MEMORIA PRÁCTICA COMPILADORES

ASIGNATURA: Compiladores

PROFESOR: Eduardo Martínez Gracia

ALUMNOS: David Fernández Expósito y

Pablo Tadeo Romero Orlowska

CORREO: david.fernandeze@um.es y pablotadeo.romeroo@um.es

DNI: [redacted]

GRUPO: PCEO

FECHA: 10 mayo de 2022

ÍNDICE DE CONTENIDOS

1. INTRODUCCIÓN

Este documento especifica la implementación de la práctica de la asignatura Compiladores. El objetivo de la práctica era diseñar un compilador para el lenguaje miniC. La descripción del lenguaje se puede encontrar en el documento de la especificación de la práctica.

Este documento consta de varias secciones. Primero, se trata el análisis léxico (herramienta Flex). Luego, el análisis sintáctico, el semántico, y la generación de código, todo con la herramienta Bison. Por último, se muestra la salida que se obtiene al compilar el programa de ejemplo disponible en la especificación de la práctica, pero ligeramente modificado para incluir las mejoras implementadas (do-while y for).

Un comentario que cabe hacer es que no se va a explicar todo el proceso seguido de forma exhaustiva, pues es imposible en cuanto a tiempo. En este documento simplemente se irán resumiendo los pasos seguidos y se destacarán los detalles más importantes del desarrollo de la práctica.

2. ANÁLISIS LÉXICO

Lo primero que se hizo en el desarrollo de la práctica fue diseñar el analizador léxico. Para ello, se usó la herramienta Flex. El archivo creado para esto es miniC.l.

Las expresiones regulares usadas para las palabras reservadas o símbolos especiales son triviales, simplemente poner la propia palabra. Por ejemplo, “const” es una de las ER. Sin embargo, sí que es importante el orden en el que se ponen las expresiones regulares. En este caso, es importante notar que las ER de las palabras reservadas se pusieron antes de la ER de los identificadores, pues si no se hiciera esto todas las palabras clave se tomarían como cadenas pues también cumplen esa ER.

Otro aspecto importante que se debe tratar en esta sección son los comentarios, que obviamente el compilador debe hacer como que no existen (siempre y cuando estén correctamente introducidos claro). Los comentarios de una sola línea se tratan fácilmente con una ER: *//.**, que captura toda la línea que comienza con //. Sin embargo, los comentarios multilínea necesitan un tratamiento algo más complejo. Para este tipo de comentarios son necesarias 3 ER diferentes que conllevarán acciones diferentes. La primera ER es /* que captura el inicio de un comentario multilínea. En este caso, lo que se hace es almacenar en una variable el número de la línea donde se inicia el comentario y se hace *BEGIN(comentario)* (comentario es como una variable que se declara antes, lo que nos permite decirle a Flex que queremos que se siga dentro del ámbito del comentario). La segunda ER es *<comentario>(.|\n)*. En este caso la acción es vacía, se sigue. La tercera es *<comentario>*/*. En este caso el comentario se ha cerrado y la acción es *BEGIN(INITIAL)*. La última opción sería que se encontrase un <<EOF>>, en cuyo caso debe saltar un error, por lo que la ER es *<comentario><<EOF>>* y la acción es llamar a una función *error_comentario()* que simplemente imprime por pantalla un mensaje de error e informa de la línea donde se encuentra el comentario sin cerrar (usando la variable comentada antes), y se contabiliza el error léxico.

En cuanto al tratamiento de errores, se encuentran 3 expresiones regulares (además de la comentada anteriormente para los comentarios no cerrados) que cabe mencionar:

- *\"([^"\n]|\\\")**, esta ER reconoce las cadenas que no se cierran correctamente con comillas.
- *{letra}({letra}|{digito}){16}({letra}|{digito})**, esta ER reconoce identificadores que no son válidos por ser más largos de lo permitido (máximo 16 caracteres).
- *{panico}+*, con pánico siendo: *[^\n\r\ta-zA-Z_0-9+\-*/();=, {}]*. Esta ER básicamente acepta toda cadena que no haya sido aceptada por expresiones puestas antes, por eso se pone la última. Además, con esto conseguimos implementar el modo pánico, porque se aceptará todo con esta ER hasta que alguna anterior acepte algo y se siga con el análisis.

