MEMORIA

Compilador

Andrés Hurtado Martínez

ÍNDICE

INTRODUCCIÓN

En este informe se especifica la implementación de un compilador que traduce desde el lenguaje denominado *miniC* a lenguaje ensamblador, interpretable por MIPS.

A lo largo del documento se explica cómo se han desarrollado las tres etapas fundamentales del compilador: análisis léxico, análisis sintáctico y análisis semántico; además de cómo se han tratado los errores y la generación de código.

Por último, se detalla cómo se puede utilizar el programa, se exponen algunas pruebas de código para ver cómo funciona y cuál es el resultado, y termina con una reflexión sobre el proyecto.

EXPLICACIÓN

1. ANÁLISIS LÉXICO

¿QUÉ ES UN ANALIZADOR LÉXICO?

Un analizador léxico es una herramienta que se utiliza para obtener de la entrada una secuencia de símbolos llamados tokens (como palabras clave, identificadores, operadores, números, etc.)

Los tokens generados por el analizador léxico se utilizan como entrada para el siguiente nivel del proceso de compilación, el analizador sintáctico, que será explicado en el próximo apartado.

En lo referente al proyecto, el analizador léxico está contenido en el primer fichero que se desarrolló, *minic.l*, mediante la herramienta Flex.

SECCIONES

1º: En esta primera sección solamente incluimos el fichero minic.tab.h procedente de la generación de la gramática; definimos la función de los errores léxicos; y la variable que cuenta el número de errores léxicos.

2º: MACROS

En esta sección se declara, en primer lugar*, %option yylineno*, para obtener el número de línea a la hora de informar de un error; y también se declaran unas macros que nos ayudan en la siguiente sección a construir las ER, entre ellas:

digito [0-9]

identificador [a-zA-a_][a-zA-Z0-9_]{0,15}

entero {digito}+

3º: EXPRESIONES REGULARES

En esta sección se han definido los terminales de la gramática, entre ellos, los identificadores, los números (enteros), las cadenas y los comentarios, cuyas expresiones regulares se explican más a fondo.

Comentarios:

[\n\t]+ -Esta ER ignora espacios, saltos de línea y tabuladores

"//"(.*)[\n] -Esta ER ignora lo que venga detrás de dos barras (comentario de una línea)

"/*"([^*]|[*]+[^*/])*[*]+"/" -Esta ER ignora lo que esté entre /* y */ (comentario multilínea disponible en el AV)

Enteros:

{entero} -Reconoce un número entero

Cadenas:

\"(\.|[^\"])*\" -Reconoce cadenas de texto

Identificadores:

{identificador} -Esta ER valida los ID's según las condiciones especificadas en la guía.

DIFICULTADES

En un principio, tenía un fichero de cabecera (.h) donde declarábamos todos los tokens, pero observamos en ejemplos del aula virtual que no era necesario incluirlo, y además me daba algunos problemas en la compilación, por lo que opté por eliminarlo.

2. ANÁLISIS SINTÁCTICO

¿QUÉ ES UN ANALIZADOR SINTÁCTICO?

El analizador sintáctico es la segunda etapa del proceso. Esta parte se encuentra en el fichero *minic.y,* y se encarga de verificar que los tokens generados por el analizador léxico sigan las reglas de la gramática del lenguaje, gramática proporcionada en la guía de la práctica.

¿QUÉ NOS ENCONTRAMOS EN EL ANALIZADOR SINTÁCTICO?

En cuanto a la implementación de esta parte, comienza con la declaración de librerías, variables globales y definición de funciones auxiliares.

En el siguiente apartado se declara la cabecera de ListaCodigo (proporcionada en el aula virtual) con *%code requires*, por lo que no hace falta declararla también en *minic.l*.

