Tinylang

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

Andimon

Contents

ffwwdd

0 | Introduction

0.1 | A tinylang program

```
fn Sq(x:float) -> float {
    return x*x;
}
fn XGreaterY(x:float , y:float) -> bool {
    let ans:bool=true ;
    if (y>x) {ans=false ; }
    return ans ;
}
// Same functionality as function above but using less code
fn XGreaterY_2 (x:float , y:float) -> bool {
    return x>y ;
}

fn AverageOfThree (x:float , y:float , z:float ) -> float {
    let total : flaot = x+y+z;
    return total/3;
}

/*
* Same functionality as function above but using less code .
* Note the use o f the brackets in the expression following
* the return statement .
*/

fn AverageOfThree_2 (x:float , y:float , z:float) -> float {
    return (x+y+z)/3 ;
}
//Execution (program entry point) starts at the first  statement
// that is not a function declaration .
```

```
let x : float = 2.4 ;
let y : float = Sq(2.5);
let z : float = Sq (x);
print y ; //6.25
print x * z ; //13.824
print XGreaterY (x , 2.3); // true
print XGreaterY 2(Sq(1.5),y); // false
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.

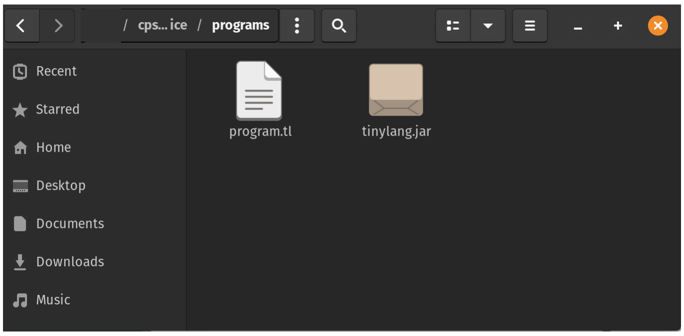


Figure 0.1: Program and compiler binary in same directory.

- Compile `program.tl` using tinylang by running command

  `java -jar tinylang program`

- We get a menu:

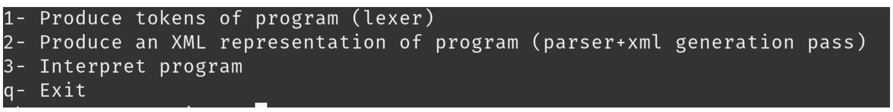


Figure 0.2: 3-option menu

```
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">
```

Figure 0.4: Option 2 : XML <-> AST

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

Figure 0.5: Option 3 : confirm that program is semantically correct + interpret

0.3 | Syntax rules of *TinyLang* in *EBNF*

```
<Letter> ::= [A-Za-z]
<Digit> ::= [0-9]
<Printable> ::= [\x20-\x7E]
<Type> ::= 'float'|'int'|'bool'|'çhar'
<BooleanLiteral> ::= 'true' | 'false'
<IntegerLiteral> ::= <Digit>{<Digit>}
<FloatLiteral> ::= <Digit>{<Digit>}'.'<Digit>{<Digit>}
<CharLiteral> ::= ‘'' <Printable’>''
<Literal> ::= <BooleanLiteral> | <IntegerLiteral> | <FloatLiteral> |
    <CharLiteral>
<Identifier> ::= ('_'|<Letter>){'_'|<Letter>|<Digit>}
<MultiplicativeOp> ::= '*'|'/'|'and'
<AdditiveOp> ::= '+'|'-'|'or'
<RelationOp> ::= '<'|'>'|'=='|'<='|'>='
<ActualParams> ::= <Expression> { ',' <Expression> }
<FunctionCall> ::= <Identifier>'('[<ActualParams>]')'
<SubExpression> ::= '('<Expression')'
<Unary> ::= ('+'|'-'|'not') <Expression>
<Factor> ::= <Literal> | <Identifier> | <FunctionCall> | <
   SubExpression> | <Unary>
<Term> ::= <Factor> {<MultiplicativeOp> <Factor>}
<SimpleExpr> ::= <Term> {<AdditiveOp> <Term>}
<Expression> ::= <SimpleExpr> {<RelationalOp> <SimpleExpr>}
<Assignment> ::= <Identifier>'='<Expresssion>
<VariableDecl> ::= 'let' <Identifier> ':' <Type> '=' <Expression>
<PrintStatement> ::= 'print' <Expression>
<RtrnStatement> ::= 'return' <Expression>
<IfStatement> ::= 'if' '('<Expression>')' <Block> ['else' <Block>]
<ForStatement> ::= 'for' '('[<VariableDecl>]';'<Expression>';'[<
   Assignment>]')' <Block>

<WhileStatement> ::= ‘’while ‘’(  Expression ‘)’  Block

<FormalParam> ::= <Identifier> ‘’: <Type>

<FormalParams> ::= <FormalParam> {','<FormalParam>}

<FunctionDecl> ::= 'fn' <Identifier> '(' [<FormalParams>] ')' '->'<
   Type><Block>

```

```
<Statement> ::= <VariableDecl>';'
              | <Assignment>';'
              | <PrintStatement>';'
              | <IfStatement>
              | <ForStatement>'
              | <WhileStatement>
              | <RtrnStatement>';'
              | <FunctionDecl>
              | <Block>
<Block> ::= '{' { <Statement> } '}'
<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** which takes a whole program as one string an breaks it down into a sequence tokens.
    - **parser** which takes all tokens produced by lexer and produce an abtract 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/vars etc.
    - **interpreter** used traverse the program AST and simulates 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 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`	`while`
`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):

<table>
<tr><th>Group and Lexeme(s)</th><th></th></tr>
<tr><td>TOK_MULTIPLICATIVE_OP</td><td>'*'|'/'|'and'</td></tr>
<tr><td>TOK_ADDITIVE_OP</td><td>'+'|'-'|'or'</td></tr>
<tr><td>TOK_RELATIONAL_OP</td><td>'<' | '>' | '==' | '=’|’<=’|’>=’!</td></tr>
</table>

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.

<table>
<tr><th>TokenType</th><th>Lexeme(s)</th></tr>
<tr><td>TOK_IDENTIFIER</td><td>('_' | <Letter>) ('_' | <Letter> | <Digit>)*</td></tr>
<tr><td>TOK_BOOLEAN_LITERAL</td><td>'true' | 'false'</td></tr>
<tr><td>TOK_INTEGER_LITERAL</td><td><Digit>{<Digit>}</td></tr>
<tr><td>TOK_FLOAT_LITERAL</td><td><Digit>{<Digit>}.<Digit>{<Digit>}</td></tr>
<tr><td>TOK_CHAR_LITERAL</td><td>'‘' <Printable> '’'</td></tr>
</table>

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 ll 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	[0x4a0x5a],[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`

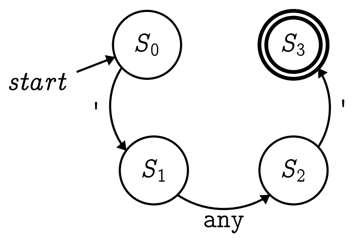


Figure 1.1: dfsa recognising lexemes in group *TOK_CHAR_LITERAL*

  - 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` :

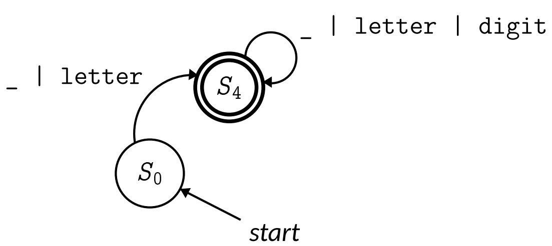


Figure 1.2: dfsa recognising lexemes in group *TOK_IDENTIFIER*

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 :

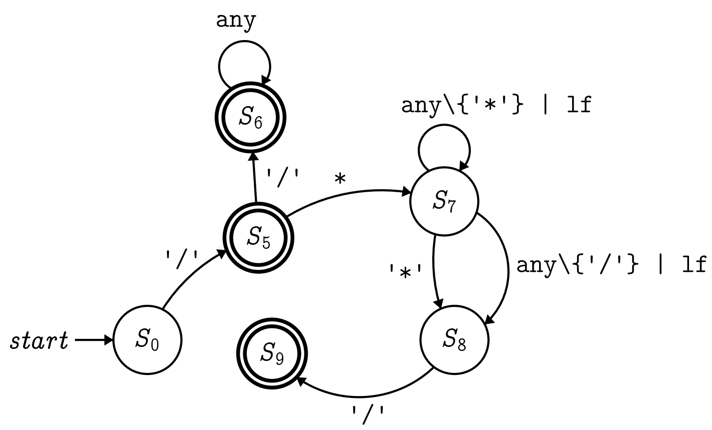


Figure 1.3: dfsa recognising lexemes in group *TOK_SKIP*

  - 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 determinstic).
  - 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` :

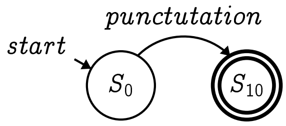


Figure 1.4: dfsa recognising lexemes in punctation groups

- 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`:

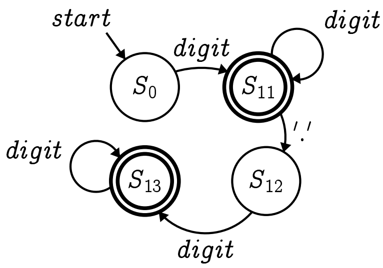


Figure 1.5: dfsa recognising lexemes in group *TOK_INT* and *TOK_FLOAT*

- 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`

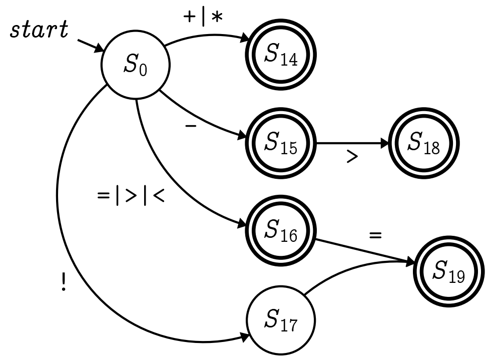


Figure 1.6: dfsa recognising lexemes in groups all operator groups and group *TOK_RIGHT_ARROW*

- 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

1.1.3 | tinylang Automaton

Merging all the sub-automata in figures 1.1, 1.2, 1.3, 1.4, 1.5 and 1.6 at the starting starting state we get the following 20-state dfsa that is able to deterministically recognise all the lexemes in tinylang and their respective groups (with the help of some extra helper function which will be discussed later).

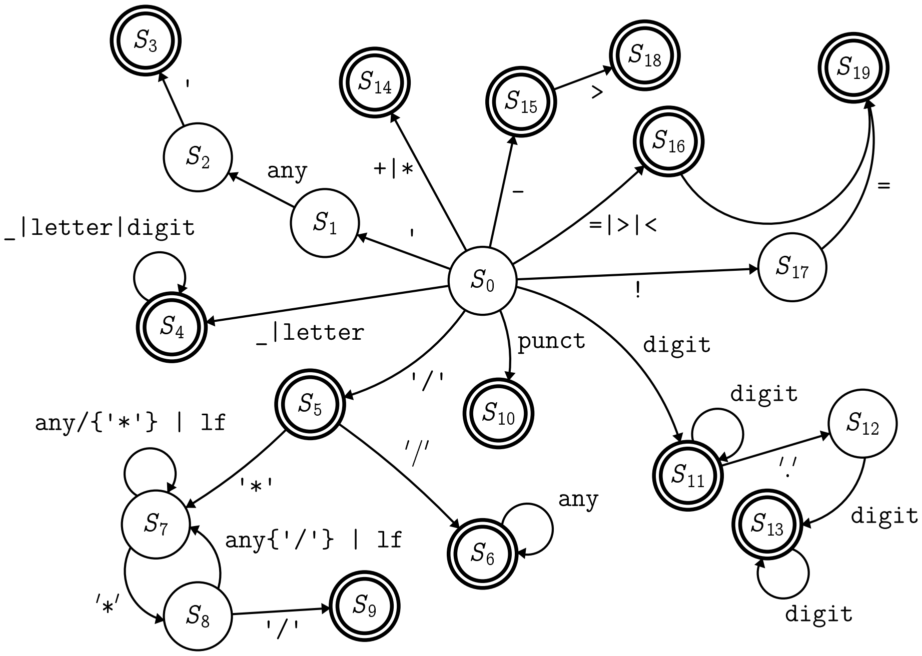


Figure 1.7: Automaton that captures all lexemes

Note: If for any given state and input the transition is not defined in the automaton shown in figure 1.7 then we assume that transition leads to an error state (this ensures completeness and makes it easier to write algorithms) . A tabular encoding of the automaton is given in figure 1.8. Lexemes leading to rejecting states are invalid and those leading to accepting states are in some group. The possible groups associated with each state is given by the following table:

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 S0 and SE respectively.

state \ input	letter	digit	_	/	*	<	>	+	-	=	!	.	’	punct	other_ printable	lf
S0	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 $\delta : S \times I \to S$ 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 **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 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)

```
int currentCharIndex <- 0;
int lineNumber <- 0;
String program;
List tokens;
//program is empty -> list is one just token EOF
if(program.length==0)
```

```
    tokens.add((EOF,""));
//otherwise if program not empty
while(currentCharIndex<program.length):
    token = getNextToken(program)
    //set line number
    token.setLineNumber(getLineNumber(tinyLangProgram))
    //if the next token is not a comment add it to list
    if token.type != TOK_SKIP:
        tokens.add(token)



/* Method getNextToken(program) includes includes
* the ideas (initialisation,scanning,rollback) of the table driven
* analysis algorithm by Cooper & Torczon
*/
Token getNextToken(program):
    // initialisation stage
    state = start_state
    lexeme = ""
    //stack of states
    Stack<States> stack
    //add sentinel state to stack
    stack.(bad_state)
    //clear white spaces and line feeds
    while(program.charAt(currentCharIndex)==space,\m , or tab):
        if(space):
            lineNumber++
        //increment char index
        currentCharIndex++
        //detect EOF
        if(currentCharIndex==program.length)
            //return EOF
            return new Token((TOK_EOF,""))
    //start scanning
    while(state != error_state and currentCharIndex<program.length)
        //obtain current char
        c = program.charAt(CurrentCharIndex)
        //append current char to lexeme
        lexeme.append(c)
        //if state is accepting clear stack
        if(state.IsAccepting):
           stack.clear
```

```
            stack.add(state)
        //obtain input category of current char (see classifier
    table)
        if(isLetter(c)):
            inputCat = letter
        else if(isDigit(c)):
            inputCat = digit
        else if(isUnderscore(c)):
            inputCat = underscore
        else if(isSlashDivide(c)):
            inputCat = slashDivide
        else if(isAsterisk(c)):
            inputCat = asterisk
        else if(isLessThan(c)):
            inputCat = lessThan
        else if(isGreaterThan(c)):
            inputCat = greaterThan
        else if(isPlus(c)):
            inputCat = plus
        else if(isHyphenMinus(c)):
            inputCat = hyphenMinus
        else if(isEqual(c)):
            inputCat = equal
        else if(isExclamationMark(c))
            inputCat = exclamationMark
        else if(isDot(c)):
            inputCat = dot
        else if(isSingleQuote(c)):
            inputCat = singleQuote
        else if(isPunct(c)):
            inputCat = punct
        else if(isOtherPrintable(c))
            inputCat = otherPrintable
        else if(isLineFeed(c)):
            inputCat = LineFeed
        else:
            throw exception char not recognised
        //transition function to get next state
        state = delta(state,inputCat)


    //rollback loop
    while(state!=error_state and currentCharIndex<tinyLangProgram.
```

```
  length):
        //pop state
        state = stack.pop()
        //truncate lexeme
        lexeme.truncate
        //move char index on stave backward
        currentCharIndex--
    //result
    if(state.getGroup(lexeme)==INVALID)
        throw exception invalid lexeme
    else
        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.

```
/*
 Testing
 the
 lexer
*/
let numru : float = (-2)+3.2;
//print numru
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

```
//a function must always return
fn forLoop()->bool{
  for(let i:int=1;i<=10;i=i+1){
    print i;
  }
  return true;
}
```

```
fn whileLoop()->bool{
  let i:int=1;
  while(i<=10){
    print i;
    i=i+1;
  }
  return false;
}
/*
a statement cannot be a function call (see EBNF)
 we assign an identifier bool x
*/
let x:bool=forLoop();
x=whileLoop();
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-Crated 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_DECLARTATION_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_BINBARY_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 a construct a tree class, `Tree`, where:

- Attributes:

```
//the type associated with each node
//e.g. AST_IDENTIFIER_NODE, AST_BINARY_OPERATOR_NODE etc.
NodeType nodeType;
//line number associated with each node
int lineNumber;
//value associated with each node (if any)
String lexeme;
Tree parent;
List<Tree> children;
```

- 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:

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

  - Add a new child node:
    - If child node has an associated value/lexeme, PSEUDOCODE:

```
 Tree addChild(NodeType nodeType, String lexeme,
String lineNumber):
     child = new Tree(nodeType,lexeme,lineNumber)
     child.parent=this
     this.children.add(child)
     return child
```

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

```
 Tree addChild(NodeType nodeType, String lineNumber
):
     child = new Tree(nodeType,lineNumber)
     child.parent=this
     this.children.add(child)
     return child
```

- Setters and getters.
  - Setters and getters where implemented for all attributes.

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

```
TINY_LANG_PROGRAM_NODE,
//NODES REPRESENTING STATEMENT TREES
AST_VARIABLE_DECLARATION_NODE,
AST_ASSIGNMENT_NODE,
AST_PRINT_STATEMENT_NODE,
AST_IF_STATEMENT_NODE,
AST_FOR_STATEMENT_NODE,
AST_WHILE_STATEMENT_NODE,
```

```
AST_RETURN_STATEMENT_NODE,
AST_FUNCTION_DECLARATION_NODE,
AST_BLOCK_NODE,
AST_ELSE_BLOCK_NODE,
//EXPRESSION NODES
AST_BINARY_OPERATOR_NODE,
AST_UNARY_OPERATOR_NODE,
AST_FUNCTION_CALL_NODE,
AST_IDENTIFIER_NODE,
//EXPRESSION NODES -> literal nodes
AST_BOOLEAN_LITERAL_NODE,
AST_INTEGER_LITERAL_NODE,
AST_FLOAT_LITERAL_NODE,
AST_CHAR_LITERAL_NODE,
//PARAMETER NODES
AST_ACTUAL_PARAMETERS_NODE,
AST_FORMAL_PARAMETERS_NODE,
AST_FORMAL_PARAMETER_NODE,
//TYPE NODE
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

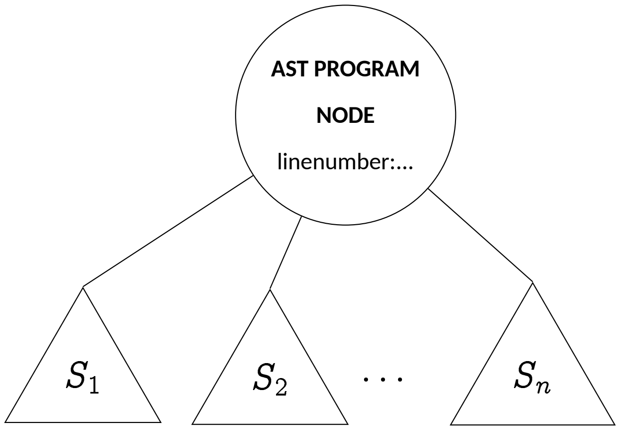


Figure 2.1: A program tree is a sequence of statement subtrees $S_1, S_2, \ldots, S_n$

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:

```
tree = new Tree(AST_PROGRAM_NODE,getCurrentToken.lineNumber)
//go through tokens until we reach EOF
while(getCurrenToken.type!=TOK_EOF):
    tree.addSubtree(parseStatement());
    //get next token (lookeahead for next statement)
    getNextToken()
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.

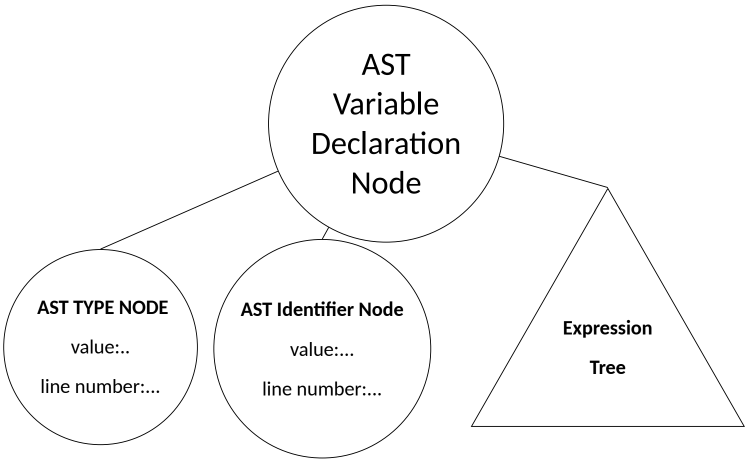


Figure 2.2: Statement tree: **Variable Declaration Statement**

```
tree = new Tree(AST_VARIABLE_DECLARATION_NODE,getCurrentToken().
    lineNumber)
//token that lead to this method should be let
if(getCurrentToken().type != TOK_LET):
     throw exception unexpected
//get next token (this updates the current token)
Token identifier = getNextToken()
//current token now should be of type identifier
if(getCurrentToken().type != TOK_IDENTIFFIER):
     throw exception unexpected
//get next token (this updates the current token)
getNextToken()
//current token now should be a colon
if(getCurrentToken().type != TOK_COLON):
     throw exception unexpected
//get next token (this updates the current token)
getNextToken()
//add type
tree.addSubtree(parseType());
//add identifier
tree.addChild(AST_IDENTIFIER_NODE,identifier.getLexeme(),identifier.
    getLineNumber())
//getNextToken()
tree.addSubtree(parseExpression())
return tree
```

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

Note in PSEUDOCODE shown in listing 2.3 their 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 differ according to current token type. The PSEUDOCODE is given in the listing below:

```
switch(getCurrentToken().getTokenType()):
    case TOK_BOOL_TYPE:
        return ast(AST_TYPE_NODE,BOOL,getCurrentToken().getLineNumber())
    case TOK_INT_TYPE:
        return ast(AST_TYPE_NODE,INT,getCurrentToken().getLineNumber())
    case TOK_FLOAT_TYPE
        return ast(AST_TYPE_NODE,FLOAT,getCurrentToken().getLineNumber())
    case TOK_CHAR_TYPE
        return ast(AST_TYPE_NODE,CHAR,getCurrentToken().getLineNumber())
    default:
        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.

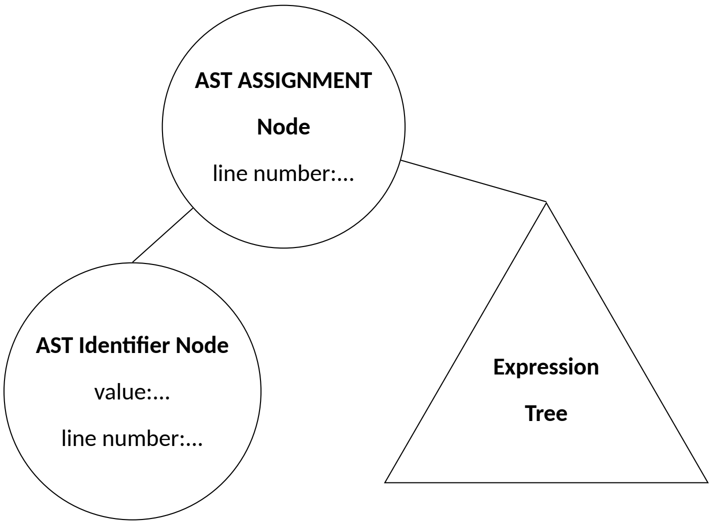


Figure 2.3: Statement tree: **Assignment Statement**

```
tree = new Tree(AST_ASSIGNMENT_NODE,getCurrentToken().lineNumber)
//token that lead to this method should be of type identifer
if(getCurrentToken().type != TOK_IDENTIFIER):
    throw exception unexpected
tree.addChild(AST_IDENTIFIER_NODE,getCurrentToken().getLexeme(),
   getCurrentToken().getLineNumber())
//get next token (this updates current token)
getNextToken()
//expect equal
if(getCurrentToken().type != TOK_EQUAL):
    throw exception unexpected
//get next token
getNextToken()
//expect expression
tree.addSubTree(parseExpression())
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.

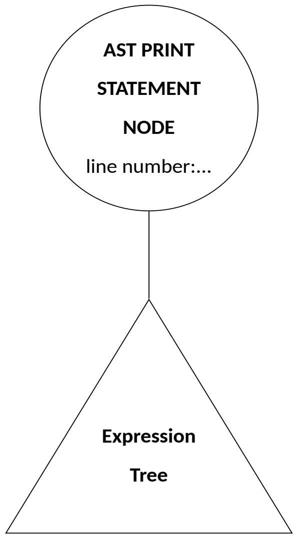


Figure 2.4: Statement tree: **PRINT STATEMENT**

```
tree = new Tree(AST_PRINT_STATEMENT_NODE,getCurrentToken().
    lineNumber)
//token that lead to this method should be of type TOK_PRINT
if(getCurrentToken().type != TOK_PRINT):
    throw exception unexpected
//get next token (this updates current token)
getNextToken()
//expect expression
tree.addSubTree(parseExpression())
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.

