COMPILADOR MINIC

Miguel Salinas Egea, Juan Botía López

miguel.salinase@um.es

j.botialopez@um.es

Índice

Contenido

1.Introducción

A lo largo de esta práctica vamos a realizar un compilador que nos permitirá reconocer un lenguaje "simplificado" de c, llamado miniC. La especificación de este proyecto se ha dividido en varias fases a la hora de generar el compilador. Distinguiremos: Análisis léxico, análisis sintáctico, análisis semántico, generación de código ensamblador.

Como herramientas externas para realizar las distintas fases del compilador hemos utilizado:

- **Flex:** Utilizado para realizar el análisis léxico.
- **Bison:** Utilizado para realizar el análisis sintáctico.

La especificación del lenguaje miniC es la siguiente:

- Sólo manejaremos constantes y variables enteras.
- El lenguaje permite operadores aritméticos de valores enteros.
- No hay operadores relacionales ni lógicos en la versión básica del lenguaje.
- En consecuencia, las expresiones aritméticas de enteros se podrán utilizar como condiciones.
- lógicas de forma similar a C (0 es falso y todos los demás enteros son verdadero).
- No hay definición de funciones salvo la del programa principal.

La gramática definida va a tener la siguiente estructura y restricciones:

- **Constantes**
    - **Enteros:** sólo serán válidos los enteros con signo que se puedan representar con el tipo word (4 bytes) en el ensamblador de MIPS, es decir, con valores entre -2^{31} y $2^{31} - 1$.
    - **Cadenas:** Secuencia caracteres delimitados por comillas dobles. Puede contener dobles comillas siempre que estén precedidas de contrabarra.
- **Identificadores:** Secuencia de letras, dígitos y símbolos de subrayado, no comenzando por dígito y no excediendo los 16 caracteres.
- **Palabras reservadas:** void, var, const, if, else, while, print y read.
- **Caracteres especiales:** ";", ",", "+", "-", "*", "/", "=", "(", ")", "{", "}".

2. Estructura de miniC

Nuestro compilador consiste en 4 ficheros principales:

main.c: Este fichero es el encargado de inicializar las variables necesarias, aparte de iniciar el analizador léxico con la función yylex(), y el analizador sintáctico también se inicializa con la función yyparse().

Makefile: Este fichero se encarga de la generación de los ficheros encargados de la compilación. Este fichero usa el compilador de C (gcc) y aparte, debe crear los ficheros en orden. Por último, elimina los ficheros no necesarios después de la compilación.

Lexico.l: El archivo con el contenido del analizador léxico. Le devuelve al analizador sintáctico los tokens que encuentra.

sintaxis.y: El archivo con el contenido del analizador sintáctico y semántico. Define la estructura del lenguaje y se encarga de ir reduciendo las expresiones con las reglas definidas.

3. Análisis léxico

El analizador léxico es la primera parte que se realiza del compilador. Se encarga de reconocer los tokens para luego poder enviarlos al analizador sintáctico. Se realiza en el archivo lexico2.l. En la implementación hemos tenido las siguientes fases.

En un primer momento hemos añadido los tokens necesarios para que se pueda reconocer completamente la gramática. Esto lo hemos hecho a partir de las expresiones regulares que reconocen los tokens.

Vamos a tener como principales expresiones regulares.

Las principales son:

Identificadores: {L}({L}|{D}|_)*

Números: {D}+

Cadenas de caracteres: \"([^\"\n]|\\\")*\"

Comentario sin cerrar: \"([^\"\n]|\\\")*

Errores en modo pánico: [^ \n\r\ta-zA-Z$_0-9+*/\-=;(){},"]+

Comentarios multilínea:

<comentario>(.|\n) ;

<comentario>"*/" { BEGIN(0);}

<comentario><<EOF>> { err_lex++; printf("Error lexico: comentario sin cerrar desde linea %d\n", inicio_comentario); return 0; }

Los errores léxicos que encuentre nuestro compilador en el código se cuentan aumentando la variable **numErrorLex** en 1. Hay varios tipos de errores que puede reconocer:

Las cadenas de caracteres no pueden ser mayor que 16 ya que se trataría de una cadena no válida. Usando la variable **yyleng** podemos saber si supera el tamaño o no.

Los números enteros tienen un límite de tamaño ya que no puede ser mayor que 2^31. En el caso de que el numero sea mayor, no se tratará como un entero.

Parecido a como implementamos el cierre de los comentarios, comprobamos que si en algún momento el compilador recibe el carácter “, también aseguramos que tengamos que recibir el carácter “ que falta para cerrar las dobles comillas, de hecho en uno de nuestros ficheros de prueba comprobamos esta misma situación.

También se ha implementado un modo pánico con una expresión regular que lo acepta todo. Así que, cuando se active el modo pánico, se reconocerá la cadena sin realizar ninguna acción hasta que se vuelve a encontrar con un token válido.

