
REPORTES DE PRACTICA

COMPILADORES

ALUMNO: ÁNGEL LÓPEZ MANRÍQUEZ

PROFESOR: TECLA PARRA ROBERTO

ESCUELA SUPERIOR DE COMPUTO

Índice

1. YACC basico: calculadora de vectores	3
1.1. Introduccion	3
1.2. Objetivo	4
1.3. Desarrollo	4
1.4. Conclusiones	5
2. Uso de Java para modo grafico	6
2.1. Introduccion	6
2.2. Objetivo	6
2.3. Desarrollo	6
2.4. Conclusiones	8
3. Tabla de simbolos	9
3.1. Introduccion	9
3.2. Objetivo	10
3.3. Desarrollo	10
3.4. Conclusiones	13
4. Maquina de pila	14
4.1. Introduccion	14
4.2. Objetivo	14
4.3. Desarrollo	14
4.4. Conclusiones	16
5. Decisiones if y ciclos while	17
5.1. Introduccion	17
5.2. Objetivo	17
5.3. Desarrollo	17
5.4. Conclusiones	21
6. Lazo for	22
6.1. Introduccion	22
6.2. Objetivo	22
6.3. Desarrollo	22
6.4. Conclusiones	24
7. Funciones y procedimientos	25
7.1. Introduccion	25
7.2. Objetivo	25

7.3. Desarrollo	25
7.4. Pruebas	30

About This File

This file was created for the benefit of all teachers and students wanting to use Latex for tests/exams/lessons/thesis/articles etc.

The entirety of the contents within this file, and folder, are free for public use.

YACC basico: calculadora de vectores

1.1. Introduccion

Los programas yacc y lex son herramientas de gran utilidad, compatibles y necesarias entre sí, para un diseñador de compiladores. Muchos compiladores se han construido utilizando estas herramientas (p.ej., el compilador de C de GNU (gcc)) o versiones más avanzadas. Los programas bison y flex son las versiones más modernas (no comerciales) de yacc y lex, y se distribuyen bajo licencia GPL con cualquier distribución de Linux (y también están disponibles para muchos otros UNIX). El programa lex genera analizadores léxicos a partir de una especificación de los componentes léxicos en términos de expresiones regulares (en el estilo de UNIX); lex toma como entrada un fichero (con la extensión.l) y produce un fichero en C (llamado "lex.yy.c") que contiene el analizador léxico. Yacc es un programa para generar analizadores sintácticos. Las siglas del nombre significan Yet Another Compiler-Compiler, es decir, "otro generador de compiladores más". Genera un analizador sintáctico (la parte de un compilador que comprueba que la estructura del código fuente se ajusta a la especificación sintáctica del lenguaje) basado en una gramática analítica escrita en una notación similar a la BNF. Yacc fue desarrollado por Stephen C. Johnson en AT&T para el sistema operativo Unix. Después se escribieron programas compatibles, por ejemplo Berkeley Yacc, GNU bison, MKS yacc y Abraxas yacc. Cada una ofrece mejoras leves y características adicionales sobre el Yacc original, pero el concepto ha seguido siendo igual. Yacc también se ha reescrito para otros lenguajes, incluyendo Ratfor, EFL, ML, Ada, Java, y Limbo.

1.2. Objetivo

El alumno completará/arreglará uno de los posibles programas que le permita generar código de soporte en lenguaje C o Java. Escribir una calculadora para: números complejos. Para esto escriba una especificación de yacc para evaluar expresiones que involucren operaciones con números complejos.

1.3. Desarrollo

La práctica consistió básicamente en desarrollar el archivo.y (modificando el archivo hoc1.y, brindado desde la carpeta donde se ejemplifica este hoc) para poder realizar la calculadora de manera correcta. Además de tener que realizarle cambios pequeños a un archivo.l. El profesor nos compartió las funciones que realizan las operaciones con vectores, por lo que esta parte solo tuvo que ser añadida al archivo.y, en el cual básicamente describimos como deben ser las entradas para posteriormente compilar el archivo y.tab.c (con compilador gcc) y poder ejecutar el programa siguiendo el análisis sintáctico brindado por yacc. El archivo.y básicamente debe tener la sección de reglas desarrolladas de manera correcta, al igual que la declaración de los tokens:

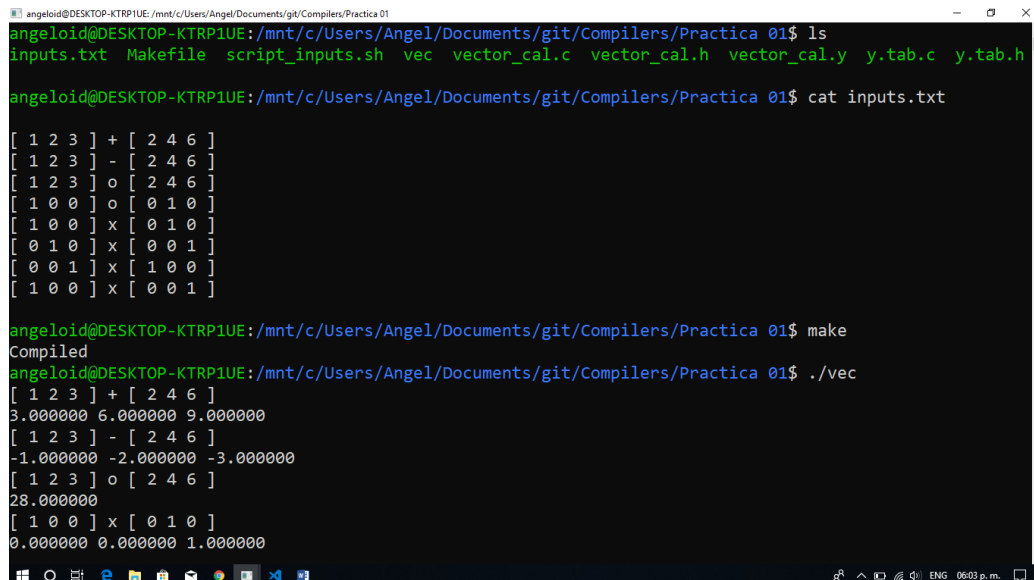
```
34 list: '\n'
35     | exp '\n' { imprimeVector($1); }
36     | number '\n' { printf("%lf\n", $1); }
37     ;
38
39 exp: vector
40     | exp '+' exp {$$ = sumaVector($1,$3); }
41     | exp '-' exp {$$ = restaVector($1,$3); }
42     | NUMBER '*' exp {$$ = escalarVector($1, $3);}
43     | exp '*' NUMBER {$$ = escalarVector($3,$1); }
44     | exp 'x' exp {$$ = productoCruz($1, $3); }
45     ;
46
47 number: NUMBER
48     | vector 'o' vector {$$ = productoPunto($1,$3);}
49     | '|' vector '|' {$$ = magnitudVector($2); }
50     ;
51
52 vector : '[' NUMBER NUMBER NUMBER ']' {Vector *v = creaVector(3);
53                                     v->vec[0] = $2;
54                                     v->vec[1] = $3;
55                                     v->vec[2] = $4;
56                                     $$ = v;
57                                     }
```

En la imagen anterior podemos ver las declaraciones de los tokens y el desarrollo correcto de la sección de reglas (considerando las funciones que hacen las operaciones, tomadas desde el archivo “vector_cal.h”, compartido por el profesor).

