Table of contents

1. Introduction

For this assignment, we implement an LL(1) parser, a particular type of top-down parser. LL(1) parser processes input from left to right, constructing the leftmost derivation of the syntax tree using one lookahead token to make the decisions. We choose the recursive descent due to its simplicity and predictability, making it ideal for the project's grammar.

A recursive descent parser is straightforward to implement because each rule in the grammar gets its recursive method in the code. This makes the code more precise and easier to maintain since it follows the grammar structure closely. LL(1) parsers also work efficiently with predictable grammars, as they don't require backtracking and can make decisions based on the current token and our created rules.

Moreover, we chose LL(1) because the grammar for this subject of Java is simple and can be easily modified to fit LL(1) constraints. We used techniques like removing left recursion and factoring to make sure the grammar could work with the parser. This allowed the parser to handle things like expressions and statements without confusion.

Overall, the recursive descent LL(1) parser offered a balance of simplicity, control and performance, making it ideal for parsing this subset of Java.

2. Specification analysis

According to the project specification, to successfully implement the design of an LL(1) parser. The Java program must satisfy several requirements to handle a token stream derived from the input string and keep track of the internal non-terminal state. Based on constructing a parsing table, the parser will determine the corresponding grammar rule at each step.

A lexical analyser will initially be implemented to convert the input string into a token stream. This analyser will separate individual symbols and transform them into a Token enumerated type, which includes the corresponding type and value to represent each terminal fully.

After that, a parsing table, which is the central part of the program, will be integrated to navigate the decision-making process. In addition, a parser will also be incorporated to act as a bridge, connecting the observed token and the internal non-terminal state with the parsing table to guide further decisions.

Finally, the parser rule function will be implemented to represent each grammar rule accurately. After deriving the action from the parsing table, this function will be called within the main parser.

2.1 Construction of Lexer and Parser Components

2.1.1 Construction of Lexer

The Lexical Analyser is the first component of the process; it is responsible for reading the raw source code and breaking it down into different types of tokens such as keywords, operators and many others.

The code provided for the lexer starts by scanning through each character in the source code and classifying it based on predefined patterns. It identifies:

- **Single-character delimiters** like {, }, (,), +, -, /, and spaces them out for easy recognition.
- **Multi-character tokens**, such as ==, &&, ||, <, > and !=, by checking the next character and forming a valid token.

Special handling is implemented for character and string literals to capture the entire literal within single or double quotes, ensuring these tokens are correctly passed to the parser. The main feature of the Lexer:

- **Token Classification:** The tokenFromString method is used to classify the recognised tokens based on their type, whether they are keywords (public, class, while), operators (+, -, *, ==, etc), literals (numbers, characters, strings), or identifiers.
- **Error Handling:** The lexer detects unrecognised tokens or symbols and throws an exception (like LexicalException) to ensure that the parser only receives valid tokens.
- **Streamlining Parser Input:** By breaking down source code into meaningful tokens, the lexer simplifies the task of the parser, which then only needs to focus on the structural correctness of the token sequence.

The lexer is crucial for generating a stream of valid tokens that feed into the parser, setting up the stage for syntax analysis. The code handles various token types and enforces rules to ensure that only syntactically valid tokens are passed to the following compilation stage.

2.1.2 Construction of Parser

The LL(1) parser is a top-down parser that processes input from left to right and produces a leftmost derivation of the input string. Our parser utilises a lookahead token to make parsing decisions based on a predefined set of grammar rules.

The **parse** method is central to the parser's operation and is responsible for managing the analysis of the token stream generated by the Lexer. This method

utilised a table to determine which parsing rule to apply based on the current token and the syntactic structure the TreeNode represents.

```
private void parse(TreeNode tn) throws SyntaxException {
    Rule rule = parsingTable.get(new Pair<>(tn.getLabel(), lookahead.getType()));
    if (rule == null) {
        throw new SyntaxException(tn.getLabel() + " Unexpected token: " + lookahead.getType());

    }
    switch(rule){
        case PROG_1:
            parseProg_1(tn);
            break;
        case LOS_1:
            parseLos_1(tn);
            break;
        case LOS_2:
            parseLos_2(tn);
```

Token and Rule Management:

- Our method begins by attempting to retrieve the Rule from the parsingTable using the current token's type (lookahead.getType()) and the label of the current TreeNode (tn.getLabel()).
- If no rule is found (rule == null), the method throws a SyntaxException, indicating that the encountered token is unexpected. This provides immediate feedback on the syntax error, essential for debugging.

Switch Statement for Rule Application:

- The method's core is a switch statement that evaluates the retrieved rule. Each case corresponds to a parsing rule that dictates how the current node in the parse tree should be processed.
- For example, if the rule is PROG_1, the method parseProg_1(tn) is called to handle parsing related to program structure; this approach ensures that the parsing rule has a dedicated method, promoting clarity in the parser's design.
- Each call to a parsing method passes the current Treenode(tn) as a parameter. This allows the parsing techniques to construct and manipulate the parse tree, ensuring each syntactic element is represented in the tree structure.