Por último, tenemos los comentarios multilínea los cuáles hemos implementado usando una condición de contexto. Señalizaremos que un comentario se trata de una condición de

contexto mediante %x comentario y entrará en ella en el caso de encontrar "/*". Una vez dentro de la condición de contexto, se aceptará todo lo que entre hasta que se reconozcan los caracteres "*/" que indicarán el final del comentario multilínea. En el caso de no encontrarse con los caracteres mencionados, habrá un fallo léxico que señalizará la falta del cierre del comentario multilínea.

4. Analisis sintáctico

El análisis sintáctico se realiza en el archivo sintaxis.y junto al análisis semántico.

Se utiliza una variable de tipo entero para controlar el número de errores que surgen en el análisis sintáctico llamada errSintac. Dicha variable aumenta cada vez que se encuentra un nuevo error en la sintaxis, aparte de mostrar el error y la línea con la variable yylineno.

A la hora de reconocer la gramática se utiliza implementando BISON utilizando una gramática proporcionada por el profesorado en notación BNF.

Para tratar la asociatividad y la precedencia, hemos indicado la preferencia y los tipos de asociatividad en los terminales:

```
%left "+" "-"

%left "*" "/"
```

Con esto indicamos que los siguientes terminales se traten con asociatividad a la izquierda.

Para tratar el conflicto desplaza-reduce del caso if-else, utilizamos el comando:

```
%expect 1
```

Esto lo hacemos ya que, por defecto BISON desplaza en vez de reducir. Sin embargo, muestra un error en el fichero de salida. Este comando anula el error para no mostrarlo.

Hemos añadido una derivación en statement para tratar los errores sintácticos de la gramática y permitir que el analizador sintáctico no quede bloqueado. En las reglas de producción de statement, emplean un símbolo especial a la derecha que representa el error, y que es reconocido como un token.

5. Análisis semántico

En cuanto al análisis semántico, realizamos varias comprobaciones para asegurar la estructura de nuestra gramática. Para esto nos han facilitado una lista enlazada que representa una lista de símbolos.

Las comprobaciones más importantes que se hacen son:

- Comprobar que el identificador **está declarado** en la tabla de símbolos ya que si trabajamos con una variable o una constante debe estar declarada.
- Comprobar que el identificador **no está declarado** en la tabla de símbolos ya que, si queremos asignar un valor a una variable, debemos confirmar que esa variable no tenga antes otro valor asignado.
- Comprobar que el identificador **está declarado en la tabla de símbolos y que no sea una constante**.
    - Esto es útil para la regla de producción **ID = expresión** ya que, si queremos realizar una asignación, debemos asegurarnos de que no sea una constante ya que no se podría modificar su valor y que esté declarada en la tabla de símbolos.
    - También es útil para la regla **read_list** ya que si leemos variables por entrada de usuario necesitamos que estén declaradas y que no sean constantes.

Hemos implementado varias funciones auxiliares:

- **compruebaTabla(char * simbolo)**: Esta función es utilizad para comprobar que el token, que es pasado por parámetro, está en la tabla de símbolos. Si recorremos la tabla y nos encontramos al final, sabremos que el token no está en la tabla. En caso contrario, el token si está en la tabla de símbolos.
- **insertaCadena(char * simbolo)**: Esta función también es dedica a insertar el token pasado por parámetro, solo que esta inserta la cadena a la tabla de código.
- **constanteComprueba(char * simbolo)**: Esta función se utiliza para comprobar si un token introducido por parámetro es constante o es una variable. Se recorre la tabla de símbolos y si se encuentra el token y si es de tipo constante, se devuelve true(1). En caso contrario, se devuelve false (0).

6. Generación de código

Para la generación de código se necesita crear una tabla de código. Cada símbolo no-terminal tiene asociada una lista de código. Con las reglas de producción unimos las listas de código a las reglas no-terminales para después concatenarlas con las expresiones que haya a la izquierda de la regla de producción.

Usamos algunas funciones auxiliares para ayudarnos con la elaboración:

- **inicializaRegs():** Se inicializa el array a 0. Este array se utilizará para saber qué registros están libres.
- **getRegistroLibre():** Se recorre el array de registros. Si encuentra un registro libre, lo ocupa y devuelve un puntero con la posición.
- **liberarReg():** Pone a 0 la posición de array que metamos por parámetro simulando la liberación del registro. Así se podrá utilizar en un futuro.
- **concatena(char * cad1, char * cad2):** Concatena las 2 cadenas de caracteres que se pasan por parámetro y las almacena en una memoria dinámica devolviendo un puntero referido a la nueva cadena.
- **nuevaEtiqueta():** Devuelve el puntero de una cadena que será usada para crear las etiquetas de los saltos. Limitando el número de caracteres de la etiqueta a 16.
- **imprimirListaSim():** Esta función se encarga de imprimir una lista de símbolos en el archivo "codigoEnsamblador.s". Recorre la lista de símbolos y escribe en el archivo las cadenas y los símbolos no cadenas, generando el formato adecuado para cada uno.
- **imprimirCodigo(ListaC codigo):** Función que se encarga de imprimir la sección de código en el fichero “codigoEnsamblador.s”. En esta parte se imprime todo lo relacionado con la lista de código, operaciones, saltos,etc.

