Compilador MiniC
Prácticas Compiladores Junio 2023

Componentes del grupo:
-Christian Matas Conesa (grupo 2.3): christian.m.c@um.es
-Julio Betanzos Legaz (grupo 3): julio.b.l@um.es

Convocatoria: Junio 2023

Análisis léxico

Nuestro analizador léxico es lexico.l, este se encarga de reconocer si todas las palabras están bien escritas, pertenecen a nuestro lenguaje y otras cosas, por ejemplo, asegurarse de que los enteros no se pasen de rango, los identificadores no tengan más de 16 caracteres y los comentarios multilinea estén bien cerrados. Todo esto es posible mediante expresiones regulares, cuando nuestro analizador léxico reconoce la palabra, signo o número gracias a la expresión regular devuelve su token correspondiente, si no lo reconoce devuelve un error indicando en que parte ha sido.

Para el tratamiento de errores tenemos un modo pánico, este detecta todos los errores que haya seguidos hasta que se encuentre con un token que sea correcto, esto se hace para que no salten múltiples errores léxicos por un único error.

Como se ha mencionado anteriormente, nuestro analizador también detecta que los identificadores no tengan más de 16 caracteres o que los enteros no se pasen de rango, esto se hace en el caso de los enteros usando la variable yytext (puntero a la última cadena de texto que se ajustó al patrón de una expresión regular), la cuál convertimos a entero para poder comprobar que no se pase de rango. Por el otro lado, para los identificadores se usa la variable yyleng (variable global que contiene la longitud de yytext), si esta es menor de 17, es válida.

Con respecto a los comentarios multilinea, hemos utilizado una condición de arranque llamada COMENTARIO, esta se activa cuando se encuentra “/*” omite todo lo que encuentre hasta que llegue a “*/”, si no se encuentra mostrará un error por pantalla indicando en que linea está el inicio del comentario sin cerrar.

Otras variables o funciones de flex que también utilizamos son: las anteriormente mencionadas yytext e yyleng, yylineno (activando la opción yylineno en la sección de declaraciones, flex genera un analizador que mantiene el número de la línea actual leída desde su entrada en la variable global yylineno), yymore() (sirve para indicar que la próxima vez que se empareje una regla, el valor nuevo debería ser concatenado al valor que contiene yytext en lugar de reemplazarlo) e yyval (es una variable especial que se utiliza para almacenar el valor asociado a un token).

Análizador sintáctico

El analizador sintáctico se encarga principalmente de evitar la ambigüedad a la hora de analizar cadenas de símbolos de acuerdo con las reglas de una gramática formal. En el caso de este analizador sintáctico cabe remarcar el tratamiento de la ambigüedad, para ello se define en sintactico.y la asociatividad y la precedencia que tienen los operadores utilizando %left PLUSOP MINUSOP, %left POR BARRAOP y %left UMINUS, lo cual indica que se asocia por la izquierda en las operaciones de suma y resta y que tienen menor precedencia que la multiplicación y división que también se asocian por la izquierda, por último la negación, la cual tiene más precedencia que las anteriores operaciones.

En cuanto a otras estructuras como el if-else tenemos otro tipo de ambigüedad, en este caso tenemos que resolver la asociatividad de cada if con su else, ya que al haber un if-else dentro de otro if-else surge el problema que hace que el compilador tenga que decidir de que if es cada else. En este caso se soluciona utilizando: %expect 1, lo que hace es suprimir la advertencia sobre conflictos que normalmente advierte Bison, solo es necesario esto, porque Bison de manera predeterminada reduce y de esa forma ya se resuelve el conflicto.

Por otro lado, respecto a la sintaxis, tenemos lo básico de C, asignar valores a variables o constantes, declarar un void, if-else, while, operaciones entre enteros, sentencias como print y read con las que podemos imprimir por pantalla o leer y asignar a una variable, etc.

Análisis semántico

El analizador semántico trabaja con una lista de símbolos con la que maneja los posibles errores que puedan darse, en primer lugar tenemos en la declaración de una función el manejo del posible error semántico que se produce si empieza de una forma distinta a esa regla, en segundo lugar tenemos en los identificadores que comprobar que no se re-declare ningún identificador haciendo una búsqueda en la lista de símbolos, si ese símbolo ya se encuentra en la lista se producirá un error semántico por re-declaración del identificador. En tercer lugar, tenemos lo mismo que ocurría para los identificadores pero en este caso para las constantes, una constante tampoco se puede re-declarar y por tanto se hace ese recorrido en la lista de símbolos para su comprobación y manejo del posible error semántico. Por último, para las expresiones se maneja de cara a la re-definición de una variable que la expresión con la que se está trabajando sea de tipo variable y no sea una constante, ya que las constantes no se pueden modificar, manejamos ese posible error semántico comprobando el tipo del símbolo, también podría ocurrir que no se encontrase en la lista de símbolos la variable que buscamos, lo que ocasionaría otro error semántico.

