Compiler Design Lab Project

Aditya Bisht - 15CO104
B. Bharath Kumar - 15CO113
Report #2 - Parser

Introduction

Parsing, syntax analysis or syntactic analysis is the process of analyzing a string of symbols, either in natural language, computer languages or data structures, conforming to the rules of a formal grammar. The term has slightly different meanings in different branches of linguistics and computer science. Traditional sentence parsing is often performed as a method of understanding the exact meaning of a sentence or word, sometimes with the aid of devices such as sentence diagrams. It usually emphasizes the importance of grammatical divisions such as subject and predicate.Within computer science, the term is used in the analysis of computer languages, referring to the syntactic analysis of the input code into its component parts in order to facilitate the writing of compilers and interpreters. The term may also be used to describe a split or separation.

A parser is a software component that takes input data (frequently text) and builds a data structure – often some kind of parse tree, abstract syntax tree or other hierarchical structure – giving a structural representation of the input, checking for correct syntax in the process. The parsing may be preceded or followed by other steps, or these may be combined into a single step. The parser is often preceded by a separate lexical analyzer, which creates tokens from the sequence of input characters; alternatively, these can be combined in scanner less parsing. Parsers may be programmed by hand or may be automatically or semi-automatically generated by a parser generator. Parsing is complementary to templating, which produces formatted output. These may be applied to different domains, but often appear together, such as the scanf/printf pair, or the input (front end parsing) and output (back end code generation) stages of a compiler.

The input to a parser is often text in some computer language, but may also be text in a natural language or less structured textual data, in which case generally only certain parts of the text are extracted, rather than a parse tree being constructed. Parsers range from very simple functions such as scanf, to complex programs such as the frontend of a C++ compiler or the HTML parser of a web browser. An important class of simple parsing is done using regular expressions, in which a group of regular expressions defines a regular language and a regular expression engine automatically generating a parser for that language, allowing pattern matching and extraction of text. In other contexts regular expressions are instead used prior to parsing, as the lexing step whose output is then used by the parser.

The use of parsers varies by input. In the case of data languages, a parser is often found as the file reading facility of a program, such as reading in HTML or XML text; these examples are markup languages. In the case of programming languages, a parser is a component of a compiler or interpreter, which parses the source code of a computer

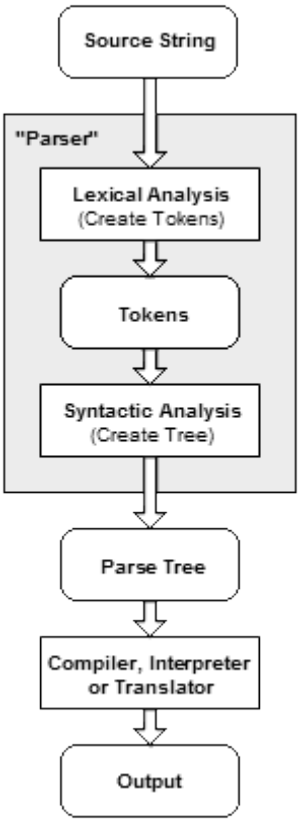
programming language to create some form of internal representation; the parser is a key step in the compiler frontend. Programming languages tend to be specified in terms of a deterministic context-free grammar because fast and efficient parsers can be written for them. For compilers, the parsing itself can be done in one pass or multiple passes.

The implied disadvantages of a one-pass compiler can largely be overcome by adding fix-ups, where provision is made for fix-ups during the forward pass, and the fix-ups are applied backwards when the current program segment has been recognized as having been completed. An example where such a fix-up mechanism would be useful would be a forward GOTO statement, where the target of the GOTO is unknown until the program segment is completed. In this case, the application of the fix-up would be delayed until the target of the GOTO was recognized. Obviously, a backward GOTO does not require a fix-up.

Context-free grammars are limited in the extent to which they can express all of the requirements of a language. Informally, the reason is that the memory of such a language is limited. The grammar cannot remember the presence of a construct over an arbitrarily long input; this is necessary for a language in which, for example, a name must be declared before it may be referenced. More powerful grammars that can express this constraint, however, cannot be parsed efficiently. Thus, it is a common strategy to create a relaxed parser for a context-free grammar which accepts a superset of the desired language constructs (that is, it accepts some invalid constructs); later, the unwanted constructs can be filtered out at the semantic analysis (contextual analysis) step.

