CSPC62

Compiler Design

Final Project Report

Creating a compiler for our programming language *SAL*

Team Members:

106120001 - Aasish Rajju

106120009 - A Lokanush

106120017 - Ashwath Vasudevan

106120121 - Skandan R

Assignment - 1

Lexical Analyzer

09/02/2023

Contributions:

106120001 - Precedence of operators and ideation, NFA

106120009 - Regular expressions and ideation, NFA

106120017 - Regular expressions and ideation, DFA

106120121 - Symbols for keywords and ideation, DFA

Aim: Building a lexical analyzer for our programming language SAL and generating tokens

An example program in SAL language -

sample.sal

```
<he.sal> # main() {
    $ a = 6.2 + 4 + 4 + 4;
    a = a - 4;
    # s = 8 - 3;
    s = s * 3;
    '5;
}
```

lexer.l -

```
%{
    #include <stdio.h>
    int countn=0;
%}

%option yylineno

alpha [a-zA-Z]
digit [0-9]

%%

"."                         { printf("DOT\t");  }
">>"                        { printf("PRINT\t");  }
"<<"                        { printf("SCAN\t");  }
"#"                         { printf("INT\t");  }
"$"                         { printf("FLOAT\t");  }
"~"                         { printf("STRING\t");  }
"_"                         { printf("BOOL\t");  }
"'"                         { printf("RET\t");  }
"@"                           { printf("FOR\t");  }
"?"                           { printf("IF\t");  }
":"                           { printf("ELSE\t");  }
^<{alpha}({alpha}|{digit})*.sala> { printf("INCLUDE\n");  }
"True"                      { printf("T\t");  }
"False"                     { printf("F\t");  }
{digit}+                    { printf("NUM\t");  }
{digit}+\.{digit}{0,6}      { printf("REAL\t");  }
{alpha}({alpha}|{digit})*   { printf("ID\t");  }
"<="                        { printf("LE\t");  }
">="                        { printf("GE\t");  }
"=="                        { printf("EQ\t");  }
"!="                        { printf("NE\t");  }
">"                           { printf("GT\t");  }
"<"                           { printf("LT\t");  }
"!"                         { printf("NOT\t"); }
"&&"                         { printf("AND\t");  }
"||"                         { printf("OR\t");  }
"+"                         { printf("ADD\t");  }
"-"                         { printf("SUB\t");  }
```

```
"/"                         { printf("DIV\t");  }
"*"                         { printf("MULT\t");  }
"="                         { printf("ASSIGN\t"); }
"{"                         { printf("BRACES_OPEN\n");  }
"}"                         { printf("BRACES_CLOSE\n"); }
"("                         { printf("BRACKET_OPEN\t"); }
")"                         { printf("BRACKET_CLOSE\t");  }
";"                         { printf("DELIM\n"); countn++; }
\/\/.*                      { printf("COMM\n"); }
["].*["]                    { printf("SENTENCE\t"); }
[\n]                        { printf("NL\n"); }
[\t]                        { ; }
" "                         { ; }
[\r]                        { ; }
.                               { printf("ERROR\n");}


%%

int yywrap() {
  return 1;
}

/* int main(){
  FILE *myfile = fopen("sample.sal", "r");
  if (!myfile) {
    printf("Cant open the file\n");
    return -1;
  }
  yyin = myfile;
  while(yylex());
  fclose(myfile);
} */
```

Explanation of code working - The regular expressions for identifying tokens have been written in the lex file which will generate a lexical analyzer according to the rules written. The tokens are printed on the screen in the order in which they appear. The line count is kept track with the option *yylineno*.

Commands to run the lexical analyzer -

```
lex lexer.l
cc lex.yy.c -ll
./a.out
```

Output -

```
INCLUDE
INT     ID      BRACKET_OPEN        BRACKET_CLOSE       BRACES_OPEN
INT     ID      ASSIGN          NUM             DELIMITER
COMMENT
BOOL    ID      ASSIGN          SENTENCE        DELIMITER
STRING  ID      ASSIGN          SENTENCE        DELIMITER
PRINT   BRACKET_OPEN            SENTENCE        BRACKET_CLOSE       DELIMITER
BRACES_CLOSE
```

Assignment - 2

Syntax Analyzer

02/03/2023

Contributions:

