COMPILADORES

Documentación MiniC Compiler

Eduardo Blazque Verdejo
eduardo.blazquezv@um.es / 48846313V

Grupo/Subgrupo: 1.1

June 16, 2024

Contents

1 Introducción

El propósito de este documento es detallar el proceso de diseño e implementación de un compilador para un lenguaje tipo C, MiniC. Para el desarrollo de dicho compilador se han utilizado las herramientas de Flex y Bison.

En los siguientes apartados de la memoria se explicarán las etapas que maneja el compilador: análisis léxico, sintáctico, semántico y la detección de errores.

2 Lenguaje MiniC

MiniC es una versión reducida de C, en este caso solo se manejan constantes y variables enteras. Los tipos booleanos se representan mediante enteros: 0 para *false* y 1 para *true*.

No dispone de operaciones relaciones o lógicas y las sentencias de control del flujo de la ejecución son: if, if-else, while y do-while.

2.1 Simbolos terminales

Los enteros (token INTLITERAL), pueden tener un valor desde -$2^3$1 hasta $2^3$1. También existen cadenas de texto (token CADENA).

Los identificadores (token ID) están formados por secuencias de letras y dígitos, empezando siempre por letra y no excediendo los 16 caracteres.

Las palabras reservadas son: var (VAR), const (CONST), if (IF), else (ELSE), while (WHILE), do (DO), print (PRINT) y read (READ).

Por último están los caracteres de separación ; (SEMICOLON) y , (COMMA); de operaciones aritméticas + (PLUS), - (MINUS), * (MULT), y / (DIV); de asignación: = (ASSIGN); y de control de precedencia y bloques: ((LPAREN),) (RPAREN), (LLAVEIZ), (LLAVEDE).

3 Análisis Léxico

El análisis léxico ha sido implementado con la herramienta Flex, en el archivo miniCLex.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. Reconoce-

mos 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.

- La expresión que reconoce un número: entero, siendo entero = digito([0-9]+), y comprueba que no exceda los 4 bytes.
- Para los ID la expresión regular es: (letra)(letra—digito)*, siendo letra definada por [a-zA-Z] y comprobando que no exceda los 32 caracteres.
- El modo pánico se define de la siguiente manera:

```
[\(^\)a-zA-Z0-9/()\(_\);,=+*\-{}"\n\r\t]+.}
```

4 Análisis Sintáctico

El análisis sintáctico ha sido implementado con la herramienta Bison, en el archivo miniC-Sin.y.

Se establecen las precedencias necesarias en los operadores:

```
%left PLUSOP MINUSOP -%left MULTOP DIVOP -%left UMINUS
```

y además se inserta una opción para que el analizador acepte el conflicto de reducción if-else. Respecto al tratamiento de errores utilizamos la función yyerror().

5 Análisis Semántico

El análisis semántico se desarrolla con una estructura de datos Lista. A partir de ella y con las funciones que vienen implementadas, junto con las auxiliares implementadas, podemos realizar el análisis satisfactoriamente.

En primera instancia tenemos que manejar la redefinición de variables, lo que provocaría un error semántico. En segundo lugar, tenemos que asegurar que las constantes no puedan cambiar de valor, de lo contrario, el analizador se encargaría de manejarlo. Esto hay que tenerlo en cuenta no solo en la redefinición de por si, sino además en los bucles WHILE y FOR, donde se produciría esta redefinición. En cuanto a la lectura de identificadores, hay que comprobar que esten dentro de la tabla de símbolos.

6 Generación de Código

Para esta parte nos basamos en las estructuras de datos proporcionadas por los ficheros listaCodigo.h y listaCodigo.c, los cuales nos sirven para crear una lista con la que generar el código ensamblador. Respecto a la generación de código, en cada regla gramatical se introduce su igual en código ensamblador, esto se guarda en en la estructura de datos Operacion que posteriormente es introducida en la lista de código.