2.2. Theoretical computation

The formal illustration of the provided Context-free grammar (CFG) illustrating the simple Java syntax can be indicated below:

$$G = (V, \Sigma, R, S)$$

With:

- V is the variables (non-terminals) set.
- Σ is the terminal set.
- R is the set of grammar rules
- S is the start state.

2.2.1. Variables Set

V= {<<prog>>, <<ID>> , <<los>>, <<stat>>, <<while>>, <<for>> ,<<if>>, <<assign>>, <<decl>>, <<print>>, <<rel expr>>, <<bool expr>>, <<for start>>, <<for arith>> , <<arith expr>> , <<else if>>, <<else?if>>, <<poss if>>, <<expr>>, <<type>>, <<poss assign>>, <<print expr>> , <<char expr>>, <<char>> , <<bool op>>, <<bool eq>> , <<bool log>>,<<rel expr'>>, <<rel op>>, <<term>>, <<arith expr'>>, <<factor>> , <<term'>> , <<num>> , <<string lit>> }

Start variable: S = {<<prog>>}

Within the Java program, this set of variables is defined via the Label enumeration (as indicated below) within the TreeNode class, representing the node's label of a parse tree.

```
public enum Label implements Symbol {
    prog, los, stat, whilestat, forstat, forstart, forarith, ifstat, elseifstat, elseorelseif,
    possif, assign, decl,possassign, print, type, expr, boolexpr, boolop, booleq, boollog,
    relexpr, relexprprime, relop, arithexpr,arithexprprime, term, termprime, factor, printexpr,
    charexpr, epsilon, terminal;

    @Override
    public boolean isVariable() {
        return true;
    }

};
```

2.2.2 Terminals Set

Σ= {+, -, *, /, %, =, ==, !=, <, >, <=, >=, (,), {, }, &&, ||, ;, public, class, static, void, main, String[], args, int, boolean, char, System.out.println, while, for, if, else, ", '}

In the Java implementation, the set of terminals is defined within a Token enumeration, representing the elements of the input token stream, indicated below:

```
public enum TokenType implements Symbol {
    PLUS, MINUS, TIMES, DIVIDE, MOD, ASSIGN, EQUAL, NEQUAL, LT, LE, GT, GE, LPAREN, RPAREN, LBRACE, RBRACE, AND, OR,
    SEMICOLON, PUBLIC, CLASS, STATIC, VOID, MAIN, STRINGARR, ARGS, TYPE, PRINT, WHILE, FOR, IF, ELSE, DQUOTE,
    SQUOTE, ID, NUM, CHARLIT, TRUE, FALSE, STRINGLIT;

    @Override
    public boolean isVariable() {
        return false;
    }

};
```

2.2.3 Rules Set

R= {<<prog>> → public class <<ID>> { public static void main (String[] args) { <<los>> } } , <<los>> → <<stat>> <<los>> | ε, …, <<print expr>> → "<<string lit>>"}

While the grammar rules have yet to be implemented within the provided Java structure, we have introduced a Rule enumeration to represent our Java program's grammar rules. Each rule will have a specific name (as detailed in Figure 1 below):

```
enum Rule {
    PROG_1, LOS_1, LOS_2, STAT_1, STAT_2, STAT_3, STAT_4, STAT_5, STAT_6, STAT_7,
    WHILE_1, FOR_1, FOR_START_1, FOR_START_2, FOR_START_3, FOR_ARITH_1, FOR_ARITH_2,
    IF_1, ELSE_IF_1, ELSE_IF_2, ELSE_IFF_1, POSS_IF_1, POSS_IF_2, ASSIGN_1, DECL_1,
    POSS_ASSIGN_1, POSS_ASSIGN_2, PRINT_1, TYPE, EXPR_1, EXPR_2,
    CHAR_EXPR_1, BOOL_EXPR_1, BOOL_EXPR_2, BOOL_OP_1, BOOL_OP_2, BOOL_EQ_1, BOOL_EQ_2,
    BOOL_LOG_1, BOOL_LOG_2, REL_EXPR_1, REL_EXPR_2, REL_EXPR_3, REL_EXPRR_1, REL_EXPRR_2,
    REL_OP_1, REL_OP_2, REL_OP_3, REL_OP_4, ARITH_EXPR_1, ARITH_EXPRR_1, ARITH_EXPRR_2,
    ARITH_EXPRR_3, TERM_1, TERMM_1, TERMM_2, TERMM_3, TERMM_4, FACTOR_1, FACTOR_2,FACTOR_3,
    PRINT_EXPR_1, PRINT_EXPR_2
};
```

