INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

COMPILADORES

3CM7

PRÁCTICA 01

Alumno:
Díaz Medina Jesús Kaimorts

Profesor:
Tecla Parra Roberto

Boleta:
2016350140

YACC BÁSICO

Calculadora para Vectores

10 de septiembre 2018

Objetivo.

El alumno comprenda el uso básico de YACC, así el cómo establecer las reglas y aclaraciones gramaticales para poder expresar una gramática que desarrolle una calculadora para vectores.

Introducción.

YACC: *Yet Another Compiler Compiler*

Llamado genéricamente así debido a que sus siglas provienen del inglés *Yet Another Compiler Compiler,* un programa que parte del archivo escrito en el lenguaje específico **Lenguaje de Descripción Gramática** y genera el código en C, C++ o Pacal. Permite un lenguaje propio para automatizar tareas repetitivas que, de otra manera, tendrían que hacerse manualmente, dilapidando mucho tiempo y subutilizando las posibilidades de las computadoras actuales.

YACC provee una herramienta general para analizar estructuralmente una entrada.
El usuario de *YACC* prepara una especificación que incluye:

- Un conjunto de reglas que **describen los elementos de entrada.**
- Un código a ser invocado cuando una regla es conocida.
- Una o más **rutinas para examinar la entrada**.

Luego, *YACC* convierte la especificación en una función en C que examina la entrada. Esta función, *un parser*, trabaja mediante la invocación de un **analizador léxico** que extrae *token*s de la entrada. Los *tokens* son **comparados** con las reglas de construcción de la entrada, llamadas ***reglas gramaticales.*** Cuando una de las reglas es reconocida, el código provisto por el usuario, para esa regla (una acción), es invocado. **Las acciones son fragmentos de código en C**, que pueden retornar valores y usar los valores retornados por otras acciones.

Tanto *el analizador léxico como el sintáctico pueden ser escritos en cualquier lenguaje de programación.* A pesar de la habilidad de tales lenguajes de propósito general, como C, *LEX* y *YACC* son más flexibles y mucho menos complejos de usar.

LEX genera el código C para un analizador léxico, y *YACC* genera el código para un *parser*. Tanto *LEX* como *YACC* toman como entrada un archivo de especificaciones que es típicamente más corto que un programa hecho a medida y más fácil de leer y entender. Por convención, la extensión del archivo de especificaciones para *LEX* es ***.l*** y para *YACC* es ***.y.*** La salida de *LEX* y *YACC* es código fuente C.

LEX crea una rutina llamada ***yylex*** en un archivo llamado ***yylex.c***. *YACC* crea una rutina llamada ***yyparse*** en un archivo llamado ***y_tab.c***.

Estas rutinas son combinadas con código fuente C provisto por el usuario, el cual se ubica típicamente en un archivo separada, pero puede ser ubicado en el archivo de especificaciones de *YACC*. El código provisto por el usuario consiste en una **rutina main** que llamada a ***yyparse*** que en su momento llamada a ***yylex***. Todas estas rutinas deben ser compiladas y, en la mayoría de los casos, las librerías *LEX* y *YACC* deben ser cargadas en tiempo de compilación. Estas librerías contienen un número de rutinas de soporte que son requeridas, si no son provistas por el usuario.

El siguiente diagrama permite observar los pasos en el desarrollo de un compilador usando *LEX* y *YACC*.

