INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO

Nombre: Cruz López Adrián

Grupo: 3CM15

Asignatura: Compiladores

Profesor: Roberto Tecla Parra

Actividad: Práctica 1 “Calculadora de Vectores”

Fecha: 09/10/2021

INTRODUCCIÓN

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 genera el código para el analizador sintáctico en el Lenguaje de programación C.

Puesto que el analizador sintáctico generado por YACC requiere un analizador léxico, se utiliza a menudo juntamente con un generador de analizador léxico, en la mayoría de los casos lex o Flex, alternativa del software libre. El estándar de IEEE POSIX P1003.2 define la funcionalidad y los requisitos a Lex y YACC.

LEX

Es un programa para generar analizadores léxicos. Lex se utiliza comúnmente con el programa YACC que se utiliza para generar análisis sintáctico. Lex es el analizador léxico estándar en los sistemas Unix, y se incluye en el estándar de POSIX. Lex toma como entrada una especificación de analizador léxico y devuelve como salida el código fuente implementando el analizador léxico en C.

Aunque tradicionalmente se trata de software propietario, existen versiones libres de lex basadas en el código original de AT&T en sistemas como OpenSolaris y Plan 9 de los laboratorios Bell. Otra versión popular de software libre de lex es Flex.

DESARROLLO

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.

Empezamos definiendo la gramática necesaria para realizar todas las operaciones mencionadas utilizando vectores, por lo cual la gramática en el formato de YACC es la siguiente: (*archivo calculadoraVect.y*):

```
%%
   inicio: /* NADA */
        | inicio list;
        ;
   list: '\n'
        | exp '\n' {imprimeVector($1);}
        | num '\n' {printf("%lf\n", $1);}
        ;
   exp: vector
        | exp '+' exp    {$$ = suma($1, $3);}            //Caso SUMA
        | exp '-' exp    {$$ = resta($1, $3);}           //Caso RESTA
        | exp '*' NUMBER {$$ = escalarporVector($3, $1);}//Caso MULT por ESCALAR 1
        | NUMBER '*' exp {$$ = escalarporVector($1, $3);}//Caso MULT por ESCALAR 2
        | exp 'x' exp     {$$ = productoCruz($1, $3);}   //Caso PROD CRUZ
        ;
    num: NUMBER
        | vector '.' vector {$$ = productoPunto($1, $3);}//Caso PROD PUNTO
        | '|' vector '|' {$$ = magnitud($2);}            //Caso MAGNITUD
        ;
    vector: '[' NUMBER NUMBER NUMBER ']' {Vector *v = crearVector(3);
                                          v -> vec[0] = $2;
                                          v -> vec[1] = $3;
                                          v -> vec[2] = $4;
                                          $$ = v;}
        ;
%%
```

Esto es equivalente a:

inicio → ε	exp → exp * NUMBER
inicio → inicio list	exp → NUMBER * exp
list → '\n'	exp → exp x exp
list → exp '\n'	num → NUMBER
list → num '\n'	num → vector . vector
exp → vector	num → | vector |
exp → exp + exp	Vector → [NUMBER NUMBER NUMBER]
exp → exp - exp	

Aunque la sintaxis presenta una recursividad por la izquierda, se resuelve de acuerdo con la jerarquía planteada. Los símbolos de +, -, *, etc., se colocan entre comillas simples y no como token, ya que, solamente tienen longitud de 1.

La jerarquía es la siguiente:

```
%left '+' '-'
%left '*'
%left 'x' '.'
```

Tanto el producto cruz, como el producto punto, tienen mayor precedencia y asocian por la izquierda. Dentro de nuestro archivo "*calculadoraVect.c*", se encuentran las funciones que realizan cada una de las operaciones.

Aquí dos ejemplos:

```
68. Vector  *escalarporVector(double c, Vector *v){
69.     Vector *r_vector = crearVector(v -> n);
70.     int i;
71.     for(i = 0; i < v -> n; i++){
72.         //Multiplica el escalar por cada valor del vector
73.         r_vector -> vec[i] = c * v->vec[i];
74.     }
75.     return r_vector;
76. }
```

La función de multiplicación por escalar recibe un double y un apuntador a vector. Dentro de la función se crea un vector auxiliar para guardar el resultado de la multiplicación y en el ciclo for se va multiplicando dicho escalar por cada valor del vector, para finalmente retornar el vector auxiliar.

```
110.double magnitud(Vector *a){
111.     double resultado = 0.0f;
112.     int i;
113.     for(i = 0; i < a->n; i++){
114.         resultado += ( a -> vec[i] * a -> vec[i] );
115.     }
116.     resultado = sqrt(resultado);
117.     return resultado;
118. }
```

Es una función de magnitud, recibe un apuntador a vector y retorna un double. Dentro del ciclo for, se va multiplicando cada elemento del vector por sí mismo para obtener el cuadrado de este, y cada resultado se va guardando en la variable "resultado" de tipo double, para que una vez terminado el ciclo se obtenga su raíz cuadrada y retorne dicho valor.

