Universidad Nacional de Ingeniería

CC0E2 Compiladores

Trabajo de Investigación

Profesor:
Jaime Osorio Ubaldo
Departamento de Ciencias de la Computación

Alumno:
Davis Keenyo Alderete Valencia

Índice general

Resumen

Your abstract.

Capítulo 1

Introducción

1.1. Problemática

Una calculadora es el uso básico y más importante de los estudiantes diarios. Es complicado realizar operaciones extensas ya sean simples o cortas en un pantalla muy pequeña como lo es de una calculadora, la sintaxis no siempre se llega a comprender al 100 por ciento, debido a las diferentes que expresiones que se utilizan para representar una operación matemática en específico.

1.2. Objetivos

1.2.1. Objetivo General

Establecer una sintaxis adecuada para realizar operaciones matemáticas simples y complejas.

1.2.2. Objetivos Especificos

1. Establecer una sintaxis para poder operar con funciones trigonométricas.
2. Establecer una sintaxis para poder operar con funciones logarítmicas.
3. Realizar calculos con operaciones simples y complejas.

1.3. Justificación y viabilidad

Si bien es cierto de puede realizar operaciones matemáticas desde simples hasta complejas en una calculadora científica física. El hecho de contar con una calculadora en la pc o una laptop seria consecuencia de instalar una aplicación calculadora, esta dispone del almacenamiento de un dispositivo, el hecho de contar con una interfaz gráfica hace que se consuma más recursos. Además, que la sintaxis de esta a veces es confusa, así como la limitación para mostrar mensajes. Mientras el tamaño de una terminal se puede modificar de tamaño (agrandar o achicar), este aspecto nos permite visualizar mejor las operaciones que cuenten con una gran cantidad de términos, mensajes largos acerca de algún requerimiento o error. Gracias a los compiladores se pueden establecer una sintaxis simple y entendible para que el uso de estas sea de uso general.

1.4. Nombre del compilador

El nombre pensado para este proyecto es **easyCalculator**.

1.5. Resultados esperados

- Se espera tener una sintaxis de input fácil e intuible para el usuario.
- Mostrar resultados y mensajes de cualquier tipo de manera entendible .

Capítulo 2

Estado del arte

2.1. Simple calculator compiler using Lex and YACC

En [3], el autor se centra en el desarrollo de un compilador simple usando LEX (Lexical Analyzer Generator) y YACC (Yet Another Compiler-Compiler) para construir una calculadora simple. Nos explica como funciona LEX, sus funcionalidades y sintaxis, de la misma manera como funciona YACC . En el diseño de un compilador para una calculadora simple, los autores toman la funcionalidad básica que es sumar y restar, asi que definen un token **INTEGER**. Yacc genera un analizador en el archivo y.tab.c y un archivo de inclusión y.tab.c. Lex incluye este archivo y utliza las definiciones para los valores de tokens.Para obtener los tokens, yacc llama a **yylex**. La función **yylex** tiene un tipo de retorno de int que devuelve un token, los valores asociados con el token son devueltos por lex en la variable **yylval**. La sección de reglas se asemeja a la gramática BNF (Backus Naur Form). Con la herramienta **lex** y **yacc** uno puede crear su propio compilador. Se trata básicamente de herramientas de compilación de lenguaje procedimental, y para admitir la orientación a objetos, es necesario trabajar del lenguaje C para admitir la orientación a objetos, lo que hace que el compilador sea bastante complejo. Usar lex y yacc en **UNIX** es fácil en comparación de otros sistemas operativos.

