



INSTITUTO POLITÉCNICO NACIONAL  
ESCUELA SUPERIOR DE CÓMPUTO

Práctica número 7:  
Calculadora para vectores  
(Con funciones y procedimientos)

9 de enero de 2021

Grupo: 3CM7

*Nombre del alumno:*  
Ramos Mesas Edgar Alain

*Número de boleta:*  
2013090243

MATERIA: COMPILADORES

## 1. Introducción

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.

## 2. Desarrollo

Esta práctica consiste en agregar a la calculadora para vectores, las modificaciones necesarias para poder ejecutar funciones y procedimientos.

Primeramente agregamos los nuevos simbolos gramaticales necesarios para poder declarar y ejecutar funciones, además se realizaron modificaciones a la unión.

```
1  /*
2  Ramos Mesas Edgar Alain
3      3CM7
4  */
5
6  %{
7  #include <stdio.h>
8  #include <math.h>
9  #include "vector_cal.h"
10 #define code2(c1,c2)      code(c1); code(c2)
11 #define code3(c1,c2,c3)   code(c1); code(c2); code(c3)
12
13 void warning(char *s, char *t);
14 int yyerror (char *s);
15 void execerror(char *s, char *t);
16 void run();
17 int indef;
18
19 %}
20
21
22 %union{
23     Symbol *sym; /*apuntador de la tabla de símbolos*/
24     Inst *inst; /* instrucción de máquina*/
25     double val;
26     Vector *vec;
27     int narg; /*Número de argumentos*/
28 }
29
```

```

30
31 %token <sym> VAR BLTIN INDEF VEC NUMERO WHILE IF ELSE PRINT STRING
32 %token <sym> FUNCTION PROCEDURE RETURN FUNC PROC READ
33 %token <narg> ARG
34 %token <val> NUMBER
35 //Agregue la siguiente linea
36 %type <inst> stmt asgn exp stmtlist cond while if end prlist begin
37 %type <sym> vector
38 %type <sym> procname
39 %type <narg> arglist
40 //type <vec> exp asgn
41 %left '+' '-'
42 %left '*'
43 %left '#' '.'
44 //Agregue la siguiente linea
45 %left OR
46 %left AND
47 %left GT GE LT LE EQ NE
48 %left NOT
49 //Sección de reglas de yacc
50 %%
51 list:
52     | list '\n'
53     | list defn '\n'
54     | list asgn '\n' {code2(pop,STOP); return 1;}
55     | list stmt '\n' {code(STOP); return 1;}
56     | list exp '\n' {code2(print,STOP); return 1;}
57     | list error '\n' {yyerror;}
58     ;
59
60 asgn: VAR '=' exp {$$ = $3; code3(varpush,(Inst)$1,assign);}
61     | ARG '=' exp {defonly("$");code2(argassign,(Inst)$1); $$ = $3;}
62     ;
63
64 stmt: exp {code(pop);}
65     | RETURN {defonly("return");code(procret);}
66     | RETURN exp {defonly("return");$$=$2;code(funcrret);}
67
68     | PRINT prlist { $$ = $2;}
69     | while cond stmt end {
70         ($1)[1] = (Inst)$3; /* cuerpo de la iteración*/
71         ($1)[2] = (Inst)$4; /* terminar si la condición
72         ↪ no se cumple*/
73
74     | if cond stmt end { /* proposición if que no emplea else*/
75
76         ($1)[1] = (Inst)$3; /* parte then */