106120001 - Symbol Table Construction and resolving ambiguity

106120009 - Symbol Table Construction and resolving ambiguity

106120017 - Filling the Symbol table and writing grammar rules

106120121 - Filling the Symbol table and writing grammar rules

Aim: Building a parser for our language and analyzing the syntax

Source Code -

lexer.l

```
%{
    #include <stdio.h>
    #include "y.tab.h"
    int countn=0;
%}

%option yylineno

alpha [a-zA-Z]
digit [0-9]

%%

"."                         { strcpy(yylval.nd_obj.name,(yytext));
printf("DOT\t");  return DOT; }
"<<"                        { strcpy(yylval.nd_obj.name,(yytext));
printf("PRINT\t");  return PRINT; }
">>"                        { strcpy(yylval.nd_obj.name,(yytext));
printf("SCAN\t");  return SCAN; }
"#"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("INT\t");  return INT; }
"$"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("FLOAT\t");  return FLOAT; }
"~"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("STRING\t");  return STRING; }
"_"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("BOOL\t");  return BOOL; }
"'"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("RET\t");  return RET; }
"@"                           { strcpy(yylval.nd_obj.name,(yytext));
printf("FOR\t");  return FOR; }
"?"                           { strcpy(yylval.nd_obj.name,(yytext));
printf("IF\t");  return IF; }
":"                           { strcpy(yylval.nd_obj.name,(yytext));
printf("ELSE\t");  return ELSE; }
^<{alpha}({alpha}|{digit})*.sala> { strcpy(yylval.nd_obj.name,(yytext));
printf("INCLUDE\n");  return INCLUDE; }
"True"                      { strcpy(yylval.nd_obj.name,(yytext));
printf("T\t");  return T; }
"False"                     { strcpy(yylval.nd_obj.name,(yytext));
printf("F\t");  return F; }
{digit}+                    { strcpy(yylval.nd_obj.name,(yytext));
printf("NUM\t");  return NUM; }
{digit}+\.{digit}{0,6}      { strcpy(yylval.nd_obj.name,(yytext));
printf("REAL\t");  return REAL; }
{alpha}({alpha}|{digit})*   { strcpy(yylval.nd_obj.name,(yytext));
printf("ID\t");  return ID; }
"<="                        { strcpy(yylval.nd_obj.name,(yytext));
printf("LE\t");  return LE; }
">="                        { strcpy(yylval.nd_obj.name,(yytext));
printf("GE\t");  return GE; }
"=="                        { strcpy(yylval.nd_obj.name,(yytext));
printf("EQ\t");  return EQ; }
```

```
"!="                        { strcpy(yylval.nd_obj.name,(yytext));
printf("NE\t");  return NE; }
">"                           { strcpy(yylval.nd_obj.name,(yytext));
printf("GT\t");  return GT; }
"<"                           { strcpy(yylval.nd_obj.name,(yytext));
printf("LT\t");  return LT; }
"!"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("NOT\t"); return NOT; }
"&&"                       { strcpy(yylval.nd_obj.name,(yytext));
printf("AND\t");  return AND; }
"||"                       { strcpy(yylval.nd_obj.name,(yytext));
printf("OR\t");  return OR; }
"+"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("ADD\t");  return ADD; }
"-"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("SUB\t");  return SUB; }
"/"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("DIV\t");  return DIV; }
"*"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("MULT\t");  return MULT; }
"="                         { strcpy(yylval.nd_obj.name,(yytext));
printf("ASSIGN\t"); return ASSIGN; }
"{"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("BRACES_OPEN\n");  return BRACES_OPEN; }
"}"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("BRACES_CLOSE\n"); return BRACES_CLOSE; }
"("                         { strcpy(yylval.nd_obj.name,(yytext));
printf("BRACKET_OPEN\t"); return BRACKET_OPEN; }
")"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("BRACKET_CLOSE\t");  return BRACKET_CLOSE; }
";"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("DELIM\n"); countn++; return DELIM; }
\/\/.*                      { strcpy(yylval.nd_obj.name,(yytext));
printf("COMM\n"); return COMM; }
[\t]                        { ; }
" "                         { ; }
[\n]                        { ; }
[\r]                        { ; }
["].*["]                    { strcpy(yylval.nd_obj.name,(yytext));
printf("SENTENCE\t"); return SENTENCE; }
.                           { ; }

%%




int yywrap() {
    return 1;
}
```

