



Instituto Politécnico Nacional
Escuela Superior de Cómputo
Departamento de Ciencias e Ingeniería
de la Computación



Compiladores

Práctica 7: Funciones y procedimientos HOC6

Opción: Calculadora de vectores

Grupo: 5CM2

Alumno:

Sandoval Hernández Eduardo

Profesor:

Tecla Parra Roberto

Fecha de entrega:

11 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 7 con el tema relacionado a HOC6, para esta práctica se reutilizó el código de la práctica 6, manteniendo en la esencia del código pero con cambios bastante significativos, todo para que la calculadora pueda funcionar de manera más óptima con funciones y procedimientos.

Desarrollo

Como ya se mencionó en la introducción, en esta práctica se realizaron bastantes cambios al código para su óptimo funcionamiento, uno de los cambios más notorios es que se separó el código en más archivos, anteriormente se tenían solo 3 archivos; `vcalc.c`, `vcalc.h` y `vcalc.h`. Sin embargo, debido a que surgieron muchos problemas relacionados con las declaraciones de las funciones, se separó el código en varios ficheros como era en un inicio el ejemplo dado en la carpeta `compipracticas`.

En esta practica se agregaron más funciones relacionadas a como se manejan las pilas, debido a que ahora también se podrán tener variables con números en lugar de solo vectores. También se agregaron más funciones de comparación ya que anteriormente solo se tenían funciones para comparar vectores, ahora se pueden comparar no solo vectores sino también escalares con escalares, vectores con escalares o escalares con vectores.

Entre el código agregado para las funciones y procedimientos se tienen variables auxiliares para el funcionamiento de las definiciones de funciones y procedimientos:

```
Inst *progbase = prog;  
int dimVector;  
int returning;
```

También se agregó una nueva estructura para almacenar ya sea el nombre de la función o procedimiento, la dirección de retorno, los argumentos y/o número de argumentos y con ello también se creo un nuevo arreglo con dicha estructura y un apuntador a la primera localidad de memoria del arreglo:

```

typedef struct Frame {
    Symbol *sp;
    Inst *retpc;
    Datum *argn;
    int nargs;
} Frame;

#define NFRAME 1000
Frame frame[NFRAME] ;
Frame *fp;

```

Se modificó la función initcode para inicializar las nuevas variables:

```

void initcode(){
    progp = progbase;
    stackp = stack;
    fp = frame;
    returning = 0;
    dimVector=0;
}

```

Al poder asignar números a variables también se tuvieron que agregar funciones similares a las utilizadas para guardar vectores en variables pero con los cambios correspondientes.

Se agregaron las funciones de HOC6: define, call, ret, funcrct, procrct, getarg, arg y argassign vistas en clase:

```

void define(Symbol *sp){
    sp->u.defn = progbase;
    progbase = progp;
}

void call(){
    Symbol *sp = (Symbol *)pc[0];
    if(fp++ >= &frame[NFRAME-1])
        execerror(sp->name, "call nested too deeply");
    fp->sp = sp;
    fp->nargs = (int)pc[1];
    fp->retpc = pc + 2;
    fp->argn = stackp - 1; /* ultimo argumento */
    execute(sp->u.defn);
    returning = 0;
}

void ret(){
    int i;
    for (i = 0; i < fp->nargs; i++)
        pop();
    pc = (Inst *)fp->retpc;
    --fp;
    returning = 1;
}

void funcrct(){
    Datum d;
    if (fp->sp->type == PROCEDURE)
        execerror(fp->sp->name, "(proc) returns value");
    d = pop();
    ret();
    push(d);
}

void procrct(){
    if(fp->sp->type == FUNCTION)
        execerror(fp->sp->name, "(func) returns no value");
    ret();
}

Vector **getarg(){
    int nargs = (int) *pc++;
    if (nargs > fp->nargs)
        execerror(fp->sp->name, "not enough arguments");
    return &fp->argn[nargs - fp->nargs].vect;
}

void arg(){
    Datum d;
    d.vect = *getarg();
    push(d);
}

void argassign(){
    Datum d;
    d = pop();
    push(d);
    *getarg() = d.vect;
}