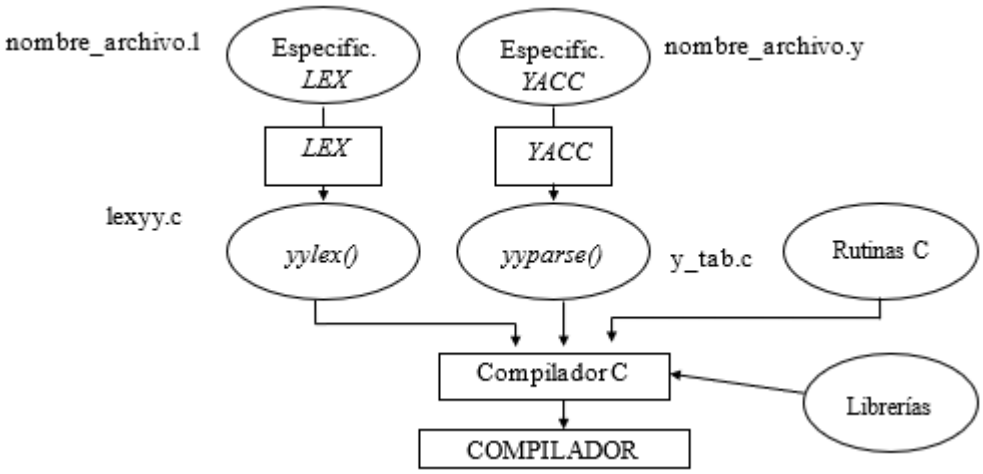
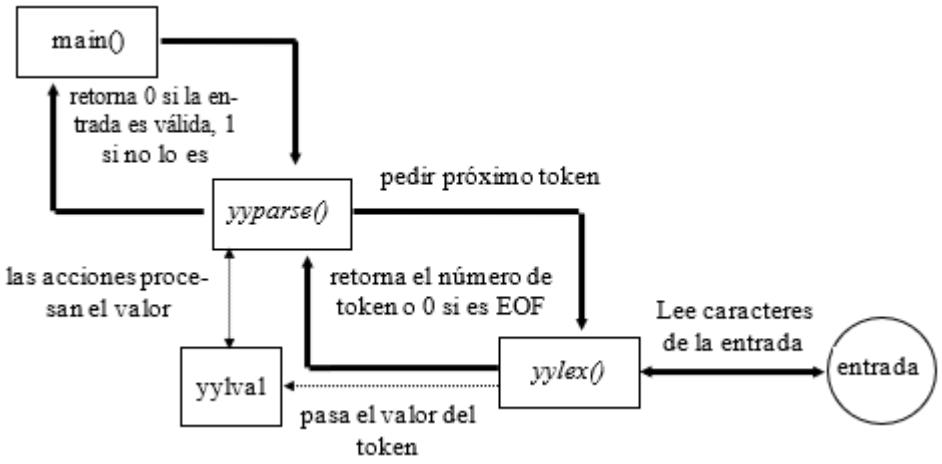


Figura 1. Proceso de desarrollo de un compilador usando LEX y YACC.

Interacción entre las rutinas Léxicas y de Parsing.

Una rutina main invoca yyparse para evaluar si la entrada es válida o no. yyparse invoca a una rutina llamada yylex cada vez que necesita un token (terminal).(yylex puede ser generado manualmente o generado por LEX) Esta rutina léxica lee la entrada, y por cada token que reconoce, retorna el número de token al parser. El léxico puede también pasar el valor del token usando la variable externa yylval. Las rutinas de acción del parse, asi como la rutina provista por el usuario pueden usar ese valor.

Las rutinas de acción pueden llamar a otras funciones que pueden estar ubicadas en la sección de código del usuario o en otros archivos de código fuente.



Desarrollo.

Escribir el código de soporte en lenguaje C o Java.

Calculadora para Vectores.

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 que involucren operaciones con vectores.

Entrada.

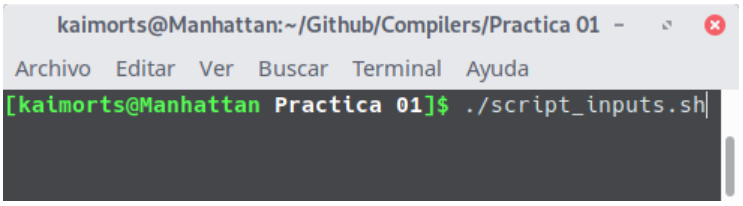
- Se creó un script con el nombre **inputs.txt** el cual contiene las entradas pertinentes con algunas de las operaciones implementadas, tales como: suma, resta, producto punto y producto cruz, y posteriormente se ejecuta en consola.

```
[ 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 ]
```

inputs.txt

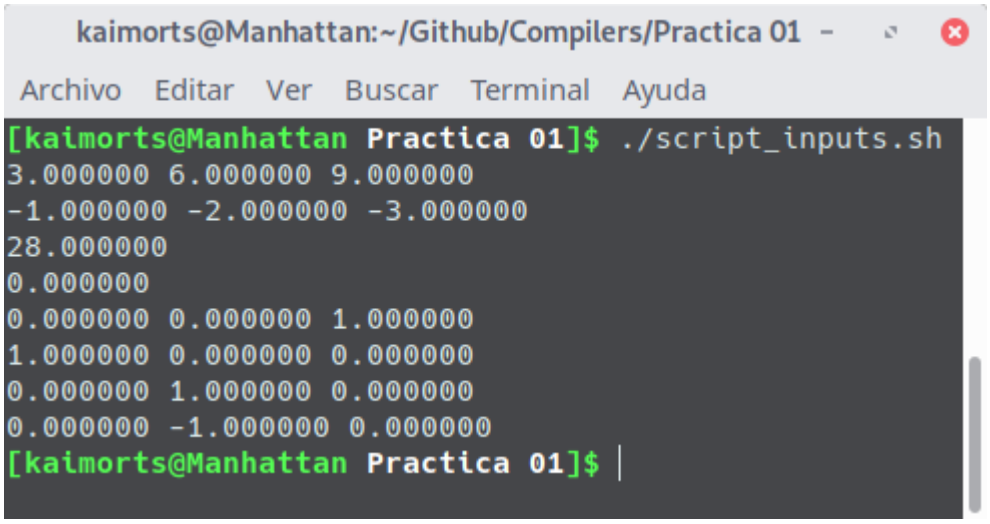
script_inputs.sh

```
#!/bin/bash
./vec < inputs.txt
```



Salida.

