Instituto Politécnico Nacional

Escuela Superior de Cómputo

Departamento de Ciencias e Ingeniería

de la Computación

Compiladores

Práctica 3: Tabla de símbolos HOC3

Opción: Calculadora de vectores

Grupo: 5CM2

Alumno:

Sandoval Hernández Eduardo

Profesor:

Tecla Parra Roberto

Fecha de entrega:

29 de abril de 2022

Introducción

Yet Another Compiler Compiler (YACC) es un generador de analizadores. A partir de un archivo fuente en yacc se puede generar un archivo fuente en C el cual contendrá el analizador sintáctico. Este analizador requerirá de un analizador léxico externo para funcionar, esto debido a que el archivo fuente en C que genera yacc contiene llamadas a la función yylex() la cual debe estar definida y debe ser capaz de devolver el tipo de lexema encontrado. Adicionalmente es necesario incluir una función yyerror() la cual será invocada cuando el analizador sintáctico encuentre un símbolo que no encaje con la gramática [1].

El presente trabajo consiste en los resultados obtenidos de la práctica 3 con el tema relacionado a Hoc3, para esta práctica se reutilizó el código de la práctica 1 manteniendo la opción de calculadora de vectores, con la diferencia de que en esta ocasión se modificará para que pueda manejar variables como lo indica Hoc3. Para ello se realizaron cambios a los 3 archivos: vcalc.c, vcalc.h y vcalc.y. Con esto se logra realizar las mismas operaciones, pero haciendo uso de variables ya sea para almacenar vectores ingresados por el usuario o el resultado de alguna operación de vectores que resulta en un vector, esto debido a que las variables solo almacenarán vectores.

Desarrollo

Repasando el código de la práctica 1, se tienen dos archivos: vcalc.h y vcalc.c, en los cuales se encuentran los prototipos y la implementación de las funciones necesarias para realizar las operaciones con vectores, estas son: struct vector, que define el cómo se estructura un vector con sus componentes; creaVector, función encargada de crear un vector en n dimensiones; imprimeVector, función encargada de imprimir en pantalla los valores de las componentes del vector que recibe como parámetro; copiaVector, función encargada de realizar una copia de un vector recibido como parámetro; sumaVector, función que suma dos vectores y retorna el vector resultante; restaVector, función que resta dos vectores y retorna el vector resultante; escalarVector, función que realiza la multiplicación de un vector por un escalar y retorna el vector resultante; productoCruz, función que realiza el producto cruz de dos vectores y retorna el vector resultante; productoPunto, función que realiza el producto punto entre dos vectores y retorna el escalar resultante; vectorMagnitud, función que retorna la magnitud de un vector. Para esta práctica se incluyó código de Hoc3, esto es: struct Symbol, que define cómo se estructura la tabla de símbolos siento esta una lista simplemente ligada; install, función que ingresa un símbolo con su valor a la tabla de símbolos; lookup, función que comprueba si existe un determinado símbolo en la tabla de símbolos.

Para la implementación en Yacc, se usó el archivo vcalc.y creado en la práctica 1 donde se agregaron más bibliotecas para su funcionamiento, además de los prototipos para las funciones warning, execerror y fpecatch las cuales fueron funciones omitidas en la práctica 1.

Se modificó el tipo de dato de la pila de Yacc agregando un apuntador a un Symbol, quedando de la siguiente manera:

```
%union{
    double val;
    Vector *vector;
    Symbol *sym;
}
```

Esto debido a que en la pila se almacenarán ya no solo los vectores y sus valores sino también los símbolos.

Para los símbolos terminales se mantuvieron los anteriores agregando un símbolo más para las variables:

```
%token<val>     NUMBER      //  Terminal para números positivos
%token<val>     nNUMBER     //  Terminal para números negativos
%token<sym>     VAR INDEF   //  Terminal para variables
```

De igual forma con los símbolos no terminales, se mantuvieron los anteriores y se agregó uno para las asignaciones a las variables:

```
%type<vector>   exp         //  No Terminal para expresiones
%type<vector>   vector      //  No Terminal para vectores
%type<vector>   asigna      //  No Terminal para asignaciones
%type<val>      NTnumber    //  No Terminal auxiliar para números
%type<val>      auxNumber   //  Segundo No Terminal auxiliar para números
```

