

Tinylang

*Design and implementation of a
tiny programming language
in **Java**.*

Andimon

Contents

Contents	2
o Introduction	4
o.1 A tinylang program	4
o.2 Using a tinylang's compiler	5
o.3 Syntax rules of <i>tinylang</i> in <i>EBNF</i>	6
o.4 Outline	7
1 Table-Driven Lexer	9
1.1 Specification : micro-syntax	9
1.2 Table-driven lexer	19
1.3 Implementation in Java	21
1.4 Test programs	22
2 Hand-Crated LL(k) Parser	25
2.1 The parser	25
2.2 Design of an AST	25
2.3 Recursive Descent	28
2.4 Parse tree of a sample tinylang program	50
2.5 Implementation in Java	51
2.6 Testing	52
3 AST XML Generation Pass	54
3.1 ENUM-Based Visitor's Design Pattern	54
3.2 Design	56
3.3 Implementation in Java	60
3.4 Testing	60
4 Semantic Analysis	62
4.1 Design	62
4.2 Implementation in Java	69
4.3 Testing	69

5	Interpreter	72
5.1	Design	72
5.2	Implementation in Java	78
5.3	Testing	78
5.4	Future implementation	81
6	Implementation	82
6.1	Lexer	82
6.2	Parser	98
6.3	XML generation	110
6.4	Semantic Analyser	115
6.5	Interpreter	126
6.6	GitHub Repo	134

o | Introduction

o.1 | A tinylang program

```
1 fn Sq(x:float) -> float {
2     return x*x;
3 }
4 fn XGreaterY(x:float , y:float) -> bool {
5     let ans:bool=true ;
6     if (y>x) {ans=false ; }
7     return ans ;
8 }
9 // Same functionality as function above but using less code
10 fn XGreaterY_2 (x:float , y:float) -> bool {
11     return x>y ;
12 }
13
14 fn AverageOfThree (x:float , y:float , z:float ) -> float {
15     let total : flaot = x+y+z;
16     return total/3;
17 }
18
19 /*
20 * Same functionality as function above but using less code .
21 * Note the use o f the brackets in the expression following
22 * the return statement .
23 */
24
25 fn AverageOfThree_2 (x:float , y:float , z:float) -> float {
26     return (x+y+z)/3 ;
27 }
28 //Execution (program entry point) starts at the first statement
29 // that is not a function declaration .
30 let x : float = 2.4 ;
31 let y : float = Sq(2.5);
32 let z : float = Sq (x);
33 print y ; //6.25
34 print x * z ; //13.824
35 print XGreaterY (x , 2.3); // true
36 print XGreaterY 2(Sq(1.5),y); // false
```

```
37 print AverageOfThree (x,y,1.2); //3.28
```

Listing 1: A semantically and syntactically correct program in *TinyLang*.

0.2 | Using a tinylang's compiler

See folder (*binary*) inside project directory.

- Place the `tinylang.jar` and `program.tl` in the same directory.

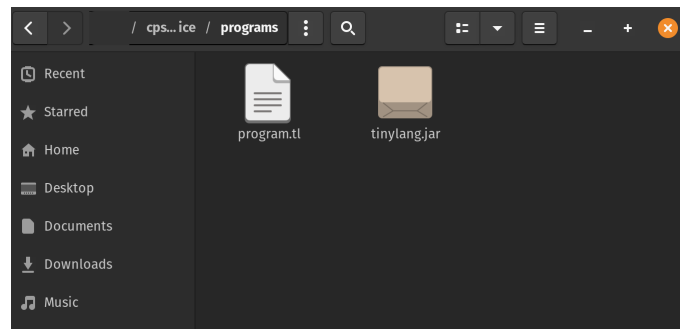


Figure 0.1: Program and compiler binary in same directory.

- Compile `program.tl` using `tinylang` by running command
`java -jar tinylang program`
- We get a menu:

```
1- Produce tokens of program (lexer)
2- Produce an XML representation of program (parser+xml generation pass)
3- Interpret program
q- Exit
```

Figure 0.2: Menu with three options

```
Choose your option : 1
<TOK_FN, (lexeme:"fn", line number:1)>
<TOK_IDENTIFIER, (lexeme:"isDigit", line number:1)>
<TOK_LEFT_ROUND_BRACKET, (lexeme:"(", line number:1)>
<TOK_IDENTIFIER, (lexeme:"x", line number:1)>
<TOK_COLON, (lexeme:":", line number:1)>
<TOK_INT_TYPE, (lexeme:"int", line number:1)>
<TOK_RIGHT_ROUND_BRACKET, (lexeme:")", line number:1)>
<TOK_RIGHT_ARROW, (lexeme:"->", line number:1)>
<TOK_BOOL_TYPE, (lexeme:"bool", line number:1)>
<TOK_LEFT_CURLY_BRACKET, (lexeme:"{", line number:1)>
<TOK_IF, (lexeme:"if", line number:2)>
<TOK_LEFT_ROUND_BRACKET, (lexeme:"(", line number:2)>
<TOK_LEFT_ROUND_BRACKET, (lexeme:"(", line number:2)>
<TOK_IDENTIFIER, (lexeme:"x", line number:2)>
<TOK_RELATIONAL_OP, (lexeme:"==", line number:2)>
<TOK_INT_LITERAL, (lexeme:"0", line number:2)>
<TOK_RIGHT_ROUND_BRACKET, (lexeme:")", line number:2)>
<TOK_ADDITIVE_OP, (lexeme:"or", line number:2)>
<TOK_LEFT_ROUND_BRACKET, (lexeme:"(", line number:2)>
<TOK_IDENTIFIER, (lexeme:"x", line number:2)>
<TOK_RELATIONAL_OP, (lexeme:"==", line number:2)>
```

Figure 0.3: Option 1 : Lexer

```

Choose your option : 2
<TinyLangProgram>
  <function declaration>
    <id type="BOOL">isDigit<\id>
    <parameters>
      <parameters>
        <id type="INTEGER">x<\id>
      <\parameters>
    <\parameters>
    <block>
      <if statement>
        <binary Op="or">
          <binary Op="==">
            <id>x<\id>
            <integer literal>0<\integer literal>
          <\binary>
          <binary Op="or">
            <binary Op="==">
              <id>x<\id>
              <integer literal>1<\integer literal>
            <\binary>
            <binary Op="or">
              <binary Op="==">
                <id>x<\id>
                <integer literal>2<\integer literal>
              <\binary>
              <binary Op="or">
                <binary Op="==">
                  <id>x<\id>
                  <integer literal>3<\integer literal>
                <\binary>
              <binary Op="or">
            <\binary>
          <\binary>
        <\binary>
      <\if statement>
    <\block>
  <\function declaration>
<\TinyLangProgram>

```

Figure 0.4: Option 2: XML \leftrightarrow AST

```

Choose your option : 3
program is semantically correct
false
'1'

```

Figure 0.5: Option 3: Confirms that the program is semantically correct and provide an interpretation of it

0.3 | Syntax rules of *tinylang* in EBNF

```

1 <Letter> ::= [A-Za-z]
2 <Digit> ::= [0-9]
3 <Printable> ::= [\x20-\x7E]
4 <Type> ::= 'float' | 'int' | 'bool' | 'char'
5 <BooleanLiteral> ::= 'true' | 'false'
6 <IntegerLiteral> ::= <Digit>{<Digit>}
7 <FloatLiteral> ::= <Digit>{<Digit>}'.'<Digit>{<Digit>}
8 <CharLiteral> ::= "'" <Printable> "'"
9 <Literal> ::= <BooleanLiteral> | <IntegerLiteral> | <FloatLiteral> |
  <CharLiteral>
10 <Identifier> ::= ('_' | <Letter>){ '_' | <Letter> | <Digit> }
11 <MultiplicativeOp> ::= '*' | '/' | 'and'
12 <AdditiveOp> ::= '+' | '-' | 'or'
13 <RelationOp> ::= '<' | '>' | '==' | '<=' | '>='
14 <ActualParams> ::= <Expression> { ',' <Expression> }
15 <FunctionCall> ::= <Identifier> '(' [<ActualParams> ] ')'
16 <SubExpression> ::= '(' <Expression> ')'

```

```

17 <Unary> ::= ('+' | '-' | 'not') <Expression>
18 <Factor> ::= <Literal> | <Identifier> | <FunctionCall> | <
    SubExpression> | <Unary>
19 <Term> ::= <Factor> {<MultiplicativeOp> <Factor>}
20 <SimpleExpr> ::= <Term> {<AdditiveOp> <Term>}
21 <Expression> ::= <SimpleExpr> {<RelationalOp> <SimpleExpr>}
22 <Assignment> ::= <Identifier> '=' <Expression>
23 <VariableDecl> ::= 'let' <Identifier> ':' <Type> '=' <Expression>
24 <PrintStatement> ::= 'print' <Expression>
25 <RtrnStatement> ::= 'return' <Expression>
26 <IfStatement> ::= 'if' '(' <Expression> ')' <Block> ['else' <Block>]
27 <ForStatement> ::= 'for' '(' [<VariableDecl> ';' <Expression> ';' [<
    Assignment>] ')' <Block>
28
29 <WhileStatement> ::= 'while' '(' <Expression> ')' <Block>
30
31 <FormalParam> ::= <Identifier> ':' <Type>
32
33 <FormalParams> ::= <FormalParam> {',' <FormalParam>}
34
35 <FunctionDecl> ::= 'fn' <Identifier> '(' [<FormalParams>] ')' '-'> <
    Type> <Block>
36
37 <Statement> ::= <VariableDecl> ';'
38               | <Assignment> ';'
39               | <PrintStatement> ';'
40               | <IfStatement>
41               | <ForStatement>
42               | <WhileStatement>
43               | <RtrnStatement> ';'
44               | <FunctionDecl>
45               | <Block>
46 <Block> ::= '{' { <Statement> } '}'
47 <Program> ::= '{ <Statement> }'

```

Listing 2: EBNF capturing the Syntax Rules of *tinylang*.

0.4 | Outline

- **tinylang** is written in Java and built with the following components:
 - **Lexer:** Takes a whole program as one string and breaks it down into a sequence of tokens.
 - **Parser:** Takes all tokens produced by the lexer and produces an abstract syntax tree, highlighting the logic of the whole program by parsing the program using the EBNF rules shown above and highlighting syntactical errors in the process.

- **XML generator:** Produces an indented XML highlighting the structure of the tree (indentation) and all its nodal properties (tags).
- **Semantic analyser:** Used to perform semantic checks such as type checking, checking if a function returns, handling undeclared functions/variables, etc.
- **Interpreter:** Used to traverse the program's abstract syntax tree and simulate a live execution of the program.

Task 1 | Table-Driven Lexer

1.1 | Specification : micro-syntax

Task: Identify rules (micro-syntax) to validate if a sequence of characters is a *lexeme* (the smallest lexical unit allowed in the language).

We can construct an infinite number of lexemes (e.g. \mathbb{Z}). To gain control, we categorise the lexemes into a finite number of groups and then write rules for each group to verify if a sequence of characters is a lexeme in the group.

The task of choosing groups/types is not deterministic; however, a typical strategy (and the one used in this implementation) is:

- Place keywords (reserved/special words in the language) in separate groups:

Group	Lexeme(s)
TOK_PRINT	print
TOK_IF	if
TOK_ELSE	else
TOK_FOR	for
TOK_WHILE	while
TOK_FN	fn
TOK_RETURN	return
TOK_INT_TYPE	int
TOK_FLOAT_TYPE	float
TOK_BOOL_TYPE	bool
TOK_CHAR_TYPE	char
TOK_LET	let
TOK_RIGHT_ARROW	->

Table 1.1: Keywords and their respective group.

- Similarly, place punctuation symbols in separate groups:

Group	Lexeme(s)
TOK_LEFT_ROUND_BRACKET	(
TOK_RIGHT_ROUND_BRACKET)
TOK_LEFT_CURLY_BRACKET	{
TOK_RIGHT_CURLY_BRACKET	}
TOK_COMMA	,
TOK_COLON	:
TOK_SEMICOLON	;

Table 1.2: Different punctuation symbols and their respective group:

- Put operators of similar type into one group (we categorise them according to EBNF spec):

Group and Lexeme(s)	
TOK_MULTIPLICATIVE_OP	'*' '/' 'and'
TOK_ADDITIVE_OP	'+' '-' 'or'
TOK_RELATIONAL_OP	'<' '>' '==' '=' '<=' '>=' !

Table 1.3: Different operations and their respective group

- Group identifiers (of variables/functions) into one group and group literals by their respective data type.

TokenType	Lexeme(s)
TOK_IDENTIFIER	('_' <Letter>) ('_' <Letter> <Digit>)*
TOK_BOOLEAN_LITERAL	'true' 'false'
TOK_INTEGER_LITERAL	<Digit>{<Digit>}
TOK_FLOAT_LITERAL	<Digit>{<Digit>}.<Digit>{<Digit>}
TOK_CHAR_LITERAL	' ' <Printable> ' '

Table 1.4: Tokens and their respective group(s)

- Special lexemes :

TokenType	Lexeme(s)
TOK_SKIP	whitespace characters //{<printable>} \n /*{<printable>}*/
TOK_EOF	EOF

Table 1.5: Special lexemes and their respective group(s)

Having all the possible groups in hand, we can construct an automaton capturing tinylang's syntax by designing sub-automata for each group and then merging the automata together at the starting state.

1.1.1 | Constructing a deterministic finite-state automaton (DFSA) that recognises all possible lexemes

Let G be the set consisting of all groups described in Tables 1.1, 1.2, 1.3, 1.4, and 1.5, and let l represent some lexeme.

We note that groups **should** partition the set of all lexemes.

- All groups cover all possible lexemes in tinylang: $\bigcup_{g \in G} \{l : l \in g\}$ is the set of all possible lexemes.
- Pairwise disjoint: $\forall g_1, g_2 \in G \implies g_1 \cap g_2 = \phi$

NB: The specification of the groups described in Tables 1.1, 1.2, 1.3, 1.4 and 1.5 contradicts the pairwise disjoint property since there exist clashes, for example, lexeme `if` can be in both groups `TOK_IDENTIFIER` and `TOK_IF`. In this case, priority is trivially given to the group `TOK_IF`. During the design stage, attention is given to these types of non-disjoint clashes to ensure that the groups partition the set of all possible lexemes.

1.1.2 | Design of the sub-automata

1.1.2.1 | Important consideration

We want the sub-automata to be deterministic finite-state automata:

- **Deterministic** : Given a state and input, we deterministically know what the next state is (i.e., given a state and input, there are no two distinct transitions taking us to different states).
- **Finite** : Gives us a handle on all possible lexemes in a group.

1.1.2.2 | Classifier Table

While sketching the automata on pen and paper and keeping in mind the EBNF rules equivalent inputs used for the sub-automata where classified as follows:

Input	Value(s)	ASCII-EQUIVALENT
letter	a,b,...,z,A,B,...,Z	[0x4a,0x5a],[0x61,0x7a]
digit	0,1,2,...,9	[0x30,0x39]
_	_	0x5f
/	/	0x2f
*	*	0x2a
<	<	0x3c
+	+	0x2b
-	-	0x2d
=	=	0x3d
!	!	0x21
.	.	0x2e
'	'	0x27
punct	(), : , { }	{ 0x28, 0x29, 0x2c, 0x3a, 0x3b, 0x7b, 0x7d }
other_printable	space,...,~	[0x20,0x7e] excluding the ASCII codes above

Table 1.6: Classifier table

Note: All the input categories are pairwise disjoint. This ensures that the automata are deterministic.

Also note: In the following sub-automata shown in figures 1.1, 1.2, 1.3, 1.4, 1.5 and 1.6 input any is an abbreviation for:

letter|digit|'_|'|/|'|*|'|<|'|>|'|+|'| -|'|='|'|!|'|'.|'|
|punct|other_printable i.e. all the printable characters allowed in
tinylang given by ASCII range [0x20-0x7e] (see section 0.3).

We start considering different groups:

- Group TOK_CHAR_LITERAL

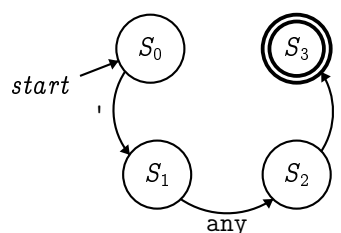
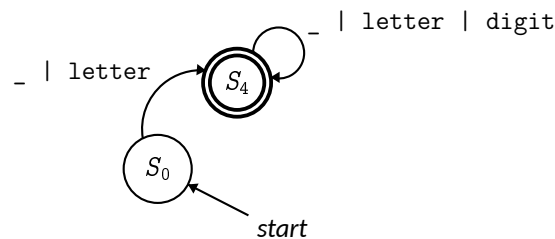


Figure 1.1: dfssa recognising lexemes in group TOK_CHAR_LITERAL

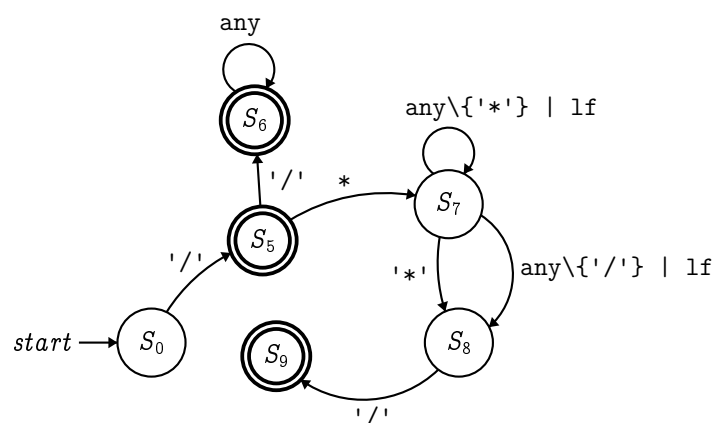
- Sequences of characters leading to state 3 are lexemes in group TOK_CHAR_LITERAL.
- Sequences of characters leading to States 0, 1 and 2 are invalid

- **NB :** This automata only capture lexeme(s) that are in group TOK_CHAR_LITERAL (i.e. no non-disjoint clashes).
- Group TOK_IDENTIFIER :

Figure 1.2: dfsa recognising lexemes in group *TOK_IDENTIFIER*

NB : This automaton also recognises lexemes that are in groups: TOK_LET, TOK_IF, TOK_ELSE, TOK_FOR, TOK_WHILE, TOK_RETURN, TOK_INT_TYPE, TOK_FLOAT_TYPE, TOK_BOOL_TYPE, TOK_CHAR_TYPE and TOK_BOOLEAN_LITERAL. We give precedence to these groups i.e. if a lexeme that is identified by this automaton is in one of these groups we consider it that it is in that group not in group TOK_IDENTIFIER (in simpler terms an identifier cannot be a reserved word). Note that these keyword groups can be given the same precedence since they are all pairwise disjoint.

- Group TOK_SKIP :

Figure 1.3: dfsa recognising lexemes in group *TOK_SKIP*

- Since the input ' / ' is utilised to identify a lexeme in group TOK_SKIP, the same automaton can capture lexeme ' / ' in group TOK_MULTIPLICATIVE_OP (this ensures that when we merge the sub-automata the main automaton remains deterministic).

- Sequence of character(s) leading to state 5 is lexeme in group TOK_MULTIPLICATIVE_OP. Sequence of character(s) leading to states 6 and 9 are lexemes in group TOK_SKIP.
- Sequence of character(s) leading to states 0,7 and 8 are invalid.
- Groups TOK_LEFT_ROUND_BRACKET, TOK_RIGHT_ROUND, TOK_LEFT_CURLY_BRACKET, TOK_RIGHT_CURLY_BRACKET, TOK_COMMA, TOK_COLON and TOK_SEMICOLON :

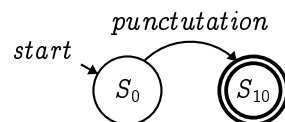


Figure 1.4: dfsa recognising lexemes in punctuation groups

- Since no punctuation is used as an initial input (from starting state) in any of the other sub-automata we can simplify the automaton by capturing all the punctuation symbols in one state ensuring that the main automaton remains deterministic when merging.
- A checker function then checks what type of punctuation it is and matches it accordingly.
- We conclude that a character leading to state 10 is a lexeme in one of the following groups : TOK_LEFT_ROUND_BRACKET, TOK_RIGHT_ROUND, TOK_LEFT_CURLY_BRACKET, TOK_RIGHT_CURLY_BRACKET, TOK_COMMA, TOK_COLON and TOK_SEMICOLON.
- Groups TOK_INT and TOK_FLOAT:

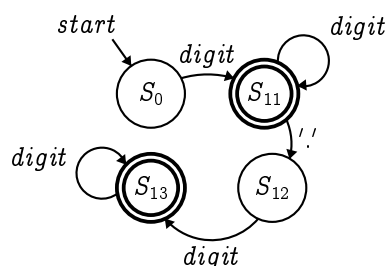


Figure 1.5: dfsa recognising lexemes in group TOK_INT and TOK_FLOAT

- Sequences of characters leading to States 11 and 13 are in groups TOK_INT and TOK_FLOAT respectively.
- Sequences of characters leading to States 0 and 12 are invalid.

- **NB** : Since 12 is rejecting, floating points like 12., 0. **are not allowed** i.e. the fractional part must contain 1 or more digit. Example of good floating point numbers are 12.3, 432.124214 etc. This strategy is taken since it conforms to the EBNF rules.
- Groups TOK_ADDITIVE_OP, MULTIPLICATIVE_OP, TOK_RELATIONAL_OP and TOK_RIGHT_ARROW

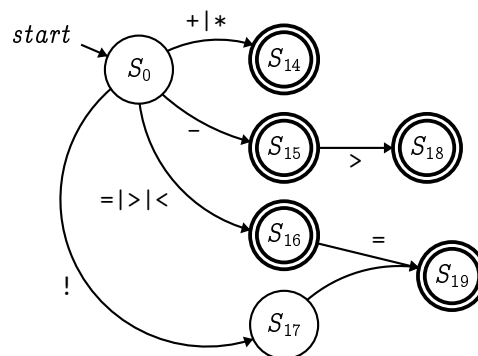


Figure 1.6: dfsa recognising lexemes in groups all operator groups and group TOK_RIGHT_ARROW

- Sequences of characters leading to State 14 are in groups TOK_ADDITIVE_OP or TOK_MULTIPLICATIVE_OP. We use a checker function to check the operator and assign the appropriate groups. Sequence of characters leading to State 15 are in group TOK_ADDITIVE_OP. Sequences of characters leading to State 16 are in group TOK_RELATIONAL_OP. Sequence of characters leading to State 17 is invalid. Sequence of characters leading to State 17 is in group TOK_RIGHT_ARROW. Sequence of characters leading to State 19 are in group TOK_RELATIONAL_OP.
- **NB** : The multiplicative op ' / ' is already dealt with in automaton shown in figure 1.3

STATES	POSSIBLE GROUP(S)
S0	invalid
S1	invalid
S2	invalid
S3	TOK_CHAR_LITERAL
S4	TOK_IDENTIFIER, TOK_FN, TOK_BOOL_TYPE, TOK_INT_TYPE, TOK_FLOAT_TYPE, TOK_BOOLEAN_LITERAL, TOK_NOT, TOK_LET TOK_CHART_TYPE, TOK_IF, TOK_ELSE, TOK_WHILE, TOK_FOR, TOK_PRINT, TOK_RETURN, TOK_MULTIPLICATIVE_OP, TOK_ADDITIVE_OP
S5	TOK_MUTLIPLICATIVE_OP
S6	TOK_SKIP
S7	invalid
S8	invalid
S9	TOK_SKIP
S10	TOK_LEFT_ROUND_BRACKET, TOK_RIGHT_ROUND_BRACKET, TOK_LEFT_CURLY_BRACKET, TOK_RIGHT_CURLY_BRACKET, TOK_COMMA, TOK_COLON, TOK_SEMICOLON
S11	TOK_INTEGER_LITERAL
S12	invalid
S13	TOK_FLOAT_LITERAL
S14	TOK_ADDITIVE_OP, TOK_MULTIPLICATIVE_OP
S15	TOK_ADDITIVE_OP
S16	TOK_RELATIONAL_OP
S17	invalid
S18	TOK_RIGHT_ARROW
S19	TOK_RELATIONAL_OP
SE	invalid

Table 1.7: Possible groups associated with each state

1.1.4 | Transition Table

NB: Starting state and Error state denoted by So and SE respectively.

<i>state</i> \ <i>input</i>	letter	digit	_	/	*	<	>	+	-	=	!	.	,	punct	other_ printable	If
So	S4	S11	S4	S5	S14	S16	S16	S14	S15	S16	S17	SE	S1	S10	SE	SE
S1	S2	S2	S2	S2	S2	S2	S2	S2	S2	S2	S2	S2	S2	S2	S2	SE
S2	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	S3	SE	SE	SE
S3	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE
S4	S4	S4	S4	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE
S5	SE	SE	SE	S6	S7	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE
S6	S6	S6	S6	S6	S6	S6	S6	S6	S6	S6	S6	S6	S6	S6	S6	SE
S7	S7	S7	S7	S7	S8	S7	S7	S7	S7	S7	S7	S7	S7	S7	S7	S7
S8	S7	S7	S7	S9	S7	S7	S7	S7	S7	S7	S7	S7	S7	S7	S7	S7
S9	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE
S10	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE
S11	SE	S11	SE	SE	SE	SE	SE	SE	SE	SE	SE	S12	SE	SE	SE	SE
S12	SE	S13	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE
S13	SE	S13	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE
S14	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE
S15	SE	SE	SE	SE	SE	SE	S18	SE	SE	SE	SE	SE	SE	SE	SE	SE
S16	SE	SE	SE	SE	SE	SE	SE	SE	SE	S19	SE	SE	SE	SE	SE	SE
S17	SE	SE	SE	SE	SE	SE	SE	SE	SE	S19	SE	SE	SE	SE	SE	SE
S18	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE
S19	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE
SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE	SE

Table 1.8: Tabular encoding of automaton shown in figure 1.7

The transition table can be read as a transition function δ . Let S and I be the set of states and inputs respectively and the transition function is defined as $\delta : S \times I \rightarrow S$ where

- The first row in the transition table is given by $\delta(S0, i), i \in I$
- The second row in the transition table is given by $\delta(S1, i), i \in I$
- \vdots
- The last row in the transition table is given by $\delta(SE, i), i \in I$

1.2 | Table-driven lexer

1.2.1 | Tokens

The job of the lexer is to generate to take the program as one big string and break it down into a sequence of tokens.

A token is of the form `<TokenType; Attribute>` where the **token type** is just the name of one of the groups shown in tables 1.1, 1.2, 1.3, 1.4, 1.3 and the **attribute** can just be the lexeme associated to that group or it can include other statistics such as in which line number the lexeme is.

Since we specified our micro-syntax in tabular form, we proceed to build a table-driven lexer. The algorithm of the lexer for generating sequences of tokens is given in the following subsection.

1.2.2 | Generating sequence of tokens (PSEUDOCODE)

```

1 int currentIndex <- 0;
2 int lineNumber <- 0;
3 String program;
4 List tokens;
5 //program is empty -> list is one just token EOF
6 if(program.length==0)
7     tokens.add((EOF,""));
8 //otherwise if program not empty
9 while(currentIndex<program.length):
10     token = getNextToken(program)
11     //set line number
12     token.setLineNumber(getLineNumber(tinyLangProgram))
13     //if the next token is not a comment add it to list
14     if token.type != TOK_SKIP:

```

```
15         tokens.add(token)
16
17
18
19 /* Method getNextToken(program) includes includes
20 * the ideas (initialisation, scanning, rollback) of the table driven
21 * analysis algorithm by Cooper & Torczon
22 */
23 Token getNextToken(program):
24     // initialisation stage
25     state = start_state
26     lexeme = ""
27     //stack of states
28     Stack<States> stack
29     //add sentinel state to stack
30     stack.(bad_state)
31     //clear white spaces and line feeds
32     while(program.charAt(currentCharIndex)==space,\n , or tab):
33         if(space):
34             lineNumber++
35             //increment char index
36             currentCharIndex++
37             //detect EOF
38             if(currentCharIndex==program.length)
39                 //return EOF
40                 return new Token((TOK_EOF,""))
41     //start scanning
42     while(state != error_state and currentCharIndex<program.length)
43         //obtain current char
44         c = program.charAt(CurrentCharIndex)
45         //append current char to lexeme
46         lexeme.append(c)
47         //if state is accepting clear stack
48         if(state.IsAccepting):
49             stack.clear
50             stack.add(state)
51         //obtain input category of current char (see classifier
table)
52         if(isLetter(c)):
53             inputCat = letter
54         else if(isDigit(c)):
55             inputCat = digit
56         else if(isUnderscore(c)):
57             inputCat = underscore
58         else if(isSlashDivide(c)):
59             inputCat = slashDivide
60         else if(isAsterisk(c)):
61             inputCat = asterisk
62         else if(isLessThan(c)):
63             inputCat = lessThan
64         else if(isGreaterThan(c)):
```

```

65         inputCat = greaterThan
66     else if(isPlus(c)):
67         inputCat = plus
68     else if(isHyphenMinus(c)):
69         inputCat = hyphenMinus
70     else if(isEqual(c)):
71         inputCat = equal
72     else if(isExclamationMark(c))
73         inputCat = exclamationMark
74     else if(isDot(c)):
75         inputCat = dot
76     else if(isSingleQuote(c)):
77         inputCat = singleQuote
78     else if(isPunct(c)):
79         inputCat = punct
80     else if(isOtherPrintable(c))
81         inputCat = otherPrintable
82     else if(isLineFeed(c)):
83         inputCat = LineFeed
84     else:
85         throw exception char not recognised
86     //transition function to get next state
87     state = delta(state,inputCat)
88
89
90     //rollback loop
91     while(state!=error_state and currentCharIndex<tinyLangProgram.
length):
92         //pop state
93         state = stack.pop()
94         //truncate lexeme
95         lexeme.truncate
96         //move char index on stave backward
97         currentCharIndex--
98     //result
99     if(state.getGroup(lexeme)==INVALID)
100         throw exception invalid lexeme
101     else
102         return (state.getGroup(lexeme),lexeme)

```

Listing 1.1: Table Driven Lexer PSEUDOCODE

1.3 | Implementation in Java

- All the possible input categories shown in the classifier table 1.6 are described as a set of predefined constants (see listing 6.3).
- All the states of the tinylang's automaton shown in figure 1.7 are described as a set of predefined constants and in the same enum class

the types associated with each state are described, giving precedence to certain types if the sequence of characters matches some expected lexeme (see listing 6.2).

- The transition function (equivalent to the transition table) is implemented using `HashMap` (see listing 6.4).
- A token is implemented as a class (see listing 6.6) to represent the pair `<TokenType, Attribute>` having the following attributes:
 - **Enum TokenType** (see listing 6.5): The group corresponding to the lexemes.
 - **String Lexeme** : The lexeme itself.
 - **Line Number** : The line number of the lexeme (for error reporting).
 -
- The lexer described by the PSEUDOCODE in listing 1.1 is implemented in Java in its own class. (see listing 6.7)
 - Contains all the required methods such as the the transition function (method name : `deltaFucntion`).

1.4 | Test programs

- Declaring a variable an printing it.

```
1 /*
2  Testing
3  the
4  lexer
5  */
6 let numru : float = (-2)+3.2;
7 //print numru
8 print numru;
```

Listing 1.2: Program 1

```

Choose your option : 1
<TOK_LET, (lexeme:"let", line number:6)>
<TOK_IDENTIFIER, (lexeme:"numru", line number:6)>
<TOK_COLON, (lexeme:":", line number:6)>
<TOK_FLOAT_TYPE, (lexeme:"float", line number:6)>
<TOK_EQUAL, (lexeme:"=", line number:6)>
<TOK_LEFT_ROUND_BRACKET, (lexeme:"(", line number:6)>
<TOK_ADDITIVE_OP, (lexeme:"-", line number:6)>
<TOK_INT_LITERAL, (lexeme:"2", line number:6)>
<TOK_RIGHT_ROUND_BRACKET, (lexeme:");", line number:6)>
<TOK_ADDITIVE_OP, (lexeme:"+", line number:6)>
<TOK_FLOAT_LITERAL, (lexeme:"3.2", line number:6)>
<TOK_SEMICOLON, (lexeme:";", line number:6)>
<TOK_PRINT, (lexeme:"print", line number:8)>
<TOK_IDENTIFIER, (lexeme:"numru", line number:8)>
<TOK_SEMICOLON, (lexeme:";", line number:8)>
<TOK_EOF, (lexeme:"", line number:9)>

```

Figure 1.8: Tokens for Program 1

- A program which prints 1,2,...,10 using a for loop and a while loop

```

1 //a function must always return
2 fn forLoop()->bool{
3   for(let i:int=1;i<=10;i=i+1){
4     print i;
5   }
6   return true;
7 }
8 fn whileLoop()->bool{
9   let i:int=1;
10  while(i<=10){
11    print i;
12    i=i+1;
13  }
14  return false;
15 }
16 /*
17 a statement cannot be a function call (see EBNF)
18 we assign an identifier bool x
19 */
20 let x:bool=forLoop();
21 x=whileLoop();
22 print(x);

```

Listing 1.3: Program 2

```

Choose your option : 1
<TOK_FN, (lexeme:"fn", line number:2)>
<TOK_IDENTIFIER, (lexeme:"forLoop", line number:2)>
<TOK_LEFT_ROUND_BRACKET, (lexeme:"(", line number:2)>
<TOK_RIGHT_ROUND_BRACKET, (lexeme:")", line number:2)>
<TOK_RIGHT_ARROW, (lexeme:"->", line number:2)>
<TOK_BOOL_TYPE, (lexeme:"bool", line number:2)>
<TOK_LEFT_CURLY_BRACKET, (lexeme:"{", line number:2)>
<TOK_FOR, (lexeme:"for", line number:3)>
<TOK_LEFT_ROUND_BRACKET, (lexeme:"(", line number:3)>
<TOK_LET, (lexeme:"let", line number:3)>
<TOK_IDENTIFIER, (lexeme:"i", line number:3)>
<TOK_COLON, (lexeme":", line number:3)>
<TOK_INT_TYPE, (lexeme:"int", line number:3)>
<TOK_EQUAL, (lexeme:"=", line number:3)>
<TOK_INT_LITERAL, (lexeme:"1", line number:3)>
<TOK_SEMICOLON, (lexeme";", line number:3)>
<TOK_IDENTIFIER, (lexeme:"i", line number:3)>
<TOK_RELATIONAL_OP, (lexeme:"<=", line number:3)>
<TOK_INT_LITERAL, (lexeme:"10", line number:3)>
<TOK_SEMICOLON, (lexeme";", line number:3)>
<TOK_IDENTIFIER, (lexeme:"i", line number:3)>
<TOK_EQUAL, (lexeme:"=", line number:3)>
<TOK_IDENTIFIER, (lexeme:"i", line number:3)>
<TOK_ADDITIVE_OP, (lexeme:"+", line number:3)>
<TOK_INT_LITERAL, (lexeme:"1", line number:3)>
<TOK_RIGHT_ROUND_BRACKET, (lexeme:")", line number:3)>
<TOK_LEFT_CURLY_BRACKET, (lexeme:"{", line number:3)>
<TOK_PRINT, (lexeme:"print", line number:4)>
<TOK_IDENTIFIER, (lexeme:"i", line number:4)>
<TOK_SEMICOLON, (lexeme";", line number:4)>
<TOK_RIGHT_CURLY_BRACKET, (lexeme:"}", line number:6)>
<TOK_RETURN, (lexeme:"return", line number:7)>
<TOK_BOOL_LITERAL, (lexeme:"true", line number:7)>
<TOK_SEMICOLON, (lexeme";", line number:7)>
<TOK_RIGHT_CURLY_BRACKET, (lexeme:"}", line number:8)>
<TOK_LET, (lexeme:"let", line number:13)>
<TOK_IDENTIFIER, (lexeme:"x", line number:13)>
<TOK_COLON, (lexeme":", line number:13)>
<TOK_BOOL_TYPE, (lexeme:"bool", line number:13)>
<TOK_EQUAL, (lexeme:"=", line number:13)>
<TOK_IDENTIFIER, (lexeme:"forLoop", line number:13)>
<TOK_LEFT_ROUND_BRACKET, (lexeme:"(", line number:13)>
<TOK_RIGHT_ROUND_BRACKET, (lexeme:")", line number:13)>
<TOK_SEMICOLON, (lexeme";", line number:13)>
<TOK_PRINT, (lexeme:"print", line number:14)>
<TOK_LEFT_ROUND_BRACKET, (lexeme:"(", line number:14)>
<TOK_IDENTIFIER, (lexeme:"x", line number:14)>
<TOK_RIGHT_ROUND_BRACKET, (lexeme:")", line number:14)>
<TOK_SEMICOLON, (lexeme";", line number:14)>
<TOK_EOF, (lexeme:"", line number:15)>

```

Figure 1.9: Tokens for Program 2

Lexer produces expected tokens for Program 1 and Program 2. More programs were tested to ensure that the lexer produces the correct tokens. **Note also that comments are not considered in the output of the token list.**

Task 2 | Hand-Crafted LL(k) Parser

Note: Production rules of tinylang's EBNF as stated in section 0.3 avoids left recursion. This avoids the problem of having recursive descent parser to loop indefinitely.