Por último, como implementamos las mejoras para poder optar a la máxima nota, incluimos en el análisis léxico lo necesario para implementar el for y el do-while. Lo único que se tuvo que hacer es añadir 2 nuevos tokens, el token FOR y el DO.

3. ANÁLISIS SINTÁCTICO

La gramática de miniC que usamos fue la misma que se proporciona con la especificación de la práctica, no hicimos ningún cambio. Lo único que cabe comentar en cuanto a la gramática es lo que se tuvo que añadir para implementar el for y el do-while.

Para el for, solo se tuvo que añadir la regla: *statement : for (NUMERO) statement*

Para el do-while, solo se tuvo que añadir la regla:

statement : do statement while (expression)

Vista la gramática, cabe comentar algunos detalles. En primer lugar, hay que notar que la gramática es ambigua, y tendremos problemas con las reglas correspondientes a las operaciones aritméticas. Para solucionar esto, se tuvo que definir la prioridad y asociatividad para las operaciones. Obviamente la prioridad que usamos es la usual: productos y divisiones tienen prioridad a sumas y restas. Para indicar a Bison que la prioridad es ésta, se añadieron las siguientes líneas de código antes de las reglas de producción:

%left "+" "-"

%left "" "/"*

%precedence UMINUS

Cuanto más abajo, más prioridad, por lo que primero se indica que si encontramos un –, éste tiene prioridad. Luego, se pone la prioridad usual antes comentada. Para que esto tenga efecto en la regla de producción de -, se añadió %prec UMINUS en la regla *expression : - expression %prec UMINUS*

Además, también vamos a tener problemas con las reglas del if. En concreto, tendremos un conflicto desplaza-reduce, pues Bison verá que puede reducir con la regla del if que no tiene else o seguir desplazando para completar el if-else. En este caso, lo que queremos

es que desplace, y por suerte Bison hace eso por defecto. Para que Bison no nos muestre un warning cada vez que se ejecute, lo que hacemos es meter la siguiente línea antes de todas las reglas de producción:

%expect 1

Con esto lo que hacemos es "avisar" a Bison de que va a haber un conflicto y que no tiene que avisar de él.

Por último, cabe mencionar que se declaró y construyó una función *yyerror* que Bison se encargará de llamar en caso de error sintáctico. En esta función, se aumenta en uno el contador de errores sintácticos y se muestra un mensaje de error por pantalla informando del mismo.

4. ANÁLISIS SEMÁNTICO

En el análisis semántico lo más destacable es cómo se gestionan los identificadores. Nuestro compilador debe ser capaz de ir almacenando en la tabla de símbolos los identificadores que va encontrando declarados, y deben saltar errores si se redeclara un identificador o se intenta usar uno que no se ha declarado previamente.

Para esta parte de la práctica, nos apoyamos en listaSimbolos.c y listaSimbolos.h, proporcionadas por los profesores. Con estos ficheros (debemos incluirlos en nuestro fichero miniC.y claro) podremos usar una lista de símbolos sin tener que implementarla nosotros.

Las comprobaciones relacionadas con el análisis semántico fueron:

- En las reglas *asig : id* y *asig : id = expression*, se busca el identificador en la lista de símbolos, y si se llega al final de la lista sin encontrarlo está todo correcto pues el identificador no ha sido previamente declarado. En caso contrario, el identificador está siendo redeclarado y se informa del error y se añade uno al conteo de errores semánticos.
- En la regla *statement : id = expression* se busca el identificador en la lista, y si se llega al final sin encontrarlo se informa y contabiliza el error pues se está usando un identificador no declarado. Además, si se encuentra pero el tipo del identificador es CONSTANTE, se informa y contabiliza el error pues no se puede hacer una asignación a un identificador de tipo constante.
- En las reglas *read_list : id* y *read_list : read_list , id* se busca el identificador y si no se encuentra se informa y contabiliza el error pues se está usando un identificador no declarado. Además, si se encuentra pero es de tipo CONSTANTE también se informa y contabiliza un error pues no se puede asignar nada después de leer a un identificador de tipo constante.
- En la regla *expression : id* se busca el identificador en la lista de símbolos y se informa y contabiliza un error en caso de no encontrarlo.