2.2. Lexical Analyzer Generator (LEX)

El trabajo realizado acerca de LEX [2] por **M. E. Lex y E. Schmidt** menciona que Lex es un generador de programas diseñado para el procesamiento léxico de flujo de entradas a caracteres. Acepta una especificación orientada a problemas de alto nivel para la coincidencia de cadenas de caracteres y produce un programa en un lenguaje de propósito general que reconoce expresiones regulares. Las expresiones regulares son especificadas por el usuario en las especificaciones de origen proporciones a Lex. El código escrito de Lex reconoce estas expresiones en un flujo de entrada y divide el flujo en entrada en cadenas que coinciden con las expresiones. En los límites entre las cadenas se ejecutan las secciones del programa proporcionadas por el usuario. El archivo fuente de Lex asocia las expresiones regulares y los fragmentos de un programa. A medida que cada expresión aparece en la entrada de un programa excrito por lex, se ejecuta el fragmento correspondiente.

2.3. Yet Another Compiler-Compiler (YACC)

Los autores **S. C. Jhonson y Murray Hill** en su trabajo [1], proveen y describen una herrmienta general que impone un tipo de input para un programa de computadora. El usuario YACC puede preparar una especificación para el input, este incluye reglas que describen la estructura del input, código que se invocará cuando se conozcan estas reglas, y una rutina de bajo nivel para realizar una entrada básica. Yacc luego genera una función para controlar el proceso de entrada. Esta función llamada analizador, llama a la rutina de entrada de bajo nivel proporcionada por el usuario (el analizador léxico) para recoger los elementos básicos (llamados tokens) del flujo de entrada. Estos tokens se organizan de acuerdo con las reglas de estructura de entrada, llamadas gramaticales; cuando se reconoce una de estas reglas, se invoca el código de usuario proporcionado para esta regla, un acción; las acciones tiene la capacidad de devolver valores y hacer uso de los valores de otras acciones.

Capítulo 3

Metodología

3.1. Herramientas, Métodos y procedimientos

La herramienta Lex ayuda a escribir programas cuyo flujo de control está dirigido por instancias de expresiones regulares en el flujo de entrada. Es muy adecuado para transformaciones de tipo editor-scripts y para segmentar entradas en preparación para una rutina de análisis. La fuente de la herramienta Lex es la tabla de expresiones regulares y los fragmentos de programa correspondientes. La tabla se traduce a un programa que lee un flujo de entrada, lo copia en un flujo de salida y divide la entrada en cadenas que coinciden con las expresiones dadas. Por otro lado, la herramienta YACC recibe información de la gramática del usuario. Partiendo de esta gramática, genera el código fuente C para el analizador. YACC invoca a Lex para escanear el código fuente y usa los tokens devueltos por Lex para construir un árbol de sintaxis. Con la ayuda de la herramienta YACC y Lex, uno puede escribir su propio compilador.

3.2. Delimitación del proyecto

Se a implementado una calculadora científica que puede realizar las operaciones que se enumeran a continuación:

- Operaciones básicas: **Suma**, **Resta**, **División** y **Multiplicación**.
- Funciones trigonométricas: **seno**, **cos** y **tan**.
- Funciones logarítmicas: **log** y **logaritmo natural(ln)**.
- La expresión se puede encerrar entre corchetes.

3.3. Tokens

Como en la calculadora se van a hacer operaciones que van a incluir delimitadores como llaves y corchetes, los tokens tienen incluir caracteristicas de operaciones básicas como suma, resta, multiplicación y división. Los siguientes tokens se establecen para el uso del programa:

- NUM, que representa a un número real.
- SUMA, será representada por el simbolo "+".

- RESTA, que presenta al simbolo "$-$".
- MUL, que presenta al simbolo "$*$".
- PAR_LEFT, que es representado por el simbolo "(".
- PAR_RIGHT, que es representado por el simbolo ")".
- COR_LEFT, que es representado por el simbolo "[".
- COR_RIGHT, que es representado por el simbolo "]".
- SIN, función trigonometrica **seno**, representada por la sintaxis "*sin*".
- COS, función trigonometrica **coseno**, representada por la sintaxis "*cos*".
- TAN, función trigonometrica **tangente**, representada por la sintaxis "*tan*".
- LOG, función **logaritmo**, representada por la sintaxis "*log*".
- NLOG, función **logaritmo natural**, representada por la sintaxis "*nlog*".

