



INSTITUTO POLITÉCNICO NACIONAL  
ESCUELA SUPERIOR DE CÓMPUTO

Práctica número 4:  
Calculadora para vectores  
(Con maquina de pila)

3 de enero de 2021

Grupo: 3CM7

*Nombre del alumno:*  
Ramos Mesas Edgar Alain

*Número de boleta:*  
2013090243

MATERIA: COMPILADORES

## 1. Introducción

La etapa inicial de un compilador construye una representación intermedia del programa fuente a partir de la cual la etapa final genera el programa objeto. Una forma comun de representacion intermedia es el código para una maquina de pila abstracta. La división de un compilador en una etapa inicial y una etapa final facilita su modificación para que funcione en una nueva maquina. En esta práctica se presenta una maquina de pila abstracta y se muestra cmo 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 bastante limitadas y estan comprendidas en tres clases: aritmetica entera, manipulacion de la pila y flujo de control.

## 2. Desarrollo

Esta práctica consiste en agregar a nuestro programa de la práctica 3, la calculadora para vectores, una maquina de pila. Para poder añadir la máquina virtual de pila es necesario crear un arreglo el cual nos servirá para simular dicho elemento. Para el desarrollo de esta práctica se escribió una especificación en YACC que evalua expresiones aritméticas que involucran operaciones con vectores. Adicionalmente y dado que empleamos una máquina de pila, se utilizan dos conceptos importantes:

- Generación de código.
- Ejecución de código.

Para poder realizar la práctica fue necesario definir el código que se llama para cada una de las reglas de la gramática, esto a través de la función `code()`. Además las reglas acciones gramaticales fueron cambiadas por macros que generan el codigo que se va a generar para cada expresión. Así, para cada operación aritmética, se genera el código de dicha operación con `code()`. De igual modo, y paritando de lo implementado en la práctica 3, se genera código para asignar una variable ocasionando así la creación un nuevo vector que se instala en la tabla de simbolos.

```
36  %%
37  list:
38      | list '\n'
39      | list asgn '\n' { code2(pop, STOP); return 1; }
40      | list expr '\n' { code2(print, STOP); return 1;}
41      | list escalar '\n' { code2(printf, STOP); return 1;}
42      | list error '\n' { yyerrok; }
43      ;
44  asgn:  VAR '=' expr { code3(varpush, (Inst)$1, assign);}
45      ;
46  expr:  vector {code2(constpush, (Inst)$1);}
47      | VAR {code3(varpush, (Inst)$1, eval);}
```

```

48 | asgn
49 |   expr '+' expr    { code(add); }
50 |   expr '-' expr    { code(sub); }
51 |   escalar '*' expr  { code(escalar); }
52 |   expr '*' escalar {code(escalar); }
53 |   expr '#' expr    { code(productoc); }
54 ;
55
56 escalar: numero {code2(constpushd, (Inst)$1);}
57 |   expr ':' expr    { code(productop); }
58 |   '|' expr '|'     { code(magnitud); }
59 ;
60
61 vector: '[' NUMBER NUMBER NUMBER ']' {Vector *vector1= creaVector(3);
   ↪   vector1->vec[0] = $2; vector1->vec[1] = $3; vector1->vec[2] = $4; $$ =
   ↪   install("", VECT , vector1);}
62 ;
63 numero: NUMBER { $$ = installd("", NUMB,$1); }
64
65 %%