parser.y

```
%{
    #include<stdio.h>
    #include<string.h>
    #include<stdlib.h>
    #include<ctype.h>
    void yyerror(const char *s);
    int yylex();
    int yywrap();
    extern FILE *yyin;
      extern FILE *yytext;
    void add(char);
    void insert_type();
    int search(char *);
      void printtree(struct node*);
    void printInorder(struct node *);
    struct node* mknode(struct node *left, struct node *right, char
*token);

    struct dataType {
        char * id_name;
        char * data_type;
        char * type;
        int line_no;
    } symbol_table[40];

    int count=0;
    int q;
    char type[10];
    extern int countn;
    struct node {
             struct node *left;
             struct node *right;
             char *token;
    };
      struct node *head;
%}

%union {
      struct var_name {
             char name[100];
             struct node* nd;
      } nd_obj;
}

%token <nd_obj> DOT PRINT SCAN INT FLOAT STRING BOOL RET FOR IF ELSE
INCLUDE T F NUM REAL ID LE GE EQ NE GT LT NOT AND OR ADD SUB DIV MULT
ASSIGN BRACES_OPEN BRACES_CLOSE BRACKET_OPEN BRACKET_CLOSE DELIM COMM
SENTENCE
%type <nd_obj> program headers main statement condition condition_optional
datatype body else init expression arithmetic relop value return

%%
head
program: headers main BRACKET_OPEN BRACKET_CLOSE BRACES_OPEN body return
BRACES_CLOSE { $2.nd = mknode($6.nd, $7.nd, "main"); $$.nd = mknode($1.nd,
$2.nd, "program"); head = $$.nd; }
;

headers: INCLUDE { add('H'); } headers { $1.nd = mknode( NULL, NULL,
$1.name ); $$.nd = mknode($1.nd, $2.nd, "headers"); }
| { $$.nd = NULL; }
;
```

```
main: datatype ID { add('F'); }
;


datatype: INT { insert_type(); }x
| STRING { insert_type(); }head
| PRINT { add('K'); } BRACKET_OPEN SENTENCE BRACKET_CLOSE DELIM body {
$$.nd = mknode(NULL, NULL, "printf"); }
| SCAN { add('K'); } BRACKET_OPEN SENTENCE ',' '&' ID BRACKET_CLOSE DELIM
body { $$.nd = mknode(NULL, NULL, "scanf"); }
| { $$.nd = NULL; }
;

else: ELSE { add('K'); } BRACES_OPEN body BRACES_CLOSE { $$.nd =
mknode(NULL, $4.nd, $1.name); }
| { $$.nd = NULL; }
;

condition: value relop value condition_optional { $$.nd = mknode($1.nd,
$3.nd, $2.name); }
| NOT condition { $1.nd = mknode(NULL,NULL,$1.name); $$.nd = mknode($1.nd,
$2.nd, "condition"); }
| T { add('K'); $$.nd = NULL; }
| F { add('K'); $$.nd = NULL; }
| value { $$.nd = mknode(NULL, NULL, $1.name); }
;

condition_optional: AND condition { $$.nd = mknode($2.nd, NULL, $1.name); }
| OR condition { $$.nd = mknode($2.nd, NULL, $1.name); }
| { $$.nd = NULL; }
;

statement: datatype ID { add('V'); } init { $2.nd = mknode(NULL, NULL,
$2.name); $$.nd = mknode($2.nd, $4.nd, "declaration"); }
| ID ASSIGN expression  { $1.nd = mknode(NULL, NULL, $1.name); $$.nd =
mknode($1.nd, $3.nd, "="); }
| ID relop expression   { $1.nd = mknode(NULL, NULL, $1.name); $$.nd =
mknode($1.nd, $3.nd, "="); }
;

init: ASSIGN expression { $$.nd = $2.nd; }
| { $$.nd = NULL; }
;

expression: value arithmetic expression { $$.nd = mknode($1.nd, $3.nd,
$2.name); }
| value { $$.nd = $1.nd; }
;

arithmetic: ADD
| SUB
| MULT
| DIV
;


relop: LT
| GT
| LE
| GE
| EQ
| NE
;

value: NUM { add('C'); $$.nd = mknode(NULL, NULL, $1.name); }
| REAL { add('C'); $$.nd = mknode(NULL, NULL, $1.name); }
| SENTENCE { add('C'); $$.nd = mknode(NULL, NULL, $1.name); }
| ID { $$.nd = mknode(NULL, NULL, $1.name); }
;

return: RET { add('K'); } expression DELIM { $1.nd = mknode(NULL, NULL,
"return"); $$.nd = mknode($1.nd, $3.nd, "RETURN"); }
;

%%




int main() {
    FILE *myfile = fopen("sample.sal", "r");
    if (!myfile) {
       printf("Cant open the file\n");
       return -1;
    }
    yyin = myfile;
       printf("File input !!\n");
       int p = -1;
    p = yyparse();
    if(!p) printf("\nSuccesfully parsed, no Syntax error found!!\n");
    printf("\n\n");
       printf("\nSYMBOL   DATATYPE   TYPE   LINE NUMBER \n");
       printf("________________________________________\n\n");
       int i=0;
       for(i=0; i<count; i++) {
             printf("%s\t%s\t%s\t%d\t\n", symbol_table[i].id_name,
symbol_table[i].data_type, symbol_table[i].type, symbol_table[i].line_no);
       }
       for(i=0;i<count;i++) {
             free(symbol_table[i].id_name);
             free(symbol_table[i].type);
       }
    fclose(myfile);
       printf("\n\n");
       printtree(head);
    return p;
}

int search(char *type) {
       int i;
       for(i=count-1; i>=0; i--) {
             if(strcmp(symbol_table[i].id_name, type)==0) {
                    return -1;
                    break;
             }
       }
       return 0;
}
```

```
void add(char c) {

  q=search(yytext);
  printf("%d %c\n",q,c);
  if(!q) {
    if(c == 'H') {
                    symbol_table[count].id_name=strdup(yytext);
                    symbol_table[count].data_type=strdup(type);
                    symbol_table[count].line_no=countn;
                    symbol_table[count].type=strdup("Header");
                    count++;
             }
             else if(c == 'K') {
                    symbol_table[count].id_name=strdup(yytext);
                    symbol_table[count].data_type=strdup("N/A");
                    symbol_table[count].line_no=countn;
                    symbol_table[count].type=strdup("Keyword\t");
                    count++;
             }
             else if(c == 'V') {
                    symbol_table[count].id_name=strdup(yytext);
                    symbol_table[count].data_type=strdup(type);
                    symbol_table[count].line_no=countn;
                    symbol_table[count].type=strdup("Variable");
                    count++;
             }
             else if(c == 'C') {
                    symbol_table[count].id_name=strdup(yytext);
                    symbol_table[count].data_type=strdup("CONST");
                    symbol_table[count].line_no=countn;
                    symbol_table[count].type=strdup("Constant");
                    count++;
             }
             else if(c == 'F') {
                    symbol_table[count].id_name=strdup(yytext);
                    symbol_table[count].data_type=strdup(type);
                    symbol_table[count].line_no=countn;
                    symbol_table[count].type=strdup("Function");
                    count++;
             }
       }
}

struct node* mknode(struct node *left, struct node *right, char *token) {

       struct node *newnode = (struct node *)malloc(sizeof(struct node));
       char *newstr = (char *)malloc(strlen(token)+1);
       strcpy(newstr, token);
       newnode->left = left;
       newnode->right = right;
       newnode->token = newstr;
       return(newnode);
}

void printtree(struct node* tree) {
       printf("\n\n Inorder traversal of the Parse Tree: \n\n");
       printInorder(tree);
       printf("\n\n");
}

void printInorder(struct node *tree) {
       int i;
       if (tree->left) {
             printInorder(tree->left);
       }
       printf("%s, ", tree->token);
       if (tree->right) {
             printInorder(tree->right);
       }
}

void insert_type() {
       strcpy(type, yytext);
}



void yyerror(const char* msg) {
  fprintf(stderr, "%s\n", msg);
}
```

Explanation of the source code -

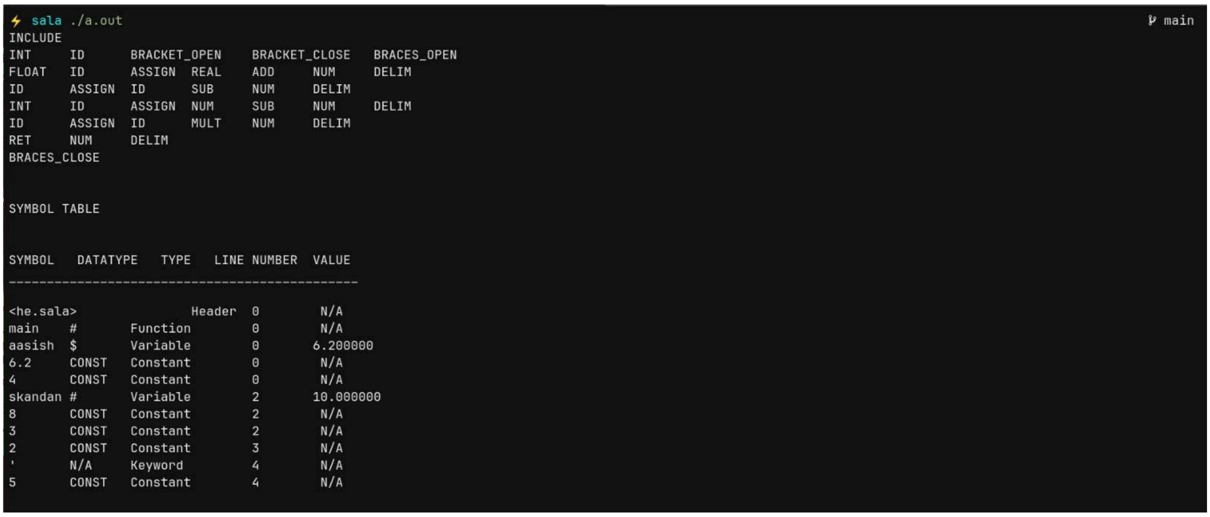
The production rules with the appropriate semantic actions are written in the parser.y file with which *yacc* will generate a parser for the defined grammar. Additionally, during the parsing phase, we are maintaining a symbol table which is filled every time an identifier is encountered. We also create nodes in the semantic actions for generating the parse tree later. The values of tokens generated in the lex phase is passed through yytext() and yylval() to the parser.