3.4. Patrones

Patrón	Lexema	Token
Números seguido de número (real o entero)	3.14,-3.5	REAL
Simbolo Suma	+	MAS
Simbolo Resta	-	RESTA
Simbolo Multiplicación	*	MUL
Simbolo División	/	DIV
Función Seno	sin	SIN
Función Coseno	cos	COS
Función Tangente	tan	TAN
Función Logaritmo base 10	log	LOG
Función Logaritmo natural	nlog	NLOG
Apertura de agrupación de alguna operación matematica	(operación	PAR_LEFT
Cierre de agrupación de alguna operación matematica	operación)	PAR_RIGHT
Apertura de agrupación de una matriz	[[4.2,5,..],[...],..	COR_RIGHT
Cierre de agrupación de una matriz	[4.2,5,..],[...],...]	COR_RIGHT

3.4.1. Prueba de Tokens

3.5. Gramática

La declaración de la gramática se hizo considerando al procedencia de operadores y los posibles conflictos que suelen ocurrir, por lo tanto no existe ambiguedad en la gramática construida para este programa.

Figura 3.1: Toma 1

Figura 3.2: Toma 2

```
%%
prog: expr '\n'    prog { printf("Resultado: %g\n",$1);   };
prog:  ;

expr: term MAS expr      { $$ = $1 + $3; }
        | term MENOS  expr       { $$ = $1 - $3; }
        | term
        |op

        ;
term: term MUL fun {$$=$1 * $3;}
        | term DIV fun {$$=$1 / $3;}
        | '(' term ')'
        |fun
        ;


fun: SIN '(' expr ')' {$$=sin($3);}
   | COS '(' expr ')' {$$=cos($3);}
   | TAN '(' expr ')' {$$=tan($3);}
   | LOG '(' expr ')' {$$=log10($3);}
   | NLOG '(' expr ')' {$$=log($3);}
   | SQRT '(' expr ')' {$$=sqrt($3);}
```

```
    | POW '(' expr ',' expr ')' {$$=pow($3,$5);}
    | SINH '(' expr ')' {$$=sinh($3);}
    | COSH '(' expr ')' {$$=cosh($3);}
    | TANH '(' expr ')' {$$=tanh($3);}
    | ASIN '(' expr ')' {$$=asin($3);}
    | ACOS '(' expr ')' {$$=acos($3);}
    | ATAN '(' expr ')' {$$=atan($3);}
    | ABS '(' expr ')' {$$=abs($3);}
    | SUM '(' expr ',' expr ')' NUMBER {$$=sumatoria($3,$5,$7);}
    | NUMBER '!' {$$=factorial($1);}
    | NUMBER {$$=$1;}
    ;

op: M '+' M
  | M '*' M
  | M '-' M
  | M
  ;
M: '[' matrix_list ']'
    ;

matrix_list: matrix
           | matrix_list ',' matrix
           ;

matrix: '[' row_list ']'
      ;

row_list: row
        | row_list ',' row
        ;

row: NUMBER {$$=$1;}
   ;
%%
```

3.6. Método Shift-Reduce

Vamos a comprobar la aceptación de una operación de entrada para nuestra calculadora. A continuación se muestra la operación planteada y el procedimiento shift-reduce.