Generación de código

Para la generación de código se necesita la lista de código, gracias a eso se traducen las operaciones a ensamblador. Para realizar esto hay que descomponer cada linea de código en varios pasos, ya que así es como lo hace el lenguaje ensamblador, por ejemplo, una simple operación como a = b + 1, primero habría que leer el valor de b y asignarlo a una variable ($t0), luego asignar el 1 a otra variable ($t1) y ya se podría realizar la suma y guardar el resultado en $t0 o $t1.

Estructuras de datos y funciones principales

En cuanto a las estructuras de datos, hay algunas operaciones que siempre se le da el mismo valor, por ejemplo, los string siempre obtienen valor 4 y siempre en el mismo registro. Sin embargo, para otros hay que comprobar que registro no está ocupado, también hay que liberar un registro que ya se ha usado, otras tienen que hacer una concatenación de un entero con un carácter (tenemos una función para realizar eso).

En ListaSimbolos se define también un tipo de datos llamado Simbolo que será almacenado en esta lista y que está definido por un atributo cadena llamada “nombre”, un tipo y un entero llamado “valor” (es nulo cuando se trata de una variable o una constante). El tipo es un enumerado que podrá ser variable, constante y cadena.

En ListaCodigo se defina el tipo de datos Operacion que contará con los siguientes atributos: un operador (almacena la clase de operación que se trata), un registro (se almacena el resultado de la operación) y dos argumentos, arg1 y arg2 (almacenan los registros en los que se encuentran los argumentos).

Por otro lado, las funciones que utilizamos son: perteneceTablaS, nuevaEntrada, esConstante, imprimir, imprimirCodigo, liberarRegistros, obtenerRegistros, concatenar, concatenar2, obtenerEtiqueta, yyerror y sinErrores.

Manual de usuario

Para comenzar, hay que abrir una terminar en la carpeta en la que están todos los ficheros de nuestro compilador y ejecutar la orden “make”, esto compilará el proyecto y generará un ejecutable llamado “miniC”, lo siguiente que hay que hacer es ejecutar la siguiente instrucción: “./miniC prueba.txt” donde prueba.txt es el fichero que quieras pasar por el compilador de miniC.

Ejemplos de funcionamiento

Primer ejemplo:

```
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";
}
```

Se imprime lo siguiente por el terminal:

```
############################
.data

# STRINGS ##################
$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"

# IDENTIFIERS ##############
_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 $t2,$t0,$t1
            li $t0,2
            sub $t1,$t2,$t0
            sw $t1,_c
            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 $t5,$t3,$t4
        li $t3,1
        add $t4,$t5,$t3
        sw $t4,_c
        b $l1
$l2:

$l4:

$l6:

        la $a0,$str7
        li $v0,4
        syscall
        la $a0,$str8
        li $v0,4
        syscall
##################
# Fin
        li $v0, 10
        syscall
```

Segundo ejemplo:

```
void test1() {
var a;
const b = 3, c = 0;
print "Test 1\n";
a = 1 + b;
read a;
print "Fin test1\n";
}
```

```
############################
.data

# STRINGS ##################
$str1:   .asciiz "Test 1\n"
$str2:   .asciiz "Fin test1\n"

# IDENTIFIERS ##############
_a:      .word 0
_b:      .word 0
_c:      .word 0
###################
# Seccion de codigo
         .text
         .globl main
main:
         li $t0,3
         sw $t0,_b
         li $t0,0
         sw $t0,_c
         la $a0,$str1
         li $v0,4
         syscall
         li $t0,1
         lw $t1,_b
         add $t2,$t0,$t1
         sw $t2,_a
         li $v0,5
         syscall
         sw $v0,_a
         la $a0,$str2
         li $v0,4
         syscall
##################
# Fin
         li $v0, 10
         syscall
```

Tercer ejemplo:

```
void test2() {
var a;
const b = 3;
var a;
const a = 0;
a = 1 + b;
}
```

La salida es la siguiente:

```
Linea 4: Variable a ya declarada
Linea 5: Variable a ya declarada
errores lexicos: 0, errores sintacticos: 0, errores semanticos: 2
Analisis semantico con errores
```