Commands to run the parser and lexer:

```
yacc -vd parser.y
lex lexer.l
cc lex.yy.c -ll
cc y.tab.c lex.yy.c
./a.out
```

Sample Output:

```
INCLUDE
INT     ID      BRACKET_OPEN            BRACKET_CLOSE           BRACES_OPEN
INT     ID      ASSIGN          NUM             DELIMITER
COMMENT
BOOL    ID      ASSIGN          SENTENCE        DELIMITER
Syntax Error
```



```
⚡ sala ./a.out                                                                                                              main
INCLUDE
INT     ID      BRACKET_OPEN    BRACKET_CLOSE   BRACES_OPEN
FLOAT   ID      ASSIGN  REAL    ADD     NUM     DELIM
ID      ASSIGN  ID      SUB     NUM     DELIM
INT     ID      ASSIGN  NUM     SUB     NUM     DELIM
ID      ASSIGN  ID      MULT    NUM     DELIM
RET     NUM     DELIM
BRACES_CLOSE


SYMBOL TABLE


SYMBOL   DATATYPE   TYPE   LINE NUMBER  VALUE
----------------------------------------------

<he.sala>               Header  0        N/A
main    #       Function        0        N/A
aasish  $       Variable        0       6.200000
6.2     CONST   Constant        0        N/A
4       CONST   Constant        0        N/A
skandan #       Variable        2       10.000000
8       CONST   Constant        2        N/A
3       CONST   Constant        2        N/A
2       CONST   Constant        3        N/A
'       N/A     Keyword         4        N/A
5       CONST   Constant        4        N/A
```

Lexical Analysis and Symbol Table generation during parsing stage

Assignment - 3

Syntax Directed Translation

06/04/2023

Contributions:

106120001 - Semantic Actions and Three Address Code generation

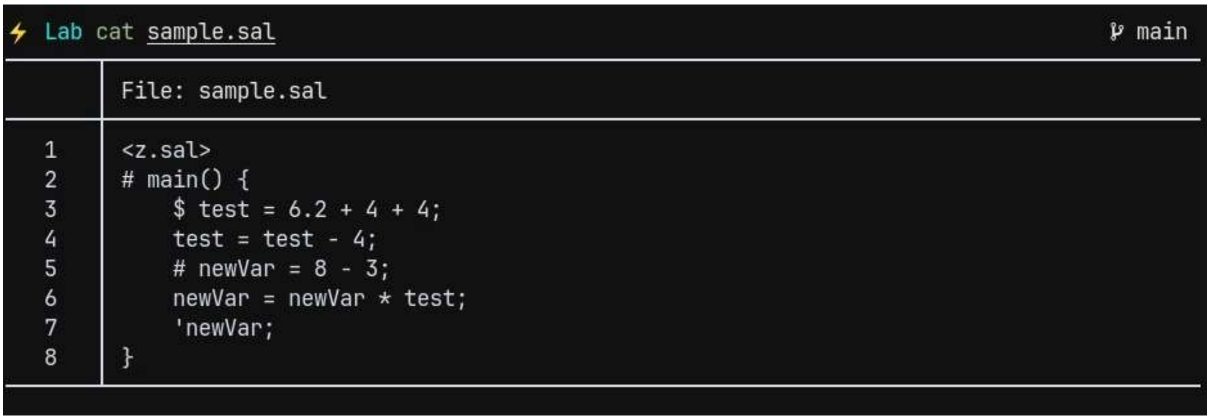
106120009 - Parse Tree and Three Address Code generation

106120017 - Semantic Actions and Parse Tree

106120121 - Parse Tree, Precedence and Backpatching

Aim: To include semantic actions in the syntax analyzer and build an intermediate code generator

Source code:



```
⚡ Lab cat sample.sal                                                       main

       File: sample.sal

       <z.sal>
       # main() {
           $ test = 6.2 + 4 + 4;
           test = test - 4;
           # newVar = 8 - 3;
           newVar = newVar * test;
           'newVar;
       }
```

lexer.l

```
%{
    #include <stdio.h>
    #include "y.tab.h"
    int countn = 0;
%}

%option yylineno

alpha [a-zA-Z]
digit [0-9]

%%

^<{alpha}({alpha}|{digit})*.sal> { strcpy(yylval.nd_obj.name,(yytext));
printf("INCLUDE\n");  return INCLUDE; }

"<<"                        { strcpy(yylval.nd_obj.name,(yytext));
printf("PRINT\t");  return PRINT; }
">>"                        { strcpy(yylval.nd_obj.name,(yytext));
printf("SCAN\t");  return SCAN; }

"#"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("INT\t");  return INT; }
"$"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("FLOAT\t");  return FLOAT; }
"~"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("STRING\t");  return STRING; }
"_"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("BOOL\t");  return BOOL; }

"'"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("RET\t");  return RET; }
```

```
"@"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("FOR\t");  return FOR; }
"?"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("IF\t");  return IF; }
":"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("ELSE\t");  return ELSE; }

{digit}+                    { strcpy(yylval.nd_obj.name,(yytext));
printf("NUM\t");  return NUM; }
{digit}+\.{digit}{0,6}      { strcpy(yylval.nd_obj.name,(yytext));
printf("REAL\t");  return REAL; }
["].*["]                    { strcpy(yylval.nd_obj.name,(yytext));
printf("SENTENCE\t"); return SENTENCE; }
"True"                      { strcpy(yylval.nd_obj.name,(yytext));
printf("T\t");  return T; }
"False"                     { strcpy(yylval.nd_obj.name,(yytext));
printf("F\t");  return F; }

{alpha}({alpha}|{digit})*   { strcpy(yylval.nd_obj.name,(yytext));
printf("ID\t");  return ID; }

"<="                        { strcpy(yylval.nd_obj.name,(yytext));
printf("LE\t");  return LE; }
">="                        { strcpy(yylval.nd_obj.name,(yytext));
printf("GE\t");  return GE; }
"=="                        { strcpy(yylval.nd_obj.name,(yytext));
printf("EQ\t");  return EQ; }
"!="                        { strcpy(yylval.nd_obj.name,(yytext));
printf("NE\t");  return NE; }
">"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("GT\t");  return GT; }
"<"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("LT\t");  return LT; }

"!"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("NOT\t"); return NOT; }
"&&"                        { strcpy(yylval.nd_obj.name,(yytext));
printf("AND\t");  return AND; }
"||"                        { strcpy(yylval.nd_obj.name,(yytext));
printf("OR\t");  return OR; }

"+"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("ADD\t");  return ADD; }
"-"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("SUB\t");  return SUB; }
"*"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("MULT\t");  return MULT; }
"/"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("DIV\t");  return DIV; }

"="                         { strcpy(yylval.nd_obj.name,(yytext));
printf("ASSIGN\t"); return ASSIGN; }

"{"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("BRACES_OPEN\n");  return BRACES_OPEN; }
"}"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("BRACES_CLOSE\n"); return BRACES_CLOSE; }
```

```
"("                         { strcpy(yylval.nd_obj.name,(yytext));
printf("BRACKET_OPEN\t"); return BRACKET_OPEN; }
")"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("BRACKET_CLOSE\t");  return BRACKET_CLOSE; }

";"                         { strcpy(yylval.nd_obj.name,(yytext));
printf("DELIM\n"); countn++; return DELIM; }

\/\/.*                      { strcpy(yylval.nd_obj.name,(yytext));
printf("COMMENT\n"); return COMMENT; }

[ \t\n\r]                   { ; }
.                           { ; }

%%

//The value of yywarap() is checked on reaching EOF
//If it is non-zero, scanning terminates
int yywrap() {
    return 1;
}
```