Cabe mencionar que, en las dos primeras comprobaciones (el primer punto), si no hay ningún error semántico, se introduce el identificador en la lista de símbolos. Aunque en

el código se usen los identificadores con una barra baja delante, decidimos no meterlos así en la lista de símbolos. Simplemente se meten con su nombre normal, y luego a la hora de generar código e imprimir por pantalla se creará la cadena correcta con la barra baja delante. Al final, el resultado obtenido es el mismo. Solamente hay que tener esto en cuenta pues a la hora de comprobar si un identificador está en la lista hay que tener claro si se ha metido con barra baja o no.

También se usa la lista de símbolos para el manejo de cadenas en los print pero eso lo comentaré en la sección de generación de código cuando trate la función *imprimirLS()*.

5. ERRORES

En los apartados anteriores se comenta cómo se van contabilizando los errores. Básicamente en el main.c se declaran 3 variables, para contabilizar los errores léxicos, sintácticos y semánticos. Además, se define una función *análisis_ok()* en miniC.y que comprueba que la suma valga 0 (es decir, que no haya errores de ningún tipo).

Por último, al final del main.c, en caso de que no sea 0 la suma del número de errores, se muestra por pantalla el número de errores de cada tipo.

6. GENERACIÓN DE CÓDIGO

En este apartado solo se comentarán los métodos o cosas que hemos añadido nosotros para la generación de código, porque es imposible comentar toda la generación de código, sería demasiado extenso. Además, en los pdf proporcionados en el Aula Virtual se indican los pasos a seguir en cada caso y lo único que se hace es ir concatenando operaciones MIPS en el orden correcto en cada caso. No obstante, sí que cabe mencionar un par de cosas. Al igual que en el análisis semántico, la generación de código implementada se apoya en el uso de listaCodigo.c y listaCodigo.h, proporcionados por los profesores, que nos permiten trabajar con listas de código sin tener que implementarlas nosotros. Además, para que no haya ningún problema, antes de empezar a generar código en cada regla se comprueba si *análisis_ok()*, porque si no es así no tiene sentido generar código (además podría producir comportamientos no deseados)

En primer lugar, cabe mencionar cómo se introducen las etiquetas en el código. Como sólo se pueden insertar operaciones, lo que se hace cada vez que se quiere meter una etiqueta en alguna parte del código es insertar una operación “etiq” con el valor de la etiqueta en el resultado. Luego, a la hora de imprimir la lista de código se comprueba si la operación es “etiq”, y en ese caso se imprime la etiqueta con el formato adecuado.

En segundo lugar, se crearon una serie de métodos para facilitar la generación de código:

- *imprimirLS()*: imprime la tabla de símbolos del análisis semántico. En un programa en MIPS al principio está la parte de datos. Por tanto, después de generar todo el código en la lista de código, lo primero que se hace es llamar a este método para que se imprima la sección de datos con el formato correcto. También me gustaría mencionar cómo se guarda el valor de un string que se quiera imprimir

con print. En estos casos lo que se hace es meter a la lista de símbolos un símbolo con nombre la cadena en sí, tipo CADENA, y valor el número de cadena (llevamos un contador de cadenas para esto). Luego, a la hora de imprimir por pantalla se puede generar el nombre correcto de la etiqueta en la sección de datos usando los valores que se almacenaron en la lista de símbolos.

