Instituto Politécnico Nacional

Escuela Superior de Cómputo

Departamento de Ciencias e Ingeniería

de la Computación

Compiladores

Práctica 5: Decisiones y ciclos HOC5

Opción: Calculadora de vectores

Grupo: 5CM2

Alumno:

Sandoval Hernández Eduardo

Profesor:

Tecla Parra Roberto

Fecha de entrega:

10 de junio 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 reporte consiste en los resultados obtenidos de la práctica 5 con el tema relacionado a HOC5, para esta práctica se reutilizó el código de la práctica 4 manteniendo la opción de la calculadora de vectores, realizando cambios para que en esta ocasión la calculadora pueda realizar decisiones y ciclos, más específicamente el condicional “if” y el ciclo “while”. Para ello se realizaron cambios en los diferentes archivos de código que componen la práctica, logrando realizar las decisiones y ciclos deseados.

Desarrollo

Para la realización de esta práctica se agregó una estructura keywords la cual contiene los valores para las palabras reservadas: if, else, while y print. Estos símbolos se agregarán a la tabla de símbolos con ayuda de la función init, con esto ya se podrá hacer uso de estas palabras para su interpretación en el código.

```
static struct {
    char *name;
    int val;
} keywords[] = {
    "if", IF,
    "else", ELSE,
    "while", WHILE,
    "print", PRINT,
    0, 0,
};

int init(){
    int i;
    Symbol *s;
    for(i = 0; keywords[i].name; i++)
        installVector(keywords[i].name, keywords[i].val, NULL);
}
```

Adicionalmente se agregaron funciones de comparación: MayorQue, MayorIgualQue, MenorQue, MenorIgualQue, IgualQue, NoIgualQue. Las cuales por como su nombre lo indica sirven para comprobar si dos vectores cumplen alguna condición, en este caso, comparándolos por su magnitud. También se encuentran las funciones booleanas And, Or y Not. Se agregaron también las funciones whilecode e ifcode, por como sus nombres indican, sirven para realizar las acciones del ciclo while y el condicional if.

En cuanto a la gramática, la pila de datos de Yacc se mantuvo sin cambios salvo por la adición de un tipo de dato que al final no se utilizó.

Se agregaron los símbolos terminales y no terminales necesarios para las producciones de los ciclos y decisiones:

```
%token<sym>         PRINT WHILE IF ELSE BLTIN
%type<inst>         stmt stmtlist cond while if end
```

La jerarquía de los operadores también se cambió ya que se agregaron los necesarios para las comparaciones y operadores booleanos:

```
%right '='                  // Asignacion
%left OR AND                // Or & And
%left GT GE LT LE EQ NE
// GreaterThan GreaterEqual LessThan LessEqual Equal NonEqual
%left '+' '-'               // Suma y resta
%left '*'                   // Producto de vector por escalar
%left 'x' '.'               // Producto cruz y producto punto
%left NOT
```

La producción de list también se modificó para agregar stmt (statement) y con ello el poder realizar varios ciclos o decisiones consecutivas:

```
list: '\n'
    | exp '\n'          {code2(printVector, STOP); return 1;}
    | asigna '\n'       {code2(pop, STOP); return 1;}
    | NTnumber '\n'     {code2(printDouble, STOP); return 1;}
    | stmt '\n'         {code(STOP); return 1;}
    | error '\n'        {yyerror;}
    ;
```

También se modificó la acción gramatical de la asignación, esto para evitar apuntadores salvajes:

```
asigna  : VAR '=' exp     {$$ = $3; code3(varpush, (Inst)$1, assign);}
        ;
```

Adicionalmente se agregaron las siguientes producciones:

La producción stmtlist para poder ingresar statements consecutivos en los ciclos y decisiones:

```
stmtlist:                   {$$ = progp;}
        | stmtlist '\n'
        | stmtlist stmt
        ;
```

La producción stmt para generar los statements que pueden ser expresiones u otros ciclos y decisiones:

```
stmt: exp                   {code(pop);}
    | PRINT exp             {code(printVector); $$ = $2;}
    | while cond stmt end   {
                            ($1)[1] = (Inst)$3;
                            ($1)[2] = (Inst)$4;
                            }
    | if cond stmt end      {
                            ($1)[1] = (Inst)$3;
                            ($1)[3] = (Inst)$4;
                            }
    | if cond stmt end ELSE stmt end   {
                            ($1)[1] = (Inst)$3;
                            ($1)[2] = (Inst)$6;
                            ($1)[3] = (Inst)$7;
                            }
    | '{' stmtlist '}'      {$$ = $2;}
    ;
```

La producción cond, que sirve para las condiciones que deben ser verdaderas para cumplirse el ciclo o la decisión:

```
cond: '(' exp ')'   {code(STOP); $$ = $2;}
    ;
```

