SmallLang Compiler

Part I

By Sean Diacono (408600L)

Contents

Introduction

The first part of the SmallLang Compiler includes implementing a Lexer, Parser and visitor classes. The lexer, parser and visitor classes for SmallLang were all written in Java version 8.

Task I

The Lexer

The first task for compilation of SmallLang is the process of lexical analysis which is the process of converting a String program into a list of tokens. These tokens are made up of a String lexeme which stores the raw value of the token and of a Token Identifier which stores the type of the token. The line number for every token is also stored to be used when reporting errors. The table-driven approach was used to build the lexer.

An example can be given to describe this process, consider the code below:

```
let lexerEx:int = 3;
```

The lexer will produce seven different tokens as seen below (note that the line number was excluded from the example output):

```
[
    {
        lexeme: "let"
        tokenIdentifer: TOK_LET
    },
    {
        lexeme: "lexerEx"
        tokenIdentifer: TOK_IDENTIFIER
    },
    {
        lexeme: ":"
        tokenIdentifer: TOK_COLON
    },
    {
        lexeme: "int"
        tokenIdentifer: TOK_INTTYPE
    },
    {
        lexeme: "="
        tokenIdentifer: TOK_EQUALS
    },
    {
        lexeme: "3"
        tokenIdentifer: TOK_INTEGER
    },
    {
        lexeme: ";"
        tokenIdentifer: TOK_SEMICOLON
    }
]
```

The lexer was built by first drawing up a transition diagram and table between states before moving onto the code implementation. This process is detailed below.

Transition Diagram & Table

The process of building the lexer began by designing a transition diagram and table. The diagram and table were built up by adding states one by one. The first transition diagram handles the state when a character or underscore is encountered which is the EBNF rule of SmallLang for identifiers.

$$\langle \mathit{Identifier} \rangle \quad ::= \ (\text{`\_'} \mid \langle \mathit{Letter} \rangle ) \ \{ \text{`\_'} \mid \langle \mathit{Letter} \rangle \mid \langle \mathit{Digit} \rangle \ \}$$

Below is the transition diagram with this state added:

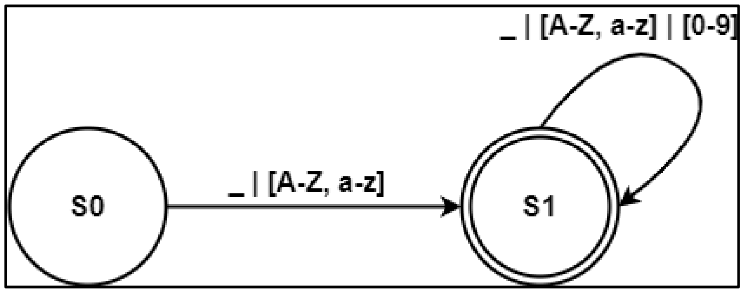


Figure 1: First State Transition Diagram

More states were added to include all characters in the EBNF, below is the final transition diagram ("char > 32 && char < 126" refers to all printable ASCII characters):