Para la precedencia, se agregó la del operador de asignación:

```
%right '='
%left '+' '-'               // Suma y resta
%left '*'                   // Producto de vector por escalar
%left 'x' '.'               // Producto cruz y producto punto
```

En cuanto a la producción de list, se agregó una más para las asignaciones:

```
/*
    list => '\n'            // Salto de linea
    list => exp '\n'        // Expresión
    list => asigna '\n'     // Asignación
    list => NTnumber '\n'   // Número
*/
    list: '\n'
        | exp '\n'          {imprimeVector($1);}
        | asigna '\n'       {imprimeVector($1);}
        | NTnumber '\n'     {printf("\tResultado: %lf\n",$1);}
        ;
```

Con lo cual también se agregó la producción de asigna para evaluar las asignaciones:

```
/*
    asigna => VAR '=' exp
*/
    asigna  : VAR '=' exp     {
                                $$ = $1->u.val = $3;
                                $1->type=VAR;
                              }
            ;
```

Para las producciones relacionadas con exp, se agregaron dos más para las variables y las asignaciones a estas:

```
/*
    exp => vector
    exp => exp '+' exp
    exp => exp '-' exp
    exp => exp '*' auxNumber
    exp => auxNumber * exp
    exp => exp 'x' exp
*/
    exp : vector
        | VAR                {
                                 if($1->type == INDEF)execerror("No se encuentra la variable: ",$1->nam
e);
                                 $$ = $1->u.val;
                             }
        | asigna             {$$ = $1;}
        | exp '+' exp        {$$ = sumaVector($1, $3);}      //  Suma de vectores
        | exp '-' exp        {$$ = restaVector($1, $3);}     //  Resta de vectores
        | exp '*' auxNumber  {$$ = escalarVector($3, $1);}   //  Producto escalar por la derecha
        | auxNumber '*' exp  {$$ = escalarVector($1, $3);}   //  Producto escalar por la izquierda
        | exp 'x' exp        {$$ = productoCruz($1, $3);}    //  Producto cruz de vectores
        ;
```

La última producción que se modificó fue para el no terminal auxiliar NTnumber:

```
/*
    NTnumber => auxNumber
    NTnumber => vector '.' vector
    NTnumber => '|' vector '|'
*/
    NTnumber: auxNumber                                     //  Numero negativo o positivo
            | exp '.' exp     {$$ = productoPunto($1, $3);}     //  Resultado del producto punto
            | '|' exp '|'        {$$ = vectorMagnitud($2);}     //  Resultado de la magnitud
            ;
```

Siendo esto último necesario para poder realizar el producto punto y obtener la magnitud de los vectores guardados en variables.

El resto de las producciones se mantuvieron igual que en la práctica 1.

Finalmente, para el código de soporte, además de agregar las funciones mencionadas al inicio del desarrollo, la modificación más importante se realizó en la función yylex(), en la cual se agregó el código necesario para la búsqueda de variables y la asignación de valores a estas, siendo dicha parte del código la siguiente:

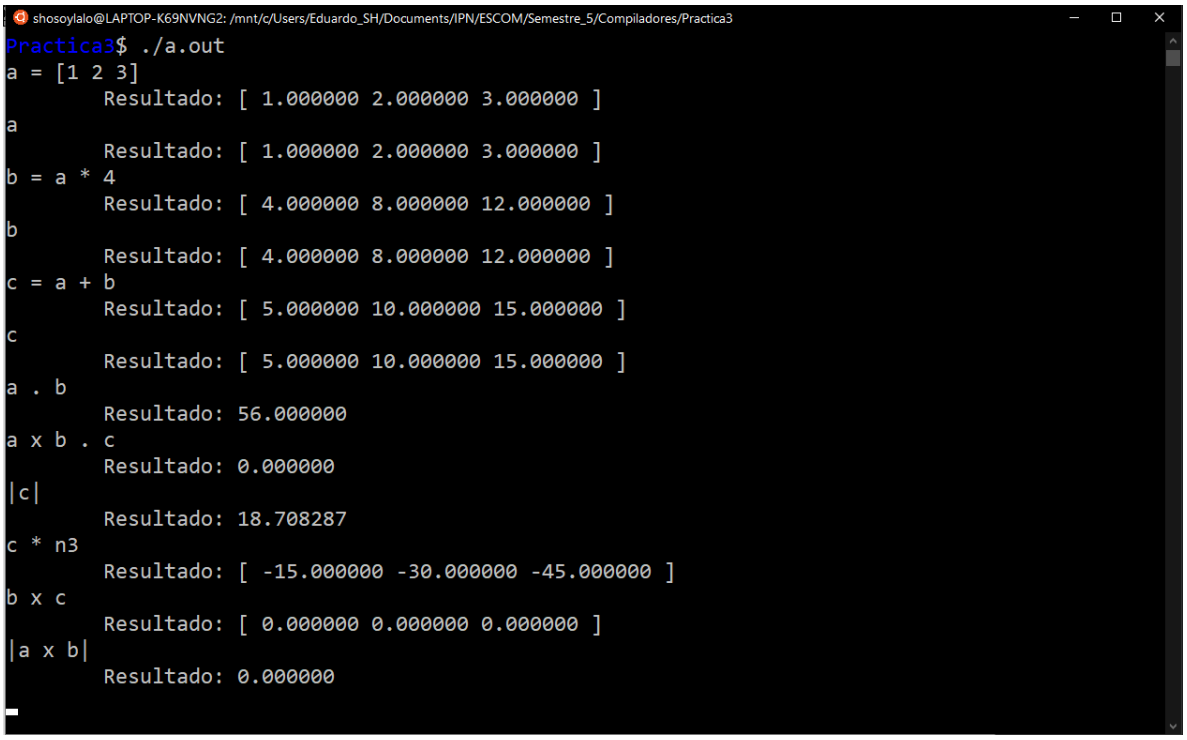
```c
if(isalpha(c) && c != 'x'){         // Comprueba que las variables sean letras excepto la 'x'
    Symbol *s;
    char sbuf[200], *p=sbuf;
    do{
        *p++ = c;
    }while((c=getchar()) != EOF && isalnum(c) && c != 'x');
    ungetc(c, stdin);
    *p = '\0';
    if((s=lookup(sbuf)) == (Symbol*)NULL)   // Comprueba si se encuentra la variable en la tabla de simbolos
        s = install(sbuf, INDEF, (Vector*)NULL);    // Si no se encuentra, la agrega
    yylval.sym=s;
    if(s->type == INDEF)
        return VAR;
    else
        return s->type;
}
```

Como se puede apreciar, el código comprobará que si un carácter alfabético a excepción de la ‘x’ (debido a que se utiliza para el producto cruz) puede ser tomado como una variable, si no existe dicha variable se agregará a la tabla de símbolos, mientras que, si ya se encuentra, retornará el valor asociado a esta.

Para la compilación y ejecución del programa se ingresan los mismos comandos que en la práctica 1:

```
shosoylalo@LAPTOP-K69NVNG2: /mnt/c/Users/Eduardo_SH/Documents/IPN/ESCOM/Semestre_5/Compiladores/Practica3
shosoylalo@LAPTOP-K69NVNG2:/mnt/c/Users/Eduardo_SH/Documents/IPN/ESCOM/Semestre_5/Compiladores/Practica3$ yacc -d vcalc.y
vcalc.y: warning: 1 shift/reduce conflict [-Wconflicts-sr]
shosoylalo@LAPTOP-K69NVNG2:/mnt/c/Users/Eduardo_SH/Documents/IPN/ESCOM/Semestre_5/Compiladores/Practica3$ gcc y.tab.c -lm
shosoylalo@LAPTOP-K69NVNG2:/mnt/c/Users/Eduardo_SH/Documents/IPN/ESCOM/Semestre_5/Compiladores/Practica3$ ./a.out
```

Para comprobar el funcionamiento del programa se probaron diversas operaciones con vectores y variables que contienen vectores:



Como se muestra en la evidencia anterior, el programa es capaz de almacenar vectores en variables, y usar dichas variables para realizar operaciones cuyo resultado dependiendo el caso puede ser asignado a una variable.

Conclusión

Esta práctica resultó un poco más sencilla de realizar comparado a su implementación anterior, esto debido a que ya teniendo funcional la calculadora de vectores, se volvió más sencillo hacer las modificaciones necesarias para implementar las variables para dicha calculadora, por tanto, se puede decir que el resultado es satisfactorio, sin embargo, eso no cierra la posibilidad de mejora en alguna práctica futura.

Referencias

[1] F. Simmross Wattenberg (s.f.). “El generador de analizadores sintácticos yacc”. [Internet]. Disponible en https://www.infor.uva.es/~mluisa/talf/docs/labo/L8.pdf