Overview of the Process

The following example demonstrates the common case of parsing a computer language with two levels of grammar: lexical and syntactic. The first stage is the token generation, or lexical analysis, by which the input character stream is split into meaningful symbols defined by a grammar of regular expressions. For example, a calculator program would look at an input such as "11 * (5 + 4)^2" and split it into the tokens 11, *, (, 5, +, 4,), ^, 2, each of which is a meaningful symbol in the context of an arithmetic expression. The lexer would contain rules to tell it that the characters *, +, ^, (and) mark the start of a new token, so meaningless tokens like "12*" or "(3" will not be generated.



The next stage is parsing or syntactic analysis, which is checking that the tokens form an allowable expression. This is usually done with reference to a context-free grammar which recursively defines components that can make up an expression

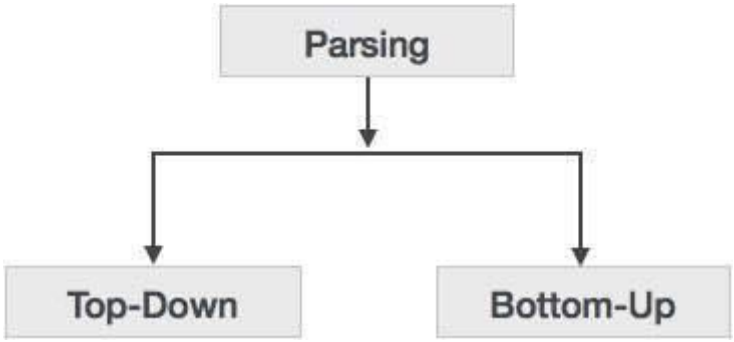
and the order in which they must appear. However, not all rules defining programming languages can be expressed by context-free grammars alone, for example type validity and proper declaration of identifiers. These rules can be formally expressed with attribute grammars.

The final phase is semantic parsing or analysis, which is working out the implications of the expression just validated and taking the appropriate action. In the case of a calculator or interpreter, the action is to evaluate the expression or program, a compiler, on the other hand, would generate some kind of code. Attribute grammars can also be used to define these actions.

Types of Parsers

The task of the parser is essentially to determine if and how the input can be derived from the start symbol of the grammar. This can be done in essentially two ways:

1. Top-down parsing - Top-down parsing can be viewed as an attempt to find left-most derivations of an input-stream by searching for parse trees using a top-down expansion of the given formal grammar rules. Tokens are consumed from left to right. Inclusive choice is used to accommodate ambiguity by expanding all alternative right-hand-sides of grammar rules.
2. Bottom-up parsing - A parser can start with the input and attempt to rewrite it to the start symbol. Intuitively, the parser attempts to locate the most basic elements, then the elements containing these, and so on. LR parsers are examples of bottom-up parsers. Another term used for this type of parser is Shift-Reduce parsing.



LL parsers and recursive-descent parser are examples of top-down parsers which cannot accommodate left recursive production rules. Although it has been believed that simple implementations of top-down parsing cannot accommodate direct and indirect left-recursion and may require exponential time and space complexity while parsing ambiguous context-free grammars, more sophisticated algorithms for top-down parsing have been created by Frost, Hafiz, and Callaghanwhich accommodate ambiguity and left recursion in

polynomial time and which generate polynomial-size representations of the potentially exponential number of parse trees. Their algorithm is able to produce both left-most and right-most derivations of an input with regard to a given context-free grammar.An important distinction with regard to parsers is whether a parser generates a leftmost derivation or a rightmost derivation (see context-free grammar). LL parsers will generate a leftmost derivation and LR parsers will generate a rightmost derivation (although usually in reverse).

In top-down parser the parser starts constructing parse tree from start symbol and then tries to get the required input. In bottom-up parser It takes the input symbol and tries to get the start symbol. Let us consider example of a bottom-up parser.