Para realizar la práctica es importante recordar que el tipo de yylval es el tipo de los elementos de la pila del yacc (en este caso, apuntador a números complejos). A partir de esto, pudimos

desarrollar correctamente las reglas sintácticas. También tuvimos que borrar la definición que se tenía del `yylex()` en el archivo `l` brindado por el profesor, y modificar el tipo de retorno de dicho archivo, el cual ya con esto, nos permitió crear los números complejos resultantes de las operaciones.

Una vez que se hicieron los cambios necesarios, se ejecutó el archivo `makefile` compartido por el profesor, y posteriormente se pudo ejecutar correctamente la calculadora, proponiendo las siguientes pruebas:



```
angeloid@DESKTOP-KTRP1UE: /mnt/c/Users/Angel/Documents/git/Compilers/Practica 01
angeloid@DESKTOP-KTRP1UE:/mnt/c/Users/Angel/Documents/git/Compilers/Practica 01$ ls
inputs.txt  Makefile  script_inputs.sh  vec  vector_cal.c  vector_cal.h  vector_cal.y  y.tab.c  y.tab.h

angeloid@DESKTOP-KTRP1UE:/mnt/c/Users/Angel/Documents/git/Compilers/Practica 01$ cat inputs.txt

[ 1 2 3 ] + [ 2 4 6 ]
[ 1 2 3 ] - [ 2 4 6 ]
[ 1 2 3 ] o [ 2 4 6 ]
[ 1 0 0 ] o [ 0 1 0 ]
[ 1 0 0 ] x [ 0 1 0 ]
[ 0 1 0 ] x [ 0 0 1 ]
[ 0 0 1 ] x [ 1 0 0 ]
[ 1 0 0 ] x [ 0 0 1 ]

angeloid@DESKTOP-KTRP1UE:/mnt/c/Users/Angel/Documents/git/Compilers/Practica 01$ make
Compiled
angeloid@DESKTOP-KTRP1UE:/mnt/c/Users/Angel/Documents/git/Compilers/Practica 01$ ./vec
[ 1 2 3 ] + [ 2 4 6 ]
3.000000 6.000000 9.000000
[ 1 2 3 ] - [ 2 4 6 ]
-1.000000 -2.000000 -3.000000
[ 1 2 3 ] o [ 2 4 6 ]
28.000000
[ 1 0 0 ] x [ 0 1 0 ]
0.000000 0.000000 1.000000
```

1.4. Conclusiones

Yacc es un programa que permite generar analizadores sintácticos basándose en una gramática analítica escrita, mientras que `lex` genera analizadores léxicos por lo que es necesario utilizar ambos para poder crear programas que simulen el proceso de compilación de otros programas. En la presente práctica pudimos comprobar el uso y efectividad de `yacc`, en conjunto con `lex`, en sus versiones respectivas: `bison` y `flex`, para Ubuntu, a la hora de crear una calculadora de números complejos que comprueba el orden y sintaxis de lo que se escribe como entrada y da un resultado de acuerdo al análisis de los operadores y números escritos.

Uso de Java para modo grafico

2.1. Introduccion

YACC proporciona una herramienta general para describir la entrada a un programa de computadora. El usuario de YACC especifica las estructuras de su entrada, junto con el código que se invocará a medida que se reconoce cada estructura. YACC convierte dicha especificación en una subrutina que maneja el proceso de entrada; con frecuencia, es conveniente y apropiado tener la mayoría del flujo de control en la aplicación del usuario manejada por esta subrutina. La entrada del programa de computadora generalmente tiene cierta estructura; de hecho, se puede pensar que cada programa de computadora que ingresa define un "idioma de entrada" que acepta. Un lenguaje de entrada puede ser tan complejo como un lenguaje de programación, o tan simple como una secuencia de números. Desafortunadamente, en la entrada habitual las instalaciones son limitadas, difíciles de usar y, a menudo, son poco estrictas a la hora de verificar si sus entradas son válidas.

2.2. Objetivo

Que el alumno escoja uno de los posibles programas que le permita generar código de soporte en lenguaje Java. Utilizar la carpeta grafibasi para dibujar círculos, líneas y rectángulos (usar polimorfismo).

2.3. Desarrollo

Para realizar este programa partiremos del código que nos fue proporcionado por el profesor, que sólo admite una cadena con un formato que no es el correcto para esta práctica. Para solu-

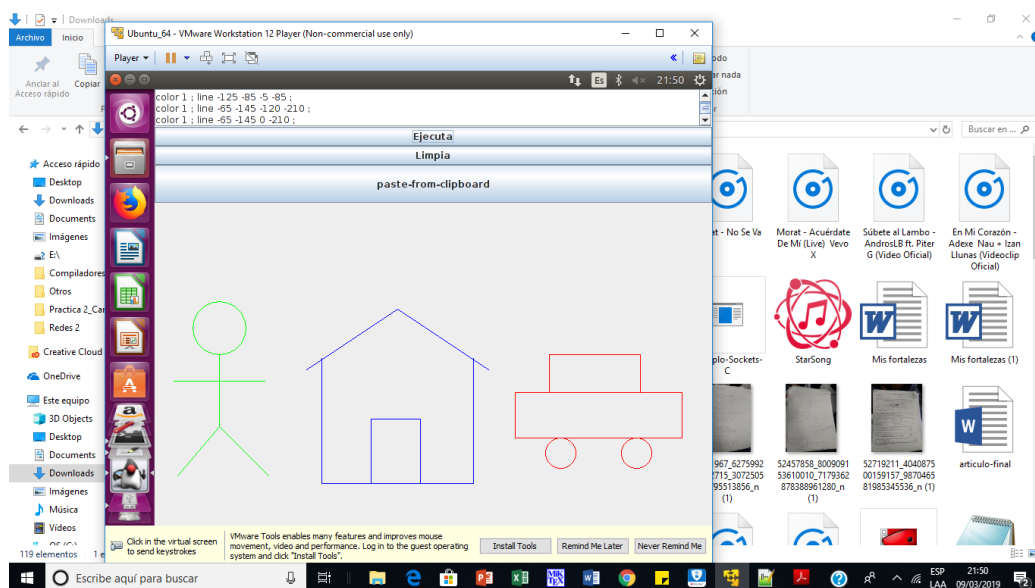
cionarlo, modificaremos código del archivo forma.y y Maquina.java La realización de ésta práctica consistió en la modificación de la gramática y del código en sí, del archivo Maquina.java que es el que contiene todo el código en Java.