```

En la maquina (archivo code.c), se definen 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]; /* la pila */
5  static Datum *stackp;      /* siguiente lugar libre en la pila */
6  #define NPROG 2000
7  Inst prog[NPROG]; /* la máquina */
8  Inst *progp;      /* siguiente lugar libre para la generación de código */
9  Inst *pc;         /* contador de programa durante la ejecución */

```

Tal como se mencionaba al principio, al tratarse de una pila es necesario implementar las operaciones push y pop. 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)      /* meter d en la pila */
19 Datum d;
20 {

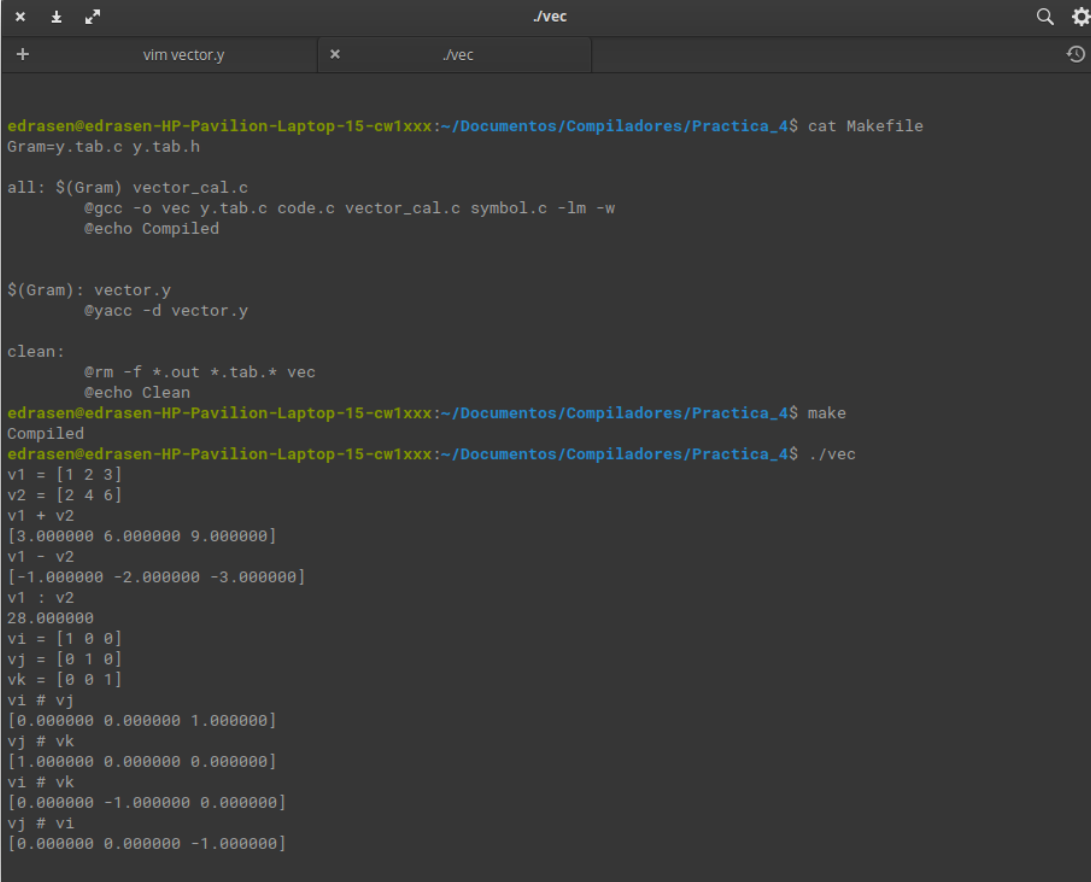
```

```
21     if (stackp >= &stack[NSTACK])
22         execerror("stack overflow", (char *) 0);
23     *stackp++ = d;
24 }
25
26 Datum pop( )          /* sacar y retornar de la pila el elemento del tope */
27 {
28     if (stackp <= stack)
29         execerror("stack underflow", (char *) 0);
30     return *--stackp;
31 }
32
33
34 void constpush( )     /* meter una constante a la pila */
35 {
36     Datum d;
37     d.val = ((Symbol *)*pc++)->u.val;
38     push(d);
39 }
40
41 void constpushd( )    /* meter una constante a la pila */
42 {
43     Datum d;
44     d.num = ((Symbol *)*pc++)->u.num;
45     push(d);
46 }
47
48 void varpush( )       /* meter una variable a la pila */
49 {
50     Datum d;
51     d.sym = (Symbol *)(*pc++);
52     push(d);
53 }
54
55 void eval( )          /* evaluar una variable en la pila */
56 {
57     Datum d;
58     d = pop();
59     if (d.sym->type == INDEF)
60         execerror("undefined variable", d.sym->name);
61     d.val = d.sym->u.val;
62     push(d);
63 }
64
65 void add( )           /* sumar los dos elementos superiores de la pila */
66 {
67     Datum d1, d2;
68     d2 = pop();
69     d1 = pop();
```

```
70     d1.val=  sumaVector(d1.val, d2.val);
71     push(d1);
72 }
73
74 void sub()
75 {
76     Datum d1, d2;
77     d2 = pop();
78     d1 = pop();
79     d1.val=  restaVector(d1.val, d2.val);
80     push(d1);
81 }
82
83 void escalar()
84 {
85     Datum d1, d2;
86     d2 = pop();
87     d1 = pop();
88     d1.val = escalarVector(d1.num, d2.val);
89     push(d1);
90 }
91
92
93 void productop( )
94 {
95     Datum d1, d2;
96     double d3;
97     d2 = pop();
98
99     d1 = pop();
100
101     d3 = productoPuntoVector(d1.val, d2.val);
102     push((Datum)d3);
103 }
104
105 void productoc( )
106 {
107     Datum d1, d2;
108     d2 = pop();
109
110     d1 = pop();
111
112     d1.val = productoCruzVector(d1.val, d2.val);
113     push(d1);
114 }
115
116 void magnitud( )
117 {
118     Datum d1;
```

```
119
120     d1 = pop();
121
122     d1.num = magnitudVector(d1.val);
123     push(d1);
124 }
125
126 void assign( )           /* asignar el valor superior al siguiente valor */
127 {
128     Datum d1, d2;
129     d1 = pop();
130     d2 = pop();
131     if (d1.sym->type != VAR && d1.sym->type != INDEF)
132         execerror("assignment to non-variable", d1.sym->name);
133     d1.sym->u.val = d2.val;
134     d1.sym->type = VAR;
135     push(d2);
136 }
137
138 void print( ) /* sacar el valor superior de la pila e imprimirlo */
139 {
140     Datum d;
141     d = pop();
142     imprimeVector(d.val);
143 }
144 void printd( ) /* sacar el valor superior de la pila e imprimirlo */
145 {
146     Datum d;
147     d = pop();
148     printf("%lf", d.num);
149 }
```