2.1 | The parser

A tinylang program is parsed using a hand-crafted predictive top-down parser. Features of the parser:

- **Top-down parsing.** Top-down parsing in computer science is a parsing strategy where one first looks at the highest level of the parse tree and works down the abstract syntax tree by using the rewriting rules of a formal grammar until we reach the leaves.
- **Recursive Descent.** The procedures required to move down the abstract syntax tree correspond to one of the non-terminal symbols of the grammar.
- **k=1.** 1 look-ahead token is enough to choose which production rule to use. This allows the parser to be efficient since it is able to make this choice deterministically without need of backtracking.

2.2 | Design of an AST

Each node in a tree is a tree in its own right. We use this recursive definition to define a general tree.

The main difference between an AST and a parse tree is that a parse tree captures the exact derivation while the AST captures the essential properties of the program e.g. for an `if-statement` we keep track of the condition and the `block of statements`, the brackets etc. are redundant. Note that if a parser needs to parse a program fully to produce an AST (ensuring the program is syntactically correct).

To build an AST we have the following requirements:

- Each node has a name to indicate to what type of tree it is. E.g. a node of type `AST_VARIABLE_DECLARATION_NODE` corresponds to a subtree generated by a variable declaration statement (see figure 2.2)
- Node may have a value/lexeme. E.g. a node of type `AST_BINARY_OPERATOR_NODE` may value of '+' to indicate that the operator corresponding to that node (equivalent to an expression tree) is
- Each and every node is associated to a line number to indicate in what part of the program the node/sub tree corresponds to (used for **error handling** in later tasks).

With this logic we can construct a tree class, `Tree`, where:

- **Attributes:**

```

1 //the type associated with each node
2 //e.g. AST_IDENTIFIER_NODE, AST_BINARY_OPERATOR_NODE etc.
3 NodeType nodeType;
4 //line number associated with each node
5 int lineNumber;
6 //value associated with each node (if any)
7 String lexeme;
8 Tree parent;
9 List<Tree> children;
10
```

- **Constructors:**

- If a node has an associated value:
`Tree(NodeType type, String lexeme, int lineNumber)`
- If a node does not have an associated value:
`Tree(NodeType type, int lineNumber)`

- **Methods:**

- Adding a subtree (as a child), PSEUDOCODE:

```

1 void addSubtree(Tree subTree):
2     this.children.add(subTree)
3
```

- Add a new child node:

◇ If child node has an associated value/lexeme, PSEUDOCODE:

```

1      Tree addChild(NodeType nodeType, String lexeme,
      String lineNumber):
2          child = new Tree(nodeType,lexeme,lineNumber)
3          child.parent=this
4          this.children.add(child)
5          return child
6

```

◇ If child node has no associated value/lexeme, PSEUDOCODE:

```

1      Tree addChild(NodeType nodeType, String lineNumber
      ):
2          child = new Tree(nodeType,lineNumber)
3          child.parent=this
4          this.children.add(child)
5          return child
6

```

- Setters and getters.

- Setters and getters where implemented for all attributes.

NodeType (ENUM) have the following values (this are identified in section 2.3).

```

1 TINY_LANG_PROGRAM_NODE,
2 //NODES REPRESENTING STATEMENT TREES
3 AST_VARIABLE_DECLARATION_NODE,
4 AST_ASSIGNMENT_NODE,
5 AST_PRINT_STATEMENT_NODE,
6 AST_IF_STATEMENT_NODE,
7 AST_FOR_STATEMENT_NODE,
8 AST_WHILE_STATEMENT_NODE,
9 AST_RETURN_STATEMENT_NODE,
10 AST_FUNCTION_DECLARATION_NODE,
11 AST_BLOCK_NODE,
12 AST_ELSE_BLOCK_NODE,
13 //EXPRESSION NODES
14 AST_BINARY_OPERATOR_NODE,
15 AST_UNARY_OPERATOR_NODE,
16 AST_FUNCTION_CALL_NODE,
17 AST_IDENTIFIER_NODE,
18 //EXPRESSION NODES -> literal nodes
19 AST_BOOLEAN_LITERAL_NODE,
20 AST_INTEGER_LITERAL_NODE,
21 AST_FLOAT_LITERAL_NODE,
22 AST_CHAR_LITERAL_NODE,
23 //PARAMETER NODES
24 AST_ACTUAL_PARAMETERS_NODE,
25 AST_FORMAL_PARAMETERS_NODE,
26 AST_FORMAL_PARAMETER_NODE,

```

```

27 //TYPE NODE
28 AST_TYPE_NODE,

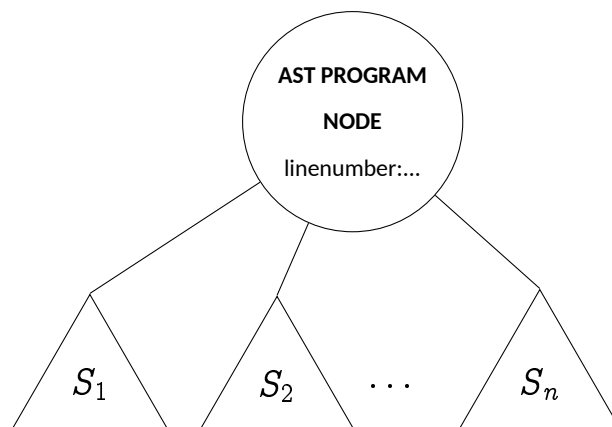
```

Listing 2.1: constants of *ENUM NodeType*

Note: Since the parser is of recursive descent it made sense to define the tree using a recursive approach. We shall now proceed of define the recursive descent parser by defining the whole AST structure by defining the structures of the subtrees.

2.3 | Recursive Descent

2.3.1 | Program Tree

Figure 2.1: A program tree is a sequence of statement subtrees S_1, S_2, \dots, S_n

2.3.1.1 | PSEUDOCODE for building a program tree

We parse the whole program by parsing statements and adding the generated sub-trees per statement as children of the root program node. The implementation of parsing a program is described by the following PSEUDOCODE:

```

1 tree = new Tree(AST_PROGRAM_NODE,getCurrentToken.lineNumber)
2 //go through tokens until we reach EOF
3 while(getCurrentToken.type!=TOK_EOF):
4     tree.addSubtree(parseStatement());
5     //get next token (lookahead for next statement)
6     getNextToken()
7 return tree

```

Listing 2.2: PSEUDOCODE for building a program tree

2.3.2 | Statement Tree(s)

The method `parseStatement()` in listing 2.2 chooses what type of statement to parse based on these lookahead tokens:

- TOK_LET -> parse variable declaration statement
- TOK_IDENTIFIER -> parse assignment statement
- TOK_PRINT -> parse print statement
- TOK_PRINT -> parse print statement
- TOK_IF -> parse if statement
- TOK_FOR -> parse for statement
- TOK_WHILE -> parse while statement
- TOK_RETURN -> parse return statement
- TOK_FN -> parse function declaration
- TOK_LEFT_CURLY_BRACKET -> parse BLOCK

For example if the lookahead token is TOK_LET `parseStatementCall()` calls `parseVariableDeclaration()` (using a switch case) etc.

2.3.2.1 | Variable Declaration Statement

If the current lookahead token is of type TOK_LET, `parseVariableDeclaration()` is called.

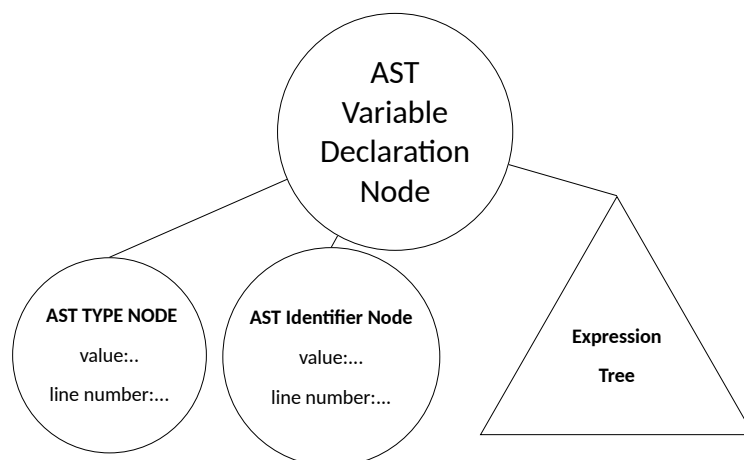


Figure 2.2: Statement tree: **Variable Declaration Statement**

```

1 tree = new Tree(AST_VARIABLE_DECLARATION_NODE,getCurrentToken().
    lineNumber)
2 //token that lead to this method should be let
3 if(getCurrentToken().type != TOK_LET):
4     throw exception unexpected
5 //get next token (this updates the current token)
6 Token identifier = getNextToken()
7 //current token now should be of type identifier
8 if(getCurrentToken().type != TOK_IDENTIFIER):
9     throw exception unexpected
10 //get next token (this updates the current token)
11 getNextToken()
12 //current token now should be a colon
13 if(getCurrentToken().type != TOK_COLON):
14     throw exception unexpected
15 //get next token (this updates the current token)
16 getNextToken()
17 //add type
18 tree.addSubtree(parseType());
19 //add identifier
20 tree.addChild(AST_IDENTIFIER_NODE,identifier.getLexeme(),identifier.
    getLineNumber())
21 //getNextToken()
22 tree.addSubtree(parseExpression())
23 return tree

```

Listing 2.3: PSEUDOCODE for building a variable declaration tree (*parseVariableDeclaration()*)

Note in PSEUDOCODE shown in listing 2.3 there are calls to 2 other methods *parseType()* and *parseExpression()*. The latter is described in section 2.3.3.

parseType() simply generates a 1-node tree of type *AST_TYPE_NODE* where the value differs according to current token type. The PSEUDOCODE is given in the listing below:

```

1 switch(getCurrentToken().getTokenType()):
2     case TOK_BOOL_TYPE:
3         return ast(AST_TYPE_NODE,BOOL,getCurrentToken().
            getLineNumber())
4     case TOK_INT_TYPE:
5         return ast(AST_TYPE_NODE,INT,getCurrentToken().getLineNumber
            ())
6     case TOK_FLOAT_TYPE:
7         return ast(AST_TYPE_NODE,FLOAT,getCurrentToken().
            getLineNumber())
8     case TOK_CHAR_TYPE:
9         return ast(AST_TYPE_NODE,CHAR,getCurrentToken().
            getLineNumber())
10    default:

```

```
11 throw exception unexpected
```

Listing 2.4: PSUEDOCODE for building a 1-node AST_TYPE_NODE tree (*parseType()*)

2.3.2.2 | Assignment Statement

If the current lookahead token is of type TOK_IDENTIFIER, *parseAssingment()* is called.

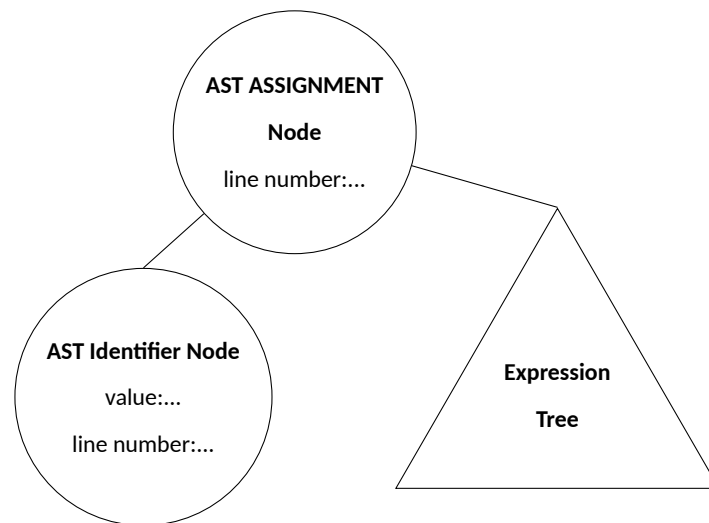


Figure 2.3: Statement tree: **Assignment Statement**

```

1 tree = new Tree(AST_ASSIGNMENT_NODE,getCurrentToken().lineNumber)
2 //token that lead to this method should be of type identifier
3 if(getCurrentToken().type != TOK_IDENTIFIER):
4     throw exception unexpected
5 tree.addChild(AST_IDENTIFIER_NODE,getCurrentToken().getLexeme(),
6     getCurrentToken().getLineNumber())
7 //get next token (this updates current token)
8 getNextToken()
9 //expect equal
10 if(getCurrentToken().type != TOK_EQUAL):
11     throw exception unexpected
12 //get next token
13 getNextToken()
14 //expect expression
15 tree.addSubTree(parseExpression())
16 return tree

```

Listing 2.5: PSEUDOCODE for building an assignment tree (*parseAssignment()*)

2.3.2.3 | Print Statement

If the current lookahead token is of type TOK_PRINT, `parsePrintStatement()` is called.

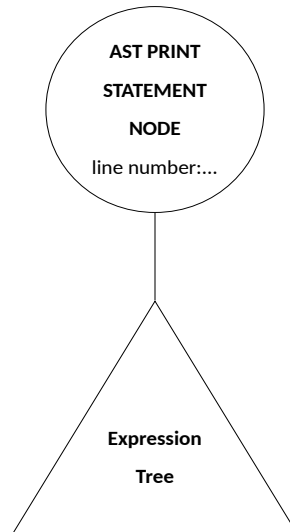


Figure 2.4: Statement tree: **PRINT STATEMENT**

```

1 tree = new Tree(AST_PRINT_STATEMENT_NODE,getCurrentToken().
    lineNumber)
2 //token that lead to this method should be of type TOK_PRINT
3 if(getCurrentToken().type != TOK_PRINT):
4     throw exception unexpected
5 //get next token (this updates current token)
6 getNextToken()
7 //expect expression
8 tree.addSubTree(parseExpression())
9 return tree
  
```

Listing 2.6: PSEUDOCODE for building a print statement tree (*printStatement()*)

2.3.2.4 | If Statement

If the current lookahead token is of type TOK_IF, `parseIfStatement()` is called.

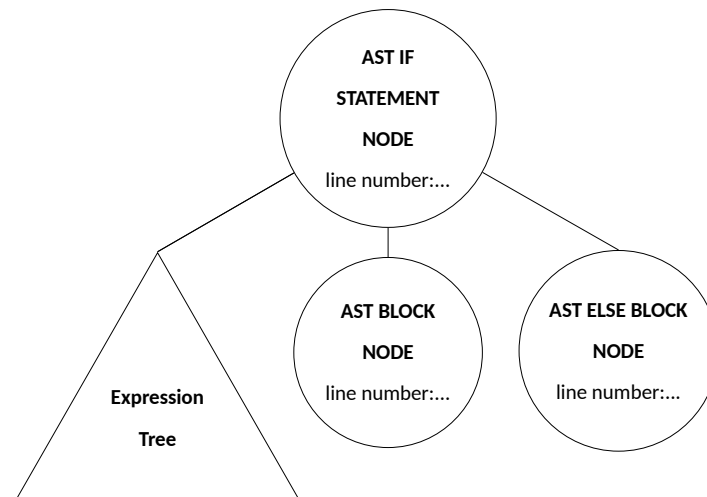


Figure 2.5: Statement tree: IF STATEMENT

Note as per EBNF rules (see section 0.3) an else block node is optional this is highlighted in the PSEUDOCODE given below.

```

1 tree = new Tree(AST_IF_STATEMENT_NODE,getCurrentToken().lineNumber)
2 //token that lead to this method should be of type TOK_IF
3 if(getCurrentToken().type != TOK_IF):
4     throw exception unexpected
5 //get next token (this updates current token)
6 getNextToken()
7 //expect (
8 if(getCurrentToken().type != TOK_LEFT_ROUND_BRACKET):
9     throw exception unexpected
10 //get next token (this updates current token)
11 getNextToken()
12 //add expression subtree
13 tree.addExpression(parseExpression());
14 //get next token( this updates current token)
15 getNextToken();
16 //expect )
17 if(getCurrentToken().type != TOK_RIGHT_ROUND_BRACKET):
18     throw exception unexpected
19 //get next token( this updates current token)
20 getNextToken();
21 //parse block
22 tree.addSubtree(parseBlock())
23
24
25 //we check for an else condition (OPTIONAL)
26 //get next token( this updates current token)
27 getNextToken();
28
29 //if current token is else i.e. we have an else block node
30 if(getCurrentToken().type != TOK_ELSE):
31     //get next token( this updates current token)
32     getNextToken()

```

```

33 //add else block
34 tree.addSubtree(parseElseBlock())
35 //else no else block
36 else
37 //get previous token( this updates current token)
38 getPrevToken();
39 return tree

```

Listing 2.7: PSEUDOCODE for building an if statement tree (*ifStatement()*)

Note that the listing above calls parsing methods: `parseExpression()`, `parseBlock()` and `parseElseBlock()`. The implementation of `parseExpression()` and `parseBlock()` is discussed in section 2.3.3 and listing 2.13 respectively. The implementation of `parseElseBlock()` is equivalent to the implementation of `parseBlock()`.

2.3.2.5 | For Statement

If the current lookahead token is of type `TOK_FOR`, `parseForStatement()` is called. If the current lookahead token is of type `TOK_LEFT_CURLY_BRACKET`, `parseBlock()` is called. A block node is equivalent to a program node with the difference that the sequence of statements are enclosed in curly brackets.

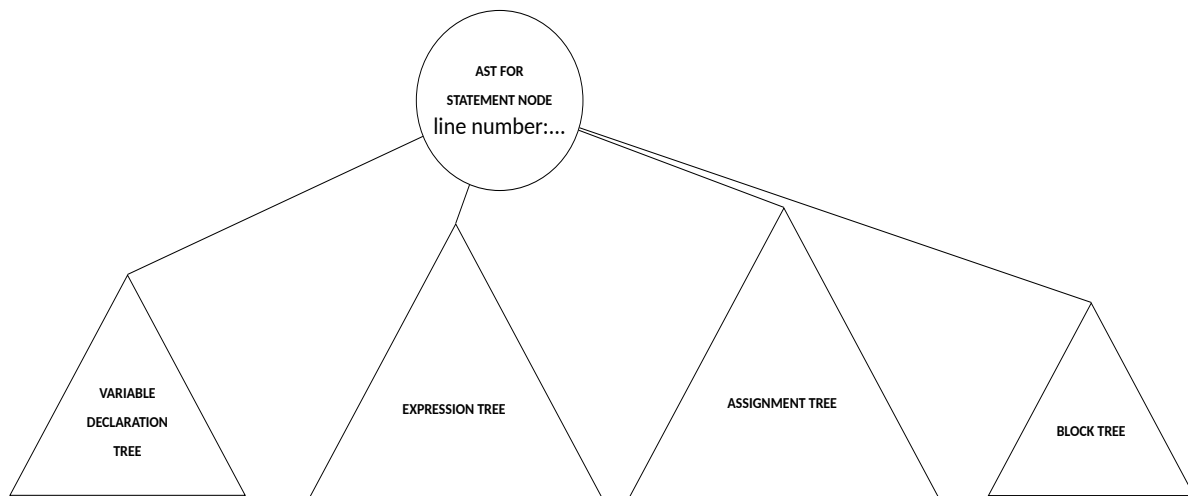


Figure 2.6: Statement tree: **FOR STATEMENT**

Note that the implementation of building variable declaration tree, expression tree, assignment tree and block tree are discussed in sections 2.3.2.1, 2.3.3, 2.3.2.2 and 2.3.2.9 respectively. **Also note** that as per EBNF rule (see section 0.3), Variable Declaration Tree and Assignment Tree are optional, this is highlighted in the PSEUDOCODE of the implementation below.

```

1 tree = new Tree(AST_FOR_STATEMENT_NODE, getCurrentToken().lineNumber)
2 //token that lead to this method should be of type TOK_FOR

```

```

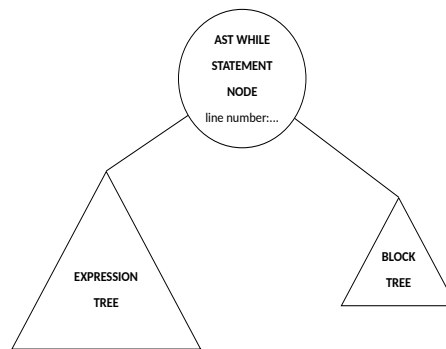
3 if(getCurrentToken().type != TOK_FOR):
4     throw exception unexpected
5 //get next token (this updates current token)
6 getNextToken()
7 //expect (
8 if(getCurrentToken().type != TOK_LEFT_ROUND_BRACKET):
9     throw exception unexpected
10 //get next token (this updates current token)
11 getNextToken()
12 //expect ; or a variable declaration (optional)
13 if(getCurrentToken().type != TOK_SEMICOLON):
14     tree.addSubTree(parseVariableDeclartion())
15     //get next token (this updates current token)
16     getNextToken()
17 //expect ;
18 if(getCurrentToken().type != TOK_SEMICOLON):
19     throw exception unexpected
20 //get next token (this updates current token)
21 getNextToken()
22 //expect expression
23 tree.addSubtree(parseExpression())
24 //get next token (this updates current token)
25 getNextToken()
26 //expect ;
27 if(getCurrentToken().type != TOK_SEMICOLON):
28     throw exception unexpected
29
30 //expect ) or assignment (optional)
31 if(getCurrentToken().type != TOK_RIGHT_ROUND_BRACKET):
32     tree.addSubtree(parseAssigment())
33     getNextToken()
34
35 //expect block
36 tree.addSubtree(parseBlock())
37 //return tree
38 return tree

```

Listing 2.8: PSEUDOCODE for building a for statement tree (`parseForStatement()`)

2.3.2.6 | While Statement

If the current lookahead token is of type `TOK_WHILE`, `parseWhileStatement()` is called.

Figure 2.7: Statement tree: **WHILE LOOP**

Note that the implementation of building expression tree and block tree are discussed in sections 2.3.3 and 2.3.2.9 respectively.

```

1 tree = new Tree(AST_FOR_STATEMENT_NODE,getCurrentToken().lineNumber)
2 //token that lead to this method should be of type TOK_WHILE
3 if(getCurrentToken().type != TOK_WHILE):
4     throw exception unexpected
5 //get next token (this updates current token)
6 getNextToken()
7 //expect (
8 if(getCurrentToken().type != TOK_LEFT_ROUND_BRACKET):
9     throw exception unexpected
10 //get next token (this updates current token)
11 getNextToken()
12 //expect expression
13 tree.addSubtree(parseExpression())
14 //get next token (this updates current token)
15 getNextToken()
16 //expect )
17 if(getCurrentToken().type != TOK_RIGHT_ROUND_BRACKET):
18     throw exception unexpected
19 //get next token (this updates current token)
20 getNextToken()
21 //expect block
22 tree.addSubtree(parseBlock())
23 return tree

```

Listing 2.9: PSEUDOCODE for building a while statement subtree (*parseWhileStatement()*)

2.3.2.7 | Return Statement

The implementation of parsing a return statement and generating a return statement subtree is analogous to parsing a print statement and generating a print statement subtree as described in section 2.3.2.3 with the difference that the parent node is of type `TOK_RETURN_STATEMENT_NODE` instead of `TOK_PRINT_STATEMENT_NODE`.

2.3.2.8 | Function Declaration Statement

If the current lookahead token is of type TOK_FN, `parseFunctionDeclaration()` is called.

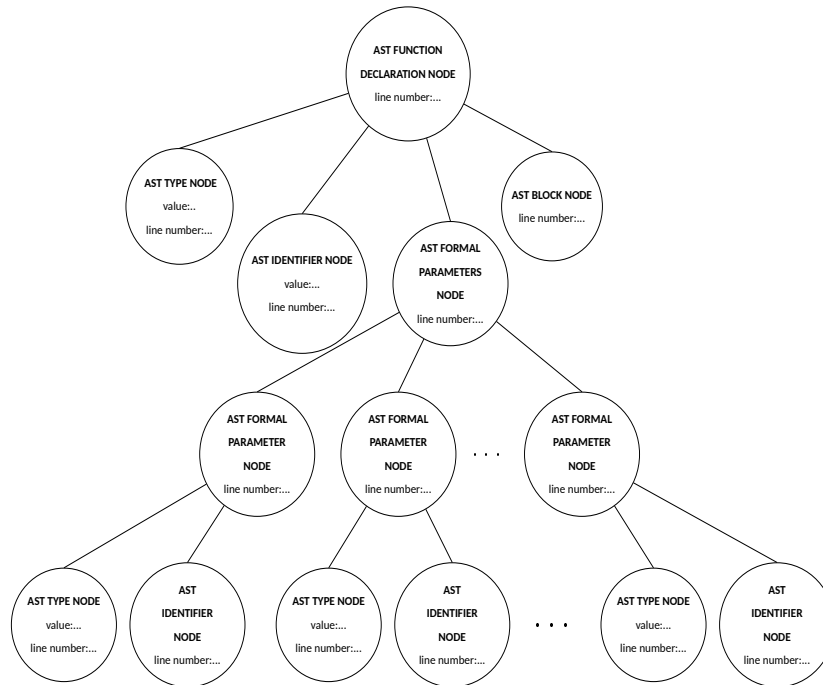


Figure 2.8: Statement tree: **FUNCTION DECLARATION**

```

1 tree = new Tree(AST_FUNCTION_DECLARATION_NODE,getCurrentToken().
    lineNumber)
2 //token that lead to this method should be of type TOK_FN
3 if(getCurrentToken().type != TOK_FN):
4     throw exception unexpected
5 //get next token (this updates the current token)
6 Token identifier = getNextToken()
7 //expect identifier
8 Token identifier
9 if getCurrentToken().type ==TOK_IDENTIFIER
10     identifier=getCurrentToken()
11 else
12     throw exception unexpected
13 //get next token (this updates the current token)
14 getNextToken()
15 //expect (
16 if(getCurrentToken().type != TOK_LEFT_ROUND_BRACKET):
17     throw exception unexpected
18 //get next token (this updates the current token)
19 getNextToken()
20 //expect 0 or more formal parameters
21 Tree formalParamsSubtree
22 //if next token is not a right round bracket we have formal
    paramaters

```

```

23 if(getCurrentToken().type != TOK_RIGHT_ROUND_BRACKET):
24     formalParamsSubtree = parseFormalParams()
25     //get next token (this updates the current token)
26     getNextToken()
27 //else just add a formal parameters node with no children
28 else
29     formalParamsSubtree = new Tree(AST_FORMAL_PARAMETERS_NODE,
30         getCurrentToken().lineNumber)
31 //expect )
32 if(getCurrentToken().type != TOK_RIGHT_ROUND_BRACKET):
33     throw exception unexpected
34 //get next token (this updates the current token)
35 getNextToken()
36 //expect ->
37 if(getCurrentToken().type != TOK_RIGHT_ARROW):
38     throw exception unexpected
39 //get next token (this updates the current token)
40 getNextToken()
41 //expect type
42 Tree typeSubtree = parseType()
43 //get next token (this updates the current token)
44 getNextToken()
45 //expect block
46 Tree blockSubtree = parseSubtree()
47 //add type subtree to function declaration tree
48 tree.addSubtree(typeSubtree)
49 //add identifier node to function declaration tree
50 tree.addChild(AST_IDENTIFIER_NODE, identifier.lexeme, identifier.
51     lineNumber)
52 //add formal params subtree to function declaration tree
53 tree.addSubtree(formalParamsSubtree)
54 //add block subtree to function declaration tree
55 tree.addSubtree(blockSubtree)
56 //return function declaration tree
57 return tree

```

Listing 2.10: PSEUDOCODE for building a function declaration statement tree (*parseFunctionDeclaration()*)

Note in PSEUDOCODE shown in listing 2.10 there are calls to 3 other parsing methods: *parseFormalParams()*, *parseType()*, and *parseBlock()*.

parseType() and *parseBlock()* are described in PSEUDOCODE in listings 2.4 and 2.13 respectively.

A diagram of a formal parameter subtree is shown in figure 2.9.

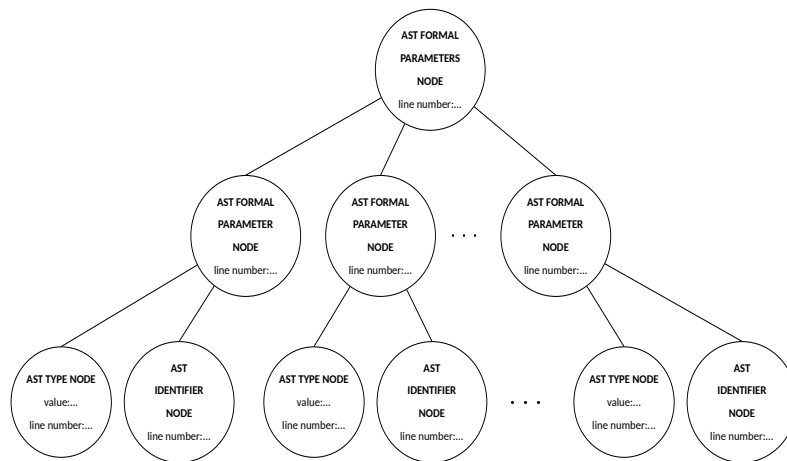


Figure 2.9: Formal parameters subtree

The following logic is formulated using EBNF rules. (see section 0.3).

Formal parameters is a sequence of formal parameter separated by a comma. Each formal parameter has 2 important attributes the identifier and type. The PSEUDOCODE for constructing a formal parameter subtree is given below.

```

1 tree = new Tree(AST_FORMAL_PARAMS_NODE,getCurrentToken().lineNumber)
2 //add formal param
3 tree.addSubtree(parseFormalParam())
4 //get next token (this updates the current token)
5 getNextToken()
6 //each formal parameter is seperated by a comma
7 while(getCurrentToken().tokenType==TOK_COMMA)
8     //get next token (this updates the current token)
9     getNextToken()
10    //parse next formal parameter
11    tree.addSubtree(parseFormalParam())
12    //get next token (this updates the current token)
13    getNextToken()
14 //get prev token (this updates the current token)
15 getPrevToken();
16 return tree

```

Listing 2.11: PSEUDOCODE for building a formal parameters subtree

Note that in listing 2.11 a call to `parseFormalParam()` is made. Since a formal parameter is made up of 2 important attributes the identifier and type. We parse a formal parameter by parsing through the identifier and type, PSEUDOCODE is given below.

```

1 tree = new Tree(AST_FORMAL_PARAMETER_NODE,getCurrentToken().
    lineNumber)
2 //expect identifier
3 if getCurrentToken().type == TOK_IDENTIFIER
4     identifier=getCurrentToken()
5 //get a hold of identifier node

```

```

6 Token identifier = getCurrentToken();
7 //get next token (this updates current token)
8 getNextToken();
9 //expect :
10 if getCurrentToken().type ==TOK_COLON
11     identifier=getCurrentToken()
12 //get next token (this updates current token)
13 getNextToken();
14 //add type subtree
15 tree.addSubtree(parseType())
16 //add identifier
17 tree.addChild(AST_IDENTIFIER,identifier.lexeme,identifier.lineNumber
18 )
19 //return tree
20 return tree

```

Listing 2.12: PSEUDOCODE for building a formal parameter subtree (*parseFormalParam()*)

Note that in listing above, a call to *parseType()* is made. Parsing of a type is discussed in listing 2.4.

2.3.2.9 | Block Statement

If the current lookahead token is of type *TOK_LEFT_CURLY_BRACKET*, *parseBlock()* is called. A block node is equivalent to a program node with the difference that the sequence of statements are enclosed in curly brackets.

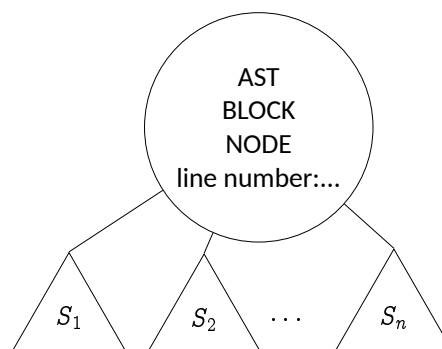


Figure 2.10: Statement tree: Block is a sequence of statements S_1, S_2, \dots, S_n

```

1 tree = new Tree(AST_BLOCK_NODE,getCurrentToken().lineNumber)
2 //token that lead to this method should be of type {
3 if(getCurrentToken().type != TOK_LEFT_CURLY_BRACKET):
4     throw exception unexpected
5 //get next token (this updates current token)
6 getNextToken()
7 //we may have one or more statements block ends using }

```



```

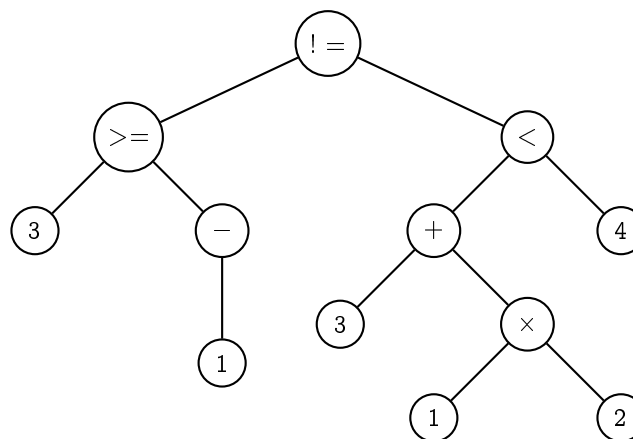
8 while(getCurrentToken().getTokenType() != TokenType.
    TOK_RIGHT_CURLY_BRACKET and getCurrentToken().getTokenType() !=
    TokenType.TOK_EOF ):
9     tree.addSubtree(parseStatement())
10    getNextToken()
11
12 //current token should be }
13 if(getCurrentToken().type != TOK_LEFT_RIGHT_BRACKET):
14     throw exception unexpected
15 return tree

```

Listing 2.13: PSEUDOCODE for building a block tree (*parseBlock()*)

2.3.3 | Expression Tree(s)

An expression tree is a tree where the intermediate nodes correspond to a binary operator and the leaf nodes are values to the corresponding binary operator.

Figure 2.11: Example of an **expression tree**

An inorder traversal of the expression tree shown in figure 2.11 gives us the expression $((3) \geq (-1))! = (((3) + ((1) \times (2))) < (4))$ which evaluates to true.

As per EBNF rules (see section 0.3) we parse an expression using the following non terminals: `<Expression>`, `<SimpleExpression>`, `<Term>` and `<Factor>`.

2.3.3.1 | <Expression>

Expression is a sequence of one or more simple expression separated by a relational operator (see section 0.3). For example suppose we have

$$se_1 \text{ relop}_1 se_2 \text{ relop}_2 \dots \text{ relop}_{n-1} se_n \equiv$$

$se_1 \text{ relop}_1 (se_2 \text{ relop}_2 \dots (se_{n-1} \text{ relop}_{n-1} se_n))$ (se denotes simple expression) then a tree representing this expression would look like the following:

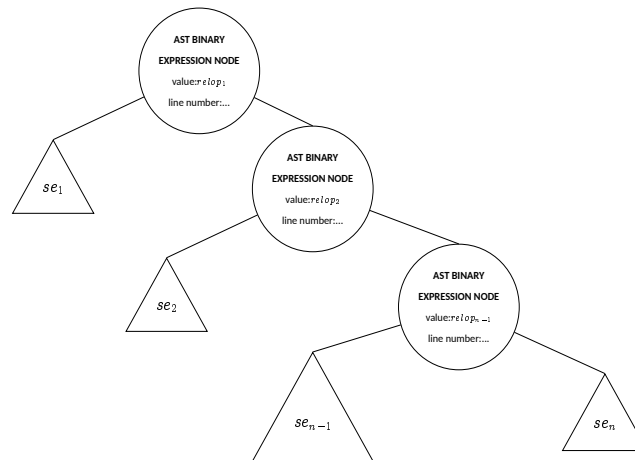
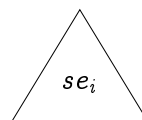


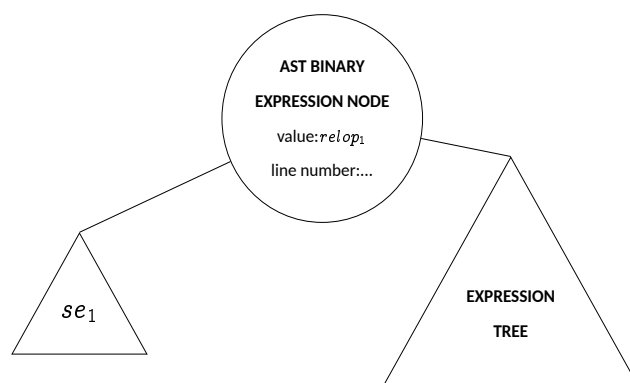
Figure 2.12: Expression is a sequence of simple expressions separated by an operator of type `TOK_RELATIONAL_OP`

The tree shown in 2.12 can be defined recursively (note that the right operand has a similar structure).

- Base Case: The expression tree is just 1 simple expression.