Primero se modificaron los métodos: rectángulo, línea y círculo. Cabe mencionar que para cada uno de ellos se tuvieron que agregar variables, ya que anteriormente solamente recibían una, y como parte de la modificación, también tuvo que agregarse más funciones pop a la pila. Siempre teniendo cuidado de hacer los castings necesarios para poder mandar las variables a las funciones que dibujan las figuras (sus parámetros deben ser enteros y los valores de la pila son de tipo double).

```

100 void line() {
101     int x1, y1, x2, y2;
102     y2 = (int) ((Double) pila.pop()).doubleValue();
103     x2 = (int) ((Double) pila.pop()).doubleValue();
104     y1 = (int) ((Double) pila.pop()).doubleValue();
105     x1 = (int) ((Double) pila.pop()).doubleValue();
106     if (g!=null) {
107         (new Linea(x1, y1, x2, y2) ).dibuja(g);
108     }
109 }

```



Por otra parte, se tuvo que modificar el archivo forma.y que contiene toda la gramática de yacc. Específicamente, en este archivo lo que se hizo fue agregar más tokens (NUMBER's). Lo anterior se hizo para poder modificar la gramática ya que aceptaba cadenas diferentes a las que necesitábamos para realizar la práctica.

```

22 inst:  NUMBER  {
23         ((Algo)$$).obj).inst=maq.code("constpush");
24         maq.code(((Algo)$1.obj).simb);
25     }
26     | RECTANGULO NUMBER NUMBER NUMBER NUMBER { // x1 y1 x2 y2
27         maq.code("constpush");
28         maq.code(((Algo)$2.obj).simb);

```



```

29         maq.code("constpush");
30         maq.code(((Algo)$3.obj).simb);
31         maq.code("constpush");
32         maq.code(((Algo)$4.obj).simb);
33         maq.code("constpush");
34         maq.code(((Algo)$5.obj).simb);
35         maq.code("rectangulo"); }
36     | LINE NUMBER NUMBER NUMBER NUMBER { // x1 y1 x2 y2
37         maq.code("constpush");
38         maq.code(((Algo)$2.obj).simb);
39         maq.code("constpush");
40         maq.code(((Algo)$3.obj).simb);
41         maq.code("constpush");
42         maq.code(((Algo)$4.obj).simb);
43         maq.code("constpush");
44         maq.code(((Algo)$5.obj).simb);
45         maq.code("line");
46     }

```

2.4. Conclusiones

Gracias a esta práctica pudimos entender el cómo se utiliza YACC en otros lenguajes, como lo es Java. También se pudo ver la diferencia al modo de compilar y ejecutar, respecto a la práctica pasada donde se utilizó lenguaje C. Java nos permitió manejar de manera sencilla gráficos, mientras que YACC permitió checar si los comandos para dibujar cada figura eran correctos. Para esta práctica se necesitaba validar una cadena bien estructurada pues un error sintáctico en ella provocaba que no se dibujara nada en el lienzo.

Tabla de símbolos

3.1. Introducción

La tabla de símbolos, también llamada “tabla de nombres” o “tabla de identificadores” tiene dos funciones principales:

- Efectuar chequeos semánticos.
- Generación de código

La tabla almacena la información que en cada momento se necesita sobre las variables del programa, información tal como: nombre, tipo, dirección de localización, tamaño, etc. La gestión de la tabla de símbolos es muy importante, ya que consume gran parte del tiempo de compilación. De ahí que su eficiencia sea crítica. Aunque también sirve para guardar información referente a los tipos creados por el usuario, tipos enumerados y, en general, a cualquier identificador creado por el usuario.

Respecto a cada una de ellas podemos guardar:

- Almacenamiento del nombre. Se puede hacer con o sin límite. Si lo hacemos con límite, emplearemos una longitud fija para cada variable, lo cual aumenta la velocidad de creación, pero limita la longitud en unos casos, y desperdicia espacio en la mayoría. Otro método es habilitar la memoria que necesitemos en cada caso para guardar el nombre.
- El tipo también se almacena en la tabla.
- Dirección de memoria en que se guardará. Esta dirección es necesaria, porque las instrucciones que referencian a una variable deben saber dónde encontrar el valor de esa

variable en tiempo de ejecución, también cuando se trata de variables globales. En lenguajes que no permiten recursividad, las direcciones se van asignando secuencialmente a medida que se hacen las declaraciones. En lenguajes con estructuras de bloques, la dirección se da con respecto al comienzo del bloque de datos de ese bloque, (función o procedimiento) en concreto.

- El número de dimensiones de una variable array, o el de parámetros de una función o procedimiento. Junto con el tipo de cada uno de ellos es útil para el chequeo semántico. Aunque esta información puede extraerse de la estructura de tipos, para un control más eficiente, se puede indicar explícitamente.

También podemos guardar información de los números de línea en los que se ha usado un identificador, y de la línea en que se declaró.

La tabla de símbolos consta de una estructura llamada símbolo. Las operaciones que puede realizar son:

- **Crear:** Crea una tabla vacía.
- **Insertar:** Parte de una tabla de símbolo y de un nodo, lo que hace es añadir ese nodo a la cabeza de la tabla.
- **Buscar:** Busca el nodo que contiene el nombre que le paso por parámetro.
- **Imprimir:** Devuelve una lista con los valores que tiene los identificadores de usuario, es decir recorre la tabla de símbolos. Este procedimiento no es necesario pero se añade por claridad, y a efectos de resumen y depuración.

3.2. Objetivo

Agregar una tabla de símbolos para permitir nombres de variables de más de una letra y agregar a la gramática la producción para el operador de asignación. Usar una lista simplemente ligada (en lenguaje C o Java).

También si es posible agregar builtins. Por ejemplo, para números complejos agregar exponencial, seno, coseno y potencia.

3.3. Desarrollo

Esta práctica consiste en agregar a nuestro programa de la práctica 1, la calculadora de complejos, lo necesario para guardar números en variables (agregar una tabla de símbolos=, para esto utilizaremos el ejemplo de hoc3 que el profesor nos brindó.

Primeramente, en la sección de declaraciones de YACC se especificó la unión que se utilizara, la cual contiene un apuntador a complejo, un apuntador a la tabla de símbolos y un doble. Además se especificaron los nuevos tokens VAR, BTLIN e INDEF que usaran la parte sym de la unión y a los tokens complejos la parte de complejo de la unión.

```
14 %union{  
15     double num;
```

```

16     Vector * val;
17     Symbol * sym;
18 }
19
20 /* Declaración de YACC*/
21 %token <num> NUMBER
22 %token <sym> VAR INDEF

```

Posteriormente, cambiaremos la gramática; esta tiene que ser modificada para cumplir con los nuevos requerimientos pedidos. Dicho lo anterior, a la gramática le declaramos la regla para las variables y la acción para que estas se instalen en la tabla de símbolos. Además agregamos la regla para BLTIN que son las funciones definidas igualmente en la tabla de símbolos.

```

35 %%
36 list:
37     | list '\n'
38     | list asgn '\n'
39     | list expr '\n' { imprimeVector($2); }
40     | list number '\n' { printf("\t%.8g\n", $2); }
41     | list error '\n' { yyerror; }
42     ;
43
44 asgn: VAR '=' expr { $$ = $1->u.val = $3;
45                     $1->type = VAR; }
46     ;
47
48 expr: vector { $$ = $1; }
49     | VAR { if( $1->type == INDEF )
50             execerror("Variable no definida", $1->name);
51             $$ = $1->u.val;
52         }
53     | asgn
54     | expr '+' expr { $$ = sumaVector( $1, $3 ); }
55     | expr '-' expr { $$ = restaVector( $1, $3 ); }
56     | NUMBER '*' expr { $$ = escalarVector( $1, $3 ); }
57     | expr '*' NUMBER { $$ = escalarVector( $3, $1 ); }
58     | expr '#' expr { $$ = productoCruz( $1, $3 ); }
59     ;
60
61 number: NUMBER
62     | expr ':' expr { $$ = productoPunto( $1, $3 ); }
63     | '|' expr '|' { $$ = magnitudVector( $2 ); }
64     ;
65
66 vector: '[' NUMBER NUMBER NUMBER ']' { $$ = creaVector(3);
67                                         $$->vec[0] = $2;
68                                         $$->vec[1] = $3;
69                                         $$->vec[2] = $4;
70                                         }
71     ;
72 %%