En esta parte también han sido necesarias la implementación de distintas funciones auxiliares para completar las ya dadas en listaCodigo. Las más destacables:

- getReg(): obtiene un registro libre.
- liberaReg(): libera un registro.
- printTablaS(): imprime el .data de ensamblador.
- printaCod(): imprime el .text de ensamblador.

7 Mejoras

Las mejoras implementadas en este proyecto son:

1. Tratamiento de errores sintácticos (mediante yyerror()).
2. Implementación de sentencia DO-WHILE.

Para la implementación de la sentencia DO-WHILE es necesario cambiar el orden del blucle while para que haga el salto al final de el cuerpo del do, hay que hacer que sea el *bnez* el que se encargue de realizar el salto en lugar de hacer un *branch* para realizar otra iteración.

8 Manual de Uso

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** para generar el programa objeto miniC.

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

```
./miniC prueba.mc > salida.s
```

Además que con el comando **make limpia** elimina del directorio los archivos generados por la compilación.

9 Pruebas

Para comprobar que el compilador funciona como esperamos vamos a utilizar el siguiente código como fichero de prueba entrada:

```
prueba() {
// comentario de una linea
/* comentario de varias lineas
  hola
  como
  estas
  */
const  a=0, b=0;
var d = 10;
var c=5+2-2;
print ("Inicio del programa\n");
if (a)  print ("a","\n");
  else if (b) print ("No a y b\n");
       else while (c)
            {
                  print ("c = ",c,"\n");
                  c = c-1;
            }
    do{
        print ("Final del programa\n");
    } while(d);
}
```

Lo que generará la siguiente salida.s:

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

$str-1:
```

```
.asciiz b
$str2:
.asciiz "Inicio del programa\n"
$str3:
.asciiz "a"
$str4:
.asciiz "\n"
$str5:
.asciiz "No a y b\n"
$str6:
.asciiz "c = "
$str7:
.asciiz "\n"
$str8:
.asciiz "Final del programa\n"
_a:
.word 0
_d:
.word 0
_c:
.word 0
###################
# Seccion de codigo
.text
.globl main
main:
li $t0,0
sw $t0,_a
li $t0,0
sw $t0,_b
li $t0,10
sw $t0,_d
li $t0,5
li $t1,2
add $t2,$t0,$t1
li $t0,2
sub $t1,$t2,$t0
sw $t1,_c
la $a0,$str2
li $v0,4
syscall
lw $t0,_a
beqz $t0,$l5
la $a0,$str3
```

```
li $v0,4
syscall
la $a0,$str4
li $v0,4
syscall
b $l6
$l5:
lw $t1,_b
beqz $t1,$l3
la $a0,$str5
li $v0,4
syscall
b $l4
$l3:
$l1:
lw $t2,_c
beqz $t2,$l2
la $a0,$str6
li $v0,4
syscall
lw $t3,_c
move $a0,$t3
li $v0,1
syscall
la $a0,$str7
li $v0,4
syscall
lw $t3,_c
li $t4,1
sub $t5,$t3,$t4
sw $t5,_c
b $l1
$l2:
$l4:
$l6:
$l7:
la $a0,$str8
li $v0,4
syscall
lw $t0,_d
addi $t1,$t1,1
blt $t1,$t0,$l7
###############
# Fin
```

```
li $v0, 10
syscall
```

10 Conclusiones

La asignatura de Compiladores ofrece una visión fundamental sobre la construcción de traductores de código para lenguajes de programación. A través de esta materia, se exploran las diversas fases del proceso de traducción, la estructura de los programas y las técnicas esenciales para resolver los desafíos que surgen en este campo.

Considero que este proyecto práctico tiene un propósito crucial: permitirnos, como estudiantes, adquirir un conocimiento útil y práctico, facilitando así la comprensión y aplicación de los conceptos teóricos aprendidos tanto en esta asignatura como en otras relacionadas.

Además, los contenidos de la asignatura de Compiladores son de gran utilidad para el desarrollo profesional de un ingeniero informático, ya quer debe entender cómo se construyen los lenguajes de programación y cómo se ensamblan para ser ejecutados en una máquina. Esta comprensión es esencial para el diseño y la implementación efectiva de futuros programas.