ESTADOS	sin(pi/2)*2+4*pow(2,4)
0	SIN (NUM) MUL NUM MAS NUM MUL POW (NUM , NUM)
03	(NUM) MUL NUM MAS NUM MUL POW (NUM , NUM)
03(36)	NUM) MUL NUM MAS NUM MUL POW (NUM , NUM)

```
03(36)1                          ) MUL NUM MAS NUM MUL POW ( NUM , NUM )
03(36)                                ) MUL NUM MAS NUM MUL POW ( NUM , NUM )
03(36)(66)                            ) MUL NUM MAS NUM MUL POW ( NUM , NUM )
03(36)(66)(97)                    MUL NUM MAS NUM MUL POW ( NUM , NUM )
0                                       MUL NUM MAS NUM MUL POW ( NUM , NUM )
0(24)                                 MUL NUM MAS NUM MUL POW ( NUM , NUM )
0(24)(59)                             NUM MAS NUM MUL POW ( NUM , NUM )
0(24)(59)1                            MAS NUM MUL POW (NUM , NUM )
0(24)(59)                             MAS NUM MUL POW (NUM , NUM )
0(24)(59)(90)                       MAS NUM MUL POW (NUM , NUM )
0                                       MAS NUM MUL POW (NUM , NUM )
0(22)                                 MAS NUM MUL POW (NUM , NUM )
0(22)(56)                             NUM MUL POW (NUM , NUM )
0(22)(56)1                            MUL POW ( NUM , NUM )
0(22)(56)                             MUL POW ( NUM , NUM )
0(22)(56)(87)                       MUL POW ( NUM , NUM )
0(22)(56)(87)(59)                   POW ( NUM , NUM )
0(22)(56)(87)(59)9                   ( NUM , NUM )
0(22)(56)(87)(59)9(42)      NUM , NUM )
0(22)(56)(87)(59)9(42)1      , NUM )
0(22)(56)(87)(59)9(42)        , NUM  )
0(22)(56)(87)(59)9(42)(72)        , NUM  )
0(22)(56)(87)(59)9(42)(72)(107)  NUM )
0(22)(56)(87)(59)9(42)(72)(107)1          )
```

0(22)(56)(87)(59)9(42)(72)(107))

0(22)(56)(87)(59)9(42)(72)(107)(123))

0(22)(56)(87)(59)9(42)(72)(107)(123)(130) $

0(22)(56)(87)(59) $

0(22)(56)(87)(59)(90) $

0(22)(56) $

0(22)(56)(87) $

0 $

ACEPTADO

3.7. Tabla de simbolos

Es una estructura de datos importante creada y mantenida por el compilador para realizar un seguimiento de la semática de las variables. es decir, almacena información sobre el alcance e información vinculante sobre nombres, información sobre nombres, información sobre instancias de varias entidades, como nombres de variables y funciones, clases, objetos, etc. La tabla de simbolos para una operación se muestra un ejemplo en la Figura 3.3.

```
(base) test01@pcs5:~/compi$ conda deactivate
test01@pcs5:~/compi$ bison pr1.y
test01@pcs5:~/compi$ gcc pr1.tab.c -lm
test01@pcs5:~/compi$ ./a.out
4-6*2+15/3+(5-1)
tabla de simbolos
0 nombre=  token=258  value=4.000000
1 nombre=  token=258  value=6.000000
2 nombre=  token=258  value=2.000000
3 nombre=  token=258  value=15.000000
4 nombre=  token=258  value=3.000000
5 nombre=  token=258  value=5.000000
6 nombre=  token=258  value=1.000000
Resultado: 1
test01@pcs5:~/compi$
```

Figura 3.3: Tabla de simbolos para una operación.

3.8. Tabla de Códigos

En la Figura 3.4, se muestra la tabla de códigos.

Figura 3.4: Tabla de codigos.

3.9. Interface gráfica Calculadora

Se desarrolló una interfaz gráfica para manejar realizar calculos, cualquier usuario puede utilizar la interfaz ya que esta es intuitiva y fácil de utilizar, las operaciones introducidas en la calculadora tambien se calculan en el terminal de fondo que esta ejecutando la calculadora, en este terminal de fondo se muestran la tabla de códigos y la tabla de simbolos, la interface se muestra en la Figura 3.5.