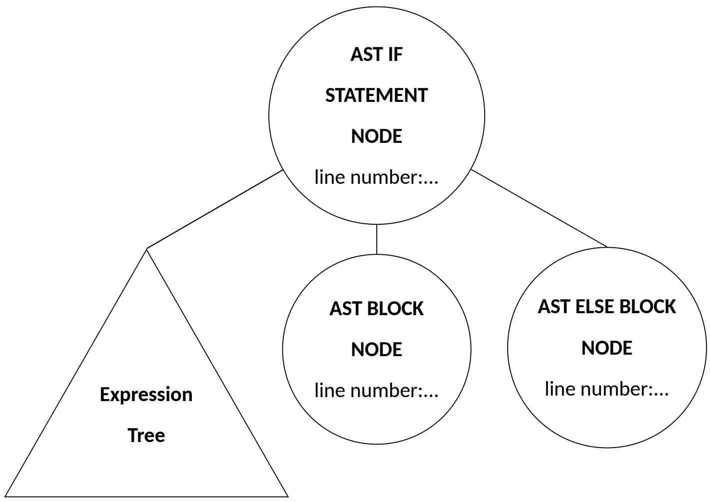


Figure 2.5: Statement tree: **IF STATEMENT**

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

```
tree = new Tree(AST_IF_STATEMENT_NODE,getCurrentToken().lineNumber)
//token that lead to this method should be of type TOK_IF
if(getCurrentToken().type != TOK_IF):
    throw exception unexpected
//get next token (this updates current token)
getNextToken()
//expect (
if(getCurrentToken().type != TOK_LEFT_ROUND_BRACKET):
    throw exception unexpected
//get next token (this updates current token)
getNextToken()
//add expression subtree
tree.addExpression(parseExpression());
//get next token( this updates current token)
getNextToken();
//expect )
if(getCurrentToken().type != TOK_RIGHT_ROUND_BRACKET):
    throw exception unexpected
//get next token( this updates current token)
getNextToken();
//parse block
tree.addSubtree(parseBlock())
```

```
//we check for an else condition (OPTIONAL)
//get next token( this updates current token)
getNextToken();

//if current token is else i.e. we have an else block node
if(getCurrentToken().type != TOK_ELSE):
    //get next token( this updates current token)
    getNextToken()
    //add else block
    tree.addSubtree(parseElseBlock())
//else no else block
else
   //get previous token( this updates current token)
    getPrevToken();
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.

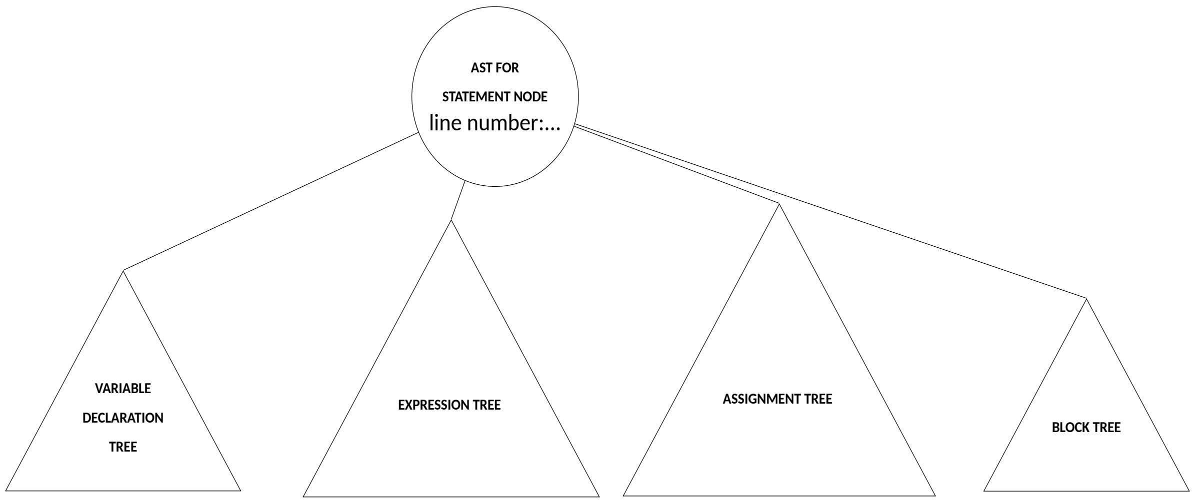


Figure 2.6: Statement tree: **FOR STATEMENT**

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.

```
tree = new Tree(AST_FOR_STATEMENT_NODE,getCurrentToken().lineNumber)
//token that lead to this method should be of type TOK_FOR
if(getCurrentToken().type != TOK_FOR):
    throw exception unexpected
//get next token (this updates current token)
getNextToken()
//expect (
if(getCurrentToken().type != TOK_LEFT_ROUND_BRACKET):
    throw exception unexpected
//get next token (this updates current token)
getNextToken()
//expect ; or a variable declaration (optional)
if(getCurrentToken().type != TOK_SEMICOLON):
    tree.addSubTree(parseVariableDeclartion())
    //get next token (this updates current token)
    getNextToken()
//expect ;
if(getCurrentToken().type != TOK_SEMICOLON):
    throw exception unexpected
//get next token (this updates current token)
```

```
getNextToken()
//expect expression
tree.addSubtree(parseExpression())
//get next token (this updates current token)
getNextToken()
//expect ;
if(getCurrentToken().type != TOK_SEMICOLON):
    throw exception unexpected

//expect ) or assignment (optional)
if(getCurrentToken().type != TOK_RIGHT_ROUND_BRACKET):
    tree.addSubtree(parseAssigment())
    getNextToken()

//expect block
tree.addSubtree(parseBlock())
//return tree
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.

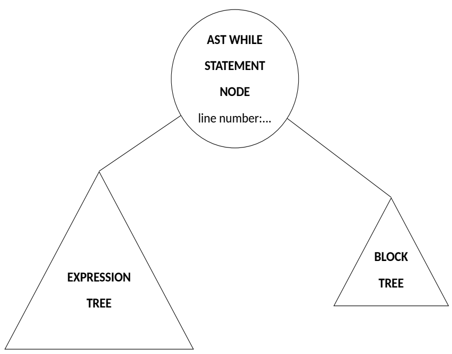


Figure 2.7: Statement tree: **WHILE LOOP**

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

```
tree = new Tree(AST_FOR_STATEMENT_NODE,getCurrentToken().lineNumber)
//token that lead to this method should be of type TOK_WHILE
if(getCurrentToken().type != TOK_WHILE):
    throw exception unexpected
//get next token (this updates current token)
getNextToken()
//expect (
if(getCurrentToken().type != TOK_LEFT_ROUND_BRACKET):
    throw exception unexpected
//get next token (this updates current token)
getNextToken()
//expect expression
tree.addSubtree(parseExpression())
//get next token (this updates current token)
getNextToken()
//expect )
if(getCurrentToken().type != TOK_RIGHT_ROUND_BRACKET):
    throw exception unexpected
//get next token (this updates current token)
getNextToken()
//expext block
tree.addSubtree(parseBlock())
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 analogolous 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.

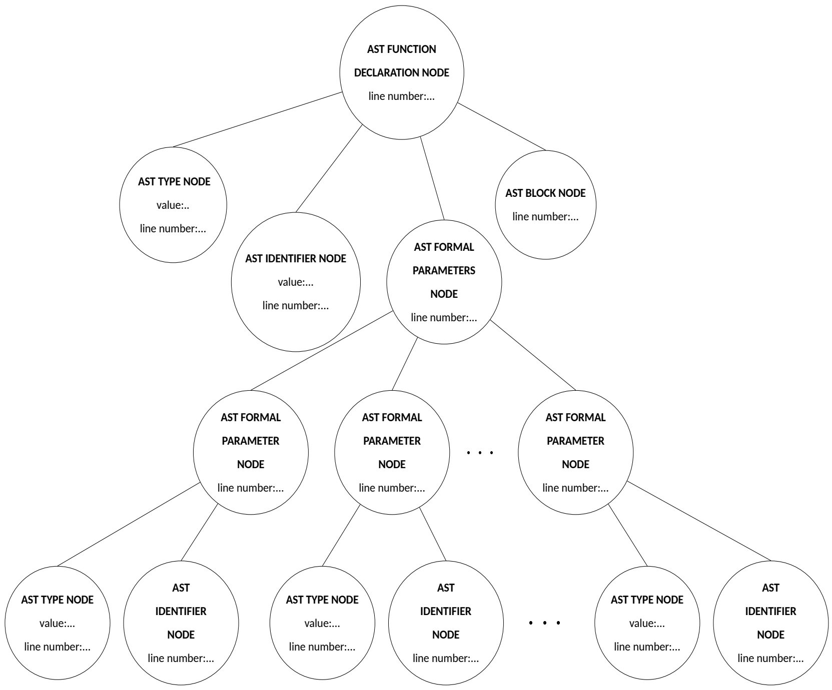


Figure 2.8: Statement tree: **FUNCTION DECLARATION**

```
tree = new Tree(AST_FUNCTION_DECLARATION_NODE,getCurrentToken().
   lineNumber)
//token that lead to this method should be of type TOK_FN
if(getCurrentToken().type != TOK_FN):
    throw exception unexpected
//get next token (this updates the current token)
Token identifier = getNextToken()
//expect identifier
Token identifier
if getCurrentToken().type ==TOK_IDENTIFIER
    identifier=getCurrentToken()
else
    throw exception unexpected
//get next token (this updates the current token)
getNextToken()
//expect (
if(getCurrentToken().type != TOK_LEFT_ROUND_BRACKET):
    throw exception unexpected
//get next token (this updates the current token)
getNextToken()
//expect 0 or more formal parameters
```

```
Tree formalParamsSubtree
//if next token is not a right round bracket we have formal paramaters
if(getCurrentToken().type != TOK_RIGHT_ROUND_BRACKET):
    formalParamsSubtree = parseFormalParams()
    //get next token (this updates the current token)
    getNextToken()
//else just add a formal parameters node with no children
else
    formalParamsSubtree = new Tree(AST_FORMAL_PARAMETERS_NODE, getCurrentToken().lineNumber)
//expect )
if(getCurrentToken().type != TOK_RIGHT_ROUND_BRACKET):
    throw exception unexpected
//get next token (this updates the current token)
getNextToken()
//expect ->
if(getCurrentToken().type != TOK_RIGHT_ARROW):
    throw exception unexpected
 //get next token (this updates the current token)
getNextToken()
//expect type
Tree typeSubtree = parseType()
//get next token (this updates the current token)
getNextToken()
//expect block
Tree blockSubtree = parseSubtree()
//add type subtree to function declaration tree
tree.addSubtree(typeSubtree)
//add identifier node to function declartion tree
tree.addChild(AST_IDENTIFIER_NODE,identifier.lexeme,identifier.lineNumber)
//add formal params subtree to function declaration tree
tree.addSubtree(formalParamsSubtree)
//add block subtree to function declaration tree
tree.addSubtree(blockSubtree)
//return function declaration tree
return tree
```

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

Note in PSEUDOCODE shown in listing 2.10 their are calls to 3 other parsing

methods: `parseFormalParams()`, `parseType()`, ,`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.

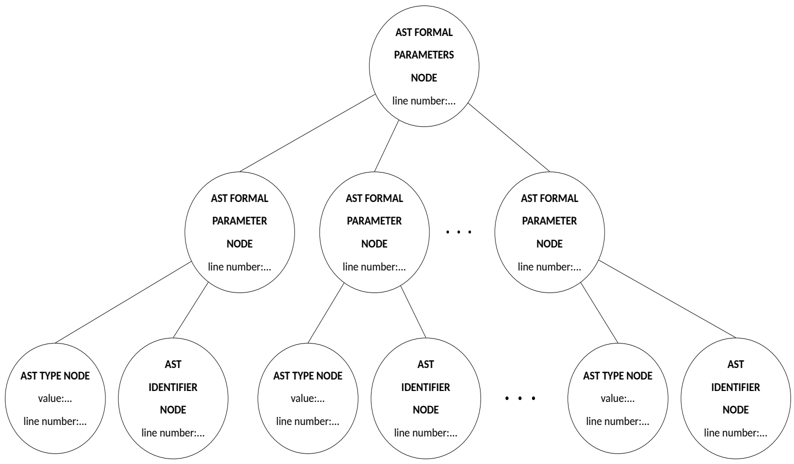


Figure 2.9: Formal parameters subtree

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.

```
tree = new Tree(AST_FORMAL_PARAMS_NODE,getCurrentToken().lineNumber)
//add formal param
tree.addSubtree(parseFormalParam())
//get next token (this updates the current token)
getNextToken()
//each formal parameter is seperated by a comma
while(getCurrentToken().tokenType==TOK_COMMA)
    //get next token (this updates the current token)
    getNextToken()
    //parse next formal parameter
    tree.addSubtree(parseFormalParam())
    //get next token (this updates the current token)
    getNextToken()
//get prev token (this updates the current token)
getPrevToken();
```

```
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.

```
tree = new Tree(AST_FORMAL_PARAMETER_NODE,getCurrentToken().
    lineNumber)
//expect identifier
if getCurrentToken().type ==TOK_IDENTIFIER
    identifier=getCurrentToken()
//get a hold of identifier node
Token identifier = getCurrentToken();
//get next token (this updates current token)
getNextToken();
//expect :
if getCurrentToken().type ==TOK_COLON
    identifier=getCurrentToken()
//get next token (this updates current token)
getNextToken();
//add type subtree
tree.addSubtree(parseType())
//add identifier
tree.addChild(AST_IDENTIFIER,identifier.lexeme,identifier.lineNumber
    )
//return tree
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.

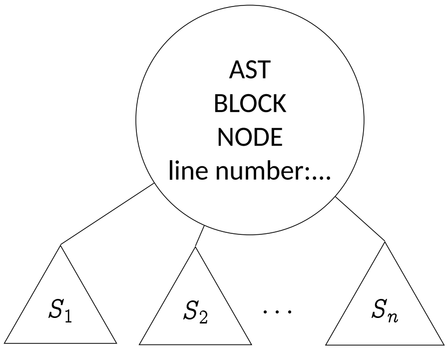


Figure 2.10: Statement tree: Block is a sequence of statements $S_1, S_2, \ldots, S_n$

```
tree = new Tree(AST_BLOCK_NODE,getCurrentToken().lineNumber)
//token that lead to this method should be of type {
if(getCurrentToken().type != TOK_LEFT_CURLY_BRACKET):
    throw exception unexpected
//get next token (this updates current token)
getNextToken()
//we may have one or more statements block ends using }
while(getCurrentToken().getTokenType()!=TokenType.TOK_RIGHT_CURLY_BRACKET and getCurrentToken().getTokenType()!=TokenType.TOK_EOF ):
    tree.addSubtree(parseStatement())
    getNextToken()

//current token should be }
if(getCurrentToken().type != TOK_LEFT_RIGHT_BRACKET):
    throw exception unexpected
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 leave nodes are values to the corresponding binary operator.

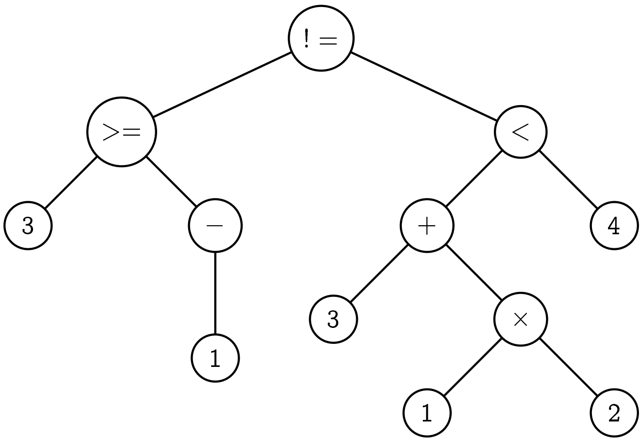


Figure 2.11: Example of an **expression tree**

An inorder traversal of the expression tree shown in figure 2.11 gives us the expression $((3) >= (-1))! = (((3) + ((1) \times (2))) < (4))$ 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
$se_1\ relop_1\ se_2\ relop_2\ \ldots\ relop_{n-1} se_n \equiv$
$se_1\ relop_1\ (se_2\ relop_2\ \ldots (se_{n-1}\ relop_{n-1} se_n))$ (se denotes simple expression) then a tree representing this expression would look like the following:

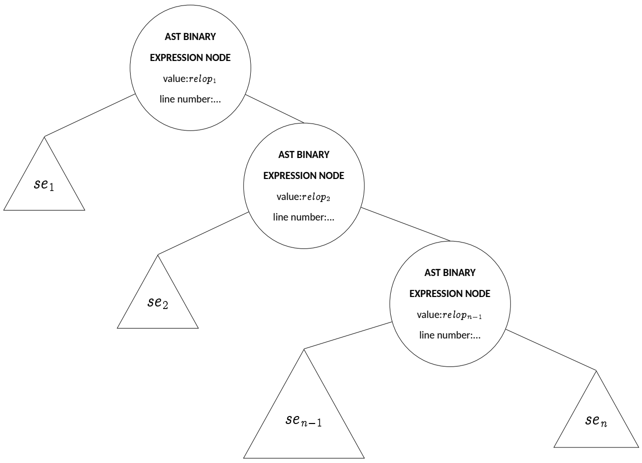
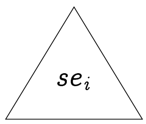


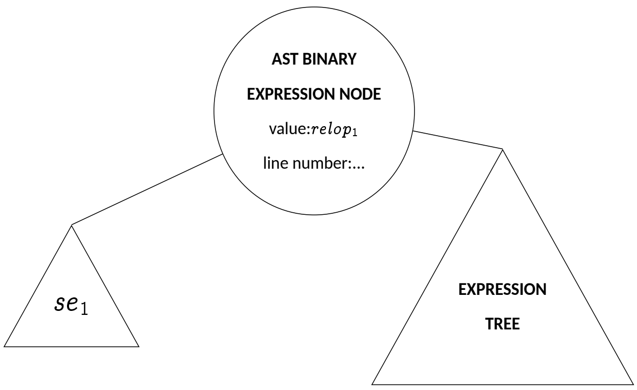
Figure 2.12: Expression is a sequence of simple expressions separated by an operator of type *TOK_RELATIONAL_OP*

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 $< Expression >$ and building an expression tree recursively is described in the following PSEUDOCODE.

```
// base case
Tree leftOperand = parseSimpleExpression()
//get next token (this updates current token)
getNextToken()
//if we have a relop run recursive case
if(getCurrentToken().type != TOK_RELATIONAL_OP):
    // build a binary tree value -> lexeme (representing the
   operator)
    tree = new Tree(AST_BINARY_OPERATOR_NODE,getCurrentToken().
   lexeme,getCurrentToken().lineNumber)
    //add left operand of binary operator
    tree.addSubtree(leftOperand)
    //recursive step
    tree.addSubtree(parseExpression())
    return tree
return leftOperand
```

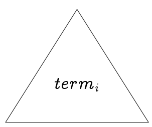
Listing 2.14: PSEUDOCODE : parsing *<Expression>* and building an expression tree (*parseExpression()*)

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

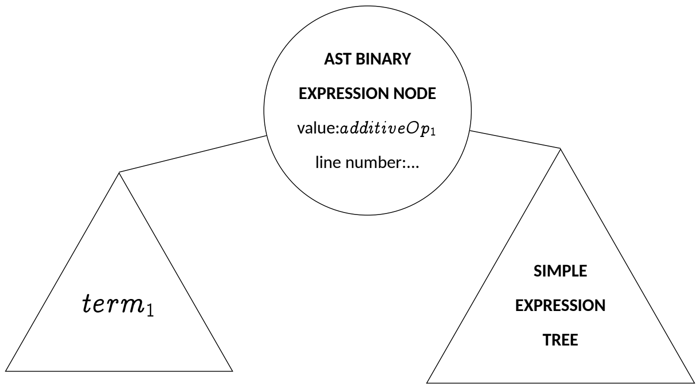
2.3.3.2 | <SimpleExpression>

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 $term_1\ additiveOp_1\ term_2\ additiveOp_2\ \ldots\ additiveOp_{n-1} term_n \equiv$ $term_1\ additiveOp_1\ (term_2\ additiveOp_2\ \ldots (term_{n-1}\ additiveOp_{n-1} term_n))$ 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 $< SimpleExpression >$ and building an expression tree recursively is described in the following pseudocode.

```
// base case
Tree leftOperand = parseTerm()
//get next token (this updates current token)
getNextToken()
//if we have a relop run recursive case
if(getCurrentToken().type != TOK_ADDITIVE_OP):
    // build a binary tree value -> lexeme (representing the operator)
    tree = new Tree(AST_BINARY_OPERATOR_NODE,getCurrentToken().lexeme,getCurrentToken().lineNumber)
    //add left operand of binary operator
    tree.addSubtree(leftOperand)
    //recursive step
    tree.addSubtree(parseSimpleExpression())
    return tree
return leftOperand
```

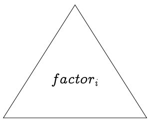
Listing 2.15: parsing <*SimpleExpression*> and building an expression tree (*parseSimpleExpression()*)

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.

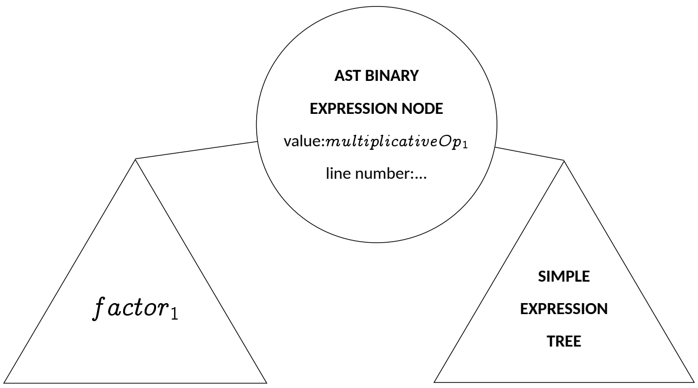
2.3.3.3 | <Term>

Term is a sequence of one or more factors separated by a multiplicative operator (see section 0.3). For example suppose we have $factor_1\ multiplicativeOp_1\ term_2\ multiplicativeOp_2\ \ldots\ multiplicativeOp_{n-1} term$ $factor_1\ additiveOp_1\ (factor_2\ multiplicativeOp_2\ \ldots (factor_{n-1}\ multiplicativeOp_{n-}$ 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 $< Term >$ and building an expression tree recursively is described in the following pseudocode.

```
// base case
Tree leftOperand = parseFactor()
//get next token (this updates current token)
getNextToken()
//if we have a relop run recursive case
if(getCurrentToken().type != TOK_MULTIPLICATIVE_OP):
    // build a binary tree value -> lexeme (representing the
   operator)
```

```
    tree = new Tree(AST_BINARY_OPERATOR_NODE,getCurrentToken().
   lexeme,getCurrentToken().lineNumber)
    //add left operand of binary operator
    tree.addSubtree(leftOperand)
    //recursive step
    tree.addSubtree(parseFactor())
    return tree
return leftOperand
```

Listing 2.16: parsing *<Term>* 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>

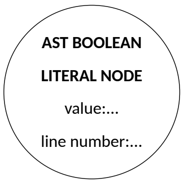
A factor represents an operand of the binary operator. Now an operand may take different forms (see section 0.3) namely:

- 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:

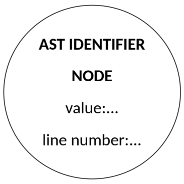
- `TOK_BOOLEAN_LITERAL` return the following node



  - 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`

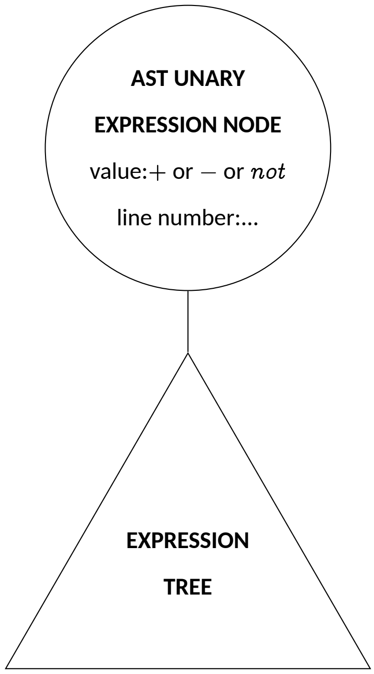


Figure 2.13: **Unary expression tree**

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 PSEUDOCECODE of given below.

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

```
//an left round bracket led to this parsing method
if(getCurrentToken ().type != TOK_LEFT_ROUND_BRACKET):
    throw exception unexpected
//get next token(this updates current token)
tree = parseExpression()
//get next token
getNextToken();
//we expect a right round bracket
if(getCurrentToken().type=TOK_RIGHT_ROUND_BRACKET)
    throw exception unexpected
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.

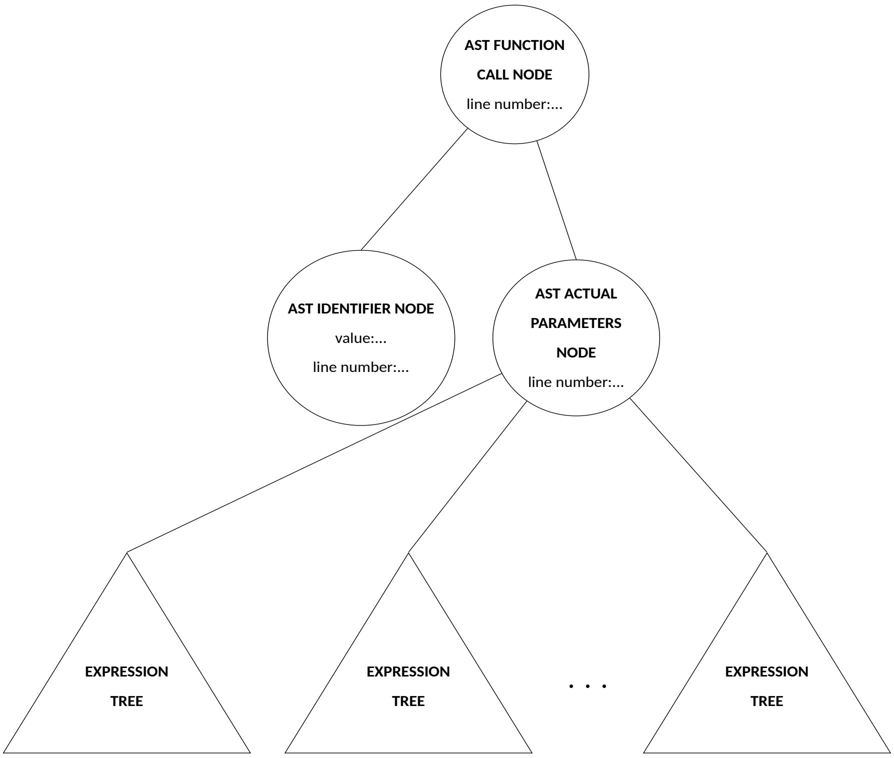


Figure 2.14: Factor tree: **FUNCTION CALL**

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.