TOKENS DE LA GRAMÁTICA Y TIPOS DE DATOS

Los tipos de datos de los símbolos se especifican con el operador %union, cuyos campos son:

char *cadena	Tipo para enteros (INT), cadenas (STRING) e identificadores (ID)
ListaC codigo.	Tipo para no terminales (expression, asig, statement…)

PRECEDENCIA DE OPERADORES Y ASOCIATIVIDAD

Para que las operaciones aritméticas se hagan en el orden adecuado hay que establecer un orden de precedencia sobre los operadores aritméticos (+, -, *, /) con el operador *%left* (asociatividad por la izquierda). En primer lugar, se declaran la suma y la resta, y luego la multiplicación y división, para que tengan mayor precedencia. Por último, declaramos un nuevo símbolo llamado *UMENOS* debajo de los otros operadores para que adquiera la precedencia máxima.

REGLAS DE LA GRAMÁTICA

Las reglas de la gramática reducen las expresiones y llevan a cabo las acciones que van construyendo las secciones *.data* y *.text* del código ensamblador.

ERRORES SINTÁCTICOS

Para el conteo de errores sintácticos se ha declarado la función yyerror(const char *s). Esta función simplemente suma uno, a un contador de errores e imprime un mensaje de error.

DIFICULTADES

Uno de los principales contratiempos en esta etapa fue un conflicto de reducción-desplazamiento en la parte del IF y IF-ELSE. Sin embargo, entendimos que este error es normal, ya que Bison interpreta que puede reducir el IF o seguir desplazando usando la regla IF-ELSE.

Una función que dio bastantes problemas en el tramo final de la implementación fue *liberarReg(char *reg)*. Pero después de depurar varias veces la ejecución y buscar distintos tipos de soluciones lo resolví cogiendo el tercer elemento del array, que se correspondería con el valor del registro (‘$t**x**’), y restándole el valor del carácter ‘0’ (48).

3. ANALIZADOR SEMÁNTICO

¿QUÉ ES UN ANALIZADOR SEMÁNTICO?

El analizador semántico es la última etapa del proceso, y se apoya en la información proporcionada por los analizadores léxico y sintáctico. Se ocupa de que el código tenga sentido más allá de su sintaxis, es decir, garantiza que el código esté bien formado y sea coherente.

ERRORES SEMÁNTICOS

Los errores que hay que comprobar son:

1. Variables o constantes sin declarar
2. Variables o constantes declaradas dos veces
3. Reasignación de constantes

El 1er error se da en asignaciones de valores a variables o constantes no declaradas previamente. Para saber si han sido o no ya declaradas, tenemos que buscar en la tabla el ID (nombre) de la variable. Esto lo podemos hacer con la función *perteneceTablaS(Lista l, char *nombre).*

El 2º error lo solucionamos buscando en la tabla el ID que se quiere añadir. Si ya está, nos encontraríamos antes una redefinición de una variable o constante.

El 3er error, también se da en asignaciones, en este caso se asigna un valor a una constante. Este error se detecta con la función *esConstante(Lista l, char *nombre)*

Al detectar cualquier error de los anteriores, se suma uno al contador de errores semánticos.

FUNCIONES AUXILIARES

Para esta etapa se ha hecho uso de los ficheros *listaSimbolos.c, listaSimbolos.h, listaCodigo.c y listaCodigo.h*, proporcionados por el aula virtual.

En este apartado y el siguiente se explican un poco las funciones añadidas a cada uno de los ficheros.

listaSimbolos.c:

1. *perteneceTabla(Lista l, char *nombre):* esta función comprueba si un ID ha sido añadido a la tabla de símbolos previamente.
2. *esConstante(Lista l, char *nombre):* esta función comprueba si un ID es de tipo constante.
3. *anadeEntrada(Lista l, char *nombre, Tipo tipo)*: esta función añade un ID a la tabla. Cabe señalar que esta función solo se utiliza cuando el tipo es variable o constante, ya que en la sección .data estos tipos se inicializan a 0, por tanto el valor por defecto del campo valor, es 0, permitiendo omitir un cuarto parámetro. En el caso de que sea una cadena, habría que gestionar el contador de cadenas, por lo que el valor del campo valor no será siempre 0. Dado que esto ocurre únicamente una vez, se ha considerado incluir las cadenas en la tabla de forma "manual", inicializando una variable tipo Simbolo con los valores correspondientes y añadirlo a la tabla mediante la función *insertaTabla()*, ya proporcionada.
4. *imprimeTablaS(Lista l)*: esta función se encarga de imprimir toda la información que contiene la lista con el formato requerido. (generación de código)

