MiniC

Compiladores
Univesidad de Murcia

Grupo 1.1

Sánchez Martínez, Hugo
hugo.s.m@um.es

Hernández Galindo, Pablo
p.hernandezgalindo@um.es

Mayo, 2024

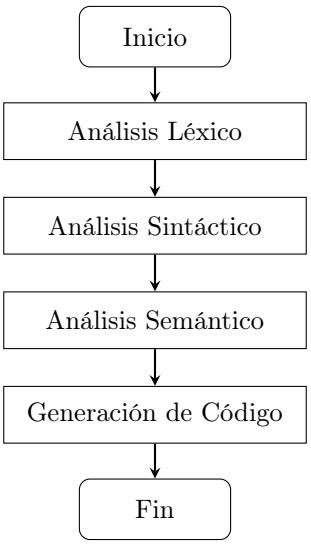
Índice

1. Introducción

MiniC es un compilador que traduce textos en lenguaje fuente miniC a código ensamblador MIPS. La descripción de dicho lenguaje se encuentra en el documento de especificación de la práctica. La implementación del compilador se divide en varias fases:

- Análisis léxico con flex
- Análisis semántico y sintáctico con Bison
- Generación de código

En este documento no se incluye todo el código fuente del proyecto, pues debido a su longitud dificultaría la lectura. No obstante, todos los ficheros usados para el proyecto se encuentran en el directorio principal de éste.



2. Diseño del compilador

2.1. Analizador léxico

La herramienta **flex** permite reconocer los tokens de la gramática de miniC mediante la definición expresiones regulares, así como detectar posibles errores en el código al entrar en modo pánico. El objetivo de este analizador crear un fichero tipo lex en C que reconozca la **secuencia de tokens** de un programa pasado por entrada, elimine los comentarios (son insignificantes para el compilador) e informe de los posibles errores léxicos.

La implementación del analizador léxico se realiza en el archivo `lexico.l`. A continuación se muestran algunas de las expresiones regulares más destacables, junto con las macros de ER usadas:

```
D       [0-9]
L       [a-zA-Z_]
E       {D}+
ERR     [^a-zA-Z_0-9+\-*\=; ()\t\r\n,{}]
```

- Comentarios: se deben detectar tanto comentarios simples como comentarios multilínea. Para los primeros, se puede identificar con una simple ER: `"//"(.*)\n`, que busca cualquier coincidencia de caracteres tales que comiencen por `//` y terminen en una nueva línea. Para los multilínea, se ha usado la ER proporcionada en los recursos `"/*"([^*]|[*]+[^*/])*[*]*"*/"`.
- Nombres de variables: usamos `({L}|_)({L}|{D}|_)*`. Se aceptan nombres que comiencen con una letra o un guion bajo, seguido de cero o más letras, dígitos o guiones bajos adicionales.
- Espacios en blanco: `[ \n\t\r]+`.
- Error léxico (modo pánico): `{ERR}+`.
- Cadenas de texto: `\"([^\"\n\r]|\\'')+\"`.

2.2. Analizadores sintáctico y semántico y generación código en MIPS

2.2.1. Analizador Sintáctico

Los objetivos del analizador sintáctico son los siguientes:

- Verificar si la secuencia de tokens proporcionada por el analizador léxico puede ser generada por la gramática del lenguaje fuente.
- Informar sobre errores sintácticos y recuperarse de ellos, ya que estos errores son comunes, para poder continuar procesando el resto de la entrada.
- Producir la salida deseada, que suele ser el árbol sintáctico. Además, puede realizar otras tareas, como:
  - Recopilar información sobre los tokens en la tabla de símbolos.
  - Realizar análisis semántico, como la verificación de tipos.
  - Generar código intermedio.