```
tree = new Tree( AST_FUNCTION_CALL_NODE , getCurrentToken().
    lineNumber )
// token that lead to this method should be of type identifier
if( getCurrentToken ().type != TOK_IDENTIFIER ):
     throw exception unexpected
//add identifier node
tree.addChild(AST_IDENTIFIER_NODE,getCurrentToken().lexeme,
    getCurrentToken().lineNumber)
// get next token (this updates current token)
getNextToken ()
// next token should be of type (
if(getCurrentToken ().type != TOK_LEFT_ROUND_BRACKET ):
     throw exception unexpected
```

```
// get next token (this updates current token)
getNextToken ()
//if the next token is not a round bracket -> we should have one or
   more actual parameters
if(getCurrentToken ().type != TOK_RIGHT_ROUND_BRACKET ):
    tree.addSubtree(parseActualParams())
    // get next token (this updates current token)
    getNextToken()
//else we add a parameter node with no children
else
    tree.addChild(AST_ACTUAL_PARAMETER_NODE,getCurrenToken().lexeme.
   getCurrentToken().linenumber)
// get next token (this updates current token)
getNextToken ()
//expect right round bracket
if(getCurrentToken ().type != TOK_RIGHT_ROUND_BRACKET):
    throw exception unexpected
//return tree
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 /

```
Tree tree = new TinyLangAst(AST_ACTUAL_PARAMETERS,getCurrentToken().
   lineNumber)
//add expression subtree
tree addSubtree(parseExpression)
//get next token (this updates current token)
getNextToken()
//we start checking if we have commas since this implies that we
   have more actual paremeters
while(getCurrenToken().type==TOK_COMMA and getCurrenToken().type!=
   TOK_EOF):
    //get next token (this updates current token)
    getNextToken()
    //add next expression subtree
    actualParamsTree.addSubtree(parseExpression())
```

```
    //get next token (this updates current token)
    getNextToken()
//move back one token (this updates current token)
getPrevToken()
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.

```
fn Sq (x:float) -> float {
    return x*x ;
}
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

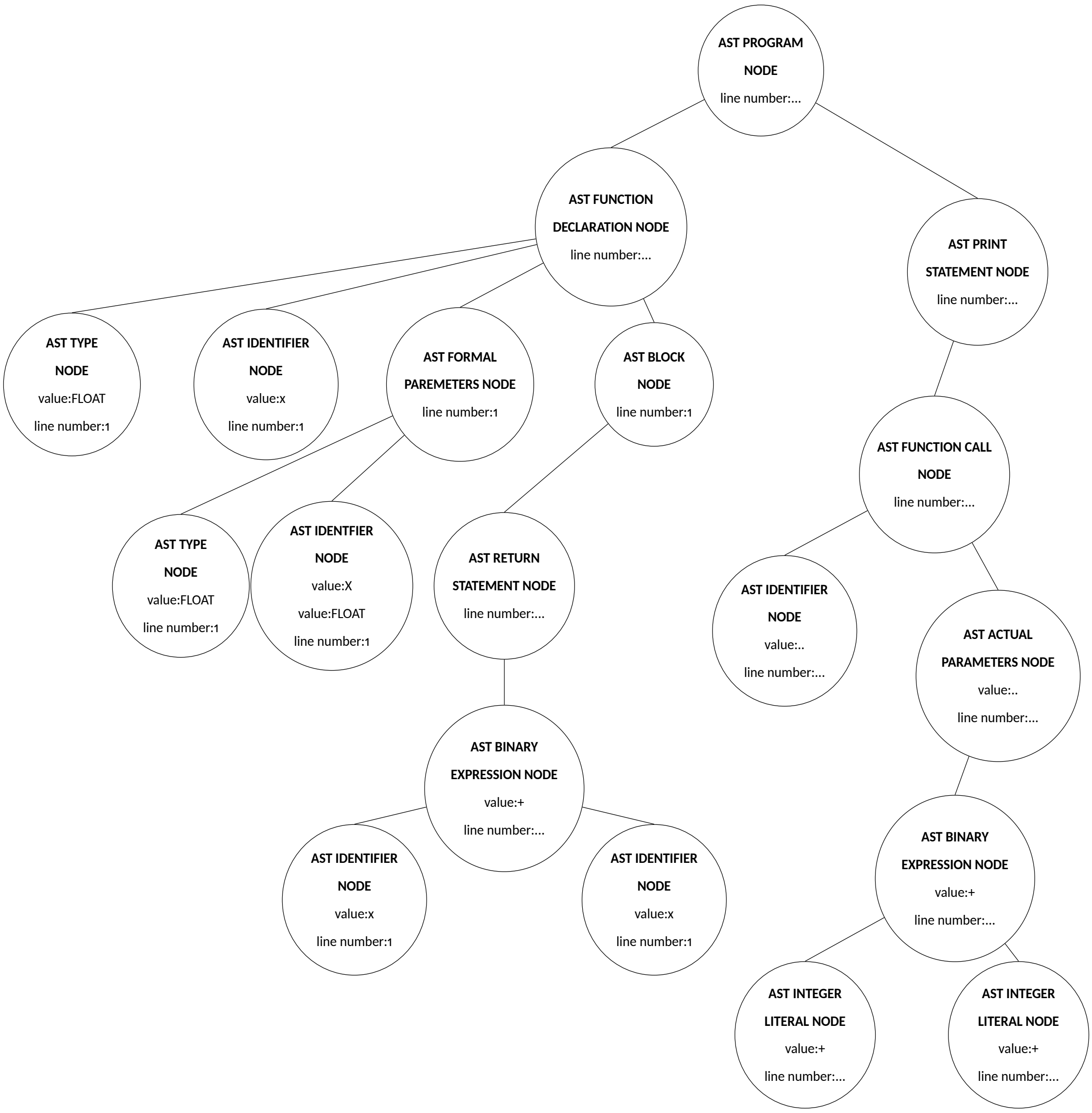


Figure 2.15: AST generated after parsing program (listing 2.21)

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
```

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 except 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.

```
xception in thread "main" java.lang.RuntimeException:   in line 13. No statemen
 begins with bool
```

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 `Vistor` 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.

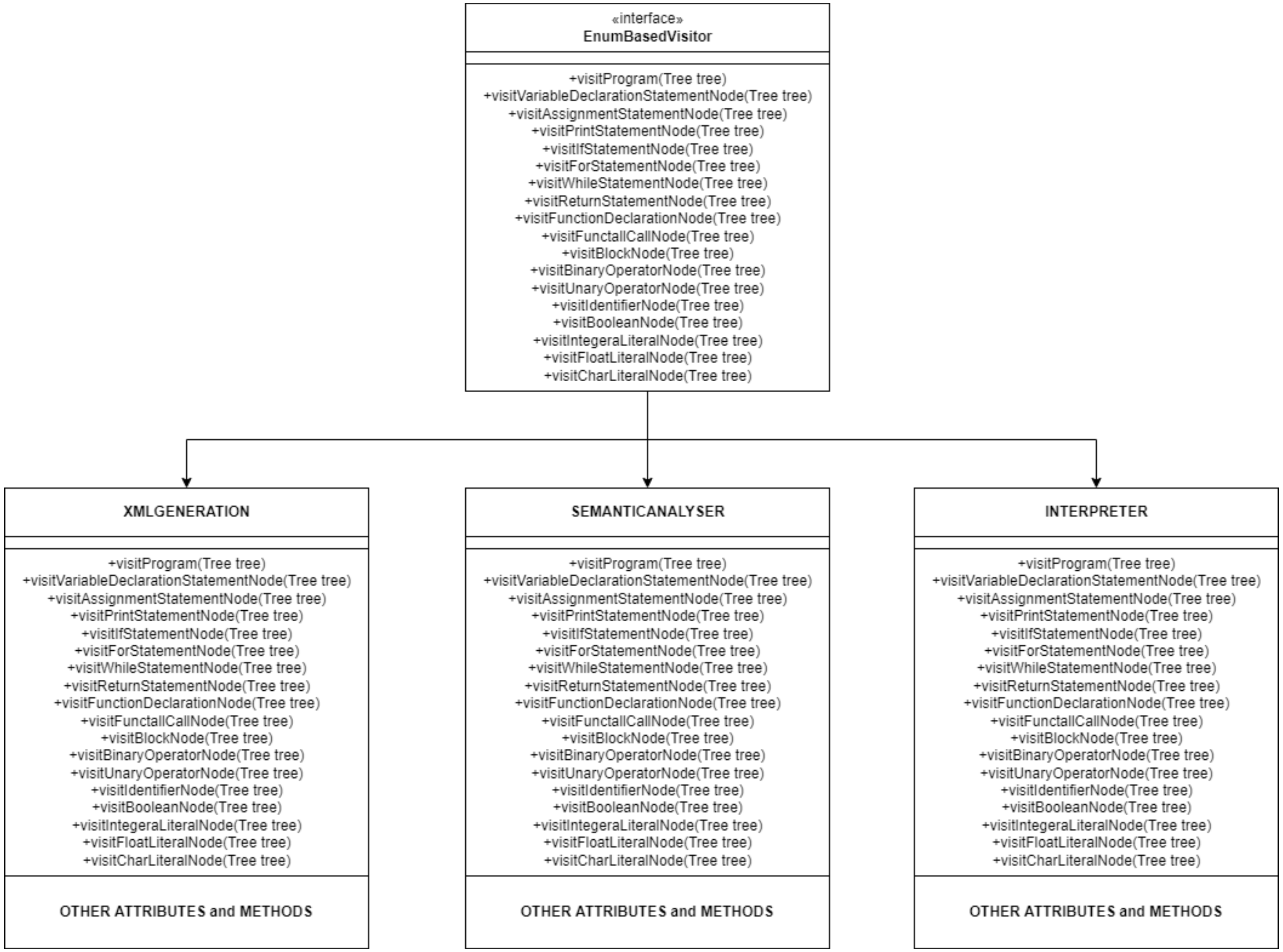


Figure 3.1: *XmlGeneration,Semantic Analyser and Interpreter* are concrete implementations of interface EnumVisitor.

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.

```
<TinyLangProgram>
    <function declaration>
        <id type="FLOAT"> Sq <\id>
        <parameters>
            <id type="FLOAT"> x <\id type>
        <\parameters>
        <block>
          <return statement>
            <binary Op>
                <id type="FLOAT"> x <\id type>
                <id type="FLOAT"> x <\id type>
            <\binay Op>
         <\return statement>
        <\block>
    <\function declaration>
    <print>
        <function call>
            <id>Sq<\id>
            <parameters>
                <actual parameter>
                    <id>x<\id>
                <\actual parameter>
                <actual parameter>
                    <id>x<\id>
                <\actual parameter>
            <\parameters
        <\function call>
    <\print>
<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`

```
getCurrentIndentation():
    String indentation = ""
    for(i<0;i<this.identation;i++):
        indentation+="     "
        return indetation
```

Listing 3.2: Get current indentation

- Constructor is given by:

```
XmlGeneration(Tree tree):
    //visit the whole abstract syntax tree is
    //equivalent to visiting a program
    visitProgram(tree)
```

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.

```
xmlRepresentation+=getCurrentIndentation()+"<TinyLangProgram>\n"
//we indent next body
indentation++
for(child : currentTree.children)
    visitStatement(child)
//unindent (tags attain same level of indentaion)
indentation--
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.

```
If tree/node is of type:
AST_VARIABLE_DECLARTION_NODE -call-> visitVariableDeclarationNode(
   tree)
AST_ASSIGNMENT_NODE -call-> visitAssignmentNode(tree)
AST_PRINT_STATEMENT_NODE -call-> visitPrintStatementNode(tree)
AST_IF_STATEMENT_NODE -call-> visitIfStatementNode(tree)
AST_FOR_STATEMENT_NODE -call-> visitForStatementNode(tree)
AST_WHILE_STATEMENT_NODE -call-> visitWhileStatementNode(tree)
AST_RETURN_STATEMENT_NODE -call-> visitReturnStatementNode(tree)
AST_FUNCTION_DECLARATION_NODE -call-> visitFunctionDeclarationNode(
   tree)
AST_BLOCK_NODE -call-> visitBlock(tree)
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 delcaration 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:

```
<variable declaration>
    <id type=TYPE> variable name<\id>
```

```
    .... | some
     .... | expression
    .... | body
<\variabld declaration>
```

Listing 3.6: XML of variable declaration statement subtree

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

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

Listing 3.7: PSEUDCODE 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 the 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:

```
AST_BINARY_OPERATOR_NODE -call-> visitBinaryOperatorNode(tree)
AST_UNARY_OPERATOR_NODE -call-> visitUnaryOperatorNode(tree)
AST_BOOLEAN_LITERAL_NODE -call-> visitBooleanLiteralNode(tree)
AST_INTEGER_LITERAL_NODE -call-> visitIntegerLiteralNode(tree)
AST_FLOAT_LITERAL_NODE -call-> visitFloatLiteralNode(tree)
AST_CHAR_LITERAL_NODE -call-> visitCharLiteralNode(tree)
AST_IDENTIFIER_NODE -call-> visitIdentifierNode(tree)
AST_FUNCTION_CALL_NODE -call-> visitFunctionCallNode(tree)
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

```
xmlRepresentation += getCurrentIndentation ()+"<binary Op="+tree.
    value+">\n"
//check for 2 children (we expect 2 children)
if(tree.getChildren().size!=2)
    throw unexpected
//we indent for next body
indentation ++
//visit first child (an expression)
visitExpression(tree.getChildren().get(0));
//visit first child (an expression)
visitExpression(tree.getChildren().get(1));
// unindent (tags attain same level of indentation )
indentation --
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

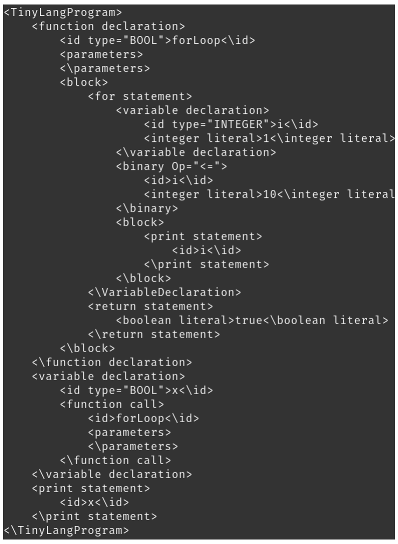
All the design decisions mentioned in 3.2 are implemented as shown in listing 6.11

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

`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.

```
//create new scope
st.push()
//traverse the program
visitProgram(tree)
//end scope
st.pop
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.

```
//create new scope
st.push();
```

```

-> add parameters of functions if any in scope
-> clear currentFunctionParameters
-> visit all statements in block

//end scope
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 $Identifier \to Type$. 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:

```
{
    let a : float = 5;
    print a;
    {
        let a : char = 'a';
        print a;
    }
}
```

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)`.

```
//second child of the tree corresponds to identifier
Tree identifier = tree.getChildren().get(1)
```

```
//check if name of identifier is already declared in current scope
if(st.isVariableNameBinded(identifier.getAssociatedNodeValue())==
    true)
    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`.

```
//check if a function is declared within a scope
boolean isFunctionAlreadyDefined(FunctionSignature signature):
    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

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

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

```
st.insertFunctionDeclaration(new FunctionSignature(functionName,
   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 an 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 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.

```
Base Case (trivial case) tree represents a return statement :
if(tree.type=AST_RETURN_STATEMENT):
    return true
//(Recursive case) if statement is a block we check if one of the statements inside the
//block returns
if(tree.type=AST_BLOCK_NODE):
    for(Tree statement : tree.getChildren()):
        //if one statement within block
        //returns the whole block returns
        if(returns(statement)):
            return true

)
//(Recursive case) if statement is an else block we check if one of the statements inside
// the else block returns
if(tree.type=AST_ELSE_BLOCK_NODE):
    for(Tree statement : tree.getChildren()):
        //if one statement within block
        //returns the whole block returns
        if(returns(statement)):
            return true
)

//(Recursive case) if statement is an if statement block has an else block
```

```
(i.e. if statement tree has 3 children)
//and check if the block and else block contains
//a returning function
if(tree.type=AST_IF_STATEMENT_NODE):
    if(tree.getChildren().size=3):
    //check if children block treee and else block tree returns
        return returns(tree.getChildren().get(1) and returns(tree.
   getChildren().get(2))
//(Recursive case) check that block inside for/while loops is
   returning
if(tree.type=AST_FOR_STATEMENT_NODE):
    //block statement is last child
    //a for statement can have different amount
    //of children
    return returns(tree.getChildren().get(tree.getChildren().size-1)
   )
if(tree.type=AST_FOR_STATEMENT_NODE):
    //block statement is second child
    return returns(tree.getChildren().get(1))
// (base case) otherwise in all other cases
else:
    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).

```
//a function must always return,this function is not guaranteed to return
fn notGuranteedToReturn(x:char)->bool{
    if(x=='a'){ return true; }
    else{
        //conditional branching
        if(x=='b'){ return true; }
    else{
            if(x=='c'){ return true; }
            else{
           //no return statement in this block
           print 'c';
            }
        }
    }
}
```

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

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

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.

```
fn notNice(x:char)->bool{
  let x:bool=false;
  //error variable x already declared
  let x:int=5;
  print x;
}
```

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)

```
fn notNice(x:char)->bool{
  fn notNice(x:char)->int{
    return 0;
  }
  return notNice('a');
}
```

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

```
fn nice(x:int)->float{
  fn return2()->int{
    return 2;
  }
  return x+return2();
}
fn nice(x:float)->float{
  fn return3()->int{
```

```
    return 3;
  }
  return x+return3();
}
print nice(1);
print nice(1.0);
```

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 also 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<Strign>> 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 currentExpresionType` 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 `currentExpresionValue` 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 `currentExpresionValue` 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 `currentExpresionValue` 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(currentExpresssionValue))`. Else if the current expression is of type bool we check i f 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 `paramaterTypes` 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,par`
  - 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.insertVariableDeclartaion(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 visit the expression (2nd child) to update `currentExpressionType` and `currentExpresssionValue`. Then we search the inner most scope where the variable is declared and update the varaible value by calling instance method `addVariableValue(varName, currentExpressionValue)` which updates the value of 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 exprssion `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 update`currentExpressionValue` 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.

```
while(currentExpressionValue.equals("true")){
    visitBlockNode(block)
    //update current expression value
    visitExpression(expression);
}
```

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.

```
while(currentExpressionValue="true"){
    //4th child correspond to block node
    visitBlockNode(4th child)
    //visit assignment node (updation)
    visitAssignment(third child)
    //update truth value
    visitExpression(2nd child)
}
```

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.

```
 //first child corresponds to expression -> update current
expression value
 visitExpression(first child)
 while(currentExpressionValue.equals("true")):
     visitBlockNode(block);
     visitExpression(expression);
```

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 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 `Intepreter` 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.

```
print 'H';
print 'e';
print 'l';
print 'l';
print 'o';
print 'W';
print 'o';
print 'r';
print 'l';
print 'd';
```

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**,....)

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

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.

```
{
    let a:int=5;
    print a;
    {
          let a:int=6;
          print 6;
    {
         let a:int=7;
         print 7;
    }
    }
}
```

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 $\mathbb{N}$.

```
fn add(a:int,b:int)->int{
    if(b==0){return a;}
    else{
      return add(a+1,b-1);
    }
}
//reuse add function
fn multiply(a:int,b:int)->int{
    if(b==0){return 0;}
    else{
      return a+multiply(a,(b-1));
    }
}
//a^b, reuse multiply function
fn power(a:int,b:int)->int{
    if(b==0){return 1;}
    else{
      if(b==1){return a;} else{
      return a*power(a,b-1);
```

```
      }
    }
}
print add(5,3);
print multiply(5,3);
print power(5,3);
```

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

```
fn add(a:int,b:int)->int{
    if(b==0){return a;}
    else{
      return add(a+1,b-1);
    }
}
//reuse add function
fn multiply(a:int,b:int)->int{
    if(b==0){return 0;}
    else{
      return a+multiply(a,(b-1));
    }
}
//a^b, reuse multiply function
fn power(a:int,b:int)->int{
    if(b==0){return 1;}
    else{
      if(b==1){return a;} else{
      return a*power(a,b-1);
      }
    }
}
let sum:int=0;
for(let i:int=0;i<=5;i=i+1){
  let a:int = power(i,2);

  let b:int = multiply(i,2);
  let c:int = add(b,2);
  let d:int = add(a,c);
    print d;
  sum=sum+d;

}
```

```
print sum;
```

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

```java
package tinylanglexer;
public enum StateType {
  ACCEPTING,
  REJECTING
}
```

Listing 6.1: State type

```java
package tinylanglexer;
public enum State {
  /**
   * The starting state of representing TinyLang's grammar.
   */
  STARTING_STATE (StateType.REJECTING),
  STATE_1 (StateType.REJECTING),
  STATE_2 (StateType.REJECTING),
  /* Lexemes leading to STATE_3 -> Lexeme of type TOK_CHAR_LITERAL */
  STATE_3 (StateType.ACCEPTING),
  /* Lexemes leading to STATE_4 -> Lexeme of type TOK_IDENTIFIER_LITERAL or other KEYWORD type
     */
  STATE_4 (StateType.ACCEPTING),
  /* Lexemes leading to STATE_5 -> Lexeme of type TOK_ MULTIPLICATIVE_OP[*/
  STATE_5 (StateType.ACCEPTING),
  /* Lexemes leading to STATE_6 -> Lexeme of type TOK_SKIP */
  STATE_6 (StateType.ACCEPTING),
  /* Lexemes leading to STATE_7 -> Lexeme of type TOK_SKIP */
  STATE_7 (StateType.REJECTING),
  STATE_8 (StateType.REJECTING),
  /* Lexemes leading to STATE_9 -> Lexeme of type TOK_SKIP */
  STATE_9 (StateType.ACCEPTING),
  /* Lexemes leading to STATE_10 -> Lexeme of some PUNCTUATION type*/
  STATE_10 (StateType.ACCEPTING),
  /* Lexemes leading to STATE_11 -> Lexeme of type TOK_INTEGER_LITERAL*/
  STATE_11 (StateType.ACCEPTING),
  STATE_12 (StateType.REJECTING),
  /* Lexemes leading to STATE_13 -> Lexeme of type TOK_FLOAT_LITERAL*/
  STATE_13 (StateType.ACCEPTING),
  /* Lexemes leading to STATE_14 -> Lexeme of type TOK_MUTIPLICATIVE_OP or TOK_ADDITIVE_OP*/
  STATE_14 (StateType.ACCEPTING),
  /* Lexemes leading to STATE_15 -> Lexeme of type TOK_ADDITIVE_OP*/
  STATE_15 (StateType.ACCEPTING),
```

```
/* Lexemes leading to STATE_16 -> Lexeme of type TOK_RELATIONAL_OP*/
STATE_16 (StateType.ACCEPTING),
STATE_17 (StateType.REJECTING),
/* Lexemes leading to STATE_18 -> Lexeme of type TOK_RELATIONAL_OP*/
STATE_18 (StateType.ACCEPTING),

/* Lexemes leading to STATE_18 -> Lexeme of type TOK_RELATIONAL_OP*/
STATE_19 (StateType.ACCEPTING),
STATE_ERROR(StateType.REJECTING),
/* STATE_BAD USED IN ALGORITHM OF GENERATING TOKENS FROM TRANSITION TABLE */
STATE_BAD(StateType.REJECTING);
private final StateType stateType;
/**
 *
 * @param stateId
 */
State(StateType stateType) {
  this.stateType = stateType;
}

/**
 * Getter method for getting a state's id
 * @return
 */
public StateType getStateType() {
  return this.stateType;
}
public TokenType getTokenType(String lexeme){

  switch(this) {
  case STATE_3:
    return TokenType.TOK_CHAR_LITERAL;

  case STATE_4:
    switch(lexeme) {
      case "fn":
        return TokenType.TOK_FN;
      case "bool":
        return TokenType.TOK_BOOL_TYPE;
      case "int":
        return TokenType.TOK_INT_TYPE;
      case "float":
        return TokenType.TOK_FLOAT_TYPE;
      case "false":
      case "true":
        return TokenType.TOK_BOOL_LITERAL;
      case "not":
        return TokenType.TOK_NOT;
      case "let":
        return TokenType.TOK_LET;
      case "char":
        return TokenType.TOK_CHAR_TYPE;
      case "if":
        return TokenType.TOK_IF;
      case "else":
        return TokenType.TOK_ELSE;
      case "while":
        return TokenType.TOK_WHILE;
      case "for":
        return TokenType.TOK_FOR;
      case "print":
```

```
                return TokenType.TOK_PRINT;
            case "return":
                return TokenType.TOK_RETURN;
            case "and":
                return TokenType.TOK_MULTIPLICATIVE_OP;
            case "or":
                return TokenType.TOK_ADDITIVE_OP;

            default:
                return TokenType.TOK_IDENTIFIER;
        }

    case STATE_5:
        return TokenType.TOK_MULTIPLICATIVE_OP;

    case STATE_6:
        return TokenType.TOK_SKIP;

    case STATE_9:
        return TokenType.TOK_SKIP;

    case STATE_10:
        switch(lexeme) {
        case ":":
            return TokenType.TOK_COLON;
        case ";":
            return TokenType.TOK_SEMICOLON;
        case "(":
            return TokenType.TOK_LEFT_ROUND_BRACKET;
        case ")":
            return TokenType.TOK_RIGHT_ROUND_BRACKET;
        case "{":
            return TokenType.TOK_LEFT_CURLY_BRACKET;
        case "}":
            return TokenType.TOK_RIGHT_CURLY_BRACKET;
        case ",":
            return TokenType.TOK_COMMA;
        case ".":
            return TokenType.TOK_DOT;
        default:
            return TokenType.INVALID;

        }
    case STATE_11:
        return TokenType.TOK_INT_LITERAL;

    case STATE_13:
        return TokenType.TOK_FLOAT_LITERAL;
    case STATE_14:
        switch(lexeme) {
            case "*":
                return TokenType.TOK_MULTIPLICATIVE_OP;
            case "+":
                return TokenType.TOK_ADDITIVE_OP;
            default:
                return TokenType.INVALID;
        }
    case STATE_15:

        return TokenType.TOK_ADDITIVE_OP;

