Instituto Politécnico Nacional

Escuela Superior de Cómputo

Compiladores

Roberto Tecla Parra

Práctica 5

Máquina Virtual de Pila

3CM16

Humberto Alejandro Ortega Alcocer (2016630495)

31 de Mayo del 2023

Introducción

El objetivo de esta práctica es la implementación de estructuras de control como lo son las condicionales y los ciclos. Para lograr esto, primero deberemos modificar la gramática en YACC para reflejar estos nuevos tokens y adecuar nuestra implementación conforme a HOC5 para que podamos realizar las expresiones lógicas correspondientes para evaluarlas y tomar decisiones.

Modificaciones en YACC

Lo primero fue añadir los tokens y los tipos para las nuevas expresiones y estructuras que queremos añadir a nuestra gramática, de la siguiente forma:

```
%token<sym> NUMBER PRINT VAR VARVECTOR VARESCALAR INDEF WHILE IF ELSE
%type   <inst>  stmt asgnVector asgnEscalar expVectorial expEscalar stmtlist cond while if end

%right '='
%left      OR
%left      AND
%left      GT GE LT LE EQ NE
%left '+' '-'
%left "*"
%left 'x'
%left      NOT
```

Lo siguiente que realizamos fue definir las expresiones que podremos encontrar en nuestra gramática de la siguiente manera:

```
expVectorial: vector
   | VARVECTOR                        { $$=code3(varpush, (Inst)$1, evalVector); }
   | expVectorial '+' expVectorial    { code(add); }
   | expVectorial '-' expVectorial    { code(sub); }
   | expVectorial 'x' expVectorial    { code(cross); }
   | expEscalar '*' expVectorial      { code(multiEscalarVect); }
   | expVectorial '*' expEscalar      { code(multiVectEscalar); }
   | '(' expVectorial ')'             {$$ = $2;}
   ;
expEscalar: NUMBER                    { $$ = code2(escalarpush, (Inst)$1); }
   |  VARESCALAR                      { $$ = code3(varpush, (Inst)$1, evalEscalar); }
   | '|' expVectorial '|'             { code(magnitude); }
   | '|' expEscalar '|'               { $$ = $2; }
   | expVectorial '*' expVectorial    { code(point); }
   | '(' expEscalar ')'               { $$ = $2; }
   //ESCALAR-ESCALAR
   | expEscalar GT expEscalar         { code(gtEE); }
          | expEscalar GE expEscalar         { code(geEE); }
          | expEscalar LT expEscalar         { code(ltEE); }
          | expEscalar LE expEscalar         { code(leEE); }
          | expEscalar EQ expEscalar         { code(eqEE); }
          | expEscalar NE expEscalar         { code(neEE); }
          | expEscalar AND expEscalar        { code(andEE); }
          | expEscalar OR expEscalar         { code(orEE); }
          | NOT expEscalar                   { $$ = $2; code(notE); }
   //VECTOR-VECTOR
   | expVectorial GT expVectorial     { code(gtVV); }
          | expVectorial GE expVectorial     { code(geVV); }
          | expVectorial LT expVectorial     { code(ltVV); }
          | expVectorial LE expVectorial     { code(leVV); }
          | expVectorial EQ expVectorial     { code(eqVV); }
          | expVectorial NE expVectorial     { code(neVV); }
          | expVectorial AND expVectorial    { code(andVV); }
          | expVectorial OR expVectorial     { code(orVV); }
          | NOT expVectorial                 { $$ = $2; code(notV); }
   //ESCALAR-VECTOR
   | expEscalar GT expVectorial     { code(gtEV); }
          | expEscalar GE expVectorial     { code(geEV); }
          | expEscalar LT expVectorial     { code(ltEV); }
          | expEscalar LE expVectorial     { code(leEV); }
          | expEscalar EQ expVectorial     { code(eqEV); }
          | expEscalar NE expVectorial     { code(neEV); }
```

```
        | expEscalar AND expVectorial     { code(andEV); }
        | expEscalar OR expVectorial      { code(orEV); }
  //VECTOR-ESCALAR
  | expVectorial GT expEscalar      { code(gtVE); }
        | expVectorial GE expEscalar      { code(geVE); }
        | expVectorial LT expEscalar      { code(ltVE); }
        | expVectorial LE expEscalar      { code(leVE); }
        | expVectorial EQ expEscalar      { code(eqVE); }
        | expVectorial NE expEscalar      { code(neVE); }
        | expVectorial AND expEscalar     { code(andVE); }
        | expVectorial OR expEscalar      { code(orVE); }
```

Como se puede observar, generamos expresiones lógicas tanto para comparaciones entre vectores, así como vectores y escalares, escalares y vectores y, finalmente, escalares y escalares.