E->E+T|T
T->T*F|F
F->ID

Consider ID*ID+ID
F*ID+ID →T*ID+ID → T*F+ID →T+ID →E+ID →E+F →E+T → E.

Now if we see the above input will be accepted by the grammar as we got the start symbol E. This is how the bottom up parser works.

Top-down parser

Recursive Descent parsing:

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity.

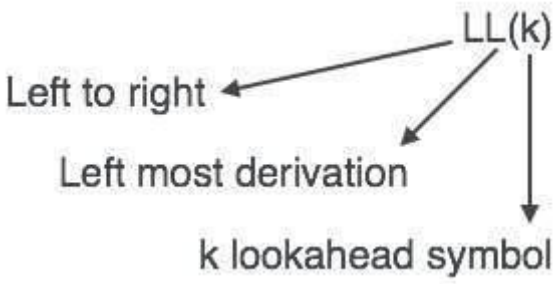
Backtracking:

Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them.

LL Parser:

An LL Parser accepts LL grammar. LL grammar is a subset of context-free grammar but with some restrictions to get the simplified version, in order to achieve easy implementation. LL grammar can be implemented by means of both algorithms namely, recursive-descent or table-driven.

LL parser is denoted as LL(1). The first L in LL(1) is parsing the input from left to right, the second L in LL(1) stands for left-most derivation and 1 itself represents the number of look ahead.



Bottom-up parser

Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node.

The main type of parsers is LR parser. The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique. LR parsers are also known as LR(k) parsers, where L stands for left-to-right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of lookahead symbols to make decisions.

There are three widely used algorithms available for constructing an LR parser:

- SLR(1) – Simple LR Parser:
    - Works on smallest class of grammar
    - Few number of states, hence very small table
    - Simple and fast construction
- LR(1) – LR Parser:
    - Works on complete set of LR(1) Grammar
    - Generates large table and large number of states
    - Slow construction
- LALR(1) – Look-Ahead LR Parser:
    - Works on intermediate size of grammar
    - Number of states are same as in SLR(1)

Implementation

First, we split the code in two parts, one is the header files and the rest of the code. We implemented the header files with 'Header' as the grammar name and the rest as start. Again, start is divided, here BODY is, thecode which will be written inside the flower braces. Body contains expressions, loops, if statements etc. at each line. For the loop and conditions, they were written separately. Single in the grammar is the one-line expressions, this is defined as in loops and conditions there may not be braces and only one expression or statement will be considered. Once all the content in the file are parsed then we call a function called the print which will print the symbol table.

If we look at theStart : TYPE ID BO ARG BC FO BODY FC Start| ;

The TYPE is the return type i.e int, float etc. ID will be main or identifier, ARG will be the arguments, BO BC are (,) and FO FC are {, } BODY is where the c code will be written.

The ARG gives 3 rules, ID in here is the identifier. There is COMMA token in the next rule which will allow us to create multiple declarations (i.e. int a, b, c;) at a time.

The BODY gives Bb BODY, Bb has the rules like they can be expressions, declarations, while loops, for loops, if-else statements, return etc. The BODY will again repeat the Bb so we can have multiple lines of same or different code. If we want to stop BODY is assigned null(epsilon).

The R in the grammar represents the expressions that will get evaluated. DEC is declaring variables. WLOOP is the while loop, WDEF is the while loop definition i.e is the content for while loop, in single I have declared a while loop, this allows me to accept multiple while loops, this is the same followed for multiple for loops and if-else statements. IDEF is for if-else definition, FDEF is for 'for' loop definition.

In lex all the tokens are parsed and are returned as tokens which are defined in the YACC, when we find the tokens those tokens are stored in linked list, this is achieved by calling the insert function and sending the required parameters as the arguments. All the tokens stored in the linked list are unique. For the identifiers and constants their type is also stored, for all the others NA is stored and when printing the symbol table only identifiers and constants are printed.