2.2.2. Gramática de miniC

Para poder asegurar los objetivos del analizador sintáctico, el compilador miniC se basa en una gramática libre de contexto descrita en Bison:

```
program → id ( ) { declarations statement_list }
declarations → declarations var identifier_list ;
             | declarations const identifier_list ;
             | lambda
identifier_list → identifier
             | identifier_list , identifier
identifier → id
             | id = expression
statement_list → statement_list statement
             | lambda
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
```

A dicha gramática se le han añadido dos reglas más:

```
| FOR PARI ENTE PARD statement
| DO statement WHILE PARI expression PARD PYCO
```

Por supuesto, se ha añadido una función capaz de detectar los errores sintácticos, y una función que me permita poder hacer cada comprobación acerca de los errores antes de generar código.

```
void yyerror(const char *msg){
    errores_sintacticos++;
    printf("Error en la linea %d: %s\n", yylineno, msg);
}
int analisis-ok(){
    return (errores_lexicos + errores_semanticos + errores_sintacticos) == 0;
}
```

Además, hay que tener en cuenta que la GLC original es ambigua respecto a la operaciones aritméticas debido a la no utlización de las reglas de prioridad, con lo que para poder arreglarla se deben hacer las siguientes declaraciones:

```
%left ADD SUB
%left MUL DIV
%precedence UMINUS
```

Por último, hay que mencionar que la gramática debe tener esta expresión para el correcto funcionamiento del `if-else` del programa, ya que con esta expresión avisamos al programa que va a haber un warning:

```
%expect 1
```

2.2.3. Analizador semántico

En el analizador semántico se ha tratado de modificar todos los identificadores posibles:

- La gestión de identificadores es esencial para el análisis semántico del compilador. La lista de símbolos proporcionada en los archivos listaSimbolos.c y listaSimbolos.h se utiliza para almacenar identificadores que se encuentran en el código fuente.
- Las principales comprobaciones relacionadas con el análisis semántico son:
  - **Declaración y redeclaración de identificador:** Si se encuentra un identificador en una regla de asignación (asig: id y asig: id = expresión), el contador se aumenta.
  - **Buscar símbolos.**
    - Si no se encuentra el identificador, se considera una declaración válida.
    - Si ya está en la lista, la redeclaración registrará un error semántico.
  - **Uso de identificadores no declarados**: Reglas de declaración: La expresión id = comprueba la presencia de un identificador en la lista de símbolos.
    - Si no se encuentra, se registra un error semántico por uso de identificador no declarado.
  - **Asignación a un identificador constante**: En la regla anterior, si se encuentra un identificador en la lista de símbolos pero está marcado como CONSTANTE, no se puede realizar ninguna asignación a un identificador constante, por lo que se registrará un error semántico.
  - **Leer identificador:** Reglas readlist: id y readlist: readlist, id verifica si el identificador está presente en la lista de símbolos.
    - Si no se encuentra, se registra un error semántico al intentar leer un identificador no declarado.
    - Además, si el identificador es de tipo CONSTANTE, se registrará otro error porque no se puede asignar un valor después de leer el identificador constante.
  - **Uso de identificadores en expresiones**: La regla de ID "Expresión: " verifica si un identificador está presente en la lista de símbolos.
    - Si no se encuentra, se registrará un error semántico.

2.2.4. Generación de código

En esta sección, se detallan los métodos y funcionalidades agregados para la generación de código en el compilador. Se emplean los archivos `listaCodigo.c` y `listaCodigo.h`, proporcionados por los profesores, para trabajar con listas de código sin necesidad de implementarlas.

Previo a iniciar la generación de código en cada regla, se verifica la validez del análisis mediante `análisis_ok()`, evitando así la generación de código innecesario y posibles comportamientos indeseados.

Para manejar las etiquetas en el código, se inserta una operación `etiq` con el valor de la etiqueta cuando es necesario. Al imprimir la lista de código, se identifica si la operación es una etiqueta y se imprime con el formato correspondiente. Además, se implementaron varios métodos para facilitar la generación de código:

- `imprimirLS()`: imprime la tabla de símbolos del análisis semántico. Después de generar el código en la lista, se utiliza para imprimir la sección de datos con el formato correcto. Para almacenar el valor de una cadena que se imprimirá con `print`, se agrega un símbolo a la lista de símbolos con el nombre de la cadena, tipo CADENA, y valor el número de cadena correspondiente.
- `imprimirLC()`: imprime la lista de código. Se recorre la lista de código e imprime las operaciones con el formato adecuado. Si la operación es "etiq", se imprime la etiqueta con el formato correspondiente.
- `obtenerReg()`: retorna el primer registro temporal disponible de los 10 disponibles.
- `liberarReg()`: libera el registro pasado por parámetro, permitiendo su reutilización en la generación de código.
- `obtenerEtiqueta()`: genera etiquetas de la forma `$lx`, donde `x` es el número de etiquetas utilizado. Se utiliza un contador para generar nombres de etiquetas de manera correcta.