4. GENERACIÓN DE CÓDIGO

listaCodigo.c:

1. *char *obtenerReg()*: esta función devuelve una cadena que hace referencia a un registro que no está siendo usado.
2. *char *nuevaEtiqueta()*: esta función se encarga de gestionar el orden de las etiquetas haciendo uso del contador de etiquetas.
3. *void liberarReg(char *reg)*: recibe la posición de un registro en el array de registros y lo pone a 0.
4. *void imprimirCodigo(ListaC codigo)*: imprime por pantalla la parte del código correspondiente a la sección *.text*.
5. *char *concatena(char c, char *l)*: esta función se usa para añadir la barra baja a las variables.

DIFICULTADES

El principal inconveniente fue añadir la barra baja a los identificadores, para solucionarlo, decidimos basarnos en la función asprintf(), vista en la función nuevaEtiqueta(), que se proporciona en el aula virtual. Esta función se utiliza para asignar una cadena de caracteres a un puntero de cadena (char *) y formatear la cadena de forma dinámica.

ERRORES

Para contabilizar los errores se han declarado en el *main.c* y en el *minic.y* tres variables para llevar el conteo de los errores léxicos, sintácticos y semánticos. Además se ha creado una función llamada *noHayErrores()*, que devuelve 1 (True) si no hay errores o 0 (False) si hay algún error.

Por otra parte, en el *main.c* se ha creado una función denominada *errores()*, que suma todos los errores para luego imprimirlos por pantalla.

MANUAL DE USO

La principal herramienta para el uso del compilador es el fichero *makefile*. En él se encuentran las órdenes para generar:

1. El fichero.l (lexico) con Flex	lexico
2. El fichero.y (gramática) con Bison	gramatica
3. Compilar el programa	compilar
4. Borrar los ficheros generados en las órdenes anteriores	limpiar
5. Ejecutar el programa resultante con un fichero de entrada	run

Para ejecutar cada orden es necesario escribir en una terminal ‘make’ antes de su nombre, por ejemplo, make run.

El orden para ejecutar un fichero de prueba sería:

1. make gramatica
2. make lexico
3. make compilar
4. make run
5. make limpiar (en caso de que queramos borrar lo generado anteriormente)

Cabe señalar que en la generación de la gramática aparece un conflicto desplazamiento/reducción y en la compilación aparece un warning por el uso de la función asprintf().

PRUEBAS DE CÓDIGO

Entrada 1:

```
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";
}
```

Salida 1:

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

Entrada 2:

```
void prueba () {
   // Declaraciones
   const a=0, b=0;
   var c=5+2-2;
   var a;

}
```

Salida 2:

```
Variable a ya declarada
0 Errores lexicos
0 Errores sintacticos
1 Errores semanticos
```

Entrada 3:

```
void prueba () {
   // Declaraciones
   const a=0, b=0;
   var c=5+20-20;
   a = 4;
}
```

Salida 3:

```
Asignacion a constante
0 Errores lexicos
0 Errores sintacticos
1 Errores semanticos
```

Entrada 4: (Error en el paréntesis)

```
void prueba()) {
const  a=0, b=0;
var c=5+2-2;
print "Inicio del programa\n";
}
```

Salida 4:

```
Error: syntax error
0 Errores lexicos
1 Errores sintacticos
0 Errores semánticos
```

Entrada 5:

```
void prueba(){
   // Declaraciones
   const n =10;
   var suma = 0;
   var r = 0;
   var i=0;
   read a;
   print "a + b = ", r;
```

Salida 5:

```
Variable a no declarada
0 Errores lexicos
0 Errores sintacticos
1 Errores semanticos
```