```

```

76      ($1)[3] = (Inst)$4; } /* terminar si la condición
      ↪ no se cumple */
77
78      | if cond stmt end ELSE stmt end { /* proposición if ocn parte else*/
79      ($1)[1] = (Inst)$3; /*parte then*/
80      ($1)[2] = (Inst)$6; /*paret else*/
81      ($1)[3] = (Inst)$7; } /*terminar si la condición
      ↪ no se cumple*/
82      | '{' stmtlist '}'
      { $$ = $2;}
83
84      ;
85
86      cond: '(' exp ')'
      {code(STOP); $$ = $2;}
87      ;
88
89      while: WHILE
      { $$ = code3(whilecode,STOP,STOP);}
90      ;
91
92      if: IF
      { $$ = code(ifcode); code3(STOP,STOP,STOP);}
93      ;
94
95      end: /* nada */
      {code(STOP); $$ = prog; }
96      ;
97
98      stmtlist: /* nada */
      {$$ = prog;}
99      | stmtlist '\n'
100     | stmtlist stmt
101     ;
102
103     exp: vector
      {$$ = code2(constpush, (Inst)$1);}
104     | VAR
      {$$ = code3(varpush,(Inst)$1,eval);}
105     | ARG
      {defonly("$"); $$ = code2(arg,(Inst)$1);}
106     | asgn
107     | FUNCTION begin '(' arglist ')' { $$ = $2;
      ↪ code3(call,(Inst)$1,(Inst)$4);}
108     | READ '(' VAR ')' {$$= code2(varread,(Inst)$3);}
109     | BLTIN '(' exp ')' {code2(bltin,(Inst)$1->u.ptr);}
110     | exp '+' exp {code(add);}
111     | exp '-' exp {code(sub);}
112     | exp '.' exp {code(punto);}
113     | exp '*' NUMBER {code(mul);}
114     | NUMBER '*' exp {code(mul);}
115     | exp '#' exp {code(cruz);}
116     | exp GT exp {code(gt);}
117     | exp GE exp {code(ge);}
118     | exp LT exp {code(lt);}
119     | exp LE exp {code(le);}

```

```

120      |exp EQ exp          {code(eq);}
121      |exp NE exp          {code(ne);}
122      |exp AND exp         {code(and);}
123      |exp OR exp          {code(or);}
124      |NOT exp             {$$ = $2; code(not);}
125      |PROCEDURE begin '(' arglist ')' { $$ = $2; code3(call,(Inst)$1,(Inst)$4);}
126      ;
127
128      begin:/*nada */      {$$ = prog;}
129      ;
130
131      prlist: exp           {code(preexpr);}
132      | STRING             {$$ = code2(prstr,(Inst)$1);}
133      | prlist ',' exp      {code(preexpr);}
134      | prlist ',' STRING   {code2(prstr,(Inst)$3);}
135      ;
136
137      defn:  FUNC procname   {$2->type=FUNCTION; indef =1;}
138            '(' ')' stmt {code(procret);define($2);indef=0;}
139            |PROC procname   {$2->type = PROCEDURE; indef = 1;}

```

Posteriormente fue necesario modificar el archivo llamado code.c, a el se ha añadido más código en donde realizamos todas las acciones que sean necesarias para ejecutar las funciones y procedimientos.

```

309  // CÓDIGO AÑADIDO EN LA PRÁCTICA 7
310  void define(Symbol *sp)
311  {
312      sp->u.defn = (Inst)progbase; /* principio de cdigo */
313      progbase = prog;           /* el siguiente cdigo comienza aqu */
314  }
315
316  void call()
317  {
318      Symbol *sp = (Symbol *)pc[0]; /*entrada en la tabla de smbolos*/
319      if (fp++ >= &frame[NFRAME - 1])
320          execerror(sp->name, "call nested too deeply");
321      fp->sp = sp;
322      fp->nargs = (int)pc[1];
323      fp->retpc = pc + 2;
324      fp->argn = stackp - 1; /*ltimo argumento*/
325      execute(sp->u.defn);
326      returning = 0;
327  }
328
329  void ret()
330  {
331      int i;