PRUEBAS DE FUNCIONAMIENTO

Suma y resta de vectores

La suma y resta de vectores, pueden realizarse tanto con números enteros como con números flotantes. El vector resultante se obtiene sumando o restando, (*dependiendo el caso*) los elementos en las mismas posiciones de cada vector; como se muestra en la siguiente figura:

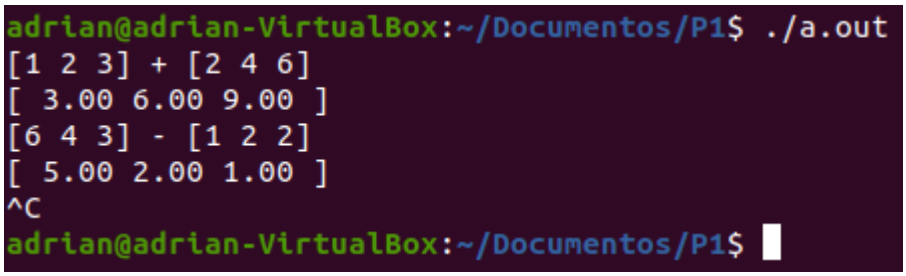

Figura 1. Ejecución del programa para suma y resta de vectores

Multiplicación por escalar

Es posible realizar la multiplicación cuando se encuentra primero el escalar o cuando esta primero el vector a multiplicar por el escalar, de igual manera el vector resultante se obtiene de multiplicar el escalar por cada uno de los elementos del vector:

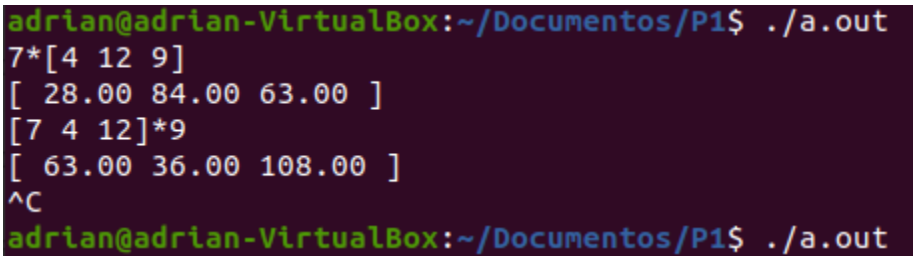

Figura 2. Ejecución del programa para la multiplicación por escalar

Producto punto

El producto punto se obtiene multiplicando los elementos en las mismas posiciones y sumando los resultados de cada multiplicación.

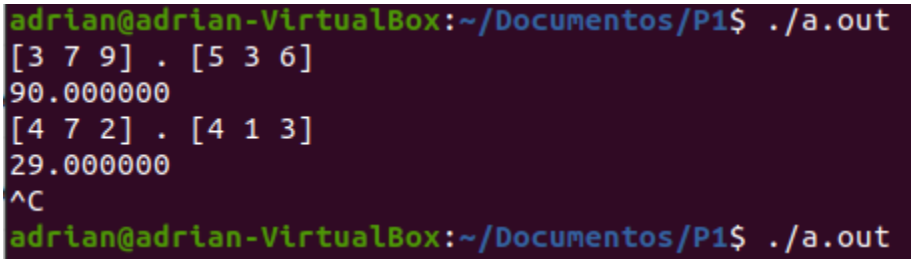

Figura 3. Ejecución del programa para el producto punto

En el primer caso el producto punto es 3*5 + 7*3 + 9*6 = 15 + 21 + 54 = 90 y en el segundo es 4*4 + 7*1 + 2*3 = 16 + 7 + 6 = 29

Producto Cruz

El producto cruz de obtiene de la siguiente manera:

$$(a_1, a_2, a_3) \times (b_1, b_2, b_3) = [(a_2b_3 - a_3b_2), (a_3b_1 - a_1b_3), (a_1b_2 - a_2b_1)]$$

Entonces para el primer caso,

$$\begin{aligned}(1 \quad 7 \quad 5) \times (1 \quad 3 \quad 8) &= [(7*8-5*3) \quad (5*1-1*8) \quad (1*3-7*1)] \\ &= [(56-15) \quad (5-8) \quad (3-7)] \\ &= [41 \quad -3 \quad -4]\end{aligned}$$

Para el segundo caso,

$$\begin{aligned}(5 \quad 1 \quad 7) \times (8 \quad 1 \quad 3) &= [(1*3-7*1) \quad (7*8-5*3) \quad (5*1-1*8)] \\ &= [(3-7) \quad (56-15) \quad (5-8)] \\ &= [-4 \quad 41 \quad -3]\end{aligned}$$

Su funcionamiento se puede observar en la siguiente figura:

```
adrian@adrian-VirtualBox:~/Documentos/P1$ ./a.out
[1 7 5] x [1 3 8]
[ 41.00 -3.00 -4.00 ]
[5 1 7] x [8 1 3]
[ -4.00 41.00 -3.00 ]
^C
adrian@adrian-VirtualBox:~/Documentos/P1$
```

Figura 4. Ejecución del programa para el producto cruz

Magnitud

La magnitud de un vector se obtiene sacando la raíz cuadrada de la suma de los cuadrados de los elementos, es decir, $\sqrt{a^2+b^2+c^2}$

```
adrian@adrian-VirtualBox:~/Documentos/P1$ ./a.out
| [8 4 3] |
9.433981
^C
adrian@adrian-VirtualBox:~/Documentos/P1$
```

Figura 5. Ejecución del programa para la magnitud de vectores

En este caso es $\sqrt{8^2+4^2+3^2} = \sqrt{64+16+9} = \sqrt{89} \approx 9.43398$

De esta manera se puede comprobar el correcto funcionamiento de cada una de las operaciones propuestas.

Conclusión

Al realizar esta práctica, se lograron comprobar los conocimientos vistos en clase de YACC básico y del hoc1. Además de poner en práctica lo visto acerca de la sintaxis que se utiliza en YACC para definir alguna gramática, acciones gramaticales, etc. Así como las respectivas definiciones de tokens y jerarquías de operaciones, que son de ayuda para eliminar la recursividad por la izquierda que pueda presentar en la gramática propuesta.