```

Cabe recordar que también es necesario modificar el arreglo keywords para que se agreguen las palabras reservadas “proc”, “func” y “return” a la tabla de símbolos:

```
static struct {
    char *name;
    int kval;
} keywords[] = {
    "if", IF,
    "else", ELSE,
    "while", WHILE,
    "print", PRINT,
    "for", FOR,
    "proc", PROC,
    "func", FUNC,
    "return", RETURN,
    0, 0,
};
```

El siguiente cambio más notorio se encuentra en la gramática ya que se modificó gran parte de esta debido a las funciones extra de comparación agregadas y las producciones para las funciones y procedimientos.

Para la pila de Yacc se agregó un campo entero el cual servirá para el número de argumentos de las funciones y procedimientos:

```
%union{
    Symbol *sym;
    Inst *inst;
    int narg;
}
```

En cuanto a los símbolos terminales y no terminales, se modificaron varios de ellos y para las funciones y procedimientos se agregaron 6 terminales nuevos: FUNCTION, PROCEDURE, RETURN, FUNC, PROC y ARG. Además, se agregaron 2 producciones nuevas específicamente para las funciones y procedimientos: procname y arglist:

```

%token<sym> NUMBER PRINT VAR VARVECTOR VARESCALAR INDEF WHILE IF
ELSE FOR
%token<sym>    FUNCTION PROCEDURE RETURN FUNC PROC
%token<narg>   ARG
%type<inst> stmt asgnVector asgnEscalar expVectorial expEscalar
stmtlist cond while if end for condfor stmtfor
%type<inst> begin
%type<sym>   procname
%type<narg>  arglist

```

Los operadores se mantuvieron casi igual con excepción de que se eliminó el ‘.’
 Para el producto punto entre vectores ya que con ayuda de las nuevas producciones se podrá hacer uso del operador “*” sin que ocurra alguna malinterpretación.

Al símbolo inicial se le agregó una nueva producción para el ingreso consecutivo de las declaraciones de funciones y procedimientos

```

list:
| list '\n'
| list asgnVector '\n' { code2(printVector, STOP); return 1; }
| list asgnEscalar '\n' { code2(printDouble, STOP); return 1; }
| list expVectorial '\n' { code2(printVector, STOP); return 1; }
| list expEscalar '\n' { code2(printDouble, STOP); return 1; }
| list defn '\n'
| list stmt '\n' { code(STOP); return 1; }
;

```

Las producción defn se agregó para las funciones y procedimientos ya que con esta se pueden definir, se apoya en la producción procname para que se diferencie entre una función y un procedimiento, y finalmente la producción arglist sirve para dar los argumentos necesarios para las funciones y procedimientos así lo desee el usuario.

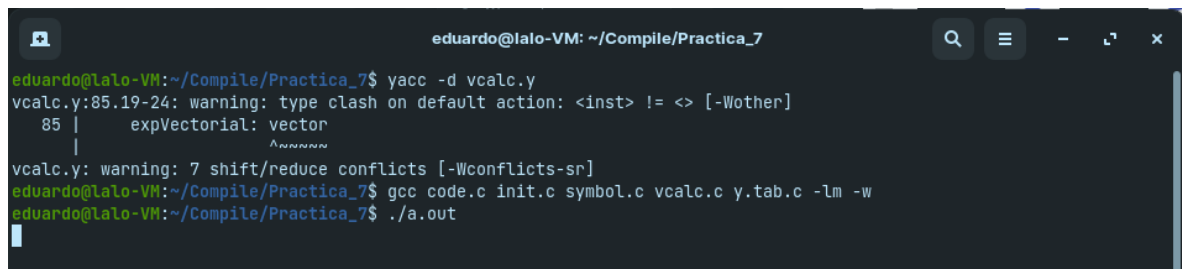
Para el código de apoyo, se modificaron varias cosas, pero la más destacable es la modificación en yylex para reconocer los argumentos de las funciones y procedimientos (\$):

```

if(c == '$'){ // Para argumentos
    int n = 0;
    while(isdigit(c=getc(fin)))
        n = 10 * n + c - '0';
    ungetc(c,fin);
    if(n == 0)
        execerror("strange $...", (char *)0);
    yylval.narg = n;
    return ARG;
}