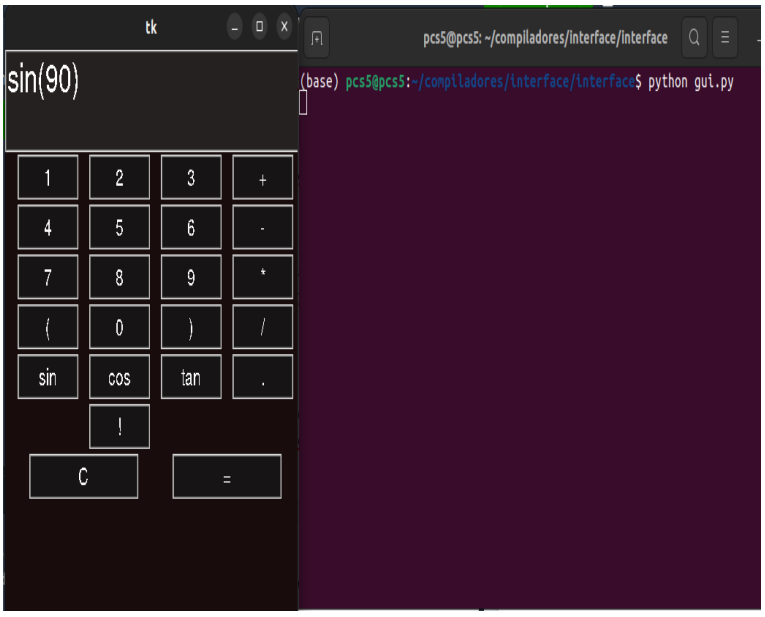

Figura 3.5: GUI de la calculadora.

Capítulo 4

Resultados

4.1. EasyCalculator vs Simple Calculator compiler [3]

La construcción de Simple Calculator compiler se puede observar en las siguientes Figuras 4.1 y 4.2.

```
            #include "calculator.h"
            #include "y.tab.h"
            void yyerror(char *);
%}

%%
[a-z]       {
              yylval.sIndex = *yytext - 'a';
              return VARIABLE;
            }
0           {
              yylval.iValue = atoi(yytext);
              return INTEGER;
            }

[1-9][0-9]*         {
                              yylval.iValue = atoi(yytext);
                              return INTEGER;
                    }

[-()<>=+*/;{}.]     {
                              return *yytext;
                    }
">="            return GE;
"<="            return LE;
"=="            return EQ;
"!="            return NE;
"while"         return WHILE;
"if"            return IF;
"else"          return ELSE;
"print"         return PRINT;

[ \t\n]+        ;       /* ignore whitespace */
.               yyerror("Unknown character");
%%
int yywrap(void) {
    return 1;
}
```