```

El segmento de código que necesitó modificaciones fue YYLEX () para que pudiera instalar una variable en la tabla de símbolos, si es que esta aún no está.

```

99  int yylex(){
100     int c;
101     while ((c = getchar()) == ' ' || c == '\t')
102         ;
103     if (c == EOF)
104         return 0;
105
106     if (c == '.' || isdigit(c) ) {
107         ungetc(c, stdin);
108         scanf("%lf\n", &yylval.num);
109         return NUMBER;
110     }
111
112     if (isalpha(c)) {
113         Symbol * s;
114         char sbuf[200];
115         char * p = sbuf;
116         do {
117             *p++=c;
118         } while((c = getchar()) != EOF && isalnum(c));
119
120         ungetc(c, stdin);
121         *p = '\0';
122         if ((s = lookup(sbuf)) == (Symbol *)NULL)
123             s = install(sbuf, INDEF, NULL);
124         yylval.sym = s;
125
126         if (s->type == INDEF)
127             return VAR;
128         else{
129             //printf("func=(%s) tipo=(%d) \n", s->name, s->type);
130             return s->type;
131         }
132     }
133     return c;
134 }

```

Quizá la parte más importante en esta práctica fue la implementación de la tabla de símbolos, cuya estructura fue especificada en el archivo `vector_cal.h`. La estructura se conforma de un nombre de símbolo, tipo, una unión que especifica a un apuntador a complejo si es variable o la función de BTIN que regresa un apuntador a complejo, y un apuntador al siguiente elemento en la tabla de símbolos.

```

1  #include "vector_cal.h"
2
3  typedef struct Symbol{    /*Entrada de la tabla de símbolos*/
4      char * name;
5      short type;          /* VAR , BLTIN , UNDEF */
6
7      union{
8          Vector * val;    /* Si es VAR */
9      }u;
10
11     struct Symbol * next; /* Es para ligarse a otro */

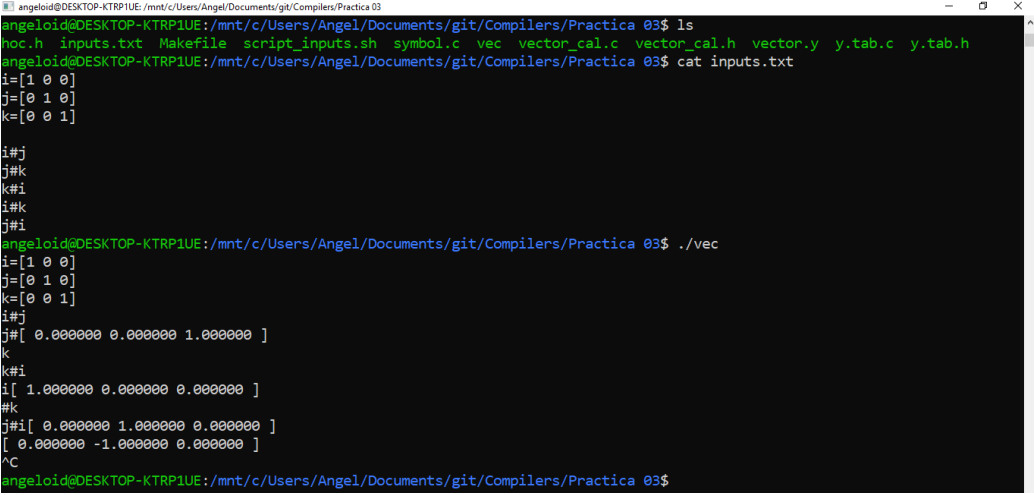
```

```

12 }Symbol;
13
14 Symbol * install(char * s, int t, Vector * d);
15 Symbol * lookup(char * s);

```

Por otra parte, también se modificó, para poder realizar las operaciones pedidas en la práctica, lo que son los archivos math.c e init.c, en el primero se definieron las operaciones, y en el segundo los builtins (palabras que al ser detectadas en tanto a su sintaxis en el programa ejecutado, ejecutan una función).



```

angeloid@DESKTOP-KTRP1UE: /mnt/c/Users/Angel/Documents/git/Compilers/Practica 03$ ls
hoc.h inputs.txt Makefile script_inputs.sh symbol.c vec vector_cal.c vector_cal.h vector.y y.tab.c y.tab.h
angeloid@DESKTOP-KTRP1UE: /mnt/c/Users/Angel/Documents/git/Compilers/Practica 03$ cat inputs.txt
i=[1 0 0]
j=[0 1 0]
k=[0 0 1]

i#j
j#k
k#i
i#k
j#i
angeloid@DESKTOP-KTRP1UE: /mnt/c/Users/Angel/Documents/git/Compilers/Practica 03$ ./vec
i=[1 0 0]
j=[0 1 0]
k=[0 0 1]
i#j
j#k 0.000000 0.000000 1.000000 ]
k
k#i
i[ 1.000000 0.000000 0.000000 ]
#k
j#i[ 0.000000 1.000000 0.000000 ]
[ 0.000000 -1.000000 0.000000 ]
^C
angeloid@DESKTOP-KTRP1UE: /mnt/c/Users/Angel/Documents/git/Compilers/Practica 03$

```

3.4. Conclusiones

El agregar una tabla de símbolos nos facilita el poder guardar las variables que el usuario requiera y operar con ellas sin importar el nombre que quiera asignarle a estas, además proporciona funciones que simplifican los cálculos. Para la tabla de símbolos se necesita una unión que contenga un apuntador a una estructura símbolo, un apuntador al tipo de los datos en este caso tipo vector pointer.

4

SECTION

Maquina de pila

4.1. Introduccion

La etapa inicial de un compilador construye una representacion intermedia del programa fuente a partir de la cual la etapa final genera el programa objeto. Una forma comun de representacion intermedia es el codigo para una maquina de pila abstracta. La division de un compilador en una etapa inicial y una etapa final facilita su modificacion para que funcione en una nueva maquina. En esta seccion se presenta una maquina de pila abstracta y se muestra como se puede generar su codigo. La maquina tiene memorias independientes para las instrucciones y los datos, y todas las operaciones aritmeticas se realizan con los valores en una pila. Las instrucciones son bastantes limitadas y estan comprendidas en tres clases: aritmetica entera, manipulacion de la pila y flujo de control.

4.2. Objetivo

Agregar una máquina de pila a la calculadora de vectores.

4.3. Desarrollo

Esta práctica consiste en agregar a nuestro programa de la práctica 3, una máquina de pila. Para el desarrollo de la práctica se escribió una especificación en YACC que evalúa expresiones aritméticas que involucran operaciones con vectores. Adicionalmente se agregó una máquina de pila para trabajar bajo el concepto de:

- Generación de código.

■ Ejecución de código

Para llevar a cabo la realización de la práctica tuvimos que definir el código que se iba a mandar a llamar para cada una de las reglas de la gramática, esto a través de la función `code()`. Por otro lado, la unión como la conocíamos tiene que ser modificada para poder recibir un nuevo apuntador a `Inst` que a su vez es un apuntador a instrucción.