- *imprimirLC()*: imprime la lista de código. Lo que se hace es recorrer la lista de código e ir imprimiendo las operaciones con el formato adecuado. Se comprueba si la operación es "etiq", pues en ese caso solo se debe imprimir la etiqueta con el formato correspondiente.
- *obtenerReg()*: tenemos 10 registros temporales para usar. Esta función recorre los registros y devuelve el primero disponible.
- *liberarReg()*: libera el registro pasado por parámetro. Esta función es importante pues nos permite, a lo largo de la generación de código, ir indicando qué registros se pueden volver a usar. Si no se hiciera esto, se irían agotando los registros y llegaría un momento en el que no se podría generar el código por falta de registros.
- *obtenerEtiqueta()*: esta función genera etiquetas. Las etiquetas son de la forma $lx donde x es el nº de etiquetas. Por tanto, esta función hace uso de un contador que controla cuantas etiquetas se van usando, para así poder generar los nombres de forma correcta.

7. PRUEBA DEL CÓDIGO FINAL

Para probar que todo funciona correctamente, se ejecutó el compilador creado junto con el programa de prueba, *prueba.txt*, y se comprobó que el código MIPS generado era correcto. Esta comprobación es sencilla pues en la especificación de la práctica se muestra la salida que se debe obtener al ejecutar el programa de prueba, y aunque nuestro programa no es exactamente el mismo pues le añadimos un par de bucles para probar que el do-while y el for funcionan, la mayoría del código generado coincide. Además, también se instaló la herramienta *spim* que permite ejecutar el código MIPS de un fichero (se redirigió la salida a un fichero salida.s para poder usar esta herramienta), por lo que se comprobó que la salida del programa era correcta.

A continuación, se muestra el programa *prueba.txt* con el que se realizaron las comprobaciones descritas:

```
/************************
 * Fichero de prueba nº 1
 ************************/


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

```
    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";
    do {
        print "hola";
        print "hola";
    } while (c);
    for(3){
        print "FOR\n";
    }
}
```

El código MIPS que se genera es:

```
.data
_a:
  .word 0
_b:
  .word 0
_c:
  .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 "Final"
$str8:
        .asciiz "\n"
$str9:
        .asciiz "hola"
$str10:
        .asciiz "hola"
$str11:
        .asciiz "FOR\n"
.text
.globl main
main:
         li $t0,0
         sw $t0,_a
         li $t0,0
         sw $t0,_b
         li $t0,5
         li $t1,20
         add $t0,$t0,$t1
         li $t1,20
         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
$l7:
        la $a0,$str9
        li $v0,4
```

```
        syscall
        la $a0,$str10
        li $v0,4
        syscall
        bnez $t0,$l7
        li $t0,0
$l8:
        bge $t0,3,$l9
        la $a0,$str11
        li $v0,4
        syscall
        addi $t0,$t0,1
        j $l8
$l9:
        jr $ra
```

La salida obtenida al ejecutar el programa con *spim* es:

```
c =5
c =4
c =3
c =2
c =1
Final
holaholaFOR
FOR
FOR
```

8. CONCLUSIONES

Como conclusión, estamos muy satisfechos con el resultado final de la práctica, ya que hemos podido realizar todo lo que se pedía de forma satisfactoria, aún encontrando ciertas dificultades, y hemos sabido añadir las funcionalidades extras requeridas para poder optar a la máxima nota en la práctica.

Durante el transcurso de la práctica, hemos encontrado ciertas dificultades, pero con la ayuda del profesor de prácticas, Eduardo, finalmente fuimos capaces de solventar los problemas. Donde más problemas tuvimos fue en la generación de código, pues en un principio no nos quedó muy clara. Pero conforme fuimos avanzando el trabajo fuimos comprendiendo y dominando más esta parte de la práctica y al final pudimos completar el trabajo de forma satisfactoria.

Así, pensamos que ésta ha sido una práctica que nos ha ayudado a comprender mejor los conceptos vistos en la teoría de la asignatura, ya que hemos podido ver cómo funcionan los algoritmos y conceptos vistos en clase, pero en un escenario más real y práctico.

En cuanto al tiempo dedicado a la práctica, la mayoría del tiempo fue sacado de las sesiones de laboratorio semanales de la asignatura, aunque al final del desarrollo de la práctica sí que tuvimos que dedicar cierto tiempo (unas 10 horas) a terminar la parte de generación de código y a elaborar esta memoria, y a comprobar que todo estuviese correctamente antes de entregar el trabajo.