```

```
    case STATE_16:
      switch(lexeme) {
      case "=":
        return TokenType.TOK_EQUAL;
      default:
        return TokenType.TOK_RELATIONAL_OP;
      }
    case STATE_18:
      return TokenType.TOK_RIGHT_ARROW;

    case STATE_19:
      return TokenType.TOK_RELATIONAL_OP;
    default:
      return TokenType.INVALID;
    }
  }
}
```

Listing 6.2: tinylang's dfsa states

```
package tinylanglexer;
/**
 * Consists of all possible inputs
 * of dfsa representing TinyLang's grammar.
 *
 * Total number of inputs : 16
 * @author andre
 *
 */
public enum InputCategory {
  /* LETTER ⍰ {a,b,...,z,A,B,...,Z} ≡ ASCII LETTER ⍰ {[0x41,0x5a],[0x61,0x7a]} */
  LETTER,
  /* DIGIT ⍰ {0,1,2,...,9} ≡ ASCII DIGIT ⍰ {[0x30,0x39]]} */
  DIGIT,
  /* UNDERSCORE ⍰ {_} ≡ ASCII UNDERSCORE ⍰ {0x5f} */
  UNDERSCORE,
  /* SLASH_DIVIDE ⍰ {/} ≡ ASCII SLASH_DIVIDE ⍰ {0x2f} */
  SLASH_DIVIDE ,
  /* ASTERISK ⍰ {*} ≡ ASCII ASTERISK ⍰ {0x2a} */
  ASTERISK,
  /* LESS_THAN ⍰ {<} ≡ ASCII LESS_THAN ⍰ {0x3c} */
  LESS_THAN,
  /* FORWARD_SLASH ⍰ {>} ≡ ASCII FORWARD_SLASH ⍰ {0x3e} */
  GREATER_THAN,
  /* PLUS ⍰ {+} ≡ ASCII FORWARD_SLASH ⍰ {0x2B} */
  PLUS,
  /* HYPHEN_MINUS ⍰ {-} ≡ ASCII HYPHEN_MINUS ⍰ {0x2d} */
  HYPHEN_MINUS,
  /* EQUAL ⍰ {=} ≡ ASCII HYPHEN_MINUS ⍰ {0x3d} */
  EQUAL,
  /* EXCLAMATION_MARK ⍰ {!} ≡ ASCII EXCLAMATION_MARK ⍰ {0x21} */
  EXCLAMATION_MARK,
  /* DOT ⍰ {.} ≡ ASCII HYPHEN_MINUS ⍰ {0x2e} */
  DOT,
  /* SINGLE_QUOTE ⍰ {'} ≡ ASCII HYPHEN_MINUS ⍰ {0x27} */
  SINGLE_QUOTE,
  /* PUNCTUATION ⍰ {(  ,)  ,,  ,:  ,;,{  ,}  } ≡ ASCII PUNCTUATION ⍰ {0x28, 0x29,0x2c, 0x3a, 0x3b,0
    x7b ,0x7d} */
  PUNCT,
  /* ASCII : OTHER_PRINTABLE ⍰ {[0x20,0x7e]}
   * \ (LETTERS, DIGITS ⍰ UNDERSCORE ⍰ FORWARD_SLASH ⍰ ASTERISK ⍰ LESS_THAN
```

```
     *     | GREATER_THAN | PLUS, MINUS | EQUAL | EXCLAMATION_MARK | DOT
     *     | SINGLE_QUOTE | PUNCTUATION) */
    OTHER_PRINTABLE,
    /* LINE_FEED | {\n} ≡ ASCII LINE_FEED | {0x0a} */
    LINE_FEED
}
```

Listing 6.3: Implementation of classifier table

```
package tinylanglexer;
import java.util.HashMap;
import java.util.Map;
public class TransitionTable {
  protected Map<TransitionInput, State> buildTransitionTable(){
    Map<TransitionInput, State> transitionTable = new HashMap<TransitionInput, State>();
    State fromState;
    /************** transition table row 1 ****/
    fromState = State.STARTING_STATE;
    transitionTable.put(new TransitionInput(fromState, InputCategory.LETTER), State.STATE_4);
    transitionTable.put(new TransitionInput(fromState, InputCategory.DIGIT), State.STATE_11);
    transitionTable.put(new TransitionInput(fromState, InputCategory.UNDERSCORE), State.STATE_4);
    transitionTable.put(new TransitionInput(fromState, InputCategory.SLASH_DIVIDE), State.STATE_5);
    transitionTable.put(new TransitionInput(fromState, InputCategory.ASTERISK), State.STATE_14);
    transitionTable.put(new TransitionInput(fromState, InputCategory.LESS_THAN), State.STATE_16);
    transitionTable.put(new TransitionInput(fromState, InputCategory.GREATER_THAN), State.STATE_16);
    transitionTable.put(new TransitionInput(fromState, InputCategory.PLUS), State.STATE_14);
    transitionTable.put(new TransitionInput(fromState, InputCategory.HYPHEN_MINUS), State.STATE_15);
    transitionTable.put(new TransitionInput(fromState, InputCategory.EQUAL), State.STATE_16);
    transitionTable.put(new TransitionInput(fromState, InputCategory.EXCLAMATION_MARK), State.STATE_17);
    transitionTable.put(new TransitionInput(fromState, InputCategory.DOT), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState, InputCategory.SINGLE_QUOTE), State.STATE_1);
    transitionTable.put(new TransitionInput(fromState, InputCategory.PUNCT), State.STATE_10);
    transitionTable.put(new TransitionInput(fromState, InputCategory.OTHER_PRINTABLE), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState, InputCategory.LINE_FEED), State.STATE_ERROR);
    /************** end transition table row 1 ****/
    /************** transition table row 2 ****/
    fromState = State.STATE_1;
    transitionTable.put(new TransitionInput(fromState, InputCategory.LETTER), State.STATE_2);
    transitionTable.put(new TransitionInput(fromState, InputCategory.DIGIT), State.STATE_2);
    transitionTable.put(new TransitionInput(fromState, InputCategory.UNDERSCORE), State.STATE_2);
    transitionTable.put(new TransitionInput(fromState, InputCategory.SLASH_DIVIDE), State.STATE_2);
    transitionTable.put(new TransitionInput(fromState, InputCategory.ASTERISK), State.STATE_2);
    transitionTable.put(new TransitionInput(fromState, InputCategory.LESS_THAN), State.STATE_2);
    transitionTable.put(new TransitionInput(fromState, InputCategory.GREATER_THAN), State.STATE_2);
    transitionTable.put(new TransitionInput(fromState, InputCategory.PLUS), State.STATE_2);
    transitionTable.put(new TransitionInput(fromState, InputCategory.HYPHEN_MINUS), State.STATE_2);
```

```java
    transitionTable.put(new TransitionInput(fromState,InputCategory.EQUAL), State.STATE_2);
    transitionTable.put(new TransitionInput(fromState,InputCategory.EXCLAMATION_MARK), State.STATE_2);
    transitionTable.put(new TransitionInput(fromState,InputCategory.DOT), State.STATE_2);
    transitionTable.put(new TransitionInput(fromState,InputCategory.SINGLE_QUOTE), State.STATE_2);
    transitionTable.put(new TransitionInput(fromState,InputCategory.PUNCT), State.STATE_2);
    transitionTable.put(new TransitionInput(fromState,InputCategory.OTHER_PRINTABLE), State.STATE_2);
    transitionTable.put(new TransitionInput(fromState,InputCategory.LINE_FEED), State.STATE_ERROR);
    /************** end transition table row 2 ****/
    /************** transition table row 3 ****/
    fromState = State.STATE_2;
    transitionTable.put(new TransitionInput(fromState,InputCategory.LETTER), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.DIGIT), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.UNDERSCORE), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.SLASH_DIVIDE), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.ASTERISK), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.LESS_THAN), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.GREATER_THAN), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.PLUS), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.HYPHEN_MINUS), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.EQUAL), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.EXCLAMATION_MARK), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.DOT), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.SINGLE_QUOTE), State.STATE_3);
    transitionTable.put(new TransitionInput(fromState,InputCategory.PUNCT), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.OTHER_PRINTABLE), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.LINE_FEED), State.STATE_ERROR);
    /************** end transition table row 3 ****/
    /************** transition table row 4  ****/
    fromState = State.STATE_3;
    for (InputCategory input : InputCategory.values()) {
      transitionTable.put(new TransitionInput(fromState,input), State.STATE_ERROR);
    }
    /************** end transition table row 4 ****/
    /************** transition table row 5 ****/
    fromState = State.STATE_4;
    transitionTable.put(new TransitionInput(fromState,InputCategory.LETTER), State.STATE_4);
    transitionTable.put(new TransitionInput(fromState,InputCategory.DIGIT), State.STATE_4);
    transitionTable.put(new TransitionInput(fromState,InputCategory.UNDERSCORE), State.STATE_4);
    transitionTable.put(new TransitionInput(fromState,InputCategory.SLASH_DIVIDE), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.ASTERISK), State.STATE_ERROR);
```

```java
transitionTable.put(new TransitionInput(fromState,InputCategory.LESS_THAN), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.GREATER_THAN), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.PLUS), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.HYPHEN_MINUS), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.EQUAL), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.EXCLAMATION_MARK), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.DOT), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.SINGLE_QUOTE), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.PUNCT), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.OTHER_PRINTABLE), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.LINE_FEED), State.STATE_ERROR);
/************** end transition table row 5 ****/
/************** transition table row 6  ****/
fromState = State.STATE_5;
transitionTable.put(new TransitionInput(fromState,InputCategory.LETTER), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.DIGIT), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.UNDERSCORE), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.SLASH_DIVIDE), State.STATE_6);
transitionTable.put(new TransitionInput(fromState,InputCategory.ASTERISK), State.STATE_7);
transitionTable.put(new TransitionInput(fromState,InputCategory.LESS_THAN), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.GREATER_THAN), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.PLUS), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.HYPHEN_MINUS), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.EQUAL), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.EXCLAMATION_MARK), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.DOT), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.SINGLE_QUOTE), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.PUNCT), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.OTHER_PRINTABLE), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.LINE_FEED), State.STATE_ERROR);
/************** end transition table row 6 ****/
/************** transition table row 7  ****/
fromState = State.STATE_6;
transitionTable.put(new TransitionInput(fromState,InputCategory.LETTER), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.DIGIT), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.UNDERSCORE), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.SLASH_DIVIDE), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.ASTERISK), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.LESS_THAN), fromState);
```

```
transitionTable.put(new TransitionInput(fromState,InputCategory.GREATER_THAN), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.PLUS), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.HYPHEN_MINUS), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.EQUAL), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.EXCLAMATION_MARK), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.DOT), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.SINGLE_QUOTE), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.PUNCT), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.OTHER_PRINTABLE), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.LINE_FEED), State.STATE_ERROR);
/************** end transition table row 7 ****/
/************** transition table row 8  ****/
fromState = State.STATE_7;
transitionTable.put(new TransitionInput(fromState,InputCategory.LETTER), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.DIGIT), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.UNDERSCORE), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.SLASH_DIVIDE), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.ASTERISK), State.STATE_8);
transitionTable.put(new TransitionInput(fromState,InputCategory.LESS_THAN), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.GREATER_THAN), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.PLUS), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.HYPHEN_MINUS), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.EQUAL), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.EXCLAMATION_MARK), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.DOT), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.SINGLE_QUOTE), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.PUNCT), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.OTHER_PRINTABLE), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.LINE_FEED), fromState);
/************** end transition table row 8 ****/
/************** transition table row 9  ****/
fromState = State.STATE_8;
transitionTable.put(new TransitionInput(fromState,InputCategory.LETTER), State.STATE_7);
transitionTable.put(new TransitionInput(fromState,InputCategory.DIGIT), State.STATE_7);
transitionTable.put(new TransitionInput(fromState,InputCategory.UNDERSCORE), State.STATE_7);
transitionTable.put(new TransitionInput(fromState,InputCategory.SLASH_DIVIDE), State.STATE_9);
transitionTable.put(new TransitionInput(fromState,InputCategory.ASTERISK), State.STATE_7);
transitionTable.put(new TransitionInput(fromState,InputCategory.LESS_THAN), State.STATE_7);
transitionTable.put(new TransitionInput(fromState,InputCategory.GREATER_THAN), State.STATE_7);
transitionTable.put(new TransitionInput(fromState,InputCategory.PLUS), State.STATE_7);
transitionTable.put(new TransitionInput(fromState,InputCategory.HYPHEN_MINUS), State.STATE_7);
transitionTable.put(new TransitionInput(fromState,InputCategory.EQUAL), State.STATE_7);
transitionTable.put(new TransitionInput(fromState,InputCategory.EXCLAMATION_MARK), State.STATE_7);
transitionTable.put(new TransitionInput(fromState,InputCategory.DOT), State.STATE_7);
transitionTable.put(new TransitionInput(fromState,InputCategory.SINGLE_QUOTE), State.STATE_7);
transitionTable.put(new TransitionInput(fromState,InputCategory.PUNCT), State.STATE_7);
transitionTable.put(new TransitionInput(fromState,InputCategory.OTHER_PRINTABLE), State.STATE_7);
transitionTable.put(new TransitionInput(fromState,InputCategory.LINE_FEED), State.STATE_7)
```

```
;
/************** end transition table row 9  ****/
/************** transition table row 10  ****/
fromState = State.STATE_9;
for (InputCategory input : InputCategory.values()) {
  transitionTable.put(new TransitionInput(fromState, input), State.STATE_ERROR);
}
/************** end transition table row 10 ****/
/************** transition table row 11 ****/
fromState = State.STATE_10;
for (InputCategory input : InputCategory.values()) {
  transitionTable.put(new TransitionInput(fromState, input), State.STATE_ERROR);
}
/************** end transition table row 11 ****/

/************** transition table row 12 ****/
fromState = State.STATE_11;
transitionTable.put(new TransitionInput(fromState, InputCategory.LETTER), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState, InputCategory.DIGIT), State.STATE_11);
transitionTable.put(new TransitionInput(fromState, InputCategory.UNDERSCORE), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState, InputCategory.SLASH_DIVIDE), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState, InputCategory.ASTERISK), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState, InputCategory.LESS_THAN), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState, InputCategory.GREATER_THAN), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState, InputCategory.PLUS), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState, InputCategory.HYPHEN_MINUS), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState, InputCategory.EQUAL), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState, InputCategory.EXCLAMATION_MARK), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState, InputCategory.DOT), State.STATE_12);
transitionTable.put(new TransitionInput(fromState, InputCategory.SINGLE_QUOTE), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState, InputCategory.PUNCT), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState, InputCategory.OTHER_PRINTABLE), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState, InputCategory.LINE_FEED), State.STATE_ERROR);
/************** end transition table row 12  ****/

/************** transition table row 13 ****/
fromState = State.STATE_12;
transitionTable.put(new TransitionInput(fromState, InputCategory.LETTER), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState, InputCategory.DIGIT), State.STATE_13);
transitionTable.put(new TransitionInput(fromState, InputCategory.UNDERSCORE), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState, InputCategory.SLASH_DIVIDE), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState, InputCategory.ASTERISK), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState, InputCategory.LESS_THAN), State.STATE_ERROR);
```

```java
transitionTable.put(new TransitionInput(fromState,InputCategory.GREATER_THAN), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.PLUS), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.HYPHEN_MINUS), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.EQUAL), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.EXCLAMATION_MARK), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.DOT), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.SINGLE_QUOTE), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.PUNCT), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.OTHER_PRINTABLE), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.LINE_FEED), State.STATE_ERROR);
/************** end transition table row 13  ****/

/************** transition table row 14  ****/
fromState = State.STATE_13;
transitionTable.put(new TransitionInput(fromState,InputCategory.LETTER), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.DIGIT), fromState);
transitionTable.put(new TransitionInput(fromState,InputCategory.UNDERSCORE), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.SLASH_DIVIDE), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.ASTERISK), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.LESS_THAN), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.GREATER_THAN), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.PLUS), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.HYPHEN_MINUS), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.EQUAL), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.EXCLAMATION_MARK), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.DOT), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.SINGLE_QUOTE), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.PUNCT), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.OTHER_PRINTABLE), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.LINE_FEED), State.STATE_ERROR);
/************** end transition table row 14 *****/
/************** transition table row 15  ****/
fromState = State.STATE_14;
for (InputCategory input : InputCategory.values()) {
  transitionTable.put(new TransitionInput(fromState,input), State.STATE_ERROR);
}
/************** end transition table row 15  ****/
/************** transition table row 16 ****/
fromState = State.STATE_15;
transitionTable.put(new TransitionInput(fromState,InputCategory.LETTER), State.STATE_ERROR
```

```java
);
transitionTable.put(new TransitionInput(fromState,InputCategory.DIGIT), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.UNDERSCORE), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.SLASH_DIVIDE), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.ASTERISK), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.LESS_THAN), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.GREATER_THAN), State.STATE_18);
transitionTable.put(new TransitionInput(fromState,InputCategory.PLUS), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.HYPHEN_MINUS), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.EQUAL), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.EXCLAMATION_MARK), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.DOT), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.SINGLE_QUOTE), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.PUNCT), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.OTHER_PRINTABLE), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.LINE_FEED), State.STATE_ERROR);
/************** end transition table row 16  ****/
/************** transition table row 17 ****/
fromState = State.STATE_16;
transitionTable.put(new TransitionInput(fromState,InputCategory.LETTER), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.DIGIT), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.UNDERSCORE), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.SLASH_DIVIDE), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.ASTERISK), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.LESS_THAN), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.GREATER_THAN), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.PLUS), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.HYPHEN_MINUS), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.EQUAL), State.STATE_19);
transitionTable.put(new TransitionInput(fromState,InputCategory.EXCLAMATION_MARK), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.DOT), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.SINGLE_QUOTE), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.PUNCT), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.OTHER_PRINTABLE), State.STATE_ERROR);
transitionTable.put(new TransitionInput(fromState,InputCategory.LINE_FEED), State.STATE_ERROR);
```

```
    /************** end transition table row 17  ****/
    /************** transition table row 18  ****/
    fromState = State.STATE_17;
    transitionTable.put(new TransitionInput(fromState,InputCategory.LETTER), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.DIGIT), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.UNDERSCORE), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.SLASH_DIVIDE), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.ASTERISK), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.LESS_THAN), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.GREATER_THAN), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.PLUS), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.HYPHEN_MINUS), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.EQUAL), State.STATE_19);
    transitionTable.put(new TransitionInput(fromState,InputCategory.EXCLAMATION_MARK), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.DOT), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.SINGLE_QUOTE), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.PUNCT), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.OTHER_PRINTABLE), State.STATE_ERROR);
    transitionTable.put(new TransitionInput(fromState,InputCategory.LINE_FEED), State.STATE_ERROR);
    /************** end transition table row 18  ****/
    /************** transition table row 19  ****/
    fromState = State.STATE_18;
    for (InputCategory input : InputCategory.values()) {
      transitionTable.put(new TransitionInput(fromState,input), State.STATE_ERROR);
    }
    /************** end transition table row 19 ****/
    /************** transition table row 20  ****/
    fromState = State.STATE_19;
    for (InputCategory input : InputCategory.values()) {
      transitionTable.put(new TransitionInput(fromState,input), State.STATE_ERROR);
    }
    /************** end transition table row 20 ****/
    return transitionTable;
  }
}
```

Listing 6.4: Implementation of transition table

```
package tinylanglexer;
/**
 * Infinite amount of possible lexemes are
 * categorised into a finite amount of groups.
 * Therefore a lexeme is a string with an
 * identified meaning in the language.
 * @author andre
 */
public enum TokenType {
  /**
```

```
 * Syntax Error Handler
 * Identifies lexemes which are not accepted by TinyLang's grammar.
 */
INVALID,
/**
 *Control Flow Keyword
 *Value(s) : if
 */
TOK_IF,
/**
 * Control Flow Keyword
 * Value(s) : else
 */
TOK_ELSE,
/**
 * Iteration Keyword
 * Value(s) : for
 */
TOK_FOR,
/**
 * Iteration Keyword
 * Value(s) : while
 */
TOK_WHILE,
/**
 * Structure Keyword
 * Value(s) : fn
 */
TOK_FN,
/**
 * Returning Keyword
 * Value(s) : fn
 */
TOK_RETURN,
/**
 * Data Type Keyword
 * Value(s) : int
 */
TOK_INT_TYPE,
/**
 * Data Type Keyword
 * Value(s) : float
 */
TOK_FLOAT_TYPE,
/**
 * Data Type Keyword
 * Value(s) : not
 */
TOK_NOT,
/**
 * Data Type Keyword
 * Value(s) : bool
 */
TOK_BOOL_TYPE,
/**
 * Data Type Keyword
 * Value(s) : char
 */
TOK_CHAR_TYPE,
/**
 * Keyword Token
```

```
     * Value(s) : let
     * identify variable declaration
     */
    TOK_LET,
    /**
     * Keyword Token
     * Value(s) : ->
     * specify return type of a function
     */
    TOK_RIGHT_ARROW,

    /**
     * Keyword Token
     * Value(s) : print
     * identify print statement
     */
    TOK_PRINT,
    /**
     * Punctuation
     * Value(s) : (
     */
    TOK_LEFT_ROUND_BRACKET,
    /**
     * Punctuation
     * Value(s) : )
     */
    TOK_RIGHT_ROUND_BRACKET,
    /**
     * Punctuation
     * Value(s) : {
     */
    TOK_LEFT_CURLY_BRACKET,
    /**
     * Punctuation
     * Value(s) : }
     */
    TOK_RIGHT_CURLY_BRACKET,
    /**
     * Punctuation
     * Value(s) : ,
     */
    TOK_COMMA,
    /**
     * Punctuation
     * Value(s) :
     */
    TOK_DOT,
    /**
     * Punctuation
     * Value(s) : :
     */
    TOK_COLON,
    /**
     * Punctuation
     * Value(s) : ;
     */
    TOK_SEMICOLON,

    /**
     * Punctuation
     * Value(s) : ;
```

```
  */
 TOK_MULTIPLICATIVE_OP,
 /**
  *
  * Value(s) : (
  */
 TOK_ADDITIVE_OP,
 /**
  * Operation Token Name
  * Value(s) : =
  */
 TOK_EQUAL,
 /**
  * Operation Token Name
  * Value(s) : '<' '>' '==' '!=' '<=' '>='
  */
 TOK_RELATIONAL_OP,
 /**
  * Token Name
  */
 TOK_IDENTIFIER,
 /**
  * Token Name
  * Value(s) : true , false
  */
 TOK_BOOL_LITERAL,
 /**
  * Token Name
  */
 TOK_INT_LITERAL,
 /**
  * Token Name
  */
 TOK_FLOAT_LITERAL,
 /**
  * Token Name
  */
 TOK_CHAR_LITERAL,

 /**
  * Special Token
  */
 TOK_SKIP,
 /**
  * Special Token
  * Used to identify end of program
  */
 TOK_EOF
}
```

Listing 6.5: Token Types

```
package tinylanglexer;
public class Token {
  //attribute associated with token type
  private String lexeme;
  //tokenType
  private TokenType tokenType;
  //line number where lexeme resided
  private int lineNumber;
  public Token(TokenType tokenType, String lexeme) {
```

```
    this.tokenType = tokenType;
    this.lexeme = lexeme;
  }
  // setters and getters
  public String getLexeme() {
    return lexeme;
  }
  public void setLexeme(String lexeme) {
    this.lexeme = lexeme;
  }
  public TokenType getTokenType() {
    return this.tokenType;
  }
  public void setTokenType(TokenType tokenType) {
    this.tokenType = tokenType;
  }
  public int getLineNumber() {
    return lineNumber;
  }
  public void setLineNumber(int lineNumber) {
    this.lineNumber = lineNumber;
  }
}
```