- Recursive Case:



The method of parsing $\langle \textit{Expression} \rangle$ and building an expression tree recursively is described in the following PSEUDOCODE.

```

1 // base case
2 Tree leftOperand = parseSimpleExpression()
3 //get next token (this updates current token)
4 getNextToken()
5 //if we have a relop run recursive case
6 if(getCurrentToken().type != TOK_RELATIONAL_OP):
7     // build a binary tree value -> lexeme (representing the
       operator)
8     tree = new Tree(AST_BINARY_OPERATOR_NODE,getCurrentToken().
       lexeme,getCurrentToken().lineNumber)
9     //add left operand of binary operator
10    tree.addSubtree(leftOperand)
11    //recursive step
12    tree.addSubtree(parseExpression())
13    return tree
14 return leftOperand

```

Listing 2.14: PSEUDOCODE : parsing $\langle \text{Expression} \rangle$ and building an expression tree (*parseExpression()*)

Note a recursive call is made in line

`tree.addSubtree(parseExpression())`. A call to parse a simple expression tree is also made via `parseSimpleExpression()`. Parsing of a simple expression is discussed in 2.3.3.2

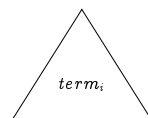
2.3.3.2 | $\langle \text{SimpleExpression} \rangle$

Simple Expression is a sequence of one or more simple expression separated by a additive operator (see section 0.3). For example suppose we have

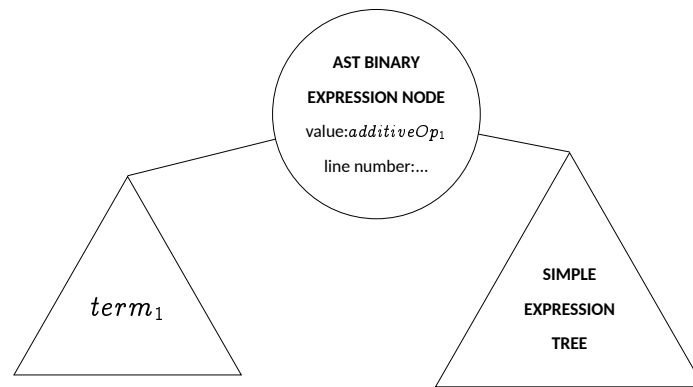
$$term_1 \text{ additiveOp}_1 term_2 \text{ additiveOp}_2 \dots \text{ additiveOp}_{n-1} term_n \equiv term_1 \text{ additiveOp}_1 (term_2 \text{ additiveOp}_2 \dots (term_{n-1} \text{ additiveOp}_{n-1} term_n))$$

A simple expression can be built recursively similar to as discussed in section 2.3.3 where

- Base Case: The simple expression tree is just 1 simple expression.



- Recursive Case:



The method of parsing $\langle SimpleExpression \rangle$ and building an expression tree recursively is described in the following pseudocode.

```

1 // base case
2 Tree leftOperand = parseTerm()
3 //get next token (this updates current token)
4 getNextToken()
5 //if we have a relop run recursive case
6 if(getCurrentToken().type != TOK_ADDITIVE_OP):
7     // build a binary tree value -> lexeme (representing the
    operator)
8     tree = new Tree(AST_BINARY_OPERATOR_NODE, getCurrentToken().
    lexeme, getCurrentToken().lineNumber)
9     //add left operand of binary operator
10    tree.addSubtree(leftOperand)
11    //recursive step
12    tree.addSubtree(parseSimpleExpression())
13    return tree
14 return leftOperand

```

Listing 2.15: parsing $\langle SimpleExpression \rangle$ and building an expression tree (`parseSimpleExpression()`)

Note a recursive call is made in line `tree.addSubtree(parseExpression())`. A call to parse a term is also made via `parseTerm()`. Parsing of a term is discussed in 2.3.3.3.

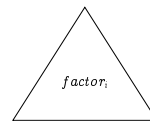
2.3.3.3 | $\langle Term \rangle$

Term is a sequence of one or more factors separated by a multiplicative operator (see section 0.3). For example suppose we have

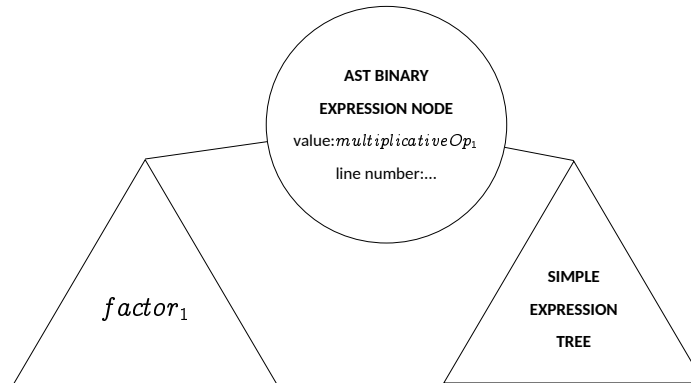
$factor_1 \text{ multiplicativeOp}_1 \text{ term}_2 \text{ multiplicativeOp}_2 \dots \text{ multiplicativeOp}_{n-1} \text{ term}_n$
 $factor_1 \text{ additiveOp}_1 (factor_2 \text{ multiplicativeOp}_2 \dots (factor_{n-1} \text{ multiplicativeOp}_{n-1} \text{ term}_n$

A term can be built recursively similar to as discussed in sections 2.3.3 and 2.3.3.2 where

- Base Case: The simple expression tree is just 1 simple expression.



- Recursive Case:



The method of parsing $\langle Term \rangle$ and building an expression tree recursively is described in the following pseudocode.

```

1 // base case
2 Tree leftOperand = parseFactor()
3 //get next token (this updates current token)
4 getNextToken()
5 //if we have a relop run recursive case
6 if(getCurrentToken().type != TOK_MULTIPLICATIVE_OP):
7     // build a binary tree value -> lexeme (representing the
    operator)
8     tree = new Tree(AST_BINARY_OPERATOR_NODE,getCurrentToken().
    lexeme,getCurrentToken().lineNumber)
9     //add left operand of binary operator
10    tree.addSubtree(leftOperand)
11    //recursive step
12    tree.addSubtree(parseFactor())
13    return tree
14 return leftOperand

```

Listing 2.16: parsing $\langle Term \rangle$ and building an expression tree (*parseTerm()*)

Note a recursive call is made in line `tree.addSubtree(parseExpression())`. A call to parse a factor is also made via `parseFactor()`. Parsing of a factor is discussed in 2.3.3.4

2.3.3.4 | **<Factor>**

A factor represents an operand of the binary operator. Now an operand may take different forms (see section 0.3) namely:

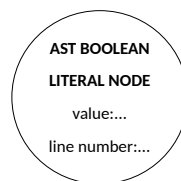
- **Literal** : A constant value e.g. 1, *true*, 5.3, '*a*' etc.
- **Identifier** : Representing a variable. The operand operates on the value of that variable.
- **FunctionCall** : A call to a function that is expected to return some value. That value is used as the operand.
- **SubExpression** : The operand might be a value return by another expression.
- **Unary** : A unary operator followed by an expression e.g. +5, -(2+3.2), not 5>3 etc.

We use a 1 look ahead token to deterministically decide what the type of the operand is and parse accordingly.

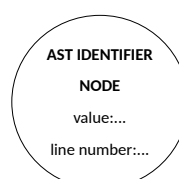
The logic of `parseFactor()` is given by the following list of cases.

If the current lookahead token is of type:

- `TOK_BOOLEAN_LITERAL` return the following node



- Return similar nodes on token types `TOK_INT_LITERAL`, `TOK_FLOAT_LITERAL` and `TOK_CHAR_LITERAL`
- `TOK_IDENTIFIER`. This leads to possible cases. The operand is either an identifier or a function call. We keep the implementation deterministic ($k=1$) by checking if the next token is a left round bracket.
 - If next token is a left round bracket we deduce that the operand is a function call and we return the subtree produced by parsing a function call (`parseFunctionCall()`) (see section 2.3.3.4.3 for discussion of function call tree)
 - Otherwise we deduce that the operand is just an identifier and return the following node:



- TOK_LEFT_ROUND_BRACKET then return the tree produced by parsing a sub expression (`parseSubExpression()`) (see section 2.3.3.4.2 for discussion of sub expression tree)
- TOK_ADDITIVE_OP or TOK_NOT then return the tree produced by parsing an unary expression (`parseUnary()`) (see section 2.3.3.4.1 for discussion of unary tree)
- for other tokens throw exception unexpected

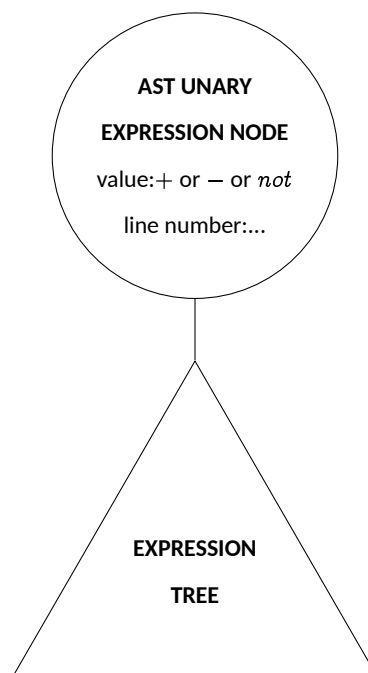


Figure 2.13: Unary expression tree

2.3.3.4.1 Factor : Unary Expression Building of an expression tree is discussed in section 2.3.3.

While parsing a unary expression we check for unary operators `+`, `-` and `not` and construct the unary node accordingly. The PSEUDOCODE of given below.

```

1 //an additive op or not led to this parsing method
2 if(getCurrentToken ().type != TOK_ADDITIVE_OP and getCurrentToken().
   type != TOK_NOT):
3     throw exception unexpected
4 tree = new Tree( AST_BLOCK_NODE ,getCurrentToken().lexeme,
   getCurrentToken().lineNumber )
5 //get next token (this updates current token )
6 getNextToken ()
7 //add expression subtree
8 tree.addSubtree(parseExpression())
  
```

```
9 return tree
```

Listing 2.17: PSEUDOCODE for building a **unary expression tree** (*parseUnary()*)

2.3.3.4.2 Factor :Sub Expression A sub expression is an expression in its own right. We get hold on the value returned by a sub-expression to use it in other expression by enclosing in its bracket.

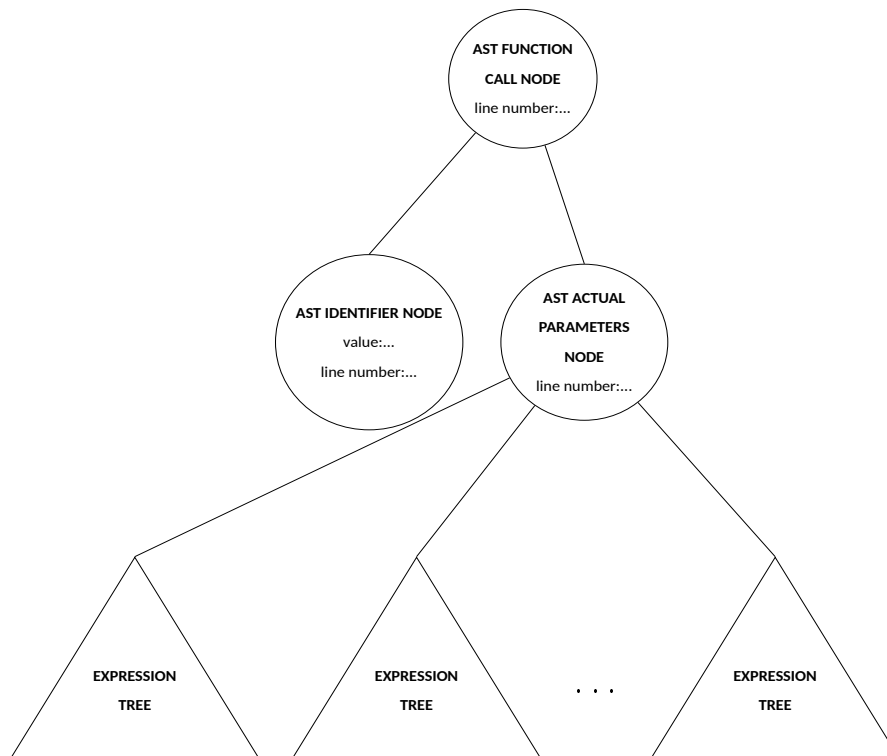
Since a sub expression is an expression the tree return by sub expression is an expression tree as described in section 2.3.3.

When parsing a sub expression we check for a left round bracket we parse an expression and then we check for a right bracket. The pseudocode is given below.

```
1 //an left round bracket led to this parsing method
2 if(getCurrentToken ().type != TOK_LEFT_ROUND_BRACKET):
3     throw exception unexpected
4 //get next token(this updates current token)
5 tree = parseExpression()
6 //get next token
7 getNextToken();
8 //we expect a right round bracket
9 if(getCurrentToken().type=TOK_RIGHT_ROUND_BRACKET)
10     throw exception unexpected
11 return tree
```

Listing 2.18: PSEUDOCODE for (*parseSubExpression()*)

2.3.3.4.3 Factor : Function Call A function call tree is similar to function declaration tree as described in section 2.3.2.8. We call a function without specifying its type and block of statements (reference to actual declaration) hence a function call tree need not have a type child and block child. But instead of formal parameters subtree we have an actual parameters subtree whose children are expression tree.

Figure 2.14: Factor tree: **FUNCTION CALL**

When parsing a function call factor we check if we have an identifier (the lookahead token that lead to parsing a function call factor) , for brackets enclosing the actual parameters. If we have no parameters the actual parameter node has no children.

The PSEUDOCODE is given below.

```

1 tree = new Tree( AST_FUNCTION_CALL_NODE , getCurrentToken().
    lineNumber )
2 // token that lead to this method should be of type identifier
3 if( getCurrentToken ().type != TOK_IDENTIFIER ):
4     throw exception unexpected
5 //add identifier node
6 tree.addChild(AST_IDENTIFIER_NODE,getCurrentToken().lexeme,
    getCurrentToken().lineNumber)
7 // get next token (this updates current token)
8 getNextToken ()
9 // next token should be of type (
10 if(getCurrentToken ().type != TOK_LEFT_ROUND_BRACKET ):
11     throw exception unexpected
12 // get next token (this updates current token)
13 getNextToken ()
14 //if the next token is not a round bracket -> we should have one or
    more actual parameters
15 if(getCurrentToken ().type != TOK_RIGHT_ROUND_BRACKET ):
16     tree.addSubtree(parseActualParams())
17     // get next token (this updates current token)
18     getNextToken()

```

```

19 //else we add a parameter node with no children
20 else
21     tree.addChild(AST_ACTUAL_PARAMETER_NODE,getCurrentToken().lexeme.
        getCurrentToken().linenumber)
22 // get next token (this updates current token)
23 getNextToken ()
24 //expect right round bracket
25 if(getCurrentToken ().type != TOK_RIGHT_ROUND_BRACKET):
26     throw exception unexpected
27 //return tree
28 return tree

```

Listing 2.19: PSEUDOCODE for building a function call expression tree

Note that in the listing above a call to `parseActualParameters()` is made when we have **1 or more actual parameters**. An actual parameters tree consists of an actual parameter node with expression subtrees as shown in figure 2.14. To parse actual parameters we need to parse 1 or more expression (see section 2.3.3). The PSEUDOCODE of parsing actual parameters is given below /

```

1 Tree tree = new TinyLangAst(AST_ACTUAL_PARAMETERS,getCurrentToken().
    lineNumber)
2 //add expression subtree
3 tree addSubtree(parseExpression)
4 //get next token (this updates current token)
5 getNextToken()
6 //we start checking if we have commas since this implies that we
    have more actual paremeters
7 while(getCurrentToken().type==TOK_COMMA and getCurrentToken().type!=
    TOK_EOF):
8     //get next token (this updates current token)
9     getNextToken()
10    //add next expression subtree
11    actualParamsTree.addSubtree(parseExpression())
12    //get next token (this updates current token)
13    getNextToken()
14 //move back one token (this updates current token)
15 getPrevToken()
16 return tree

```

Listing 2.20: parsing 1 or more actual parameters (*parseActualParams()*)

2.4 | Parse tree of a sample tinylang program

Consider the following tinylang program.

```

1 fn Sq (x:float) -> float {
2     return x*x ;
3 }

```

```
4 print Sq(5+2);
```

Listing 2.21: a tinylang program

Using the recursive descent parse described using the the methods above starting from `parseTinyLangProgram()` we generate the following AST

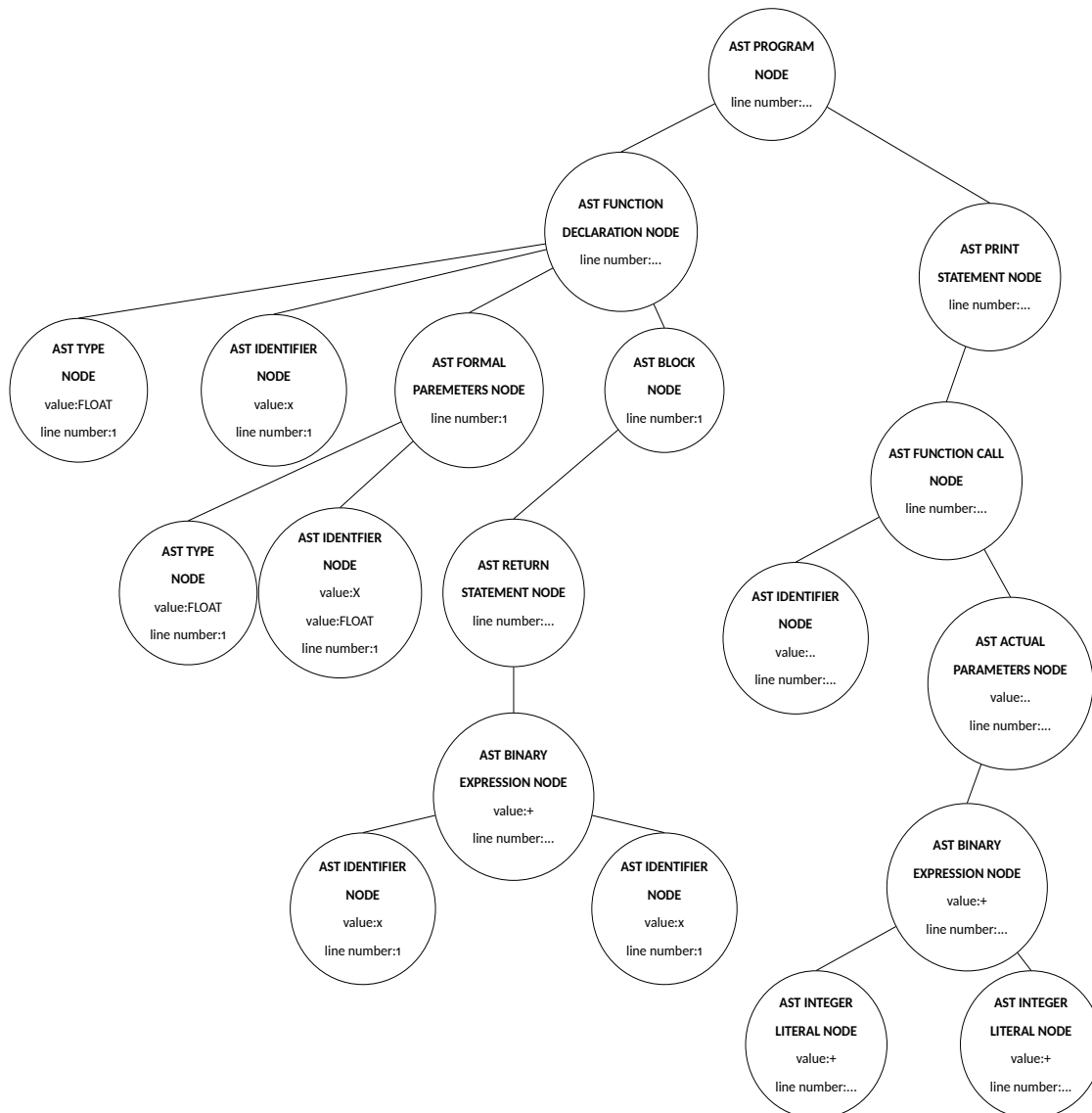


Figure 2.15: AST generated after parsing program (listing 2.21)

2.5 | Implementation in Java

- The implementation of a general AST (see section 2.2), and the enum constants (see listing 2.1) used to indicate the the type of subtree is given in listings 6.8 and 6.9 respectively.

- The implementation of parsing methods discussed in section 2.3 is given in listing 6.10.

2.6 | Testing

We test a program that has a number of syntax errors and fix it.

```

1 //a function must always return
2 let x bool=false;
3 fn forLoop()->bool{
4   for(let i:int=1;i<=10;i=i+1){
5     print i;
6   }
7   return true;
8 }
9 /*
10 a statement cannot be a function call (see EBNF)
11 we assign an identifier bool x
12 */
13 bool=forLoop();
14 if(x==(true)) {print 'T';} else {print 'F';}
```

Listing 2.22: Program 3

- When executing we get an exception message that we have a missing colon in line 2.

```
Exception in thread "main" java.lang.RuntimeException: expect colon in line 2
    at tinylangparser.TinyLangParser.parseVariableDeclaration(TinyLangParser.java:14)
```

Figure 2.16: Exception 1

- After we add the colon (i.e. `let x bool=false; -> let x : bool=false;`). We execute once again and we get that we have an error indicating that we expect a semicolon at line 6.

```
Program in considration: program3.tl
Exception in thread "main" java.lang.RuntimeException: expected semicolon;; , in
line 6
```

Figure 2.17: Exception 2

- After we add the semicolon (i.e. `print i -> print i;`). We execute once again and we get that we have an error in line 13 indicating that no statement can begin with `bool`.

```
Exception in thread "main" java.lang.RuntimeException: in line 13. No statement
begins with bool
    at tinylangparser.TinyLangParser.parseStatement(TinyLangParser.java:12)
```

Figure 2.18: Exception 3

- After fixing the error (i.e. `bool=-forLoop(); -> x=forLoop();`). We execute once again and we get the following error.

```
Exception in thread "main" java.lang.RuntimeException: expected right round bracket,  
) , in line 14
```

Figure 2.19: Exception 4

- After fixing the error i.e.
(`if(x=true){...} else {...} ->`
`if(x=true)){...} else {...}`). **We get no errors**

```
Note: program is semantically correct
```

Figure 2.20: Success

Note figure 2.20 also implies that the program is semantically correct, this notion is discussed in Task 4.

Task 3 | **AST XML Generation Pass**

3.1 | **ENUM-Based Visitor's Design Pattern**

In Tasks 3,4,5 we need to traverse each and every node and perform some specific operation. Each node has a `type` which is an enum value. We use an enum-based visitors pattern to identify the type of a node (subtree) and carry out the required operations.

The design requires us to have an interface `Visitor` holding a visitor method for each type. `Visitor` is implemented by the concrete classes to ensure that all the nodes in an AST are visited and acted upon accordingly.

A diagram showcasing the design pattern is given below.

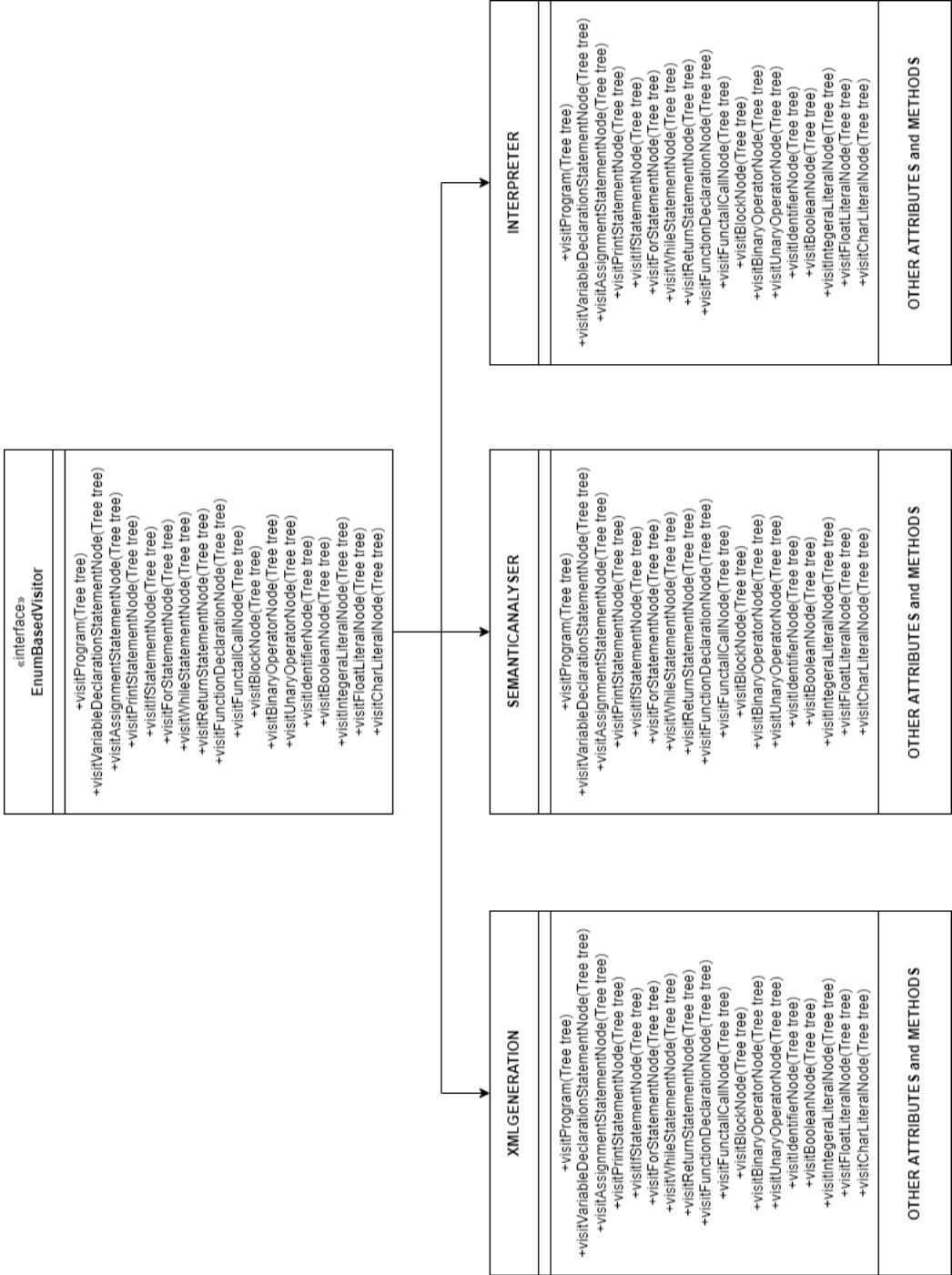


Figure 3.1: *XmlGeneration, Semantic Analyser and Interpreter* are concrete implementations of interface EnumVisitor.

3.2 | Design

We want to generate a string representation of an abstract tree. We use XML representation. Now each node corresponds to a different subtree and each tree corresponds to a different XML tag. Hence we use the visitor's design pattern and use the required visitor method to generate the appropriate XML tags and content.

Consider the AST in figure 2.15 an XML representation is given below.

```

1 <TinyLangProgram>
2   <function declaration>
3     <id type="FLOAT"> Sq <\id>
4     <parameters>
5       <id type="FLOAT"> x <\id type>
6     <\parameters>
7     <block>
8       <return statement>
9         <binary Op>
10          <id type="FLOAT"> x <\id type>
11          <id type="FLOAT"> x <\id type>
12        <\binay Op>
13      <\return statement>
14    <\block>
15  <\function declaration>
16  <print>
17    <function call>
18      <id>Sq<\id>
19      <parameters>
20        <actual parameter>
21          <id>x<\id>
22        <\actual parameter>
23        <actual parameter>
24          <id>x<\id>
25        <\actual parameter>
26      <\parameters>
27    <\function call>
28  <\print>
29 </TinyLangProgram>

```

Listing 3.1: XML representation of AST shown in figure 2.15

Class XMLGeneration implements interface Visitor as shown in figure 3.1. Apart from the visitor methods the class also holds these attributes and methods:

- String xmlRepresentation -> the actual XML of the program in consideration

- `int indentation` -> keeps track of the current indentation level and updates accordingly
 - we have a method which generate an indentation as a sequence of tabs (0x09) spaces where the number of white spaces correspond to indentation

```

1 getCurrentIndentation():
2     String indentation = ""
3     for(i<0;i<this.indentation;i++):
4         indentation+="    "
5     return indetation
6

```

Listing 3.2: Get current indentation

- Constructor is given by:

```

1 XmlGeneration(Tree tree):
2     //visit the whole abstract syntax tree is
3     //equivalent to visiting a program
4     visitProgram(tree)
5

```

Listing 3.3: XmlGeneration Constructor

3.2.1 | visit Program and Statement(s)

The root node of any AST is of type `AST_PROGRAM_NODE` hence we start any XML Generation with tag `<TinyLangProgram>`.

The children of the program node are statement subtrees (as described in section 2.3.2).

Hence for each child we call a method `visitStatment(tree)` which identifies the type of tree and calls the required visitor accordingly so the right tags and content are produced.

```

1 xmlRepresentation+=getCurrentIndentation()+"<TinyLangProgram>\n"
2 //we indent next body
3 indentation++
4 for(child : currentTree.children)
5     visitStatement(child)
6 //unindent (tags attain same level of indentaion)
7 indentation--
8 xmlRepresentation+=getCurrentIndentation()+"<\TinyLangProgram>\n"

```

Listing 3.4: PSEUDOCODE of `visitProgram(tree)`

Note: opening and closing tags attains the same level of indentation
 Method `visitStatement(tree)` call another other visitor method that visit nodes of statement type based on the type of the node.

```

1 If tree/node is of type:
2 AST_VARIABLE_DECLARATION_NODE -call-> visitVariableDeclarationNode(
   tree)
3 AST_ASSIGNMENT_NODE -call-> visitAssignmentNode(tree)
4 AST_PRINT_STATEMENT_NODE -call-> visitPrintStatementNode(tree)
5 AST_IF_STATEMENT_NODE -call-> visitIfStatementNode(tree)
6 AST_FOR_STATEMENT_NODE -call-> visitForStatementNode(tree)
7 AST_WHILE_STATEMENT_NODE -call-> visitWhileStatementNode(tree)
8 AST_RETURN_STATEMENT_NODE -call-> visitReturnStatementNode(tree)
9 AST_FUNCTION_DECLARATION_NODE -call-> visitFunctionDeclarationNode(
   tree)
10 AST_BLOCK_NODE -call-> visitBlock(tree)
11 otherwise -> throw exception unexpected

```

Listing 3.5: PSEUDOCODE for *visitStatement(tree)*

Let us take a look at an example of a statement type visitor method :

visitVariableDeclaration(Tree tree).

A variable declaration statement tree has 3 children. The first child of type *AST_TYPE_NODE* which correspond to the type of the expression, the second child is of *AST_IDENTIFIER_NODE* which corresponds to the name given to the variable, and the third child is an expression subtree. Visitor methods whose type is related to expression are discussed in section 3.2.2 .

We use this structure of the tree to generate its corresponding XML representation:

```

1 <variable declaration>
2   <id type=TYPE> variable name<\id>
3   .... | some
4   .... | expression
5   .... | body
6 <\variable declaration>

```

Listing 3.6: XML of variable declaration statement subtree

The PSEUDOCODE to build a XML representation of the variable declaration statement is as follows .

```

1 xmlRepresentation += getCurrentIndentation ()+"<variable declaration
   >\n"
2 //we indent for next body
3 indentation ++
4 //get type from first child and identifier from second child
5 xmlRepresentation += getCurrentIndentation ()+"<id type="+tree.
   getChildren().get(0).value+">" + tree.getChildren().get(1).value
   +"<\id>\n"
6 //visit expression -> third child
7 visitExpression(tinyLangAst.getChildren().get(2))
8 // unindent (tags attain same level of indentation )
9 indentation --

```

```

10 xmlRepresentation += getCurrentIndentation ()+"<\variable
    declaration>\n"

```

Listing 3.7: PSEUDOCODE of *visitVariableDeclarationNode(Tree tree)*

Note: opening and closing tags attains the same level of indentation

The other statement visit methods are implemented similar to as shown in listing above.

3.2.2 | visit Expression(s)

Almost all statements have an expression subtree. So we need to decide what the type of expression is so we produce the right nested tags and expressions. Whenever a call to an expression visit method needs to be made, we first call *visitExpression(Tree tree)*, and visit expression calls the required visit method according to the type of the tree. For example if the tree is of type *AST_UNARY_OPERATOR_NODE* then we call *visit*

In general we have the following:

```

1 AST_BINARY_OPERATOR_NODE -call-> visitBinaryOperatorNode(tree)
2 AST_UNARY_OPERATOR_NODE -call-> visitUnaryOperatorNode(tree)
3 AST_BOOLEAN_LITERAL_NODE -call-> visitBooleanLiteralNode(tree)
4 AST_INTEGER_LITERAL_NODE -call-> visitIntegerLiteralNode(tree)
5 AST_FLOAT_LITERAL_NODE -call-> visitFloatLiteralNode(tree)
6 AST_CHAR_LITERAL_NODE -call-> visitCharLiteralNode(tree)
7 AST_IDENTIFIER_NODE -call-> visitIdentifierNode(tree)
8 AST_FUNCTION_CALL_NODE -call-> visitFunctionCallNode(tree)
9 otherwise -> throw exception unexpected

```

Listing 3.8: PSEUDOCODE for *visitExpression(Tree tree)*

For example for a binary operator tree we have that the root of the tree is a binary tree whose root is a binary operator and its 2 children are expression tree in its own right. Therefore the 2 intended tags inside a binary operation expression will be expression tags in their own right.

The visitor *visitBinaryOperatorNode(tree)* is defined recursively as follows. //we obtain value of binary operator and append it to opening tag

```

1 xmlRepresentation += getCurrentIndentation ()+"<binary Op="+tree.
    value+">\n"
2 //check for 2 children (we expect 2 children)
3 if(tree.getChildren().size!=2)
4     throw unexpected
5 //we indent for next body
6 indentation ++
7 //visit first child (an expression)
8 visitExpression(tree.getChildren().get(0));
9 //visit first child (an expression)

```

```

10 visitExpression(tree.getChildren().get(1));
11 // unindent (tags attain same level of indentation )
12 indentation --
13 xmlRepresentation += getCurrentIndentation ()+"<\variable
    declaration>\n"

```

Listing 3.9: PSEUDOCODE for visitBinaryOperatorNode(Tree tree)

visitFunctionCallNode(Tree tree) and

visitUnaryOperatorNode(Tree tree) has an analogous recursive definition to one given in the listing above.

The implementations of the other expression visitor methods i.e.

visitBooleanLiteralNode (tree), visitIntegerLiteralNode (tree),

visitFloatLiteralNode (tree), visitCharLiteralNode(tree) and visitIdentifierNode (tree) is trivial and they act as the **base case** for the recursion.

3.3 | Implementation in Java

All the design decisions mentioned in 3.2 are implemented as shown in listing 6.11

3.4 | Testing

Consider Program 1,2,3 discussed in testing Tasks 1,2,3 their XML representations are given below:

```

<TinyLangProgram>
  <variable declaration>
    <id type="FLOAT">numru</id>
    <binary Op="+>
      <unary Op="->
        <integer literal>2</integer literal>
      </unary>
      <float literal>3.2</float literal>
    </binary>
  </variable declaration>
  <print statement>
    <id>numru</id>
  </print statement>
</TinyLangProgram>

```

Figure 3.2: XML representation for program 1

```

<TinyLangProgram>
  <function declaration>
    <id type="BOOL">forLoop<\id>
    <parameters>
    <\parameters>
    <block>
      <for statement>
        <variable declaration>
          <id type="INTEGER">i<\id>
          <integer literal>1<\integer literal>
        <\variable declaration>
        <binary Op>=<+>
          <id>i<\id>
          <integer literal>10<\integer literal>
        <\binary>
        <block>
          <print statement>
            <id>i<\id>
          <\print statement>
        <\block>
        <\VariableDeclaration>
        <return statement>
          <boolean literal>true<\boolean literal>
        <\return statement>
      <\block>
    <\function declaration>
    <variable declaration>
      <id type="BOOL">x<\id>
    <function call>
      <id>forLoop<\id>
      <parameters>
      <\parameters>
    <\function call>
    <\variable declaration>
    <print statement>
      <id>x<\id>
    <\print statement>
  <\TinyLangProgram>

```

Figure 3.3: XML representation for program 2

```

<TinyLangProgram>
  <variable declaration>
    <id type="BOOL">x<\id>
    <boolean literal>false<\boolean literal>
  <\variable declaration>
  <function declaration>
    <id type="BOOL">forLoop<\id>
    <parameters>
    <\parameters>
    <block>
      <for statement>
        <variable declaration>
          <id type="INTEGER">i<\id>
          <integer literal>1<\integer literal>
        <\variable declaration>
        <binary Op>=<+>
          <id>i<\id>
          <integer literal>10<\integer literal>
        <\binary>
        <block>
          <print statement>
            <id>i<\id>
          <\print statement>
        <\block>
        <\VariableDeclaration>
        <return statement>
          <boolean literal>true<\boolean literal>
        <\return statement>
      <\block>
    <\function declaration>
    <assignment>
      <id>x<\id>
      <id>x<\id>
    <\assignment>
    <if statement>
      <binary Op>=<+>
        <id>x<\id>
        <boolean literal>true<\boolean literal>
      <\binary>
      <block>
        <print statement>
          <char literal>'T'<\char literal>
        <\print statement>
      <\block>
    <else block>
      <print statement>
        <char literal>'F'<\char literal>
      <\print statement>
    <\else block>
    <\if statement>
  <\TinyLangProgram>

```

Figure 3.4: XML representation for program 3

Task 4 | Semantic Analysis

The SemanticAnalyser class implements visitor methods to traverse the AST to ensure that the program semantics are correct before moving to the interpretation tree

4.1 | Design

4.1.1 | Scopes

We keep track of scopes. A global scope is created in the constructor, and then we visit the program tree. A global scope is destroyed after a successful visit of the program tree without any errors. If we manage to reach the end of the global scope, then we conclude that the program is semantically correct.

```
1 //create new scope
2 st.push()
3 //traverse the program
4 visitProgram(tree)
5 //end scope
6 st.pop()
7 print("program semantically correct")
```

Listing 4.1: PSEUDOCODE : constructor (start of program traversal)

A new local scope is created and destroyed when control enters and leaves a block respectively. Therefore each time we visit a block we push a new scope to the symbol table and at the end of the visit we pop out the scope.

```
1 //create new scope
2 st.push();
3
4 -> add parameters of functions if any in scope
5 -> clear currentFunctionParameters
6 -> visit all statements in block
7
8 //end scope
9 st.pop();
```

Listing 4.2: PSEUDOCODE : visitBlockNode(Tree tree)

Note: Whenever a new function declaration is made we update a map `currentFunctionParameters` defined in semantic analyser class as $Identifier \rightarrow Type$. So whenever we enter a new scope from the function declaration we have a reference to the identifier and type of function parameters.