```

```
332     for (i = 0; i < fp->nargs; i++)
333         pop(); /*saca argumentos*/
334     pc = (Inst *)fp->retpc;
335     --fp;
336     returning = 1;
337 }
338
339 void funcret()
340 {
341     Datum d;
342     if (fp->sp->type == PROCEDURE)
343         execerror(fp->sp->name, "(proc) returns value");
344     d = pop(); /* preservar el valor de regreso a a funcion*/
345     ret();
346     push(d);
347 }
348
349 void procret()
350 {
351     if (fp->sp->type == FUNCTION)
352         execerror(fp->sp->name, "(func) return no value");
353     ret();
354 }
355
356 Vector **getarg()
357 {
358     int nargs = (int)*pc++;
359     if (nargs > fp->nargs)
360         execerror(fp->sp->name, "not enough arguments");
361     return &fp->argn[nargs - fp->nargs].val;
362 }
363
364 void arg()
365 { /*meter el aargumento en la pila*/
366     Datum d;
367     d.val = *getarg();
368     push(d);
369 }
370
371 void argassign()
372 {
373     Datum d;
374     d = pop();
375     push(d);
376     *getarg() = d.val;
377 }
378
379 void prstr()
380 {
```

```

381     printf("%s", (char *)*pc++);
382 }
383
384 void varread()
385 {
386     Datum d;
387     extern FILE *fin;
388     Symbol *var = (Symbol *)*pc++;
389     Again:
390     switch (fscanf(fin, "%lf", &var->u.val))
391     {
392     case EOF:
393         if (moreinput())
394             goto Again;
395         d.val = var->u.val = NULL;
396         break;
397     case 0:
398         execerror("non-number read into", var->name);
399         break;
400     default:
401         d.val = NULL;
402         break;
403     }
404     var->type = VAR;
405     push(d);
406 }

```

Debido a que se deben generar marcos de función fue necesario crear una estructura que cual contendrá la información de cada función. También se ha creado la pila de llamadas en la cual iremos apilando los marcos de función.

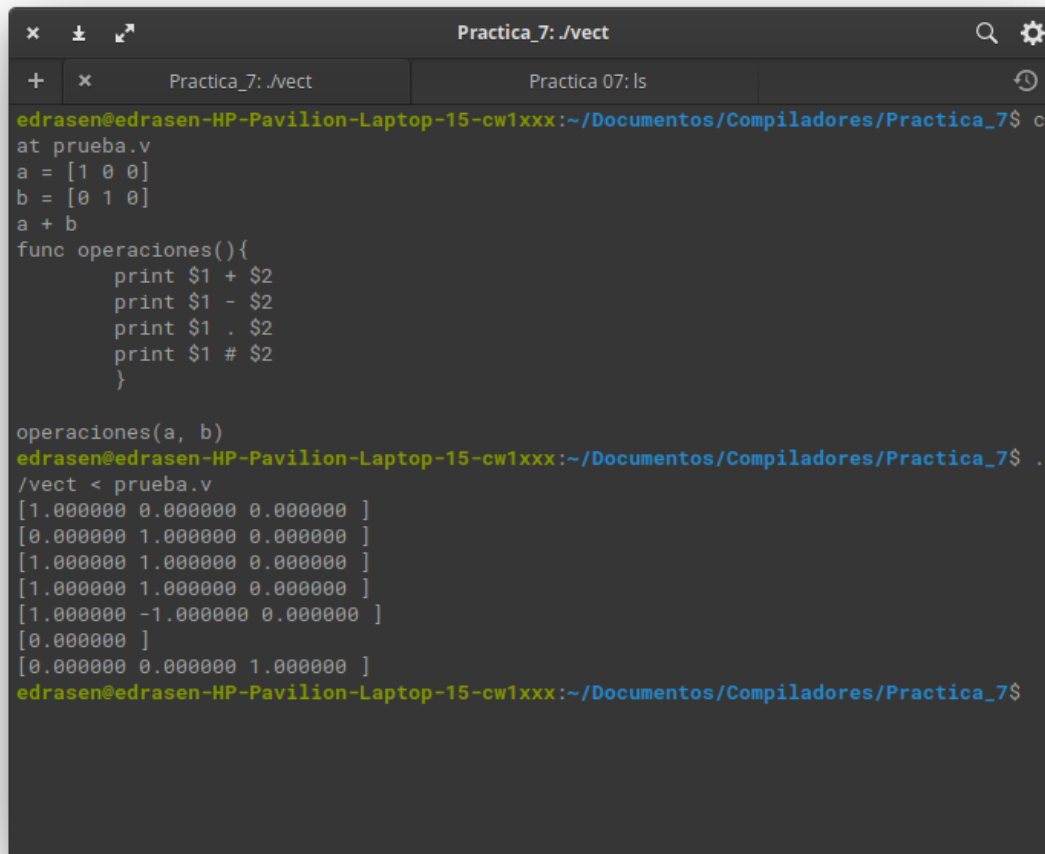
```

18 Inst *progbase = prog; /* empieza el subprograma actual*/
19 int returning;        /* 1 si ve proposición return */
20
21 typedef struct Frame
22 {
23     Symbol *sp; /*entrada en la tabla de smbolos*/
24     Inst *retpc; /*donde continuar despues de regresar*/
25     Datum *argn; /*n-simo argumento en la pila*/
26     int nargs; /*numero de argumentos*/
27 } Frame;
28
29 #define NFRAME 100
30 Frame frame[NFRAME];
31 Frame *fp;

