el lexema "for" y permitiendo dos variantes sintácticas:

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

Estas variantes permiten especificar el identificador del iterador, su valor inicial, una condición de continuación (que se asume relacionada con ese iterador), y opcionalmente un incremento o decremento. Todo ello precede al bloque de instrucciones del bucle.

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

1. Etiqueta de inicio de bucle.
2. Lista de expresión condición.
3. Instrucción de branch a la terminación de bucle.
4. Lista de cuerpo de bucle.
5. Instrucciones para cargar, incrementar y guardar iterador.
6. Instrucción de salto al inicio (bucle).
7. 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

Para generar el código, se emplea una función que toma las listas de instrucciones de ambas expresiones y el operador correspondiente. La función concatena las instrucciones necesarias para evaluar ambas expresiones, y luego añade una instrucción de comparación que utiliza los registros resultantes de cada una. El resultado es una nueva lista de instrucciones, cuyo valor final es un registro con el resultado de la comparación.