parser.y

```
%{
    #include<stdio.h>
    #include<string.h>
    #include<stdlib.h>
    #include<ctype.h>
    #include <math.h>

    void yyerror(const char *s);
    int yylex();
    int yywrap();
    extern FILE *yyin;
    extern FILE *yytext;
    void add(char);
    void insert_type();
    int search(char *);
    void fill(char *, float);
    struct node* mknode(struct node *left, struct node *right, char
*token);

    struct dataType {
        char * id_name;
        char * data_type;
        char * type;
        float value;
        int line_no;
    } symbol_table[100];

    int count=0;
    int ic_idx=0;
    int label=0;
    int temp_var=0;
    int is_for=0;
    char icg[50][100];
    int q;
    char type[10];
    extern int countn;
    struct node {
        struct node *left;
        struct node *right;
        float value;
        char *token;
    };
    struct node *head;

       struct lbs{
             int for_goto;
             int for_jmp_false;
       };

       void back_patch( int addr,  enum code_ops operation, int arg  )
       {
         code[addr].op  = operation;
         code[addr].arg = arg;
       }

    float calculate(float operand_1, float operand_2, char* operator){
        if(strcmp(operator, "+") == 0){
            return operand_1 + operand_2;
        } else if(strcmp(operator, "-") == 0){
            return operand_1 - operand_2;
        } else if(strcmp(operator, "*") == 0){
            return operand_1 * operand_2;
        } else if(strcmp(operator, "/") == 0){
            return operand_1 / operand_2;
        }
    }

    void fill(char* identifier, float new_value){
        int index = search(identifier);
        symbol_table[index].value = new_value;
    }

%}

%union {
    struct var_name {
        char name[100];
        struct node* nd;
        float value;
    } nd_obj;
}

%token <nd_obj> PRINT SCAN INT FLOAT STRING BOOL RET FOR IF ELSE INCLUDE T
F NUM REAL ID LE GE EQ NE GT LT NOT AND OR ADD SUB DIV MULT ASSIGN
BRACES_OPEN BRACES_CLOSE BRACKET_OPEN BRACKET_CLOSE DELIM COMMENT SENTENCE
%type <nd_obj> program headers main statement condition condition_optional
datatype body else init expression arithmetic relop value return

%left ADD SUB
%left MULT DIV
```

```
%%
program: headers main BRACKET_OPEN BRACKET_CLOSE BRACES_OPEN body return
BRACES_CLOSE { $2.nd = mknode($6.nd, $7.nd, "main"); $$.nd = mknode($1.nd,
$2.nd, "program"); head = $$.nd; }
;

headers: INCLUDE { add('H'); } headers { $1.nd = mknode( NULL, NULL,
$1.name ); $$.nd = mknode($1.nd, NULL, "headers"); }
| { $$.nd = NULL; }
;


main: datatype ID { add('F'); }
;

datatype: INT { insert_type(); }
| FLOAT { insert_type(); }
| STRING { insert_type(); }
| { $$.nd = NULL; }
;

body: FOR { add('K'); } BRACKET_OPEN statement DELIM condition DELIM
statement BRACKET_CLOSE BRACES_OPEN body BRACES_CLOSE {back_patch($1-
>for_jmp_false, JMP_FALSE, mknode());} body
| IF { add('K'); } BRACKET_OPEN condition BRACKET_CLOSE BRACES_OPEN body
BRACES_CLOSE else body
| statement DELIM body { $$.nd = mknode($1.nd, $3.nd, "bline"); }
| PRINT { add('K'); } BRACKET_OPEN SENTENCE BRACKET_CLOSE DELIM body {
$$.nd = mknode(NULL, NULL, "printf"); }
| SCAN { add('K'); } BRACKET_OPEN SENTENCE ',' '&' ID BRACKET_CLOSE DELIM
body { $$.nd = mknode(NULL, NULL, "scanf"); }
| { $$.nd = NULL; }
;

else: ELSE { add('K'); } BRACES_OPEN body BRACES_CLOSE { $$.nd =
mknode(NULL, $4.nd, $1.name); back_patch($1->for_jmp_false, JMP_FALSE,
mknode());}
| { $$.nd = NULL; back_patch($1->for_goto, GOTO, mknode());}
;

condition: value relop value condition_optional { $$.nd = mknode($1.nd,
$3.nd, $2.name); }
| NOT condition { $1.nd = mknode(NULL,NULL,$1.name); $$.nd = mknode($1.nd,
$2.nd, "condition"); }
| T { add('K'); $$.nd = NULL; }
| F { add('K'); $$.nd = NULL; }
| value { $$.nd = mknode(NULL, NULL, $1.name); }
;

condition_optional: AND condition { $$.nd = mknode($2.nd, NULL, $1.name); }
| OR condition { $$.nd = mknode($2.nd, NULL, $1.name); }
| { $$.nd = NULL; }
;

statement: datatype ID { add('V'); } init { $2.nd = mknode(NULL, NULL,
$2.name); $$.nd = mknode($2.nd, $4.nd, "declaration"); $2.value = $4.value;
fill($2.name, $2.value); sprintf(icg[ic_idx++], "=\t%s\t%f\n", $2.name,
$2.value); }
```

```
| ID ASSIGN expression  { $1.nd = mknode(NULL, NULL, $1.name); $$.nd =
mknode($1.nd, $3.nd, "="); $1.value = $3.value; fill($1.name, $1.value);
char str[100]; sprintf(str, "%s = %d", $1.name, $3.value);
sprintf(icg[ic_idx++], "=\t%s\t%f\n", $1.name, $1.value); }
| ID relop expression   { $1.nd = mknode(NULL, NULL, $1.name); $$.nd =
mknode($1.nd, $3.nd, "="); }
;

init: ASSIGN expression { $$.nd = $2.nd; $$.value = $2.value; }
| { $$.nd = NULL; }
;

expression: value arithmetic expression { $$.nd = mknode($1.nd, $3.nd,
$2.name); $$.value = calculate($1.value, $3.value, $2.name); char str[100];
sprintf(str, "%s\t%s\t%s\t%s", $$.name, $1.name, $2.name, $3.name);