Code:

Lex.l

lex.l

```
%option noyywrap
%{
 #include "yacc.tab.h"
 void insert(char *yytext,char t,int l,char *type);
 struct Storage
{
 char *name, token[20], Data[10];
 int val;
 int li;
 struct Storage *next;
}*st,*head;
int var_cnt=0;
char data[10];
int yylineno=1;
void print();
%}
datatype "int"|"float"|"char"
A [a-zA-Z]
digit [0-9]
header "#include<"
had  ".h>"
ope ">="|"<="|"=="|">"|"<"|"!="|"-="|"+="|"*="|"/="
conope "&&"|"||"
unary     "++"|"--"
dot [.]
decimal {digit}{digit}*{dot}{digit}{digit}*
sign [-]
negnum {sign}{digit}+
%%
[\t ]
[\n] {yylineno++;}
{ope}      { insert(yytext,'O',yylineno,"NA"); return OP;}
{conope}       { insert(yytext,'O',yylineno,"NA"); return COP;}
{unary}            { insert(yytext,'O',yylineno,"NA"); return UN;}
{header}(.)*{had} {  return HEAD;}
while              {  insert(yytext,'K',yylineno,"NA");return WHILE;}
for                {  insert(yytext,'K',yylineno,"NA");return FOR;}
if        {  insert(yytext,'K',yylineno,"NA");return IF;}
else               {  insert(yytext,'K',yylineno,"NA");return ELSE;}
break              {  insert(yytext,'K',yylineno,"NA");return BREAK;}
return           {  insert(yytext,'K',yylineno,"NA");return RETURN;}
```

lex.l yacc.y

```
break             {  insert(yytext,'K',yylineno,"NA");return BREAK;}
return           {  insert(yytext,'K',yylineno,"NA");return RETURN;}
{datatype}         { strcpy(data,yytext); insert(yytext,'D',yylineno,"NA");return TYPE;}
main        {  insert(yytext,'M',yylineno,"NA");return ID;}
";"         {  insert(yytext,'P',yylineno,"NA");return SEMI;}
","         {  insert(yytext,'P',yylineno,"NA");return COMMA;}
"("         {  insert(yytext,'P',yylineno,"NA");return BO;}
")"         {  insert(yytext,'P',yylineno,"NA");return BC;}
"{"         {  insert(yytext,'P',yylineno,"NA");return FO;}
"}"         {  insert(yytext,'P',yylineno,"NA");return FC;}
{digit}+   {insert(yytext,'N',yylineno,"Int");  return NUM;}
{decimal}   {insert(yytext,'N',yylineno,"Float");  return NUM;}
{negnum}    {insert(yytext,'N',yylineno,"Int");  return NUM;}
{A}({A}|{digit})* { insert(yytext,'V',yylineno,data); return ID;}
\/\/.*          ;
\/\*(.\n)*\*\/  ;
.                return yytext[0];
%%


void insert(char *yytext,char t,int l, char *type)
{
  int len1 = strlen(yytext);
  int i;
    char token[20];
    struct Storage *symbol,*temp,*nextptr;
    nextptr=head;
    switch(t)
    {
        case 'V':
            strcpy(token,"Identifier");
            break;
        case 'N':
              strcpy(token,"Constants");
              break;
        case 'M':
              strcpy(token,"Function");
              break;
        case 'O':
              strcpy(token,"Operator");
              break;
```

lex.l yacc.y

```
                    break;
            case 'O':
                    strcpy(token,"Operator");
                    break;
            case 'K':
                    strcpy(token,"Keyword");
                    break;
            case 'P':
                    strcpy(token,"Puntuators");
                    break;
            case 'D':
                    strcpy(token,"Datatype");
                    break;

        }

            for(i=0;i<var_cnt;i++,nextptr=nextptr->next)
            {
                symbol = nextptr;
                if(strcmp(symbol->name,yytext)==0)
                    break;
            }
            if(i==var_cnt)
            {
                temp = (struct Storage*)malloc(sizeof(struct Storage));
                temp->name = (char*)malloc((len1+1)*sizeof(char));
                strcpy(temp->name,yytext);
                strcpy(temp->token,token);
                temp->val = i;
                temp->li = l;
                temp->next = NULL;
                strcpy(temp->Data,type);
                if(var_cnt==0)
                    head = temp;
                else
                    symbol->next = temp;
                var_cnt++;

            }


