INSTITUTO POLITECNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO (ESCOM)

COMPILADORES

NOMBRE DEL ALUMNO:

- SANTOS MÉNDEZ ULISES JESÚS

PRÁCTICA:

- CALCULADORA DE VECTORES

NÚMERO DE PRÁCTICA: 1

OPCIÓN 3:

- MOSAICO DE IMÁGENES

FECHA DE ENTREGA:

- 27/03/2023

GRUPO:

- 3CM14

Calculadora de vectores

Introducción:

El objetivo principal de la práctica fue aplicar los conocimientos vistos en la clase sobre cómo se compone un programa de YACC además de comprender que realiza cada una de las funciones del programa, se busca llevarlos a la práctica partiendo de la instalación de LEX & YACC en Ubuntu.

Lex & Yacc

Para que un programa reciba entradas ya sea de forma interactiva o en un entorno por lotes, debe proporcionar otro programa o una rutina para recibir la entrada. La entrada complicada requiere código adicional para dividir la entrada en partes que significan algo más para el programa.

Lex y Yacc son herramientas de generación de analizadores léxicos y sintácticos utilizadas para la creación de compiladores y otros programas que requieren análisis de lenguaje.

Lex es un generador de analizadores léxicos, lo que significa que se utiliza para analizar y dividir el código fuente en "tokens" o unidades léxicas que se utilizan como entrada para un analizador sintáctico. Los tokens son los componentes básicos del lenguaje, como palabras clave, identificadores, números y símbolos. Lex toma una descripción de los patrones de los tokens en el código fuente y genera un programa en C que realiza la tarea de reconocimiento de tokens.

Por otro lado, Yacc es un generador de analizadores sintácticos que toma una descripción de la gramática de un lenguaje y genera un programa en C que realiza el análisis sintáctico del código fuente. El analizador sintáctico utiliza la información proporcionada por el analizador léxico para verificar si la estructura del programa está de acuerdo con la gramática del lenguaje.

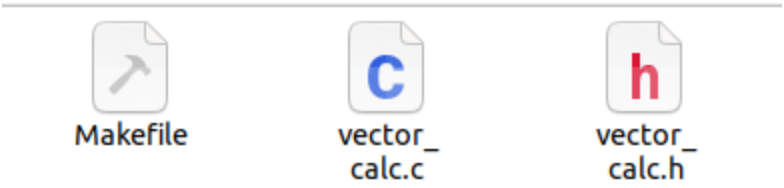
Juntos, Lex y Yacc proporcionan una forma eficiente y automatizada de generar compiladores y otros programas que requieren análisis de lenguaje.

Una vez que se conoce lo que es Lex & Yacc se busca resolver una problemática planteada que es:

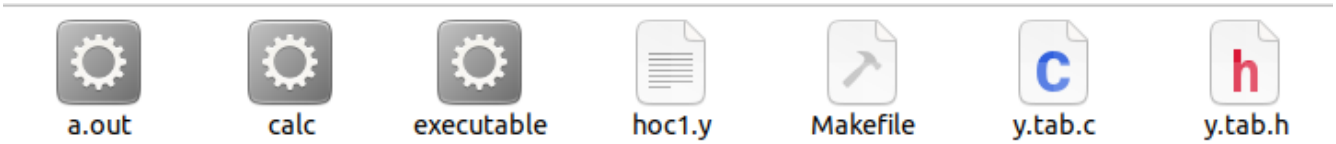
“Suponga que cuenta con el código del producto punto, el producto punto cruz, la multiplicación por un escalar, la suma, la resta y la magnitud. Escribir una especificación de Yacc para evaluar expresiones qiue involucren operaciones con vectores.”

Desarrollo

En la carpeta que nos fue compartida por el maestro, se realizaron las modificaciones necesarias para poder implementar las funciones necesarias para resolver las operaciones entre vectores y escalares.



También se realizaron cambios en el hoc1 que nos proporcionó el maestro, de ahí se tomo la estructura básica de una calculadora, al editarse se busco agregar los no terminales y producciones necesarias para que sea posible evaluar las expresiones.



La primera etapa de esta práctica consistía en compilar y ejecutar el hoc1 con las indicaciones que nos dieron en la sesión de laboratorio:

```
Compilar con yacc *.y
Luego gcc y.tab.c y ejecuto el .exe
Ejecutar el ejecutable
correr el hoc1 que es una calculadora
./nombre
```

Ahora se comienzan realizando los cambios en el archivo vector_calc.y agregando la cabecera vector_calc.h que es en donde se encuentran definidas todas las funciones vectoriales y enteras para poder ejecutar las operaciones entre vectores:

```
%{

#include "vector_calc.h"
void yyerror (char *s);
int yylex ();
void warning(char *s, char *t);
%}
```

Después se agrego una unión que será necesaria para definir los no terminales

```
%union{

double num;
Vector *vect;
}
%token<num> NUMBER
%type<vect> vector
%type<vect> expVectorial
%type<num> expEscalar
```