7. Manual de uso del usuario

Hemos creado un archivo makefile para facilitar la compilación de los ficheros deseados.

Sólo haría falta ejecutar:

```
make
```

Para ejecutar un fichero tenemos primero la posibilidad de ejecutarlo con el comando:

```
make run
```

Si desearas cambiar el fichero a ejecutar, simplemente se tendría que modificar el fichero makefile para que al volver a ejecutar el comando de nuevo, se ejecute el fichero deseado. El resultado de la compilación se encuentra en el fichero que indica por pantalla. codigoEnsamblador.s

También existe la posibilidad de utilizar el comando:

```
./miniC fichero.mc
```

Sería una forma más directa de ejecutar y compilar el fichero deseado. Además si quisieras imprimir el resultado en un fichero de tu gusto, podrías utilizar la opción de Linux que permite enviar a un fichero de tu creación la salida generada.

```
> salidaMIPS.s
```

8. Ejemplo de compilación y ejecución correcta

Para comprobar el funcionamiento de nuestro programa, hemos utilizado el fichero que se puso a nuestra disposición con el nombre prueba.mc. Dentro del .zip se encuentran también dos ficheros de prueba más que utilizamos para comprobar que el compilador detectaba los errores de forma correcta.

prueba.mc:

```
void prueba() {
const  a=0, b=0;
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-2+1;
            }
  print "Final","\n";
}
```

codigoEnsamblador.s:

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

$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 "Final"
$str8:
        .asciiz "\n"
_a:
        .word 0
_b:
        .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 $t0,5
        li $t1,2
        add $t0,$t0,$t1
        li $t1,2
        sub $t0,$t0,$t1
        sw $t0,_c
        li $v0,4
        la $a0,$str1
        syscall
        lw $t0,_a
        beqz $t0,$l5
        li $v0,4
        la $a0,$str2
        syscall
        li $v0,4
        la $a0,$str3
        syscall
        b $l6
$l5:
        lw $t1,_b
        beqz $t1,$l3
        li $v0,4
        la $a0,$str4
        syscall
        b $l4
$l3:
$l1:
        lw $t2,_c
        beqz $t2,$l2
        li $v0,4
        la $a0,$str5
        syscall
        lw $t3,_c
        li $v0,1
        move $a0,$t3
        syscall
        li $v0,4
        la $a0,$str6
        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:
        li $v0,4
```

```
        la $a0,$str7
        syscall
        li $v0,4
        la $a0,$str8
        syscall

############################
# Fin
        li $v0, 10
        syscall
```

Ejecución en Mars:

```
Inicio del programa
c = 5
c = 4
c = 3
c = 2
c = 1
Final

-- program is finished running --
```

Como podemos comprobar el programa realiza la ejecución correcta.

Respecto al fichero de prueba1.mc que contiene errores léxicos y semánticos, la salida es la siguiente:

```
alumno@FIUM:~/Escritorio/COMPILADORES/pruebaCambioNombre$ ./miniC prueba1.mc
Fallo en linea 5: syntax error, unexpected id, expecting ; or ","
Fallo (l◆xico): lexemas no v◆lidos en la l◆nea 5: "\"
Fallo (l◆xico): lexemas no v◆lidos en la l◆nea 6: "\"
Fallo (l◆xico): lexemas no v◆lidos en la l◆nea 9: "\"
Fallo (l◆xico): lexemas no v◆lidos en la l◆nea 12: "\"
Fallo (l◆xico):  cadena sin cerrar en la l◆nea 12
Errores durante compilación:
-->Errores léxicos: 5
-->Errores sintácticos: 1
-->Errores semánticos: 0
```

Código correspondiente:

```
void prueba() {
const  a=0, b=0;
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-2+1;
             }
  print "Final","\n";
}
```

Prueba2.mc:

```
alumno@FIUM:~/Escritorio/COMPILADORES/pruebaCambioNombre$ ./miniC prueba2.mc
Fallo en linea 2: syntax error, unexpected id, expecting ; or ","
Errores durante compilación:
-->Errores léxicos: 0
-->Errores sintácticos: 1
-->Errores semánticos: 0
```

```
void prueba() {
const  a=0 b=0;
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-2+1;
            }
  print "Final","\n";
}
```

Prueba3.mc:

```
alumno@FIUM:~/Escritorio/COMPILADORES/pruebaCambioNombre$ ./miniC prueba3.mc
Error semantico en linea 7: c no declarada
Error semantico en linea 9: c no declarada
Error semantico en linea 10: c no declarada
Error semantico en linea 10: c no esta declarada
Errores durante compilación:
-->Errores léxicos: 0
-->Errores sintácticos: 0
-->Errores semánticos: 4
```

```
void prueba() {
const  a=0, b=0;

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;
            }
  print "Final","\n";
}
```