```

lex.l yacc.y

```
            temp->next = NULL;
            strcpy(temp->Data,type);
            if(var_cnt==0)
                head = temp;
            else
                symbol->next = temp;
            var_cnt++;

        }


}
 void print()
 {
  struct Storage *nextp=head;
  for(;nextp!=NULL;nextp=nextp->next)\
      if(strcmp(nextp->token,"Identifier")==0 || strcmp(nextp->token,"Constants")==0)
            printf("%s \t %s \t %d \t %s\n",nextp->name,nextp->token,nextp->li,nextp->Data);
 }


```

Yacc.y

lex.l yacc.y

```
%{
#include<stdio.h>
#include<stdlib.h>

%}

%token ID NUM TYPE HEAD WHILE OP COP FOR UN IF ELSE RETURN ASSIGNMENT BREAK SEMI COMMA BO BC FO FC

%%
S : Header Start {printf("\nParsing Completed\n");
                  printf("Lexeme\tDescription\tLine no\tType\n");
                  printf("-------------------------------------------------------------\n");
      print();
      exit(0);
     }
Header : HEAD Header
       | |
       ;

Start : TYPE ID BO ARG BC FO BODY FC Start
      |
      ;

ARG : TYPE ID
    | TYPE ID COMMA ARG
    |
    ;

BODY : Bb BODY
     |
     |error
     ;

Bb :  R SEMI
   | DEC SEMI
   | WLOOP
   | FLOOP
   | IFEL
   | EXP SEMI
   | RET SEMI
   | BREAK SEMI
   | SEMI ;

single : R SEMI
       |DEC SEMI
       |WLOOP
       |FLOOP
       |RET SEMI
       |BREAK SEMI
       |EXP SEMI
```

lex.l yacc.y

```
49              |BREAK SEMI
50              |EXP SEMI
51              |COND SEMI
52              | IFEL
53              | SEMI ;
54
55      RET :  RETURN K
56          | error ;
57      K : ID
58        |NUM ;
59
60      R : ID '=' A
61        | A
62        | error ;
63      A : A '+' B
64        | A '-' B
65        | B;
66      B : B '*' C
67        | B '/' C
68        | C ;
69      C : ID UN
70        | ID
71        | NUM
72        | '(' A ')' ;
73
74      DEC : TYPE VAR;
75      VAR : TT COMMA VAR
76          | TT
77          | error;
78      TT : ID
79         | ID '=' ID
80         | ID '=' NUM;
81
82      WLOOP : WHILE BO COND BC WDEF
83            | error;
84      WDEF : FO BODY FC
85           | single ;
86
87      COND : EXP COP COND
88           | EXP
89           | error;
90      EXP :  ID
91          | ID OP ID
92          | ID OP NUM
93          |NUM
94          | error;
95
96      FLOOP : FOR BO Aa SEMI Bb SEMI Cc BC FDEF ;
97      Aa : TYPE R
98         | R
```

lex.l | yacc.y

```
91        | ID OP ID
92        | ID OP NUM
93        |NUM
94        | error;
95
96    FLOOP : FOR BO Aa SEMI Bb SEMI Cc BC FDEF ;
97    Aa : TYPE R
98       | R
99       |
100      ;
101   Bb : COND
102      |
103      ;
104   Cc : R
105      |
106      ;
107   FDEF : FO BODY FC
108        | single ;
109
110   IFEL : IF BO COND BC IDEF
111        | IF BO COND BC IDEF ELSE IDEF ;
112
113   IDEF : FO BODY FC
114        | single ;
115
116   %%
117
118   extern int yylineno;
119   extern int yytext;
120   extern void print();
121   void yyerror()
122    {
123     printf("Invalid expression at %d \n",yylineno);
124    }
125    extern FILE *yyin;
126    extern FILE *yyout;
127   int main()
128    {
129
130     yyin=fopen("input.txt","r");
131     yyparse();
132
133    }
134
```