En este fragmento de código se expresa como que NUMBER ocupa la parte num de la unión que es un double, vector ocupa la parte vect de la unión que es un Vector, expVectorial ocupa la parte vect que es Vector y expEscalar ocupa la parte num de la unión que es un double.

```
%%
list:
| list '\n'
| list expVectorial '\n' {imprimeVector($2); }
| list expEscalar '\n' {printf("\033[0;36m%f\n\033[0;0m", $2); }
;
expVectorial:vector { $$ = $1; }
| expVectorial '+' expVectorial { $$ = sumaVector($1,$3);}
| expVectorial '-' expVectorial { $$ = restaVector($1,$3);}
| expVectorial 'x' expVectorial { $$ = cruzVector($1,$3);}
| expEscalar '*' expVectorial { $$ = multiplicaEscalarVector($1,$3);}
| expVectorial '*' expEscalar { $$ = multiplicaEscalarVector($3,$1);}
| '(' expVectorial ')' { $$ = $2;}
;
expEscalar: NUMBER {$$ = $1; }
| '|' expVectorial '|' {$$ = magnitudVector($2);}
| '|' expEscalar '|' {$$ = $2; }
| expVectorial '*' expVectorial {$$ = puntoVector($1,$3);}
| '(' expEscalar ')' {$$ = $2;}
;
vector: '['listnum']' {$$ = creaVectorEntrada(); }
;
listnum:
| NUMBER listnum {aux=push($1,aux);}
;
%%
```

En el fragmento de código se realizan las producciones necesarias para que puedan efectuarse las operaciones donde se mandan a llamar a las operaciones dependiendo de las producciones que se deseen al ingresar los caracteres a la consola, donde a cada producción se le asocia una acción gramatical que es lo que se encuentra entre { } , cada una de estas llamadas a función reciben parámetros en $1 y $3.

En el archivo vector_calc.h se definieron las funciones a llamar con todo y sus cabeceras que serán útiles para realizar la operación como math.h

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
```

En el siguiente fragmento de código resulta más visible las funciones que se implementaron para realizar las operaciones y con su respectivo tipo de retorno, donde en su mayoría es de origen vectorial.

```
Vector *creaVector(int n);
void imprimeVector(Vector *a);
int vectoresIguales(Vector *a, Vector *b);
Vector *sumaVector(Vector *a, Vector *b);
Vector *restaVector(Vector *a, Vector *b);
Vector *cruzVector(Vector *a, Vector *b);
double puntoVector(Vector *a, Vector *b);
Vector *multiplicaEscalarVector(double num, Vector *a);
double magnitudVector(Vector *a);
```

Se procede a inspeccionar el analizador léxico que se nos fue dado:

```
int yylex (){
int c;

while ((c = getchar ()) == ' ' || c == '\t')
; //enunciado nulo
if (c == EOF)
return 0;
if (c == '.' || isdigit (c)) {//NUM
ungetc (c, stdin);
scanf ("%lf", &yylval.num);//lexema
return NUMBER;//tipo token
}
if(c == '\n')
lineno++;
return c; //+, -, *, x, [, ]
}
```

El analizador yylex() es similar al que se estuvo vinedo en lase para el HOC1 en donde se obtienen puros números de tipo double.

Por último también es importante analizar la Pila que fue inicializada en el método init() la cual nos va a servir para que en la acción gramatical de listnum se puedan meter en la pila todos los números ingresados y después cuando se ejecute el método crearVectorEntrada() de la acción gramatical de vector, se puedan meter todos los números de la pila en un Vector* y se puedan usar todas las funciones de vector_calc.h.

```
typedef struct Stack{
double dato;
```

```
struct Stack *sig;
}*Pila;

Pila empty();

Pila push(double e, Pila p);

int isEmpty(Pila p);

double top(Pila p);

Pila pop(Pila p);

int size(Pila p);
```

Compilación y ejecución

```
ulisespc04@ulisessm04:~/Descargas/pracvec$ yacc vector_calc.y
ulisespc04@ulisessm04:~/Descargas/pracvec$ gcc y.tab.c vector_calc.c Pila.c -o calculadora -lm
ulisespc04@ulisessm04:~/Descargas/pracvec$ ./calculadora
```

Ejemplo propuesto:

```
[1 2 3] + [2 4 6]
[ 3.000000 6.000000 9.000000 ]
[1 2 3 4 5] + [2 4 6 8 10]
[ 3.000000 6.000000 9.000000 12.000000 15.000000 ]
```

Ejemplos faltantes con resta, producto punto, producto cruz, multiplicación de un escalar por un vector y la magnitud de un vector

```
[5 10 15 20] - [8 16 24 32]
[ -3.000000 -6.000000 -9.000000 -12.000000 ]
[3 7 6 9] x [7 14 21 0]
 Error: Producto cruz solo posible con R3
[3 6 7] x [0 1 30]
[ 173.000000 -90.000000 3.000000 ]
[6 12 19 23] * [9 18 44 2]
1152.000000
| [2 5 6 9] |
12.083046
```

Conclusión:

En conclusión es importante conocer el uso que se le puede dar a Lex & Yacc para generar distintas operaciones que van más allá que las matemáticas, se le puede dar diversos usos como el de analizar el lenguaje que ingresas así como el de ordenar datos por medio del A.A.S