Note: A program is visited in a similar way to Task 3 and we deduce what visit method to use based on the type of the root node.

4.1.2 | Variable re-declaration

Note a scope keeps a mapping between a variable name and a type.

Note:

```

1 {
2     let a : float = 5;
3     print a;
4     {
5         let a : char = 'a';
6         print a;
7     }
8 }
```

Listing 4.3: The same variable name can be used in different scopes

The first print method will print 5 and the second print method will print 'a'. We check if a variable is already declared in current scope by checking if it is mapped to some type in that scope via function `isVariableNameBinded(String varName)`.

```

1 //second child of the tree corresponds to identifier
2 Tree identifier = tree.getChildren().get(1)
3 //check if name of identifier is already declared in current scope
4 if(st.isVariableNameBinded(identifier.getAssociatedNodeValue())==
   true)
5     throw exception
```

Listing 4.4: PSUEDOCODE : checking if a variable is already declared in currentScope (in method `visitVariableDeclarationNode(Tree tree)`)

4.1.3 | Function Overloading

This section describes how we allow for function overloading and described the visitor method:

We allow for function overloading by defining the signature of a function.

A function signature is made up by the name of the function and the types of the parameters. For example,

- (a) `fn Sq (int x , float y) -> int ...`
- (b) `fn Sq (int a , float b) -> char ...`
- (c) `fn Sq (float x , float y) -> float ...`

(a) and **(b)** have the same signature, **(b)** and **(c)** do not have the same signature.

This is implemented by constructing a `FunctionSignature` class which contains `String functionName` and a stack of type `Type` where `Type` is an ENUM defined by constants `BOOL`, `INTEGER`, `CHAR` and `FLOAT`).

Each unique instance of `functionSignature` is a unique function signature. In each scope we have a mapping between `functionSignature` and enum `Type`.

```
1 //check if a function is declared within a scope
2 boolean isFunctionAlreadyDefined(FunctionSignature signature):
3     return functionDeclarationMap.containsKey(signature)
```

Listing 4.5: PSEUDOCODE of `isFunctionAlreadyDefined(FunctionSignature signature)` inside class `SCOPE`

Any scope can contain function declaration and once a scope dies that function is undeclared.

When declaring a new function we check if a function with the same signature is already declared in the current and even outer scopes (for variable we only check the current scope i.e. variable can be declared in new scopes even if they are already declared in outer ones). Therefore in `visitFunctionDeclaration(Tree tree)` we obtain the function name and the function parameter types from the children and we check if they are defined

```
1 for(Scope scope : st.getScopes()):
2     if(scope.isFunctionAlreadyDefined(new FunctionSignature(
3         functionName, functionParameters))):
4         throw error
```

Listing 4.6: PSEUDOCODE: checking if a function is already declared in all scopes

```
1 st.insertFunctionDeclaration(new FunctionSignature(functionName,
2     functionParameters))
```

Listing 4.7: PSEUDOCODE: if a function is not declared in all scopes we push it to current scope via class `SymbolTable`

4.1.4 | Type Checking

4.1.4.1 | Visit Expression

A semantic analyser class has an attribute `Type currentExpressionType` to make reference to the type of the current expression.

We visit an expression to find the type value returned by the expression and update `currentExpressionType` for **type checking**.

An expression can take many forms. Whenever we have to visit an expression tree we call `visitExpression()`. This method calls the required visitor according to the node/expression type as discussed in 3.8

We check the type of the expression as follows.

- If expression is of type **binary expression**.
`visitBinaryOperatorNode(Tree tree)` is implemented as follows. We get hold on what the operator is by checking the value associated with node/tree. Since the 2 children are expression in their own right we make a recursive call to `visitExpression(Tree)` to obtain the type of both operands. Then we perform type checking and update `currentExpressionType` accordingly.
 - If the operator is 'and' or 'or' we check that both operands types are bool else we throw exception. We also set `currentExpressionType` to bool.
 - Else If the operator is +, -, / and * we check that both operands types is of numeric type (float or int) else throw exception. If one of the operands is of type float we set `currentExpressionType` to float otherwise we set it to int.
 - Else If the operator is <, >, <= and >= we check that both operands types is of numeric type (float or int) else throw exception. We also set `currentExpressionType` to bool.
 - Else If the operator is ==, != we check that both operands are of the same type otherwise we throw error. We set `currentExpressionType` to bool.
 - Else we throw **exception unrecognised operator**
- If expression is of type **unary expression**.
 - We check that the unary operator is +, -, or not otherwise we throw exception.

- The only child of the unary operator tree is an expression in its own right. We visit the unary tree child and update `currentExpressionType()`.
- If current expression type is of numeric type (i.e. integer or float) we check if the operator is `-` or `+` otherwise we throw error.
- If current expression type is of bool type we check if the operator is `not` otherwise we throw error.
- If expression is of type **integer literal node expression**.
 - We just set `currentExpressionType` to `int`.
- If expression is of type **float literal expression**.
 - We just set `currentExpressionType` to `float`.
- If expression is of type **boolean literal expression**.
 - We just set `currentExpressionType` to `boolean`.
- If expression is of type **char literal expression**.
 - We just set `currentExpressionType` to `char`.
- If expression is of type **identifier**
 - Start traversing the scopes from the innermost scopes to check in which most inner scope the identifier is declared. Then we set `currentExpressionType` to the type of the variable in that scope.
- If expression is of type **function call**
 - Get hold of the function signature by checking function name and each type of the actual parameters (expression).
Start traversing scopes to see where the function is defined and use method `getFunctionType(signature)` in that scope to obtain the type of the function and set `currentExpressionType` to it.

4.1.4.2 | Considerations

We allow integer literals to resolve to float type.

E.g. `let x:float=5;` is **allowed**.

We do not allow float literals to resolve to integer type. E.g. `let x:int=5.01;` is **not allowed**.

- **Variable Declaration.**

- We visit the expression (3rd child) then we check if `currentExpressionType` is the same as the type of the variable. Note that by the consideration shown above the type of var can be float and the type of expression can be int but not the other way around.

- **Function Declaration.**

- We check that a function returns.

A check to see if the type of expression it returns matches the type of the function was not implemented.

- **Assignment.**

- Similar to the case of variable declaration but we obtain the type of the variable by searching from the innermost scope, where the variable is declared and obtain its type by calling `getVariableType(identifier)` in that scope instance.

4.1.5 | Checking if a function returns

A function must reach a return statement. This can be tricky when we have branching.

A predicate function `returns(Tree tree)` takes a block tree and returns true if a return statement is reached unconditionally.

The method is built recursively to deal with statements which have blocks.

Otherwise if a statement is a simple statement (i.e. no blocks) and is not a return statement then `returns(statement)` is guaranteed to be false.

For the recursive parts we consider the following cases:

- For an if statement we check if we have both the normal block and the else block else the statement is not guaranteed to return. Then we check if both block returns.

- A block/else-block is returning if it contains a statement that returns.
- For for and while loops we check if the block returns.

The PSEUDOCODE is given below.

```

1 Base Case (trivial case) tree represents a return statement :
2 if(tree.type=AST_RETURN_STATEMENT):
3     return true
4 //(Recursive case) if statement is a block we check if one of the
   statements inside the
5 //block returns
6 if(tree.type=AST_BLOCK_NODE):
7     for(Tree statement : tree.getChildren()):
8         //if one statement within block
9         //returns the whole block returns
10        if(returns(statement)):
11            return true
12
13 )
14 //(Recursive case) if statement is an else block we check if one of
   the statements inside
15 // the else block returns
16 if(tree.type=AST_ELSE_BLOCK_NODE):
17     for(Tree statement : tree.getChildren()):
18         //if one statement within block
19         //returns the whole block returns
20        if(returns(statement)):
21            return true
22 )
23
24 //(Recursive case) if statement is an if statement block has an else
   block
25 (i.e. if statement tree has 3 children)
26 //and check if the block and else block contains
27 //a returning function
28 if(tree.type=AST_IF_STATEMENT_NODE):
29     if(tree.getChildren().size=3):
30         //check if children block tree and else block tree returns
31         return returns(tree.getChildren().get(1) and returns(tree.
   getChildren().get(2))
32 //(Recursive case) check that block inside for/while loops is
   returning
33 if(tree.type=AST_FOR_STATEMENT_NODE):
34     //block statement is last child
35     //a for statement can have different amount
36     //of children
37     return returns(tree.getChildren().get(tree.getChildren().size-1)
   )
38 if(tree.type=AST_FOR_STATEMENT_NODE):
39     //block statement is second child
40     return returns(tree.getChildren().get(1))

```

```

41 // (base case) otherwise in all other cases
42 else:
43     return false

```

Listing 4.8: PSEUDOCODE: Defining predicate *returns(Tree tree)*

4.2 | Implementation in Java

- The implementation class `FunctionSignature` whose instance are used to check if functions are already declared is given in listing 6.12
- The implementation class `Scope` and the data structures and methods required to make reference to the variables, functions is given in listing 6.13
- The implementation of class symbol table which holds a stack of scopes, and method to insert/destroy scopes etc is given in listing 6.14
- All points discussed through the design section 4.1 are implemented as shown in listing 6.15

4.3 | Testing

Let us test some program that are is semantically incorrect and ensure that an appropriate exception is produced.

- A program with a function that is not guaranteed to return (conditional branching).

```

1 //a function must always return, this function is not guaranteed to return
2 fn notGuranteedToReturn(x:char)->bool{
3     if(x=='a'){ return true; }
4     else{
5         //conditional branching
6         if(x=='b'){ return true; }
7     }
8     if(x=='c'){ return true; }
9     else{
10        //no return statement in this block
11        print 'c';
12    }
13 }
14 }
15 }
16

```

Listing 4.9: Program 4

The following exception is thrown:

```
Exception in thread "main" java.lang.RuntimeException: function
notGuranteedToReturn in line 2 not expected to return
```

Figure 4.1: Exception

- Expecting a bool expression

```
1 //expecting an expression type of bool
2 // i+10 is not of type bool -> error
3 let i:int=1;
4 while(i+10){
5   print i;
6 }
7
```

Listing 4.10: Program 5

The following exception is thrown:

```
Exception in thread "main" java.lang.RuntimeException: expected
while condition to be a predicate in line 4
```

Figure 4.2: Exception

- A program where 2 variables with the same name are defined in the same scope.

```
1 fn notNice(x:char)->bool{
2   let x:bool=false;
3   //error variable x already declared
4   let x:int=5;
5   print x;
6 }
7
8
```

Listing 4.11: Program 6

The following exception is thrown:

```
Exception in thread "main" java.lang.RuntimeException: variable x in line
2 was already declared previously
```

Figure 4.3: Exception

- A function redeclared inside same program (same signature)

```
1 fn notNice(x:char)->bool{
2   fn notNice(x:char)->int{
3     return 0;
4   }
5   return notNice('a');
6 }
7
```

Listing 4.12: Program 7

The following exception is thrown:

```
Exception in thread "main" java.lang.RuntimeException: function notNice in line
2 with the same parameter types already defined previously
```

Figure 4.4: Exception

- A program that is semantically correct with **Function Overloading** -> a program containing program with same identifier but having parameter of different types

```
1 fn nice(x:int)->float{
2   fn return2()->int{
3     return 2;
4   }
5   return x+return2();
6 }
7 fn nice(x:float)->float{
8   fn return3()->int{
9     return 3;
10  }
11  return x+return3();
12 }
13 print nice(1);
14 print nice(1.0);
15
```

Listing 4.13: Program 8

The program is semantically correct and the following output is thrown when interpreted (Task 5):

```
Note: program is semantically correct
3
4.0
```

Figure 4.5: Exception

Task 5 | Interpreter

5.1 | Design

This task is similar to task 4. We need to have an appropriate symbol table to keep track of scopes. In this task, we also require that the symbol table holds values so we can simulate an interpreter which executes the test program.

5.1.1 | Scope

The following functionality was added in classes `SemanticAnalyser` and `Scope`.

- In each scope we keep a mapping between variable names and their values `Map<String,String> variableValues`. A mapping between a function signature and its block tree `Map<FunctionSignature,Tree> functionBlock`. A mapping between function signatures and their respective parameter names `Map<FunctionSignature,Stack<String>> functionParameterNames`.
- Methods `addVariableValue(String varName,String varValue)` to map a value to a variable,
`addFunctionBlock(FunctionSignature functionSignature,Tree block)` to map a block tree to a function and
`addFunctionParameterNames(FunctionSignature fs, Stack<String> names)` to map parameter identifiers to function.

Ensure that program semantics are correct to ensure that a program can be interpreted correctly. A call to semantic analyser is made in constructor of the interpreter via
`new SemanticAnalyser(treeIntermediateRepresentation).`

5.1.2 | Evaluation of expressions

An interpreter class has values : `Type currentExpressionType` and `String currentExpressionValue` to keep track of the value and type of the latest evaluated expression.

Whenever a statement needs to evaluate an expression (i.e. has an expression subtree) we call method `visitExpression(Tree tree)` which then makes a call to another visitor method base on the current node type/type of expression (see listing 3.8).

- If current node type is of type `AST_BINARY_OPERATOR_NODE` a call to `visitBinaryOperatorNode(tree)` is made.
 - We keep hold of the operator which is given by the node value. The left and right operands (2 children) are expression trees in their own right. A recursive call `visitExpression(tree)` is made on both operands to obtain their type and value (by seeking the values of `currentExpressionType` and `currentExpressionValues`). Then we update the value of `currentExpressionType` and `currentExpressionValue` based on the binary operator and the values and types of both operands. For example consider the following scenarios:
 - ◇ Operator is "+" and both operators are of type `int` with values "1" and "3". We set the `currentExpressionType` to `int` and the `currentExpressionValue` to `String.valueOf(Int.parseInt("1")+Int.parseInt("3"))="4"`
 - ◇ **(Implicit Typecasting Case)** Operator is "*" and one operators of type `int` and the other is of type `float` with values "2" and "3.3". We set the `currentExpressionType` to `float` and the `currentExpressionValue` to `String.valueOf(Float.parseFloat("2")*Float.parseFloat("3.3"))="6.6"`
 - ◇ (`currentExpressionType` **depends on value case**)
 - ◇ Operator is "==" and both operands are of the same type otherwise we throw error unexpected. We set the `currentExpressionType` to the type of the operands and the `currentExpressionValue` to "true" if both operands have the same value (i.e. `value1.equals(value2)`), otherwise we set it to false.

- If current node type is of type `AST_UNARY_OPERATOR_NODE` a call to `visitUnaryOperatorNode(tree)` is made.
 - A unary tree has an expression subtree as its child. We call `visitExpression(Tree tree)` on its child to update `currentExpressionValue` and `currentExpressionType`. If the current expression type is `int` or `float` we check if the unary operator is `-`. If it is we update the value of the current expression e.g.
`currentExpressionValue=String.valueOf(-1*Integer.parseInt(currentExpressionValue))`. Else if the current expression is of type `bool` we check if the unary operator is `not` we update current expression value to `false` if it was `true` and to `true` if it was `false`. Else we throw error `unexpected`.
- If current node type is of type `AST_BOOLEAN_LITERAL_NODE` a call to `visitBooleanLiteralNode(tree)` is made.
 - When we visit a node of type `boolean literal` we set the current expression type to `bool` and the current expression value to the value associated with the node (`true` or `false`).
 - Nodes of type `AST_INT_LITERAL`, `AST_FLOAT_LITERAL` and `AST_CHAR_LITERAL` are dealt with in a similar way.
- If current node type is of type `AST_IDENTIFIER_NODE` a call to `visitIdentifierNode(tree)` is made.
 - We get hold on the value/identifier name of the node. We search the most inner scope the variable name is declared by calling `scope.isVariableNameBinded(identifier)` in each scope. We obtain the type and value of the variable at that scope by calling methods `scope.getVariableType(identifier)` and `scope.getVariableName(identifier)` and we update the `currentExpressionValue` and `currentExpressionType`.
- If current node type is of type `AST_FUNCTION_CALL_NODE` a call to `visitFunctionCallNode(tree)` is made.
 - Class interpreter have 2 other data structures `parameterTypes` and `parameterValues`.
 - For a function call we obtain the name of the function and the expression, and we visit all the actual parameters/expressions to

obtain the `currentExpressionType` and push it to `parameterTypes` and obtain `currentExpressionValue` and push it to `parameterValues`

- We start searching the scopes to find where the function is declared. We check if a function is declared in a scope using the instance method `isFunctionAlreadyDefine(new FunctionSignature(functionName, parameterTypes, parameterValues))`.
- The interpreter also has a data structure `parameterNames` so when we visit a block node corresponding to the function we can declare the local function variables.
- We obtain the block corresponding to a declared function in some scope by using the instance method `getBlock(Function Signature)`

5.1.3 | Evaluation of statements

A program is a sequence of statements, for each statement we call method `visitStatement(Tree tree)` which then makes a call to another visitor method base on the current node type of statement (see listing 3.5).

- If current node type is of type `AST_VARIABLE_DECLARATION_NODE` a call to `visitVariableDeclarationNode(tree)` is made.
 - We obtain the type and var name from the first and second children respectively. We visit the expression tree (3rd child) to update `currentExpressionType` and `currentExpressionValue`. We push the variable type, name and value in current scope.
 - ◇ `st.insertVariableDeclaration(varName, varType)`
 - ◇ `st.insertVariableValue(varName, currentExpressionValue)`
- If current node type is of type `AST_ASSIGNMENT_NODE` a call to `visitAssignmentNode(tree)` is made.
 - Obtain the identifier name from the value associated with the first child. Visit the expression (2nd child) to update `currentExpressionType` and `currentExpressionValue`. Then we search the inner most scope where the variable is declared and update the variable value by calling instance method

`addVariableValue(varName, currentExpressionValue)`
 which updates the value of the map $varName \rightarrow varValue$
 in the innermost scope.

- If current node type is of type `AST_PRINT_STATEMENT_NODE` a call to `visitPrintStatementNode(tree)` is made.
 - **This allows us to test the program by verifying the output.**
 - we visit expression tree (first child) and update the current expression value then we print the value of the current expression
`System.out.println(currentExpressionValue)`
- If current node type is of type `AST_IF_STATEMENT_NODE` a call to `visitIfStatementNode(tree)` is made.
 - We visit the expression fir child and `updatecurrentExpressionValue` and `currentExpressionType`. We check if the current expression.
 - If the `currentExpressionValue` is true we visit the block node.
 - If the `currentExpressionValue` is false we visit the else block (we also check that an else block exists i.e. we check also that if statement has 3 children).
- If current node type is of type `AST_FOR_STATEMENT_NODE` a call to `visitForStatementNode(tree)` is made.
 - To deal with the different cases the for loop was encoded as a while loop.
 - When the for loop has no variable declaration and assignment (only an expression) we keep repeating the statements in the block statements and update the expression (to update truth value). PSEUDOCODE code is given below.

```

1 while(currentExpressionValue.equals("true")){
2     visitBlockNode(block)
3     //update current expression value
4     visitExpression(expression);
5 }
6

```

Listing 5.1: `for(;expression;){...}` encoded as while loop

- When the for loop has both a variable declaration and an assignment the first child is a variable declaration statement so we call `visitVariableDeclarationNode(first child subtree)` to declare the variable in current scope and we visit the expression (2nd child) to update `currentExpressionValue` to check if it is true or false. Then for loop is encoded in a while loop as given in the following PSEUDOCODE.

```

1 while(currentExpressionValue="true"){
2     //4th child correspond to block node
3     visitBlockNode(4th child)
4     //visit assignment node (upadation)
5     visitAssignment(third child)
6     //update truth value
7     visitExpression(2nd child)
8 }
9

```

Listing 5.2: *for(declaration;expression;assignment){...}*

- After the while loop stops then we delete the variable assigned in the while loop from the current instance by calling instance method `st.deleteVariable`
- **We deal with the other cases i.e. case where we have no assignment and case where we have no variable declaration using a similar strategy.**
- If current node type is of type `AST_WHILE_STATEMENT_NODE` a call to `visitWhileStatementNode(tree)` is made. A while statement is easily implemented using a while loop itself.

```

1 //first child corresponds to expression -> update current
  expression value
2 visitExpression(first child)
3 while(currentExpressionValue.equals("true")):
4     visitBlockNode(block);
5     visitExpression(expression);
6

```

Listing 5.3: encoding of while loop

- If current node type is of type `AST_RETURN_STATEMENT_NODE` a call to `visitReturnStatementNode(tree)` is made.

- All we do is just update `currentExpressionType` and `currentExpressionValue` by visiting the expression tree (1st child).
- If current node type is of type `AST_FUNCTION_DECLARATION_NODE` a call to `visitFunctionDeclarationNode(tree)` is made.
 - We obtain the type, name and block tree of the function from the first, second and fourth child respectively. We traverse the formal parameters tree (3rd child) to obtain a stack `functionParameterTypes` and a stack `functionParameterNames` of function parameter types and names respectively.
 - We insert the function declaration in current scope by calling the instance method `st.insertFunctionDeclaration(new FunctionSignature(functionName,functionParameters))`. Similarly we map the function parameter names and the function block to the function signature in the current scope.
- If current node type is of type `AST_BLOCK_NODE` a call to `visitBlockNode(tree)` is made.
 - We create a new local scope `st.push()` when we start traversing the block subtree and destroy scope `st.pop` when we leave.
 - Also before we start traversing the statements we declare the function parameters inside the newly created local scope.
 - After they have been declared we clear any data structures in class `Interpreter` holding information about formal parameters.

5.2 | Implementation in Java

All the design decisions implemented in section 5 are implemented in Java as shown in listing 6.16

5.3 | Testing

Let us interpret some programs

- Printing HelloWorld character by character.

```
1 print 'H';
2 print 'e';
3 print 'l';
4 print 'l';
5 print 'o';
6 print 'W';
7 print 'o';
8 print 'r';
9 print 'l';
10 print 'd';
11
```

Listing 5.4: helloworld.tl

Interpreter produces the following output:

```
Note: program is semantically correct
'H'
'e'
'l'
'l'
'o'
'W'
'o'
'r'
'l'
'd'
```

Figure 5.1: Output of helloworld.tl

- **(Recursion)** Find the 12th Fibonacci number
(1,1,2,3,5,8,13,21,34,55,89,**144**,....)

```
1 fn fib(n:int)->int
2 {
3   if(n<=1) {return n;}
4   else {return (fib(n-1)+fib(n-2));}
5 }
6 print fib(12);
7
```

Listing 5.5: fibonacci.tl

Interpreter produces the following output:

```
Note: program is semantically correct
144
```

Figure 5.2: Output of fibonacci.tl

- We consider variable values in the most inner scope.

```
1 {
2   let a:int=5;
3   print a;
4   {
5     let a:int=6;
6     print 6;
```

```

7   {
8       let a:int=7;
9       print 7;
10  }
11  }
12 }
13

```

Listing 5.6: variables variables.tl

Interpreter produces the following output:

```

Note: program is semantically correct
5
6
7

```

Figure 5.3: Output of variables.tl

- Some recursive operators on \mathbb{N} .

```

1 fn add(a:int,b:int)->int{
2     if(b==0){return a;}
3     else{
4         return add(a+1,b-1);
5     }
6 }
7 //reuse add function
8 fn multiply(a:int,b:int)->int{
9     if(b==0){return 0;}
10    else{
11        return a+multiply(a,(b-1));
12    }
13 }
14 //a^b, reuse multiply function
15 fn power(a:int,b:int)->int{
16     if(b==0){return 1;}
17     else{
18         if(b==1){return a;} else{
19             return a*power(a,b-1);
20         }
21     }
22 }
23 print add(5,3);
24 print multiply(5,3);
25 print power(5,3);
26

```

Listing 5.7: recursive.tl

```

Note: program is semantically correct
8
15
125

```

Figure 5.4: Output of recursive.tl

- A program that uses the previous functions to work out the summation

$$\sum_{k=0}^5 k^2 + (2 * k + 2) = 2 + 5 + 10 + 17 + 26 + 37 = 97$$


```

1 fn add(a:int,b:int)->int{
2   if(b==0){return a;}
3   else{
4     return add(a+1,b-1);
5   }
6 }
7 //reuse add function
8 fn multiply(a:int,b:int)->int{
9   if(b==0){return 0;}
10  else{
11    return a+multiply(a,(b-1));
12  }
13 }
14 //a^b, reuse multiply function
15 fn power(a:int,b:int)->int{
16   if(b==0){return 1;}
17   else{
18     if(b==1){return a;} else{
19       return a*power(a,b-1);
20     }
21   }
22 }
23 let sum:int=0;
24 for(let i:int=0;i<=5;i=i+1){
25   let a:int = power(i,2);
26
27   let b:int = multiply(i,2);
28   let c:int = add(b,2);
29   let d:int = add(a,c);
30   print d;
31   sum=sum+d;
32 }
33 }
34 print sum;
35

```

Listing 5.8: sum.tl

Note: program is semantically correct
97

Figure 5.5: Output of sum.tl

5.4 | Future implementation

I wish to add the following features to the next iteration of tinylang:

- Allow use of more complex data structures such as string and arrays.
- Have more expressive printing methods.
- Allow a program to make references to other programs.

6 | Implementation

6.1 | Lexer

```
1 package tinylanglexer;
2 public enum StateType {
3     ACCEPTING,
4     REJECTING
5 }
```

Listing 6.1: State type

```
1 package tinylanglexer;
2 public enum State {
3     /**
4      * The starting state of representing TinyLang's grammar.
5      */
6     STARTING_STATE (StateType.REJECTING),
7     STATE_1 (StateType.REJECTING),
8     STATE_2 (StateType.REJECTING),
9     /* Lexemes leading to STATE_3 -> Lexeme of type TOK_CHAR_LITERAL */
10    STATE_3 (StateType.ACCEPTING),
11    /* Lexemes leading to STATE_4 -> Lexeme of type TOK_IDENTIFIER_LITERAL or other KEYWORD type
12       */
13    STATE_4 (StateType.ACCEPTING),
14    /* Lexemes leading to STATE_5 -> Lexeme of type TOK_MULTIPLICATIVE_OP */
15    STATE_5 (StateType.ACCEPTING),
16    /* Lexemes leading to STATE_6 -> Lexeme of type TOK_SKIP */
17    STATE_6 (StateType.ACCEPTING),
18    /* Lexemes leading to STATE_7 -> Lexeme of type TOK_SKIP */
19    STATE_7 (StateType.REJECTING),
20    STATE_8 (StateType.REJECTING),
21    /* Lexemes leading to STATE_9 -> Lexeme of type TOK_SKIP */
22    STATE_9 (StateType.ACCEPTING),
23    /* Lexemes leading to STATE_10 -> Lexeme of some PUNCTUATION type */
24    STATE_10 (StateType.ACCEPTING),
25    /* Lexemes leading to STATE_11 -> Lexeme of type TOK_INTEGER_LITERAL */
26    STATE_11 (StateType.ACCEPTING),
27    STATE_12 (StateType.REJECTING),
28    /* Lexemes leading to STATE_13 -> Lexeme of type TOK_FLOAT_LITERAL */
29    STATE_13 (StateType.ACCEPTING),
30    /* Lexemes leading to STATE_14 -> Lexeme of type TOK_MUTIPLICATIVE_OP or TOK_ADDITIVE_OP */
31    STATE_14 (StateType.ACCEPTING),
32    /* Lexemes leading to STATE_15 -> Lexeme of type TOK_ADDITIVE_OP */
33    STATE_15 (StateType.ACCEPTING),
34    /* Lexemes leading to STATE_16 -> Lexeme of type TOK_RELATIONAL_OP */
35    STATE_16 (StateType.ACCEPTING),
36    STATE_17 (StateType.REJECTING),
37    /* Lexemes leading to STATE_18 -> Lexeme of type TOK_RELATIONAL_OP */
38    STATE_18 (StateType.ACCEPTING),
39    /* Lexemes leading to STATE_18 -> Lexeme of type TOK_RELATIONAL_OP */
40    STATE_19 (StateType.ACCEPTING),
41    STATE_ERROR (StateType.REJECTING),
42    /* STATE_BAD USED IN ALGORITHM OF GENERATING TOKENS FROM TRANSITION TABLE */
43    STATE_BAD (StateType.REJECTING);
44    private final StateType stateType;
```

```

45  /**
46   *
47   * @param stateId
48   */
49  State(StateType stateType) {
50      this.stateType = stateType;
51  }
52
53  /**
54   * Getter method for getting a state's id
55   * @return
56   */
57  public StateType getStateType() {
58      return this.stateType;
59  }
60  public TokenType getTokenType(String lexeme){
61
62      switch(this) {
63          case STATE_3:
64              return TokenType.TOK_CHAR_LITERAL;
65
66          case STATE_4:
67              switch(lexeme) {
68                  case "fn":
69                      return TokenType.TOK_FN;
70                  case "bool":
71                      return TokenType.TOK_BOOL_TYPE;
72                  case "int":
73                      return TokenType.TOK_INT_TYPE;
74                  case "float":
75                      return TokenType.TOK_FLOAT_TYPE;
76                  case "false":
77                  case "true":
78                      return TokenType.TOK_BOOL_LITERAL;
79                  case "not":
80                      return TokenType.TOK_NOT;
81                  case "let":
82                      return TokenType.TOK_LET;
83                  case "char":
84                      return TokenType.TOK_CHAR_TYPE;
85                  case "if":
86                      return TokenType.TOK_IF;
87                  case "else":
88                      return TokenType.TOK_ELSE;
89                  case "while":
90                      return TokenType.TOK_WHILE;
91                  case "for":
92                      return TokenType.TOK_FOR;
93                  case "print":
94                      return TokenType.TOK_PRINT;
95                  case "return":
96                      return TokenType.TOK_RETURN;
97                  case "and":
98                      return TokenType.TOK_MULTIPLICATIVE_OP;
99                  case "or":
100                     return TokenType.TOK_ADDITIVE_OP;
101
102                 default:
103                     return TokenType.TOK_IDENTIFIER;
104             }
105
106          case STATE_5:
107              return TokenType.TOK_MULTIPLICATIVE_OP;
108
109          case STATE_6:
110              return TokenType.TOK_SKIP;
111
112          case STATE_9:
113              return TokenType.TOK_SKIP;
114
115          case STATE_10:
116              switch(lexeme) {
117                  case ":":

```

```

118     return TokenType.TOK_COLON;
119     case ";":
120     return TokenType.TOK_SEMICOLON;
121     case "(":
122     return TokenType.TOK_LEFT_ROUND_BRACKET;
123     case ")":
124     return TokenType.TOK_RIGHT_ROUND_BRACKET;
125     case "{":
126     return TokenType.TOK_LEFT_CURLY_BRACKET;
127     case "}":
128     return TokenType.TOK_RIGHT_CURLY_BRACKET;
129     case ",":
130     return TokenType.TOK_COMMA;
131     case ".":
132     return TokenType.TOK_DOT;
133     default:
134     return TokenType.INVALID;
135
136 }
137 case STATE_11:
138     return TokenType.TOK_INT_LITERAL;
139
140 case STATE_13:
141     return TokenType.TOK_FLOAT_LITERAL;
142 case STATE_14:
143     switch (lexeme) {
144     case "*":
145     return TokenType.TOK_MULTIPLICATIVE_OP;
146     case "+":
147     return TokenType.TOK_ADDITIVE_OP;
148     default:
149     return TokenType.INVALID;
150     }
151 case STATE_15:
152
153     return TokenType.TOK_ADDITIVE_OP;
154
155 case STATE_16:
156     switch (lexeme) {
157     case "=":
158     return TokenType.TOK_EQUAL;
159     default:
160     return TokenType.TOK_RELATIONAL_OP;
161     }
162 case STATE_18:
163     return TokenType.TOK_RIGHT_ARROW;
164
165 case STATE_19:
166     return TokenType.TOK_RELATIONAL_OP;
167     default:
168     return TokenType.INVALID;
169 }
170 }
171 }

```

Listing 6.2: tinylang's dfsa states

```

1 package tinylanglexer;
2 /**
3  * Consists of all possible inputs
4  * of dfsa representing TinyLang's grammar.
5  *
6  * Total number of inputs : 16
7  * @author andre
8  *
9  */
10 public enum InputCategory {
11     /* LETTER [a,b,...,z,A,B,...,Z] ≡ ASCII LETTER [0x41,0x5a],[0x61,0x7a] */
12     LETTER,
13     /* DIGIT [0,1,2,...,9] ≡ ASCII DIGIT [0x30,0x39] */
14     DIGIT,
15     /* UNDERSCORE [_] ≡ ASCII UNDERSCORE [0x5f] */

```

```

16  UNDERSCORE ,
17  /* SLASH_DIVIDE ? { / } = ASCII SLASH_DIVIDE ? {0x2f} */
18  SLASH_DIVIDE ,
19  /* ASTERISK ? { * } = ASCII ASTERISK ? {0x2a} */
20  ASTERISK ,
21  /* LESS_THAN ? { < } = ASCII LESS_THAN ? {0x3c} */
22  LESS_THAN ,
23  /* FORWARD_SLASH ? { > } = ASCII FORWARD_SLASH ? {0x3e} */
24  GREATER_THAN ,
25  /* PLUS ? { + } = ASCII FORWARD_SLASH ? {0x2B} */
26  PLUS ,
27  /* HYPHEN_MINUS ? { - } = ASCII HYPHEN_MINUS ? {0x2d} */
28  HYPHEN_MINUS ,
29  /* EQUAL ? { = } = ASCII HYPHEN_MINUS ? {0x3d} */
30  EQUAL ,
31  /* EXCLAMATION_MARK ? { ! } = ASCII EXCLAMATION_MARK ? {0x21} */
32  EXCLAMATION_MARK ,
33  /* DOT ? { . } = ASCII HYPHEN_MINUS ? {0x2e} */
34  DOT ,
35  /* SINGLE_QUOTE ? { ' } = ASCII HYPHEN_MINUS ? {0x27} */
36  SINGLE_QUOTE ,
37  /* PUNCTUATION ? { ( , ) , , , : , ; , { , } } = ASCII PUNCTUATION ? {0x28 , 0x29,0x2c , 0x3a , 0x3b,0
    x7b ,0x7d} */
38  PUNCT ,
39  /* ASCII : OTHER_PRINTABLE ? {[0x20,0x7e]}
40  * \ (LETTERS,DIGITS ? UNDERSCORE ? FORWARD_SLASH ? ASTERISK ? LESS_THAN
41  *      ? GREATER_THAN ? PLUS ,MINUS ? EQUAL ? EXCLAMATION_MARK ? DOT
42  *      ? SINGLE_QUOTE ? PUNCTUATION) */
43  OTHER_PRINTABLE ,
44  /* LINE_FEED ? { \n } = ASCII LINE_FEED ? {0x0a} */
45  LINE_FEED
46 }