```

Finalmente, tras realizar todas las modificaciones mencionadas se compilo todo desde un archivo Makefile. Todas las instrucciones son cargadas desde un archivo llamado prueba.v cuyo código se muestra a continuación.

```
1 a = [1 0 0]
2 b = [0 1 0]
3 a + b
4 func operaciones(){
5     print $1 + $2
6     print $1 - $2
7     print $1 . $2
8     print $1 # $2
9 }
10
11 operaciones(a, b)
```



```
Practica_7: ./vect
edrasen@edrasen-HP-Pavilion-Laptop-15-cw1xxx:~/Documentos/Compiladores/Practica_7$ c
at prueba.v
a = [1 0 0]
b = [0 1 0]
a + b
func operaciones(){
    print $1 + $2
    print $1 - $2
    print $1 . $2
    print $1 # $2
}

operaciones(a, b)
edrasen@edrasen-HP-Pavilion-Laptop-15-cw1xxx:~/Documentos/Compiladores/Practica_7$ .
/vect < prueba.v
[1.000000 0.000000 0.000000 ]
[0.000000 1.000000 0.000000 ]
[1.000000 1.000000 0.000000 ]
[1.000000 1.000000 0.000000 ]
[1.000000 -1.000000 0.000000 ]
[0.000000 ]
[0.000000 0.000000 1.000000 ]
edrasen@edrasen-HP-Pavilion-Laptop-15-cw1xxx:~/Documentos/Compiladores/Practica_7$
```

Comenzamos declarando dos vectores “a” y “b”, dichos vectores serán sumados para comprobar la ejecución de las operaciones aritméticas. Posteriormente declaramos la función operaciones en la cual ejecutamos operaciones como la suma, la resta, el producto punto y el producto cruz, imprimiendo en cada caso, el resultado de la operación.



### 3. Conclusiones

Las modificaciones realizadas en esta práctica han proporcionado a la calculadora para vectores un funcionamiento mucho más parecido a un mini lenguaje de programación basado en vectores, lo cual resulta realmente útil pues permite realizar de manera automática algunas operaciones o funciones con simplemente declararlas y llamarlas durante el proceso de ejecución del programa. Se aprendió también a hacer el manejo de los marcos de función para que de esa manera el programa supiera en que momentos llamar y ejecutar una función y en que momento terminar con la misma y volver al punto en el que quedó previo a la llamada. En resumen, esta práctica ha permitido implementar algo muy parecido a lo que es un lenguaje de programación.