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

```
package tinylanglexer;
import java.util.ArrayList;
import java.util.Map;
import java.util.Stack;

/**
 * Class for lexer implementation of TinyLang
 * extends TransitionTable
 * @author andre
 */
public class TinyLangLexer extends TransitionTable{
  // Obtain transition table from class TransitionTable
  private Map<TransitionInput,State> transitionTable = buildTransitionTable();
  // List of tokens
  private ArrayList<Token> tokens = new ArrayList<>();
  // Scanning -> traverse program char by char -> keep track of current char
  private int currentCharIndex = 0;
  // Keep track of line number
  private int lineNumber = 0;
  /**
   * Constructor for class TinyLangLexer
   * @param TinyLangProgram
   * @throws Exception
   */
  public TinyLangLexer(String tinyLangProgram) {
    // build transition table
    this.buildTransitionTable();
    // program is empty -> only one EOF token
        if(tinyLangProgram.length()==0)
        this.tokens.add(new Token(TokenType.TOK_EOF,""));
    // if program is not empty -> loop until current char is not at the end of file
    while(currentCharIndex<tinyLangProgram.length()) {
      // obtain next token
      Token nextToken = getNextToken(tinyLangProgram);
      // set line number
```

```
      nextToken.setLineNumber(getLineNumber(tinyLangProgram));
      // if token is not of type TOK_SKIP add to list of tokens
      if(nextToken.getTokenType()!= TokenType.TOK_SKIP) {
        this.tokens.add(nextToken);
      }
    }
  }
  /**
   * Table Driven Analysis Algorithm -> Cooper & Torczon Engineer a Compiler.
   * @param TinyLangProgram
   */
  private Token getNextToken(String tinyLangProgram) {
    /* start initialisation stage */
    // Set current state to start state
    State state = State.STARTING_STATE;
    // Current lexeme
        String lexeme = "";
    // Create Stack Of States
        Stack<State> stack = new Stack<State>();
        // Push BAD state to the stack
        stack.add(State.STATE_BAD);
    /* end initialisation stage */
        while(tinyLangProgram.charAt(currentCharIndex) == 0x0a || tinyLangProgram.charAt(
    currentCharIndex)==0x20 ||  tinyLangProgram.charAt(currentCharIndex)==0x09) {
          if(tinyLangProgram.charAt(currentCharIndex) == 0x0a)
            lineNumber++;
          // increment char number
          this.currentCharIndex++;
            // detect EOF
          if(currentCharIndex==tinyLangProgram.length())
            return new Token(TokenType.TOK_EOF,"");
        }
        InputCategory inputCategory;
        char currentChar;
        while(state!=State.STATE_ERROR &&currentCharIndex<tinyLangProgram.length()) {
            // obtain current CHAR
        currentChar = tinyLangProgram.charAt(currentCharIndex);
        // char to lexeme
        lexeme+=currentChar;
        // if state is accepting clear stack
        if (state.getStateType()==StateType.ACCEPTING) {
          stack.clear();
        }
        // push state to stack
        stack.add(state);
        if (isLetter(currentChar)) {
          inputCategory = InputCategory.LETTER;
            }
            else if (isDigit(currentChar)) {
          inputCategory = InputCategory.DIGIT;
            }
            else if (isUnderscore(currentChar)) {
          inputCategory = InputCategory.UNDERSCORE;
            }
            else if (isSlashDivide(currentChar)) {
          inputCategory = InputCategory.SLASH_DIVIDE;
            }
            else if (isAsterisk(currentChar)) {
          inputCategory = InputCategory.ASTERISK;
            }
            else if (isLessThan(currentChar)) {
```

```
          inputCategory = InputCategory.LESS_THAN;
            }
            else if (isGreaterThan(currentChar)) {
          inputCategory = InputCategory.GREATER_THAN;
            }
            else if (isPlus(currentChar)) {
          inputCategory = InputCategory.PLUS;
            }
            else if (isHyphenMinus(currentChar)) {
          inputCategory = InputCategory.HYPHEN_MINUS;
            }
            else if (isEqual(currentChar)) {
          inputCategory = InputCategory.EQUAL;
            }
            else if (isExclamationMark(currentChar)) {
          inputCategory = InputCategory.EXCLAMATION_MARK;
            }
            else if (isDot(currentChar)) {
          inputCategory = InputCategory.DOT;
            }
            else if (isSingleQuote(currentChar)) {
          inputCategory = InputCategory.SINGLE_QUOTE;
            }
            else if (isPunctuation(currentChar)) {
          inputCategory = InputCategory.PUNCT;
            }
            else if (isOtherPrintable(currentChar)) {
          inputCategory = InputCategory.OTHER_PRINTABLE;
            }
            else if (isLineFeed(currentChar)) {
          inputCategory = InputCategory.LINE_FEED;
            }
            else {
              throw new java.lang.RuntimeException("char "+currentChar+" in line " +lineNumber
    +" not recognised by TinyLang's grammar");
            }
              // get next transition as per transition table

        state = deltaFunction(state,inputCategory);
        // move to next char
            currentCharIndex++;



          }
        /*          begin rollback loop              */
        while(state!=State.STATE_BAD && state.getStateType()!=StateType.ACCEPTING) {
          // pop state
          state=stack.pop();
          //truncate string
          lexeme = (lexeme==null||lexeme.length()==0)?null:(lexeme.substring(0, lexeme.length
    ()-1));
          // move char index one step backwards
          currentCharIndex--;
        }
      if(state.getTokenType(lexeme)==TokenType.INVALID)
        throw new java.lang.RuntimeException(tokens.get(tokens.size()-1).getLexeme()+
    tinyLangProgram.charAt(currentCharIndex+1)+" in line "+lineNumber+" not recognised by
    TinyLang's grammar");
      else
        return new Token(state.getTokenType(lexeme),lexeme);
```

```
      // end lineNumber
    }
  // predicate functions to check input category
  private boolean isLetter(char input) {
    return ( (0x41 <= input && input  <= 0x5a) || (0x61 <= input && input <= 0x7a) );
  }
  private boolean isDigit(char input) {
    return (0x30 <= input && input  <= 0x39);
  }
  private boolean isUnderscore(char input) {
    return (input == 0x5f);
  }
  private boolean isSlashDivide(char input) {
    return (input == 0x2f);
  }
  private boolean isAsterisk(char input) {
    return (input == 0x2a);
  }
  private boolean isLessThan(char input) {
    return (input == 0x3c);
  }
  private boolean isGreaterThan(char input) {
    return (input == 0x3e);
  }
  private boolean isPlus(char input) {
    return (input == 0x2b);
  }
  private boolean isHyphenMinus(char input) {
    return (input == 0x2d);
  }
  private boolean isEqual(char input) {
    return (input == 0x3d);
  }
  private boolean isExclamationMark(char input) {
    return (input == 0x21);
  }
  private boolean isDot(char input) {
    return (input == 0x2e);
  }
  private boolean isSingleQuote(char input) {
    return (input == 0x27);
  }
  private boolean isPunctuation(char input) {
    return (input == 0x28||input == 0x29||input == 0x2c||input==0x3a||input==0x3b||input== 0
    x7b||input==0x7d);
  }
  private boolean isOtherPrintable(char input) {
    return ( 0x20 <= input && input <= 0x7e&&!isLetter(input) && !isDigit(input) && !
    isUnderscore(input) &&
        !isSlashDivide(input) && !isAsterisk(input) && !isLessThan(input) && !isGreaterThan(
    input)
        && !isPlus(input) && !isHyphenMinus(input) && !isEqual(input) && !isExclamationMark(
    input)
        && !isDot(input) && !isSingleQuote(input)) && !isPunctuation(input);
  }
  private boolean isLineFeed(char input) {
    lineNumber++;
    return (input == 0x0a);
  }
  private State deltaFunction(State state,InputCategory inputCategory) {
    return transitionTable.get(new TransitionInput(state,inputCategory));
```

```java
  }

  // setter and getter methods
  public ArrayList<Token> getTokens(){
    return tokens;
  }


  private int getLineNumber(String tinyLangProgram) {
    lineNumber = 1;
    for(int i=0;i<currentCharIndex;i++)
      if (tinyLangProgram.charAt(i)==0x0a)
        lineNumber++;
    return lineNumber;


  }

}
```

Listing 6.7: Table Driven Lexer

6.2 | Parser

```java
package tinylangparser;
import java.util.LinkedList;
import java.util.List;
public class TinyLangAst {
  /* node */
  private TinyLangAstNodes associatedNodeType;
  private String associatedNodeValue = "";
  private int lineNumber = 0;
    TinyLangAst parent;
    List<TinyLangAst> children;

    public TinyLangAst(TinyLangAstNodes associatedNodeType,int lineNumber) {
        this.associatedNodeType = associatedNodeType;
        this.lineNumber=lineNumber;
        this.children = new LinkedList<TinyLangAst>();
    }
    public TinyLangAst(TinyLangAstNodes associatedNodeType, String associatedNodeValue,int
    lineNumber) {
        this.associatedNodeType = associatedNodeType;
        this.associatedNodeValue = associatedNodeValue;
        this.lineNumber=lineNumber;
        this.children = new LinkedList<TinyLangAst>();
    }
    //add root of a subtree to abstract syntax tree
    public void addSubtree(TinyLangAst subTree) {
      this.children.add(subTree);
    }
    public TinyLangAst addChild(TinyLangAstNodes associatedNodeType,int lineNumber) {
      TinyLangAst childNode = new TinyLangAst(associatedNodeType,lineNumber);
        childNode.parent = this;
        this.children.add(childNode);
        return childNode;
```

```
    }
    public TinyLangAst addChild(TinyLangAstNodes associatedNodeType,String associatedNodeValue
    ,int lineNumber) {
      TinyLangAst childNode = new TinyLangAst(associatedNodeType,associatedNodeValue,
    lineNumber);
        childNode.parent = this;
        this.children.add(childNode);
        return childNode;
    }

    // setters and getters
    public TinyLangAstNodes getAssociatedNodeType(){
      return associatedNodeType;
    };
    public String getAssociatedNodeValue(){
      return associatedNodeValue;
    }
   // get children
    public List<TinyLangAst> getChildren(){
      return children;
    }
    public void setLineNumber(int lineNumber) {
      this.lineNumber=lineNumber;
    }
    public int getLineNumber() {
      return lineNumber;
    }
}
```

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

```
package tinylangparser;
public enum TinyLangAstNodes {
  //program node
  TINY_LANG_PROGRAM_NODE,
  //statement nodes
  AST_VARIABLE_DECLARATION_NODE,
  AST_ASSIGNMENT_NODE,
  AST_PRINT_STATEMENT_NODE,
  AST_IF_STATEMENT_NODE,
  AST_FOR_STATEMENT_NODE,
  AST_WHILE_STATEMENT_NODE,
  AST_RETURN_STATEMENT_NODE,
  AST_FUNCTION_DECLARATION_NODE,
  AST_BLOCK_NODE,
  AST_ELSE_BLOCK_NODE,
  //expression nodes
  AST_BINARY_OPERATOR_NODE,
  AST_UNARY_OPERATOR_NODE,
  AST_FUNCTION_CALL_NODE,
  //literal nodes
  AST_BOOLEAN_LITERAL_NODE,
  AST_INTEGER_LITERAL_NODE,
  AST_FLOAT_LITERAL_NODE,
  AST_CHAR_LITERAL_NODE,
  //parameters nodes
  AST_ACTUAL_PARAMETERS_NODE,
  AST_FORMAL_PARAMETERS_NODE,
  AST_FORMAL_PARAMETER_NODE,
  //type node
  AST_TYPE_NODE,
```

```
  //expression nodes
  //expression nodes leaves
  AST_IDENTIFIER_NODE
}
```

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

```
package tinylangparser;
import java.util.ArrayList;
import tinylanglexer.TinyLangLexer;
import tinylanglexer.Token;
import tinylanglexer.TokenType;
public class TinyLangParser {
  // root of ast -> describes ast capturing all the program
  private TinyLangAst tinyLangProgramAbstractSyntaxTree;
  // list of tokens
  private ArrayList<Token> tokens;
  // current token index
  private int currentTokenIndex = 0;
  // method for obtaining current token
  private Token getCurrentToken(){
    return tokens.get(currentTokenIndex);
  }
  // method for obtaining next token
  private Token getNextToken(){
    currentTokenIndex++;
    return getCurrentToken();
  }
  // method for obtaining previous token
  private Token getPrevToken(){
    currentTokenIndex--;
    return getCurrentToken();
  }
  /**
   * Constructor for TinyLangParserClass
   * @param tinyLangLexer
   */
  public TinyLangParser(TinyLangLexer tinyLangLexer) {
    tokens = tinyLangLexer.getTokens();
    tinyLangProgramAbstractSyntaxTree = parseTinyLangProgram();
  }
  /**
   * Parse whole TinyLangProgram using recursive descent
   * to call other sub parsers until TOK_EOF is reached.
   */
  private TinyLangAst parseTinyLangProgram() {
    //program tree capturing whole syntax of tiny lang program
    TinyLangAst programTree = new TinyLangAst(TinyLangAstNodes.TINY_LANG_PROGRAM_NODE,
     getCurrentToken().getLineNumber());
    // traverse until current token reach EOF i.e. no more tokens to process
    while(getCurrentToken().getTokenType()!=TokenType.TOK_EOF) {
      // parse statement one by one
      programTree.addSubtree(parseStatement());
      // get next token
      getNextToken();
    }
    return programTree;
  }
  /**
```

```
     * Parse a statement
   * <Statement> -> <VariableDecl> ';'
     *<Statement> -> <Assignment> ';'
     *<Statement> -> <PrintStatement> ';'
     *<Statement> -> <IfStatement>  ';'
     *<Statement> -> <ForStatement> ';'
     *<Statement> -> <WhileStatement> ';'
     *<Statement> -> <RtrnStatement> ';'
     *<Statement> -> <FunctionDecl>
     *<Statement> -> <Block>
     *described by an  LL(1) grammar i.e. decide immediately which grammar rule to use with
    TokenTypes
     *  TOK_LET,TOK_IDENTIFIER,TOK_PRINT,TOK_WHILE,TOK_RETURN,TOK_FN,TOK_LEFT_CURLY otherwise
    undefined.
   * @param lookAhead
   * @param parent
   */
  public TinyLangAst parseStatement() {
    TinyLangAst statementTree;
    switch(getCurrentToken().getTokenType()){
    // if lookAhead = TOK_LET Statement leads to variable declaration
    case TOK_LET:
      //parse variable declaration
      statementTree =parseVariableDeclaration();
      //get next token
      getNextToken();
      //expecting ;
      if(getCurrentToken().getTokenType()!=TokenType.TOK_SEMICOLON)
        //not as expected
        throw new java.lang.RuntimeException("expected semicolon,; , in line " +
    getCurrentToken().getLineNumber());
      return statementTree;
    case TOK_IDENTIFIER:
      //parse assignment
      statementTree =  parseAssignment();
      //get next token
      getNextToken();
      //expecting ;
      if(getCurrentToken().getTokenType()!=TokenType.TOK_SEMICOLON)
        //not as expected
        throw new java.lang.RuntimeException("expected semicolon,; , in line " +
    getCurrentToken().getLineNumber());
      return statementTree;
    case TOK_PRINT:
      statementTree = parsePrintStatement();
      //expecting ;
      if(getNextToken().getTokenType()!=TokenType.TOK_SEMICOLON)
        //not as expected
        throw new java.lang.RuntimeException("expected semicolon,; , in line " +
    getCurrentToken().getLineNumber());
      return statementTree;
    case TOK_IF:
      return parseIfStatement();
    case TOK_FOR:
      return parseForStatement();
    case TOK_WHILE:
      return parseWhileStatement();
    case TOK_RETURN:
      statementTree = parseReturnStatement();
      //get next token
      getNextToken();
```

```
      //expecting ;
      if(getCurrentToken().getTokenType()!=TokenType.TOK_SEMICOLON)
        //not as expected
        throw new java.lang.RuntimeException("expected semicolon,; , in line " +
    getCurrentToken().getLineNumber());
      return statementTree;
    case TOK_FN:
      return parseFunctionDeclaration();
    case TOK_RIGHT_CURLY_BRACKET:
      return parseBlock();
    default:
      throw new java.lang.RuntimeException(" in line "+getCurrentToken().getLineNumber()
      +". No statement begins with "+getCurrentToken().getLexeme());
    }
  }
  //parse variable declaration
  private TinyLangAst parseVariableDeclaration() {
    //create variable declaration syntax tree
    TinyLangAst variableDeclarationTree = new TinyLangAst(TinyLangAstNodes.
    AST_VARIABLE_DECLARATION_NODE, getCurrentToken().getLineNumber());
    //expect token let
    if(getCurrentToken().getTokenType()!=TokenType.TOK_LET)
      throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+ " in
    line " + getCurrentToken().getLineNumber());

    //expect that next token is identifier
    Token identifier = getNextToken();
    //check if identifier
    if(getCurrentToken().getTokenType()!=TokenType.TOK_IDENTIFIER)
      throw new java.lang.RuntimeException(getCurrentToken().getLexeme()+ " in line " +
    getCurrentToken().getLineNumber()+" is not a valid variable name");
    //get next token
    getNextToken();
    //expect :
    if(getCurrentToken().getTokenType()!=TokenType.TOK_COLON)
      throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+ " in
    line " + getCurrentToken().getLineNumber());
    //get next token
    getNextToken();
    //expect type tree
    variableDeclarationTree.addSubtree(parseType());

    //add identifier
    variableDeclarationTree.addChild(TinyLangAstNodes.AST_IDENTIFIER_NODE,identifier.getLexeme
    (),identifier.getLineNumber());
    //get next token
    getNextToken();
    //expect =
    if(getCurrentToken().getTokenType()!=TokenType.TOK_EQUAL)
      throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+ " in
    line " + getCurrentToken().getLineNumber());
    //get next token
    getNextToken();
    variableDeclarationTree.addSubtree(parseExpression());
    return variableDeclarationTree;
  }
  //parse assignment
  private TinyLangAst parseAssignment() {
    //create assignment syntax tree
    TinyLangAst assignmentTree = new TinyLangAst(TinyLangAstNodes.AST_ASSIGNMENT_NODE,
    getCurrentToken().getLineNumber());
```

```
    //expect identifier
    if(getCurrentToken().getTokenType()!=TokenType.TOK_IDENTIFIER) {
      throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+" in line " + getCurrentToken().getLineNumber());
    }
      //add identifier node
    assignmentTree.addChild(TinyLangAstNodes.AST_IDENTIFIER_NODE,getCurrentToken().getLexeme(),getCurrentToken().getLineNumber());
    //get next token
    getNextToken();
    //expect equal
    if(getCurrentToken().getTokenType()!=TokenType.TOK_EQUAL)
      throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+" in line " + getCurrentToken().getLineNumber());
    //get next token
    getNextToken();
    //expect expression
    assignmentTree.addSubtree(parseExpression());
    return assignmentTree;
  }
  //parse print statement
  private TinyLangAst parsePrintStatement() {
    //create assignment syntax tree
    TinyLangAst printStatementTree = new TinyLangAst(TinyLangAstNodes.AST_PRINT_STATEMENT_NODE,getCurrentToken().getLineNumber());
    //expect print keyword
    if(getCurrentToken().getTokenType()!=TokenType.TOK_PRINT)
      throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+" in line " + getCurrentToken().getLineNumber());
    //get next token
    getNextToken();
    //expect expression
    printStatementTree.addSubtree(parseExpression());
    return printStatementTree;
  }
  //parse if statement
  private TinyLangAst parseIfStatement() {
    //create if statement syntax tree
    TinyLangAst ifStatementTree = new TinyLangAst(TinyLangAstNodes.AST_IF_STATEMENT_NODE,getCurrentToken().getLineNumber());
    //expect if keyword
    if(getCurrentToken().getTokenType()!=TokenType.TOK_IF)
      throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+" in line " + getCurrentToken().getLineNumber());
    //get next token
    getNextToken();
    //expect (
    if(getCurrentToken().getTokenType()!=TokenType.TOK_LEFT_ROUND_BRACKET)
      throw new java.lang.RuntimeException("expected left round bracket,( , in line "+getCurrentToken().getLineNumber());
    //get next token
    getNextToken();
    //add expression subtree to if statement tree
    ifStatementTree.addSubtree(parseExpression());
    //get next token
    getNextToken();
    //expected )
    if(getCurrentToken().getTokenType()!=TokenType.TOK_RIGHT_ROUND_BRACKET)
      throw new java.lang.RuntimeException("expected left round bracket,( , in line "+getCurrentToken().getLineNumber());
    //get next token
```

```
    getNextToken();
    //parse block
    ifStatementTree.addSubtree(parseBlock());
    //getNextToken()
    getNextToken();
    //if we have else condition
    if(getCurrentToken().getTokenType()==TokenType.TOK_ELSE) {
      //get next token
      getNextToken();
      //get else block
      ifStatementTree.addSubtree(parseElseBlock());
    }
    else
      //get previous token
      getPrevToken();
    //return if statement tree
    return ifStatementTree;
  }
  //parse for statement
  private TinyLangAst parseForStatement() {
    //create block syntax tree
    TinyLangAst forStatementTree = new TinyLangAst(TinyLangAstNodes.AST_FOR_STATEMENT_NODE,
    getCurrentToken().getLineNumber());
    //expect for keywordF
    if(getCurrentToken().getTokenType()!=TokenType.TOK_FOR)
      //not as expected
      throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+" in
    line " + getCurrentToken().getLineNumber());
    //get next token
    getNextToken();
    //expect (
    if(getCurrentToken().getTokenType()!=TokenType.TOK_LEFT_ROUND_BRACKET)
      //not as expected
      throw new java.lang.RuntimeException("expected left round bracket,( , in line "+
    getCurrentToken().getLineNumber());
    //get next token
    getNextToken();
    //expect semicolon or variable declaration
    if(getCurrentToken().getTokenType()!=TokenType.TOK_SEMICOLON)
    {
      //expect variable declaration
      forStatementTree.addSubtree(parseVariableDeclaration());
      //consume variable declaration
      getNextToken();
    }
    //expect ;
    if(getCurrentToken().getTokenType()!=TokenType.TOK_SEMICOLON)
      throw new java.lang.RuntimeException("expected semicolon,; , in line "+ getCurrentToken
    ().getLineNumber());
    //get next token
    getNextToken();
    //expect expression
    forStatementTree.addSubtree(parseExpression());
    //expect ;
    if(getNextToken().getTokenType()!=TokenType.TOK_SEMICOLON)
      throw new java.lang.RuntimeException("expected semicolon,; , in line "+ getCurrentToken
    ().getLineNumber());
    //expect right round bracket or assignment
    if(getNextToken().getTokenType()!=TokenType.TOK_RIGHT_ROUND_BRACKET)
    {
      //expect variable declaration
```

```
      forStatementTree.addSubtree(parseAssignment());
      //consume variable declaration
      getNextToken();
    }
    if(getCurrentToken().getTokenType()!=TokenType.TOK_RIGHT_ROUND_BRACKET)
      throw new java.lang.RuntimeException("expected right round bracket,) , in line "+
    getCurrentToken().getLineNumber());
    //get next token
    getNextToken();
    //expect block
    forStatementTree.addSubtree(parseBlock());
    //return for statement tree
    return forStatementTree;
  }
  //parse while statement
  private TinyLangAst parseWhileStatement() {
    //create while statement syntax tree syntax tree
    TinyLangAst whileStatementTree = new TinyLangAst(TinyLangAstNodes.AST_WHILE_STATEMENT_NODE
    ,getCurrentToken().getLineNumber());
    //expect while keyword
    if(getCurrentToken().getTokenType()!=TokenType.TOK_WHILE)
      //not as expected
      throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+ " in
    line " + getCurrentToken().getLineNumber());
    //get next token
    getNextToken();
    //expect (
    if(getCurrentToken().getTokenType()!=TokenType.TOK_LEFT_ROUND_BRACKET)
      //not as expected
      throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+ " in
    line " + getCurrentToken().getLineNumber());
    //get next token
    getNextToken();
    //expect expression
    whileStatementTree.addSubtree(parseExpression());
    //get next token
    getNextToken();
    //expect )
    if(getCurrentToken().getTokenType()!=TokenType.TOK_RIGHT_ROUND_BRACKET)
      //not as expected
      throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+ " in
    line " + getCurrentToken().getLineNumber());
    //get next token
    getNextToken();
    //expect block
    whileStatementTree.addSubtree(parseBlock());
    //return syntax tree
    return whileStatementTree;
  }
  //parse return statement
  private TinyLangAst parseReturnStatement() {
    //create while statement syntax tree syntax tree
    TinyLangAst returnStatementTree = new TinyLangAst(TinyLangAstNodes.
    AST_RETURN_STATEMENT_NODE, getCurrentToken().getLineNumber());

    //expect return keyword
    if(getCurrentToken().getTokenType()!=TokenType.TOK_RETURN)
      //not as expected
      throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+ " in
    line " + getCurrentToken().getLineNumber());
    //get next token
```

```
    getNextToken();
    //expect expression
    returnStatementTree.addSubtree(parseExpression());
    //return syntax tree
    return returnStatementTree;
  }
  //parse function declaration
  private TinyLangAst parseFunctionDeclaration() {
    //create function declaration syntax tree syntax tree
    TinyLangAst functionDeclarationTree = new TinyLangAst(TinyLangAstNodes.AST_FUNCTION_DECLARATION_NODE, getCurrentToken().getLineNumber());
    //expect return keyword
    if(getCurrentToken().getTokenType()!=TokenType.TOK_FN)
      //not as expected
      throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+" in line " + getCurrentToken().getLineNumber());
    //get next token
    getNextToken();
    //expect expression
    Token identifier;
    if(getCurrentToken().getTokenType()==TokenType.TOK_IDENTIFIER)
      identifier = getCurrentToken();
    else
      //not valid function name
      throw new java.lang.RuntimeException(getCurrentToken().getLexeme()+" in line " + getCurrentToken().getLineNumber()+" not a valid funciton name");
    //get next token
    getNextToken();
    //expect (
    if(getCurrentToken().getTokenType()!=TokenType.TOK_LEFT_ROUND_BRACKET)
      //not as expected
      throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+" in line " + getCurrentToken().getLineNumber());
    //get next token
    getNextToken();
    //expect 0 or more formal parameters
    TinyLangAst formalParamsSubtree;
    //if not right round bracket -> we have parameters
    if(getCurrentToken().getTokenType()!=TokenType.TOK_RIGHT_ROUND_BRACKET){
      formalParamsSubtree = parseFormalParams();
      //get next token (expected round bracket in next token)
      getNextToken();
    }
    else
      //add parameter node
      formalParamsSubtree = new TinyLangAst(TinyLangAstNodes.AST_FORMAL_PARAMETERS_NODE, getCurrentToken().getLineNumber());
    //expect )

    if(getCurrentToken().getTokenType()!=TokenType.TOK_RIGHT_ROUND_BRACKET)
      //not as expected
      throw new java.lang.RuntimeException("expected right round bracket,) ,"+" in line "+ getCurrentToken().getLineNumber());
    //get next token
    getNextToken();

    //expect ->
    if(getCurrentToken().getTokenType()!=TokenType.TOK_RIGHT_ARROW)
      //not as expected
      throw new java.lang.RuntimeException("expected right arrow,-> ,in line "+getCurrentToken().getLineNumber());
```

```
    //get next token
    getNextToken();
    //parse type

    TinyLangAst typeSubtree = parseType();
    //get next token
    getNextToken();

    //parse block
    TinyLangAst blockSubtree = parseBlock();
    //add type subtree to function declaration tree
    functionDeclarationTree.addSubtree(typeSubtree);
    //add identifier node to function declaration tree
    functionDeclarationTree.addChild(TinyLangAstNodes.AST_IDENTIFIER_NODE,identifier.getLexeme(),identifier.getLineNumber());
    //add formal parameters subtree to function declaration tree
    functionDeclarationTree.addSubtree(formalParamsSubtree);
    //add block subtree to function declaration tree
    functionDeclarationTree.addSubtree(blockSubtree);
    //return function declaration tree
    return functionDeclarationTree;
  }
  //parse block
  private TinyLangAst parseBlock() {
    //create block syntax tree
    TinyLangAst blockTree = new TinyLangAst(TinyLangAstNodes.AST_BLOCK_NODE,getCurrentToken().getLineNumber());
    //expected {
    //set line number
    blockTree.setLineNumber(getCurrentToken().getLineNumber());
    if(getCurrentToken().getTokenType() != TokenType.TOK_LEFT_CURLY_BRACKET)
      //not as expected
      throw new java.lang.RuntimeException("expected { in line "+getCurrentToken().getLineNumber() );
    // get next token
    getNextToken();
    // we may have one or more statements
    // block ends using }
    while(getCurrentToken().getTokenType()!=TokenType.TOK_RIGHT_CURLY_BRACKET &&
        getCurrentToken().getTokenType()!=TokenType.TOK_EOF) {
      // parse statement one by one

      blockTree.addSubtree(parseStatement());
      // get next token
      getNextToken();
    }
    //expected }
    if(getCurrentToken().getTokenType()!=TokenType.TOK_RIGHT_CURLY_BRACKET)
      //not as expected
      throw new java.lang.RuntimeException("expected } in line "+getCurrentToken().getLineNumber());
    //return block tree
    return blockTree;
  }
  //parse else block
  private TinyLangAst parseElseBlock() {
    //create block syntax tree
    TinyLangAst elseBlockTree = new TinyLangAst(TinyLangAstNodes.AST_ELSE_BLOCK_NODE,getCurrentToken().getLineNumber());
    //expected {
    if(getCurrentToken().getTokenType() != TokenType.TOK_LEFT_CURLY_BRACKET)
```

```
      //not as expected
      throw new java.lang.RuntimeException("expected { in line "+getCurrentToken().
    getLineNumber() );
    // get next token
    getNextToken();
    // we may have one or more statements
    // block ends using }
    while(getCurrentToken().getTokenType()!=TokenType.TOK_RIGHT_CURLY_BRACKET &&
        getCurrentToken().getTokenType()!=TokenType.TOK_EOF) {
      // parse statement one by one
      elseBlockTree.addSubtree(parseStatement());
      // get next token
      getNextToken();

    }
    //expected }
    if(getCurrentToken().getTokenType()!=TokenType.TOK_RIGHT_CURLY_BRACKET)
      //not as expected
      throw new java.lang.RuntimeException("expected } in line "+getCurrentToken().
    getLineNumber());
    //return else block syntax tree
    return elseBlockTree;
  }
  //parse type
  private TinyLangAst parseType() {
    // add node
    switch(getCurrentToken().getTokenType()) {
      case TOK_BOOL_TYPE:
        return new TinyLangAst(TinyLangAstNodes.AST_TYPE_NODE,Type.BOOL.toString(),
    getCurrentToken().getLineNumber());
      case TOK_INT_TYPE:
        return new TinyLangAst(TinyLangAstNodes.AST_TYPE_NODE,Type.INTEGER.toString(),
    getCurrentToken().getLineNumber());
      case TOK_FLOAT_TYPE:
        return new TinyLangAst(TinyLangAstNodes.AST_TYPE_NODE,Type.FLOAT.toString(),
    getCurrentToken().getLineNumber());
      case TOK_CHAR_TYPE:
        return new TinyLangAst(TinyLangAstNodes.AST_TYPE_NODE,Type.CHAR.toString(),
    getCurrentToken().getLineNumber());
      default:
        throw new java.lang.RuntimeException(getCurrentToken().getLexeme()+ " in line " +
    getCurrentToken().getLineNumber() +" is not a valid type" );
    }
  }
  //parse expression
  private TinyLangAst parseExpression() {
    //parse simple expression
    TinyLangAst left = parseSimpleExpression();
    //get next token
    getNextToken();
    //expecting 0 or more expressions separated by a relational operator
    if(getCurrentToken().getTokenType()==TokenType.TOK_RELATIONAL_OP) {
      //create a binary expression tree with root node containing current binary operator
      TinyLangAst binaryExpressionTree = new TinyLangAst(TinyLangAstNodes.
    AST_BINARY_OPERATOR_NODE,getCurrentToken().getLexeme(),getCurrentToken().getLineNumber())
    ;
      //add left operand of the binary operator
      binaryExpressionTree.addSubtree(left);
      //move to next token
      getNextToken();
      //add right operand
```

```
            binaryExpressionTree.addSubtree(parseExpression());
            return binaryExpressionTree;
        }
        getPrevToken();
        //case of no relational operator
        return left;
    }
    //parse simple expression
    private TinyLangAst parseSimpleExpression() {
        //parse simple expression
        TinyLangAst left = parseTerm();
        //get next token
        getNextToken();
        //expecting 0 or more expressions separated by a relational operator
        if(getCurrentToken().getTokenType()==TokenType.TOK_ADDITIVE_OP) {
            //create a binary expression tree with root node containing current binary operator
            TinyLangAst binaryExpressionTree = new TinyLangAst(TinyLangAstNodes.
        AST_BINARY_OPERATOR_NODE,getCurrentToken().getLexeme(),getCurrentToken().getLineNumber())
        ;
            //add left operand of the binary operator
            binaryExpressionTree.addSubtree(left);
            //move to next token
            getNextToken();
            //add right operand
            binaryExpressionTree.addSubtree(parseSimpleExpression());
            return binaryExpressionTree;
        }
        getPrevToken();
        //case of no relational operator
        return left;
    }
    //parse term
    private TinyLangAst parseTerm() {
        //parse factor
        TinyLangAst left = parseFactor();
        //get next token
        getNextToken();
        //expecting 0 or more expressions separated by a multiplicative operator
        if(getCurrentToken().getTokenType()==TokenType.TOK_MULTIPLICATIVE_OP) {
            //create a binary expression tree with root node containing current binary operator
            TinyLangAst binaryExpressionTree = new TinyLangAst(TinyLangAstNodes.
        AST_BINARY_OPERATOR_NODE,getCurrentToken().getLexeme(),getCurrentToken().getLineNumber())
        ;
            //add left operand of the binary operator
            binaryExpressionTree.addSubtree(left);
            //move to next token
            getNextToken();
            //add right operand
            binaryExpressionTree.addSubtree(parseTerm());
            return binaryExpressionTree;
        }
        getPrevToken();
        //case of no relational operator
        return left;
    }
    //parse term
    private TinyLangAst parseFactor() {
        switch(getCurrentToken().getTokenType()) {
        //literals
        case TOK_BOOL_LITERAL:
            return new TinyLangAst(TinyLangAstNodes.AST_BOOLEAN_LITERAL_NODE,getCurrentToken().
```

```
getLexeme(),getCurrentToken().getLineNumber());
    case TOK_INT_LITERAL:
      return new TinyLangAst(TinyLangAstNodes.AST_INTEGER_LITERAL_NODE,getCurrentToken().
getLexeme(),getCurrentToken().getLineNumber());
    case TOK_FLOAT_LITERAL:
      return new TinyLangAst(TinyLangAstNodes.AST_FLOAT_LITERAL_NODE,getCurrentToken().
getLexeme(),getCurrentToken().getLineNumber());
    case TOK_CHAR_LITERAL:
      return new TinyLangAst(TinyLangAstNodes.AST_CHAR_LITERAL_NODE,getCurrentToken().
getLexeme(),getCurrentToken().getLineNumber());
    //identifier or function call
    case TOK_IDENTIFIER:
      getNextToken();
      if(getCurrentToken().getTokenType()==TokenType.TOK_LEFT_ROUND_BRACKET){
        getPrevToken();
        return parseFunctionCall();
      }
      else {
        getPrevToken();
        return new TinyLangAst(TinyLangAstNodes.AST_IDENTIFIER_NODE,getCurrentToken().
getLexeme(),getCurrentToken().getLineNumber());
      }
    case TOK_LEFT_ROUND_BRACKET:
      return parseSubExpression();
    case TOK_ADDITIVE_OP:
    case TOK_NOT:
      return parseUnary();
    default:
      throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+" in
line "+getCurrentToken().getLineNumber());
    }
  }
  private TinyLangAst parseSubExpression() {
    //expect left round bracket
    if(getCurrentToken().getTokenType()!=TokenType.TOK_LEFT_ROUND_BRACKET)
      throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+" in
line "+getCurrentToken().getLineNumber());
    //get next token
    getNextToken();
    //expect expression
    TinyLangAst expressionTree = parseExpression();

    //get next token
    getNextToken();
    //expect right round bracket
    if(getCurrentToken().getTokenType()!=TokenType.TOK_RIGHT_ROUND_BRACKET)
      throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+" in
line "+getCurrentToken().getLineNumber());
    //return expression tree
    return expressionTree;
  }
  private TinyLangAst parseUnary() {

    //expect not or additive
    if(getCurrentToken().getTokenType()!=TokenType.TOK_ADDITIVE_OP && getCurrentToken().
getTokenType()!=TokenType.TOK_NOT)
      throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+" in
line "+getCurrentToken().getLineNumber());
    //create unary tree with unary operator
    TinyLangAst unaryTree = new TinyLangAst(TinyLangAstNodes.AST_UNARY_OPERATOR_NODE,
getCurrentToken().getLexeme(),getCurrentToken().getLineNumber());
```

```
    //get next token
    getNextToken();
    //expect expression
    unaryTree.addSubtree(parseExpression());

    return unaryTree;
  }
  private TinyLangAst parseFunctionCall() {
    TinyLangAst functionCallTree = new TinyLangAst(TinyLangAstNodes.AST_FUNCTION_CALL_NODE, getCurrentToken().getLineNumber());
    if(getCurrentToken().getTokenType()!=TokenType.TOK_IDENTIFIER)
      throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+" in line"+getCurrentToken().getLineNumber());
    //add identifier node
    functionCallTree.addChild(TinyLangAstNodes.AST_IDENTIFIER_NODE,getCurrentToken().getLexeme(),getCurrentToken().getLineNumber());
    getNextToken();
    if(getCurrentToken().getTokenType()!=TokenType.TOK_LEFT_ROUND_BRACKET)
      //not as expected
      throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+" in line"+getCurrentToken().getLineNumber());

    getNextToken();
    //if not right round bracket -> we have parameters
    if(getCurrentToken().getTokenType()!=TokenType.TOK_RIGHT_ROUND_BRACKET){
      functionCallTree.addSubtree(parseActualParams());
      //get next token (expected round bracket in next token)
      getNextToken();
    }
    else
      //add parameter node
      functionCallTree.addChild(TinyLangAstNodes.AST_ACTUAL_PARAMETERS_NODE,getCurrentToken().getLineNumber());


    if(getCurrentToken().getTokenType()!=TokenType.TOK_RIGHT_ROUND_BRACKET)
      //not as expected
      throw new java.lang.RuntimeException("expected right round bracket,) ,"+getCurrentToken().getLexeme()+" in line"+getCurrentToken().getLineNumber());
    return functionCallTree;
  }
  TinyLangAst parseActualParams() {
    //parse expression
    TinyLangAst actualParamsTree = new TinyLangAst(TinyLangAstNodes.AST_ACTUAL_PARAMETERS_NODE,getCurrentToken().getLineNumber());
    //add expression tree
    actualParamsTree.addSubtree(parseExpression());
    //get next token
    getNextToken();

    while(getCurrentToken().getTokenType()==TokenType.TOK_COMMA && getCurrentToken().getTokenType()!=TokenType.TOK_EOF )
    {
      //get next token
      getNextToken();
      actualParamsTree.addSubtree(parseExpression());
      //get next token
      getNextToken();
    }
    getPrevToken();
    return actualParamsTree;
```

```
  }
  //parse formal parameters
  TinyLangAst parseFormalParams() {
    //parse expression
    TinyLangAst formalParamsTree = new TinyLangAst(TinyLangAstNodes.AST_FORMAL_PARAMETERS_NODE
    ,getCurrentToken().getLineNumber());
    //add formal param tree
    formalParamsTree.addSubtree(parseFormalParam());
    //get next token
    getNextToken();

    while(getCurrentToken().getTokenType()==TokenType.TOK_COMMA)
    {
      //get next token
      getNextToken();
      formalParamsTree.addSubtree(parseFormalParam());
      //get next token
      getNextToken();
    }
    getPrevToken();

    return formalParamsTree;
  }
  //parse formal parameter
  TinyLangAst parseFormalParam() {
    //parse expression
    TinyLangAst formalParamTree = new TinyLangAst(TinyLangAstNodes.AST_FORMAL_PARAMETER_NODE,
    getCurrentToken().getLineNumber());
    //expect identifier
    if(getCurrentToken().getTokenType()!=TokenType.TOK_IDENTIFIER)
      throw new java.lang.RuntimeException(getCurrentToken().getLexeme()+" in line "+
    getCurrentToken().getLineNumber()+" is not a valid parameter name");
    //add identifier node
    Token identifier = getCurrentToken();
    //get next token
    getNextToken();
    // expect :
    if(getCurrentToken().getTokenType()!=TokenType.TOK_COLON)
      //not as expected
      throw new java.lang.RuntimeException("unexpected "+getCurrentToken().getLexeme()+" in
    line"+getCurrentToken().getLineNumber());
    //get next token
    getNextToken();
    formalParamTree.addSubtree(parseType());
    formalParamTree.addChild(TinyLangAstNodes.AST_IDENTIFIER_NODE, identifier.getLexeme(),
    identifier.getLineNumber());
    return formalParamTree;
  }
  public TinyLangAst getTinyLangAbstraxSyntaxTree() {
    return tinyLangProgramAbstractSyntaxTree;
  }
}
```