2.2.4. First Set

Regarding the LL(1) parser, for a non-terminal symbol, the FIRST set represents all possible terminal symbols that can appear at the beginning of any string extracted from that non-terminal state. Moreover, if that non-terminal variable can derive the empty string (ε), then ε should also be included within the first set of that state. By adhering to these indicated rules and definitions, the FIRST set corresponding to each Tree Node within the project can be shown as below:

- FIRST (<<prog>>) = { public }
- FIRST (<<ID>>) = { any valid variable name }
- FIRST (<<los>>) = {ε, if, while, for, <<ID>>, int, boolean, char, System.out.println, ; }
- FIRST (<<stat>>) = { if, while, for, <<ID>>, int, boolean, char, System.out.println, ; }
- FIRST (<<while>>) = { while }
- FIRST (<<for>>) = { for }
- FIRST(<<if>>) = { if }
- FIRST(<<assign>>) = {<<ID>>}
- FIRST(<<decl>>) = { int, boolean, char }
- FIRST (<<print>>) = {System.out.println }
- FIRST (<<rel expr>>) ={ true , false , (, <<ID>>, <<num>> }
- FIRST (<<bool expr>>) = { == , != , && , || , ε }
- FIRST(<<for start>>) = { int, boolean, char, <<ID>>, ε}
- FIRST(<<for arith>>) = { (, <<ID>>, <<num>>, ε }
- FIRST(<<arith expr>>) = { (, <<ID>>, <<num>> }
- FIRST(<<else if>>) = { ε , else }
- FIRST(<<else?if>>) = { else }
- FIRST(<<poss if>>) = { if , ε }
- FIRST(<<expr>>) = { true , false , (, <<ID>>, <<num>>, ' }

- FIRST(<<type>>) = { int , boolean , char }
- FIRST(<<poss assign>>) = { =, ε }
- FIRST(<<print expr>>) = { true , false , (, <<ID>>, <<num>>, " }
- FIRST(<<char expr>>) = { ' }
- FIRST(<<char>>) = { any valid character }
- FIRST(<<bool op>>) = { == , != , && , || }
- FIRST(<<bool eq>>) = { == , != }
- FIRST(<<bool log>>) = { && , || }
- FIRST(<<rel expr'>>) = { ε, < , <= , > , >= }
- FIRST(<<rel op>>) = { < , <= , > , >= }
- FIRST(<<term>>) = { (, <<ID>> , <<num>> }
- FIRST(<<arith expr'>>) = { + , - , ε }
- FIRST(<<factor>>) = { (, <<ID>>, <<num>> }
- FIRST(<<term'>>) = { *, / , % , ε }
- FIRST(<<num>>) = { any valid integer }
- FIRST(<<string lit>>) = { any valid string }

2.2.5. Follow Set

Similarly, for any non-terminal symbols, the FOLLOW set demonstrates all possible terminal symbols that can come right after the non-terminal variable in any part of the grammar. Indicating the expected tokens after that non-terminal variable in the input stream. Moreover, if the non-terminal is at the end of the rule or is followed by a non-terminal symbol that possesses ε within its FIRST set, then the FOLLOW set of this non-terminal will also include the FOLLOW set of its parent node. Finally, the end-of-input marker ($) is added to the FOLLOW set of the start variable. By following these rules, the FOLLOW set of the provided CFG can be illustrated below:

- FOLLOW (<<prog>>) = {$}
- FOLLOW(<<los>>)={ } }
- FOLLOW(<<stat>>) = { if, while, for, <<ID>>, int, boolean, char, System.out.println, ;, } }
- FOLLOW(<<while>>) = { if, while, for, <<ID>>, int, boolean, char, System.out.println, ;, } }
- FOLLOW(<<for>>) = { if, while, for, <<ID>>, int, boolean, char, System.out.println, ;, } }
- FOLLOW(<<for start>>) = { ; }
- FOLLOW(<<for arith>>) = {) }
- FOLLOW(<<if>>) = { if, while, for, <<ID>>, int, boolean, char, System.out.println, ;, } }
- FOLLOW(<<else if>>) = { if, while, for, <<ID>>, int, boolean, char, System.out.println, ;, } }
- FOLLOW(<<else? if>>) = { { }
- FOLLOW(<<poss if>>) = { { }
- FOLLOW(<<assign>>) = { ; }
- FOLLOW(<<decl>>) = { ; }