- La salida de las respectivas operaciones realizadas se muestra a continuación.



Gramática.

- Asimismo, se implementó las declaraciones y la sección de reglas gramaticales, utilizando como base la calculadora del **HOC1.** Dichas implementaciones se muestran debajo.

Declaraciones

```
%union{
  double val;
  Vector *vec;
}
```

```
/*Declaración de YACC*/
%token <val> NUMBER
%type <vec> exp
%type <vec> vector
%type <val> number
```

```
/*Asociatividad*/
%left '+' '-'
%left '*'
%left 'x' 'o'
```

Sección de reglas

```
%%
input: /*Cadena vacia*/
    |input list
    ;

list: '\n'
    | exp '\n' { imprimeVector($1); }
    | number '\n' {printf("%lf\n", $1);}
    ;

exp:  vector
    | exp '+' exp {$$ = sumaVector($1,$3); }
    | exp '-' exp {$$ = restaVector($1,$3); }
    |NUMBER '*' exp {$$ = escalarVector($1, $3);}
    | exp '*' NUMBER {$$ = escalarVector($3,$1);}
    | exp 'x' exp {$$ = productoCruz($1, $3); }
    ;
```

```
number: NUMBER
     | vector 'o' vector {$$ = productoPunto($1,$3);}
     | '|' vector '|' {$$ = magnitudVector($2); }
     ;

vector : '[' NUMBER NUMBER NUMBER ']' {Vector *v = creaVector(3);
                                       v->vec[0] = $2;
                                       v->vec[1] = $3;
                                       v->vec[2] = $4;
                                       $$ = v;
                                      }
%%
```

Operaciones con vectores.

- Librería

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

struct vector {
      char name;
      int n;
      double *vec;
};
typedef struct vector Vector;
Vector *creaVector(int n);
void imprimeVector(Vector *a);
Vector *copiaVector(Vector *a);
Vector *sumaVector(Vector *a, Vector *b);
Vector *restaVector(Vector *a, Vector *b);
Vector *escalarVector(double escalar, Vector *a);
double productoPunto(Vector *a, Vector *b);
Vector *productoCruz(Vector *a, Vector *b);
double magnitudVector(Vector *a);
```

Suma	Resta

`Vector *sumaVector(Vector *a, Vector *b){`
`  Vector *c;`
`  int i;`
`  c=creaVector(a->n);`
`  for(i=0; i< a->n;i++)`
`  c->vec[i]=a->vec[i]+b->vec[i];`
`  return c;`
`}`
	`Vector *restaVector(Vector *a, Vector *b){`
	`  Vector *c;`
	`  int i;`
	`  c=creaVector(a->n);`
	`  for(i=0; i< a->n;i++)`
	`  c->vec[i]=a->vec[i]-b->vec[i];`
	`  return c;`
	`}`

- Multiplicación por un escalar

```
Vector *escalarVector(double escalar, Vector *a){
   Vector *c;
   int i;
   c = creaVector(a->n);
   for(i=0; i < a->n ; i++)
       c->vec[i] = escalar * a->vec[i];

   return c;
}
```

- Magnitud de Vector

```
double magnitudVector(Vector *a){
   double m = 0.0;
   int i=0;
   for( ; i < a->n ; i++)
      m += (a->vec[i] * a->vec[i]);

   return sqrt(m);
}
```

- Producto Cruz

```
Vector *productoCruz(Vector *a, Vector *b){
   Vector *c;
   c = creaVector(a->n);

   if( a->n == 2){  /*Vector con 2 componentes*/
       c->vec[0] = a->vec[0] * b->vec[1];
       c->vec[1] = -(a->vec[1] * b->vec[0]);
   }else if( a->n == 3){ /*Vector con 3 componentes*/
             c->vec[0] = a->vec[1] * b->vec[2] - a->vec[2] * b->vec[1];
         //c->vec[1] = -(a->vec[0] * b->vec[2] - a->vec[2] * b->vec[0]);
            c->vec[1] = a->vec[2] * b->vec[0] - a->vec[0] * b->vec[2];
            c->vec[2] = a->vec[0] * b->vec[1] - a->vec[1] * b->vec[0];
   }

   return c;
}
```

- Producto Punto

```
double productoPunto(Vector *a, Vector *b){
   double c = 0.0;
   int i ;
   for( i=0; i< a->n ; i++)
      c += a->vec[i] * b->vec[i];

   return c;
}
```

Conclusión.

El alumno comprenda el uso básico de YACC, así el cómo establecer las reglas y aclaraciones gramaticales para poder expresar una gramática que desarrolle una calculadora para vectores.