Sabemos que para implementar estructuras de control, debemos de poder especificar las acciones adscritas a dichas estructuras, a esto le llamaremos *statements* y se ubican en la gramática de la siguiente manera:

```
stmt:   asgnEscalar                 { code(pop1); }
  | asgnVector                  { code(pop1); }
        | PRINT expEscalar              { code(prexpresc); $$ = $2;}
        | PRINT expVectorial            { code(prexprvec); $$ = $2;}
        | while cond stmt end           { ($1)[1] = (Inst)$3;     /* cuerpo de la iteracion */
                                          ($1)[2] = (Inst)$4; }   /* terminar si la condicion no se cumple */
        | if cond stmt end              {    /* proposicion if que no emplea else */
                                             ($1)[1] = (Inst)$3;     /* parte then */
                                             ($1)[3] = (Inst)$4;
                                    }   /* terminar si la condicion no se cumple */
        | if cond stmt end ELSE stmt end  {  /* proposicion if con parte else */
                                    ($1)[1]   =   (Inst)$3;    /*  parte then  */
                                    ($1)[2]   =   (Inst)$6;    /* parte else   */
                                    ($1)[3]   =   (Inst)$7;   } /*   terminar si la condicion no se cumple  */
        |   '{'   stmtlist   '}'          {   $$  =  $2;   }
  ;
```

Con ese *statement* nosotros podremos definir lo que sucederá si se cumple, o no, la estructura de control en cuestión.

Posteriormente realizamos la definición de la condición, de la siguiente manera:

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

La cual nos define como se verá una condición en nuestro programa final, es decir, una expresión entre dos paréntesis, parecido a la sintáxis de lenguaje C.

Lo siguiente fue definir entonces la estructura general de nuestro *if* y nuestro *while* de la siguiente manera:

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

Que realmente corresponden más a las acciones gramaticales que nos insertarán en nuestro mapa de memoria los STOP correspondientes a las direcciones de memoria que necesitamos para poder operar las estructuras mencionadas.

Por parte del analizador léxico, realizamos las siguientes adecuaciones:

```
//Analizador léxico
int yylex(){
   int c;
   while  ((c=getchar()) ==  ' ' ||  c ==   '\t')
            ;
   if (c == EOF)
      return 0;
   if (c == '.' || isdigit(c)) {   /* numero */
      double d;
      ungetc(c, stdin);
      scanf("%lf", &d);
      yylval.sym = install("", NUMBER, d);
      return NUMBER;
   }
   if (isalpha(c) && c!=120) {
      Symbol *s;
      char sbuf[100], *p = sbuf;
      do {
         if (p >= sbuf + sizeof(sbuf) - 1) {
            *p = '\0';
            printf("\x1b[31m%s\n\x1b[0m", "name too long");
         }
      *p++ = c;
      } while ((c=getc(stdin)) != EOF && isalnum(c));
      ungetc(c, stdin);
      *p = '\0';
      if ((s=lookup(sbuf)) == 0)
               s=install(sbuf, INDEF, 0.0);
      yylval.sym = s;
      return s->type == INDEF ? VAR : s->type;
   }
      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 '\n';
      default:                  return c;
      }
}
```

Con las cuales se puede observar que ahora podemos especificar distintos tipos de condiciones lógicas y recibiremos la evaluación de las expresiones en función de las acciones gramaticales correspondientes.

Para poder realizar la evaluación correspondiente a las expresiones, modificamos las funciones asociadas a los operadores para que trabajaran con vectores:

```
// VECTOR-VECTOR
void gtVV()
{
   Datum d1, d2;
   d2 = pop();
   d1 = pop();
   d1.num = (double)(magnitudVector(d1.vect) > magnitudVector(d2.vect));
   push(d1);
}

void ltVV()
{
   Datum d1, d2;
   d2 = pop();
   d1 = pop();
   d1.num = (double)(magnitudVector(d1.vect) < magnitudVector(d2.vect));
   push(d1);
}

void geVV()
{
   Datum d1, d2;
   d2 = pop();
   d1 = pop();
   d1.num = (double)(magnitudVector(d1.vect) >= magnitudVector(d2.vect));
```

```
   push(d1);
}

void leVV()
{
   Datum d1, d2;
   d2 = pop();
   d1 = pop();
   d1.num = (double)(magnitudVector(d1.vect) <= magnitudVector(d2.vect));
   push(d1);
}

void eqVV()
{
   Datum d1, d2;
   d2 = pop();
   d1 = pop();
   d1.num = (double)(magnitudVector(d1.vect) == magnitudVector(d2.vect));
   push(d1);
}
```

Estos son solo algunos ejemplos, pero este tipo de adecuaciones se realizaron para todas las operaciones lógicas posibles.