- FOLLOW(<<poss assign>>) = { ; }
- FOLLOW(<<print>>) = { ; }
- FOLLOW(<<type>>) = { ID }
- FOLLOW(<<expr>>) = { ; }
- FOLLOW(<<char expr>>) = { ; }
- FOLLOW(<<bool expr>>) = {), ; }
- FOLLOW(<<bool op>>) = { true , false , (, <<ID>>, <<num>>}
- FOLLOW(<<bool eq>>) = { true , false , (, <<ID>>, <<num>> }
- FOLLOW(<<bool log>>) = { true , false , (, <<ID>>, <<num>> }
- FOLLOW(<<rel expr>>) = { ==, !=, &&, ||,), ; }
- FOLLOW(<<rel expr'>>) = { ==, !=, &&, ||,), ; }
- FOLLOW(<<rel op>>) = { (, <<ID>>, <<num>> }
- FOLLOW(<<arith expr>>) = {), <, <=, >, >=, ==, !=, &&, ||, ; }
- FOLLOW(<<arith expr'>>) = {), <, <=, >, >=, ==, !=, &&, ||, ; }
- FOLLOW(<<term>>) = { +, -,), <, <=, >, >=, ==, !=, &&, ||, ; }
- FOLLOW(<<term'>>) = { +, -,), <, <=, >, >=, ==, !=, &&, ||, ; }
- FOLLOW(<<factor>>) = { *, /, %, +, -,), <, <=, >, >=, ==, !=, &&, ||, ; }
- FOLLOW(<<print exp>>) = {) }
- FOLLOW(<<ID>> = { {, =, ; , *, /, %, +, -,), <, <=, >, >=, ==, !=, &&, || }
- FOLLOW(<<num>>) = { *, /, %, +, -,), <, <=, >, >=, ==, !=, &&, ||, ; }
- FOLLOW(<<char>>) = { ‘ }
- FOLLOW(<<string lit>>) = { " }

2.2.6 Parsing Table

In the LL(1) parser, a parsing table is constructed to navigate the parsing process by determining which grammar rule to apply based on the current leftmost input token and the current non-terminal symbol.

To implement the parsing table for an LL(1) parser, we use the FIRST and FOLLOW sets to determine which production rules to place in the table. Here are three rules that we learned from our lecture that are applied to build this table:

Rule 1: For each $a \in FIRST(w)$, add $X \rightarrow w$ to $M(X, a)$.

Rule 2: If $\varepsilon \in FIRST(w)$, for each $b \in FOLLOW(X)$, add $X \rightarrow w$ to $M(X, b)$.

Rule 3: If $\$ \in FOLLOW(X)$, add $X \rightarrow w$ to $M(X, \$)$.

How to Implement These Rules:

- We start by computing the FIRST and FOLLOW sets for all non-terminals in the grammar.
- For each production rule X -> w, we use FIRST(w) to identify the terminal symbols that can appear at the start of the right-hand side and fill out the parsing table according to Rule 1.
- If ε i in FIRST(w), we look at the FOLLOW set of X and apply Rule 2, adding the production to the table for any terminal that follows X.
- Lastly, if the en-of-input marker $ is in FOLLOW(X), we apply Rule 3 to handle the end of the input.

After creating it, we use a HashMap to implement the parsing table for our LL(1) parser. The HashMap allows us to efficiently map pairs of non-terminal

symbols and input tokens to the corresponding production rules. In our code, we define the table as:

```
private Map<Pair<TreeNode.Label, Token.TokenType>, Rule> parsingTable = new HashMap<>();
```

After being constructed, the table is utilised by the primary parse function (as indicated in section 2.1.2) using the observed leftmost token and the current non-terminal symbol to derive the grammar rule needed to navigate the tree's span. The parsing table is indicated below:

```
private void initializeParsingTable() {
    // Grammar rules for prog
    parsingTable.put(new Pair<>(TreeNode.Label.prog, Token.TokenType.PUBLIC), Rule.PROG_1);

    //Grammar rules for list of statements
    parsingTable.put(new Pair<>(TreeNode.Label.los, Token.TokenType.IF), Rule.LOS_1);
    parsingTable.put(new Pair<>(TreeNode.Label.los, Token.TokenType.WHILE), Rule.LOS_1);
    parsingTable.put(new Pair<>(TreeNode.Label.los, Token.TokenType.FOR), Rule.LOS_1);
    parsingTable.put(new Pair<>(TreeNode.Label.los, Token.TokenType.ID), Rule.LOS_1);
    parsingTable.put(new Pair<>(TreeNode.Label.los, Token.TokenType.TYPE), Rule.LOS_1);
    parsingTable.put(new Pair<>(TreeNode.Label.los, Token.TokenType.PRINT), Rule.LOS_1);
    parsingTable.put(new Pair<>(TreeNode.Label.los, Token.TokenType.SEMICOLON), Rule.LOS_1);
    parsingTable.put(new Pair<>(TreeNode.Label.los, Token.TokenType.RBRACE), Rule.LOS_2);
```