```

Para la compilación y ejecución de la práctica se utilizan los siguientes comandos:



```

eduardo@lalo-VM: ~/Compile/Practica_7
eduardo@lalo-VM:~/Compile/Practica_7$ yacc -d vcalc.y
vcalc.y:85.19-24: warning: type clash on default action: <inst> != <> [-Wother]
85 |     expVectorial: vector
    |     ^~~~~~
vcalc.y: warning: 7 shift/reduce conflicts [-Wconflicts-sr]
eduardo@lalo-VM:~/Compile/Practica_7$ gcc code.c init.c symbol.c vcalc.c y.tab.c -lm -w
eduardo@lalo-VM:~/Compile/Practica_7$ ./a.out

```

Para probar el funcionamiento del programa, se declarará un procedimiento el cual sus parámetros serán el statement inicial, la condición y el statement final de un ciclo for el cual imprimirá en pantalla los cambios que sufra una variable en cada iteración. En el caso de las funciones, se realizará una función que permita calcular la factorial de un número y para mostrar el retorno de la función, el resultado se podrá almacenar en una variable, para este ejemplo se calculará la factorial de 4 y el de 10:


```
eduardo@lalo-VM: ~/Compile/Practica_7
vcalc.y:85.19-24: warning: type clash on default action: <inst> != <> [-Wother]
85 |     expVectorial: vector
    |     ^~~~~~
vcalc.y: warning: 7 shift/reduce conflicts [-Wconflicts-sr]
eduardo@lalo-VM:~/Compile/Practica_7$ gcc code.c init.c symbol.c vcalc.c y.tab.c -lm -w
eduardo@lalo-VM:~/Compile/Practica_7$ ./a.out
proc contador(){for(i=$1;i<=$2;i=i+$3){print i}}
contador([1],[20],[2])
Resultado: [ 1.000000 ]
Resultado: [ 3.000000 ]
Resultado: [ 5.000000 ]
Resultado: [ 7.000000 ]
Resultado: [ 9.000000 ]
Resultado: [ 11.000000 ]
Resultado: [ 13.000000 ]
Resultado: [ 15.000000 ]
Resultado: [ 17.000000 ]
Resultado: [ 19.000000 ]
func factorial(){for(i=[1];i<=[1];i=i+[1]){res = [1]} for(i=[1];i<=$1;i=i+[1]){res=res*[i]} return res}
factorialdecuatro = factorial([4])
Resultado: [ 24.000000 ]
factorialdecuatro
Resultado: [ 24.000000 ]
factorialdediez = factorial([10])
Resultado: [ 3628800.000000 ]
factorialdediez
Resultado: [ 3628800.000000 ]
```

Como se puede apreciar, el procedimiento y la función operan de manera adecuada, a pesar de que la implementación de la función factorial no es la más eficiente, es lo mejor que se pudo hacer debido a que con otros intentos resultaba en un error desconocido probablemente causado por un apuntador salvaje, sin embargo, se logró hacer una función que cumple con su propósito.

Conclusión

Esta práctica final ha resultado ser la más laboriosa debido a que se tuvieron que agregar más funciones a causa de las nuevas producciones y los intentos fallidos de realizar una función para obtener la factorial de manera eficiente, pero a pesar de las complicaciones ha sido muy útil para tener una idea de lo que se necesita para el proyecto final, y como todo, aún se podrá mejorar en un futuro.

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>