```

Listing 6.3: Implementation of classifier table

```

1  package tinyanglexer;
2  import java.util.HashMap;
3  import java.util.Map;
4  public class TransitionTable {
5      protected Map<TransitionInput , State > buildTransitionTable() {
6          Map<TransitionInput , State > transitionTable = new HashMap<TransitionInput , State >();
7          State fromState;
8          /***** transition table row 1 *****/
9          fromState = State.STARTING_STATE;
10         transitionTable.put(new TransitionInput(fromState , InputCategory.LETTER) , State.STATE_4);
11         transitionTable.put(new TransitionInput(fromState , InputCategory.DIGIT) , State.STATE_11);
12         transitionTable.put(new TransitionInput(fromState , InputCategory.UNDERSCORE) , State.STATE_4
13         );
14         transitionTable.put(new TransitionInput(fromState , InputCategory.SLASH_DIVIDE) , State .
15         STATE_5);
16         transitionTable.put(new TransitionInput(fromState , InputCategory.ASTERISK) , State.STATE_14)
17         ;
18         transitionTable.put(new TransitionInput(fromState , InputCategory.LESS_THAN) , State.STATE_16
19         );
20         transitionTable.put(new TransitionInput(fromState , InputCategory.GREATER_THAN) , State .
21         STATE_16);
22         transitionTable.put(new TransitionInput(fromState , InputCategory.PLUS) , State.STATE_14);
23         transitionTable.put(new TransitionInput(fromState , InputCategory.HYPHEN_MINUS) , State .
24         STATE_15);
25         transitionTable.put(new TransitionInput(fromState , InputCategory.EQUAL) , State.STATE_16);
26         transitionTable.put(new TransitionInput(fromState , InputCategory.EXCLAMATION_MARK) , State .
27         STATE_17);
28         transitionTable.put(new TransitionInput(fromState , InputCategory.DOT) , State.STATE_ERROR);
29         transitionTable.put(new TransitionInput(fromState , InputCategory.SINGLE_QUOTE) , State .
30         STATE_1);
31         transitionTable.put(new TransitionInput(fromState , InputCategory.PUNCT) , State.STATE_10);
32         transitionTable.put(new TransitionInput(fromState , InputCategory.OTHER_PRINTABLE) , State .
33         STATE_ERROR);
34         transitionTable.put(new TransitionInput(fromState , InputCategory.LINE_FEED) , State .
35         STATE_ERROR);
36         /***** end transition table row 1 *****/
37         /***** transition table row 2 *****/

```

```

28 fromState = State.STATE_1;
29 transitionTable.put(new TransitionInput(fromState, InputCategory.LETTER), State.STATE_2);
30 transitionTable.put(new TransitionInput(fromState, InputCategory.DIGIT), State.STATE_2);
31 transitionTable.put(new TransitionInput(fromState, InputCategory.UNDERSCORE), State.STATE_2);
32 transitionTable.put(new TransitionInput(fromState, InputCategory.SLASH_DIVIDE), State.STATE_2);
33 transitionTable.put(new TransitionInput(fromState, InputCategory.ASTERISK), State.STATE_2);
34 transitionTable.put(new TransitionInput(fromState, InputCategory.LESS_THAN), State.STATE_2);
35 transitionTable.put(new TransitionInput(fromState, InputCategory.GREATER_THAN), State.STATE_2);
36 transitionTable.put(new TransitionInput(fromState, InputCategory.PLUS), State.STATE_2);
37 transitionTable.put(new TransitionInput(fromState, InputCategory.HYPHEN_MINUS), State.STATE_2);
38 transitionTable.put(new TransitionInput(fromState, InputCategory.EQUAL), State.STATE_2);
39 transitionTable.put(new TransitionInput(fromState, InputCategory.EXCLAMATION_MARK), State.STATE_2);
40 transitionTable.put(new TransitionInput(fromState, InputCategory.DOT), State.STATE_2);
41 transitionTable.put(new TransitionInput(fromState, InputCategory.SINGLE_QUOTE), State.STATE_2);
42 transitionTable.put(new TransitionInput(fromState, InputCategory.PUNCT), State.STATE_2);
43 transitionTable.put(new TransitionInput(fromState, InputCategory.OTHER_PRINTABLE), State.STATE_2);
44 transitionTable.put(new TransitionInput(fromState, InputCategory.LINE_FEED), State.STATE_ERROR);
45 /***** end transition table row 2 *****/
46 /***** transition table row 3 *****/
47 fromState = State.STATE_2;
48 transitionTable.put(new TransitionInput(fromState, InputCategory.LETTER), State.STATE_ERROR);
49 transitionTable.put(new TransitionInput(fromState, InputCategory.DIGIT), State.STATE_ERROR);
50 transitionTable.put(new TransitionInput(fromState, InputCategory.UNDERSCORE), State.STATE_ERROR);
51 transitionTable.put(new TransitionInput(fromState, InputCategory.SLASH_DIVIDE), State.STATE_ERROR);
52 transitionTable.put(new TransitionInput(fromState, InputCategory.ASTERISK), State.STATE_ERROR);
53 transitionTable.put(new TransitionInput(fromState, InputCategory.LESS_THAN), State.STATE_ERROR);
54 transitionTable.put(new TransitionInput(fromState, InputCategory.GREATER_THAN), State.STATE_ERROR);
55 transitionTable.put(new TransitionInput(fromState, InputCategory.PLUS), State.STATE_ERROR);
56 transitionTable.put(new TransitionInput(fromState, InputCategory.HYPHEN_MINUS), State.STATE_ERROR);
57 transitionTable.put(new TransitionInput(fromState, InputCategory.EQUAL), State.STATE_ERROR);
58 transitionTable.put(new TransitionInput(fromState, InputCategory.EXCLAMATION_MARK), State.STATE_ERROR);
59 transitionTable.put(new TransitionInput(fromState, InputCategory.DOT), State.STATE_ERROR);
60 transitionTable.put(new TransitionInput(fromState, InputCategory.SINGLE_QUOTE), State.STATE_3);
61 transitionTable.put(new TransitionInput(fromState, InputCategory.PUNCT), State.STATE_ERROR);
62 transitionTable.put(new TransitionInput(fromState, InputCategory.OTHER_PRINTABLE), State.STATE_ERROR);
63 transitionTable.put(new TransitionInput(fromState, InputCategory.LINE_FEED), State.STATE_ERROR);
64 /***** end transition table row 3 *****/
65 /***** transition table row 4 *****/
66 fromState = State.STATE_3;
67 for (InputCategory input : InputCategory.values()) {
68     transitionTable.put(new TransitionInput(fromState, input), State.STATE_ERROR);
69 }
70 /***** end transition table row 4 *****/
71 /***** transition table row 5 *****/
72 fromState = State.STATE_4;
73 transitionTable.put(new TransitionInput(fromState, InputCategory.LETTER), State.STATE_4);
74 transitionTable.put(new TransitionInput(fromState, InputCategory.DIGIT), State.STATE_4);
75 transitionTable.put(new TransitionInput(fromState, InputCategory.UNDERSCORE), State.STATE_4);
76 transitionTable.put(new TransitionInput(fromState, InputCategory.SLASH_DIVIDE), State.STATE_4);

```

```

STATE_ERROR);
77 transitionTable.put(new TransitionInput(fromState, InputCategory.ASTERISK), State.
STATE_ERROR);
78 transitionTable.put(new TransitionInput(fromState, InputCategory.LESS_THAN), State.
STATE_ERROR);
79 transitionTable.put(new TransitionInput(fromState, InputCategory.GREATER_THAN), State.
STATE_ERROR);
80 transitionTable.put(new TransitionInput(fromState, InputCategory.PLUS), State.STATE_ERROR);
81 transitionTable.put(new TransitionInput(fromState, InputCategory.HYPHEN_MINUS), State.
STATE_ERROR);
82 transitionTable.put(new TransitionInput(fromState, InputCategory.EQUAL), State.STATE_ERROR)
;
83 transitionTable.put(new TransitionInput(fromState, InputCategory.EXCLAMATION_MARK), State.
STATE_ERROR);
84 transitionTable.put(new TransitionInput(fromState, InputCategory.DOT), State.STATE_ERROR);
85 transitionTable.put(new TransitionInput(fromState, InputCategory.SINGLE_QUOTE), State.
STATE_ERROR);
86 transitionTable.put(new TransitionInput(fromState, InputCategory.PUNCT), State.STATE_ERROR)
;
87 transitionTable.put(new TransitionInput(fromState, InputCategory.OTHER_PRINTABLE), State.
STATE_ERROR);
88 transitionTable.put(new TransitionInput(fromState, InputCategory.LINE_FEED), State.
STATE_ERROR);
89 /***** end transition table row 5 *****/
90 /***** transition table row 6 *****/
91 fromState = State.STATE_5;
92 transitionTable.put(new TransitionInput(fromState, InputCategory.LETTER), State.STATE_ERROR
);
93 transitionTable.put(new TransitionInput(fromState, InputCategory.DIGIT), State.STATE_ERROR)
;
94 transitionTable.put(new TransitionInput(fromState, InputCategory.UNDERSCORE), State.
STATE_ERROR);
95 transitionTable.put(new TransitionInput(fromState, InputCategory.SLASH_DIVIDE), State.
STATE_6);
96 transitionTable.put(new TransitionInput(fromState, InputCategory.ASTERISK), State.STATE_7);
97 transitionTable.put(new TransitionInput(fromState, InputCategory.LESS_THAN), State.
STATE_ERROR);
98 transitionTable.put(new TransitionInput(fromState, InputCategory.GREATER_THAN), State.
STATE_ERROR);
99 transitionTable.put(new TransitionInput(fromState, InputCategory.PLUS), State.STATE_ERROR);
100 transitionTable.put(new TransitionInput(fromState, InputCategory.HYPHEN_MINUS), State.
STATE_ERROR);
101 transitionTable.put(new TransitionInput(fromState, InputCategory.EQUAL), State.STATE_ERROR)
;
102 transitionTable.put(new TransitionInput(fromState, InputCategory.EXCLAMATION_MARK), State.
STATE_ERROR);
103 transitionTable.put(new TransitionInput(fromState, InputCategory.DOT), State.STATE_ERROR);
104 transitionTable.put(new TransitionInput(fromState, InputCategory.SINGLE_QUOTE), State.
STATE_ERROR);
105 transitionTable.put(new TransitionInput(fromState, InputCategory.PUNCT), State.STATE_ERROR)
;
106 transitionTable.put(new TransitionInput(fromState, InputCategory.OTHER_PRINTABLE), State.
STATE_ERROR);
107 transitionTable.put(new TransitionInput(fromState, InputCategory.LINE_FEED), State.
STATE_ERROR);
108 /***** end transition table row 6 *****/
109 /***** transition table row 7 *****/
110 fromState = State.STATE_6;
111 transitionTable.put(new TransitionInput(fromState, InputCategory.LETTER), fromState);
112 transitionTable.put(new TransitionInput(fromState, InputCategory.DIGIT), fromState);
113 transitionTable.put(new TransitionInput(fromState, InputCategory.UNDERSCORE), fromState);
114 transitionTable.put(new TransitionInput(fromState, InputCategory.SLASH_DIVIDE), fromState);
115 transitionTable.put(new TransitionInput(fromState, InputCategory.ASTERISK), fromState);
116 transitionTable.put(new TransitionInput(fromState, InputCategory.LESS_THAN), fromState);
117 transitionTable.put(new TransitionInput(fromState, InputCategory.GREATER_THAN), fromState);
118 transitionTable.put(new TransitionInput(fromState, InputCategory.PLUS), fromState);
119 transitionTable.put(new TransitionInput(fromState, InputCategory.HYPHEN_MINUS), fromState);
120 transitionTable.put(new TransitionInput(fromState, InputCategory.EQUAL), fromState);
121 transitionTable.put(new TransitionInput(fromState, InputCategory.EXCLAMATION_MARK),
fromState);
122 transitionTable.put(new TransitionInput(fromState, InputCategory.DOT), fromState);
123 transitionTable.put(new TransitionInput(fromState, InputCategory.SINGLE_QUOTE), fromState);
124 transitionTable.put(new TransitionInput(fromState, InputCategory.PUNCT), fromState);

```

```

125 transitionTable.put(new TransitionInput(fromState, InputCategory.OTHER_PRINTABLE),
126 fromState);
127 transitionTable.put(new TransitionInput(fromState, InputCategory.LINE_FEED), State.
128 STATE_ERROR);
129 /***** end transition table row 7 *****/
130 /***** transition table row 8 *****/
131 fromState = State.STATE_7;
132 transitionTable.put(new TransitionInput(fromState, InputCategory.LETTER), fromState);
133 transitionTable.put(new TransitionInput(fromState, InputCategory.DIGIT), fromState);
134 transitionTable.put(new TransitionInput(fromState, InputCategory.UNDERSCORE), fromState);
135 transitionTable.put(new TransitionInput(fromState, InputCategory.SLASH_DIVIDE), fromState);
136 transitionTable.put(new TransitionInput(fromState, InputCategory.ASTERISK), State.STATE_8);
137 transitionTable.put(new TransitionInput(fromState, InputCategory.LESS_THAN), fromState);
138 transitionTable.put(new TransitionInput(fromState, InputCategory.GREATER_THAN), fromState);
139 transitionTable.put(new TransitionInput(fromState, InputCategory.PLUS), fromState);
140 transitionTable.put(new TransitionInput(fromState, InputCategory.HYPHEN_MINUS), fromState);
141 transitionTable.put(new TransitionInput(fromState, InputCategory.EQUAL), fromState);
142 transitionTable.put(new TransitionInput(fromState, InputCategory.EXCLAMATION_MARK),
143 fromState);
144 transitionTable.put(new TransitionInput(fromState, InputCategory.DOT), fromState);
145 transitionTable.put(new TransitionInput(fromState, InputCategory.SINGLE_QUOTE), fromState);
146 transitionTable.put(new TransitionInput(fromState, InputCategory.PUNCT), fromState);
147 transitionTable.put(new TransitionInput(fromState, InputCategory.OTHER_PRINTABLE),
148 fromState);
149 transitionTable.put(new TransitionInput(fromState, InputCategory.LINE_FEED), fromState);
150 /***** end transition table row 8 *****/
151 /***** transition table row 9 *****/
152 fromState = State.STATE_8;
153 transitionTable.put(new TransitionInput(fromState, InputCategory.LETTER), State.STATE_7);
154 transitionTable.put(new TransitionInput(fromState, InputCategory.DIGIT), State.STATE_7);
155 transitionTable.put(new TransitionInput(fromState, InputCategory.UNDERSCORE), State.STATE_7);
156 transitionTable.put(new TransitionInput(fromState, InputCategory.SLASH_DIVIDE), State.
157 STATE_9);
158 transitionTable.put(new TransitionInput(fromState, InputCategory.ASTERISK), State.STATE_7);
159 transitionTable.put(new TransitionInput(fromState, InputCategory.LESS_THAN), State.STATE_7);
160 ;
161 transitionTable.put(new TransitionInput(fromState, InputCategory.GREATER_THAN), State.
162 STATE_7);
163 transitionTable.put(new TransitionInput(fromState, InputCategory.PLUS), State.STATE_7);
164 transitionTable.put(new TransitionInput(fromState, InputCategory.HYPHEN_MINUS), State.
165 STATE_7);
166 transitionTable.put(new TransitionInput(fromState, InputCategory.EQUAL), State.STATE_7);
167 transitionTable.put(new TransitionInput(fromState, InputCategory.EXCLAMATION_MARK), State.
168 STATE_7);
169 transitionTable.put(new TransitionInput(fromState, InputCategory.DOT), State.STATE_7);
170 transitionTable.put(new TransitionInput(fromState, InputCategory.SINGLE_QUOTE), State.
171 STATE_7);
172 transitionTable.put(new TransitionInput(fromState, InputCategory.PUNCT), State.STATE_7);
173 transitionTable.put(new TransitionInput(fromState, InputCategory.OTHER_PRINTABLE), State.
174 STATE_7);
175 transitionTable.put(new TransitionInput(fromState, InputCategory.LINE_FEED), State.STATE_7)
176 ;
177 /***** end transition table row 9 *****/
178 /***** transition table row 10 *****/
179 fromState = State.STATE_9;
180 for (InputCategory input : InputCategory.values()) {
181     transitionTable.put(new TransitionInput(fromState, input), State.STATE_ERROR);
182 }
183 /***** end transition table row 10 *****/
184 /***** transition table row 11 *****/
185 fromState = State.STATE_10;
186 for (InputCategory input : InputCategory.values()) {
187     transitionTable.put(new TransitionInput(fromState, input), State.STATE_ERROR);
188 }
189 /***** end transition table row 11 *****/
190 /***** transition table row 12 *****/
191 fromState = State.STATE_11;
192 transitionTable.put(new TransitionInput(fromState, InputCategory.LETTER), State.STATE_ERROR);
193 ;
194 transitionTable.put(new TransitionInput(fromState, InputCategory.DIGIT), State.STATE_11);
195 transitionTable.put(new TransitionInput(fromState, InputCategory.UNDERSCORE), State.

```



```

STATE_ERROR);
184 transitionTable.put(new TransitionInput(fromState, InputCategory.SLASH_DIVIDE), State .
STATE_ERROR);
185 transitionTable.put(new TransitionInput(fromState, InputCategory.ASTERISK), State .
STATE_ERROR);
186 transitionTable.put(new TransitionInput(fromState, InputCategory.LESS_THAN), State .
STATE_ERROR);
187 transitionTable.put(new TransitionInput(fromState, InputCategory.GREATER_THAN), State .
STATE_ERROR);
188 transitionTable.put(new TransitionInput(fromState, InputCategory.PLUS), State.STATE_ERROR);
189 transitionTable.put(new TransitionInput(fromState, InputCategory.HYPHEN_MINUS), State .
STATE_ERROR);
190 transitionTable.put(new TransitionInput(fromState, InputCategory.EQUAL), State.STATE_ERROR)
;
191 transitionTable.put(new TransitionInput(fromState, InputCategory.EXCLAMATION_MARK), State .
STATE_ERROR);
192 transitionTable.put(new TransitionInput(fromState, InputCategory.DOT), State.STATE_12);
193 transitionTable.put(new TransitionInput(fromState, InputCategory.SINGLE_QUOTE), State .
STATE_ERROR);
194 transitionTable.put(new TransitionInput(fromState, InputCategory.PUNCT), State.STATE_ERROR)
;
195 transitionTable.put(new TransitionInput(fromState, InputCategory.OTHER_PRINTABLE), State .
STATE_ERROR);
196 transitionTable.put(new TransitionInput(fromState, InputCategory.LINE_FEED), State .
STATE_ERROR);
197 /***** end transition table row 12 ****/
198
199 /***** transition table row 13 ****/
200 fromState = State.STATE_12;
201 transitionTable.put(new TransitionInput(fromState, InputCategory.LETTER), State.STATE_ERROR
);
202 transitionTable.put(new TransitionInput(fromState, InputCategory.DIGIT), State.STATE_13);
203 transitionTable.put(new TransitionInput(fromState, InputCategory.UNDERSCORE), State .
STATE_ERROR);
204 transitionTable.put(new TransitionInput(fromState, InputCategory.SLASH_DIVIDE), State .
STATE_ERROR);
205 transitionTable.put(new TransitionInput(fromState, InputCategory.ASTERISK), State .
STATE_ERROR);
206 transitionTable.put(new TransitionInput(fromState, InputCategory.LESS_THAN), State .
STATE_ERROR);
207 transitionTable.put(new TransitionInput(fromState, InputCategory.GREATER_THAN), State .
STATE_ERROR);
208 transitionTable.put(new TransitionInput(fromState, InputCategory.PLUS), State.STATE_ERROR);
209 transitionTable.put(new TransitionInput(fromState, InputCategory.HYPHEN_MINUS), State .
STATE_ERROR);
210 transitionTable.put(new TransitionInput(fromState, InputCategory.EQUAL), State.STATE_ERROR)
;
211 transitionTable.put(new TransitionInput(fromState, InputCategory.EXCLAMATION_MARK), State .
STATE_ERROR);
212 transitionTable.put(new TransitionInput(fromState, InputCategory.DOT), State.STATE_ERROR);
213 transitionTable.put(new TransitionInput(fromState, InputCategory.SINGLE_QUOTE), State .
STATE_ERROR);
214 transitionTable.put(new TransitionInput(fromState, InputCategory.PUNCT), State.STATE_ERROR)
;
215 transitionTable.put(new TransitionInput(fromState, InputCategory.OTHER_PRINTABLE), State .
STATE_ERROR);
216 transitionTable.put(new TransitionInput(fromState, InputCategory.LINE_FEED), State .
STATE_ERROR);
217 /***** end transition table row 13 ****/
218
219 /***** transition table row 14 ****/
220 fromState = State.STATE_13;
221 transitionTable.put(new TransitionInput(fromState, InputCategory.LETTER), State.STATE_ERROR
);
222 transitionTable.put(new TransitionInput(fromState, InputCategory.DIGIT), fromState);
223 transitionTable.put(new TransitionInput(fromState, InputCategory.UNDERSCORE), State .
STATE_ERROR);
224 transitionTable.put(new TransitionInput(fromState, InputCategory.SLASH_DIVIDE), State .
STATE_ERROR);
225 transitionTable.put(new TransitionInput(fromState, InputCategory.ASTERISK), State .
STATE_ERROR);
226 transitionTable.put(new TransitionInput(fromState, InputCategory.LESS_THAN), State .
STATE_ERROR);

```

```

227 transitionTable.put(new TransitionInput(fromState, InputCategory.GREATER_THAN), State.
STATE_ERROR);
228 transitionTable.put(new TransitionInput(fromState, InputCategory.PLUS), State.STATE_ERROR);
229 transitionTable.put(new TransitionInput(fromState, InputCategory.HYPHEN_MINUS), State.
STATE_ERROR);
230 transitionTable.put(new TransitionInput(fromState, InputCategory.EQUAL), State.STATE_ERROR)
;
231 transitionTable.put(new TransitionInput(fromState, InputCategory.EXCLAMATION_MARK), State.
STATE_ERROR);
232 transitionTable.put(new TransitionInput(fromState, InputCategory.DOT), State.STATE_ERROR);
233 transitionTable.put(new TransitionInput(fromState, InputCategory.SINGLE_QUOTE), State.
STATE_ERROR);
234 transitionTable.put(new TransitionInput(fromState, InputCategory.PUNCT), State.STATE_ERROR)
;
235 transitionTable.put(new TransitionInput(fromState, InputCategory.OTHER_PRINTABLE), State.
STATE_ERROR);
236 transitionTable.put(new TransitionInput(fromState, InputCategory.LINE_FEED), State.
STATE_ERROR);
237 /***** end transition table row 14 *****/
238 /***** transition table row 15 *****/
239 fromState = State.STATE_14;
240 for (InputCategory input : InputCategory.values()) {
241     transitionTable.put(new TransitionInput(fromState, input), State.STATE_ERROR);
242 }
243 /***** end transition table row 15 *****/
244 /***** transition table row 16 *****/
245 fromState = State.STATE_15;
246 transitionTable.put(new TransitionInput(fromState, InputCategory.LETTER), State.STATE_ERROR
);
247 transitionTable.put(new TransitionInput(fromState, InputCategory.DIGIT), State.STATE_ERROR)
;
248 transitionTable.put(new TransitionInput(fromState, InputCategory.UNDERSCORE), State.
STATE_ERROR);
249 transitionTable.put(new TransitionInput(fromState, InputCategory.SLASH_DIVIDE), State.
STATE_ERROR);
250 transitionTable.put(new TransitionInput(fromState, InputCategory.ASTERISK), State.
STATE_ERROR);
251 transitionTable.put(new TransitionInput(fromState, InputCategory.LESS_THAN), State.
STATE_ERROR);
252 transitionTable.put(new TransitionInput(fromState, InputCategory.GREATER_THAN), State.
STATE_18);
253 transitionTable.put(new TransitionInput(fromState, InputCategory.PLUS), State.STATE_ERROR);
254 transitionTable.put(new TransitionInput(fromState, InputCategory.HYPHEN_MINUS), State.
STATE_ERROR);
255 transitionTable.put(new TransitionInput(fromState, InputCategory.EQUAL), State.STATE_ERROR)
;
256 transitionTable.put(new TransitionInput(fromState, InputCategory.EXCLAMATION_MARK), State.
STATE_ERROR);
257 transitionTable.put(new TransitionInput(fromState, InputCategory.DOT), State.STATE_ERROR);
258 transitionTable.put(new TransitionInput(fromState, InputCategory.SINGLE_QUOTE), State.
STATE_ERROR);
259 transitionTable.put(new TransitionInput(fromState, InputCategory.PUNCT), State.STATE_ERROR)
;
260 transitionTable.put(new TransitionInput(fromState, InputCategory.OTHER_PRINTABLE), State.
STATE_ERROR);
261 transitionTable.put(new TransitionInput(fromState, InputCategory.LINE_FEED), State.
STATE_ERROR);
262 /***** end transition table row 16 *****/
263 /***** transition table row 17 *****/
264 fromState = State.STATE_16;
265 transitionTable.put(new TransitionInput(fromState, InputCategory.LETTER), State.STATE_ERROR
);
266 transitionTable.put(new TransitionInput(fromState, InputCategory.DIGIT), State.STATE_ERROR)
;
267 transitionTable.put(new TransitionInput(fromState, InputCategory.UNDERSCORE), State.
STATE_ERROR);
268 transitionTable.put(new TransitionInput(fromState, InputCategory.SLASH_DIVIDE), State.
STATE_ERROR);
269 transitionTable.put(new TransitionInput(fromState, InputCategory.ASTERISK), State.
STATE_ERROR);
270 transitionTable.put(new TransitionInput(fromState, InputCategory.LESS_THAN), State.
STATE_ERROR);
271 transitionTable.put(new TransitionInput(fromState, InputCategory.GREATER_THAN), State.

```

```

STATE_ERROR);
272 transitionTable.put(new TransitionInput(fromState, InputCategory.PLUS), State.STATE_ERROR);
273 transitionTable.put(new TransitionInput(fromState, InputCategory.HYPHEN_MINUS), State.
STATE_ERROR);
274 transitionTable.put(new TransitionInput(fromState, InputCategory.EQUAL), State.STATE_19);
275 transitionTable.put(new TransitionInput(fromState, InputCategory.EXCLAMATION_MARK), State.
STATE_ERROR);
276 transitionTable.put(new TransitionInput(fromState, InputCategory.DOT), State.STATE_ERROR);
277 transitionTable.put(new TransitionInput(fromState, InputCategory.SINGLE_QUOTE), State.
STATE_ERROR);
278 transitionTable.put(new TransitionInput(fromState, InputCategory.PUNCT), State.STATE_ERROR)
;
279 transitionTable.put(new TransitionInput(fromState, InputCategory.OTHER_PRINTABLE), State.
STATE_ERROR);
280 transitionTable.put(new TransitionInput(fromState, InputCategory.LINE_FEED), State.
STATE_ERROR);
281 /***** end transition table row 17 *****/
282 /***** transition table row 18 *****/
283 fromState = State.STATE_17;
284 transitionTable.put(new TransitionInput(fromState, InputCategory.LETTER), State.STATE_ERROR
);
285 transitionTable.put(new TransitionInput(fromState, InputCategory.DIGIT), State.STATE_ERROR)
;
286 transitionTable.put(new TransitionInput(fromState, InputCategory.UNDERSCORE), State.
STATE_ERROR);
287 transitionTable.put(new TransitionInput(fromState, InputCategory.SLASH_DIVIDE), State.
STATE_ERROR);
288 transitionTable.put(new TransitionInput(fromState, InputCategory.ASTERISK), State.
STATE_ERROR);
289 transitionTable.put(new TransitionInput(fromState, InputCategory.LESS_THAN), State.
STATE_ERROR);
290 transitionTable.put(new TransitionInput(fromState, InputCategory.GREATER_THAN), State.
STATE_ERROR);
291 transitionTable.put(new TransitionInput(fromState, InputCategory.PLUS), State.STATE_ERROR);
292 transitionTable.put(new TransitionInput(fromState, InputCategory.HYPHEN_MINUS), State.
STATE_ERROR);
293 transitionTable.put(new TransitionInput(fromState, InputCategory.EQUAL), State.STATE_19);
294 transitionTable.put(new TransitionInput(fromState, InputCategory.EXCLAMATION_MARK), State.
STATE_ERROR);
295 transitionTable.put(new TransitionInput(fromState, InputCategory.DOT), State.STATE_ERROR);
296 transitionTable.put(new TransitionInput(fromState, InputCategory.SINGLE_QUOTE), State.
STATE_ERROR);
297 transitionTable.put(new TransitionInput(fromState, InputCategory.PUNCT), State.STATE_ERROR)
;
298 transitionTable.put(new TransitionInput(fromState, InputCategory.OTHER_PRINTABLE), State.
STATE_ERROR);
299 transitionTable.put(new TransitionInput(fromState, InputCategory.LINE_FEED), State.
STATE_ERROR);
300 /***** end transition table row 18 *****/
301 /***** transition table row 19 *****/
302 fromState = State.STATE_18;
303 for (InputCategory input : InputCategory.values()) {
304     transitionTable.put(new TransitionInput(fromState, input), State.STATE_ERROR);
305 }
306 /***** end transition table row 19 *****/
307 /***** transition table row 20 *****/
308 fromState = State.STATE_19;
309 for (InputCategory input : InputCategory.values()) {
310     transitionTable.put(new TransitionInput(fromState, input), State.STATE_ERROR);
311 }
312 /***** end transition table row 20 *****/
313 return transitionTable;
314 }
315 }

```

Listing 6.4: Implementation of transition table

```

1 package tinylanglexer;
2 /**
3  * Infinite amount of possible lexemes are
4  * categorised into a finite amount of groups.
5  * Therefore a lexeme is a string with an

```

```

6  * identified meaning in the language.
7  * @author andre
8  */
9  public enum TokenType {
10     /**
11      * Syntax Error Handler
12      * Identifies lexemes which are not accepted by TinyLang's grammar.
13      */
14     INVALID ,
15     /**
16      * Control Flow Keyword
17      * Value(s) : if
18      */
19     TOK_IF ,
20     /**
21      * Control Flow Keyword
22      * Value(s) : else
23      */
24     TOK_ELSE ,
25     /**
26      * Iteration Keyword
27      * Value(s) : for
28      */
29     TOK_FOR ,
30     /**
31      * Iteration Keyword
32      * Value(s) : while
33      */
34     TOK_WHILE ,
35     /**
36      * Structure Keyword
37      * Value(s) : fn
38      */
39     TOK_FN ,
40     /**
41      * Returning Keyword
42      * Value(s) : fn
43      */
44     TOK_RETURN ,
45     /**
46      * Data Type Keyword
47      * Value(s) : int
48      */
49     TOK_INT_TYPE ,
50     /**
51      * Data Type Keyword
52      * Value(s) : float
53      */
54     TOK_FLOAT_TYPE ,
55     /**
56      * Data Type Keyword
57      * Value(s) : not
58      */
59     TOK_NOT ,
60     /**
61      * Data Type Keyword
62      * Value(s) : bool
63      */
64     TOK_BOOL_TYPE ,
65     /**
66      * Data Type Keyword
67      * Value(s) : char
68      */
69     TOK_CHAR_TYPE ,
70     /**
71      * Keyword Token
72      * Value(s) : let
73      * identify variable declaration
74      */
75     TOK_LET ,
76     /**
77      * Keyword Token
78      * Value(s) : ->

```

```

79  * specify return type of a function
80  */
81  TOK_RIGHT_ARROW ,
82
83  /**
84   * Keyword Token
85   * Value(s) : print
86   * identify print statement
87   */
88  TOK_PRINT ,
89  /**
90   * Punctuation
91   * Value(s) : (
92   */
93  TOK_LEFT_ROUND_BRACKET ,
94  /**
95   * Punctuation
96   * Value(s) : )
97   */
98  TOK_RIGHT_ROUND_BRACKET ,
99  /**
100   * Punctuation
101   * Value(s) : {
102   */
103  TOK_LEFT_CURLY_BRACKET ,
104  /**
105   * Punctuation
106   * Value(s) : }
107   */
108  TOK_RIGHT_CURLY_BRACKET ,
109  /**
110   * Punctuation
111   * Value(s) : ,
112   */
113  TOK_COMMA ,
114  /**
115   * Punctuation
116   * Value(s) :
117   */
118  TOK_DOT ,
119  /**
120   * Punctuation
121   * Value(s) : :
122   */
123  TOK_COLON ,
124  /**
125   * Punctuation
126   * Value(s) : ;
127   */
128  TOK_SEMICOLON ,
129
130  /**
131   * Punctuation
132   * Value(s) : ;
133   */
134  TOK_MULTIPLICATIVE_OP ,
135  /**
136   *
137   * Value(s) : (
138   */
139  TOK_ADDITIVE_OP ,
140  /**
141   * Operation Token Name
142   * Value(s) : =
143   */
144  TOK_EQUAL ,
145  /**
146   * Operation Token Name
147   * Value(s) : '<' '>' '==' '!=' '<=' '>='
148   */
149  TOK_RELATIONAL_OP ,
150  /**
151   * Token Name

```

```

152  */
153  TOK_IDENTIFIER ,
154  /**
155   * Token Name
156   * Value(s) : true , false
157   */
158  TOK_BOOL_LITERAL ,
159  /**
160   * Token Name
161   */
162  TOK_INT_LITERAL ,
163  /**
164   * Token Name
165   */
166  TOK_FLOAT_LITERAL ,
167  /**
168   * Token Name
169   */
170  TOK_CHAR_LITERAL ,
171
172  /**
173   * Special Token
174   */
175  TOK_SKIP ,
176  /**
177   * Special Token
178   * Used to identify end of program
179   */
180  TOK_EOF
181 }

```

Listing 6.5: Token Types

```

1 package tinylanglexer;
2 public class Token {
3     //attribute associated with token type
4     private String lexeme;
5     //tokenType
6     private TokenType tokenType;
7     //line number where lexeme resided
8     private int lineNumber;
9     public Token(TokenType tokenType, String lexeme) {
10         this.tokenType = tokenType;
11         this.lexeme = lexeme;
12     }
13     // setters and getters
14     public String getLexeme() {
15         return lexeme;
16     }
17     public void setLexeme(String lexeme) {
18         this.lexeme = lexeme;
19     }
20     public TokenType getTokenType() {
21         return this.tokenType;
22     }
23     public void setTokenType(TokenType tokenType) {
24         this.tokenType = tokenType;
25     }
26     public int getLineNumber() {
27         return lineNumber;
28     }
29     public void setLineNumber(int lineNumber) {
30         this.lineNumber = lineNumber;
31     }
32 }

```

Listing 6.6: Token=(TokenType,(Lexeme,LineNumber))

```

1 package tinylanglexer;
2 import java.util.ArrayList;
3 import java.util.Map;

```

```

4 import java.util.Stack;
5
6 /**
7  * Class for lexer implementation of TinyLang
8  * extends TransitionTable
9  * @author andre
10 */
11 public class TinyLangLexer extends TransitionTable{
12     // Obtain transition table from class TransitionTable
13     private Map<TransitionInput,State> transitionTable = buildTransitionTable();
14     // List of tokens
15     private ArrayList<Token> tokens = new ArrayList<>();
16     // Scanning -> traverse program char by char -> keep track of current char
17     private int currentCharIndex = 0;
18     // Keep track of line number
19     private int lineNumber = 0;
20     /**
21     * Constructor for class TinyLangLexer
22     * @param TinyLangProgram
23     * @throws Exception
24     */
25     public TinyLangLexer(String tinyLangProgram) {
26         // build transition table
27         this.buildTransitionTable();
28         // program is empty -> only one EOF token
29         if(tinyLangProgram.length()==0)
30             this.tokens.add(new Token(TokenType.TOK_EOF, ""));
31         // if program is not empty -> loop until current char is not at the end of file
32         while(currentCharIndex<tinyLangProgram.length()) {
33             // obtain next token
34             Token nextToken = getNextToken(tinyLangProgram);
35             // set line number
36             nextToken.setLineNumber(getLineNumber(tinyLangProgram));
37             // if token is not of type TOK_SKIP add to list of tokens
38             if(nextToken.getTokenType()!= TokenType.TOK_SKIP) {
39                 this.tokens.add(nextToken);
40             }
41         }
42     }
43     /**
44     * Table Driven Analysis Algorithm -> Cooper & Torczon Engineer a Compiler.
45     * @param TinyLangProgram
46     */
47     private Token getNextToken(String tinyLangProgram) {
48         /* start initialisation stage */
49         // Set current state to start state
50         State state = State.STARTING_STATE;
51         // Current lexeme
52         String lexeme = "";
53         // Create Stack Of States
54         Stack<State> stack = new Stack<State>();
55         // Push BAD state to the stack
56         stack.add(State.STATE_BAD);
57         /* end initialisation stage */
58         while(tinyLangProgram.charAt(currentCharIndex) == 0xa || tinyLangProgram.charAt(
59             currentCharIndex)==0x20 || tinyLangProgram.charAt(currentCharIndex)==0x09) {
60             if(tinyLangProgram.charAt(currentCharIndex) == 0xa)
61                 lineNumber++;
62             // increment char number
63             this.currentCharIndex++;
64             // detect EOF
65             if(currentCharIndex==tinyLangProgram.length())
66                 return new Token(TokenType.TOK_EOF, "");
67         }
68         InputCategory inputCategory;
69         char currentChar;
70         while(state!=State.STATE_ERROR &&currentCharIndex<tinyLangProgram.length()) {
71             // obtain current CHAR
72             currentChar = tinyLangProgram.charAt(currentCharIndex);
73             // char to lexeme
74             lexeme+=currentChar;
75             // if state is accepting clear stack
76             if (state.getStateType()==StateType.ACCEPTING) {