Variable	Terminal	Rule's Name	Grammar Rule
<<prog>>	PUBLIC	PROG_1	<<prog>> → public class <<ID>> { public static void main (String[] args) { <<los>> } }
<<los>>	SEMI-COLON, TYPE, WHILE, FOR, IF, ID, PRINT	LOS_1	<<los>> → <<stat>> <<los>>
<<los>>	RBRACE	LOS_2	<<los>> → ε
<<stat>>	WHILE	STAT_1	<<stat>> → <<while>>
<<stat>>	FOR	STAT_2	<<stat>> → <<for>>
<<stat>>	IF	STAT_3	<<stat>> →<<if>>
<<stat>>	ID	STAT_4	<<stat>> →<<assign>>
<<stat>>	TYPE	STAT_5	<<stat>> → <<decl>>
<<stat>>	PRINT	STAT_6	<<stat>> →<<print>>
<<stat>>	SEMI-COLON	STAT_7	<<stat>> → ;

Variable	Terminal	Rule's Name	Grammar Rule
<<while>>	WHILE	WHILE_1	<<while>> → while (<<rel expr>> <<bool expr>>) { <<los>> }
<<for>>	FOR	FOR_1	<<for>> → for (<<for start>> ; <<rel expr>> <<bool expr>> ; <<for arith>>) { <<los>> }
<<for start>>	TYPE	FOR_START_1	<<for start>> → <<decl>>
<<for start>>	ID	FOR_START_2	<<for start>> → <<assign>>
<<for start>>	SEMI-COLON	FOR_START_3	<<for start>> → ε
<<for arith>>	LPAREN, ID, NUM	FOR_ARITH_1	<<for arith>> → <<arith expr>>
<<for arith>>	RPAREN	FOR_ARITH_2	<<for arith>> → ε
<<if>>	IF	IF_1	<<if>> → if (<<rel expr>> <<bool expr>>) { <<los>> } <<else if>>
<<else if>>	ELSE	ELSE_IF_1	<<else if>> → <<else?if>> { <<los>> } <<else if>>
<<else if>>	RBRACE, SEMICOLON, TYPE, PRINT, WHILE, FOR, IF, ID	ELSE_IF_2	<<else if>> → ε
<<else?if>>	ELSE	ELSE_IFF_1	<<else?if>> → else <<poss if>>
<<poss if>>	IF	POSS_IF_1	<<poss if>> → if (<<rel expr>> <<bool expr>>)
<<poss if>>	LBRACE	POSS_IF_2	<<poss if>> → ε
<<assign>>	ID	ASSIGN_1	<<assign>> → <<ID>> = <<expr>>
<<decl>>	TYPE	DECL_1	<<decl>> → <<type>> <<ID>> <<poss assign>>
<<poss assign>>	ASSIGN	POSS_ASSIGN_1	<<poss assign>> → = <<expr>>
<<poss assign>>	SEMI-COLON	POSS_ASSIGN_2	<<poss assign>> →ε
<<print>>	PRINT	PRINT_1	<<print>> → System.out.println (<<print expr>>)

Variable	Terminal	Rule's Name	Grammar Rule
<<type>>	TYPE ("int")	TYPE_1	<<type>> → int
<<type>>	TYPE ("boolean")	TYPE_2	<<type>> → boolean
<<type>>	TYPE("char")	TYPE_3	<<type>> → char
<<expr>>	LPAREN, ID, NUM,TRUE, FALSE	EXPR_1	<<expr>> → <<rel expr>> <<bool expr>>
<<expr>>	SQUOTE	EXPR_2	<<expr>> → <<char expr>>
<<char expr>>	SQUOTE	CHAR_EXPR_1	<<char expr>> → ' <<char>> '
<<bool expr>>	EQUAL, NEQUAL, AND, OR	BOOL_EXPR_1	<<bool expr>> → <<bool op>> <<rel expr>> <<bool expr>>
<<bool expr>>	RPAREN, SEMI-COLON	BOOL_EXPR_2	<<bool expr>> → ε
<<bool op>>	EQUAL, NEQUAL	BOOL_OP_1	<<bool op>> → <<bool eq>>
<<bool op>>	AND, OR	BOOL_OP_2	<<bool op>> → <<bool log>>
<<bool eq>>	EQUAL	BOOL_EQ_1	<<bool eq>> → ==
<<bool eq>>	NEQUAL	BOOL_EQ_2	<<bool eq>> → !=
<<bool log>>	AND	BOOL_LOG_1	<<bool log>> → &&
<<bool log>>	OR	BOOL_LOG_2	<<bool log>> → ||
<<rel expr>>	LPAREN, ID, NUM	REL_EXPR_1	<<rel expr>> → <<arith expr>> <<rel expr'>>
<<rel expr>>	TRUE	REL_EXPR_2	<<rel expr>> → true
<<rel expr>>	FALSE	REL_EXPR_3	<<rel expr>> → false
<<rel expr'>>	GT, LT,GE, LE	REL_EXPRR_1	<<rel expr'>> → <<rel op>> <<arith expr>>
<<rel expr'>>	EQUAL, NEQUAL	REL_EXPRR_2	<<rel expr'>> → ε
<<rel op>>	LT	REL_OP_1	<<rel op>> → <
<<rel op>>	LE	REL_OP_2	<<rel op>> → <=
<<rel op>>	GT	REL_OP_3	<<rel op>> → >