Listing 6.10: Implementation of recursive descent parser

6.3 | XML generation

```
package tinylangvisitor;
import tinylangparser.TinyLangAst;
import tinylangparser.TinyLangAstNodes;
public class XmlGeneration implements Visitor {
  private String xmlRepresentation = "";
  private int indentation = 0;
  private String getCurrentIndentationLevel() {
    String indentation = "";
    for(int i=0;i<this.indentation;i++)
      //add indentation
      indentation+="    ";//(char)0x09;
    return indentation;
  }
  //method which runs statement type visit method based on node type
  public void visitStatement(TinyLangAst tinyLangAst) {
    switch(tinyLangAst.getAssociatedNodeType()) {
    case AST_VARIABLE_DECLARATION_NODE:
      visitVariableDeclarationNode(tinyLangAst);
      break;
    case AST_ASSIGNMENT_NODE:
      visitAssignmentNode(tinyLangAst);
      break;
    case AST_PRINT_STATEMENT_NODE:
      visitPrintStatementNode(tinyLangAst);
      break;
    case AST_IF_STATEMENT_NODE:
      visitIfStatementNode(tinyLangAst);
      break;
    case AST_FOR_STATEMENT_NODE:
      visitForStatementNode(tinyLangAst);
      break;
    case AST_WHILE_STATEMENT_NODE:
      visitWhileStatementNode(tinyLangAst);
      break;
    case AST_RETURN_STATEMENT_NODE:
      visitReturnStatementNode(tinyLangAst);
      break;
    case AST_FUNCTION_DECLARATION_NODE:
      visitFunctionDeclarationNode(tinyLangAst);
      break;
    case AST_BLOCK_NODE:
      visitBlockNode(tinyLangAst);
      break;
    default:
      throw new java.lang.RuntimeException("Unrecognised statement of  type "+tinyLangAst.
    getAssociatedNodeType());
    }
  }
  private void visitExpression(TinyLangAst tinyLangAst){
    switch(tinyLangAst.getAssociatedNodeType()) {
    case AST_BINARY_OPERATOR_NODE:
      visitBinaryOperatorNode(tinyLangAst);
      break;
    case AST_UNARY_OPERATOR_NODE:
      visitUnaryOperatorNode(tinyLangAst);
      break;
    case AST_BOOLEAN_LITERAL_NODE:
      visitBooleanLiteralNode(tinyLangAst);
      break;
    case AST_INTEGER_LITERAL_NODE:
      visitIntegerLiteralNode(tinyLangAst);
```

```java
        break;
      case AST_FLOAT_LITERAL_NODE:
        visitFloatLiteralNode(tinyLangAst);
        break;
      case AST_CHAR_LITERAL_NODE:
        visitCharLiteralNode(tinyLangAst);
        break;
      case AST_IDENTIFIER_NODE:
        visitIdentifierNode(tinyLangAst);
        break;
      case AST_FUNCTION_CALL_NODE:
        visitFunctionCallNode(tinyLangAst);
        break;
      default:
        throw new java.lang.RuntimeException("Unrecognised expression node of type "+tinyLangAst.getAssociatedNodeType());
    }
  }
  public XmlGeneration(TinyLangAst tinyLangAst) {
    visitTinyLangProgram(tinyLangAst);
  }

  @Override
  public void visitTinyLangProgram(TinyLangAst tinyLangAst) {
    xmlRepresentation+=getCurrentIndentationLevel()+"<TinyLangProgram>\n";
    //indent
    indentation++;
    for(TinyLangAst child : tinyLangAst.getChildren())
      visitStatement(child);
    //unindent
    indentation--;
    xmlRepresentation+=getCurrentIndentationLevel()+"<\\TinyLangProgram>\n";
  }
  @Override
  public void visitVariableDeclarationNode(TinyLangAst tinyLangAst) {

    xmlRepresentation+=getCurrentIndentationLevel()+"<variable declaration>\n";
    //indent
    indentation++;
    //visit children
    //add function identifier
    xmlRepresentation+=getCurrentIndentationLevel()+"<id type=\""+tinyLangAst.getChildren().get(0).getAssociatedNodeValue()+"\">"+tinyLangAst.getChildren().get(1).getAssociatedNodeValue()+"<\\id>\n";
    //add expression tag
    visitExpression(tinyLangAst.getChildren().get(2));
    //unindent
    indentation--;
    xmlRepresentation+=getCurrentIndentationLevel()+"<\\variable declaration>\n";
  }

  @Override
  public void visitPrintStatementNode(TinyLangAst tinyLangAst) {
    xmlRepresentation+=getCurrentIndentationLevel()+"<print statement>\n";
    //indent
    indentation++;
    visitExpression(tinyLangAst.getChildren().get(0));
    //unindent
    indentation--;
    xmlRepresentation+=getCurrentIndentationLevel()+"<\\print statement>\n";
  }
```

```java
  @Override
  public void visitIfStatementNode(TinyLangAst tinyLangAst) {
    xmlRepresentation+=getCurrentIndentationLevel()+"<if statement>\n";
    //indent
    indentation++;
    //expect first child to be expression
    visitExpression(tinyLangAst.getChildren().get(0));
    //expect second child to be block
    visitBlockNode(tinyLangAst.getChildren().get(1));
    //check if we have else block
    if(tinyLangAst.getChildren().size()==3)
      visitBlockNode(tinyLangAst.getChildren().get(2));
    //unindent
    indentation--;
    xmlRepresentation+=getCurrentIndentationLevel()+"<\\if statement>\n";
  }
  @Override
  public void visitForStatementNode(TinyLangAst tinyLangAst) {
    //add for statement tag
    xmlRepresentation+=getCurrentIndentationLevel()+"<for statement>\n";
    //indent
    indentation++;
    //expect first child is variable declaration or expression
    if(tinyLangAst.getChildren().get(0).getAssociatedNodeType()==TinyLangAstNodes.AST_VARIABLE_DECLARATION_NODE)
      visitVariableDeclarationNode(tinyLangAst.getChildren().get(0));

    else
      visitExpression(tinyLangAst.getChildren().get(0));

    //second child is assignment or block or expression
    if(tinyLangAst.getChildren().get(1).getAssociatedNodeType()==TinyLangAstNodes.AST_ASSIGNMENT_NODE)
      visitAssignmentNode(tinyLangAst.getChildren().get(1));
    else if(tinyLangAst.getChildren().get(1).getAssociatedNodeType()==TinyLangAstNodes.AST_BLOCK_NODE)
      visitBlockNode(tinyLangAst.getChildren().get(1));
    else
      visitExpression(tinyLangAst.getChildren().get(1));
    //if we have 3 or more children
    if(tinyLangAst.getChildren().size()>=3 && tinyLangAst.getChildren().get(2).getAssociatedNodeType()==TinyLangAstNodes.AST_ASSIGNMENT_NODE)
      visitAssignmentNode(tinyLangAst.getChildren().get(2));
    else if(tinyLangAst.getChildren().size()>=3 && tinyLangAst.getChildren().get(2).getAssociatedNodeType()==TinyLangAstNodes.AST_BLOCK_NODE)
      visitBlockNode(tinyLangAst.getChildren().get(2));
    else
      throw new java.lang.RuntimeException("unexpected node of type "+tinyLangAst.getChildren().get(2).getAssociatedNodeType());
    //if we have 4 children
    if(tinyLangAst.getChildren().size()==4)
      visitBlockNode(tinyLangAst.getChildren().get(3));
    indentation--;
    xmlRepresentation+=getCurrentIndentationLevel()+"<\\VariableDeclaration>\n";
  }
  @Override
  public void visitWhileStatementNode(TinyLangAst tinyLangAst) {
    xmlRepresentation+=getCurrentIndentationLevel()+"<while statement>\n";
    //indent
    indentation++;
    //expected 2 children expression and nodes
```

```
        if(tinyLangAst.getChildren().size()!=2)
            throw new java.lang.RuntimeException("while statement node has "+tinyLangAst.getChildren().size()+" expected 2");
        if(tinyLangAst.getChildren().get(1).getAssociatedNodeType()!=TinyLangAstNodes.AST_BLOCK_NODE)
            throw new java.lang.RuntimeException("second child of while statement is "+tinyLangAst.getChildren().get(1).getAssociatedNodeType()+" expected AST_BLOCK_NODE");
        //visit expression and block
        visitExpression(tinyLangAst.getChildren().get(0));
        visitBlockNode(tinyLangAst.getChildren().get(1));
        indentation--;
        xmlRepresentation+=getCurrentIndentationLevel()+"<\\while statement>\n";
    }
    @Override
    public void visitReturnStatementNode(TinyLangAst tinyLangAst) {
        xmlRepresentation+=getCurrentIndentationLevel()+"<return statement>\n";
        //indent
        indentation++;
        //visit expression
        visitExpression(tinyLangAst.getChildren().get(0));
        indentation--;
        xmlRepresentation+=getCurrentIndentationLevel()+"<\\return statement>\n";
    }
    @Override
    public void visitFunctionDeclarationNode(TinyLangAst tinyLangAst) {
        xmlRepresentation+=getCurrentIndentationLevel()+"<function declaration>\n";
        //expected 4 children of types identifier,formal parameters,type and block
        //indent
        indentation++;
        //add function identifier
        xmlRepresentation+=getCurrentIndentationLevel()+"<id type=\""+tinyLangAst.getChildren().get(0).getAssociatedNodeValue()+"\">"+tinyLangAst.getChildren().get(1).getAssociatedNodeValue()+"<\\id>\n";
        //add parameters
        xmlRepresentation+=getCurrentIndentationLevel()+"<parameters>\n";
            indentation++;
            for(TinyLangAst child : tinyLangAst.getChildren().get(2).getChildren()) {
                xmlRepresentation+=getCurrentIndentationLevel()+"<parameters>\n";
                indentation++;
                xmlRepresentation+=getCurrentIndentationLevel()+"<id type=\""+child.getChildren().get(0).getAssociatedNodeValue()+"\">"+child.getChildren().get(1).getAssociatedNodeValue()+"<\\id>\n";
                indentation--;
                xmlRepresentation+=getCurrentIndentationLevel()+"<\\parameters>\n";
            }
            indentation--;
        xmlRepresentation+=getCurrentIndentationLevel()+"<\\parameters>\n";

            visitBlockNode(tinyLangAst.getChildren().get(3));
        //unindent
        indentation--;
        xmlRepresentation+=getCurrentIndentationLevel()+"<\\function declaration>\n";

    }

    @Override
    public void visitFunctionCallNode(TinyLangAst tinyLangAst) {
        xmlRepresentation+=getCurrentIndentationLevel()+"<function call>\n";
        //expected 4 children of types identifier,formal parameters,type and block
        //indent
        indentation++;
```

```
    //add function identifier
    visitIdentifierNode(tinyLangAst.getChildren().get(0));
    //add parameters
    xmlRepresentation+=getCurrentIndentationLevel()+"<parameters>\n";
      indentation++;
      for(TinyLangAst child : tinyLangAst.getChildren().get(1).getChildren()) {
        xmlRepresentation+=getCurrentIndentationLevel()+"<actual parameter>\n";
        indentation++;
        visitExpression(child);
        indentation--;
        xmlRepresentation+=getCurrentIndentationLevel()+"<\\actual parameter>\n";
      }
      indentation--;
    xmlRepresentation+=getCurrentIndentationLevel()+"<\\parameters>\n";

    //unindent
    indentation--;
    xmlRepresentation+=getCurrentIndentationLevel()+"<\\function call>\n";

  }
  @Override
  public void visitBlockNode(TinyLangAst tinyLangAst) {
    if(tinyLangAst.getAssociatedNodeType()==TinyLangAstNodes.AST_ELSE_BLOCK_NODE)
      xmlRepresentation+=getCurrentIndentationLevel()+"<else block>\n";
    else
      xmlRepresentation+=getCurrentIndentationLevel()+"<block>\n";
    //indent
    indentation++;
    //children are statements
    for(TinyLangAst child:tinyLangAst.getChildren())
      visitStatement(child);
    indentation--;
    if(tinyLangAst.getAssociatedNodeType()==TinyLangAstNodes.AST_ELSE_BLOCK_NODE)
      xmlRepresentation+=getCurrentIndentationLevel()+"<\\else block>\n";
    else
      xmlRepresentation+=getCurrentIndentationLevel()+"<\\block>\n";

  }
  @Override
  public void visitBinaryOperatorNode(TinyLangAst tinyLangAst) {
    xmlRepresentation+=getCurrentIndentationLevel()+"<binary Op=\""+tinyLangAst.
    getAssociatedNodeValue()+"\">\n";
    //expected binary operator -> 2 children expression
    if(tinyLangAst.getChildren().size()!=2)
      throw new java.lang.RuntimeException("binary node has "+tinyLangAst.getChildren().size()
    +" child(ren) expected 2");
    //indent
    indentation++;
    //visit expression
    visitExpression(tinyLangAst.getChildren().get(0));
    visitExpression(tinyLangAst.getChildren().get(1));

    indentation--;
    xmlRepresentation+=getCurrentIndentationLevel()+"<\\binary>\n";
  }
  @Override
  public void visitUnaryOperatorNode(TinyLangAst tinyLangAst) {
    xmlRepresentation+=getCurrentIndentationLevel()+"<unary Op=\""+tinyLangAst.
    getAssociatedNodeValue()+"\">\n";
    //expected unary expression node -> one child
    if(tinyLangAst.getChildren().size()!=1)
```

```
        throw new java.lang.RuntimeException("unary node has "+tinyLangAst.getChildren().size()
      +" children expected 1");
      //indent
      indentation++;
      //visit expression
        visitExpression(tinyLangAst.getChildren().get(0));
      //unindent
      indentation--;
      xmlRepresentation+=getCurrentIndentationLevel()+"<\\unary>\n";
    }
    @Override
    public void visitBooleanLiteralNode(TinyLangAst tinyLangAst) {
      xmlRepresentation+=getCurrentIndentationLevel()+"<boolean literal>"+tinyLangAst.
      getAssociatedNodeValue()+"<\\boolean literal>\n";
    }
    @Override
    public void visitIntegerLiteralNode(TinyLangAst tinyLangAst) {
      xmlRepresentation+=getCurrentIndentationLevel()+"<integer literal>"+tinyLangAst.
      getAssociatedNodeValue()+"<\\integer literal>\n";
    }
    @Override
    public void visitFloatLiteralNode(TinyLangAst tinyLangAst) {
      xmlRepresentation+=getCurrentIndentationLevel()+"<float literal>"+tinyLangAst.
      getAssociatedNodeValue()+"<\\float literal>\n";
    }
    @Override
    public void visitCharLiteralNode(TinyLangAst tinyLangAst) {
      xmlRepresentation+=getCurrentIndentationLevel()+"<char literal>"+tinyLangAst.
      getAssociatedNodeValue()+"<\\char literal>\n";
    }

    @Override
    public void visitIdentifierNode(TinyLangAst tinyLangAst) {
      xmlRepresentation+=getCurrentIndentationLevel()+"<id>"+tinyLangAst.getAssociatedNodeValue
      ()+"<\\id>\n";
    }

    public void printXmlTree() {
      System.out.println(xmlRepresentation);
    }
//    @Override
//    public void visitElseBlockNode(TinyLangAst tinyLangAst) {
//      xmlRepresentation+=getCurrentIndentationLevel()+"<else block>\n";
//      //indent
//      indentation++;
//      //children are statements
//      for(TinyLangAst child:tinyLangAst.getChildren())
//        visitStatement(child);
//      indentation--;
//      xmlRepresentation+=getCurrentIndentationLevel()+"<\\else block>\n";
//
//    }
//

    public void visitAssignmentNode(TinyLangAst tinyLangAst) {
      xmlRepresentation+=getCurrentIndentationLevel()+"<assignment>\n";
      indentation++;
        //add identifier and expression tags
        visitIdentifierNode(tinyLangAst.getChildren().get(0));
        visitExpression(tinyLangAst.getChildren().get(0));
      indentation--;
```

```
    xmlRepresentation+=getCurrentIndentationLevel()+"<\\assignment>\n";
  }
}
```