Test Case 1:

```
    ├----+----1----+----2----
 1  //INPUT
 2
 3  #include<stdio.h>
 4
 5  int main() {
 6          int a, b;
 7          a = 5;
 8          b = a + 2;
 9          int c;
10          c=3*2+5;
11          if(c>=10)
12             c+=10;
13          else
14             c-=10;
15
16          return 0;
17  }
18
19  //OUTPUT
20  //Parse Complete
21
```

```
G:\6th sem proj\CD_Lab-master\Parser>a.exe

Parsing Completed
Lexeme  Description     Line no Type
-----------------------------------------------------------
a       Identifier      6       int
b       Identifier      6       int
5       Constants       7       Int
2       Constants       8       Int
c       Identifier      9       int
3       Constants       10      Int
10      Constants       11      Int
0       Constants       16      Int

G:\6th sem proj\CD_Lab-master\Parser>
```

Test Case 2:

```
    ├----+----1----+----2----+----3---
 1  //INPUT
 2
 3  #include<stdio.h>
 4
 5  int main() {
 6          float a = 3.0;
 7          while(a > 1) a /= 1.5;
 8          return 0;
 9  }
10
11  //OUTPUT
12  //Parse Complete
13
```

```
G:\6th sem proj\CD_Lab-master\Parser>a.exe

Parsing Completed
Lexeme  Description     Line no Type
-----------------------------------------------------------
a       Identifier      6       float
3.0     Constants       6       Float
1       Constants       7       Int
1.5     Constants       7       Float
0       Constants       8       Int

G:\6th sem proj\CD_Lab-master\Parser>
```

Test Case 3:

```
    ├----+----1----+----2----+----3---
 1  //INPUT
 2
 3  #include<stdio.h>
 4
 5  int main() {
 6          int int a;
 7          return 0;
 8  }
 9
10  //OUTPUT
11  //Parse Error in line 4
```

```
G:\6th sem proj\CD_Lab-master\Parser>a.exe
Invalid expression at 6

Parsing Completed
Lexeme  Description     Line no Type
-----------------------------------------------------------
a       Identifier      6       int
0       Constants       7       Int

G:\6th sem proj\CD_Lab-master\Parser>
```

Test Case 4:

```
1  //INPUT
2
3  #include<stdio.h>
4
5  int main() {
6          float x
7          return 0;
8  }
9
10 //OUTPUT
11 //Parse Error in line 4
```

```
G:\6th sem proj\CD_Lab-master\Parser>a.exe
Invalid expression at 7

Parsing Completed
Lexeme  Description     Line no Type
---------------------------------------------------------
x       Identifier      6       float
0       Constants       7       Int

G:\6th sem proj\CD_Lab-master\Parser>
```

Test Case 5:

```
1  //INPUT
2
3  #include<stdio.h>
4
5  int main() {
6          for(i = 0; i < 10; i++) {
7                  i *= -2;
8          }
9          return 0;
10 }
11
12 //OUTPUT
13 //Parse Complete
```

```
G:\6th sem proj\CD_Lab-master\Parser>a.exe

Parsing Completed
Lexeme  Description     Line no Type
---------------------------------------------------------
i       Identifier      6       int
0       Constants       6       Int
10      Constants       6       Int
-2      Constants       7       Int

G:\6th sem proj\CD_Lab-master\Parser>
```

Test Case 6:

```
1  //INPUT
2
3  #include<stdio.h>
4
5  int main() {
6          int a;
7          if(1) {
8                  a++;
9                  if(a==10) break;
10         }
11         else else if(a==1) a=0;
12         return 0;
13 }
14
15 //OUTPUT
16 //Parse Error in line 9
```