sprintf(icg[ic_idx++], "%s\t%s\t%s\t%f\n", $2.name, $1.name, $3.name,
$$.value); }
| value { $$.nd = $1.nd; $$.value = $1.value; char str[100]; sprintf(str,
"%s = %s", $$.name, $1.name); sprintf(icg[ic_idx++], "=\t%s\tN/A\t%s\n",
$1.name, $$.name); }
;

arithmetic: ADD
| SUB
| MULT
| DIV
;


relop: LT
| GT
| LE
| GE
| EQ
| NE
;

value: NUM { add('C'); $$.nd = mknode(NULL, NULL, $1.name); $$.value =
atoi($1.name); }
| REAL { add('C'); $$.nd = mknode(NULL, NULL, $1.name); $$.value =
atof($1.name); }
| SENTENCE { add('C'); $$.nd = mknode(NULL, NULL, $1.name); }
| ID { $$.nd = mknode(NULL, NULL, $1.name); int index = search($1.name);
if(index != -1) { $1.value = symbol_table[index].value; } $$.value =
$1.value; }
;

return: RET { add('K'); } expression DELIM { $1.nd = mknode(NULL, NULL,
"return"); $$.nd = mknode($1.nd, $3.nd, "RETURN"); }
;

%%

void printBTHelper(char* prefix, struct node* ptr, int isLeft) {
    if( ptr != NULL ) {
        printf(prefix);
        if(isLeft) { printf("├──"); }
        else { printf("└──"); }
        printf(ptr->token);
        printf("\n");
        char* addon = isLeft ? "│   " : "    ";
        int len2 = strlen(addon);
        int len1 = strlen(prefix);
        char* result = (char*)malloc(len1 + len2 + 1);
        strcpy(result, prefix);
        strcpy(result + len1, addon);
        printBTHelper(result, ptr->left, 1);
        printBTHelper(result, ptr->right, 0);
        free(result);
    }
}

void printBT(struct node* ptr) {
    printf("\n");
    printBTHelper("", ptr, 0);
}

int main() {
    FILE *myfile = fopen("sample.sal", "r");
    if (!myfile) {
        printf("Cant open the file\n");
        return -1;
    }
    yyin = myfile;
    int p = -1;
    p = yyparse();
    /* if(!p) printf("\nSuccesfully parsed, no Syntax error found!!\n"); */
    printf("\n\n");
    printf("SYMBOL TABLE");
    printf("\n\n");
    printf("\nSYMBOL   DATATYPE   TYPE   LINE NUMBER  VALUE\n");
    printf("______________________________________________\n\n");
    int i=0;
    for(i=0; i<count; i++) {
        printf("%s\t%s\t%s\t%d\t", symbol_table[i].id_name,
symbol_table[i].data_type, symbol_table[i].type, symbol_table[i].line_no);
        if(strcmp(symbol_table[i].type, "Variable") == 0) printf("%f\n",
symbol_table[i].value);
        else printf(" N/A \n");
    }
    printf("\n\n");
    printf("PARSE TREE");
    printf("\n\n");
    printBT(head);
    printf("\n\n");
    printf("THREE ADDRESS CODE");
    printf("\n\n");
    for(int i=0; i<ic_idx; i++){
        printf("%s", icg[i]);
    }
    printf("\n\n");
    for(i=0;i<count;i++) {
        free(symbol_table[i].id_name);
        free(symbol_table[i].type);
    }
    fclose(myfile);
    return p;
}
```

```
int search(char *type) {
    int i;
    for(i=count-1; i>=0; i--) {
        if(strcmp(symbol_table[i].id_name, type)==0) {
            return i;
            break;
        }
    }
    return -1;
}

void add(char c) {

  q=search(yytext);
  if(q == -1) {
    if(c == 'H') {
            symbol_table[count].id_name=strdup(yytext);
            symbol_table[count].data_type=strdup(type);
            symbol_table[count].line_no=countn;
            symbol_table[count].type=strdup("Header");
            count++;
        }
        else if(c == 'K') {
            symbol_table[count].id_name=strdup(yytext);
            symbol_table[count].data_type=strdup("N/A");
            symbol_table[count].line_no=countn;
            symbol_table[count].type=strdup("Keyword\t");
            count++;
        }
        else if(c == 'V') {
            symbol_table[count].id_name=strdup(yytext);
            symbol_table[count].data_type=strdup(type);
            symbol_table[count].line_no=countn;
            symbol_table[count].type=strdup("Variable");
            count++;
        }
        else if(c == 'C') {
            symbol_table[count].id_name=strdup(yytext);
            symbol_table[count].data_type=strdup("CONST");
            symbol_table[count].line_no=countn;
            symbol_table[count].type=strdup("Constant");
            count++;
        }
        else if(c == 'F') {
            symbol_table[count].id_name=strdup(yytext);
            symbol_table[count].data_type=strdup(type);
            symbol_table[count].line_no=countn;
            symbol_table[count].type=strdup("Function");
            count++;
        }
    }
}

struct node* mknode(struct node *left, struct node *right, char *token) {
    struct node *newnode = (struct node *)malloc(sizeof(struct node));
    char *newstr = (char *)malloc(strlen(token)+1);
    strcpy(newstr, token);
    newnode->left = left;
```

```
        newnode->right = right;
        newnode->token = newstr;
        return(newnode);
    }

    void insert_type() {
        strcpy(type, yytext);
    }

    void yyerror(const char* msg) {
      fprintf(stderr, "%s\n", msg);
    }
```