Nota: usamos la magnitud de los vectores puesto que implementar la comparación de elemento a elemento era prácticamente imposible o sumamente complejo para fines de la práctica actual.

Lo siguiente fue añadir la definición de lo que sucede con un *while* y para eso usamos el siguiente código y modificamos que usara la parte de la unión correspondiente a nuestro vector:

```
void whilecode()
{
   Datum d;
   Inst *savepc = pc;   /*  cuerpo de la iteracion  */
   execute(savepc + 2); /*   condicion   */
   d = pop();
   while (d.num)
   {
      execute(*((Inst **)(savepc))); /*  cuerpo  */
      execute(savepc + 2);
      d = pop();
   }

   pc = *((Inst **)(savepc + 1)); /*   siguiente proposicion   */
}
```

Y las modificaciones que realizamos para el *if* son similares:

```
void ifcode()
{
   Datum d;
   Inst *savepc = pc;   /* parte then */
   execute(savepc + 3); /*  condicion   */
   d = pop();
   if (d.num)
      execute(*((Inst **)(savepc)));
   else if (*((Inst **)(savepc + 1))) /*  parte else?   */
      execute(*((Inst **)(savepc + 1)));
   pc = *((Inst **)(savepc + 2)); /*  siguiente proposicion   */
}
```

En la sección de palabras clave, se añadieron las palabras reservadas para ambas estructuras, de la siguiente manera:

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

Las cuales podemos ver que se usan en varias ocasiones en la gramática de YACC.

Prueba de Funcionamiento

Compilación

Para compilar el programa lo primero que debemos hacer será generar el parser con YACC usando el siguiente comando:

```
$ yacc -y -d calculadora_vectores.y
```

Una vez que tengamos los archivos `y.tab.c` y `y.tab.h`, entonces compilaremos el programa general:

```
$ gcc *.c -lm -o calculadora_vectores
```

Con esto contaremos con el programa listo para ejecutarse.

Ejecución

Para la ejecución realizaremos las siguientes pruebas:

```
./calculadora_vectores
hugo=[1 2 3]
[ 1.000000 2.000000 3.000000 ]
paco=[2 4 6]
[ 2.000000 4.000000 6.000000 ]
hugo + paco
[ 3.000000 6.000000 9.000000 ]
luis = paco - hugo
[ -1.000000 -2.000000 -3.000000 ]
luis
```

```
[ -1.000000 -2.000000 -3.000000 ]
if (hugo < paco) { print luis }
[ -1.000000 -2.000000 -3.000000 ]
if (paco < hugo) { print luis }
contador = [0 1 0]
[ 0.000000 1.000000 0.000000 ]
while (contador < 5) { contador = contador + [0 1 0] print contador }
[ 0.000000 2.000000 0.000000 ]
[ 0.000000 3.000000 0.000000 ]
[ 0.000000 4.000000 0.000000 ]
[ 0.000000 5.000000 0.000000 ]
```

Con lo cual se puede observar que el programa retiene su capacidad de almacenar y usar variables, así como de generar condiciones con *if* y ciclos con la sentencia *while*.

Conclusión

Para realizar esta práctica tuve que estudiar mucho el código de HOC5 puesto que no entendía realmente qué hacían los STOPs en varios lugares, por ejemplo para generar el código del *while*. Luego de varias revisiones entendí que se usan para almacenar direcciones de memoria y son dónde nosotros, mediante el contador de programa, acudimos para decidir qué hacer.

Me pareció curioso que debamos implementar casi todas las comparaciones mediante la cardinalidad (o cantidad) de cada vector, esto si bien es funcional propone errores en caso de que las comparaciones pudieran ser erróneas, por ejemplo comparar si dos vectores son iguales actualmente solo compara que su número de elementos sea el mismo pero no compara que los elementos en cuestión sean iguales ni mucho menos compara que se encuentren en el mismo orden.

– Humberto Alejandro Ortega Alcocer.