Finalmente, tras realizar todas las modificaciones mencionadas se compilo todo desde un archivo Makefile y se ejecutó el programa obteniendo los siguientes resultados:



```
edrasen@edrasen-HP-Pavilion-Laptop-15-cw1xxx:~/Documentos/Compiladores/Practica_4$ cat Makefile
Gram=y.tab.c y.tab.h

all: $(Gram) vector_cal.c
    @gcc -o vec y.tab.c code.c vector_cal.c symbol.c -lm -w
    @echo Compiled

$(Gram): vector.y
    @yacc -d vector.y

clean:
    @rm -f *.out *.tab.* vec
    @echo Clean
edrasen@edrasen-HP-Pavilion-Laptop-15-cw1xxx:~/Documentos/Compiladores/Practica_4$ make
Compiled
edrasen@edrasen-HP-Pavilion-Laptop-15-cw1xxx:~/Documentos/Compiladores/Practica_4$ ./vec
v1 = [1 2 3]
v2 = [2 4 6]
v1 + v2
[3.000000 6.000000 9.000000]
v1 - v2
[-1.000000 -2.000000 -3.000000]
v1 : v2
20.000000
v1 = [1 0 0]
vj = [0 1 0]
vk = [0 0 1]
v1 # vj
[0.000000 0.000000 1.000000]
vj # vk
[1.000000 0.000000 0.000000]
v1 # vk
[0.000000 -1.000000 0.000000]
vj # v1
[0.000000 0.000000 -1.000000]
```

### 3. 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.