De igual forma las reglas gramaticales ahora indican el código que se va a generar para cada expresión. Así para las operaciones aritméticas, se genera el código de cada operación con `code()`. Igualmente se genera el código para asignar una variable. Cuando se crea un vector se instala en la tabla de símbolos.

```

39 %%
40 list:
41     | list '\n'
42     | list asgn '\n' { code2(pop, STOP); return 1; }
43     | list expr '\n' { code2(print, STOP); return 1; }
44     | list escalar '\n' { code2(printd, STOP) return 1; }
45     | list error '\n' { yyerror; }
46     ;
47
48 asgn: VAR '=' expr { code3(varpush, (Inst)$1, assign); }
49     ;
50
51 expr: vector { code2(constpush, (Inst)$1); }
52     | VAR { code3(varpush, (Inst)$1, eval); }
53     | asgn
54     | expr '+' expr { code(add); }
55     | expr '-' expr { code(sub); }
56     | escalar '*' expr { code(escalar); }
57     | expr '*' escalar { code(escalar); }
58     | expr '#' expr { code(producto_cruz); }
59     ;
60
61 escalar: numero { code2(constpushd, (Inst)$1); }
62         | expr ':' expr { code(producto_punto); }
63         | '|' expr '|' { code(magnitud); }
64         ;
65
66 vector: '[' NUMBER NUMBER NUMBER ']' { Vector * vector1 = creaVector(3);
67                                         vector1->vec[0] = $2;
68                                         vector1->vec[1] = $3;
69                                         vector1->vec[2] = $4;
70                                         $$ = install("", VECT , vector1);
71                                         }
72     ;
73
74 numero: NUMBER { $$ = installd("", NUMB, $1); }
75
76 %%

```

La función principal, `main` tiene la ejecución de dicho código generado por la especificación de la gramática.


```

87 void main(int argc, char * argv[]) {
88     progname = argv[0];
89     setjmp(begin);
90     signal(SIGFPE, fpecatch);
91     for (initcode(); yyparse (); initcode())
92         execute(prog);
93 }

```

En la maquina (code.c), se define las funciones que se van a ejecutar. Se define la pila de tipo Datum con un tamaño de 256, esta pila es un arreglo que contendrá los vector y las definiciones de la tabla de símbolos, también se define un apuntador que servirá para indicar el siguiente elemento en la pila de la máquina y la RAM que son las instrucciones que se van a ejecutar como prog[NPROG] con un tamaño de 200, además se definen los apuntadores a este para indicar el siguiente lugar para la generación de código y el contador de programa.

```

1  #include "hoc.h"
2  #include "y.tab.h"
3  #define NSTACK 256
4  static Datum stack[NSTACK];      /* Pila */
5  static Datum *stackp;            /* Tope de la Pila*/
6  #define NPROG 2000
7  Inst prog[NPROG];               /* La máquina virtual - RAM: Se guardan las
   ↳ instrucciones */
8  Inst *progp;                   /* Siguiete lugar libre para la generación de código:
   ↳ Dice en donde se guarda nuestra proxima instruccion
   ↳ */
10 Inst *pc;                      /* Contador del programa durante la ejecución */

```

En la función push se mete un valor de tipo Datum a la pila, antes se checa que no esté llena la pila. En la función pop se obtiene el valor que este arriba de la pila, también se verifica que no se quiera sacar un valor cuando la pila ya está vacía. Posteriormente, aparecen las demás funciones, las algebraicas.

```

18 void push(d)
19     Datum d;
20 {     /* Se mete d en la pila*/
21     if( stackp >= &stack[NSTACK] )
22         execerror("stack overflow", (char *) 0);
23     *stackp++ = d;
24 }

```

4.4. Conclusiones

La pila de datos que ofrece Hoc 4, junto con la creación de código intermedio y ejecución (sus dos etapas) es otra forma de manejar la calculadora; una forma mucho más eficiente y elegante ya que es posible agilizar las operaciones e incluso entenderlas un poco mejor.

Los cambios como tal en esta práctica son de manera “interna” y al hacer las pruebas como tal quizás no se vea mucha diferencia con la práctica anterior, sin embargo, claramente los procesos que se llevan a cabo son diferentes al tomar en cuenta que en esta ocasión las funciones van de acuerdo a un orden postfijo que va guardándose en una pila de datos.

Decisiones if y ciclos while

5.1. Introduccion

Las sentencias de decisión o condición, son estructuras de control que realizan una pregunta la cual retorna verdadero o falso (evalúa una condición) y selecciona la siguiente instrucción a ejecutar dependiendo la respuesta o resultado.

En nuestros algoritmos, muchas veces tenemos que tomar una decisión en cuanto a que se debe ejecutar basándonos en una condición.

Los ciclos while son también una estructura cíclica, que nos permite ejecutar una o varias líneas de código de manera repetitiva sin necesidad de tener un valor inicial e incluso a veces sin siquiera conocer cuándo se va a dar el valor final que esperamos, los ciclos while, no dependen directamente de valores numéricos, sino de valores booleanos, es decir su ejecución depende del valor de una condición dada, verdadera o falso, nada más.

5.2. Objetivo

Agregar decisiones (if) y ciclos (while).

5.3. Desarrollo

Esta práctica consiste en agregar a nuestro programa de la práctica 4, la calculadora, los ciclos y sentencias de decisión, while e if, respectivamente.

Para el desarrollo de esta práctica se tomó en cuenta el mapa de memoria que se genera para las sentencias if y while. Esto fue esencial para poder realizar correctamente la adición de estas sentencias a la calculadora que hemos estado construyendo. A través del uso correcto de las

instrucciones STOP y STMT es posible realizar los saltos que implican los ciclos y las decisiones, lo que es esencial en los mapas de memoria del programa que se utilizaron en esta práctica.

En la gramática se define un tipo stmt que puede ser la función print, la cual imprime un vector, la función while o las condiciones if o if-else. En la declaración while se especifica la condición de la iteración y hasta donde termina la iteración y continúa la siguiente instrucción. Con la ayuda del mapa de memoria de while se sabe que STOP contiene el cuerpo y el final. Se genera el código. El mapa de memoria de if funciona de manera similar, guardando el cuerpo de la condición, el final y en el caso del if-else el segundo cuerpo si la condición no se cumple. En end se indica el final de la iteración o condición indicando el siguiente espacio para la instrucción.

```

43 %%
44 list:
45   | list '\n'
46   | list asgn '\n' { code2(pop, STOP); return 1; }
47   | list stmt '\n' { code(STOP); return 1; }
48   | list expr '\n' { code2(print, STOP); return 1; }
49   | list escalar '\n' { code2(printd, STOP) return 1; }
50   | list error '\n' { yyerror; }
51   ;
52
53 asgn: VAR '=' expr { $$ = $3; code3(varpush, (Inst)$1, assign); }
54   ;
55
56 stmt: expr { code(pop); }
57   | PRINT expr { code(print); $$ = $2; }
58   | while cond stmt end {
59       ($1)[1] = (Inst)$3; /* Cuerpo de la iteracion */
60       ($1)[2] = (Inst)$4; /* Termina si la condicion no se cumple */
61   }
62   | if cond stmt end { /* Proposicion if*/
63       ($1)[1] = (Inst)$3; /* Parte then */
64       ($1)[2] = (Inst)$4; /* Termina si la condicion no se cumple
65       ↪ */
66   }
67   | if cond stmt end ELSE stmt end { /* Proposicion if-else */
68       ($1)[1] = (Inst)$3; /* Parte then */
69       ($1)[2] = (Inst)$6 /* Parte else */
70       ($1)[3] = (Inst)$7; /* Termina si las condiciones no se
71       ↪ cumplen*/
72   }
73   | '{' stmtlist '}' { $$ = $2; }
74   ;
75
76 cond: '(' expr ')' { code(STOP); $$ = $2; }
77   ;
78
79 while: WHILE { $$ = code3(whilecode, STOP, STOP); }
80   ;
81
82 if: IF { $$ = code(ifcode);
83       code3(STOP, STOP, STOP);
84     }