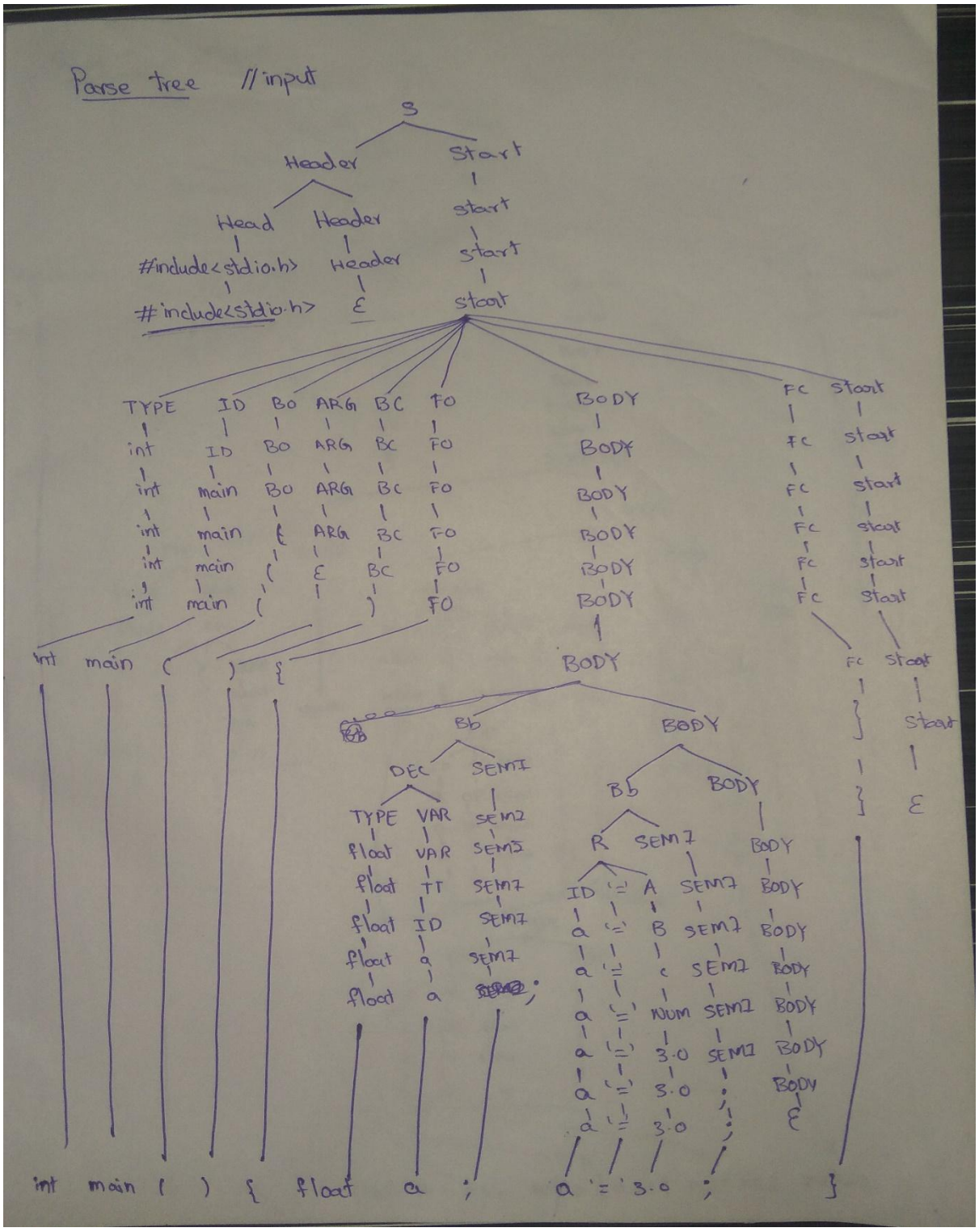
```
G:\6th sem proj\CD_Lab-master\Parser>a.exe
Invalid expression at 11

Parsing Completed
Lexeme  Description     Line no Type
---------------------------------------------------------
a       Identifier      6       int
1       Constants       7       Int
10      Constants       9       Int
0       Constants       11      Int

G:\6th sem proj\CD_Lab-master\Parser>
```

Parse Tree:

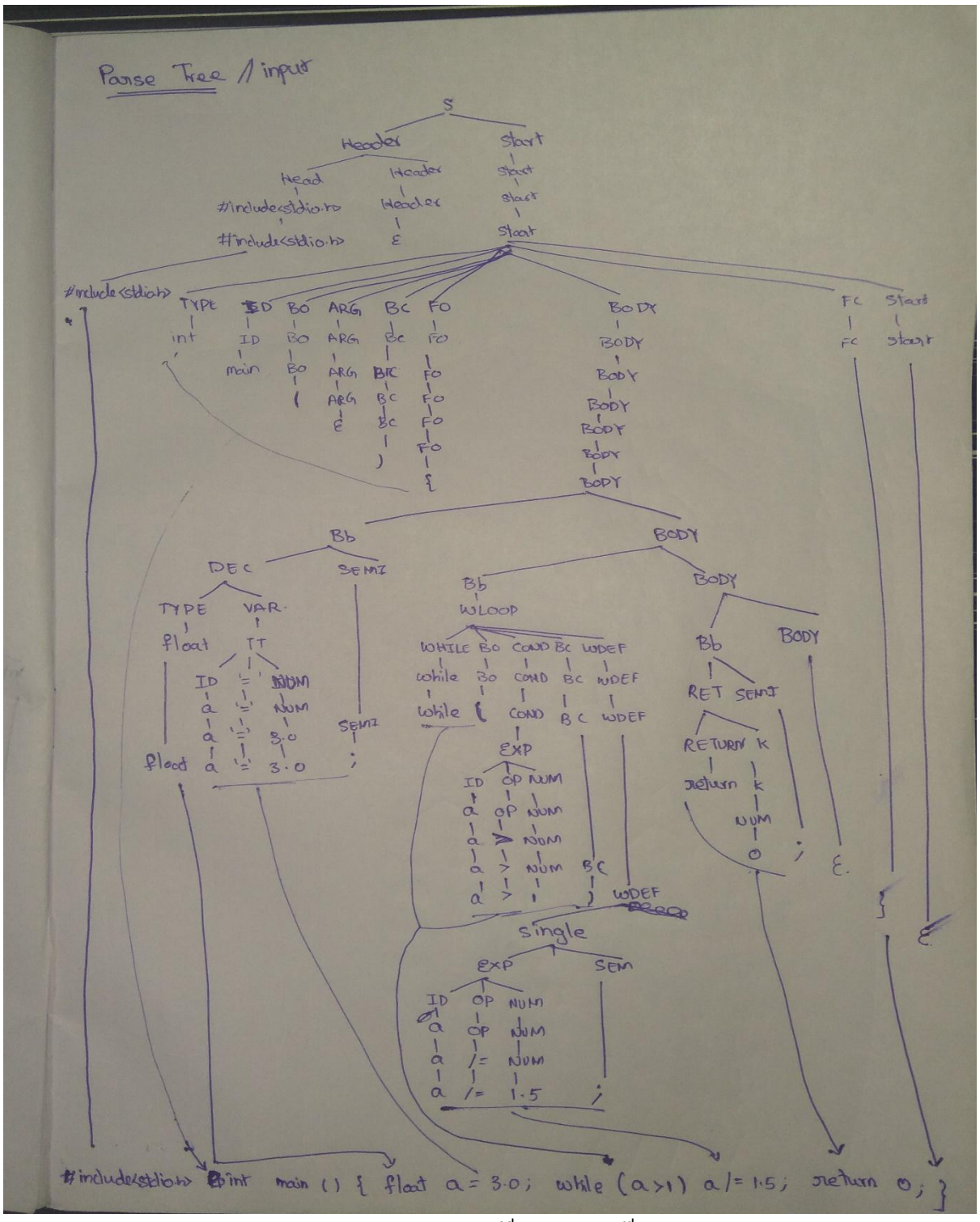
```
#include<stdio.h>
int main(){
float a;
a=3.0;
}
```



For the above parse tree first Bb is done next BODY after that FC and next start.

Test Case 2:

```
int main() {
        float a = 3.0;
        while(a > 1) a /= 1.5;
        return 0;
}
```



For the above parse tree first Bb is done next 2nd Bb next 3rd Bb then last FC and start.

Conclusion

The second phase of this project is the syntax analysis phase that takes tokens generated by lex analyzer as input and displays the errors if any and updates the symbol table.

The output will be the parse tree which is given as input to the next phase i.e. semantic checker and the symbol table built is used during checking.