Instituto Politécnico Nacional

Escuela Superior de Cómputo

Compiladores

Roberto Tecla Parra

Práctica 3

Tabla de Símbolos

3CM16

Humberto Alejandro Ortega Alcocer (2016630495)

24 de Mayo del 2023

Introducción

El objetivo de esta práctica es implementar la tabla de símbolos correspondiente a HOC pero en nuestro programa que es una calculadora de vectores. Para adaptar correctamente el código primero modificamos la gramática en YACC y luego realizamos adecuaciones en las acciones gramaticales a fin de que funcionaran correctamente.

Modificaciones en YACC

Lo primero que modificamos en YACC fueron los tipos de la pila de YACC mediante la unión así como la definición de los tokens y tipos con los que vamos a trabajar:

```
// Unión con los tipos de datos para la pila de YACC.
%union{
    Vector *vector;
    double numero;
    Symbol *sym;
}

// Definición de tokens.
%token <numero> NUMBER // Número.
%token <sym> VAR INDEF // Variable.

// Defincón de tipos.
%type <vector> exp vector component asgn
%type <numero> escalar
```

En YACC realizamos modificaciones en términos de la gramática correspondiente a los tipos para vectores y escalares, así como las asignaciones de las acciones gramaticales correspondientes.

```
// Lista de entradas (entrada)
list:
  | list          '\n'                          // Cuando es solo un enter.
  | list exp      '\n'  { imprimeVector($2); }    // Cuando es una expresión vectorial.
  | list asgn     '\n'  { imprimeVector($2); }    // Cuando es una asignación vectorial a = (1,2)
  | list escalar  '\n'  { printf("%.2f\n", $2); } // Cuando es una expresión escalar 1.2
  ;

// Asignación de variables.
asgn:
  | VAR '=' exp {$$ = $1->u.val = $3; $1->type=VAR; }
  ;

// Operaciones con escalares.
escalar:
  | exp '*' exp                {$$ = productoPunto($1, $3);}
  | '|'exp'|'                  {$$ = magnitud($2);}
  | NUMBER                           {$$ = $1;}
  | '-'NUMBER %prec UNARYMINUS  {$$ = -$2;}
  ;

// Evaluación de expresiones.
exp:
  | vector          { $$ = $1;  }
  | VAR             {  if($1->type == INDEF)execerror("Variable no definida, ", $1->name);
                       $$=$1->u.val;
                    }
  | asgn            { $$ = $1;  }
  | exp '+' exp     { $$ = sumaVector  ($1, $3);  }
  | exp '-' exp     { $$ = restaVector ($1, $3);  }
```

```
  | exp 'X' exp     { $$ = productoCruz($1, $3);  }
  | '('exp')'       { $$ = $2; }
  | escalar '*' exp { $$ = productoEscalar($1, $3);  }
  | exp '*' escalar { $$ = productoEscalar($3, $1);  }
;

// Definición del vector.
vector:
  | '('component')' { $$ = $2;}
  ;

// Lista de números adentro del vector.
component: NUMBER            { $$ = creaVector(3, $1); }
  | component','NUMBER { $$ = unirVectores($1, $3);}
  ;
```

Acciones Gramaticales

Las modificaciones que realizamos en las acciones gramaticales fueron en varios puntos puesto que HOC trabaja con elementos de tipo *double* y en nuestra versión trabajamos con vectores, aquí un ejemplo de una función de operación suma modificada:

```
/**
 * @brief Realiza la suma de dos vectores.
 *
 * @param a El primer vector.
 * @param b El segundo vector.
 * @return Vector* El vector resultante.
 */
Vector *sumaVector(Vector *a, Vector *b)
{
  // Variables locales.
  Vector *c; // Vector para almacenar el resultado.
  int i;     // Iterador.

  // Creamos el vector para almacenar el resultado.
  c = creaVector(a->n, 0);
  // Iteramos sobre los elementos de los vectores.
  for (i = 0; i < a->n; i++)
  {
    // Realizamos la suma.
    c->vec[i] = a->vec[i] + b->vec[i];
  }

  // Regresamos el vector con el resultado.
  return c;
}
```

La función *install* se encarga de añadir elementos a la tabla de símbolos:

```
/**
 * @brief Función que instala un símbolo en la tabla de símbolos.
 *
 * @param s El nombre del símbolo.
 * @param t El tipo del símbolo.
 * @param d El Vector del símbolo.
 * @return Symbol* Un apuntador al símbolo instalado en la tabla de símbolos.
 */
Symbol *install(char *s, int t, Vector *d)
{
  // Variables locales.
  Symbol *sp; // El símbolo a instalar.

  // Asignamos memoria para el símbolo.
  sp = (Symbol *)malloc(sizeof(Symbol));

  // Asignamos la memoria para el nombre del símbolo.
  sp->name = (char *)malloc(strlen(s) + 1);

  // Copiamos el nombre en el nombre del símbolo.
  strcpy(sp->name, s);

  // Asignamos los valores al símbolo.
```

```
  sp->type = t;
  sp->u.val = d;
  sp->next = symlist; // Lo colocamos al inicio de la lista simplemente ligada.
  symlist = sp;

  // Regresamos el símbolo.
  return sp;
}
```

Y la función *lookup* se utiliza para buscar un elemento dentro de la tabla de símbolos:

```
/**
 * @brief Función que busca un símbolo en la tabla de símbolos.
 *
 * @param s El nombre del símbolo a buscar.
 * @return Symbol* El símbolo encontrado (o NULL, 0).
 */
Symbol *lookup(char *s)
{
  // Variables locales.
  Symbol *sp; // Iterador para la tabla de símbolos.

  // Iteramos sobre la tabla de símbolos.
  for (sp = symlist; sp != (Symbol *)0; sp = sp->next)
  {
    // Validamos si el nombre del símbolo es igual al que buscamos.
    if (strcmp(sp->name, s) == 0)
    {
      // Si lo encontramos, lo regresamos.
      return sp;
    }
  }

  // Si no lo encontramos, regresamos 0 (NULL).
  return 0;
}
```

Otra modificación necesaria fue en la estructura de *symbol* que corresponde a los elementos que estarán presentes en la tabla de símbolos:

```
// Estructura para una entrada en la tabla de símbolos.
typedef struct Symbol
{
  char *name; // Nombre del símbolo.
  short type; // Tipo del símbolo: VAR, BLTIN, UNDEF.
  union
  {
    Vector *val;       // Si es VAR para almacenar el vector asociado a la variable.
    double (*ptr)();   // Si es BLTIN para almacenar la función asociada.
  } u;                 // Unión para almacenar el valor o la función.
  struct Symbol *next; // Apuntador al siguiente elemento en la tabla de símbolos.
} Symbol;
```

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:

```
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

El realizar esta práctica me ayudó a entender de mejor manera cómo funciona la tabla de símbolos y cómo está implementada. Al adaptar varias de las funciones que se proveen en HOC para que puedan trabajar con vectores es muy fácil entender más claramente cómo es que se está llevando a cabo la implementación y cuáles son sus limitantes.

Me costó mucho trabajo lograr que funcionara correctamente al principio puesto que estaba haciendo mal la asignación de los apuntadores para las cadenas asociadas a los nombres de las variables en la tabla de símbolos pero, después de un rato viendo que el problema es que se estaban guardando con nombre incompleto, fue fácil de solucionar.

– Humberto Alejandro Ortega Alcocer.