Figura 4.1: Bloque Lex de Simple Calculator compiler

- El uso de la sintaxis para la costrucción de una calculadora en este trabajo es más sencilla y fácil de entender.

```
nodeType *opr(int oper, int nops, ...);
nodeType *id(int i);
nodeType *con(int value);
void freeNode(nodeType *p);
int ex(nodeType *p);
int yylex(void);
void yyerror(char *s);
int sym[26];                    /* symbol table */
%}
%union {
    int iValue;                 /* integer value */
    char sIndex;                /* symbol table index */
    nodeType *nPtr;             /* node pointer */
};
%token <iValue> INTEGER
%token <sIndex> VARIABLE
%token WHILE IF PRINT
%nonassoc IFX
%nonassoc ELSE
%left GE LE EQ NE '>' '<'
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS
%type <nPtr> stmt expr stmt_list

%%
program:
        function                { exit(0); }
        ;
function:
          function stmt         { ex($2); freeNode($2); }
        | /* NULL */
        ;
stmt:
          ';'                   { $$ = opr(';', 2, NULL, NULL); }
        | expr ';'              { $$ = $1; }
        | PRINT expr ';'        { $$ = opr(PRINT, 1, $2); }
        | VARIABLE '=' expr ';'         { $$ = opr('=', 2, id($1), $3); }
        | WHILE '(' expr ')' stmt       { $$ = opr(WHILE, 2, $3, $5); }
        | IF '(' expr ')' stmt %prec IFX { $$ = opr(IF, 2, $3, $5); }
        | IF '(' expr ')' stmt ELSE stmt { $$ = opr(IF, 3, $3, $5, $7); }
        | '{' stmt_list '}'             { $$ = $2; }
        ;
stmt_list:
          stmt                  { $$ = $1; }
        | stmt_list stmt        { $$ = opr(';', 2, $1, $2); }
        ;
expr:
          INTEGER               { $$ = con($1); }
        | VARIABLE              { $$ = id($1); }
        | '-' expr %prec UMINUS { $$ = opr(UMINUS, 1, $2); }
        | expr '+' expr         { $$ = opr('+', 2, $1, $3); }
        | expr '-' expr         { $$ = opr('-', 2, $1, $3); }
        | expr '*' expr         { $$ = opr('*', 2, $1, $3); }
        | expr '/' expr         { $$ = opr('/', 2, $1, $3); }
        | expr '<' expr         { $$ = opr('<', 2, $1, $3); }
        | expr '>' expr         { $$ = opr('>', 2, $1, $3); }
        | expr GE expr          { $$ = opr(GE, 2, $1, $3); }
        | expr LE expr          { $$ = opr(LE, 2, $1, $3); }
        | expr NE expr          { $$ = opr(NE, 2, $1, $3); }
        | expr EQ expr          { $$ = opr(EQ, 2, $1, $3); }
        | '(' expr ')'          { $$ = $2; }
        ;
%%
#define SIZEOF_NODETYPE ((char *)&p->con - (char *)p)
nodeType *con(int value) {
    nodeType *p;
    size_t nodeSize;
    /* allocate node */
    nodeSize = SIZEOF_NODETYPE + sizeof(conNodeType);
    if ((p = malloc(nodeSize)) == NULL)
        yyerror("out of memory");
    /* copy information */
    p->type = typeCon;
    p->con.value = value;
    return p;
}
nodeType *id(int i) {
    nodeType *p;
    size_t nodeSize;
    /* allocate node */
    nodeSize = SIZEOF_NODETYPE + sizeof(idNodeType);
    if ((p = malloc(nodeSize)) == NULL)
        yyerror("out of memory");

    /* copy information */
    p->type = typeId;
    p->id.i = i;

    return p;
}
nodeType *opr(int oper, int nops, ...) {
    va_list ap;
    nodeType *p;
    size_t nodeSize;
    int i;

    /* allocate node */
    nodeSize = SIZEOF_NODETYPE + sizeof(oprNodeType) +
        (nops - 1) * sizeof(nodeType*);
    if ((p = malloc(nodeSize)) == NULL)
        yyerror("out of memory");

    /* copy information */
    p->type = typeOpr;
    p->opr.oper = oper;
    p->opr.nops = nops;
    va_start(ap, nops);
    for (i = 0; i < nops; i++)
        p->opr.op[i] = va_arg(ap, nodeType*);
    va_end(ap);
    return p;
}
void freeNode(nodeType *p) {
    int i;

    if (!p) return;
    if (p->type == typeOpr) {
        for (i = 0; i < p->opr.nops; i++)
            freeNode(p->opr.op[i]);
    }
    free (p);
}
```

Figura 4.2: Bloque Yacc de Simple Calculator compiler

- EasyCalculator hace uso de la gramática de manera adecuada y evitando posibles conflictos de predencia de operadores y ambiguedades.
- Simple Calculator usa sentencias propias de YACC que ayudan a eliminar las ambiguedades en la gramatica, por ejemplo, la sentencia $\%left$, que hace referencia a agrupamiento a la izquierda.
- Simple Calculator compiling usa expresiones regulares para el reconocimiento de los caracteres ingresados por teclado. Se hace uso de la función **atoi** para la conversión de una cadena de texto a un número entero si la cadena representa un número. Además hace uso de la libreria "y.tab.h.[en] el bloque LEX para hacer la conexión con el bloque YACC.
- Simple Calcualtor usa punteros y memoria dinámica para las variables y operaciones.

4.2. Árboles sintácticos de operaciones

Para mostrar árboles sintácticos que sigue la calculadora, se realizó operaciones matemáticas de prueba. Las pruebas realizadas brindaron los resultados correctos, esto nos dice que las operaciones de precedencia y las posibles ambiguedades en la gramática no suceden, lo cual también se observa al momento de compilar la calculadora (Figura). Los arboles sintacticos se muestran en las Figuras 4.3, 4.4 y 4.5.