Variable	Terminal	Rule's Name	Grammar Rule
<<rel op>>	GE	REL_OP_4	<<rel op>> → >=
<<arith expr>>	LAPREN, ID, NUM	ARITH_EXPR_1	<<arith expr>> → <<term>> <<arith expr'>>
<<arith expr'>>	PLUS	ARITH_EXPRR_1	<<arith expr'>> → + <<term>> <<arith expr'>>
<<arith expr'>>	MINUS	ARITH_EXPRR_2	<<arith expr'>> → - <<term>> <<arith expr'>>
<<arith expr'>>	EQUAL, NEQUAL, GT, LT, GE, LE, RPAREN, AND, OR, SEMI-COLON	ARITH_EXPRR_3	<<arith expr'>> → ε
<<term>>	LAPREN, ID, NUM	TERM_1	<<term>> → <<factor>> <<term'>>
<<term'>>	MULTIPLY	TERMM_1	<<term'>> → * <<factor>> <<term'>>
<<term'>>	DIVIDE	TERMM_2	<<term'>> → / <<factor>> <<term'>>
<<term'>>	MOD	TERMM_3	<<term'>> → % <<factor>> <<term'>>
<<term'>>	PLUS, MINUS, EQUAL, NEQUAL, GT, LT, GE, LE, RPAREN, AND, OR, SEMI-COLON	TERMM_4	<<term'>> → ε
<<factor>>	LPAREN	FACTOR_1	<<factor>> → (<<arith expr>>)
<<factor>>	ID	FACTOR_2	<<factor>> → <<ID>>
<<factor>>	NUM	FACTOR_3	<<factor>> → <<num>>
<<print expr>>	LPAREN, ID, NUM, TRUE, FALSE	PRINT_EXPR_1	<<print expr>> → <<rel expr>> <<bool expr>>
<<print expr>>	DQUOTE	PRINT_EXPR_2	<<print expr>> → "<<string lit>> "

Figure 1: Parsing Table

(Link to online repository: https://docs.google.com/spreadsheets/d/1rEB57OhSOyuGHSJf-n7JlpdRityNslHf7Hc8CEIC14E/edit?usp=sharing)

Each specific grammar rule will then be handled utilising the construction of a particular parser for each of them. For instance, when parsing an If statement (IF_1), its following grammar rule will be <<if>> → if (<<rel expr>> <<bool expr>>) { <<los>> } <<else if>>; this rule starts with the “If” keyword, followed up with a condition within the parentheses, a block of statements within braces and an optional ‘else if’ or “else” clause. Below is the code implemented for the if statement

```
public void parseIf_1(TreeNode tn) throws SyntaxException{
    expect(Token.TokenType.IF);
    TreeNode if_1 = new TreeNode(TreeNode.Label.terminal, new Token(Token.TokenType.IF), tn);

    expect(Token.TokenType.LPAREN);
    TreeNode lpa1 = new TreeNode(TreeNode.Label.terminal, new Token(Token.TokenType.LPAREN), tn);

    TreeNode rel_exprNode=new TreeNode(TreeNode.Label.relexpr, tn);
    parse(rel_exprNode);

    TreeNode bool_exprNode=new TreeNode(TreeNode.Label.boolexpr, tn);
    parse(bool_exprNode);

    expect(Token.TokenType.RPAREN);
    TreeNode rpa1 = new TreeNode(TreeNode.Label.terminal, new Token(Token.TokenType.RPAREN), tn);

    expect(Token.TokenType.LBRACE);
    TreeNode lbra1 = new TreeNode(TreeNode.Label.terminal, new Token(Token.TokenType.LBRACE), tn);

    TreeNode losNode=new TreeNode(TreeNode.Label.los, tn);
    parse(losNode);

    expect(Token.TokenType.RBRACE);
    TreeNode rbra1 = new TreeNode(TreeNode.Label.terminal, new Token(Token.TokenType.RBRACE), tn);

    TreeNode else_ifNode=new TreeNode(TreeNode.Label.elseifstat, tn);
    parse(else_ifNode);;

    tn.addChild(if_1);
    tn.addChild(lpa1);
    tn.addChild(rel_exprNode);
    tn.addChild(bool_exprNode);
    tn.addChild(rpa1);
    tn.addChild(lbra1);
    tn.addChild(losNode);
    tn.addChild(rbra1);
    tn.addChild(else_ifNode);

}
```

If a terminal symbol is read, the function will utilise the expected function (indicated below) to process the corresponding token. If the token is matched, it will be consumed, and the reader will move to the next token. However, it is different from the expected token in the grammar rule. A Syntax Exception will be thrown.

```
private void expect(Token.TokenType expectedType) throws SyntaxException {
    if (lookahead == null) {
        throw new SyntaxException("Unexpected end of input. Expected: " + expectedType);
    }
    if (lookahead.getType() != expectedType) {
        throw new SyntaxException("Expected: " + expectedType + " but found: " + lookahead.getType());
    }

    consume();
}
```

Moreover, within the functions that handle the grammar rules, non-terminal symbols (represented by labels) are parsed by recursively building tree nodes for each non-terminal using methods like parse(declNode). These nodes become part of the syntax tree, with each non-terminal node leading to further parsing according to its associated production rules.