```

```

76     stack.clear();
77 }
78 // push state to stack
79 stack.add(state);
80 if (isLetter(currentChar)) {
81     inputCategory = InputCategory.LETTER;
82 }
83     else if (isDigit(currentChar)) {
84         inputCategory = InputCategory.DIGIT;
85     }
86     else if (isUnderscore(currentChar)) {
87         inputCategory = InputCategory.UNDERSCORE;
88     }
89     else if (isSlashDivide(currentChar)) {
90         inputCategory = InputCategory.SLASH_DIVIDE;
91     }
92     else if (isAsterisk(currentChar)) {
93         inputCategory = InputCategory.ASTERISK;
94     }
95     else if (isLessThan(currentChar)) {
96         inputCategory = InputCategory.LESS_THAN;
97     }
98     else if (isGreaterThan(currentChar)) {
99         inputCategory = InputCategory.GREATER_THAN;
100     }
101     else if (isPlus(currentChar)) {
102         inputCategory = InputCategory.PLUS;
103     }
104     else if (isHyphenMinus(currentChar)) {
105         inputCategory = InputCategory.HYPHEN_MINUS;
106     }
107     else if (isEqual(currentChar)) {
108         inputCategory = InputCategory.EQUAL;
109     }
110     else if (isExclamationMark(currentChar)) {
111         inputCategory = InputCategory.EXCLAMATION_MARK;
112     }
113     else if (isDot(currentChar)) {
114         inputCategory = InputCategory.DOT;
115     }
116     else if (isSingleQuote(currentChar)) {
117         inputCategory = InputCategory.SINGLE_QUOTE;
118     }
119     else if (isPunctuation(currentChar)) {
120         inputCategory = InputCategory.PUNCT;
121     }
122     else if (isOtherPrintable(currentChar)) {
123         inputCategory = InputCategory.OTHER_PRINTABLE;
124     }
125     else if (isLineFeed(currentChar)) {
126         inputCategory = InputCategory.LINE_FEED;
127     }
128     else {
129         throw new java.lang.RuntimeException("char "+currentChar+" in line " +lineNumber
130         +" not recognised by TinyLang's grammar");
131     }
132     // get next transition as per transition table
133
134     state = deltaFunction(state, inputCategory);
135     // move to next char
136     currentCharIndex++;
137
138
139 }
140 /*      begin rollback loop      */
141 while(state!=State.STATE_BAD && state.getStateType()!=StateType.ACCEPTING) {
142     // pop state
143     state=stack.pop();
144     //truncate string
145     lexeme = (lexeme==null || lexeme.length()==0)? null:(lexeme.substring(0, lexeme.length
146     () -1));
147     // move char index one step backwards

```



```

147         currentCharIndex--;
148     }
149     if (state.getTokenType(lexeme) == TokenType.INVALID)
150         throw new java.lang.RuntimeException(tokens.get(tokens.size() - 1).getLexeme() +
        tinyLangProgram.charAt(currentCharIndex + 1) + " in line " + lineNumber + " not recognised by
        TinyLang's grammar");
151     else
152         return new Token(state.getTokenType(lexeme), lexeme);
153     // end lineNumber
154 }
155 // predicate functions to check input category
156 private boolean isLetter(char input) {
157     return ( (0x41 <= input && input <= 0x5a) || (0x61 <= input && input <= 0x7a) );
158 }
159 private boolean isDigit(char input) {
160     return (0x30 <= input && input <= 0x39);
161 }
162 private boolean isUnderscore(char input) {
163     return (input == 0x5f);
164 }
165 private boolean isSlashDivide(char input) {
166     return (input == 0x2f);
167 }
168 private boolean isAsterisk(char input) {
169     return (input == 0x2a);
170 }
171 private boolean isLessThan(char input) {
172     return (input == 0x3c);
173 }
174 private boolean isGreaterThan(char input) {
175     return (input == 0x3e);
176 }
177 private boolean isPlus(char input) {
178     return (input == 0x2b);
179 }
180 private boolean isHyphenMinus(char input) {
181     return (input == 0x2d);
182 }
183 private boolean isEqual(char input) {
184     return (input == 0x3d);
185 }
186 private boolean isExclamationMark(char input) {
187     return (input == 0x21);
188 }
189 private boolean isDot(char input) {
190     return (input == 0x2e);
191 }
192 private boolean isSingleQuote(char input) {
193     return (input == 0x27);
194 }
195 private boolean isPunctuation(char input) {
196     return (input == 0x28 || input == 0x29 || input == 0x2c || input == 0x3a || input == 0x3b || input == 0
        x7b || input == 0x7d);
197 }
198 private boolean isOtherPrintable(char input) {
199     return ( 0x20 <= input && input <= 0x7e && !isLetter(input) && !isDigit(input) && !
        isUnderscore(input) &&
200         !isSlashDivide(input) && !isAsterisk(input) && !isLessThan(input) && !isGreaterThan(
        input)
201         && !isPlus(input) && !isHyphenMinus(input) && !isEqual(input) && !isExclamationMark(
        input)
202         && !isDot(input) && !isSingleQuote(input) && !isPunctuation(input);
203 }
204 private boolean isLineFeed(char input) {
205     lineNumber++;
206     return (input == 0x0a);
207 }
208 private State deltaFunction(State state, InputCategory inputCategory) {
209     return transitionTable.get(new TransitionInput(state, inputCategory));
210 }
211 // setter and getter methods
212 public ArrayList<Token> getTokens() {

```

```

214     return tokens;
215 }
216
217
218 private int getLineNumber(String tinyLangProgram) {
219     lineNumber = 1;
220     for(int i=0;i<currentCharIndex;i++)
221         if (tinyLangProgram.charAt(i)==0x0a)
222             lineNumber++;
223     return lineNumber;
224 }
225
226 }
227
228 }

```

Listing 6.7: Table Driven Lexer

6.2 | Parser

```

1 package tinylangparser;
2 import java.util.LinkedList;
3 import java.util.List;
4 public class TinyLangAst {
5     /* node */
6     private TinyLangAstNodes associatedNodeType;
7     private String associatedNodeValue = "";
8     private int lineNumber = 0;
9     TinyLangAst parent;
10    List<TinyLangAst> children;
11
12    public TinyLangAst(TinyLangAstNodes associatedNodeType,int lineNumber) {
13        this.associatedNodeType = associatedNodeType;
14        this.lineNumber=lineNumber;
15        this.children = new LinkedList<TinyLangAst>();
16    }
17    public TinyLangAst(TinyLangAstNodes associatedNodeType, String associatedNodeValue,int
18        lineNumber) {
19        this.associatedNodeType = associatedNodeType;
20        this.associatedNodeValue = associatedNodeValue;
21        this.lineNumber=lineNumber;
22        this.children = new LinkedList<TinyLangAst>();
23    }
24    //add root of a subtree to abstract syntax tree
25    public void addSubtree(TinyLangAst subTree) {
26        this.children.add(subTree);
27    }
28    public TinyLangAst addChild(TinyLangAstNodes associatedNodeType,int lineNumber) {
29        TinyLangAst childNode = new TinyLangAst(associatedNodeType,lineNumber);
30        childNode.parent = this;
31        this.children.add(childNode);
32        return childNode;
33    }
34    public TinyLangAst addChild(TinyLangAstNodes associatedNodeType,String associatedNodeValue
35        ,int lineNumber) {
36        TinyLangAst childNode = new TinyLangAst(associatedNodeType,associatedNodeValue,
37        lineNumber);
38        childNode.parent = this;
39        this.children.add(childNode);
40        return childNode;
41    }
42    // setters and getters
43    public TinyLangAstNodes getAssociatedNodeType(){
44        return associatedNodeType;
45    };
46    public String getAssociatedNodeValue(){

```

```

45     return associatedNodeValue;
46 }
47 // get children
48 public List<TinyLangAst> getChildren(){
49     return children;
50 }
51 public void setLineNumber(int lineNumber) {
52     this.lineNumber=lineNumber;
53 }
54 public int getLineNumber() {
55     return lineNumber;
56 }
57 }

```

Listing 6.8: general structures of an AST (class *TinyLangAst*)

```

1 package tinylangparser;
2 public enum TinyLangAstNodes {
3     //program node
4     TINY_LANG_PROGRAM_NODE,
5     //statement nodes
6     AST_VARIABLE_DECLARATION_NODE,
7     AST_ASSIGNMENT_NODE,
8     AST_PRINT_STATEMENT_NODE,
9     AST_IF_STATEMENT_NODE,
10    AST_FOR_STATEMENT_NODE,
11    AST_WHILE_STATEMENT_NODE,
12    AST_RETURN_STATEMENT_NODE,
13    AST_FUNCTION_DECLARATION_NODE,
14    AST_BLOCK_NODE,
15    AST_ELSE_BLOCK_NODE,
16    //expression nodes
17    AST_BINARY_OPERATOR_NODE,
18    AST_UNARY_OPERATOR_NODE,
19    AST_FUNCTION_CALL_NODE,
20    //literal nodes
21    AST_BOOLEAN_LITERAL_NODE,
22    AST_INTEGER_LITERAL_NODE,
23    AST_FLOAT_LITERAL_NODE,
24    AST_CHAR_LITERAL_NODE,
25    //parameters nodes
26    AST_ACTUAL_PARAMETERS_NODE,
27    AST_FORMAL_PARAMETERS_NODE,
28    AST_FORMAL_PARAMETER_NODE,
29    //type node
30    AST_TYPE_NODE,
31    //expression nodes
32    //expression nodes leaves
33    AST_IDENTIFIER_NODE
34 }

```

Listing 6.9: types associated with each node/subtree (enum *TinyLangAstNodes*)

```

1 package tinylangparser;
2 import java.util.ArrayList;
3 import tinylanglexer.TinyLangLexer;
4 import tinylanglexer.Token;
5 import tinylanglexer.TokenType;
6 public class TinyLangParser {
7     // root of ast -> describes ast capturing all the program
8     private TinyLangAst tinyLangProgramAbstractSyntaxTree;
9     // list of tokens
10    private ArrayList<Token> tokens;
11    // current token index
12    private int currentTokenIndex = 0;
13    // method for obtaining current token
14    private Token getCurrentToken(){
15        return tokens.get(currentTokenIndex);
16    }

```

```

17 // method for obtaining next token
18 private Token getNextToken(){
19     currentTokenIndex++;
20     return getCurrentToken();
21 }
22 // method for obtaining previous token
23 private Token getPrevToken(){
24     currentTokenIndex--;
25     return getCurrentToken();
26 }
27 /**
28  * Constructor for TinyLangParserClass
29  * @param tinyLangLexer
30  */
31 public TinyLangParser(TinyLangLexer tinyLangLexer) {
32     tokens = tinyLangLexer.getTokens();
33     tinyLangProgramAbstractSyntaxTree = parseTinyLangProgram();
34 }
35 /**
36  * Parse whole TinyLangProgram using recursive descent
37  * to call other sub parsers until TOK_EOF is reached.
38  */
39 private TinyLangAst parseTinyLangProgram() {
40     //program tree capturing whole syntax of tiny lang program
41     TinyLangAst programTree = new TinyLangAst(TinyLangAstNodes.TINY_LANG_PROGRAM_NODE,
42         getCurrentToken().getLineNumber());
43     // traverse until current token reach EOF i.e. no more tokens to process
44     while(getCurrentToken().getTokenType() != TokenType.TOK_EOF) {
45         // parse statement one by one
46         programTree.addSubtree(parseStatement());
47         // get next token
48         getNextToken();
49     }
50     return programTree;
51 }
52 /**
53  * Parse a statement
54  * <Statement> -> <VariableDecl> ';'
55  * <Statement> -> <Assignment> ';'
56  * <Statement> -> <PrintStatement> ';'
57  * <Statement> -> <IfStatement> ';'
58  * <Statement> -> <ForStatement> ';'
59  * <Statement> -> <WhileStatement> ';'
60  * <Statement> -> <RtrnStatement> ';'
61  * <Statement> -> <FunctionDecl>
62  * <Statement> -> <Block>
63  * described by an LL(1) grammar i.e. decide immediately which grammar rule to use with
64  * TokenTypes
65  * TOK_LET, TOK_IDENTIFIER, TOK_PRINT, TOK_WHILE, TOK_RETURN, TOK_FN, TOK_LEFT_CURLY otherwise
66  * undefined.
67  * @param lookAhead
68  * @param parent
69  */
70 public TinyLangAst parseStatement() {
71     TinyLangAst statementTree;
72     switch(getCurrentToken().getTokenType()){
73         // if lookAhead = TOK_LET Statement leads to variable declaration
74         case TOK_LET:
75             //parse variable declaration
76             statementTree = parseVariableDeclaration();
77             //get next token
78             getNextToken();
79             //expecting ;
80             if (getCurrentToken().getTokenType() != TokenType.TOK_SEMICOLON)
81                 throw new java.lang.RuntimeException("expected semicolon ; , in line " +
82                     getCurrentToken().getLineNumber());
83             return statementTree;
84         case TOK_IDENTIFIER:
85             //parse assignment
86             statementTree = parseAssignment();
87             //get next token
88             getNextToken();

```

```

86 //expecting ;
87 if (getCurrentToken().getTokenType() != TokenType.TOK_SEMICOLON)
88 //not as expected
89 throw new java.lang.RuntimeException("expected semicolon ; , in line " +
getCurrentToken().getLineNumber());
90 return statementTree;
91 case TOK_PRINT:
92 statementTree = parsePrintStatement();
93 //expecting ;
94 if (getNextToken().getTokenType() != TokenType.TOK_SEMICOLON)
95 //not as expected
96 throw new java.lang.RuntimeException("expected semicolon ; , in line " +
getCurrentToken().getLineNumber());
97 return statementTree;
98 case TOK_IF:
99 return parseIfStatement();
100 case TOK_FOR:
101 return parseForStatement();
102 case TOK_WHILE:
103 return parseWhileStatement();
104 case TOK_RETURN:
105 statementTree = parseReturnStatement();
106 //get next token
107 getNextToken();
108 //expecting ;
109 if (getCurrentToken().getTokenType() != TokenType.TOK_SEMICOLON)
110 //not as expected
111 throw new java.lang.RuntimeException("expected semicolon ; , in line " +
getCurrentToken().getLineNumber());
112 return statementTree;
113 case TOK_FN:
114 return parseFunctionDeclaration();
115 case TOK_RIGHT_CURLY_BRACKET:
116 return parseBlock();
117 default:
118 throw new java.lang.RuntimeException(" in line "+getCurrentToken().getLineNumber()
119 +". No statement begins with "+getCurrentToken().getLexeme());
120 }
121 }
122 //parse variable declaration
123 private TinyLangAst parseVariableDeclaration() {
124 //create variable declaration syntax tree
125 TinyLangAst variableDeclarationTree = new TinyLangAst(TinyLangAstNodes.
AST_VARIABLE_DECLARATION_NODE, getCurrentToken().getLineNumber());
126 //expect token let
127 if (getCurrentToken().getTokenType() != TokenType.TOK_LET)
128 throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+ " in
line " + getCurrentToken().getLineNumber());
129
130 //expect that next token is identifier
131 Token identifier = getNextToken();
132 //check if identifier
133 if (getCurrentToken().getTokenType() != TokenType.TOK_IDENTIFIER)
134 throw new java.lang.RuntimeException(getCurrentToken().getLexeme()+ " in line " +
getCurrentToken().getLineNumber()+ " is not a valid variable name");
135 //get next token
136 getNextToken();
137 //expect :
138 if (getCurrentToken().getTokenType() != TokenType.TOK_COLON)
139 throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+ " in
line " + getCurrentToken().getLineNumber());
140 //get next token
141 getNextToken();
142 //expect type tree
143 variableDeclarationTree.addSubtree(parseType());
144
145 //add identifier
146 variableDeclarationTree.addChild(TinyLangAstNodes.AST_IDENTIFIER_NODE, identifier.getLexeme()
, identifier.getLineNumber());
147 //get next token
148 getNextToken();
149 //expect =
150 if (getCurrentToken().getTokenType() != TokenType.TOK_EQUAL)

```

```

151     throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+ " in
152     line " + getCurrentToken().getLineNumber());
153     //get next token
154     getNextToken();
155     variableDeclarationTree.addSubtree(parseExpression());
156     return variableDeclarationTree;
157 }
158 //parse assignment
159 private TinyLangAst parseAssignment() {
160     //create assignment syntax tree
161     TinyLangAst assignmentTree = new TinyLangAst(TinyLangAstNodes.AST_ASSIGNMENT_NODE,
162     getCurrentToken().getLineNumber());
163     //expect identifier
164     if (getCurrentToken().getTokenType() != TokenType.TOK_IDENTIFIER) {
165         throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+ " in
166         line " + getCurrentToken().getLineNumber());
167     }
168     //add identifier node
169     assignmentTree.addChild(TinyLangAstNodes.AST_IDENTIFIER_NODE, getCurrentToken().getLexeme(),
170     getCurrentToken().getLineNumber());
171     //get next token
172     getNextToken();
173     //expect equal
174     if (getCurrentToken().getTokenType() != TokenType.TOK_EQUAL)
175         throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+ " in
176         line " + getCurrentToken().getLineNumber());
177     //get next token
178     getNextToken();
179     //expect expression
180     assignmentTree.addSubtree(parseExpression());
181     return assignmentTree;
182 }
183 //parse print statement
184 private TinyLangAst parsePrintStatement() {
185     //create assignment syntax tree
186     TinyLangAst printStatementTree = new TinyLangAst(TinyLangAstNodes.AST_PRINT_STATEMENT_NODE,
187     getCurrentToken().getLineNumber());
188     //expect print keyword
189     if (getCurrentToken().getTokenType() != TokenType.TOK_PRINT)
190         throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+ " in
191         line " + getCurrentToken().getLineNumber());
192     //get next token
193     getNextToken();
194     //expect expression
195     printStatementTree.addSubtree(parseExpression());
196     return printStatementTree;
197 }
198 //parse if statement
199 private TinyLangAst parseIfStatement() {
200     //create if statement syntax tree
201     TinyLangAst ifStatementTree = new TinyLangAst(TinyLangAstNodes.AST_IF_STATEMENT_NODE,
202     getCurrentToken().getLineNumber());
203     //expect if keyword
204     if (getCurrentToken().getTokenType() != TokenType.TOK_IF)
205         throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+ " in
206         line " + getCurrentToken().getLineNumber());
207     //get next token
208     getNextToken();
209     //expect (
210     if (getCurrentToken().getTokenType() != TokenType.TOK_LEFT_ROUND_BRACKET)
211         throw new java.lang.RuntimeException("expected left round bracket,( , in line "+
212         getCurrentToken().getLineNumber());
213     //get next token
214     getNextToken();
215     //add expression subtree to if statement tree
216     ifStatementTree.addSubtree(parseExpression());
217     //get next token
218     getNextToken();
219     //expected )
220     if (getCurrentToken().getTokenType() != TokenType.TOK_RIGHT_ROUND_BRACKET)
221         throw new java.lang.RuntimeException("expected left round bracket,( , in line "+
222         getCurrentToken().getLineNumber());
223     //get next token

```

```

213 getNextToken();
214 //parse block
215 ifStatementTree.addSubtree(parseBlock());
216 //getNextToken()
217 getNextToken();
218 //if we have else condition
219 if (getCurrentToken().getTokenType() == TokenType.TOK_ELSE) {
220     //get next token
221     getNextToken();
222     //get else block
223     ifStatementTree.addSubtree(parseElseBlock());
224 }
225 else
226     //get previous token
227     getPrevToken();
228 //return if statement tree
229 return ifStatementTree;
230 }
231 //parse for statement
232 private TinyLangAst parseForStatement() {
233     //create block syntax tree
234     TinyLangAst forStatementTree = new TinyLangAst(TinyLangAstNodes.AST_FOR_STATEMENT_NODE,
235         getCurrentToken().getLineNumber());
236     //expect for keywordF
237     if (getCurrentToken().getTokenType() != TokenType.TOK_FOR)
238         //not as expected
239         throw new java.lang.RuntimeException("unexpected " + getCurrentToken().getLexeme() + " in
240         line " + getCurrentToken().getLineNumber());
241     //get next token
242     getNextToken();
243     //expect (
244     if (getCurrentToken().getTokenType() != TokenType.TOK_LEFT_ROUND_BRACKET)
245         //not as expected
246         throw new java.lang.RuntimeException("expected left round bracket,( , in line " +
247         getCurrentToken().getLineNumber());
248     //get next token
249     getNextToken();
250     //expect semicolon or variable declaration
251     if (getCurrentToken().getTokenType() != TokenType.TOK_SEMICOLON)
252     {
253         //expect variable declaration
254         forStatementTree.addSubtree(parseVariableDeclaration());
255         //consume variable declaration
256         getNextToken();
257     }
258     //expect ;
259     if (getCurrentToken().getTokenType() != TokenType.TOK_SEMICOLON)
260         throw new java.lang.RuntimeException("expected semicolon,; , in line " + getCurrentToken
261         ().getLineNumber());
262     //get next token
263     getNextToken();
264     //expect expression
265     forStatementTree.addSubtree(parseExpression());
266     //expect ;
267     if (getNextToken().getTokenType() != TokenType.TOK_SEMICOLON)
268         throw new java.lang.RuntimeException("expected semicolon,; , in line " + getCurrentToken
269         ().getLineNumber());
270     //expect right round bracket or assignment
271     if (getNextToken().getTokenType() != TokenType.TOK_RIGHT_ROUND_BRACKET)
272     {
273         //expect variable declaration
274         forStatementTree.addSubtree(parseAssignment());
275         //consume variable declaration
276         getNextToken();
277     }
278     if (getCurrentToken().getTokenType() != TokenType.TOK_RIGHT_ROUND_BRACKET)
279         throw new java.lang.RuntimeException("expected right round bracket,) , in line " +
280         getCurrentToken().getLineNumber());
281     //get next token
282     getNextToken();
283     //expect block
284     forStatementTree.addSubtree(parseBlock());
285     //return for statement tree

```

```

280     return forStatementTree;
281 }
282 //parse while statement
283 private TinyLangAst parseWhileStatement() {
284     //create while statement syntax tree syntax tree
285     TinyLangAst whileStatementTree = new TinyLangAst(TinyLangAstNodes.AST_WHILE_STATEMENT_NODE
286     ,getCurrentToken().getLineNumber());
287     //expect while keyword
288     if(getCurrentToken().getTokenType() != TokenType.TOK_WHILE)
289         //not as expected
290         throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+ " in
291         line " + getCurrentToken().getLineNumber());
292     //get next token
293     getNextToken();
294     //expect (
295     if(getCurrentToken().getTokenType() != TokenType.TOK_LEFT_ROUND_BRACKET)
296         //not as expected
297         throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+ " in
298         line " + getCurrentToken().getLineNumber());
299     //get next token
300     getNextToken();
301     //expect expression
302     whileStatementTree.addSubtree(parseExpression());
303     //get next token
304     getNextToken();
305     //expect )
306     if(getCurrentToken().getTokenType() != TokenType.TOK_RIGHT_ROUND_BRACKET)
307         //not as expected
308         throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+ " in
309         line " + getCurrentToken().getLineNumber());
310     //get next token
311     getNextToken();
312     //expect block
313     whileStatementTree.addSubtree(parseBlock());
314     //return syntax tree
315     return whileStatementTree;
316 }
317 //parse return statement
318 private TinyLangAst parseReturnStatement() {
319     //create while statement syntax tree syntax tree
320     TinyLangAst returnStatementTree = new TinyLangAst(TinyLangAstNodes.
321     AST_RETURN_STATEMENT_NODE,getCurrentToken().getLineNumber());
322     //expect return keyword
323     if(getCurrentToken().getTokenType() != TokenType.TOK_RETURN)
324         //not as expected
325         throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+ " in
326         line " + getCurrentToken().getLineNumber());
327     //get next token
328     getNextToken();
329     //expect expression
330     returnStatementTree.addSubtree(parseExpression());
331     //return syntax tree
332     return returnStatementTree;
333 }
334 //parse function declaration
335 private TinyLangAst parseFunctionDeclaration() {
336     //create function declaration syntax tree syntax tree
337     TinyLangAst functionDeclarationTree = new TinyLangAst(TinyLangAstNodes.
338     AST_FUNCTION_DECLARATION_NODE,getCurrentToken().getLineNumber());
339     //expect return keyword
340     if(getCurrentToken().getTokenType() != TokenType.TOK_FN)
341         //not as expected
342         throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+ " in
343         line " + getCurrentToken().getLineNumber());
344     //get next token
345     getNextToken();
346     //expect expression
347     Token identifier;
348     if(getCurrentToken().getTokenType() == TokenType.TOK_IDENTIFIER)
349         identifier = getCurrentToken();
350     else
351         //not valid function name

```



```

345     throw new java.lang.RuntimeException(getCurrentToken().getLexeme() + " in line " +
346     getCurrentToken().getLineNumber() + " not a valid function name");
347     //get next token
348     getNextToken();
349     //expect (
350     if (getCurrentToken().getTokenType() != TokenType.TOK_LEFT_ROUND_BRACKET)
351     //not as expected
352     throw new java.lang.RuntimeException("unexpected " + getCurrentToken().getLexeme() + " in
353     line " + getCurrentToken().getLineNumber());
354     //get next token
355     getNextToken();
356     //expect 0 or more formal parameters
357     TinyLangAst formalParamsSubtree;
358     //if not right round bracket -> we have parameters
359     if (getCurrentToken().getTokenType() != TokenType.TOK_RIGHT_ROUND_BRACKET){
360     formalParamsSubtree = parseFormalParams();
361     //get next token (expected round bracket in next token)
362     getNextToken();
363     }
364     else
365     //add parameter node
366     formalParamsSubtree = new TinyLangAst(TinyLangAstNodes.AST_FORMAL_PARAMETERS_NODE,
367     getCurrentToken().getLineNumber());
368     //expect )
369     if (getCurrentToken().getTokenType() != TokenType.TOK_RIGHT_ROUND_BRACKET)
370     //not as expected
371     throw new java.lang.RuntimeException("expected right round bracket, ) , "+" in line " +
372     getCurrentToken().getLineNumber());
373     //get next token
374     getNextToken();
375     //expect ->
376     if (getCurrentToken().getTokenType() != TokenType.TOK_RIGHT_ARROW)
377     //not as expected
378     throw new java.lang.RuntimeException("expected right arrow, -> , in line " + getCurrentToken()
379     ().getLineNumber());
380     //get next token
381     getNextToken();
382     //parse type
383     TinyLangAst typeSubtree = parseType();
384     //get next token
385     getNextToken();
386     //parse block
387     TinyLangAst blockSubtree = parseBlock();
388     //add type subtree to function declaration tree
389     functionDeclarationTree.addSubtree(typeSubtree);
390     //add identifier node to function declaration tree
391     functionDeclarationTree.addChild(TinyLangAstNodes.AST_IDENTIFIER_NODE, identifier.getLexeme()
392     (), identifier.getLineNumber());
393     //add formal parameters subtree to function declaration tree
394     functionDeclarationTree.addSubtree(formalParamsSubtree);
395     //add block subtree to function declaration tree
396     functionDeclarationTree.addSubtree(blockSubtree);
397     //return function declaration tree
398     return functionDeclarationTree;
399     }
400     //parse block
401     private TinyLangAst parseBlock() {
402     //create block syntax tree
403     TinyLangAst blockTree = new TinyLangAst(TinyLangAstNodes.AST_BLOCK_NODE, getCurrentToken().
404     getLineNumber());
405     //expected {
406     //set line number
407     blockTree.setLineNumber(getCurrentToken().getLineNumber());
408     if (getCurrentToken().getTokenType() != TokenType.TOK_LEFT_CURLY_BRACKET)
409     //not as expected
410     throw new java.lang.RuntimeException("expected { in line " + getCurrentToken().
411     getLineNumber());
412     // get next token
413     getNextToken();

```

```

410 // we may have one or more statements
411 // block ends using }
412 while (getCurrentToken().getTokenType() != TokenType.TOK_RIGHT_CURLY_BRACKET &&
413        getCurrentToken().getTokenType() != TokenType.TOK_EOF) {
414     // parse statement one by one
415
416     blockTree.addSubtree(parseStatement());
417     // get next token
418     getNextToken();
419 }
420 //expected }
421 if (getCurrentToken().getTokenType() != TokenType.TOK_RIGHT_CURLY_BRACKET)
422     //not as expected
423     throw new java.lang.RuntimeException("expected } in line " + getCurrentToken().
424        getLineNumber());
425 //return block tree
426 return blockTree;
427 }
428 //parse else block
429 private TinyLangAst parseElseBlock() {
430     //create block syntax tree
431     TinyLangAst elseBlockTree = new TinyLangAst(TinyLangAstNodes.AST_ELSE_BLOCK_NODE,
432        getCurrentToken().getLineNumber());
433     //expected {
434     if (getCurrentToken().getTokenType() != TokenType.TOK_LEFT_CURLY_BRACKET)
435         //not as expected
436         throw new java.lang.RuntimeException("expected { in line " + getCurrentToken().
437            getLineNumber());
438     // get next token
439     getNextToken();
440     // we may have one or more statements
441     // block ends using }
442     while (getCurrentToken().getTokenType() != TokenType.TOK_RIGHT_CURLY_BRACKET &&
443            getCurrentToken().getTokenType() != TokenType.TOK_EOF) {
444         // parse statement one by one
445         elseBlockTree.addSubtree(parseStatement());
446         // get next token
447         getNextToken();
448     }
449     //expected }
450     if (getCurrentToken().getTokenType() != TokenType.TOK_RIGHT_CURLY_BRACKET)
451         //not as expected
452         throw new java.lang.RuntimeException("expected } in line " + getCurrentToken().
453            getLineNumber());
454     //return else block syntax tree
455     return elseBlockTree;
456 }
457 //parse type
458 private TinyLangAst parseType() {
459     // add node
460     switch (getCurrentToken().getTokenType()) {
461         case TOK_BOOL_TYPE:
462             return new TinyLangAst(TinyLangAstNodes.AST_TYPE_NODE, Type.BOOL.toString(),
463                getCurrentToken().getLineNumber());
464         case TOK_INT_TYPE:
465             return new TinyLangAst(TinyLangAstNodes.AST_TYPE_NODE, Type.INTEGER.toString(),
466                getCurrentToken().getLineNumber());
467         case TOK_FLOAT_TYPE:
468             return new TinyLangAst(TinyLangAstNodes.AST_TYPE_NODE, Type.FLOAT.toString(),
469                getCurrentToken().getLineNumber());
470         case TOK_CHAR_TYPE:
471             return new TinyLangAst(TinyLangAstNodes.AST_TYPE_NODE, Type.CHAR.toString(),
472                getCurrentToken().getLineNumber());
473         default:
474             throw new java.lang.RuntimeException(getCurrentToken().getLexeme() + " in line " +
475                getCurrentToken().getLineNumber() + " is not a valid type");
476     }
477 }
478 //parse expression
479 private TinyLangAst parseExpression() {
480     //parse simple expression
481     TinyLangAst left = parseSimpleExpression();

```

```

474 //get next token
475 getNextToken();
476 //expecting 0 or more expressions separated by a relational operator
477 if (getCurrentToken().getTokenType() == TokenType.TOK_RELATIONAL_OP) {
478     //create a binary expression tree with root node containing current binary operator
479     TinyLangAst binaryExpressionTree = new TinyLangAst(TinyLangAstNodes.
AST_BINARY_OPERATOR_NODE, getCurrentToken().getLexeme(), getCurrentToken().getLineNumber());
480 //add left operand of the binary operator
481 binaryExpressionTree.addSubtree(left);
482 //move to next token
483 getNextToken();
484 //add right operand
485 binaryExpressionTree.addSubtree(parseExpression());
486 return binaryExpressionTree;
487 }
488 getPrevToken();
489 //case of no relational operator
490 return left;
491 }
492 //parse simple expression
493 private TinyLangAst parseSimpleExpression() {
494     //parse simple expression
495     TinyLangAst left = parseTerm();
496     //get next token
497     getNextToken();
498     //expecting 0 or more expressions separated by a relational operator
499     if (getCurrentToken().getTokenType() == TokenType.TOK_ADDITIVE_OP) {
500         //create a binary expression tree with root node containing current binary operator
501         TinyLangAst binaryExpressionTree = new TinyLangAst(TinyLangAstNodes.
AST_BINARY_OPERATOR_NODE, getCurrentToken().getLexeme(), getCurrentToken().getLineNumber());
502 //add left operand of the binary operator
503 binaryExpressionTree.addSubtree(left);
504 //move to next token
505 getNextToken();
506 //add right operand
507 binaryExpressionTree.addSubtree(parseSimpleExpression());
508 return binaryExpressionTree;
509 }
510 getPrevToken();
511 //case of no relational operator
512 return left;
513 }
514 //parse term
515 private TinyLangAst parseTerm() {
516     //parse factor
517     TinyLangAst left = parseFactor();
518     //get next token
519     getNextToken();
520     //expecting 0 or more expressions separated by a multiplicative operator
521     if (getCurrentToken().getTokenType() == TokenType.TOK_MULTIPLICATIVE_OP) {
522         //create a binary expression tree with root node containing current binary operator
523         TinyLangAst binaryExpressionTree = new TinyLangAst(TinyLangAstNodes.
AST_BINARY_OPERATOR_NODE, getCurrentToken().getLexeme(), getCurrentToken().getLineNumber());
524 //add left operand of the binary operator
525 binaryExpressionTree.addSubtree(left);
526 //move to next token
527 getNextToken();
528 //add right operand
529 binaryExpressionTree.addSubtree(parseTerm());
530 return binaryExpressionTree;
531 }
532 getPrevToken();
533 //case of no relational operator
534 return left;
535 }
536 //parse term
537 private TinyLangAst parseFactor() {
538     switch (getCurrentToken().getTokenType()) {
539         //literals
540         case TOK_BOOL_LITERAL:
541             return new TinyLangAst(TinyLangAstNodes.AST_BOOLEAN_LITERAL_NODE, getCurrentToken().
getLexeme(), getCurrentToken().getLineNumber());
542         case TOK_INT_LITERAL:

```

```

543     return new TinyLangAst(TinyLangAstNodes.AST_INTEGER_LITERAL_NODE, getCurrentToken().
getLexeme(), getCurrentToken().getLineNumber());
544 case TOK_FLOAT_LITERAL:
545     return new TinyLangAst(TinyLangAstNodes.AST_FLOAT_LITERAL_NODE, getCurrentToken().
getLexeme(), getCurrentToken().getLineNumber());
546 case TOK_CHAR_LITERAL:
547     return new TinyLangAst(TinyLangAstNodes.AST_CHAR_LITERAL_NODE, getCurrentToken().
getLexeme(), getCurrentToken().getLineNumber());
548 //identifier or function call
549 case TOK_IDENTIFIER:
550     getNextToken();
551     if (getCurrentToken().getTokenType() == TokenType.TOK_LEFT_ROUND_BRACKET) {
552         getPrevToken();
553         return parseFunctionCall();
554     }
555     else {
556         getPrevToken();
557         return new TinyLangAst(TinyLangAstNodes.AST_IDENTIFIER_NODE, getCurrentToken().
getLexeme(), getCurrentToken().getLineNumber());
558     }
559 case TOK_LEFT_ROUND_BRACKET:
560     return parseSubExpression();
561 case TOK_ADDITIVE_OP:
562 case TOK_NOT:
563     return parseUnary();
564 default:
565     throw new java.lang.RuntimeException("unexpected " + getCurrentToken().getLexeme() + " in
line " + getCurrentToken().getLineNumber());
566 }
567 }
568 private TinyLangAst parseSubExpression() {
569     //expect left round bracket
570     if (getCurrentToken().getTokenType() != TokenType.TOK_LEFT_ROUND_BRACKET)
571         throw new java.lang.RuntimeException("unexpected " + getCurrentToken().getLexeme() + " in
line " + getCurrentToken().getLineNumber());
572     //get next token
573     getNextToken();
574     //expect expression
575     TinyLangAst expressionTree = parseExpression();
576
577     //get next token
578     getNextToken();
579     //expect right round bracket
580     if (getCurrentToken().getTokenType() != TokenType.TOK_RIGHT_ROUND_BRACKET)
581         throw new java.lang.RuntimeException("unexpected " + getCurrentToken().getLexeme() + " in
line " + getCurrentToken().getLineNumber());
582     //return expression tree
583     return expressionTree;
584 }
585 private TinyLangAst parseUnary() {
586
587     //expect not or additive
588     if (getCurrentToken().getTokenType() != TokenType.TOK_ADDITIVE_OP && getCurrentToken().
getTokenType() != TokenType.TOK_NOT)
589         throw new java.lang.RuntimeException("unexpected " + getCurrentToken().getLexeme() + " in
line " + getCurrentToken().getLineNumber());
590     //create unary tree with unary operator
591     TinyLangAst unaryTree = new TinyLangAst(TinyLangAstNodes.AST_UNARY_OPERATOR_NODE,
getCurrentToken().getLexeme(), getCurrentToken().getLineNumber());
592     //get next token
593     getNextToken();
594     //expect expression
595     unaryTree.addSubtree(parseExpression());
596
597     return unaryTree;
598 }
599 private TinyLangAst parseFunctionCall() {
600     TinyLangAst functionCallTree = new TinyLangAst(TinyLangAstNodes.AST_FUNCTION_CALL_NODE,
getCurrentToken().getLineNumber());
601     if (getCurrentToken().getTokenType() != TokenType.TOK_IDENTIFIER)
602         throw new java.lang.RuntimeException("unexpected " + getCurrentToken().getLexeme() + " in
line " + getCurrentToken().getLineNumber());
603     //add identifier node