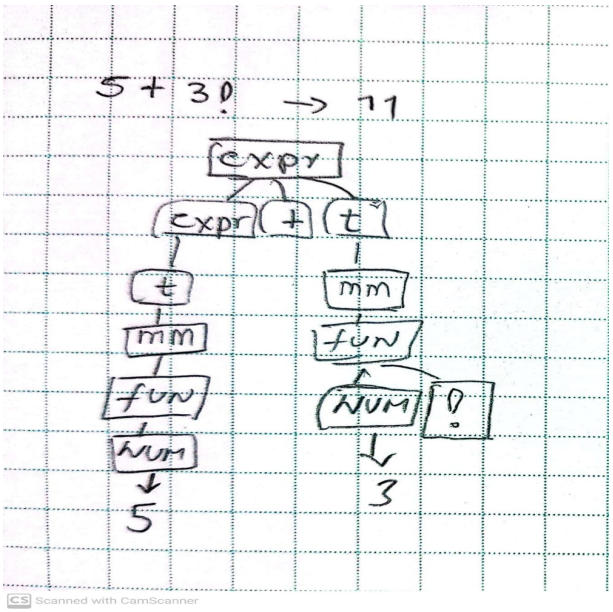

Figura 4.3: Árbol sitactico operación 1

4.3. Cálculos hechos desde la interfaz

Las Figuras de la expresión que se ingresa para hacer el cálculo y su respectiva respuesta se muestran en las Figuras 4.6 y 4.7 respectivamente.