3. Efficiency and Potential Improvements

At the end of the project, through effective collaboration and teamwork, we can confidently say that the efficiency and advantages of the LL(1) grammar have been successfully translated into the Java language. As outlined below, the parser's drawbacks have also been identified and discussed to explore potential improvements.

3.1. Algorithm Efficient Implementation

Within the program, the parser is designed to process input tokens in a single linear pass, resulting in a time complexity of O(n), where n is the number of tokens. This efficiency is achieved using a lookahead variable that examines only one token at a time. At the start of each grammar rule, the parser can detect the leftmost token while having prior knowledge of the current TreeNode. This allows the parser to determine the appropriate authority to apply based on the parsing table.

```
private void parse(TreeNode tn) throws SyntaxException {
    Rule rule = parsingTable.get(new Pair<>(tn.getLabel(), lookahead.getType()));
    if (rule == null) {
        throw new SyntaxException(tn.getLabel() + " Unexpected token: " + lookahead.getType());

    }
    switch(rule){
        case PROG_1:
            parseProg_1(tn);
            break;
        case LOS_1:
            parseLos_1(tn);
            break;
        case LOS_2:,
            parseLos_2(tn);
            break;
        case STAT_1:
            parseStat_1(tn);
            break;
```

Furthermore, the parser is designed to maintain a token stream and create an abstract syntax tree during parsing, which helps the program reduce memory usage. For more details, this is managed by consuming tokens from the top of the list (extracted by the top-down and leftmost ideas behind the LL(1) grammar) and only creating a TreeNode instance when needed. The Code below shows the function “consume”, which handles Token consumption and the “expect” function handling error by throwing out “SyntaxException” upon countering unexpected tokens.

```
// Move to the next token
private void consume() {

    this.lookahead = this.tokens.poll();
}

// Expect a specific token and consume it if it matches
private void expect(Token.TokenType expectedType) throws SyntaxException {
    if (lookahead == null) {
        throw new SyntaxException("Unexpected end of input. Expected: " + expectedType);
    }
    if (lookahead.getType() != expectedType) {
        throw new SyntaxException("Expected: " + expectedType + " but found: " + lookahead.getType());
    }

    consume();
}
```

In the grammar rule functions, after all the expected tokens have been matched, the parser handles any non-terminal symbols associated with that rule. It recursively calls parsing methods for each non-terminal, building tree nodes in the process. These nodes collectively form the syntax tree, where each non-terminal node can be further expanded based on its production rules. This recursive parsing allows the parser to construct a comprehensive and hierarchical representation of the program's structure, accurately reflecting the nested and sequential nature of the language constructs.

3.2. Potential Improvement

Throughout our discussions and research, our team has finally come up with some abstract ideas for improving the reliability and performance of the current Java Program. Some improvements have already been made, and some can be significantly valuable for further consideration, which are indicated below:

- **Improved Error Messages:** Providing more informative error messages could help others debug the code more easily. We could add a more informative comment and line spacing to enhance this. In the provided project, several error messages, such as the one indicated below, have been updated, which appears to be significantly helpful for our team in debugging.

```
throw new SyntaxException("Expected: " + expectedType + " but found: " + lookahead.getType());
```

- **Optimisation of the Parsing Table:** the current parsing table implements a hashmap that provides average constant time performance but may have overhead due to hashing. So, a potential improvement is that we can switch to using a 2D array for faster indexing based on ordinal values of enums in future implementation.
- **Optimisation of Lookahead Mechanism:** our current parser uses a single token lookahead, which limits the ability to parse more complex grammars. To improve this, we could implement LL(k) parsing. With LL(k) parsing, the parser could look ahead to more than one token, enabling handling more complex grammar.

4. Limitations and Solutions

During the project construction, the recursive descent LL(1) parser concept is relatively straightforward and logical when translating into a programming project. However, numerous challenges and limitations are worth indicating and considering for future implementations.

4.1. Keeping track of both terminal and variable.

At the beginning of the process, a significant challenge was effectively managing both terminal ("Token") and non-terminal ("TreeNode Label") symbols during the parsing process. Fortunately, following the constructed parsing table (as indicated below) and implementing various sub-methods, the program can handle both terminal and non-terminal tokens step by step. This is achieved as follows:

Terminals ("Token") are read and handled within the parse function of a single CFG rule, which is selected based on the current state and the leftmost token. Within the specific CFG rule function, any existing terminal token will be consumed and verified using the expected method to ensure it matches the predicted token type. If it does not match the token, a SyntaxException will be thrown.