```

```

83      ;
84
85  end: /* Nada */ { code(STOP); $$ = progp; }
86      ;
87
88  stmtlist: /* Nada */ { $$ = progp; }
89      | stmtlist '\n'
90      | stmtlist stmt
91      ;
92
93  expr: vector { code2(constpush, (Inst)$1); }
94      | VAR { code3(varpush, (Inst)$1, eval); }
95      | asgn
96      | BLTIN '(' expr ')' { $$ = $3; code2(bltin, (Inst)$1->u.ptr); }
97      | expr '+' expr { code(add); }
98      | expr '-' expr { code(sub); }
99      | escalar '*' expr { code(escalar); }
100     | expr '*' escalar { code(escalar); }
101     | expr '#' expr { code(producto_cruz); }
102     | expr GT expr { code(gt); }
103     | expr LT expr { code(lt); }
104     | expr GE expr { code(ge); }
105     | expr LE expr { code(le); }
106     | expr EQ expr { code(eq); }
107     | expr NE expr { code(ne); }
108     | expr OR expr { code(or); }
109     | expr AND expr { code(and); }
110     | NOT expr { $$ = $2; code(not); }
111     ;
112
113  escalar: numero { code2(constpushd, (Inst)$1); }
114      | expr ':' expr { code(producto_punto); }
115      | '|' expr '|' { code(magnitud); }
116      ;
117
118  vector: '[' NUMBER NUMBER NUMBER ']' { Vector * vector1 = creaVector(3);
119                                          vector1->vec[0] = $2;
120                                          vector1->vec[1] = $3;
121                                          vector1->vec[2] = $4;
122                                          $$ = install("", VECT , vector1);
123                                          }
124      ;
125
126  numero: NUMBER { $$ = installd("", NUMB, $1); }
127
128  %%

```

En la maquina se especifica el código para la ejecución del while e if. En whilecode primero se salva la posición donde empieza el while y después se ejecuta la condición indicando la posición de dicha condición en este caso, savepc+2. Posteriormente se saca el resultado de la pila y se empieza la iteración si es verdadera, ejecutando el cuerpo de la iteración que se encuentra en el primer STOP al que este momento apunta savepc, terminando la ejecución del cuerpo se vuelve a ejecutar la condición y se obtiene el resultado con pop de la pila y se somete al while de nuevo

si es cierta, en caso contrario se ejecuta la instrucción que se encuentra al termino del while y se guardó en el segundo STOP. Igualmente en ifcode se guarda la posición del if y se ejecuta la condición, se obtiene el resultado con pop de la pila y si es verdadera entra al if ejecutando el cuerpo del if guardada en el primer STOP, si no es cierta la condición se ejecuta la parte else que esta guarda en el segundo STOP.

```

55 void bltin(){    /*Evaluar un predefinido en el tope de la pila */
56     Datum d;
57     d = pop();
58     d.val = (*(Vector * (*)() )(*pc++))(d.val);
59     push(d);
60 }
61
62 /* Ciclo WHILE */
63 void whilecode(){
64     Datum d;
65     Inst * savepc = pc;    /* Cuerpo de la iteración */
66     execute(savepc + 2);    /* Condición */
67     d = pop();
68
69     while(d.val){
70         execute(* ( (Inst **)(savepc) ));    /* Cuerpo del ciclo*/
71         execute(savepc + 2);
72         d = pop();
73     }
74
75     pc = *((Inst **)(savepc + 1));    /*Vamos a la siguiente posicion*/
76 }
77
78 /* Condición IF */
79 void ifcode(){
80     Datum d;
81     Inst * savepc = pc;    /* Parte then */
82     execute(savepc + 3);    /*condicion*/
83     d = pop();
84     if(d.val)
85         execute(*((Inst **)(savepc)));
86     else if(*((Inst **)(savepc + 1)));    /*Parte del else*/
87         execute(*((Inst **)(savepc + 1)));
88
89     pc = *((Inst **)(savepc + 2));    /*Vamos a la siguiente posicion de la
90         pila*/
91 }
92
93 void eval( ){
94     Datum d;
95     d = pop();
96     if( d.sym->type == INDEF )
97         execerror("undefined variable",d.sym->name);
98
99     d.val = d.sym->u.val;
100    push(d);
101 }

```

Es importante tomar en cuenta el uso de funciones externas en hoc.h, y la adición de una nueva estructura de palabras “keywords”, en init.c, que serán las que nos permitirán reconocer si hacer un while o un if (o if-else) a partir de la sintaxis.

```
1  #include "hoc.h"
2  #include "y.tab.h"
3  #include <math.h>
4
5  static struct{
6      char * name;    /* Keywords */
7      int kval;
8  }keywords[] = {
9      "if", IF,
10     "else", ELSE,
11     "while", WHILE,
12     "print", PRINT,
13     0,0,
14 };
15
16 int init(){ /* Se instalan las constantes y predefinidos en la tabla */
17     int i;
18     Symbol * s;
19     for (i = 0; keywords[i].name; i++)
20         install(keywords[i].name, keywords[i].kval, NULL);
21 }
```

5.4. Conclusiones

El manejo correcto de las instrucciones stop, stmt y, por supuesto de los tokens desde el archivo y, nos permiten poder realizar correctamente los saltos y el uso de los ciclos que implica el uso de Hoc 5. La adición de nuevos tipos de sentencias a la calculadora fue bastante interesante ya que aprendimos a manejar el programa de manera interna, tomando en cuenta cómo funcionan internamente el while y el if.

6

SECTION

Lazo for

6.1. Introduccion

Los ciclos for son lo que se conoce como estructuras de control de flujo cíclicas o simplemente estructuras cíclicas, estos ciclos, como su nombre lo sugiere, nos permiten ejecutar una o varias líneas de código de forma iterativa, conociendo un valor específico inicial y otro valor final, además nos permiten determinar el tamaño del paso entre cada "giro.º iteración del ciclo.

En resumen, un ciclo for es una estructura de control iterativa, que nos permite ejecutar de manera repetitiva un bloque de instrucciones, conociendo previamente un valor de inicio, un tamaño de paso y un valor final para el ciclo.

6.2. Objetivo

Agregar ciclo for. Empezar por dibujar el mapa de memoria. Tener en cuenta que el código se genera en postfijo para dibujar dicho mapa de memoria.