```

```

604 functionCallTree.addChild(TinyLangAstNodes.AST_IDENTIFIER_NODE, getCurrentToken().getLexeme
    (), getCurrentToken().getLineNumber());
605 getNextToken();
606 if (getCurrentToken().getTokenType() != TokenType.TOK_LEFT_ROUND_BRACKET)
607     //not as expected
608     throw new java.lang.RuntimeException("unexpected " + getCurrentToken().getLexeme() + " in
        line " + getCurrentToken().getLineNumber());
609
610 getNextToken();
611 //if not right round bracket -> we have parameters
612 if (getCurrentToken().getTokenType() != TokenType.TOK_RIGHT_ROUND_BRACKET) {
613     functionCallTree.addSubtree(parseActualParams());
614     //get next token (expected round bracket in next token)
615     getNextToken();
616 }
617 else
618     //add parameter node
619     functionCallTree.addChild(TinyLangAstNodes.AST_ACTUAL_PARAMETERS_NODE, getCurrentToken().
        getLineNumber());
620
621
622 if (getCurrentToken().getTokenType() != TokenType.TOK_RIGHT_ROUND_BRACKET)
623     //not as expected
624     throw new java.lang.RuntimeException("expected right round bracket,)" + getCurrentToken
        ().getLexeme() + " in line " + getCurrentToken().getLineNumber());
625 return functionCallTree;
626 }
627 TinyLangAst parseActualParams() {
628     //parse expression
629     TinyLangAst actualParamsTree = new TinyLangAst(TinyLangAstNodes.AST_ACTUAL_PARAMETERS_NODE
        , getCurrentToken().getLineNumber());
630     //add expression tree
631     actualParamsTree.addSubtree(parseExpression());
632     //get next token
633     getNextToken();
634
635     while (getCurrentToken().getTokenType() == TokenType.TOK_COMMA && getCurrentToken().
        getTokenType() != TokenType.TOK_EOF )
636     {
637         //get next token
638         getNextToken();
639         actualParamsTree.addSubtree(parseExpression());
640         //get next token
641         getNextToken();
642     }
643     getPrevToken();
644     return actualParamsTree;
645 }
646 //parse formal parameters
647 TinyLangAst parseFormalParams() {
648     //parse expression
649     TinyLangAst formalParamsTree = new TinyLangAst(TinyLangAstNodes.AST_FORMAL_PARAMETERS_NODE
        , getCurrentToken().getLineNumber());
650     //add formal param tree
651     formalParamsTree.addSubtree(parseFormalParam());
652     //get next token
653     getNextToken();
654
655     while (getCurrentToken().getTokenType() == TokenType.TOK_COMMA)
656     {
657         //get next token
658         getNextToken();
659         formalParamsTree.addSubtree(parseFormalParam());
660         //get next token
661         getNextToken();
662     }
663     getPrevToken();
664
665     return formalParamsTree;
666 }
667 //parse formal parameter
668 TinyLangAst parseFormalParam() {
669     //parse expression

```

```

670 TinyLangAst formalParamTree = new TinyLangAst(TinyLangAstNodes.AST_FORMAL_PARAMETER_NODE,
        getCurrentToken().getLineNumber());
671 //expect identifier
672 if(getCurrentToken().getTokenType() != TokenType.TOK_IDENTIFIER)
673     throw new java.lang.RuntimeException(getCurrentToken().getLexeme()+" in line "+
        getCurrentToken().getLineNumber()+" is not a valid parameter name");
674 //add identifier node
675 Token identifier = getCurrentToken();
676 //get next token
677 getNextToken();
678 // expect :
679 if(getCurrentToken().getTokenType() != TokenType.TOK_COLON)
680     //not as expected
681     throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+" in
        line "+getCurrentToken().getLineNumber());
682 //get next token
683 getNextToken();
684 formalParamTree.addSubtree(parseType());
685 formalParamTree.addChild(TinyLangAstNodes.AST_IDENTIFIER_NODE, identifier.getLexeme(),
        identifier.getLineNumber());
686 return formalParamTree;
687 }
688 public TinyLangAst getTinyLangAbstraxSyntaxTree() {
689     return tinyLangProgramAbstractSyntaxTree;
690 }
691 }

```

Listing 6.10: Implementation of recursive descent parser

6.3 | XML generation

```

1 package tinylangvisitor;
2 import tinylangparser.TinyLangAst;
3 import tinylangparser.TinyLangAstNodes;
4 public class XmlGeneration implements Visitor {
5     private String xmlRepresentation = "";
6     private int indentation = 0;
7     private String getCurrentIndentationLevel() {
8         String indentation = "";
9         for(int i=0;i<this.indentation;i++)
10             //add indentation
11             indentation+="    "; //(char)0x09;
12     return indentation;
13 }
14 //method which runs statement type visit method based on node type
15 public void visitStatement(TinyLangAst tinyLangAst) {
16     switch(tinyLangAst.getAssociatedNodeType()) {
17     case AST_VARIABLE_DECLARATION_NODE:
18         visitVariableDeclarationNode(tinyLangAst);
19         break;
20     case AST_ASSIGNMENT_NODE:
21         visitAssignmentNode(tinyLangAst);
22         break;
23     case AST_PRINT_STATEMENT_NODE:
24         visitPrintStatementNode(tinyLangAst);
25         break;
26     case AST_IF_STATEMENT_NODE:
27         visitIfStatementNode(tinyLangAst);
28         break;
29     case AST_FOR_STATEMENT_NODE:
30         visitForStatementNode(tinyLangAst);
31         break;
32     case AST_WHILE_STATEMENT_NODE:
33         visitWhileStatementNode(tinyLangAst);
34         break;
35     case AST_RETURN_STATEMENT_NODE:
36         visitReturnStatementNode(tinyLangAst);

```

```

37     break;
38     case AST_FUNCTION_DECLARATION_NODE:
39         visitFunctionDeclarationNode(tinyLangAst);
40         break;
41     case AST_BLOCK_NODE:
42         visitBlockNode(tinyLangAst);
43         break;
44     default:
45         throw new java.lang.RuntimeException("Unrecognised statement of type "+tinyLangAst.
46             getAssociatedNodeType());
47     }
48     private void visitExpression(TinyLangAst tinyLangAst){
49         switch(tinyLangAst.getAssociatedNodeType()) {
50             case AST_BINARY_OPERATOR_NODE:
51                 visitBinaryOperatorNode(tinyLangAst);
52                 break;
53             case AST_UNARY_OPERATOR_NODE:
54                 visitUnaryOperatorNode(tinyLangAst);
55                 break;
56             case AST_BOOLEAN_LITERAL_NODE:
57                 visitBooleanLiteralNode(tinyLangAst);
58                 break;
59             case AST_INTEGER_LITERAL_NODE:
60                 visitIntegerLiteralNode(tinyLangAst);
61                 break;
62             case AST_FLOAT_LITERAL_NODE:
63                 visitFloatLiteralNode(tinyLangAst);
64                 break;
65             case AST_CHAR_LITERAL_NODE:
66                 visitCharLiteralNode(tinyLangAst);
67                 break;
68             case AST_IDENTIFIER_NODE:
69                 visitIdentifierNode(tinyLangAst);
70                 break;
71             case AST_FUNCTION_CALL_NODE:
72                 visitFunctionCallNode(tinyLangAst);
73                 break;
74             default:
75                 throw new java.lang.RuntimeException("Unrecognised expression node of type "+tinyLangAst
76                     .getAssociatedNodeType());
77         }
78     }
79     public XmlGeneration(TinyLangAst tinyLangAst) {
80         visitTinyLangProgram(tinyLangAst);
81     }
82     @Override
83     public void visitTinyLangProgram(TinyLangAst tinyLangAst) {
84         xmlRepresentation+=getCurrentIndentationLevel()+"<TinyLangProgram>\n";
85         //indent
86         indentation++;
87         for(TinyLangAst child : tinyLangAst.getChildren())
88             visitStatement(child);
89         //unindent
90         indentation--;
91         xmlRepresentation+=getCurrentIndentationLevel()+"<\\TinyLangProgram>\n";
92     }
93     @Override
94     public void visitVariableDeclarationNode(TinyLangAst tinyLangAst) {
95         xmlRepresentation+=getCurrentIndentationLevel()+"<variable declaration>\n";
96         //indent
97         indentation++;
98         //visit children
99         //add function identifier
100        xmlRepresentation+=getCurrentIndentationLevel()+"<id type=\""+tinyLangAst.getChildren().
101            get(0).getAssociatedNodeValue()+"\">"+tinyLangAst.getChildren().get(1).
102            getAssociatedNodeValue()+"<\\id>\n";
103        //add expression tag
104        visitExpression(tinyLangAst.getChildren().get(2));
105        //unindent
106        indentation--;

```

```

106     xmlRepresentation+=getCurrentIndentationLevel()+"<\\variable declaration >\n";
107 }
108
109 @Override
110 public void visitPrintStatementNode(TinyLangAst tinyLangAst) {
111     xmlRepresentation+=getCurrentIndentationLevel()+"<print statement>\n";
112     //indent
113     indentation++;
114     visitExpression(tinyLangAst.getChildren().get(0));
115     //unindent
116     indentation--;
117     xmlRepresentation+=getCurrentIndentationLevel()+"<\\print statement>\n";
118 }
119
120 @Override
121 public void visitIfStatementNode(TinyLangAst tinyLangAst) {
122     xmlRepresentation+=getCurrentIndentationLevel()+"<if statement>\n";
123     //indent
124     indentation++;
125     //expect first child to be expression
126     visitExpression(tinyLangAst.getChildren().get(0));
127     //expect second child to be block
128     visitBlockNode(tinyLangAst.getChildren().get(1));
129     //check if we have else block
130     if(tinyLangAst.getChildren().size()==3)
131         visitBlockNode(tinyLangAst.getChildren().get(2));
132     //unindent
133     indentation--;
134     xmlRepresentation+=getCurrentIndentationLevel()+"<\\if statement>\n";
135 }
136
137 @Override
138 public void visitForStatementNode(TinyLangAst tinyLangAst) {
139     //add for statement tag
140     xmlRepresentation+=getCurrentIndentationLevel()+"<for statement>\n";
141     //indent
142     indentation++;
143     //expect first child is variable declaration or expression
144     if(tinyLangAst.getChildren().get(0).getAssociatedNodeType()==TinyLangAstNodes.
145     AST_VARIABLE_DECLARATION_NODE)
146         visitVariableDeclarationNode(tinyLangAst.getChildren().get(0));
147
148     else
149         visitExpression(tinyLangAst.getChildren().get(0));
150
151     //second child is assignment or block or expression
152     if(tinyLangAst.getChildren().get(1).getAssociatedNodeType()==TinyLangAstNodes.
153     AST_ASSIGNMENT_NODE)
154         visitAssignmentNode(tinyLangAst.getChildren().get(1));
155     else if(tinyLangAst.getChildren().get(1).getAssociatedNodeType()==TinyLangAstNodes.
156     AST_BLOCK_NODE)
157         visitBlockNode(tinyLangAst.getChildren().get(1));
158     else
159         visitExpression(tinyLangAst.getChildren().get(1));
160     //if we have 3 or more children
161     if(tinyLangAst.getChildren().size()>=3 && tinyLangAst.getChildren().get(2).
162     getAssociatedNodeType()==TinyLangAstNodes.AST_ASSIGNMENT_NODE)
163         visitAssignmentNode(tinyLangAst.getChildren().get(2));
164     else if(tinyLangAst.getChildren().size()>=3 && tinyLangAst.getChildren().get(2).
165     getAssociatedNodeType()==TinyLangAstNodes.AST_BLOCK_NODE)
166         visitBlockNode(tinyLangAst.getChildren().get(2));
167     else
168         throw new java.lang.RuntimeException("unexpected node of type "+tinyLangAst.getChildren
169         ().get(2).getAssociatedNodeType());
170     //if we have 4 children
171     if(tinyLangAst.getChildren().size()==4)
172         visitBlockNode(tinyLangAst.getChildren().get(3));
173     indentation--;
174     xmlRepresentation+=getCurrentIndentationLevel()+"<\\VariableDeclaration>\n";
175 }
176
177 @Override
178 public void visitWhileStatementNode(TinyLangAst tinyLangAst) {
179     xmlRepresentation+=getCurrentIndentationLevel()+"<while statement>\n";
180     //indent
181     indentation++;

```



```

173 //expected 2 children expression and nodes
174 if (tinyLangAst.getChildren().size() != 2)
175     throw new java.lang.RuntimeException("while statement node has "+tinyLangAst.getChildren()
176     .size()+" expected 2");
177 if (tinyLangAst.getChildren().get(1).getAssociatedNodeType() != TinyLangAstNodes.
178     AST_BLOCK_NODE)
179     throw new java.lang.RuntimeException("second child of while statement is "+tinyLangAst.
180     getChildren().get(1).getAssociatedNodeType()+" expected AST_BLOCK_NODE");
181 //visit expression and block
182 visitExpression(tinyLangAst.getChildren().get(0));
183 visitBlockNode(tinyLangAst.getChildren().get(1));
184 indentation--;
185 xmlRepresentation+=getCurrentIndentationLevel()+"<\\while statement>\\n";
186 }
187 @Override
188 public void visitReturnStatementNode(TinyLangAst tinyLangAst) {
189     xmlRepresentation+=getCurrentIndentationLevel()+"<return statement>\\n";
190     //indent
191     indentation++;
192     //visit expression
193     visitExpression(tinyLangAst.getChildren().get(0));
194     indentation--;
195     xmlRepresentation+=getCurrentIndentationLevel()+"<\\return statement>\\n";
196 }
197 @Override
198 public void visitFunctionDeclarationNode(TinyLangAst tinyLangAst) {
199     xmlRepresentation+=getCurrentIndentationLevel()+"<function declaration>\\n";
200     //expected 4 children of types identifier ,formal parameters ,type and block
201     //indent
202     indentation++;
203     //add function identifier
204     xmlRepresentation+=getCurrentIndentationLevel()+"<id type=\\\""+tinyLangAst.getChildren().
205     get(0).getAssociatedNodeValue()+"\\\">"+tinyLangAst.getChildren().get(1).
206     getAssociatedNodeValue()+"<\\id>\\n";
207     //add parameters
208     xmlRepresentation+=getCurrentIndentationLevel()+"<parameters>\\n";
209     indentation++;
210     for (TinyLangAst child : tinyLangAst.getChildren().get(2).getChildren()) {
211         xmlRepresentation+=getCurrentIndentationLevel()+"<parameters>\\n";
212         indentation++;
213         xmlRepresentation+=getCurrentIndentationLevel()+"<id type=\\\""+child.getChildren().get
214         (0).getAssociatedNodeValue()+">"+child.getChildren().get(1).getAssociatedNodeValue()+"<\\
215         id>\\n";
216         indentation--;
217         xmlRepresentation+=getCurrentIndentationLevel()+"<\\parameters>\\n";
218     }
219     indentation--;
220     xmlRepresentation+=getCurrentIndentationLevel()+"<\\parameters>\\n";
221
222     visitBlockNode(tinyLangAst.getChildren().get(3));
223     //unindent
224     indentation--;
225     xmlRepresentation+=getCurrentIndentationLevel()+"<\\function declaration>\\n";
226 }
227
228 @Override
229 public void visitFunctionCallNode(TinyLangAst tinyLangAst) {
230     xmlRepresentation+=getCurrentIndentationLevel()+"<function call>\\n";
231     //expected 4 children of types identifier ,formal parameters ,type and block
232     //indent
233     indentation++;
234     //add function identifier
235     visitIdentifierNode(tinyLangAst.getChildren().get(0));
236     //add parameters
237     xmlRepresentation+=getCurrentIndentationLevel()+"<parameters>\\n";
238     indentation++;
239     for (TinyLangAst child : tinyLangAst.getChildren().get(1).getChildren()) {
240         xmlRepresentation+=getCurrentIndentationLevel()+"<actual parameter>\\n";
241         indentation++;
242         visitExpression(child);
243         indentation--;
244         xmlRepresentation+=getCurrentIndentationLevel()+"<\\actual parameter>\\n";

```

```

239     }
240     indentation--;
241     xmlRepresentation+=getCurrentIndentationLevel()+"<\\parameters>\\n";
242
243     //unindent
244     indentation--;
245     xmlRepresentation+=getCurrentIndentationLevel()+"<\\function call>\\n";
246
247 }
248 @Override
249 public void visitBlockNode(TinyLangAst tinyLangAst) {
250     if (tinyLangAst.getAssociatedNodeType()==TinyLangAstNodes.AST_ELSE_BLOCK_NODE)
251         xmlRepresentation+=getCurrentIndentationLevel()+"<else block>\\n";
252     else
253         xmlRepresentation+=getCurrentIndentationLevel()+"<block>\\n";
254     //indent
255     indentation++;
256     //children are statements
257     for (TinyLangAst child: tinyLangAst.getChildren())
258         visitStatement(child);
259     indentation--;
260     if (tinyLangAst.getAssociatedNodeType()==TinyLangAstNodes.AST_ELSE_BLOCK_NODE)
261         xmlRepresentation+=getCurrentIndentationLevel()+"<\\else block>\\n";
262     else
263         xmlRepresentation+=getCurrentIndentationLevel()+"<\\block>\\n";
264
265 }
266 @Override
267 public void visitBinaryOperatorNode(TinyLangAst tinyLangAst) {
268     xmlRepresentation+=getCurrentIndentationLevel()+"<binary Op=\\\""+tinyLangAst.
269     getAssociatedNodeValue()+"\\\">\\n";
270     //expected binary operator -> 2 children expression
271     if (tinyLangAst.getChildren().size()!=2)
272         throw new java.lang.RuntimeException("binary node has "+tinyLangAst.getChildren().size()
273         +" child(ren) expected 2");
274     //indent
275     indentation++;
276     //visit expression
277     visitExpression(tinyLangAst.getChildren().get(0));
278     visitExpression(tinyLangAst.getChildren().get(1));
279
280     indentation--;
281     xmlRepresentation+=getCurrentIndentationLevel()+"<\\binary>\\n";
282 }
283 @Override
284 public void visitUnaryOperatorNode(TinyLangAst tinyLangAst) {
285     xmlRepresentation+=getCurrentIndentationLevel()+"<unary Op=\\\""+tinyLangAst.
286     getAssociatedNodeValue()+"\\\">\\n";
287     //expected unary expression node -> one child
288     if (tinyLangAst.getChildren().size()!=1)
289         throw new java.lang.RuntimeException("unary node has "+tinyLangAst.getChildren().size()
290         +" children expected 1");
291     //indent
292     indentation++;
293     //visit expression
294     visitExpression(tinyLangAst.getChildren().get(0));
295     //unindent
296     indentation--;
297     xmlRepresentation+=getCurrentIndentationLevel()+"<\\unary>\\n";
298 }
299 @Override
300 public void visitBooleanLiteralNode(TinyLangAst tinyLangAst) {
301     xmlRepresentation+=getCurrentIndentationLevel()+"<boolean literal>"+tinyLangAst.
302     getAssociatedNodeValue()+"<\\boolean literal>\\n";
303 }
304 @Override
305 public void visitIntegerLiteralNode(TinyLangAst tinyLangAst) {
306     xmlRepresentation+=getCurrentIndentationLevel()+"<integer literal>"+tinyLangAst.
307     getAssociatedNodeValue()+"<\\integer literal>\\n";
308 }
309 @Override
310 public void visitFloatLiteralNode(TinyLangAst tinyLangAst) {
311     xmlRepresentation+=getCurrentIndentationLevel()+"<float literal>"+tinyLangAst.

```

```

    getAssociatedNodeValue()+"<\\float literal >\\n";
306 }
307 @Override
308 public void visitCharLiteralNode(TinyLangAst tinyLangAst) {
309     xmlRepresentation+=getCurrentIndentationLevel()+"<char literal >"+tinyLangAst.
        getAssociatedNodeValue()+"<\\char literal >\\n";
310 }
311
312 @Override
313 public void visitIdentifierNode(TinyLangAst tinyLangAst) {
314     xmlRepresentation+=getCurrentIndentationLevel()+"<id >"+tinyLangAst.getAssociatedNodeValue
        ()+"<\\id >\\n";
315 }
316
317 public void printXmlTree() {
318     System.out.println(xmlRepresentation);
319 }
320 // @Override
321 // public void visitElseBlockNode(TinyLangAst tinyLangAst) {
322 //     xmlRepresentation+=getCurrentIndentationLevel()+"<else block >\\n";
323 //     //indent
324 //     indentation++;
325 //     //children are statements
326 //     for (TinyLangAst child:tinyLangAst.getChildren())
327 //         visitStatement(child);
328 //     indentation--;
329 //     xmlRepresentation+=getCurrentIndentationLevel()+"<\\else block >\\n";
330 // }
331 // }
332 //
333
334 public void visitAssignmentNode(TinyLangAst tinyLangAst) {
335     xmlRepresentation+=getCurrentIndentationLevel()+"<assignment >\\n";
336     indentation++;
337     //add identifier and expression tags
338     visitIdentifierNode(tinyLangAst.getChildren().get(0));
339     visitExpression(tinyLangAst.getChildren().get(1));
340     indentation--;
341     xmlRepresentation+=getCurrentIndentationLevel()+"<\\assignment >\\n";
342 }
343 }

```

Listing 6.11: Generating an XML representation of AST

6.4 | Semantic Analyser

```

1 package tinylangvisitor;
2 import java.util.Objects;
3 import java.util.Stack;
4 import tinylangparser.Type;
5 public class FunctionSignature {
6     private String functionName = "";
7     private int hashCode;
8     Stack<Type> parameterType = new Stack<Type>();
9     public FunctionSignature(String functionName, Stack<Type> parameterType) {
10         //set functionName
11         this.functionName=functionName;
12         //set parameter types stack
13         this.parameterType=parameterType;
14         //set hash
15         hashCode = Objects.hash(functionName, parameterType);
16     }
17     public String getFunctionName() {
18         return functionName;
19     }
20     public Stack<Type> getParametersTypes() {
21         return parameterType;

```

```

22     }
23     /*
24     *functions that allows us to use classes
25     *as map keys where 2 object keys are
26     *equivalent iff they have same attribute values
27     *rather than same object address value
28     */
29     @Override
30     public boolean equals(Object o) {
31         if (this == o)
32             return true;
33         if (o == null || getClass() != o.getClass())
34             return false;
35         FunctionSignature that = (FunctionSignature) o;
36         return functionName.equals(that.functionName) && parameterType.equals(that.
parameterType);
37     }
38     @Override
39     public int hashCode() {
40         return this.hashCode();
41     }
42 }

```

Listing 6.12: Function Signature

```

1 package tinylangvisitor;
2 import java.util.HashMap;
3 import java.util.Map;
4 import java.util.Stack;
5
6 import tinylangparser.TinyLangAst;
7 import tinylangparser.Type;
8 public class Scope {
9     //Signature
10    //name binding i.e. name |-> object e.g. variable,function etc
11    Map<String,Type> variableDeclaration = new HashMap<String,Type>();
12    Map<FunctionSignature,Type> functionDeclaration = new HashMap<FunctionSignature,Type>();
13    Map<FunctionSignature,Stack<String>> functionParameterNames = new HashMap<FunctionSignature,
Stack<String>>();
14    Map<FunctionSignature,TinyLangAst> functionBlock= new HashMap<FunctionSignature,TinyLangAst
>();
15
16    // map := variable ↗ value
17    Map<String,String> variableValues = new HashMap<String,String>();
18
19    // map := function name ↗ value
20
21    public void addVariableDeclaration(String variableName,Type type) {
22        variableDeclaration.put(variableName, type);
23    }
24    //add function declaration
25    public void addFunctionDeclaration(FunctionSignature functionSignature,Type type) {
26        functionDeclaration.put(functionSignature, type);
27    }
28    public TinyLangAst getBlock(FunctionSignature functionSignature) {
29        return functionBlock.get(functionSignature);
30    }
31    public Stack<String> getParameterNames(FunctionSignature functionSignature) {
32        return functionParameterNames.get(functionSignature);
33    }
34
35
36
37    //add value to variable x
38    public void addVariableValue(String x,String value) {
39        variableValues.put(x, value);
40    }
41    public void deleteVariable(String variableName) {
42        variableValues.remove(variableName);
43        variableDeclaration.remove(variableName);
44    }

```

```

45 public void addFunctionParameterNames(FunctionSignature functionSignature , Stack<String>
    variableNames) {
46     functionParameterNames.put(functionSignature , variableNames);
47 }
48 public void addFunctionBlock(FunctionSignature functionSignature , TinyLangAst block) {
49     functionBlock.put(functionSignature , block);
50 }
51
52 public boolean isFunctionAlreadyDefined(FunctionSignature functionSignature) {
53     return functionDeclaration.containsKey(functionSignature);
54 }
55
56
57 //check if name is binded to an entity
58 public boolean isVariableNameBinded(String name) {
59     return variableDeclaration.containsKey(name);
60 }
61 //check if value of variable x is null (does not exists)
62 public boolean isVariableValueNull(String x) {
63     return variableValues.containsKey(x);
64 }
65 public Type getVariableType(String name) {
66     if(isVariableNameBinded(name))
67         return variableDeclaration.get(name);
68     else
69         throw new java.lang.RuntimeException("entity with identifier "+name+" does not exist");
70 }
71 //get value associated with variable x
72 public String getVariableValue(String x) {
73     if(isVariableValueNull(x))
74         return variableValues.get(x);
75     else
76         throw new java.lang.RuntimeException("entity with identifier "+x+" is associated with no
            value");
77 }
78 public Type getFunctionType(FunctionSignature functionSignature) {
79     if(isFunctionAlreadyDefined(functionSignature))
80         return functionDeclaration.get(functionSignature);
81     else
82         throw new java.lang.RuntimeException("function with identifier "+functionSignature.
            getFunctionName()
83             +" and type(s) "+functionSignature.getParametersTypes() +" does not
            exist");
84 }
85 public Map<FunctionSignature ,Type> getFunctionDeclaration(){
86     return functionDeclaration;
87 }
88 }

```

Listing 6.13: Scope

```

1 package tinylangvisitor;
2 import java.util.Stack;
3
4 import tinylangparser.TinyLangAst;
5 import tinylangparser.Type;
6 public class SymbolTable {
7
8     //current function parameter values
9     private Stack<Scope> scopes = new Stack<Scope>();
10    public void push() {
11        Scope newScope = new Scope();
12        scopes.add(newScope);
13    }
14    public void insertVariableDeclaration(String name,Type type) {
15        getCurrentScope().addVariableDeclaration(name, type);
16    }
17    //add value to variable x
18    public void insertVariableValue(String x,String value) {
19        getCurrentScope().addVariableValue(x, value);
20    }
21    public void insertFunctionDeclaration(FunctionSignature functionSignature ,Type type) {

```

```

22     getCurrentScope().addFunctionDeclaration(functionSignature, type);
23 }
24 public void insertFunctionParameterNames(FunctionSignature functionSignature, Stack<String>
    functionParameterNames) {
25     getCurrentScope().addFunctionParameterNames(functionSignature, functionParameterNames);
26 }
27 public void insertFunctionBlock(FunctionSignature functionSignature, TinyLangAst
    functionBlock) {
28     getCurrentScope().addFunctionBlock(functionSignature, functionBlock);
29 }
30 }
31 public void deleteVariable(String name) {
32     getCurrentScope().deleteVariable(name);
33 }
34 }
35 public boolean isVariableNameBinded(String name) {
36     //check is identifier is already binded in current scope
37     return getCurrentScope().isVariableNameBinded(name);
38 }
39 public Type getVariableType(String name) {
40     return getCurrentScope().getVariableType(name);
41 }
42 public void pop(){
43     scopes.pop();
44 }
45 public Stack<Scope> getScopes(){
46     return scopes;
47 }
48 public Scope getCurrentScope() {
49     return scopes.peek();
50 }
51 }

```

Listing 6.14: Symbol Table

```

1 package tinylangvisitor;
2
3 import java.util.HashMap;
4 import java.util.Map;
5 import java.util.Stack;
6
7 import tinylangparser.TinyLangAst;
8 import tinylangparser.TinyLangAstNodes;
9 import tinylangparser.Type;
10
11 public class SemanticAnalyser implements Visitor {
12     /**
13      * Constructor for semantic analysis ,
14      * pass in AST of TinyLang program
15      * to semantically analyse it.
16      * @param programTree
17      */
18     public SemanticAnalyser(TinyLangAst programTree) {
19         //create global scope
20         st.push();
21         //traverse program
22         visitTinyLangProgram(programTree);
23         //confirmation
24         st.pop();
25         System.out.println("Note: program is semantically correct");
26     }
27     //this is used to analyse types of expressions
28     private Type currentExpressionType;
29     //set symbol table
30     private SymbolTable st = new SymbolTable();
31     //get a hold of current function types
32     Stack<Type> function = new Stack<Type>();
33     //get a hold of current function parameters
34     Map<String, Type> currentFunctionParameters = new HashMap<String, Type>();
35
36     //Map<String, Type> currentFunctionParameters = new HashMap<String, Type>();
37

```

```

38 //method which runs statement visit method based on node type
39 public void visitStatement(TinyLangAst tinyLangAst) {
40     switch (tinyLangAst.getAssociatedNodeType()) {
41         case AST_VARIABLE_DECLARATION_NODE:
42             visitVariableDeclarationNode(tinyLangAst);
43             break;
44         case AST_ASSIGNMENT_NODE:
45             visitAssignmentNode(tinyLangAst);
46             break;
47         case AST_PRINT_STATEMENT_NODE:
48             visitPrintStatementNode(tinyLangAst);
49             break;
50         case AST_IF_STATEMENT_NODE:
51             visitIfStatementNode(tinyLangAst);
52             break;
53         case AST_FOR_STATEMENT_NODE:
54             visitForStatementNode(tinyLangAst);
55             break;
56         case AST_WHILE_STATEMENT_NODE:
57             visitWhileStatementNode(tinyLangAst);
58             break;
59         case AST_RETURN_STATEMENT_NODE:
60             visitReturnStatementNode(tinyLangAst);
61             break;
62         case AST_FUNCTION_DECLARATION_NODE:
63             visitFunctionDeclarationNode(tinyLangAst);
64             break;
65         case AST_BLOCK_NODE:
66             visitBlockNode(tinyLangAst);
67             break;
68         default:
69             throw new java.lang.RuntimeException("Unrecognised statement of type "+tinyLangAst.
getAssociatedNodeType());
70     }
71 }
72 //visit expression based on node type
73 private void visitExpression(TinyLangAst tinyLangAst){
74     switch (tinyLangAst.getAssociatedNodeType()) {
75         case AST_BINARY_OPERATOR_NODE:
76             visitBinaryOperatorNode(tinyLangAst);
77             break;
78         case AST_UNARY_OPERATOR_NODE:
79             visitUnaryOperatorNode(tinyLangAst);
80             break;
81         case AST_BOOLEAN_LITERAL_NODE:
82             visitBooleanLiteralNode(tinyLangAst);
83             break;
84         case AST_INTEGER_LITERAL_NODE:
85             visitIntegerLiteralNode(tinyLangAst);
86             break;
87         case AST_FLOAT_LITERAL_NODE:
88             visitFloatLiteralNode(tinyLangAst);
89             break;
90         case AST_CHAR_LITERAL_NODE:
91             visitCharLiteralNode(tinyLangAst);
92             break;
93         case AST_IDENTIFIER_NODE:
94             visitIdentifierNode(tinyLangAst);
95             break;
96         case AST_FUNCTION_CALL_NODE:
97             visitFunctionCallNode(tinyLangAst);
98             break;
99         default:
100             throw new java.lang.RuntimeException("Unrecognised expression node of type "+tinyLangAst
.getAssociatedNodeType());
101     }
102 }
103 @Override
104 public void visitTinyLangProgram(TinyLangAst tinyLangAst) {
105     //traverse all statements
106     for (TinyLangAst statement : tinyLangAst.getChildren())
107         //visit statement
108         visitStatement(statement);

```

```

109 }
110
111 @Override
112 public void visitVariableDeclarationNode(TinyLangAst tinyLangAst) {
113     //get expression
114     TinyLangAst identifier = tinyLangAst.getChildren().get(1);
115     TinyLangAst expression = tinyLangAst.getChildren().get(2);
116     //if identifier is already declared -> ERROR
117     if (st.isVariableNameBinded(identifier.getAssociatedNodeValue())==true)
118         throw new java.lang.RuntimeException(" variable "+identifier.getAssociatedNodeValue()+"
119             in line "
120                 +identifier.getLineNumber()+" was already declared previously");
121
122     //visit expression -> update current expression type
123     visitExpression(expression);
124
125     /* type checking */
126     //we allow type(variable)=float and type(expression)=int (since int can resolve to float)
127     Type varType = Type.valueOf(tinyLangAst.getChildren().get(0).getAssociatedNodeValue());
128     if (varType==Type.FLOAT && getCurrentExpressionType()==Type.INTEGER)
129         //name binding
130         st.insertVariableDeclaration(identifier.getAssociatedNodeValue(), varType);
131     else if (varType==getCurrentExpressionType())
132         //name binding
133         st.insertVariableDeclaration(identifier.getAssociatedNodeValue(), varType);
134     else
135         throw new java.lang.RuntimeException("type mismatch, identifier in line "
136             +identifier.getLineNumber()
137             +" of type"+varType
138             +" and expression in line "
139             +expression.getLineNumber()
140             +" of type"+getCurrentExpressionType());
141 }
142
143 @Override
144 public void visitAssignmentNode(TinyLangAst tinyLangAst) {
145     //get identifier name
146     String variableName = tinyLangAst.getChildren().get(0).getAssociatedNodeValue();
147     //visit expression
148     TinyLangAst expression = tinyLangAst.getChildren().get(1);
149     //get a hold of all scopes
150     Stack<Scope> scopes = st.getScopes();
151     int i = 0;
152     /*
153     * start traversing from inner scope to outer scope to find in
154     * which innermost scope variable is declared
155     */
156     for (i=scopes.size()-1; i>=0; i--) {
157         if (scopes.get(i).isVariableNameBinded(variableName))
158             break;
159     }
160     if (i<0)
161         throw new java.lang.RuntimeException(" identifier "+variableName+" was never declared");
162     //obtain type from scope
163     Type type = scopes.get(i).getVariableType(variableName);
164     //visit expression & update the current expression type
165     visitExpression(expression);
166
167     //handle assignment type mismatch
168
169     //allow integer to resolve to float
170     if (type==Type.FLOAT && getCurrentExpressionType()==Type.INTEGER);
171     else if (type!=getCurrentExpressionType())
172         throw new java.lang.RuntimeException("type mismatch : variable "+variableName
173             +" in line "
174             + tinyLangAst.getChildren()
175             .get(0).getLineNumber()
176             +" of type "+type.toString()
177             +" assigned to expression of type"
178             + getCurrentExpressionType().toString());
179 }
180
181 @Override

```



```

180 public void visitPrintStatementNode(TinyLangAst tinyLangAst) {
181     //visit expression -> update current expression type
182     visitExpression(tinyLangAst.getChildren().get(0));
183 }
184
185 @Override
186 public void visitIfStatementNode(TinyLangAst tinyLangAst) {
187     //get expression
188     TinyLangAst expression = tinyLangAst.getChildren().get(0);
189     //visit expression and update expression current type
190     visitExpression(expression);
191     //check that expression is boolean
192     if(getCurrentExpressionType() != Type.BOOL)
193         throw new java.lang.RuntimeException("if condition in line "
194             + tinyLangAst.getLineNumber()
195             + " is not a predicate expression");
196     //visit if block
197     visitBlockNode(tinyLangAst.getChildren().get(1));
198     //if exists an else block visit it
199     if(tinyLangAst.getChildren().size() == 3)
200         visitBlockNode(tinyLangAst.getChildren().get(2));
201 }
202
203 @Override
204 public void visitForStatementNode(TinyLangAst tinyLangAst) {
205     //first child is variable declaration or expression
206     if(tinyLangAst.getChildren().get(0).getAssociatedNodeType() == TinyLangAstNodes.
207         AST_VARIABLE_DECLARATION_NODE)
208         visitVariableDeclarationNode(tinyLangAst.getChildren().get(0));
209     else
210         visitExpression(tinyLangAst.getChildren().get(0));
211
212     //second child is assignment or block or expression
213     if(tinyLangAst.getChildren().get(1).getAssociatedNodeType() == TinyLangAstNodes.
214         AST_ASSIGNMENT_NODE)
215         visitAssignmentNode(tinyLangAst.getChildren().get(1));
216     else if(tinyLangAst.getChildren().get(1).getAssociatedNodeType() == TinyLangAstNodes.
217         AST_BLOCK_NODE)
218         visitBlockNode(tinyLangAst.getChildren().get(1));
219     else
220         visitExpression(tinyLangAst.getChildren().get(1));
221
222     //if we have 3 children
223     //third child is assignment or block
224     if(tinyLangAst.getChildren().size() == 3 && tinyLangAst.getChildren().get(2).
225         getAssociatedNodeType() == TinyLangAstNodes.AST_ASSIGNMENT_NODE)
226         visitAssignmentNode(tinyLangAst.getChildren().get(2));
227     else if(tinyLangAst.getChildren().size() == 3 && tinyLangAst.getChildren().get(2).
228         getAssociatedNodeType() == TinyLangAstNodes.AST_BLOCK_NODE)
229         visitBlockNode(tinyLangAst.getChildren().get(2));
230
231     //if we have 4 children
232     //fourth child is block
233     if(tinyLangAst.getChildren().size() == 4)
234         visitBlockNode(tinyLangAst.getChildren().get(3));
235 }
236
237 @Override
238 public void visitWhileStatementNode(TinyLangAst tinyLangAst) {
239     //got a hold on expression condition
240     TinyLangAst expression = tinyLangAst.getChildren().get(0);
241     //get a hold on block node
242     TinyLangAst block = tinyLangAst.getChildren().get(1);
243     //visit expression and update current value expression type
244     visitExpression(expression);
245     //expect that the expression is a predicate
246     if(getCurrentExpressionType() != Type.BOOL)
247         throw new java.lang.RuntimeException("expected while condition to be a predicate in line "
248             + tinyLangAst.getLineNumber());
249     //visit block
250     visitBlockNode(block);
251 }
252
253 @Override