Entrada 6:

```
void prueba2(){
   // Declaraciones
   const n =10;
   var suma = 0;
   var r = 0;
   var i=0;
   var a ,b;
   print "Introduce dos numeros: ";
   read a, b;
   r=a+b;
   print "a + b = ", r, "\n";
   do{
      // Entramos en el bucle

      i = i + 2;
      suma = suma + i;
      print "Numero par -> ",i,"\n";
   } while(i-n);
   print "La suma de los primeros numeros pares es: ",suma,"\n";
}
```

Salida 6:

```
# Seccion de datos
   .data

$str1:
   .asciiz "Introduce dos numeros: "
$str2:
   .asciiz "a + b = "
$str3:
   .asciiz "\n"
$str4:
   .asciiz "Primer par de la sucesion -> "
$str5:
   .asciiz "\n"
$str6:
   .asciiz "La suma de los primeros numeros pares es: "
```

```
$str7:
   .asciiz "\n"
_n:
        .word 0
_suma:
        .word 0
_r:
        .word 0
_i:
        .word 0
_a:
        .word 0
_b:
        .word 0

####################
# Seccion de codigo
   .text
   .globl main
main:
  li $t0, 10
  sw $t0, _n
  li $t0, 0
  sw $t0, _suma
  li $t0, 0
  sw $t0, _r
  li $t0, 0
  sw $t0, _i
  la $a0, $str1
  li $v0, 4
  syscall
  li $v0, 5
  syscall
  sw $v0, _a
  li $v0, 5
  syscall
  sw $v0, _b
  lw $t0, _a
  lw $t1, _b
  add $t0, $t0, $t1
  sw $t0, _r
  la $a0, $str2
  li $v0, 4
  syscall
  lw $t0, _r
  move $a0, $t0
  li $v0, 1
```

```
  syscall
  la $a0, $str3
  li $v0, 4
  syscall
$l1:
  lw $t0, _i
  li $t1, 2
  add $t0, $t0, $t1
  sw $t0, _i
  lw $t0, _suma
  lw $t1, _i
  add $t0, $t0, $t1
  sw $t0, _suma
  la $a0, $str4
  li $v0, 4
  syscall
  lw $t0, _i
  move $a0, $t0
  li $v0, 1
  syscall
  la $a0, $str5
  li $v0, 4
  syscall
  lw $t0, _i
  lw $t1, _n
  sub $t0, $t0, $t1
  bnez $t0, $l1
  la $a0, $str6
  li $v0, 4
  syscall
  lw $t0, _suma
  move $a0, $t0
  li $v0, 1
  syscall
  la $a0, $str7
  li $v0, 4
  syscall

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

MEJORAS

He incluido la mejora del DO WHILE.

La implementación es sencilla una vez entendido e implementado el resto de statements. A nivel de generación de código ensamblador, el do-while tan solo necesita: una etiqueta inicial, a la que retornará cuando la condición de salida no se cumpla, en este caso, que la variable dentro del while sea cero; y una operación de comparación que compruebe la condición y salte o continúe en función del resultado.

A nivel de gramática el primer statement se puede reducir en un statement, o en una lista de statements ({statement}), por eso no escribimos las llaves en la declaración de do-while. A partir de ahí ya puede reducirse, por ejemplo, en un print_list -> print_item -> STRING o en una expresión para decrementar o incrementar la variable de control…

Para poder reconocer esta mejora, hemos tenido que añadir el token DO en el fichero.l

“do” return DO;

Seguidamente, lo incluimos en la lista de tokens del fichero.y

%token DO…

Por ultimo, hay que añadir la sentencia do-while en los statements

| DO statement WHILE LPAR expression RPAR SEMICOMA

CONCLUSIONES

En mi opinión, el desarrollo del compilador me ha parecido un buen proyecto para comprender y afianzar los contenidos de la asignatura, además de aprender el manejo de las herramientas Flex y Bison.

En cuanto al proceso de implementación, cada fase ha presentado una serie de dificultades que se han solucionado a base de ensayo y error, investigando en internet, etc.