6.3. Desarrollo

Primeramente, se especifica la sintaxis que seguirá el ciclo for al escribirse en la consola, además de definir qué será de tipo instrucción. La iteración del for está compuesta por una expresión que es la inicialización, un marcador que indica el inicio de la condición, otro marcador que indica el inicio de la expresión que es el incremento y el cuerpo de la función con el final del for.

```
109  stmt: exp
110  | PRINT exp
    _ $2;}
```

```
{code(pop);}
{code(print); $$ =
```

```

111 | while cond stmt end          {($1)[1] =
    ↳ (Inst)$3;                  ($1)[2] =
                                ↳ (Inst)$4;}
112
113
114 | if cond stmt end           {($1)[1] =
    ↳ (Inst)$3;                  ($1)[3] =
                                ↳ (Inst)$4;}
115
116
117 | if cond stmt end ELSE stmt end {($1)[1] =
    ↳ (Inst)$3;                  ($1)[2] =
                                ↳ (Inst)$6;
118                                ($1)[3] =
                                ↳ (Inst)$7;}
119
120
121 | for '(' exprn ';' exprn ';' exprn ')' stmt end {($1)[1] =
    ↳ (Inst)$5;                  ($1)[2] =
                                ↳ (Inst)$7;
122                                ($1)[3] =
                                ↳ (Inst)$9;
123                                ($1)[4] =
                                ↳ (Inst)$10;}
124
125 | '{' stmtlst '}'           {$$ = $2;}
126 ;
127
128 cond: '(' exp ')'          {code(STOP); $$ = $2;}
129 ;
130
131 while: WHILE                {$$ = code3(whilecode, STOP,
    ↳ STOP);}
132 ;
133
134 if: IF                       {$$ = code(ifcode);
                                code3(STOP, STOP, STOP);}
135
136 ;
137
138 end: /* NADA */             {code(STOP); $$ = prog;}
139 ;
140
141 stmtlst: /* NADA */         {$$ = prog;}
142 | stmtlst '\n'
143 | stmtlst stmt
144 ;
145
146 //PRÁCTICA 6
147 for: FOR                     {$$ = code(forcode); code3(STOP,
    ↳ STOP, STOP); code(STOP);}
148 ;
149

```



```

150     exprn: exp                                { $$ = $1; code(STOP); }
151     |   '{' stmtlst '}'                      { $$ = $2; }
152     ;

```

En la función forcode se ejecuta primero la expresión de inicialización y está solo se ejecutara una sola vez, posteriormente se ejecuta la condición y se saca el resultado de la pila, si es verdadero entra al while y se ejecuta el cuerpo del for, posteriormente se ejecuta la expresión incremento, y por último se ejecuta de nuevo la condición y si es cierta se vuelve a entrar al while así hasta que no se cumpla la condición, en caso de que no sea verdadera se ejecuta la siguiente instrucción fuera del for.

```

261 void forcode(){
262     Datum d;
263     Inst* savepc = pc;
264     execute(savepc + 4);
265     execute(*((Inst **)(savepc)));
266     //Se saca la instrucción
267     d = pop();
268     while(d.val){
269         execute(* ( (Inst **)(savepc + 2))); /* Cuerpo del ciclo*/
270         execute(* ( (Inst **)(savepc + 1))); // Último campo
271         pop();
272         execute(*((Inst **)(savepc)));      /* CONDICION */
273         d = pop();
274     }
275     pc = *((Inst **)(savepc + 3)); /*Vamos a la siguiente posicion*/
276 }

```

6.4. Conclusiones

El manejo correcto de las instrucciones stop, stmt y, por supuesto de los tokens desde el archivo y, nos permiten poder realizar correctamente los saltos y el uso del ciclo for. Es bastante importante tomar en cuenta que la generación del código de dicho ciclo se hizo con base en el mapa de memoria, que fue el primer paso a tomar para realizar la presente práctica. Saber construir el mapa generado por el for, implica poder implementar el ciclo de manera correcta. Gracias a esta práctica, entendí con más claridad cómo funcionan los stop y su importancia, así como me ayudo a ver cómo generar ciclos de otra manera desde la calculadora.

Funciones y procedimientos

7.1. Introduccion

Una función es un conjunto de líneas de código que realizan una tarea específica y puede retornar un valor. Las funciones pueden tomar parámetros que modifiquen su funcionamiento. Las funciones son utilizadas para descomponer grandes problemas en tareas simples y para implementar operaciones que son comúnmente utilizadas durante un programa y de esta manera reducir la cantidad de código. Cuando una función es invocada se le pasa el control a la misma, una vez que esta finalizó con su tarea el control es devuelto al punto desde el cual la función fue llamada.

7.2. Objetivo

En esta séptima práctica se han añadido funciones y procedimientos. Para realizar esto se han añadido más funciones a code.c, y se han añadido más símbolos gramaticales.

7.3. Desarrollo

Nuevamente se ha modificado la gramática, se han añadido más símbolos gramaticales, se han añadido más elementos a la unión, esto se muestra a continuación

```
45  %%  
46  list:  
47      | list '\n'  
48      | list defn '\n'  
49      | list asgn '\n'{code2(pop,STOP); return 1;}  
50      | list stmt '\n' {code(STOP); return 1;}
```

```

51 | list exp '\n'{code2(print,STOP); return 1;}
52 | list error '\n' {yyerror;}
53 ;
54
55 asgn: VAR '=' exp { $$ = $3; code3(varpush,(Inst)$1,assign);}
56 | ARG '=' exp {defonly("$");code2(argassign,(Inst)$1); $$ = $3;}
57 ;
58
59 stmt: exp {code(pop);}
60 | RETURN {defonly("return");code(procret);}
61 | RETURN exp
62 | {defonly("return"); $$=$2;code(funcrret);}
63
64 | PRINT prlist { $$ = $2;}
65 | while cond stmt end
66 { ($1)[1] = (Inst)$3; /* cuerpo de la
67 iteración*/
68 ($1)[2] = (Inst)$4; /* terminar si la
69 condición no se cumple*/
70
71 | if cond stmt end { /* proposición if que no emplea
72 else*/
73 ($1)[1] = (Inst)$3; /* parte then */
74 ($1)[3] = (Inst)$4; } /* terminar si la
75 condición no se cumple */
76
77 | if cond stmt end ELSE stmt end { /* proposición if ocn parte
78 else*/
79 ($1)[1] = (Inst)$3; /*parte then*/
80 ($1)[2] = (Inst)$6; /*paret else*/
81 ($1)[3] = (Inst)$7; } /*terminar si la
82 condición no se cumple*/
83
84 | '{' stmtlist '}' { $$ = $2;}
85 ;
86
87 cond: '(' exp ')' {code(STOP); $$ = $2;}
88 ;
89
90 while: WHILE { $$ = code3(whilecode,STOP,STOP);}
91 ;
92
93 if: IF { $$ = code(ifcode);
94 | code3(STOP,STOP,STOP);}
95 ;
96
97 end: /* nada */ {code(STOP); $$ = prog; }
98 ;
99
100 stmtlist: /* nada */ { $$ = prog;}
101 | stmtlist '\n'
102 | stmtlist stmt