2.3. Mejoras implementadas

2.3.1. Implementación de la sentencia `do-while`

Para la construcción de esta mejora, se realiza lo siguiente:

1. Se verifica si el análisis es válido, para crear una nueva lista de código.
2. Se obtiene una etiqueta única para el inicio del bucle y se inserta la operación de etiqueta con la etiqueta única.
3. Se concatena el código generado por el `do-while` y se libera la lista de codigo generada por statement.
4. Se concatena la expresión de control del `do-while`, para insertar la operación de salto condicional hacia la etiqueta obtenida
5. Por último, se libera el registro temporal usado.

```
| DO statement WHILE PARI expression PARD PYCO {

            if (analisis_ok()) {

                $$ = creaLC();
                char * etiqueta1 = obtenerEtiqueta();

                Operacion o1;
                o1.op = "etiq";
                o1.res = etiqueta1;
                o1.arg1 = NULL;
                o1.arg2 = NULL;
                insertaLC($$, finalLC($$), o1);

                concatenaLC($$, $2);
                liberaLC($2);
                concatenaLC($$, $5);

                Operacion o3;
                o3.op = "bnez";
                o3.res = recuperaResLC($5);
                o3.arg1 = etiqueta1;
                o3.arg2 = NULL;
                insertaLC($$, finalLC($$), o3);

                liberaLC($5);
                liberarReg(o3.res);

            }

}
```

Tanto este código como el siguiente se pueden visualizar en el fichero `sintactico.y`.

2.3.2. Implementación de la sentencia `for`

1. Se vuelve a verificar si el análisis es válido para crear una nueva lista de código.
2. A continuación, se crea un registro temporal y se inicializa el índice con un 0, donde se obtienen etiquetas únicas para el inicio y fin del bucle.
3. Después, se usa `bge` para hacer la comparación de salto y se concatena el código por la regla statement del `for`.
4. Por último, se realiza el incremento del índice, se inserta una operación de salto incondicional y se añade una etiqueta para terminar liberando el registro temporal y la lista de código generada.

```
| FOR PARI ENTE PARD statement {

            if (analisis_ok()) {

                $$ = creaLC();
                char * indice = obtenerReg();
                Operacion o;
                o.op = "li";
                o.res = indice;
                o.arg1 = "0";
                o.arg2 = NULL;
                insertaLC($$, finalLC($$), o);
                * etiqueta1 = obtenerEtiqueta();
                char * etiqueta2 = obtenerEtiqueta();
                Operacion o1;
                o1.op = "etiq";
                o1.res = etiqueta1;
                o1.arg1 = NULL;
                o1.arg2 = NULL;
                insertaLC($$, finalLC($$), o1);
                Operacion o2;
                o2.op = "bge";
                o2.res = indice;
                o2.arg1 = $3;
                o2.arg2 = etiqueta2;
                insertaLC($$, finalLC($$), o2);
                concatenaLC($$, $5);
                Operacion o6;
                o6.op = "addi";
                o6.res = indice;
                o6.arg1 = indice;
                o6.arg2 = "1";
                insertaLC($$, finalLC($$), o6);
                Operacion o4;
                o4.op = "b";
                o4.res = etiqueta1;
                o4.arg1 = NULL;
                o4.arg2 = NULL;
                insertaLC($$, finalLC($$), o4);
                Operacion o5;
                o5.op = "etiq";
                o5.res = etiqueta2;
                o5.arg1 = NULL;
                o5.arg2 = NULL;
                insertaLC($$, finalLC($$), o5);
                liberaLC($5);
                liberarReg(indice);

        }
}
```

3. Manual de usuario

3.1. Compilación y ejecución de MiniC

Para poder compilar bien el proyecto, se usa este Makefile:

```
miniC : lex.yy.c sintactico.tab.c listaCodigo.c listaSimbolos.c
gcc main.c lex.yy.c sintactico.tab.c listaSimbolos.c listaCodigo.c -ll -o miniC

lex.yy.c : sintactico.tab.h lexico.l
 flex lexico.l

sintactico.tab.h sintactico.tab.c: sintactico.y
 bison -d sintactico.y

clean :
 rm -f sintactico.tab.h sintactico.tab.c lex.yy.c miniC

run: miniC prueba.mc
 ./miniC prueba.mc > miniC.s
```