Listing 6.11: Generating an XML representation of AST

6.4 | Semantic Analyser

```
package tinylangvisitor;
import java.util.Objects;
import java.util.Stack;
import tinylangparser.Type;
public class FunctionSignature {
  private String functionName ="";
  private int hashCode;
  Stack<Type> parameterType = new Stack<Type>();
  public FunctionSignature(String functionName,Stack<Type> parameterType) {
    //set functionName
    this.functionName=functionName;
    //set parameter types stack
    this.parameterType=parameterType;
    //set hash
    hashCode = Objects.hash(functionName,parameterType);
  }
    public String getFunctionName() {
      return functionName;
    }
    public Stack<Type> getParametersTypes(){
      return parameterType;
    }
  /*
   *functions that allows us to use classes
     *as map keys where 2 object keys are
     *equivalent iff they have same attribute values
     *rather than same object address value
     */
    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;
        FunctionSignature that = (FunctionSignature) o;
        return functionName.equals(that.functionName) && parameterType.equals(that.parameterType);
    }
    @Override
    public int hashCode() {
        return this.hashCode;
    }
}
```

Listing 6.12: Function Signature

```java
package tinylangvisitor;
import java.util.HashMap;
import java.util.Map;
import java.util.Stack;

import tinylangparser.TinyLangAst;
import tinylangparser.Type;
public class Scope {
  //Signature
  //name binding  i.e. name |->  object e.g. variable,function etc
  Map<String,Type> variableDeclaration = new HashMap<String,Type>();
  Map<FunctionSignature,Type> functionDeclaration = new HashMap<FunctionSignature,Type>();
  Map<FunctionSignature,Stack<String>> functionParameterNames = new HashMap<FunctionSignature,Stack<String>>();
  Map<FunctionSignature,TinyLangAst> functionBlock= new HashMap<FunctionSignature,TinyLangAst>();

  // map := variable □ value
  Map<String,String> variableValues = new HashMap<String,String>();

  //  map := function name □ value

  public void addVariableDeclaration(String variableName,Type type) {
    variableDeclaration.put(variableName, type);
  }
  //add function declaration
  public void addFunctionDeclaration(FunctionSignature functionSignature,Type type) {
    functionDeclaration.put(functionSignature, type);
  }
  public TinyLangAst getBlock(FunctionSignature functionSignature) {
    return functionBlock.get(functionSignature);
  }
  public Stack<String> getParameterNames(FunctionSignature functionSignature) {
    return functionParameterNames.get(functionSignature);
  }



  //add value to variable x
  public void addVariableValue(String x,String value) {
    variableValues.put(x, value);
  }
  public void deleteVariable(String variableName) {
    variableValues.remove(variableName);
    variableDeclaration.remove(variableName);
  }
  public void addFunctionParameterNames(FunctionSignature functionSignature, Stack<String> variableNames) {
    functionParameterNames.put(functionSignature, variableNames);
  }
  public void addFunctionBlock(FunctionSignature functionSignature, TinyLangAst block) {
    functionBlock.put(functionSignature, block);
  }

  public boolean isFunctionAlreadyDefined(FunctionSignature functionSignature) {
    return functionDeclaration.containsKey(functionSignature);
  }


  //check if name is binded to an entity
  public boolean isVariableNameBinded(String name) {
```

```
    return variableDeclaration.containsKey(name);
  }
  //check if value of variable x is null (does not exists)
  public boolean isVariableValueNull(String x) {
    return variableValues.containsKey(x);
  }
  public Type getVariableType(String name) {
    if(isVariableNameBinded(name))
      return variableDeclaration.get(name);
    else
      throw new java.lang.RuntimeException("entity with identifier "+name+" does not exist");
  }
  //get value associated with variable x
  public String getVariableValue(String x) {
    if(isVariableValueNull(x))
      return variableValues.get(x);
    else
      throw new java.lang.RuntimeException("entity with identifier "+x+" is associated with no
      value");
  }
  public Type getFunctionType(FunctionSignature functionSignature) {
    if(isFunctionAlreadyDefined(functionSignature))
      return functionDeclaration.get(functionSignature);
    else
      throw new java.lang.RuntimeException("function with identifier "+functionSignature.
    getFunctionName()
                         +" and type(s) "+functionSignature.getParametersTypes() +" does not
    exist");
  }
  public Map<FunctionSignature,Type> getFunctionDeclaration(){
    return functionDeclaration;
  }
}
```

Listing 6.13: Scope

```
package tinylangvisitor;
import java.util.Stack;

import tinylangparser.TinyLangAst;
import tinylangparser.Type;
public class SymbolTable {

  //current function parameter values
  private Stack<Scope> scopes = new Stack<Scope>();
  public void push() {
    Scope newScope = new Scope();
    scopes.add(newScope);
  }
  public void insertVariableDeclaration(String name,Type type) {
    getCurrentScope().addVariableDeclaration(name, type);
  }
  //add value to variable x
  public void insertVariableValue(String x,String value) {
    getCurrentScope().addVariableValue(x, value);
  }
  public void insertFunctionDeclaration(FunctionSignature functionSignature,Type type) {
    getCurrentScope().addFunctionDeclaration(functionSignature, type);
  }
  public void insertFunctionParameterNames(FunctionSignature functionSignature, Stack<String>
    functionParameterNames) {
```

```
    getCurrentScope().addFunctionParameterNames(functionSignature, functionParameterNames);
  }
  public void insertFunctionBlock(FunctionSignature functionSignature, TinyLangAst
     functionBlock) {
    getCurrentScope().addFunctionBlock(functionSignature, functionBlock);

  }
  public void deleteVariable(String name) {
    getCurrentScope().deleteVariable(name);
  }

  public boolean isVariableNameBinded(String name) {
    //check is identifier is already binded in current scope
    return getCurrentScope().isVariableNameBinded(name);
  }
  public Type getVariableType(String name) {
    return getCurrentScope().getVariableType(name);
  }
  public void pop(){
    scopes.pop();
  }
  public Stack<Scope> getScopes(){
    return scopes;
  }
  public Scope getCurrentScope() {
    return scopes.peek();
  }
}
```

Listing 6.14: Symbol Table

```
package tinylangvisitor;

import java.util.HashMap;
import java.util.Map;
import java.util.Stack;

import tinylangparser.TinyLangAst;
import tinylangparser.TinyLangAstNodes;
import tinylangparser.Type;

public class SemanticAnalyser implements Visitor {
  /**
   * Constructor for semantic analysis,
   * pass in AST of TinyLang program
   * to semantically analyse it.
   * @param programTree
   */
  public SemanticAnalyser(TinyLangAst programTree) {
    //create global scope
    st.push();
    //traverse program
    visitTinyLangProgram(programTree);
    //confirmation
    st.pop();
    System.out.println("Note: program is semantically correct");
  }
  //this is used to analyse types of expressions
  private Type currentExpressionType;
  //set symbol table
  private SymbolTable st = new SymbolTable();
```

```
//get a hold of current function types
Stack<Type> function = new Stack<Type>();
//get a hold of current function parameters
Map<String,Type> currentFunctionParameters = new HashMap<String,Type>();

//Map<String,Type> currentFunctionParameters = new HashMap<String,Type>();

//method which runs statement visit method based on node type
public void visitStatement(TinyLangAst tinyLangAst) {
  switch(tinyLangAst.getAssociatedNodeType()) {
    case AST_VARIABLE_DECLARATION_NODE:
      visitVariableDeclarationNode(tinyLangAst);
      break;
    case AST_ASSIGNMENT_NODE:
      visitAssignmentNode(tinyLangAst);
      break;
    case AST_PRINT_STATEMENT_NODE:
      visitPrintStatementNode(tinyLangAst);
      break;
    case AST_IF_STATEMENT_NODE:
      visitIfStatementNode(tinyLangAst);
      break;
    case AST_FOR_STATEMENT_NODE:
      visitForStatementNode(tinyLangAst);
      break;
    case AST_WHILE_STATEMENT_NODE:
      visitWhileStatementNode(tinyLangAst);
      break;
    case AST_RETURN_STATEMENT_NODE:
      visitReturnStatementNode(tinyLangAst);
      break;
    case AST_FUNCTION_DECLARATION_NODE:
      visitFunctionDeclarationNode(tinyLangAst);
      break;
    case AST_BLOCK_NODE:
      visitBlockNode(tinyLangAst);
      break;
    default:
      throw new java.lang.RuntimeException("Unrecognised statement of  type "+tinyLangAst.
  getAssociatedNodeType());
    }
  }
//visit expression based on node type
private void visitExpression(TinyLangAst tinyLangAst){
  switch(tinyLangAst.getAssociatedNodeType()) {
  case AST_BINARY_OPERATOR_NODE:
    visitBinaryOperatorNode(tinyLangAst);
    break;
  case AST_UNARY_OPERATOR_NODE:
    visitUnaryOperatorNode(tinyLangAst);
    break;
  case AST_BOOLEAN_LITERAL_NODE:
    visitBooleanLiteralNode(tinyLangAst);
    break;
  case AST_INTEGER_LITERAL_NODE:
    visitIntegerLiteralNode(tinyLangAst);
    break;
  case AST_FLOAT_LITERAL_NODE:
    visitFloatLiteralNode(tinyLangAst);
    break;
  case AST_CHAR_LITERAL_NODE:
```

```
      visitCharLiteralNode(tinyLangAst);
      break;
    case AST_IDENTIFIER_NODE:
      visitIdentifierNode(tinyLangAst);
      break;
    case AST_FUNCTION_CALL_NODE:
      visitFunctionCallNode(tinyLangAst);
      break;
    default:
      throw new java.lang.RuntimeException("Unrecognised expression node of type "+tinyLangAst
    .getAssociatedNodeType());
    }
  }
  @Override
  public void visitTinyLangProgram(TinyLangAst tinyLangAst) {
    //traverse all statements
    for(TinyLangAst statement : tinyLangAst.getChildren())
      //visit statement
      visitStatement(statement);
  }

  @Override
  public void visitVariableDeclarationNode(TinyLangAst tinyLangAst) {
    //get expression
    TinyLangAst identifier = tinyLangAst.getChildren().get(1);
    TinyLangAst expression = tinyLangAst.getChildren().get(2);
    //if identifier is already declared -> ERROR
    if(st.isVariableNameBinded(identifier.getAssociatedNodeValue())==true)
      throw new java.lang.RuntimeException("variable "+identifier.getAssociatedNodeValue()+"
    in line "
                                  +identifier.getLineNumber()+" was already declared previously");

    //visit expression -> update current expression type
    visitExpression(expression);

    /* type checking */
    //we allow type(variable)=float and type(expression)=int (since int can resolve to float)
    Type varType = Type.valueOf(tinyLangAst.getChildren().get(0).getAssociatedNodeValue());
    if(varType==Type.FLOAT && getCurrentExpressionType()==Type.INTEGER)
      //name binding
      st.insertVariableDeclaration(identifier.getAssociatedNodeValue(), varType);
    else if(varType==getCurrentExpressionType())
      //name binding
      st.insertVariableDeclaration(identifier.getAssociatedNodeValue(), varType);
    else
      throw new java.lang.RuntimeException("type mismatch, identifier in line"
                          +identifier.getLineNumber()
                          +"of type"+varType
                          +" and expression in line"
                          +expression.getLineNumber()
                          +" of type"+getCurrentExpressionType());
  }

  @Override
  public void visitAssignmentNode(TinyLangAst tinyLangAst) {
    //get identifier name
    String variableName = tinyLangAst.getChildren().get(0).getAssociatedNodeValue();
    //visit expression
    TinyLangAst expression = tinyLangAst.getChildren().get(1);
    //get a hold of all scopes
    Stack<Scope> scopes = st.getScopes();
```

```
    int i = 0;
    /*
     * start traversing from inner scope to outer scope to find in
     * which innermost scope variable is declared
     */
    for(i=scopes.size()-1;i>=0;i--) {
      if(scopes.get(i).isVariableNameBinded(variableName))
        break;
    }
    if(i<0)
      throw new java.lang.RuntimeException("identifier "+variableName+" was never declared");
    //obtain type from scope
    Type type = scopes.get(i).getVariableType(variableName);
    //visit expression & update the current expression type
    visitExpression(expression);

    //handle assignment type mismatch

    //allow integer to resolve to float
    if(type==Type.FLOAT && getCurrentExpressionType()==Type.INTEGER);
    else if(type!=getCurrentExpressionType())
      throw new java.lang.RuntimeException("type mismatch : variable "+variableName
                                +" in line "
                                + tinyLangAst.getChildren()
                                .get(0).getLineNumber()
                                + " of type "+type.toString()
                                + "assigned to expression of type"
                                + getCurrentExpressionType().toString());
  }

  @Override
  public void visitPrintStatementNode(TinyLangAst tinyLangAst) {
    //visit expression -> update current expression type
    visitExpression(tinyLangAst.getChildren().get(0));
  }

  @Override
  public void visitIfStatementNode(TinyLangAst tinyLangAst) {
    //get expression
    TinyLangAst expression = tinyLangAst.getChildren().get(0);
    //visit expression and update expression current type
    visitExpression(expression);
    //check that expression is boolean
    if(getCurrentExpressionType()!=Type.BOOL)
      throw new java.lang.RuntimeException("if condition in line "
                          +tinyLangAst.getLineNumber()
                          +" is not a predicate expression");
    //visit if block
    visitBlockNode(tinyLangAst.getChildren().get(1));
    //if exists an else block visit it
    if(tinyLangAst.getChildren().size()==3)
      visitBlockNode(tinyLangAst.getChildren().get(2));
  }

  @Override
  public void visitForStatementNode(TinyLangAst tinyLangAst) {
    //first child is variable declaration or expression
    if(tinyLangAst.getChildren().get(0).getAssociatedNodeType()==TinyLangAstNodes.AST_VARIABLE_DECLARATION_NODE)
      visitVariableDeclarationNode(tinyLangAst.getChildren().get(0));
    else
```

```
      visitExpression(tinyLangAst.getChildren().get(0));

    //second child is assignment or block or expression
    if(tinyLangAst.getChildren().get(1).getAssociatedNodeType()==TinyLangAstNodes.
    AST_ASSIGNMENT_NODE)
      visitAssignmentNode(tinyLangAst.getChildren().get(1));
    else if(tinyLangAst.getChildren().get(1).getAssociatedNodeType()==TinyLangAstNodes.
    AST_BLOCK_NODE)
      visitBlockNode(tinyLangAst.getChildren().get(1));
    else
      visitExpression(tinyLangAst.getChildren().get(1));

    //if we have 3 children
    //third child is assignment or block
    if(tinyLangAst.getChildren().size()==3&&tinyLangAst.getChildren().get(2).
    getAssociatedNodeType()==TinyLangAstNodes.AST_ASSIGNMENT_NODE)
      visitAssignmentNode(tinyLangAst.getChildren().get(2));
    else if(tinyLangAst.getChildren().size()==3&&tinyLangAst.getChildren().get(2).
    getAssociatedNodeType()==TinyLangAstNodes.AST_BLOCK_NODE)
      visitBlockNode(tinyLangAst.getChildren().get(2));
    //if we have 4 children
    //fourth child is block
    if(tinyLangAst.getChildren().size()==4)
      visitBlockNode(tinyLangAst.getChildren().get(3));
  }

  @Override
  public void visitWhileStatementNode(TinyLangAst tinyLangAst) {
    //got a hold on expression condition
    TinyLangAst expression  = tinyLangAst.getChildren().get(0);
    //get a hold on block node
    TinyLangAst  block = tinyLangAst.getChildren().get(1);
    //visit expression and update current value expression type
    visitExpression(expression);
    //expect that the expression is a predicate
    if(getCurrentExpressionType()!=Type.BOOL)
      throw new java.lang.RuntimeException("expected while condition to be a predicate in line
     "+tinyLangAst.getLineNumber());
    //visit block
    visitBlockNode(block);
  }

  @Override
  public void visitReturnStatementNode(TinyLangAst tinyLangAst) {
    //get expression
    TinyLangAst expression = tinyLangAst.getChildren().get(0);
    //visit expression and update current expression time
    visitExpression(expression);
    //we allow to return integer if function is of type float
    if(!function.empty() && getCurrentExpressionType()==Type.INTEGER && function.peek()==Type.
    FLOAT);
    //check that expression is has the same type as the function
    else if(!function.empty() && getCurrentExpressionType()!=function.peek())
      throw new java.lang.RuntimeException("return in line "
          +tinyLangAst.getLineNumber()+" returns expression of type "+
          getCurrentExpressionType()+" expected type "+function.peek());
  }

  @Override
  public void visitFunctionDeclarationNode(TinyLangAst tinyLangAst) {
    //get function type
```

```
    Type functionType = Type.valueOf(tinyLangAst.getChildren().get(0).getAssociatedNodeValue());

    //get function identifier
    String functionName = tinyLangAst.getChildren().get(1).getAssociatedNodeValue();

    //get function parameter types
    Stack<Type> functionParameterTypes = new Stack<Type>();
    //get stack of names to check for duplicate parameter names
    Stack<String> functionParameterNames = new Stack<String>();
    //add types

    //get current paramater name
    TinyLangAst parameterName;
    for(TinyLangAst formalParameterTypes : tinyLangAst.getChildren().get(2).getChildren()) {
      parameterName = formalParameterTypes.getChildren().get(1);
      functionParameterTypes.push(Type.valueOf(
          formalParameterTypes.getChildren()
          .get(0).getAssociatedNodeValue()));
      //if parameter name is duplicate throw exception
      if(functionParameterNames.contains(parameterName.getAssociatedNodeValue()))
        throw new java.lang.RuntimeException("function parameter name "+parameterName.getAssociatedNodeValue()+
            " already defined in line "+parameterName.getLineNumber());
      functionParameterNames.push(parameterName.getAssociatedNodeValue());
    }

    //check in all scopes that the function is not already defined
    for(Scope scope : st.getScopes())
      if(scope.isFunctionAlreadyDefined(new FunctionSignature(functionName, functionParameterTypes)))
        throw new java.lang.RuntimeException("function "+functionName+" in line "+tinyLangAst.getChildren().get(1).getLineNumber()+" with the same parameter types already defined previously");
    //add function to st
    st.insertFunctionDeclaration(new FunctionSignature(functionName,functionParameterTypes), functionType);
    //record current function in stack
    function.push(functionType);
    //empty current function parameters
    currentFunctionParameters.clear();
    for(TinyLangAst formalParameter : tinyLangAst.getChildren().get(2).getChildren())
      currentFunctionParameters.put(formalParameter.getChildren().get(1).getAssociatedNodeValue(),
                  Type.valueOf(formalParameter.getChildren()
                  .get(0).getAssociatedNodeValue()));
    //visit block
    visitBlockNode(tinyLangAst.getChildren().get(3));
    //pop type
    function.pop();
    //check if function returns
    if(!returns(tinyLangAst.getChildren().get(3)))
      throw new java.lang.RuntimeException("function "+functionName+" in line "+tinyLangAst.getLineNumber()+" not expected to return");
  }

  @Override
  public void visitFunctionCallNode(TinyLangAst tinyLangAst) {
    //determine the signature of the function
    Stack<Type> parameterTypes = new Stack<Type>();
    String functionIdentifier = tinyLangAst.getChildren().get(0).getAssociatedNodeValue();
```

```
    //identify the expressions and update  stack current expression types
    for(TinyLangAst expression :tinyLangAst.getChildren().get(1).getChildren()) {
      visitExpression(expression);
      parameterTypes.push(getCurrentExpressionType());
    }
    Stack<Scope> scopes = st.getScopes();
    int i;

    for(i=scopes.size()-1;i>=0;i--)
      if(scopes.get(i).isFunctionAlreadyDefined(new FunctionSignature(functionIdentifier,
    parameterTypes)))
        break;

    if(i<0)
      throw new java.lang.RuntimeException("function "+functionIdentifier+" in line "+
    tinyLangAst.getLineNumber()+" is not defined");

    //if defined set current expression type to return value of the function
    setCurrentExpressionType(scopes.get(i).getFunctionType(new FunctionSignature(
    functionIdentifier,parameterTypes)));
  }

  @Override
  public void visitBlockNode(TinyLangAst tinyLangAst) {
    //create new scope
    st.push();
    //add parameters of functions if any in scope
    for(String variableName:currentFunctionParameters.keySet())
      st.insertVariableDeclaration(variableName,currentFunctionParameters.get(variableName));
    //clear parameter map
    currentFunctionParameters.clear();
    //traverse statements in block
    for(TinyLangAst statement:tinyLangAst.getChildren())
      visitStatement(statement);
    //visit statements in block
    //end scope
    st.pop();
  }

//  @Override
//  public void visitElseBlockNode(TinyLangAst tinyLangAst) {
//    visitBlockNode(tinyLangAst);
//  }

  @Override
  public void visitBinaryOperatorNode(TinyLangAst tinyLangAst) {
    //get operator
    String operator = tinyLangAst.getAssociatedNodeValue();
    //get left node  (left operand)
    TinyLangAst leftOperand = tinyLangAst.getChildren().get(0);
    //visit expression to update current char type
    visitExpression(leftOperand);
    //obtain the type of the left operand
    Type leftOperandType = getCurrentExpressionType();

    //REDO for right node (right operand)
    //get left node  (left operand)
    TinyLangAst rightOperand = tinyLangAst.getChildren().get(1);
    //visit expression to update current char type
    visitExpression(rightOperand);
    //obtain the type of the left operand
```

```java
    Type rightOperandType = getCurrentExpressionType();

    /*
     * Operators
     *
     * Operators 'and | 'or' must have operands of type bool
     *
     * Operator '+' | '-' | '/' | '*' | '<' | '>' | '<=' | '>=' work on numeric operators
     *
     * Operators '==' | '!=' operates on any 2 operands of the same type both numeric or both boolean or both char
     */
    if(operator.equals("and")||operator.equals("or")) {
      if(leftOperandType==Type.BOOL && rightOperandType==Type.BOOL)
        setCurrentExpressionType(Type.BOOL);

      else
        throw new java.lang.RuntimeException("expected 2 operands of boolean type for operator "
                              +operator+" in line "+tinyLangAst.getLineNumber());
    }
    else if(operator.equals("+")||operator.equals("-")||operator.equals("/")||operator.equals("*")){
      if(!isNumericType(leftOperandType)||!isNumericType(rightOperandType))
        throw new java.lang.RuntimeException("expected 2 operands of numeric type for operator "
            +operator+" in line "+tinyLangAst.getLineNumber());

      //if both are numeric if one of them is float the operator returns float otherwise returns integer
      if(leftOperandType==Type.FLOAT||rightOperandType==Type.FLOAT)
        setCurrentExpressionType(Type.FLOAT);
      else
        setCurrentExpressionType(Type.INTEGER);

    }
    else if(operator.equals("<")||operator.equals(">")||operator.equals("<=")||operator.equals(">=")) {
      if(!isNumericType(leftOperandType)||!isNumericType(rightOperandType))
        throw new java.lang.RuntimeException("expected 2 operands of numeric type for operator "
            +operator+" in line "+tinyLangAst.getLineNumber());
      //if both are numeric set relation operators returns a boolean value
      setCurrentExpressionType(Type.BOOL);
    }

    else if(operator.equals("==")||operator.equals("!=")) {
      //handle mismatch not that float and integers are
      //both considered as one numerical type
      if((leftOperandType!=rightOperandType) &&
          (!isNumericType(leftOperandType)||!isNumericType(rightOperandType)))
        throw new java.lang.RuntimeException("operand mismatch in line "+tinyLangAst.getLineNumber());
      //if operands match
      setCurrentExpressionType(Type.BOOL);
    }
    else
      throw new java.lang.RuntimeException("binary operator "+operator+" unrecognised");
  }

  @Override
```

```java
public void visitUnaryOperatorNode(TinyLangAst tinyLangAst) {
  //unary operator
  String operator = tinyLangAst.getAssociatedNodeValue();
  //visit expression
  visitExpression(tinyLangAst.getChildren().get(0));
  //if current expression is numerical
  if(getCurrentExpressionType()==Type.INTEGER || getCurrentExpressionType()==Type.FLOAT)
    //check if operator is '-' | '+'
    if(!operator.equals("-")&&!operator.equals("+"))
      throw new java.lang.RuntimeException("operator "+operator+" not allowed in front of numerical expression in line "+tinyLangAst.getChildren().get(0).getLineNumber());

  else if(getCurrentExpressionType()==Type.BOOL )
    //check if operator is not
    if(!operator.equals("not"))
        throw new java.lang.RuntimeException("operator "+operator+" not allowed in front of predicate expreesion in line "+tinyLangAst.getChildren().get(0).getLineNumber());
  else
    throw new java.lang.RuntimeException("unary operator "+operator+" is incompatible with expression in line "+tinyLangAst.getLineNumber());
}

@Override
public void visitIdentifierNode(TinyLangAst tinyLangAst) {
  //find scope where identifier is defined
  Stack<Scope> scopes = st.getScopes();
  int i;
  for(i=scopes.size()-1;i>=0;i--)
    if(scopes.get(i).isVariableNameBinded(tinyLangAst.getAssociatedNodeValue()))
      break;
  if(i<0)
    throw new java.lang.RuntimeException("variable name "+tinyLangAst.getAssociatedNodeValue()+" in line "+tinyLangAst.getLineNumber()+" is not defined");
  setCurrentExpressionType(scopes.get(i).getVariableType(tinyLangAst.getAssociatedNodeValue()));
}

@Override
public void visitBooleanLiteralNode(TinyLangAst tinyLangAst) {
  setCurrentExpressionType(Type.BOOL);
}

@Override
public void visitIntegerLiteralNode(TinyLangAst tinyLangAst) {
  setCurrentExpressionType(Type.INTEGER);
}

@Override
public void visitFloatLiteralNode(TinyLangAst tinyLangAst) {
  setCurrentExpressionType(Type.FLOAT);
}

@Override
public void visitCharLiteralNode(TinyLangAst tinyLangAst) {
  setCurrentExpressionType(Type.CHAR);
}
private boolean isNumericType(Type type) {
  if(type ==Type.INTEGER||type ==Type.FLOAT)
    return true;
  else
    return false;
```

```java

  }

  private boolean returns(TinyLangAst tinyLangAst) {
    //if given statement is a return statement
    //then obviously we have that the function returns
    if(tinyLangAst.getAssociatedNodeType()==TinyLangAstNodes.AST_RETURN_STATEMENT_NODE)
      return true;
    //given a block we check if one of the statement returns
    if(tinyLangAst.getAssociatedNodeType()==TinyLangAstNodes.AST_BLOCK_NODE) {
      for(TinyLangAst statement:tinyLangAst.getChildren())
        if(returns(statement))
          return true;
    }
    //given a block we check if one of the statement returns
    if(tinyLangAst.getAssociatedNodeType()==TinyLangAstNodes.AST_ELSE_BLOCK_NODE) {
      for(TinyLangAst statement:tinyLangAst.getChildren())
        if(returns(statement))
          return true;
    }
    //if statement with an else block returns if both statement returns
    if(tinyLangAst.getAssociatedNodeType()==TinyLangAstNodes.AST_IF_STATEMENT_NODE)
      //if statement has else block
      if(tinyLangAst.getChildren().size()==3) {
        //block and else block both return
        return returns(tinyLangAst.getChildren().get(1)) && returns(tinyLangAst.getChildren().
    get(2));
      }
    //if statement with an for block returns if both statement returns
    if(tinyLangAst.getAssociatedNodeType()==TinyLangAstNodes.AST_FOR_STATEMENT_NODE)
        return returns(tinyLangAst.getChildren().get(tinyLangAst.getChildren().size()-1));

    //if statement with an else block returns if both statement returns
    if(tinyLangAst.getAssociatedNodeType()==TinyLangAstNodes.AST_WHILE_STATEMENT_NODE)
      return returns(tinyLangAst.getChildren().get(1));
    else
      //in all other cases the function do not return
      return false;
  }
  public void setCurrentExpressionType(Type currentExpressionType) {
    this.currentExpressionType=currentExpressionType;
  }
  public Type getCurrentExpressionType() {
    return currentExpressionType;
}
}
```