For instance, in the parseStat_5 function, after parsing the declaration, a terminal semicolon is expected to follow. The expect method is invoked to verify this. If the semicolon is present, a terminal node (TreeNode.Label.terminal) is created with the semicolon token and added as a child to the current parse tree node (tn.addChild(sem)).

```
public void parseStat_5(TreeNode tn) throws SyntaxException{
    TreeNode declNode=new TreeNode(TreeNode.Label.decl, tn);
    parse(declNode);

    tn.addChild(declNode);
    expect(Token.TokenType.SEMICOLON);
    TreeNode sem= new TreeNode(TreeNode.Label.terminal, new Token(Token.TokenType.SEMICOLON), tn);

    tn.addChild(sem);
}
```

Meanwhile, non-terminals (TreeNode Label) within the grammar rule are then expected to be parsed to recursively build tree nodes for each non-terminal rule using parse methods such as parse(declNode). These nodes become part of the syntax tree, with each non-terminal leading to further parsing of its associated production rules.

4.2. The problem with TYPE tokens.

Due to the project requirements, various categories of TYPE tokens exist, including char, int, and boolean. However, the supplied code only provides a single TYPE token to represent all these categories, which could lead to incorrect processing by the LL(1) parser.

To address this issue, the program reads the token type from the token queue and considers the specific value of each TYPE token. This allows the program to apply the appropriate grammar rules using the parsing table, ensuring accurate parsing.

```
case PRINT_1:
    parsePrint_1(tn);
    break;
case TYPE:
    switch (lookahead.getValue().get()) {
        case "int":
            parseType_1(tn);
            break;
        case "boolean":
            parseType_2(tn);
            break;
        case "char":
            parseType_3(tn);
            break;
        default:
            throw new SyntaxException("Wrong");
    }
    break;
```

Within the program, the value of TYPE tokens is read using looked.getValue().get() and is then handled by a switch case statement to determine the corresponding grammar rules for that leftmost token and its tree node. If no matching value is found, the program will throw a SyntaxException.

4.3. Further Consideration

Within the project, the provided context-free grammar (CFG) has already been preprocessed to remove left recursion and has been left-factored. However, in future implementations, the original CFG may exhibit theseq issues, as indicated by the constraints of the LL(1) parser, and will require preprocessing. Therefore, it is essential to consider these constraints as a challenge and discover an effective way to deal with them.

5. Contribution

Our group, including Anh Tuan Hoang, Nhat Quang Do and Dinh Khai Luu, worked together closely on all aspects of the assignment. We collaborated as a team in every phase, ensuring that all team members contributed equally to both the theoretical understanding and the practical implementation of the parser.

1. **Theoretical Work:**

- Together, we constructed the FIRST and FOLLOW sets for all non-terminal symbols in the grammar. This involved analysing the grammar rules, identifying ambiguities, and resolving potential conflicts.
- We worked as a team to develop the parsing table, ensuring that each entry was correct and adhered to our defined grammar. During this process, we also discussed and clarified uncertainties about specific tokens like **public**, **while**, and **char**, ensuring the sets were accurate and aligned with the Java language structure.
- All members reviewed and validated the parsing table to ensure it was usable for the implementation phase.

2. **Code Implementation:**

   We worked closely together on the coding portion, implementing the parser step-by-step. Each member contributed to different functions and sections of the code, but we regularly collaborated to ensure consistency across the project.

- All group members participated in writing, testing, and debugging the code to ensure that it correctly followed the grammar and parsing rules we developed. We assured the code could handle the grammar rules and produce valid parse trees.

3. **Testing:**

- After implementing the parser, we tested it against various sample inputs. Each member contributed to running each test case and identifying any issues.
- As a team, we made refinements based on test results and addressed any errors. This included revisiting our theoretical work to ensure the parsing table and grammar were adequately reflected in the implementation.

4. **Reporting:**

- Anh Tuan Hoang wrote the **Introduction** and **Contribution** sections, detailing our collaborative efforts and the project's key objectives.
- Dinh Khai Luu contributed by discussing the **Efficiency** and **Potential Improvements** in the parser's design, focusing on the performance of the recursive descent approach.

- Nhat Quang Do described the **Limitations** of our current implementation and suggested potential solutions for overcoming these challenges in future.
- The **Specification Analysis** section was written collaboratively, reflecting our joint theoretical analysis and code implementation efforts.

We collaborated closely and communicated effectively throughout the project, ensuring that each member was equally involved in all project parts. By working together on both the theoretical and practical components, we ensured that the parser was implemented correctly and aligned with the grammar we had defined.