Las producciones while, if y end que sirven para agregar las llamadas a whilecode, ifcode y los STOP necesarios para la ejecución de los ciclos y condiciones:

```
while: WHILE    {$$ = code3(whilecode,STOP,STOP);}
    ;
```

```
if: IF  {$$ = code(ifcode); code3(STOP,STOP,STOP);}
    ;
```

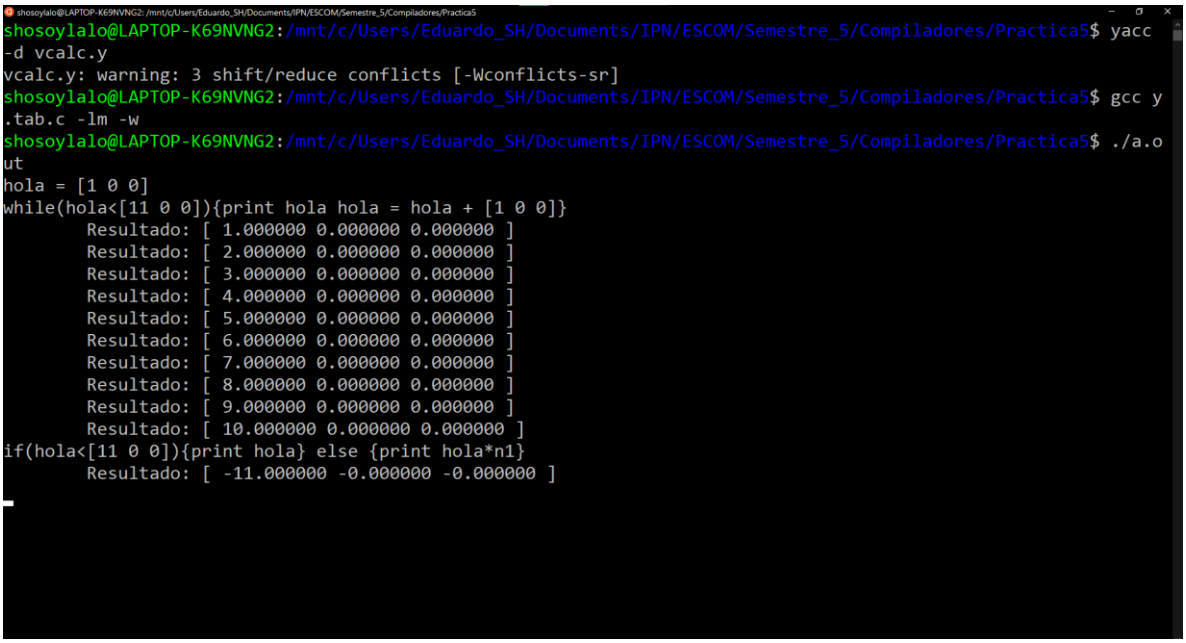
```
end:    {code(STOP); $$ = progp;}
    ;
```

En cuanto al código de apoyo, se agregó la función follow la cual servirá para que junto a una condición switch el analizador léxico pueda reconocer los operadores de comparación y no sean confundidos con otros como podrían ser el operador de asignación o el utilizado para obtener la magnitud de un vector:

```
int follow(int expect, int ifyes, int ifno){
    int c = getchar();
    if(c == expect)
        return ifyes;
    ungetc(c,stdin);
    return ifno;
}
```

```
switch(c){
    case '>': return follow('=',GE,GT);
    case '<': return follow('=',LE,LT);
    case '=': return follow('=',EQ,'=');
    case '!': return follow('=',NE,NOT);
    case '|': return follow('|',OR,'|');
    case '&': return follow('&',AND,'&');
    case '\n': lineno++; return c;
    default: return c;
}
```

Para la ejecución y compilación de la práctica se ejecutan los mismos comandos que en la práctica anterior, para probar los ciclos se realizará uno que se repetirá 10 veces y posteriormente una condición if con el resultado final del ciclo:



Como se muestra en la evidencia anterior, el programa puede realizar los ciclos y decisiones sin complicación alguna, esto se demuestra en como el ciclo muestra en todo momento el estado de la variable “hola” y se termina cuando deja de cumplirse la condición, lo que también se demuestra en la decisión ya que al no cumplir con la condición se ejecuta lo que se encuentra en el statement de else.

Conclusión

Esta práctica resultó más sencilla que la anterior ya que solo se agregaron producciones nuevas y con ello fragmentos de código nuevos, dejando casi intacto el código generado en la practica anterior, lo más interesante fue entender como es que se ejecutan los ciclos y decisiones con las funciones whilecode e ifcode, ya que entender el funcionamiento de estas es una pieza clave para la realización de la práctica 6.

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