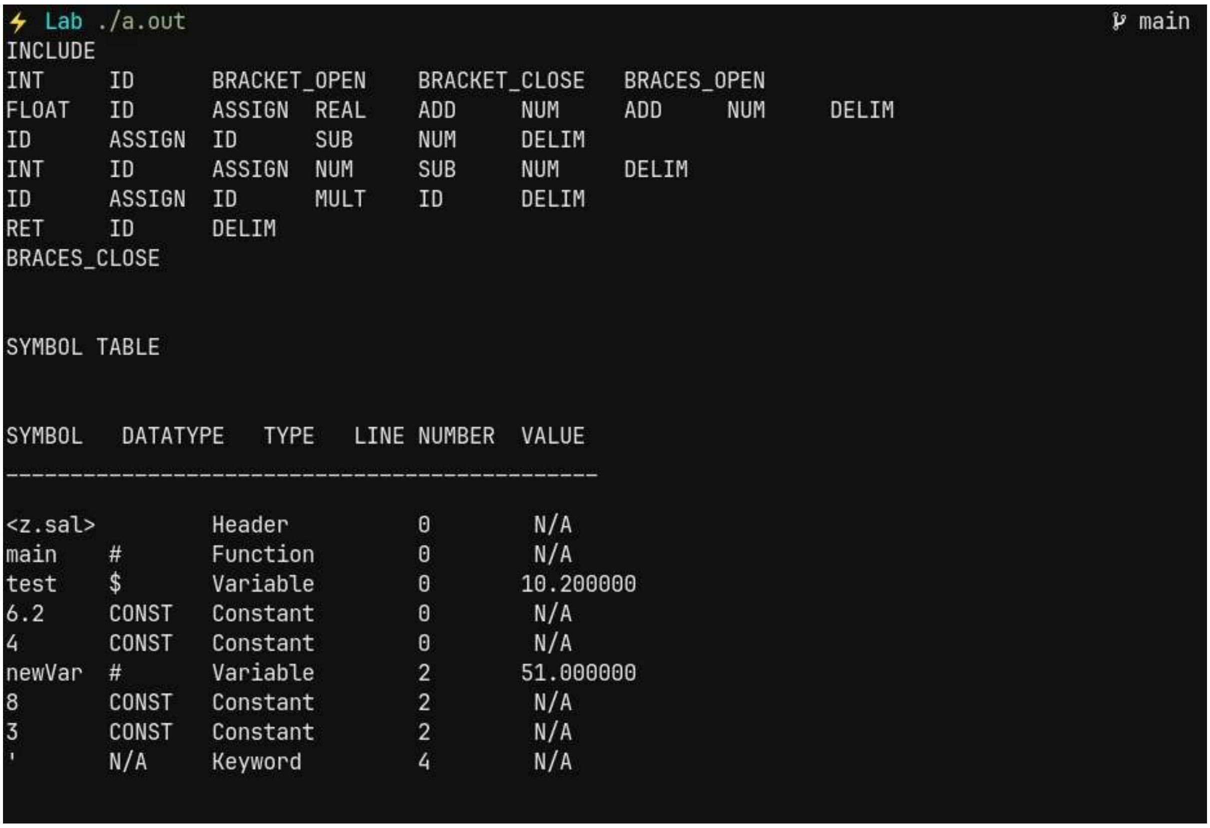
Explanation of the source code -

The parse tree is generated using the nodes created in the previous assignment and printed on the screen. We have also implemented 3 address code in the form of quadruples. In case of conditional and iterative statements, backpatching has been done to jump to appropriate locations. The symbol table contains the values of all variables at the end of the program, which is also being printed.

Commands to run:

```
yacc -vd parser.y
lex lexer.l
cc lex.yy.c y.tab.c -ll
cc y.tab.c lex.yy.c
./a.out
```

Sample Output:



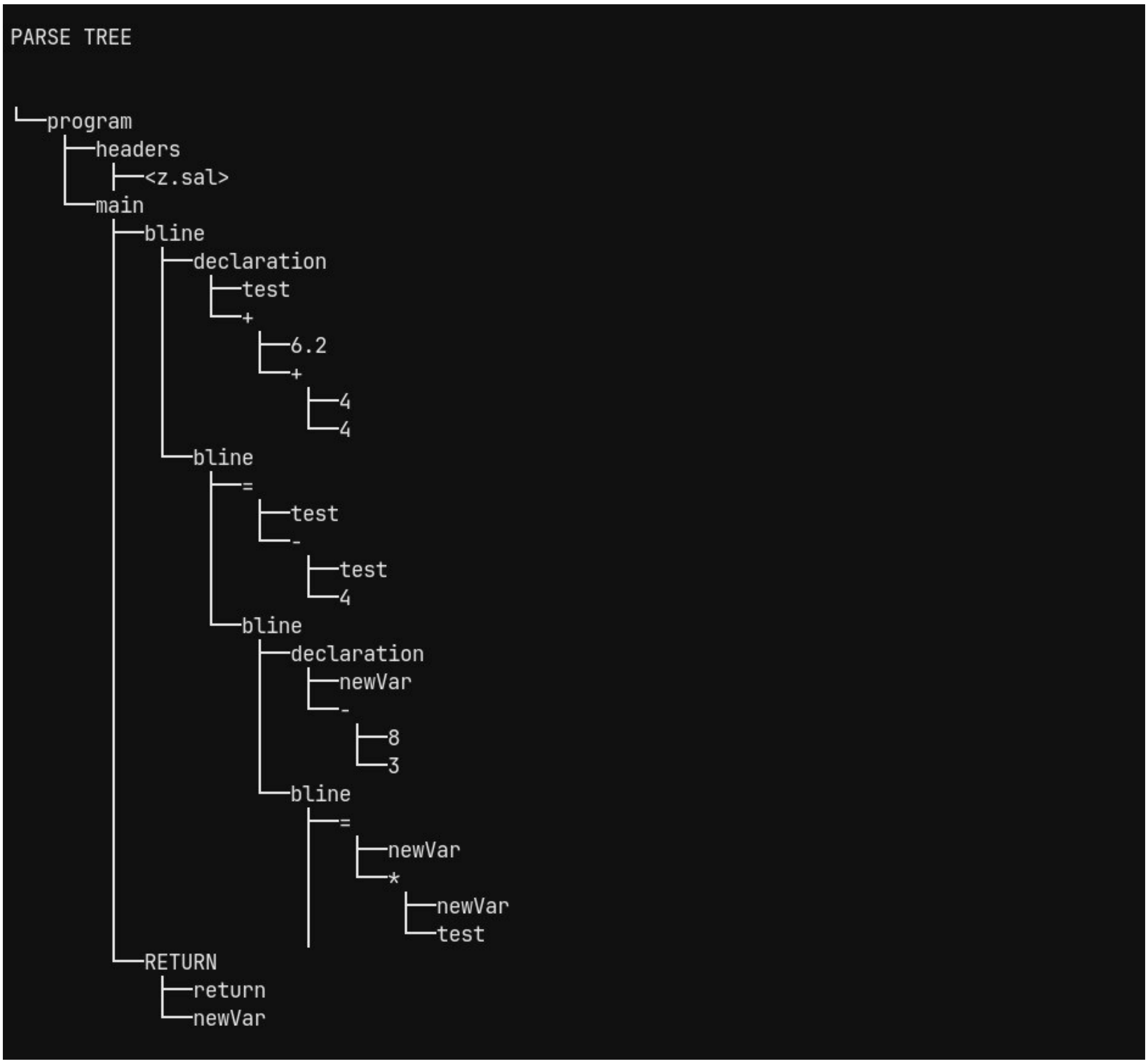
```
⚡ Lab ./a.out                                                                     main
INCLUDE
INT     ID      BRACKET_OPEN    BRACKET_CLOSE   BRACES_OPEN
FLOAT   ID      ASSIGN  REAL    ADD     NUM     ADD     NUM     DELIM
ID      ASSIGN  ID      SUB     NUM     DELIM
INT     ID      ASSIGN  NUM     SUB     NUM     DELIM
ID      ASSIGN  ID      MULT    ID      DELIM
RET     ID      DELIM
BRACES_CLOSE


SYMBOL TABLE


SYMBOL   DATATYPE   TYPE   LINE NUMBER  VALUE
----------------------------------------------

<z.sal>         Header          0        N/A
main    #       Function        0        N/A
test    $       Variable        0       10.200000
6.2     CONST   Constant        0        N/A
4       CONST   Constant        0        N/A
newVar  #       Variable        2       51.000000
8       CONST   Constant        2        N/A
3       CONST   Constant        2        N/A
'       N/A     Keyword         4        N/A
```

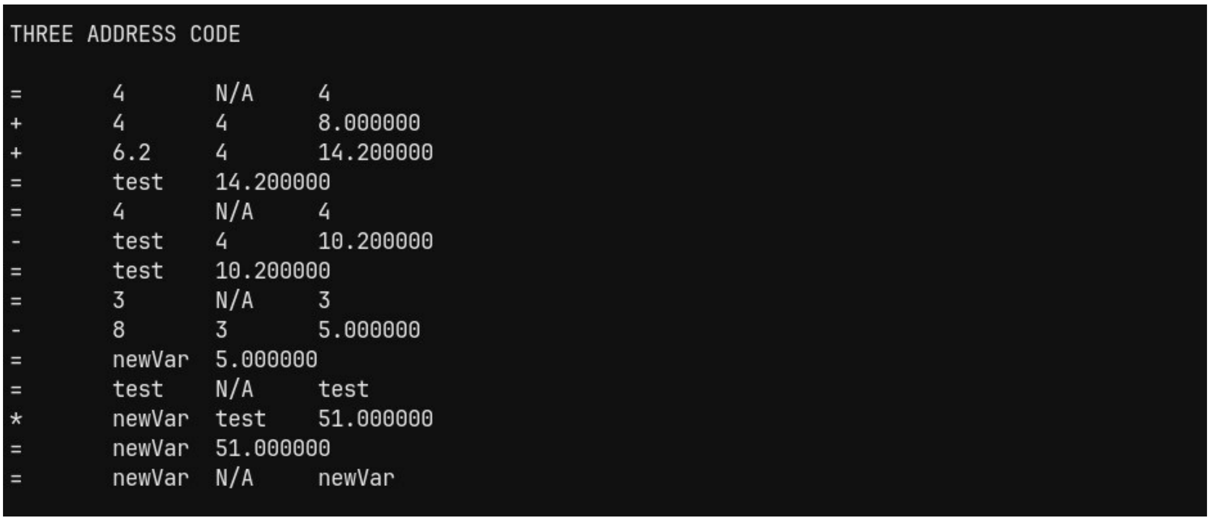
Lexical Analysis and Symbol Table Generation



```
PARSE TREE


└──program
    ├──headers
    │   ├──<z.sal>
    └──main
        ├──bline
        │   ├──declaration
        │   │   ├──test
        │   │   └──+
        │   │       ├──6.2
        │   │       └──+
        │   │           ├──4
        │   │           └──4
        │   └──bline
        │       ├──=
        │       │   ├──test
        │       │   └──-
        │       │       ├──test
        │       │       └──4
        │       └──bline
        │           ├──declaration
        │           │   ├──newVar
        │           │   └──-
        │           │       ├──8
        │           │       └──3
        │           └──bline
        │               ├──=
        │               │   ├──newVar
        │               │   └──*
        │               │       ├──newVar
        │               │       └──test
        └──RETURN
            ├──return
            └──newVar
```

Printing the Parse Tree



```
THREE ADDRESS CODE

=       4       N/A     4
+       4       4       8.000000
+       6.2     4       14.200000
=       test    14.200000
=       4       N/A     4
-       test    4       10.200000
=       test    10.200000
=       3       N/A     3
-       8       3       5.000000
=       newVar  5.000000
=       test    N/A     test
*       newVar  test    51.000000
=       newVar  51.000000
=       newVar  N/A     newVar
```

Three Address Code in the form of quadruples