Instituto Politécnico Nacional

Escuela Superior de Cómputo

Compiladores

Roberto Tecla Parra

Práctica 4

Máquina Virtual de Pila

3CM16

Humberto Alejandro Ortega Alcocer (2016630495)

31 de Mayo del 2023

Introducción

El objetivo de esta práctica es implementar una máquina virtual de pila en nuestra calculadora de vectores. Esta nos permitirá tener un espacio de memoria (virtual) dónde podremos cargar definiciones de las operaciones así como variables y otros procedimientos. Para lograr la implementación, primero modificaremos la gramática en YACC, ajustaremos algunas acciones gramaticales y mostraemos su ejecución final.

Modificaciones en YACC

Lo primero fue definir las macros para poder usar las funciones que generan código, de la siguiente manera:

```
#define code2(c1, c2)     code(c1); code(c2);
#define code3(c1, c2, c3) code(c1); code(c2); code(c3);
```

En el lado de la gramática, las modificaciones fueron las siguientes:

```
list:
   | list '\n'
   | list asgnVector '\n'     { code2(printVector, STOP); return 1; }
   | list asgnEscalar '\n'    { code2(printEscalar, STOP); return 1; }
   | list expVectorial '\n'   { code2(printVector, STOP); return 1; }
   | list expEscalar '\n'     { code2(printEscalar, STOP); return 1; }
   ;
asgnVector: VAR '=' expVectorial          { code3(varpush, (Inst)$1, assignVector); } //Declaracion
          | VARESCALAR '=' expVectorial   { code3(varpush, (Inst)$1, assignVector); } //Redefiniciones
          | VARVECTOR '=' expVectorial    { code3(varpush, (Inst)$1, assignVector); }
          ;
asgnEscalar: '#' VAR '=' expEscalar          { code3(varpush, (Inst)$2, assignEscalar); } //Declaracion
           | '#' VARESCALAR '=' expEscalar   { code3(varpush, (Inst)$2, assignEscalar); }  //Redefeniciones
           | '#' VARVECTOR '='expEscalar     { code3(varpush, (Inst)$2, assignEscalar); }
           ;
expVectorial: vector
   | VAR                              { code3(varpush, (Inst)$1, evalVector); }
   | 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 ')'
   ;
expEscalar: NUMBER                    { code2(escalarpush, (Inst)$1); }
   |  VARESCALAR                      { code3(varpush, (Inst)$1, evalEscalar); }
   | '|' expVectorial '|'             { code(magnitude); }
   | '|' expEscalar '|'
   | expVectorial '*' expVectorial    { code(point); }
   | '(' expEscalar ')'
   ;
vector: '[' listnum ']' { code(vectorpush); }
   ;
listnum:
   | NUMBER listnum { code2(numpush, (Inst)$1); }
   ;
```

Que, como se puede observar, agrega el uso de la función *code* en varias partes (así como sus homónimos code2 y code3) para poder hacer llamadas a funciones que instalaremos en la Máquina Virtual de Pila.

Acciones Gramaticales

Lo primero que tuvimos que modificar fue la función *initConstants* a modo de que pudiera trabajar con vectores:

```
initConstants() /* instalar constantes y predefinidos en la tabla */
{
          int i;
          Symbol *s;

          for (i = 0; consts[i].name; i++)
                    s = install(consts[i].name, VARESCALAR, consts[i].cval);
          s->u.varVector = NULL;
}
```

Con esto, ahora nos dirigimos a *symbol* para modificar los valores que también deberían trabajar con nuestros vectores:

```
typedef struct Symbol
{ /* entrada de la tabla de símbolos */
          char *name;
          short type; /* VAR, VARESCALAR, VARVECTOR INDEF */
          union
          {
                    double varEscalar; /* si es VARESCALAR*/
                    Vector *varVector; /* si es VARVECTOR*/
          } u;
          struct Symbol *next; /* para ligarse al sig. */
} Symbol;

Symbol *install(char *s, int t, double d), *lookup(char *s);

typedef union Datum
{ /* tipo de la pila del interprete */
          double num;
          Vector *vect;
          Symbol *sym;
} Datum;

extern Datum pop();
typedef void (*Inst)(void); /* instruccion de maquina */
```

Siguiendo con las modificaciones, tuvimos que modificar las funciones de *lookup* e *install* a modo de que también pudieran trabajar con nuestros tipos de datos de vectores usando la MVP y no la tabla de símbolos:

```
Symbol *lookup(char *s) /* encontrar s en la tabla de símbolos */
{
          Symbol *sp;
          for (sp = symlist; sp != (Symbol *)0; sp = sp->next)
                    if (strcmp(sp->name, s) == 0)
                              return sp;
          return 0; /* 0 ==> no se encontró */
}

Symbol *install(char *s, int t, double d) /* instalar s en la tabla de símbolos */
{
          Symbol *sp;
          char *emalloc();
          sp = (Symbol *)emalloc(sizeof(Symbol));
          sp->name = emalloc(strlen(s) + 1); /* +1 para '\0' */
          strcpy(sp->name, s);
          sp->type = t;
          sp->u.varEscalar = d;
          sp->next = symlist; /*  poner al frente de la lista   */
          symlist = sp;
          return sp;
}

char *emalloc(unsigned n) /*   revisar el regreso desde malloc  */
{
          char *p;
          p = malloc(n);
          if (p == 0)
```

```
                printf("out of memory");
        return p;
}
```

Con estas modificaciones fue suficiente para lograr que nuestra práctica anterior pudiera hacer uso del código provisto por el profesor en HOC4 para poder usar la nueva Máquina Virtual de Pila.

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)
1.00 0.00 0.00 2.00 3.00
paco=(2,4,6)
2.00 0.00 0.00 4.00 6.00
hugo
1.00 0.00 0.00 2.00 3.00
paco
2.00 0.00 0.00 4.00 6.00
hugo + paco
3.00 0.00 0.00 6.00 9.00
paco - hugo
1.00 0.00 0.00 2.00 3.00
```

Con lo cual se puede observar que el programa es capaz de guardar variables y de realizar las acciones semánticas asociadas a las operaciones básicas con éstas mismas.

Conclusión

Realizar la traducción de la práctica 3 a esta me costó poco trabajo puesto que en HOC4 ya se proveé de la implementación base de la máquina virtual de pila y por lo tanto sólamente tuve que cuidar que las operaciones que requerían usar el tipo de dato de vector no tuviera problemas.

Uno de los retos importantes aquí fue poder usar escalares y vectores sin que "chocaran" entre sí al momento de usar las partes correspondientes de la unión. Si bien, al final resultó que era mucho más fácil de lo que pensaba, al inicio tuve que implementar varias cosas desde cero puesto que no entendía porque no estaban funcionando.

El uso de una máquina virtual de pila me ayudó a entender mejor cómo es el proceso de poder manejar de forma más flexible la memoria de la computadora desde la perspectiva de un compilador.

– *Humberto Alejandro Ortega Alcocer*.