Usando make clean se haría una limpieza de la salida generada anteriormente, y con el make run, se ejecutaría el programa usando prueba.mc como entrada del ejecutable miniC, guardando el resultado en miniC.s que tendría este formato:

```
.data
_a:
          .word 0
_b:
          .word 0
_c:
          .word 0
_d:
          .word 0
$str1:
       .asciiz "Inicio del programa\n"
$str2:
       .asciiz "a"
$str3:
       .asciiz "\n"
$str4:
       .asciiz "No a y b\n"
$str5:
       .asciiz "c = "
$str6:
       .asciiz "\n"
$str7:
       .asciiz "DO/while"
$str8:
       .asciiz "\n"
$str9:
       .asciiz "for"
$str10:
       .asciiz "\n"
$str11:
       .asciiz "Final"
$str12:
       .asciiz "\n"
.text
.globl main
main:
```

```
        li $t0,0
        sw $t0,_a
        li $t0,0
        sw $t0,_b
        li $t0,5
        li $t1,2
        add $t0,$t0,$t1
        li $t1,2
        sub $t0,$t0,$t1
        sw $t0,_c
        li $t0,3
        sw $t0,_d
        la $a0,$str1
        li $v0,4
        syscall
        lw $t0,_a
        beqz $t0,$l5
        la $a0,$str2
        li $v0,4
        syscall
        la $a0,$str3
        li $v0,4
        syscall
        b $l6
$l5:
        lw $t1,_b
        beqz $t1,$l3
        la $a0,$str4
        li $v0,4
        syscall
        b $l4
$l3:
$l1:
        lw $t2,_c
        beqz $t2,$l2
        la $a0,$str5
        li $v0,4
        syscall
        lw $t3,_c
        move $a0,$t3
        li $v0,1
        syscall
        la $a0,$str6
        li $v0,4
        syscall
        lw $t3,_c
        li $t4,2
        sub $t3,$t3,$t4
        li $t4,1
        add $t3,$t3,$t4
        sw $t3,_c
        b $l1
$l2:
$l4:
$l6:
$l7:
        la $a0,$str7
        li $v0,4
        syscall
        la $a0,$str8
        li $v0,4
```

```
        syscall
        lw $t0,_d
        li $t1,1
        sub $t0,$t0,$t1
        sw $t0,_d
        lw $t0,_d
        bnez $t0,$l7
        li $t0,0
$l8:
        bge $t0,3,$l9
        la $a0,$str9
        li $v0,4
        syscall
        la $a0,$str10
        li $v0,4
        syscall
        addi $t0,$t0,1
        b $l8
$l9:
        la $a0,$str11
        li $v0,4
        syscall
        la $a0,$str12
        li $v0,4
        syscall
```

3.2. Ejecución miniC.s

El código generado llamado miniC.s, se ejecuta en MIPS, generando este resultado:

```
Inicio del programa
c = 5
c = 4
c = 3
c = 2
c = 1
DO/while
DO/while
DO/while
for
for
for
Final

-- program is finished running (dropped off bottom) --
```

4. Entrada de prueba

Este es el codigo en C++ que genera el codigo MIPS anterior.

Entrada de prueba escrita en MiniC

```
prueba() {
const  a=0, b=0;
var c=5+2-2;
var d=3;
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-2+1;
            }

do{
   print ("DO/while","\n");
   d=d-1;
}while(d);


for(3){
  print ("for","\n");
}
  print ("Final","\n");
}
```

5. Conclusión

Este proyecto ha sido una experiencia reconfortante debido a que gracias a él, hemos sido capaces de comprender cómo funciona un compilador de forma interactiva y cómo se llega a un lenguaje destino, en este caso MIPS, desde un lenguaje fuente.

Hemos visto las distintas fases que abarca un proceso de compilación, desde el análisis léxico del programa inicial hasta el análisis semántico y la generación de código final pasando por el análisis sintáctico, utilizando herramientos como Flex y Bison.

Por último, agradecer a los profesores de la asignatura Manuel y Eduardo por la ayuda brindada durante el proceso, y por hacer que la experiencia de entender cómo funciona un compilador por dentro fuera tan satisfactoria.