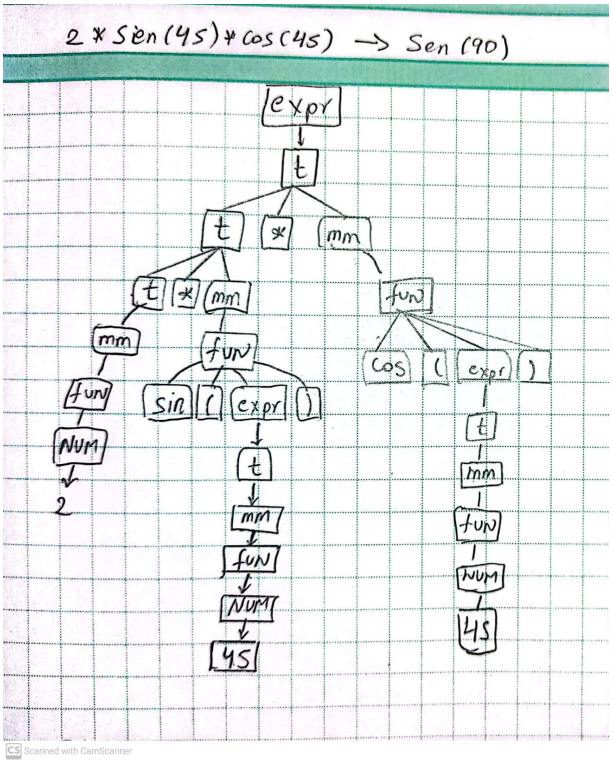

Figura 4.4: Árbol sitactico operación 2

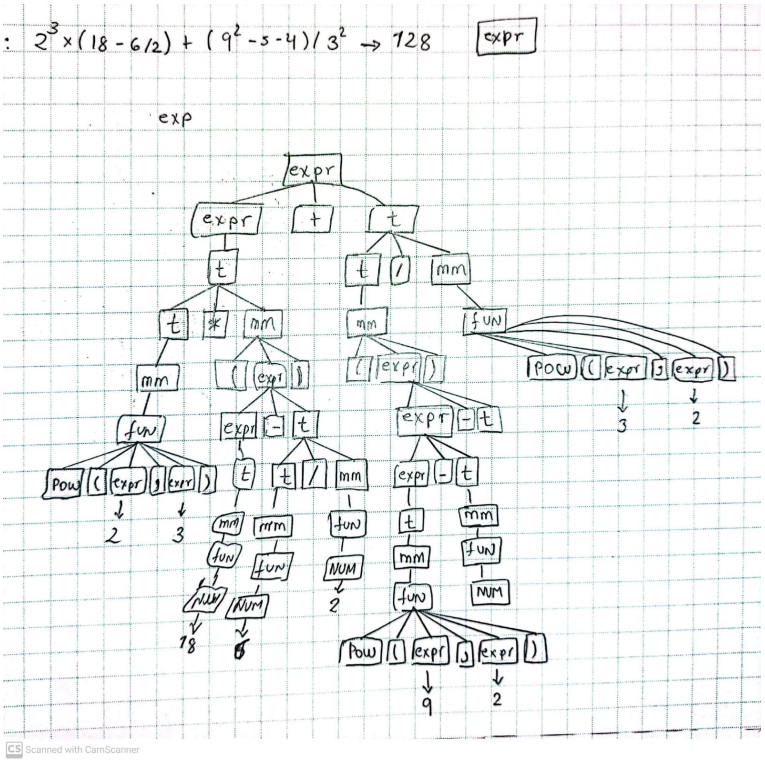

Figura 4.5: Árbol sitactico operación 3

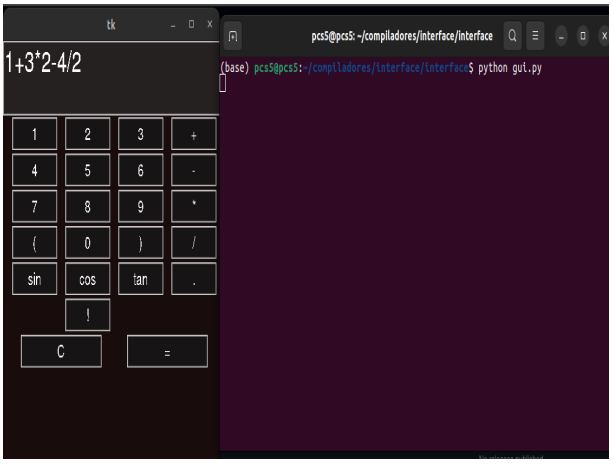

Figura 4.6: Ingresando la operación a calcular

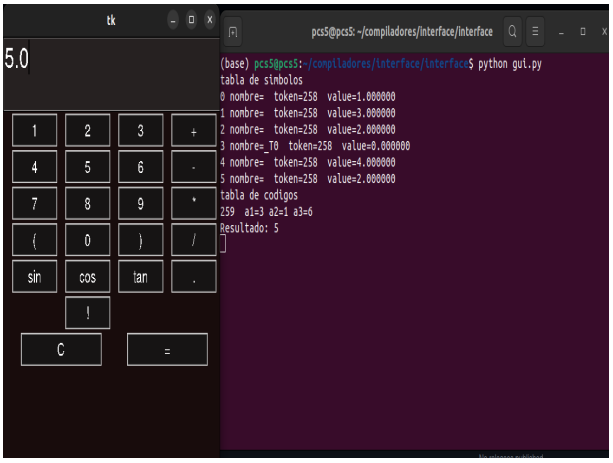

Figura 4.7: Visualizando resultados de la operación dada en 4.6

Bibliografía

[1] S. C. Johnson y Murray Hill. "Yacc: Yet Another Compiler-Compiler". En: 1978.

[2] E. Schmidt M. E. Lesk. *Lex - A Lexical Analyzer Generator*. Inf. téc. Murray Hill, New Jersey 07974: ATT Bell Laboratories, PS1:16-1 - PS1:16-12. URL: http://kjellggu.myocard.net/misc/tutorials/lex.pdf.

[3] Mohit Upadhyaya. "Simple calculator compiler using Lex and YACC". En: *2011 3rd International Conference on Electronics Computer Technology*. Vol. 6. IEEE. 2011, págs. 182-187.