```

```

96 ;
97
98 exp: vector { $$ = code2(constpush, (Inst)$1); }
99 | VAR { $$ = code3(varpush, (Inst)$1, eval); }
100 | ARG { defonly("$"); $$ =
    ↪ code2(arg, (Inst)$1); }
101 | asgn
102 | FUNCTION begin '(' arglist ')' { $$ = $2;
    ↪ code3(call, (Inst)$1, (Inst)$4); }
103 | READ '(' VAR ')' { $$ = code2(varread, (Inst)$3); }
104 | BLTIN '(' exp ')' { code2(bltin, (Inst)$1->u.ptr); }
105 | exp '+' exp { code(add); }
106 | exp '-' exp { code(sub); }
107 | exp '.' exp { code(punto); }
108 | exp '*' NUMBER { code(mul); }
109 | NUMBER '*' exp { code(mul); }
110 | exp '#' exp { code(cruz); }
111 | exp GT exp { code(gt); }
112 | exp GE exp { code(ge); }
113 | exp LT exp { code(lt); }
114 | exp LE exp { code(le); }
115 | exp EQ exp { code(eq); }
116 | exp NE exp { code(ne); }
117 | exp AND exp { code(and); }
118 | exp OR exp { code(or); }
119 | NOT exp { $$ = $2; code(not); }
120 | PROCEDURE begin '(' arglist ')' { $$ = $2;
    ↪ code3(call, (Inst)$1, (Inst)$4); }
121 ;
122
123 begin: /*nada */ { $$ = progp; }
124 ;
125
126 prlist: exp { code(prexp); }
127 | STRING { $$ = code2(prstr, (Inst)$1); }
128 | prlist ',' exp { code(prexp); }
129 | prlist ',' STRING { code2(prstr, (Inst)$3); }
130 ;
131
132 defn: FUNC procname { $2->type=FUNCTION; indef =1; }
133 | '(' ')' stmt
    ↪ { code(procret); define($2); indef=0; }
134 | PROC procname { $2->type = PROCEDURE; indef = 1; }
135 | '(' ')' stmt { code(procret); define($2);
    ↪ indef=0; }
136
137 ;
138
139 procname: VAR
140 | FUNCTION
141 | PROCEDURE
142 ;
143

```

```

144     arglist: /*nada*/                {$$=0;}
145         | exp                        {$$ = 1;}
146         | arglist ',' exp            {$$ = $1 + 1;}
147     ;
148
149     vector: '[' NUMBER NUMBER NUMBER ']' {Vector *v = creaVector(3);
150                                         v->vec[0] = $2;
151                                         v->vec[1] = $3;
152                                         v->vec[2] = $4;
153                                         $$ = install("",VEC,v);}
154     ;
155

```

También se ha añadido más código a code.c en donde realizamos todas las acciones que sean necesarias para ejecutar las funciones.

```

306 void define(Symbol *sp)
307 {
308     sp->u.defn = (Inst)progbase; /* principio de cdigo */
309     progbase = progp;           /* el siguiente cdigo comienza aqu */
310 }
311
312 void call()
313 {
314     Symbol *sp = (Symbol *)pc[0]; /*entrada en la tabla de smbolos*/
315     if (fp++ >= &frame[NFRAME - 1])
316         execerror(sp->name, "call nested too deeply");
317     fp->sp = sp;
318     fp->nargs = (int)pc[1];
319     fp->retpc = pc + 2;
320     fp->argn = stackp - 1; /*ltimo argumento*/
321     execute(sp->u.defn);
322     returning = 0;
323 }
324
325 void ret()
326 {
327     int i;
328     for (i = 0; i < fp->nargs; i++)
329         pop(); /*saca argumentos*/
330     pc = (Inst *)fp->retpc;
331     --fp;
332     returning = 1;
333 }
334
335 void funcrct()
336 {
337     Datum d;
338     if (fp->sp->type == PROCEDURE)
339         execerror(fp->sp->name, "(proc) returns value");
340     d = pop(); /* preservar el valor de regreso a a funcion*/
341     ret();
342     push(d);
343 }

```

```

344
345 void procret()
346 {
347     if (fp->sp->type == FUNCTION)
348         execerror(fp->sp->name, "(func) return no value");
349     ret();
350 }
351
352 Vector **getarg()
353 {
354     int nargs = (int)*pc++;
355     if (nargs > fp->nargs)
356         execerror(fp->sp->name, "not enough arguments");
357     return &fp->argn[nargs - fp->nargs].val;
358 }
359
360 void arg()
361 { /*meter el aergumento en la pila*/
362     Datum d;
363     d.val = *getarg();
364     push(d);
365 }
366
367 void argassign()
368 {
369     Datum d;
370     d = pop();
371     push(d);
372     *getarg() = d.val;
373 }
374
375 void prstr()
376 {
377     printf("%s", (char *)*pc++);
378 }
379
380 void varread()
381 {
382     Datum d;
383     extern FILE *fin;
384     Symbol *var = (Symbol *)*pc++;
385     Again:
386     switch (fscanf(fin, "%lf", &var->u.val))
387     {
388     case EOF:
389         if (moreinput())
390             goto Again;
391         d.val = var->u.val = NULL;
392         break;
393     case 0:
394         execerror("non-number read into", var->name);
395         break;
396     default:
397         d.val = NULL;

```

```

398     break;
399 }
400 var->type = VAR;
401 push(d);
402 }

```

Como se deben generar marcos de función se ha creado una estructura la cual contendrá la información de cada función. También se ha creado la pila de llamadas en la cual iremos apilando los marcos de función.

```

14 Inst *progbase = prog; /* empieza el subprograma actual*/
15 int returning;        /* 1 si ve proposición return */
16
17 typedef struct Frame
18 {
19     Symbol *sp; /*entrada en la tabla de smbolos*/
20     Inst *retpc; /*donde continuar despues de regresar*/
21     Datum *argn; /*n-simo argumento en la pila*/
22     int nargs; /*numero de argumentos*/
23 } Frame;
24
25 #define NFRAME 100
26 Frame frame[NFRAME];
27 Frame *fp;

```

7.4. Pruebas

Archivo fuente

```

1 func operaciones() {
2     print $1 + $2
3     print $1 - $2
4     print $1 . $2
5     print $1 # $2
6 }
7
8 func fibo() {
9     count = [0 0 0]
10    x = [1 0 0]
11    y = [1 0 0]
12    z = [0 0 0]
13    while (count < 1) {
14        z = x
15        x = y
16        y = z + y
17        count = count + [1 0 0]
18    }
19    return z
20 }
21
22 func fact() {
23     n = 1
24     if (n < [1 0 0]) {

```

```

25         return [1 0 0]
26     }
27     return n . fact(n - [1 0 0])
28 }
29
30 res = fibo([4 0 0])
31 print res
32 res = fact([3 0 0])
33 print res

```

Resultado

A screenshot of a terminal window. The title bar at the top reads "Title: Default". The terminal prompt is "1: fish /home/angelos/Documents/glt/own/Compilers/Práctica 07". The user has entered the command "~ /D/g/o/C/Práctica 07 on master ./vect < source.v". The output shows a 6x10 matrix of floating-point numbers. The first row is [4.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The second row is [0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The third row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The fourth row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The fifth row is [0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The sixth row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The seventh row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The eighth row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The ninth row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The tenth row is [3.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The eleventh row is [3.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The twelfth row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The thirteenth row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The fourteenth row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The fifteenth row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The sixteenth row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The seventeenth row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The eighteenth row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The nineteenth row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The twentieth row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The twenty-first row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The twenty-second row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The twenty-third row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The twenty-fourth row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The twenty-fifth row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The twenty-sixth row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The twenty-seventh row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The twenty-eighth row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The twenty-ninth row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The thirtieth row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The thirty-first row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The thirty-second row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The thirty-third row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The thirty-fourth row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The thirty-fifth row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The thirty-sixth row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The thirty-seventh row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The thirty-eighth row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The thirty-ninth row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The fortieth row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The forty-first row is [1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000]. The forty-second row is [1.000000 0.000000 0.000