Listing 6.15: Semantic Analyser

6.5 | Interpreter

```java
package tinylangvisitor;
import java.util.Stack;
import tinylangparser.TinyLangAst;
import tinylangparser.TinyLangAstNodes;
import tinylangparser.Type;
```

```
/**
 * Class interpreter
 * @author andre
 *
 */
public class Interpreter implements Visitor{
  //create a symbol table
  private SymbolTable st = new SymbolTable();
  //save current expression type for evaluation
  private Type currentExpressionType;
  //save current expression value for evaluation
  private String currentExpressionValue;

  //save temporary information on function call parameters
  private Stack<Type> parameterTypes = new Stack<Type>();
  private Stack<String> parameterNames= new Stack<String>();
  private Stack<String> parameterValues= new Stack<String>();


  public Interpreter(TinyLangAst intermediateRepresentation) {
    //analyse the representation semantically
    new SemanticAnalyser(intermediateRepresentation);
    //push global scope
    st.push();
    //interpret tinyLangProgram
    visitTinyLangProgram(intermediateRepresentation);
  }
  //method which runs statement visit method based on node type
  private void visitStatement(TinyLangAst tinyLangAst) {
    switch(tinyLangAst.getAssociatedNodeType()) {
      case AST_VARIABLE_DECLARATION_NODE:
        visitVariableDeclarationNode(tinyLangAst);
        break;
      case AST_ASSIGNMENT_NODE:
        visitAssignmentNode(tinyLangAst);
        break;
      case AST_PRINT_STATEMENT_NODE:
        visitPrintStatementNode(tinyLangAst);
        break;
      case AST_IF_STATEMENT_NODE:
        visitIfStatementNode(tinyLangAst);
        break;
      case AST_FOR_STATEMENT_NODE:
        visitForStatementNode(tinyLangAst);
        break;
      case AST_WHILE_STATEMENT_NODE:
        visitWhileStatementNode(tinyLangAst);
        break;
      case AST_RETURN_STATEMENT_NODE:
        visitReturnStatementNode(tinyLangAst);
        break;
      case AST_FUNCTION_DECLARATION_NODE:
        visitFunctionDeclarationNode(tinyLangAst);
        break;
      case AST_BLOCK_NODE:
        visitBlockNode(tinyLangAst);
        break;
      default:
        throw new java.lang.RuntimeException("Unrecognised statement of  type "+tinyLangAst.
    getAssociatedNodeType());
      }
```

```
    }
  //visit expression
  //visit expression based on node type
    private void visitExpression(TinyLangAst tinyLangAst){
      switch(tinyLangAst.getAssociatedNodeType()) {
      case AST_BINARY_OPERATOR_NODE:
        visitBinaryOperatorNode(tinyLangAst);
        break;
      case AST_UNARY_OPERATOR_NODE:
        visitUnaryOperatorNode(tinyLangAst);
        break;
      case AST_BOOLEAN_LITERAL_NODE:
        visitBooleanLiteralNode(tinyLangAst);
        break;
      case AST_INTEGER_LITERAL_NODE:
        visitIntegerLiteralNode(tinyLangAst);
        break;
      case AST_FLOAT_LITERAL_NODE:
        visitFloatLiteralNode(tinyLangAst);
        break;
      case AST_CHAR_LITERAL_NODE:
        visitCharLiteralNode(tinyLangAst);
        break;
      case AST_IDENTIFIER_NODE:
        visitIdentifierNode(tinyLangAst);
        break;
      case AST_FUNCTION_CALL_NODE:
        visitFunctionCallNode(tinyLangAst);
        break;
      default:
        throw new java.lang.RuntimeException("Unrecognised expression node of type "+
    tinyLangAst.getAssociatedNodeType());
      }
    }
  @Override
  public void visitTinyLangProgram(TinyLangAst tinyLangAst) {
    //program ≡ sequence of statements : traverse all statement nodes
    for(TinyLangAst statement : tinyLangAst.getChildren())
      visitStatement(statement);
  }
  @Override
  public void visitVariableDeclarationNode(TinyLangAst tinyLangAst) {
    //get variable type
    Type varType = Type.valueOf(tinyLangAst.getChildren().get(0).getAssociatedNodeValue());
    //get hold on identifier
    String varName = tinyLangAst.getChildren().get(1).getAssociatedNodeValue();
    //visit expression and update current expression value
    TinyLangAst expression = tinyLangAst.getChildren().get(2);
    visitExpression(expression);
    //add variable declaration in current scope
    st.insertVariableDeclaration(varName, varType);
    //add value assigned to variable
    st.insertVariableValue(varName,currentExpressionValue);
  }
  @Override
  public void visitAssignmentNode(TinyLangAst tinyLangAst) {
    //get identifier name
    String varName = tinyLangAst.getChildren().get(0).getAssociatedNodeValue();
    //update current expression value
    TinyLangAst expression = tinyLangAst.getChildren().get(1);
    visitExpression(expression);
```

```
    int i;
    /*
     * start traversing from inner scope to outer scope to find in
     * which innermost scope variable is declared
     */
    for(i=st.getScopes().size()-1;i>=0;i--) {
      if(st.getScopes().get(i).isVariableNameBinded(varName))
        break;
    }
    /*
     * go in that innermost scope and update the value
     */
    st.getScopes().get(i).addVariableValue(varName, currentExpressionValue);
  }

  @Override
  public void visitPrintStatementNode(TinyLangAst tinyLangAst) {
    visitExpression(tinyLangAst.getChildren().get(0));
    System.out.println(currentExpressionValue);
  }
  @Override
  public void visitIfStatementNode(TinyLangAst tinyLangAst) {
    TinyLangAst expression= tinyLangAst.getChildren().get(0);
    //evaluate if condition
    visitExpression(expression);
    //check condition
    if(currentExpressionValue.equals("true"))
      visitBlockNode(tinyLangAst.getChildren().get(1));
    //if we have an else block
    else if (currentExpressionValue.equals("false") && tinyLangAst.getChildren().size()==3)
      visitBlockNode(tinyLangAst.getChildren().get(2));

  }

  @Override
  public void visitForStatementNode(TinyLangAst tinyLangAst) {
    //we have a list of possibilities for a for loop statement

    //no variable declaration and no assignment

    /*
     *         for loop
     *            / \
     *           /   \
         *     expression      block
     */

    //this can be encoded as a while loop statement
    if(tinyLangAst.getChildren().size()==2) {
      TinyLangAst expression  = tinyLangAst.getChildren().get(0);
      TinyLangAst block = tinyLangAst.getChildren().get(1);
      visitExpression(expression);
      while(currentExpressionValue.equals("true")) {
        //visit block
        visitBlockNode(block);
        //update current expression value
        visitExpression(expression);
      }
    }
    //if we have both variable declaration and assignment

```

```
    /*
     *             for loop---\
     *               / /     \      \
     *                / /       \      \
     *           /   |         \    block
         *                 / expression  \
     *           variable               \
     *           declaration          updation/assignment
     *
    */
    else if(tinyLangAst.getChildren().size()==4) {
      TinyLangAst variableDeclaration = tinyLangAst.getChildren().get(0);
      //visit variable declaration
      visitVariableDeclarationNode(variableDeclaration);
      //visit expression and update current expression value
      visitExpression(tinyLangAst.getChildren().get(1));
      while(currentExpressionValue.equals("true")) {
        //visit block
        visitBlockNode(tinyLangAst.getChildren().get(3));
        //carry out updation/assignment
        visitAssignmentNode(tinyLangAst.getChildren().get(2));
        //update current expression value
        visitExpression(tinyLangAst.getChildren().get(1));
      }
      st.deleteVariable(variableDeclaration.getChildren().get(1).getAssociatedNodeValue());
    }
  //if we have variable declaration and no assignment

    /*
     *             for loop
     *                /   /       \
     *           /   /         \
         *                 / expression  \
     *           variable               \
     *           declaration          block
     *
    */
    else if(tinyLangAst.getChildren().get(0).getAssociatedNodeType()==TinyLangAstNodes.AST_VARIABLE_DECLARATION_NODE) {
      TinyLangAst variableDeclaration = tinyLangAst.getChildren().get(0);
      //declare variable
      visitVariableDeclarationNode(variableDeclaration);
      //update current expression value
      visitExpression(tinyLangAst.getChildren().get(0));
      while(currentExpressionValue.equals("true")) {
        //execute statements
        visitBlockNode(tinyLangAst.getChildren().get(2));
        //update current expression value
        visitExpression(tinyLangAst.getChildren().get(0));

      }
      st.deleteVariable(variableDeclaration.getChildren().get(1).getAssociatedNodeValue());
    }
  //if we have assignment and no variable declaration

    /*
     *             for loop
     *                /     /     \
     *           /     /         \
         *                 /     /          \
         *               /     assignment  \
```

```
   *      expression                  \
   *                                  block
   *
   */
  else if(tinyLangAst.getChildren().get(1).getAssociatedNodeType()==TinyLangAstNodes.
  AST_ASSIGNMENT_NODE)
  {
    //visit expression and update current expression value
    visitExpression(tinyLangAst.getChildren().get(0));
    while(currentExpressionValue.equals("true")) {
      //visit block
      //carry out update/assignment
      visitAssignmentNode(tinyLangAst.getChildren().get(1));
      //update current expression value
      visitExpression(tinyLangAst.getChildren().get(0));
    }
  }
  else
    throw new java.lang.RuntimeException("unexpected for loop case in line "+tinyLangAst.
  getLineNumber());
}

@Override
public void visitWhileStatementNode(TinyLangAst tinyLangAst) {
  //get a hold on block of while loop
  TinyLangAst block = tinyLangAst.getChildren().get(1);
  //update current expression value
  TinyLangAst expression  = tinyLangAst.getChildren().get(0);
  visitExpression(expression);
  //while current expression value is true
  //keep on looping
  while(currentExpressionValue.equals("true")) {
    //visit block
    visitBlockNode(block);
    //update current expression value
    visitExpression(expression);
  }
}

@Override
public void visitReturnStatementNode(TinyLangAst tinyLangAst) {
  //update current expression value
  visitExpression(tinyLangAst.getChildren().get(0));
}

@Override
public void visitFunctionDeclarationNode(TinyLangAst tinyLangAst) {
  //add function definition and values to symbol table
  //get function block ast
  TinyLangAst functionBlock = tinyLangAst.getChildren().get(3);
  //get variable type
  Type functionType = Type.valueOf(tinyLangAst.getChildren().get(0).getAssociatedNodeValue()
  );
  //get hold on identifier
  String functionName = tinyLangAst.getChildren().get(1).getAssociatedNodeValue();
  //get function parameter types
  Stack<Type> functionParameterTypes = new Stack<Type>();
  Stack<String> functionParameterNames = new Stack<String>();
  //add parameters types and values
  for(TinyLangAst formalParameter : tinyLangAst.getChildren().get(2).getChildren()) {
    functionParameterTypes.push(Type.valueOf(formalParameter.getChildren().get(0).
```

```
    getAssociatedNodeValue()));
      functionParameterNames.push(formalParameter.getChildren().get(1).getAssociatedNodeValue());
    }
    //add function parameter types and names to st
    st.insertFunctionDeclaration(new FunctionSignature(functionName,functionParameterTypes),functionType);
    st.insertFunctionParameterNames(new FunctionSignature(functionName,functionParameterTypes),functionParameterNames);
    st.insertFunctionBlock(new FunctionSignature(functionName,functionParameterTypes),functionBlock);
  }
  @Override
  public void visitFunctionCallNode(TinyLangAst tinyLangAst) {

    //function name
    String functionName = tinyLangAst.getChildren().get(0).getAssociatedNodeValue();
    for(TinyLangAst expression :tinyLangAst.getChildren().get(1).getChildren()) {
      visitExpression(expression);
      parameterTypes.push(currentExpressionType);
      parameterValues.push(currentExpressionValue);
    }
    //function signature types of parameters
    int i;
    for(i=st.getScopes().size()-1;i>=0;i--)
      if(st.getScopes().get(i).isFunctionAlreadyDefined(new FunctionSignature(functionName,parameterTypes)))
        break;
    //add temporary function parameters names
    parameterNames.addAll(st.getScopes().get(i).getParameterNames(new FunctionSignature(functionName,parameterTypes)));
    //visit corresponding function block
    visitBlockNode(st.getScopes().get(i).getBlock(new FunctionSignature(functionName,parameterTypes)));

  }

  @Override
  public void visitBlockNode(TinyLangAst tinyLangAst) {
    //enter a new scope
    st.push();
    //check all temporary function parameter stacks are of the same size
    if(!(parameterTypes.size()==parameterNames.size()&&parameterNames.size()==parameterValues.size()))
      throw new java.lang.RuntimeException("error with function call handling");
    //add parameters of functions if any in scope
    for(int i=0;i<parameterTypes.size();i++) {
      //add variable declaration in current scope
      st.insertVariableDeclaration(parameterNames.get(i), parameterTypes.get(i));
      //add value assigned to variable
      st.insertVariableValue(parameterNames.get(i),parameterValues.get(i));
      }
    //clear temporary function parameters data
    parameterTypes.clear();
    parameterNames.clear();
    parameterValues.clear();
    //traverse statements in block
    for(TinyLangAst statement:tinyLangAst.getChildren())
      visitStatement(statement);
    //leave scope
    st.pop();
```

```java
}


@Override
public void visitBinaryOperatorNode(TinyLangAst tinyLangAst) {
  //get operator

  String operator = tinyLangAst.getAssociatedNodeValue();

  //get left node  (left operand)
  TinyLangAst leftOperand = tinyLangAst.getChildren().get(0);
  //visit expression to update current char type
  visitExpression(leftOperand);
  //obtain the type of the left operand
  Type leftOperandType = currentExpressionType;
  //obtain the value of the left operand
  String leftOperandValue = currentExpressionValue;

  //redo for right operand
  TinyLangAst rightOperand = tinyLangAst.getChildren().get(1);
  visitExpression(rightOperand);
  Type rightOperandType = currentExpressionType;
  String rightOperandValue = currentExpressionValue;
  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)) {
      //int+int -> int
      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 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"))
        currentExpressionValue = "true";
      else
        currentExpressionValue = "false";
    }
    //comparison types
    else if(operator.equals("==")) {
      currentExpressionType = Type.BOOL;
      if(leftOperandValue.equals(rightOperandValue))
        currentExpressionValue = "true";
```

```
        else
          currentExpressionValue = "false";
      }
      else if(operator.equals("!=")) {
        currentExpressionType = Type.BOOL;
        if(!leftOperandValue.equals(rightOperandValue))
          currentExpressionValue = "true";
        else
          currentExpressionValue = "false";
      }
      else if(operator.equals("<")) {
        currentExpressionType = Type.BOOL;
        if(Float.parseFloat(leftOperandValue)<Float.parseFloat(rightOperandValue))
          currentExpressionValue = "true";
        else
          currentExpressionValue = "false";
      }
      else if(operator.equals("<=")) {
        currentExpressionType = Type.BOOL;
        if(Float.parseFloat(leftOperandValue)<=Float.parseFloat(rightOperandValue))
          currentExpressionValue = "true";
        else
          currentExpressionValue = "false";
      }
      else if(operator.equals(">")) {
        currentExpressionType = Type.BOOL;
        if(Float.parseFloat(leftOperandValue)>Float.parseFloat(rightOperandValue))
          currentExpressionValue = "true";
        else
          currentExpressionValue = "false";
      }
      else if(operator.equals(">=")) {
        currentExpressionType = Type.BOOL;
        if(Float.parseFloat(leftOperandValue)>=Float.parseFloat(rightOperandValue))
          currentExpressionValue = "true";
        else
          currentExpressionValue = "false";
      }
      else {
        throw new java.lang.RuntimeException("unexcepted binary operator error in line "+
      tinyLangAst.getLineNumber());
      }
      }
    @Override
    public void visitUnaryOperatorNode(TinyLangAst tinyLangAst) {
      TinyLangAst expression = tinyLangAst.getChildren().get(0);
      visitExpression(expression);
      String operator = tinyLangAst.getAssociatedNodeValue();
      if(currentExpressionType==Type.FLOAT) {
        if(operator.equals("-"))
          currentExpressionValue = String.valueOf(-1*Float.parseFloat(currentExpressionValue));
      }
      else if(currentExpressionType==Type.INTEGER) {
        if(operator.equals("-")) {
          currentExpressionValue = String.valueOf(-1*Integer.parseInt(currentExpressionValue));
        }
      }
      else if(currentExpressionType==Type.BOOL) {
        if(operator.equals("not")) {
          if(currentExpressionValue.equals("true"))
            currentExpressionValue = "false";
```

```
        else
          currentExpressionValue = "true";
      }
    }
    else
      throw new java.lang.RuntimeException
      ("unexpected error when handling unary opertor in line "+tinyLangAst.getLineNumber());
  }
  @Override
  public void visitIdentifierNode(TinyLangAst tinyLangAst) {
    //Identifier name
    String identifier = tinyLangAst.getAssociatedNodeValue();
    //traverse the scopes to find the identifier type and value
    int i;
    for(i=st.getScopes().size()-1;i>=0;i--) {
      if(st.getScopes().get(i).isVariableNameBinded(identifier))
        break;
    }
    currentExpressionType = st.getScopes().get(i).getVariableType(identifier);
    currentExpressionValue = st.getScopes().get(i).getVariableValue(identifier);
  }

  @Override
  public void visitBooleanLiteralNode(TinyLangAst tinyLangAst) {
    String boolIdentifier = tinyLangAst.getAssociatedNodeValue();
    currentExpressionType = Type.BOOL;
    currentExpressionValue = boolIdentifier;
  }

  @Override
  public void visitIntegerLiteralNode(TinyLangAst tinyLangAst) {
    String integerIdentifier = tinyLangAst.getAssociatedNodeValue();
    currentExpressionType = Type.INTEGER;
    currentExpressionValue = integerIdentifier;
  }

  @Override
  public void visitFloatLiteralNode(TinyLangAst tinyLangAst) {
    String floatIdentifier = tinyLangAst.getAssociatedNodeValue();
    currentExpressionType = Type.FLOAT;
    currentExpressionValue = floatIdentifier;
  }
  @Override
  public void visitCharLiteralNode(TinyLangAst tinyLangAst) {
    String charIdentifier = tinyLangAst.getAssociatedNodeValue();
    currentExpressionType = Type.CHAR;
    currentExpressionValue = charIdentifier;
  }
}
```

Listing 6.16: Interpreter

6.6 | GitHub Repo

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

Link