```

```

247 public void visitReturnStatementNode(TinyLangAst tinyLangAst) {
248     //get expression
249     TinyLangAst expression = tinyLangAst.getChildren().get(0);
250     //visit expression and update current expression time
251     visitExpression(expression);
252     //we allow to return integer if function is of type float
253     if (!function.empty() && getCurrentExpressionType() == Type.INTEGER && function.peek() == Type.FLOAT);
254     //check that expression is has the same type as the function
255     else if (!function.empty() && getCurrentExpressionType() != function.peek())
256         throw new java.lang.RuntimeException("return in line "
257             +tinyLangAst.getLineNumber()+" returns expression of type "+
258             getCurrentExpressionType()+" expected type "+function.peek());
259 }
260
261 @Override
262 public void visitFunctionDeclarationNode(TinyLangAst tinyLangAst) {
263     //get function type
264     Type functionType = Type.valueOf(tinyLangAst.getChildren().get(0).getAssociatedNodeValue());
265
266     //get function identifier
267     String functionName = tinyLangAst.getChildren().get(1).getAssociatedNodeValue();
268
269     //get function parameter types
270     Stack<Type> functionParameterTypes = new Stack<Type>();
271     //get stack of names to check for duplicate parameter names
272     Stack<String> functionParameterNames = new Stack<String>();
273     //add types
274
275     //get current paramater name
276     TinyLangAst parameterName;
277     for(TinyLangAst formalParameterTypes : tinyLangAst.getChildren().get(2).getChildren()) {
278         parameterName = formalParameterTypes.getChildren().get(1);
279         functionParameterTypes.push(Type.valueOf(
280             formalParameterTypes.getChildren()
281             .get(0).getAssociatedNodeValue()));
282         //if parameter name is duplicate throw exception
283         if (functionParameterNames.contains(parameterName.getAssociatedNodeValue()))
284             throw new java.lang.RuntimeException("function parameter name "+parameterName.
285                 getAssociatedNodeValue()+
286                 " already defined in line "+parameterName.getLineNumber());
287         functionParameterNames.push(parameterName.getAssociatedNodeValue());
288     }
289
290     //check in all scopes that the function is not already defined
291     for(Scope scope : st.getScopes())
292         if (scope.isFunctionAlreadyDefined(new FunctionSignature(functionName,
293             functionParameterTypes)))
294             throw new java.lang.RuntimeException("function "+functionName+" in line "+tinyLangAst.
295                 getChildren().get(1).getLineNumber()+" with the same parameter types already defined
296                 previously");
297     //add function to st
298     st.insertFunctionDeclaration(new FunctionSignature(functionName, functionParameterTypes,
299         functionType);
300     //record current function in stack
301     function.push(functionType);
302     //empty current function parameters
303     currentFunctionParameters.clear();
304     for(TinyLangAst formalParameter : tinyLangAst.getChildren().get(2).getChildren())
305         currentFunctionParameters.put(formalParameter.getChildren().get(1).
306             getAssociatedNodeValue(),
307             Type.valueOf(formalParameter.getChildren()
308                 .get(0).getAssociatedNodeValue()));
309     //visit block
310     visitBlockNode(tinyLangAst.getChildren().get(3));
311     //pop type
312     function.pop();
313     //check if function returns
314     if (!returns(tinyLangAst.getChildren().get(3)))
315         throw new java.lang.RuntimeException("function "+functionName+" in line "+tinyLangAst.
316             getLineNumber()+" not expected to return");
317 }

```

```

311
312 @Override
313 public void visitFunctionCallNode(TinyLangAst tinyLangAst) {
314     //determine the signature of the function
315     Stack<Type> parameterTypes = new Stack<Type>();
316     String functionIdentifier = tinyLangAst.getChildren().get(0).getAssociatedNodeValue();
317     //identify the expressions and update stack current expression types
318     for(TinyLangAst expression : tinyLangAst.getChildren().get(1).getChildren()) {
319         visitExpression(expression);
320         parameterTypes.push(getCurrentExpressionType());
321     }
322     Stack<Scope> scopes = st.getScopes();
323     int i;
324
325     for(i=scopes.size()-1;i>=0;i--)
326         if(scopes.get(i).isFunctionAlreadyDefined(new FunctionSignature(functionIdentifier,
327             parameterTypes)))
328             break;
329
330     if(i<0)
331         throw new java.lang.RuntimeException("function "+functionIdentifier+" in line "+
332             tinyLangAst.getLineNumber()+" is not defined");
333
334     //if defined set current expression type to return value of the function
335     setCurrentExpressionType(scopes.get(i).getFunctionType(new FunctionSignature(
336         functionIdentifier, parameterTypes)));
337 }
338
339 @Override
340 public void visitBlockNode(TinyLangAst tinyLangAst) {
341     //create new scope
342     st.push();
343     //add parameters of functions if any in scope
344     for(String variableName:currentFunctionParameters.keySet())
345         st.insertVariableDeclaration(variableName, currentFunctionParameters.get(variableName));
346     //clear parameter map
347     currentFunctionParameters.clear();
348     //traverse statements in block
349     for(TinyLangAst statement:tinyLangAst.getChildren())
350         visitStatement(statement);
351     //visit statements in block
352     //end scope
353     st.pop();
354 }
355
356 // @Override
357 // public void visitElseBlockNode(TinyLangAst tinyLangAst) {
358 //     visitBlockNode(tinyLangAst);
359 // }
360
361 @Override
362 public void visitBinaryOperatorNode(TinyLangAst tinyLangAst) {
363     //get operator
364     String operator = tinyLangAst.getAssociatedNodeValue();
365     //get left node (left operand)
366     TinyLangAst leftOperand = tinyLangAst.getChildren().get(0);
367     //visit expression to update current char type
368     visitExpression(leftOperand);
369     //obtain the type of the left operand
370     Type leftOperandType = getCurrentExpressionType();
371
372     //REDO for right node (right operand)
373     //get left node (left operand)
374     TinyLangAst rightOperand = tinyLangAst.getChildren().get(1);
375     //visit expression to update current char type
376     visitExpression(rightOperand);
377     //obtain the type of the left operand
378     Type rightOperandType = getCurrentExpressionType();
379
380     /*
381      * Operators
382      *
383      * Operators 'and' | 'or' must have operands of type bool

```

```

381 *
382 * Operator '+' | '-' | '/' | '*' | '<' | '>' | '<=' | '>=' work on numeric operators
383 *
384 * Operators '==' | '!=' operates on any 2 operands of the same type both numeric or both
boolean or both char
385 */
386 if (operator.equals("and") || operator.equals("or")) {
387     if (leftOperandType==Type.BOOL && rightOperandType==Type.BOOL)
388         setCurrentExpressionType (Type.BOOL);
389
390     else
391         throw new java.lang.RuntimeException("expected 2 operands of boolean type for operator
"
392
393             +operator+" in line "+tinyLangAst.getLineNumber());
394 }
395 else if (operator.equals("+") || operator.equals("-") || operator.equals("/") || operator.equals
(" * ")) {
396     if (!isNumericType(leftOperandType) || !isNumericType(rightOperandType))
397         throw new java.lang.RuntimeException("expected 2 operands of numeric type for operator
"
398
399             +operator+" in line "+tinyLangAst.getLineNumber());
400
401     //if both are numeric if one of them is float the operator returns float otherwise
returns integer
402     if (leftOperandType==Type.FLOAT || rightOperandType==Type.FLOAT)
403         setCurrentExpressionType (Type.FLOAT);
404     else
405         setCurrentExpressionType (Type.INTEGER);
406 }
407 else if (operator.equals("<") || operator.equals(">") || operator.equals("<=") || operator.equals
(">=")) {
408     if (!isNumericType(leftOperandType) || !isNumericType(rightOperandType))
409         throw new java.lang.RuntimeException("expected 2 operands of numeric type for operator
"
410
411             +operator+" in line "+tinyLangAst.getLineNumber());
412     //if both are numeric set relation operators returns a boolean value
413     setCurrentExpressionType (Type.BOOL);
414 }
415 else if (operator.equals("==") || operator.equals("!=")) {
416     //handle mismatch not that float and integers are
417     //both considered as one numerical type
418     if ((leftOperandType!=rightOperandType) &&
419         (!isNumericType(leftOperandType) || !isNumericType(rightOperandType)))
420         throw new java.lang.RuntimeException("operand mismatch in line "+tinyLangAst.
421             getLineNumber());
422     //if operands match
423     setCurrentExpressionType (Type.BOOL);
424 }
425 else
426     throw new java.lang.RuntimeException("binary operator "+operator+" unrecognised");
427 }
428
429 @Override
430 public void visitUnaryOperatorNode(TinyLangAst tinyLangAst) {
431     //unary operator
432     String operator = tinyLangAst.getAssociatedNodeValue();
433     //visit expression
434     visitExpression(tinyLangAst.getChildren().get(0));
435     //if current expression is numerical
436     if (getCurrentExpressionType()==Type.INTEGER || getCurrentExpressionType()==Type.FLOAT)
437         //check if operator is '-' | '+'
438         if (!operator.equals("-") && !operator.equals("+"))
439             throw new java.lang.RuntimeException("operator "+operator+" not allowed in front of
numerical expression in line "+tinyLangAst.getChildren().get(0).getLineNumber());
440
441     else if (getCurrentExpressionType()==Type.BOOL )
442         //check if operator is not
443         if (!operator.equals("not"))
444             throw new java.lang.RuntimeException("operator "+operator+" not allowed in front of
predicate exreesion in line "+tinyLangAst.getChildren().get(0).getLineNumber());
445     else

```

```

444     throw new java.lang.RuntimeException("unary operator "+operator+" is incompatible with
445     expression in line "+tinyLangAst.getLineNumber());
446 }
447 @Override
448 public void visitIdentifierNode(TinyLangAst tinyLangAst) {
449     //find scope where identifier is defined
450     Stack<Scope> scopes = st.getScopes();
451     int i;
452     for(i=scopes.size()-1;i>=0;i--)
453         if(scopes.get(i).isVariableNameBound(tinyLangAst.getAssociatedNodeValue()))
454             break;
455     if(i<0)
456         throw new java.lang.RuntimeException("variable name "+tinyLangAst.getAssociatedNodeValue()
457         +" in line "+tinyLangAst.getLineNumber()+" is not defined");
458     setCurrentExpressionType(scopes.get(i).getVariableType(tinyLangAst.getAssociatedNodeValue()));
459 }
460 @Override
461 public void visitBooleanLiteralNode(TinyLangAst tinyLangAst) {
462     setCurrentExpressionType(Type.BOOL);
463 }
464 @Override
465 public void visitIntegerLiteralNode(TinyLangAst tinyLangAst) {
466     setCurrentExpressionType(Type.INTEGER);
467 }
468 @Override
469 public void visitFloatLiteralNode(TinyLangAst tinyLangAst) {
470     setCurrentExpressionType(Type.FLOAT);
471 }
472 @Override
473 public void visitCharLiteralNode(TinyLangAst tinyLangAst) {
474     setCurrentExpressionType(Type.CHAR);
475 }
476 private boolean isNumericType(Type type) {
477     if(type ==Type.INTEGER||type ==Type.FLOAT)
478         return true;
479     else
480         return false;
481 }
482 private boolean returns(TinyLangAst tinyLangAst) {
483     //if given statement is a return statement
484     //then obviously we have that the function returns
485     if(tinyLangAst.getAssociatedNodeType()==TinyLangAstNodes.AST_RETURN_STATEMENT_NODE)
486         return true;
487     //given a block we check if one of the statement returns
488     if(tinyLangAst.getAssociatedNodeType()==TinyLangAstNodes.AST_BLOCK_NODE) {
489         for(TinyLangAst statement:tinyLangAst.getChildren())
490             if(returns(statement))
491                 return true;
492     }
493     //given a block we check if one of the statement returns
494     if(tinyLangAst.getAssociatedNodeType()==TinyLangAstNodes.AST_ELSE_BLOCK_NODE) {
495         for(TinyLangAst statement:tinyLangAst.getChildren())
496             if(returns(statement))
497                 return true;
498     }
499     //if statement with an else block returns if both statement returns
500     if(tinyLangAst.getAssociatedNodeType()==TinyLangAstNodes.AST_IF_STATEMENT_NODE)
501         //if statement has else block
502         if(tinyLangAst.getChildren().size()==3) {
503             //block and else block both return
504             return returns(tinyLangAst.getChildren().get(1)) && returns(tinyLangAst.getChildren().
505             get(2));
506         }
507     //if statement with an for block returns if both statement returns
508     if(tinyLangAst.getAssociatedNodeType()==TinyLangAstNodes.AST_FOR_STATEMENT_NODE)

```

```

513     return returns(tinyLangAst.getChildren().get(tinyLangAst.getChildren().size()-1));
514
515     //if statement with an else block returns if both statement returns
516     if(tinyLangAst.getAssociatedNodeType()==TinyLangAstNodes.AST_WHILE_STATEMENT_NODE)
517         return returns(tinyLangAst.getChildren().get(1));
518     else
519         //in all other cases the function do not return
520         return false;
521 }
522 public void setCurrentExpressionType(Type currentExpressionType) {
523     this.currentExpressionType=currentExpressionType;
524 }
525 public Type getCurrentExpressionType() {
526     return currentExpressionType;
527 }
528 }

```

Listing 6.15: Semantic Analyser

6.5 | Interpreter

```

1 package tinylangvisitor;
2 import java.util.Stack;
3 import tinylangparser.TinyLangAst;
4 import tinylangparser.TinyLangAstNodes;
5 import tinylangparser.Type;
6 /**
7  * Class interpreter
8  * @author andre
9  *
10 */
11 public class Interpreter implements Visitor{
12     //create a symbol table
13     private SymbolTable st = new SymbolTable();
14     //save current expression type for evaluation
15     private Type currentExpressionType;
16     //save current expression value for evaluation
17     private String currentExpressionValue;
18
19     //save temporary information on function call parameters
20     private Stack<Type> parameterTypes = new Stack<Type>();
21     private Stack<String> parameterNames= new Stack<String>();
22     private Stack<String> parameterValues= new Stack<String>();
23
24
25     public Interpreter(TinyLangAst intermediateRepresentation) {
26         //analyse the representation semantically
27         new SemanticAnalyser(intermediateRepresentation);
28         //push global scope
29         st.push();
30         //interpret tinyLangProgram
31         visitTinyLangProgram(intermediateRepresentation);
32     }
33     //method which runs statement visit method based on node type
34     private void visitStatement(TinyLangAst tinyLangAst) {
35         switch(tinyLangAst.getAssociatedNodeType()) {
36             case AST_VARIABLE_DECLARATION_NODE:
37                 visitVariableDeclarationNode(tinyLangAst);
38                 break;
39             case AST_ASSIGNMENT_NODE:
40                 visitAssignmentNode(tinyLangAst);
41                 break;
42             case AST_PRINT_STATEMENT_NODE:
43                 visitPrintStatementNode(tinyLangAst);
44                 break;
45             case AST_IF_STATEMENT_NODE:
46                 visitIfStatementNode(tinyLangAst);

```

```

47         break;
48     case AST_FOR_STATEMENT_NODE:
49         visitForStatementNode(tinyLangAst);
50         break;
51     case AST_WHILE_STATEMENT_NODE:
52         visitWhileStatementNode(tinyLangAst);
53         break;
54     case AST_RETURN_STATEMENT_NODE:
55         visitReturnStatementNode(tinyLangAst);
56         break;
57     case AST_FUNCTION_DECLARATION_NODE:
58         visitFunctionDeclarationNode(tinyLangAst);
59         break;
60     case AST_BLOCK_NODE:
61         visitBlockNode(tinyLangAst);
62         break;
63     default:
64         throw new java.lang.RuntimeException("Unrecognised statement of type "+tinyLangAst.
getAssociatedNodeType());
65     }
66 }
67 //visit expression
68 //visit expression based on node type
69 private void visitExpression(TinyLangAst tinyLangAst){
70     switch(tinyLangAst.getAssociatedNodeType()) {
71     case AST_BINARY_OPERATOR_NODE:
72         visitBinaryOperatorNode(tinyLangAst);
73         break;
74     case AST_UNARY_OPERATOR_NODE:
75         visitUnaryOperatorNode(tinyLangAst);
76         break;
77     case AST_BOOLEAN_LITERAL_NODE:
78         visitBooleanLiteralNode(tinyLangAst);
79         break;
80     case AST_INTEGER_LITERAL_NODE:
81         visitIntegerLiteralNode(tinyLangAst);
82         break;
83     case AST_FLOAT_LITERAL_NODE:
84         visitFloatLiteralNode(tinyLangAst);
85         break;
86     case AST_CHAR_LITERAL_NODE:
87         visitCharLiteralNode(tinyLangAst);
88         break;
89     case AST_IDENTIFIER_NODE:
90         visitIdentifierNode(tinyLangAst);
91         break;
92     case AST_FUNCTION_CALL_NODE:
93         visitFunctionCallNode(tinyLangAst);
94         break;
95     default:
96         throw new java.lang.RuntimeException("Unrecognised expression node of type "+
tinyLangAst.getAssociatedNodeType());
97     }
98 }
99 @Override
100 public void visitTinyLangProgram(TinyLangAst tinyLangAst) {
101     //program ≡ sequence of statements : traverse all statement nodes
102     for(TinyLangAst statement : tinyLangAst.getChildren())
103         visitStatement(statement);
104 }
105 @Override
106 public void visitVariableDeclarationNode(TinyLangAst tinyLangAst) {
107     //get variable type
108     Type varType = Type.valueOf(tinyLangAst.getChildren().get(0).getAssociatedNodeValue());
109     //get hold on identifier
110     String varName = tinyLangAst.getChildren().get(1).getAssociatedNodeValue();
111     //visit expression and update current expression value
112     TinyLangAst expression = tinyLangAst.getChildren().get(2);
113     visitExpression(expression);
114     //add variable declaration in current scope
115     st.insertVariableDeclaration(varName, varType);
116     //add value assigned to variable
117     st.insertVariableValue(varName, currentExpressionValue);

```

```

118 }
119 @Override
120 public void visitAssignmentNode(TinyLangAst tinyLangAst) {
121     //get identifier name
122     String varName = tinyLangAst.getChildren().get(0).getAssociatedNodeValue();
123     //update current expression value
124     TinyLangAst expression = tinyLangAst.getChildren().get(1);
125     visitExpression(expression);
126     int i;
127     /*
128     * start traversing from inner scope to outer scope to find in
129     * which innermost scope variable is declared
130     */
131     for(i=st.getScopes().size()-1; i>=0; i--) {
132         if(st.getScopes().get(i).isVariableNameBinded(varName))
133             break;
134     }
135     /*
136     * go in that innermost scope and update the value
137     */
138     st.getScopes().get(i).addVariableValue(varName, currentExpressionValue);
139 }
140
141 @Override
142 public void visitPrintStatementNode(TinyLangAst tinyLangAst) {
143     visitExpression(tinyLangAst.getChildren().get(0));
144     System.out.println(currentExpressionValue);
145 }
146 @Override
147 public void visitIfStatementNode(TinyLangAst tinyLangAst) {
148     TinyLangAst expression = tinyLangAst.getChildren().get(0);
149     //evaluate if condition
150     visitExpression(expression);
151     //check condition
152     if(currentExpressionValue.equals("true"))
153         visitBlockNode(tinyLangAst.getChildren().get(1));
154     //if we have an else block
155     else if (currentExpressionValue.equals("false") && tinyLangAst.getChildren().size()==3)
156         visitBlockNode(tinyLangAst.getChildren().get(2));
157 }
158
159 @Override
160 public void visitForStatementNode(TinyLangAst tinyLangAst) {
161     //we have a list of possibilities for a for loop statement
162
163     //no variable declaration and no assignment
164
165     /*
166     *           for loop
167     *           / \
168     *           / \
169     *   *   expression   block
170     */
171
172     //this can be encoded as a while loop statement
173     if(tinyLangAst.getChildren().size()==2) {
174         TinyLangAst expression = tinyLangAst.getChildren().get(0);
175         TinyLangAst block = tinyLangAst.getChildren().get(1);
176         visitExpression(expression);
177         while(currentExpressionValue.equals("true")) {
178             //visit block
179             visitBlockNode(block);
180             //update current expression value
181             visitExpression(expression);
182         }
183     }
184     //if we have both variable declaration and assignment
185
186     /*
187     *           for loop ---\
188     *           / / \ \ \
189     *           / / \ \ \
190     */

```



```

191      *      / |      \      block
192      *      *      / expression \
193      *      variable      \
194      *      declaration      updation/assignment
195      *
196      */
197      else if (tinyLangAst.getChildren().size() == 4) {
198          TinyLangAst variableDeclaration = tinyLangAst.getChildren().get(0);
199          //visit variable declaration
200          visitVariableDeclarationNode(variableDeclaration);
201          //visit expression and update current expression value
202          visitExpression(tinyLangAst.getChildren().get(1));
203          while (currentExpressionValue.equals("true")) {
204              //visit block
205              visitBlockNode(tinyLangAst.getChildren().get(3));
206              //carry out updation/assignment
207              visitAssignmentNode(tinyLangAst.getChildren().get(2));
208              //update current expression value
209              visitExpression(tinyLangAst.getChildren().get(1));
210          }
211          st.deleteVariable(variableDeclaration.getChildren().get(1).getAssociatedNodeValue());
212      }
213      //if we have variable declaration and no assignment
214
215      /*
216      *      for loop
217      *      / / / \
218      *      / / \
219      *      *      / expression \
220      *      variable      \
221      *      declaration      block
222      *
223      */
224      else if (tinyLangAst.getChildren().get(0).getAssociatedNodeType() == TinyLangAstNodes.
AST_VARIABLE_DECLARATION_NODE) {
225          TinyLangAst variableDeclaration = tinyLangAst.getChildren().get(0);
226          //declare variable
227          visitVariableDeclarationNode(variableDeclaration);
228          //update current expression value
229          visitExpression(tinyLangAst.getChildren().get(0));
230          while (currentExpressionValue.equals("true")) {
231              //execute statements
232              visitBlockNode(tinyLangAst.getChildren().get(2));
233              //update current expression value
234              visitExpression(tinyLangAst.getChildren().get(0));
235          }
236          st.deleteVariable(variableDeclaration.getChildren().get(1).getAssociatedNodeValue());
237      }
238
239      //if we have assignment and no variable declaration
240
241      /*
242      *      for loop
243      *      / / / \
244      *      / / \
245      *      *      / / \
246      *      *      / assignment \
247      *      expression      \
248      *      block
249      *
250      */
251      else if (tinyLangAst.getChildren().get(1).getAssociatedNodeType() == TinyLangAstNodes.
AST_ASSIGNMENT_NODE)
252      {
253          //visit expression and update current expression value
254          visitExpression(tinyLangAst.getChildren().get(0));
255          while (currentExpressionValue.equals("true")) {
256              //visit block
257              //carry out update/assignment
258              visitAssignmentNode(tinyLangAst.getChildren().get(1));
259              //update current expression value
260              visitExpression(tinyLangAst.getChildren().get(0));
261          }

```

```

262     }
263     else
264         throw new java.lang.RuntimeException("unexpected for loop case in line "+tinyLangAst.
            getLineNumber());
265     }
266
267     @Override
268     public void visitWhileStatementNode(TinyLangAst tinyLangAst) {
269         //get a hold on block of while loop
270         TinyLangAst block = tinyLangAst.getChildren().get(1);
271         //update current expression value
272         TinyLangAst expression = tinyLangAst.getChildren().get(0);
273         visitExpression(expression);
274         //while current expression value is true
275         //keep on looping
276         while(currentExpressionValue.equals("true")) {
277             //visit block
278             visitBlockNode(block);
279             //update current expression value
280             visitExpression(expression);
281         }
282     }
283
284     @Override
285     public void visitReturnStatementNode(TinyLangAst tinyLangAst) {
286         //update current expression value
287         visitExpression(tinyLangAst.getChildren().get(0));
288     }
289
290     @Override
291     public void visitFunctionDeclarationNode(TinyLangAst tinyLangAst) {
292         //add function definition and values to symbol table
293         //get function block ast
294         TinyLangAst functionBlock = tinyLangAst.getChildren().get(3);
295         //get variable type
296         Type functionType = Type.valueOf(tinyLangAst.getChildren().get(0).getAssociatedNodeValue());
297         //get hold on identifier
298         String functionName = tinyLangAst.getChildren().get(1).getAssociatedNodeValue();
299         //get function parameter types
300         Stack<Type> functionParameterTypes = new Stack<Type>();
301         Stack<String> functionParameterNames = new Stack<String>();
302         //add parameters types and values
303         for(TinyLangAst formalParameter : tinyLangAst.getChildren().get(2).getChildren()) {
304             functionParameterTypes.push(Type.valueOf(formalParameter.getChildren().get(0).
                getAssociatedNodeValue()));
305             functionParameterNames.push(formalParameter.getChildren().get(1).getAssociatedNodeValue());
306         }
307         //add function parameter types and names to st
308         st.insertFunctionDeclaration(new FunctionSignature(functionName, functionParameterTypes),
            functionType);
309         st.insertFunctionParameterNames(new FunctionSignature(functionName, functionParameterTypes),
            functionParameterNames);
310         st.insertFunctionBlock(new FunctionSignature(functionName, functionParameterTypes),
            functionBlock);
311     }
312     @Override
313     public void visitFunctionCallNode(TinyLangAst tinyLangAst) {
314
315         //function name
316         String functionName = tinyLangAst.getChildren().get(0).getAssociatedNodeValue();
317         for(TinyLangAst expression : tinyLangAst.getChildren().get(1).getChildren()) {
318             visitExpression(expression);
319             parameterTypes.push(currentExpressionType);
320             parameterValues.push(currentExpressionValue);
321         }
322         //function signature types of parameters
323         int i;
324         for(i=st.getScopes().size()-1; i>=0; i--)
325             if(st.getScopes().get(i).isFunctionAlreadyDefined(new FunctionSignature(functionName,
                parameterTypes)))
326                 break;

```

```

327 //add temporary function parameters names
328 parameterNames.addAll(st.getScopes().get(i).getParameterNames(new FunctionSignature(
functionName, parameterTypes)));
329 //visit corresponding function block
330 visitBlockNode(st.getScopes().get(i).getBlock(new FunctionSignature(functionName,
parameterTypes)));
331
332 }
333
334 @Override
335 public void visitBlockNode(TinyLangAst tinyLangAst) {
336 //enter a new scope
337 st.push();
338 //check all temporary function parameter stacks are of the same size
339 if (!(parameterTypes.size() == parameterNames.size() && parameterNames.size() == parameterValues.
size()))
340 throw new java.lang.RuntimeException("error with function call handling");
341 //add parameters of functions if any in scope
342 for(int i=0; i<parameterTypes.size(); i++) {
343 //add variable declaration in current scope
344 st.insertVariableDeclaration(parameterNames.get(i), parameterTypes.get(i));
345 //add value assigned to variable
346 st.insertVariableValue(parameterNames.get(i), parameterValues.get(i));
347 }
348 //clear temporary function parameters data
349 parameterTypes.clear();
350 parameterNames.clear();
351 parameterValues.clear();
352 //traverse statements in block
353 for(TinyLangAst statement: tinyLangAst.getChildren())
354 visitStatement(statement);
355 //leave scope
356 st.pop();
357 }
358
359
360 @Override
361 public void visitBinaryOperatorNode(TinyLangAst tinyLangAst) {
362 //get operator
363
364 String operator = tinyLangAst.getAssociatedNodeValue();
365
366 //get left node (left operand)
367 TinyLangAst leftOperand = tinyLangAst.getChildren().get(0);
368 //visit expression to update current char type
369 visitExpression(leftOperand);
370 //obtain the type of the left operand
371 Type leftOperandType = currentExpressionType;
372 //obtain the value of the left operand
373 String leftOperandValue = currentExpressionValue;
374
375 //redo for right operand
376 TinyLangAst rightOperand = tinyLangAst.getChildren().get(1);
377 visitExpression(rightOperand);
378 Type rightOperandType = currentExpressionType;
379 String rightOperandValue = currentExpressionValue;
380 if (operator.equals("+")) {
381 //check operand type
382 if (leftOperandType.equals(Type.INTEGER) && rightOperandType.equals(Type.INTEGER)) {
383 //int+int -> int
384 currentExpressionType = Type.INTEGER;
385 currentExpressionValue = String.valueOf(Integer.parseInt(leftOperandValue)+Integer.
parseInt(rightOperandValue));
386 }
387 //if one is floating
388 else if (leftOperandType.equals(Type.FLOAT) || rightOperandType.equals(Type.FLOAT)) {
389 //int+int -> int
390 currentExpressionType = Type.FLOAT;
391 currentExpressionValue = String.valueOf(Float.parseFloat(leftOperandValue)+Float.
parseFloat(rightOperandValue));
392 }
393 else {
394 throw new java.lang.RuntimeException("unexpected operator processing exception in line

```

```

    "+tinyLangAst.getLineNumber());
    }
}
else if (operator.equals("-")){
    //check operand type
    if (leftOperandType.equals(Type.INTEGER)&&rightOperandType.equals(Type.INTEGER)) {
        //int+int -> int
        currentExpressionType = Type.INTEGER;
        currentExpressionValue = String.valueOf(Integer.parseInt(leftOperandValue)-Integer.
        parseInt(rightOperandValue));
    }
    //if one is floating
    else if (leftOperandType.equals(Type.FLOAT)||rightOperandType.equals(Type.FLOAT)) {
        currentExpressionType = Type.FLOAT;
        currentExpressionValue = String.valueOf(Float.parseFloat(leftOperandValue)-Float.
        parseFloat(rightOperandValue));
    }
    else
        throw new java.lang.RuntimeException("unexpected operator processing exception in line
        "+tinyLangAst.getLineNumber());
}

else if (operator.equals("*")){
    //check operand type
    if (leftOperandType.equals(Type.INTEGER)&&rightOperandType.equals(Type.INTEGER)) {
        //int+int -> int
        currentExpressionType = Type.INTEGER;
        currentExpressionValue = String.valueOf(Integer.parseInt(leftOperandValue)*Integer.
        parseInt(rightOperandValue));
    }
    //if one is floating
    else if (leftOperandType.equals(Type.FLOAT)||rightOperandType.equals(Type.FLOAT)) {
        currentExpressionType = Type.FLOAT;
        currentExpressionValue = String.valueOf(Float.parseFloat(leftOperandValue)*Float.
        parseFloat(rightOperandValue));
    }
    else
        throw new java.lang.RuntimeException("unexpected operator processing exception in line
        "+tinyLangAst.getLineNumber());
}

else if (operator.equals("/")){
    //check if right operand is 0
    if (Float.parseFloat(rightOperandValue)==0)
        throw new java.lang.RuntimeException("division by 0 undefined in line "+tinyLangAst.
        getLineNumber());
    //check operand type
    if (leftOperandType.equals(Type.INTEGER)&&rightOperandType.equals(Type.INTEGER)) {
        //int+int -> int
        currentExpressionType = Type.INTEGER;
        currentExpressionValue = String.valueOf(Integer.parseInt(leftOperandValue)/Integer.
        parseInt(rightOperandValue));
    }
    //if one is floating
    else if (leftOperandType.equals(Type.FLOAT)||rightOperandType.equals(Type.FLOAT)) {
        currentExpressionType = Type.FLOAT;
        currentExpressionValue = String.valueOf(Float.parseFloat(leftOperandValue)/Float.
        parseFloat(rightOperandValue));
    }
    else
        throw new java.lang.RuntimeException("unexpected runtime exception in line "+
        tinyLangAst.getLineNumber());
}

//boolean operators
else if (operator.equals("and")) {
    currentExpressionType = Type.BOOL;
    if (leftOperandValue.equals("true") && rightOperandValue.equals("true"))
        currentExpressionValue = "true";
    else
        currentExpressionValue = "false";
}

else if (operator.equals("or")) {
    currentExpressionType = Type.BOOL;
    if (leftOperandValue.equals("true") || rightOperandValue.equals("true"))

```

```

457     currentExpressionValue = "true ";
458     else
459         currentExpressionValue = "false ";
460 }
461 //comparison types
462 else if (operator.equals("=")) {
463     currentExpressionType = Type.BOOL;
464     if (leftOperandValue.equals(rightOperandValue))
465         currentExpressionValue = "true ";
466     else
467         currentExpressionValue = "false ";
468 }
469 else if (operator.equals("!=")) {
470     currentExpressionType = Type.BOOL;
471     if (!leftOperandValue.equals(rightOperandValue))
472         currentExpressionValue = "true ";
473     else
474         currentExpressionValue = "false ";
475 }
476 else if (operator.equals("<")) {
477     currentExpressionType = Type.BOOL;
478     if (Float.parseFloat(leftOperandValue)<Float.parseFloat(rightOperandValue))
479         currentExpressionValue = "true ";
480     else
481         currentExpressionValue = "false ";
482 }
483 else if (operator.equals("<=")) {
484     currentExpressionType = Type.BOOL;
485     if (Float.parseFloat(leftOperandValue)<=Float.parseFloat(rightOperandValue))
486         currentExpressionValue = "true ";
487     else
488         currentExpressionValue = "false ";
489 }
490 else if (operator.equals(">")) {
491     currentExpressionType = Type.BOOL;
492     if (Float.parseFloat(leftOperandValue)>Float.parseFloat(rightOperandValue))
493         currentExpressionValue = "true ";
494     else
495         currentExpressionValue = "false ";
496 }
497 else if (operator.equals(">=")) {
498     currentExpressionType = Type.BOOL;
499     if (Float.parseFloat(leftOperandValue)>=Float.parseFloat(rightOperandValue))
500         currentExpressionValue = "true ";
501     else
502         currentExpressionValue = "false ";
503 }
504 else {
505     throw new java.lang.RuntimeException("unexcepted binary operator error in line "+
tinyLangAst.getLineNumber());
506 }
507 }
508 @Override
509 public void visitUnaryOperatorNode(TinyLangAst tinyLangAst) {
510     TinyLangAst expression = tinyLangAst.getChildren().get(0);
511     visitExpression(expression);
512     String operator = tinyLangAst.getAssociatedNodeValue();
513     if (currentExpressionType==Type.FLOAT) {
514         if (operator.equals("-"))
515             currentExpressionValue = String.valueOf(-1*Float.parseFloat(currentExpressionValue));
516     }
517     else if (currentExpressionType==Type.INTEGER) {
518         if (operator.equals("-")) {
519             currentExpressionValue = String.valueOf(-1*Integer.parseInt(currentExpressionValue));
520         }
521     }
522     else if (currentExpressionType==Type.BOOL) {
523         if (operator.equals("not")) {
524             if (currentExpressionValue.equals("true"))
525                 currentExpressionValue = "false ";
526             else
527                 currentExpressionValue = "true ";
528         }
529     }

```


```

529     }
530     else
531         throw new java.lang.RuntimeException
532             ("unexpected error when handling unary opertor in line "+tinyLangAst.getLineNumber());
533     }
534     @Override
535     public void visitIdentifierNode(TinyLangAst tinyLangAst) {
536         //Identifier name
537         String identifier = tinyLangAst.getAssociatedNodeValue();
538         //traverse the scopes to find the identifier type and value
539         int i;
540         for(i=st.getScopes().size()-1;i>=0;i--) {
541             if(st.getScopes().get(i).isVariableNameBinded(identifier))
542                 break;
543         }
544         currentExpressionType = st.getScopes().get(i).getVariableType(identifier);
545         currentExpressionValue = st.getScopes().get(i).getVariableValue(identifier);
546     }
547
548     @Override
549     public void visitBooleanLiteralNode(TinyLangAst tinyLangAst) {
550         String boolIdentifier = tinyLangAst.getAssociatedNodeValue();
551         currentExpressionType = Type.BOOL;
552         currentExpressionValue = boolIdentifier;
553     }
554
555     @Override
556     public void visitIntegerLiteralNode(TinyLangAst tinyLangAst) {
557         String integerIdentifier = tinyLangAst.getAssociatedNodeValue();
558         currentExpressionType = Type.INTEGER;
559         currentExpressionValue = integerIdentifier;
560     }
561
562     @Override
563     public void visitFloatLiteralNode(TinyLangAst tinyLangAst) {
564         String floatIdentifier = tinyLangAst.getAssociatedNodeValue();
565         currentExpressionType = Type.FLOAT;
566         currentExpressionValue = floatIdentifier;
567     }
568     @Override
569     public void visitCharLiteralNode(TinyLangAst tinyLangAst) {
570         String charIdentifier = tinyLangAst.getAssociatedNodeValue();
571         currentExpressionType = Type.CHAR;
572         currentExpressionValue = charIdentifier;
573     }
574 }

```

Listing 6.16: Interpreter

6.6 | [GitHub Repo](#)

 Repo Link [publicly available from 19th June 2022] : [TinyLang Repository Link](#)