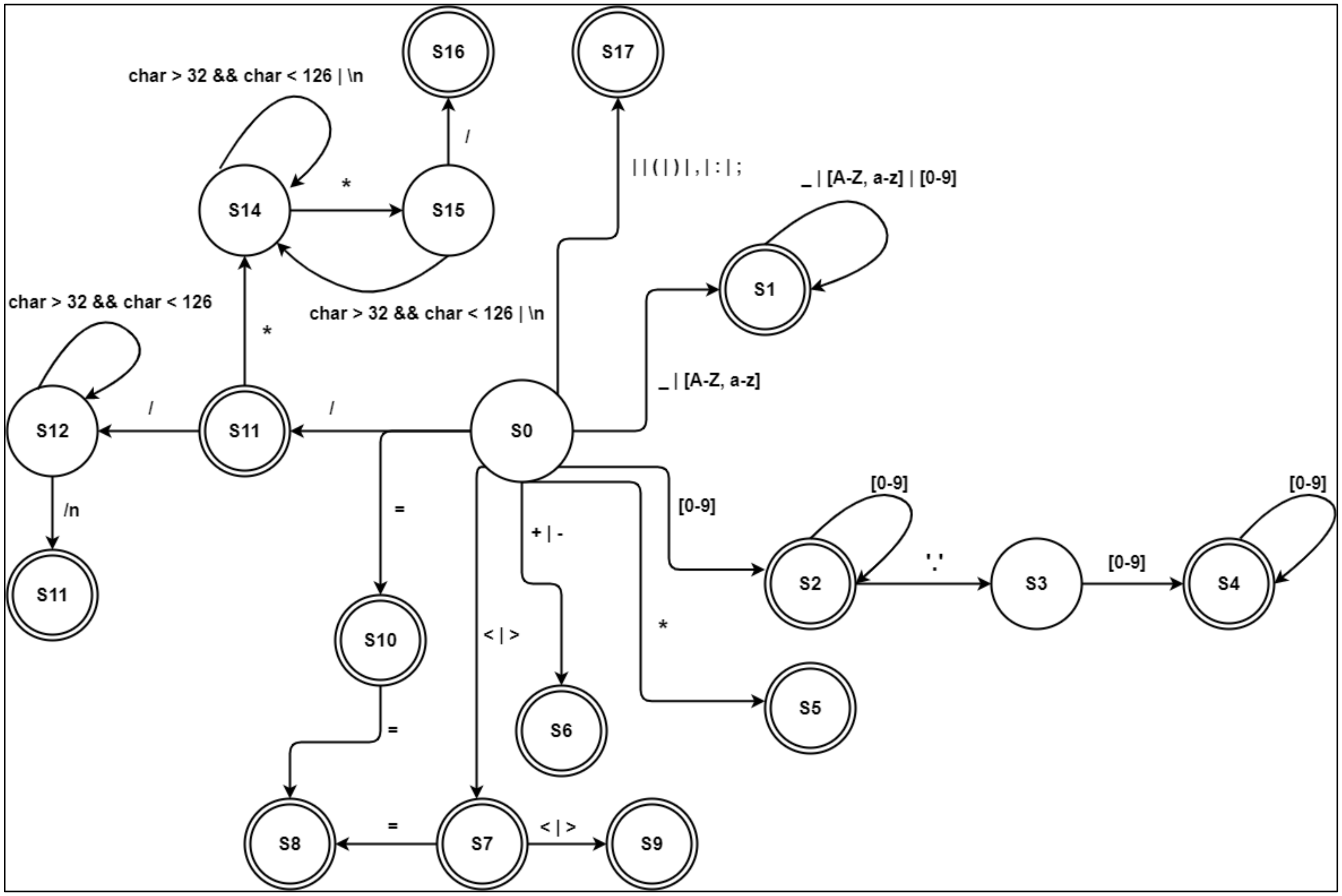


Figure 2: State Transition Diagram, Double circles represent end states.

Once the state transition diagram was built the transition table was drawn up. Below is the final transition table:

	_ | [A-Z, a-z]	**[0-9]**	**.**	*****	**+ | -**	**< | >**	**=**	**/**	**| | (|) | : | , | ;**	**char >32 && char < 126**	**\n**
S0	S1	S2	-1	S5	S6	S7	S10	S11	S17	-1	-1
S1	S1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S2	-1	S2	S3	-1	-1	-1	-1	-1	-1	-1	-1
S3	-1	S4	-1	-1	-1	-1	-1	-1	-1	-1	-1
S4	-1	S4	-1	-1	-1	-1	-1	-1	-1	-1	-1
S5	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S6	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S7	-1	-1	-1	-1	-1	S9	S8	-1	-1	-1	-1
S8	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S9	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S10	-1	-1	-1	-1	-1	-1	S8	-1	-1	-1	-1
S11	-1	-1	-1	S14	-1	-1	-1	S12	-1	-1	-1
S12	-1	-1	-1	-1	-1	-1	-1	-1	-1	S12	S13
S13	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S14	-1	-1	-1	S15	-1	-1	-1	-1	-1	S14	S14
S15	-1	-1	-1	-1	-1	-1	-1	S16	-1	S14	S14
S16	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S17	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Table 1 : Greyed out states are Error states

Implementation

Once the state transition diagram and table were drawn up the implementation of the lexer began. The first step was to create a Token Class.

Token Class

This class is used to store the data for every token. The class defines the token type as an enum of various constants. The token types created can be seen in Figure 3 below.

```
public enum tokenIdentifier {
        TOK_IDENTIFIER,
        TOK_INTEGER,
        TOK_FLOAT,
        TOK_ADDITIONOP,
        TOK_MULTIPLICATIONOP,
        TOK_BOOLEAN,
        TOK_EQUALS,
        TOK_FLOATTYPE,
        TOK_INTTYPE,
        TOK_BOOLTYPE,
        TOK_AUTOTYPE,
        TOK_NOT,
        TOK_LET,
        TOK_PRINT,
        TOK_RETURN,
        TOK_IF,
        TOK_ELSE,
        TOK_FOR,
        TOK_WHILE,
        TOK_FF,
        TOK_RELATIONALOP,
        TOK_LEFTBRACE,
        TOK_RIGHTBRACE,
        TOK_LEFTBRACKET,
        TOK_RIGHTBRACKET,
        TOK_COMMA,
        TOK_SEMICOLON,
        TOK_COLON,
        TOK_COMMENT,
        TOK_ERROR,
        TOK_EOF
}
```

Figure 3: Token Types

A token type was created for every keyword in SmallLang such as, “while”, “for”, “let”, “int”, etc. A token type was also created for all punctuation marks, all types of variable values, all addition and multiplication operators and one for identifiers.

The class also includes a String to store the lexeme the token and an Int to store the line number the token was found at. These are mainly stored for error reporting.

Lexer Class

The class includes several variables which are used during the tokenization process. These are:

- An ArrayList of characters to store all the characters read from the text file.
- An Integer to keep track of which character the lexer is currently at.
- An Integer to keep track of which line number the lexer is currently at.
- A 2D array of integers which stores the state transition table described in Table 1 above.
- A Set of Strings which stores all the keywords of SmallLang such as, "for", "ff", etc.
- A Set of Integers which stores all the states which are considered as acceptable end states (i.e. the double circles from Figure 2).

Naturally the class also features several methods used during lexical analysis.

Since the lexer is the first stage of the compiler it is also responsible for reading the SmallLang program from a text file. Therefore, the Lexer class includes a method, 'readInput()', which reads the program character by character and adding those characters to the ArrayList of characters to be used later, these characters include non-printable characters too, such as: '\n' and '\t'.

The tokenization process begins with the "tokenize()" method which returns the next token found based on the current characters found within the ArrayList of characters. The following variables are declared in the method:

```
int currentState = 0;

String lexeme = "";

Stack<Integer> visitedStates = new Stack<>();

char currentChar;
```

The error state is first pushed to the Stack of visited states and then all trailing white spaces or tabs are ignored before tokenization begins. The tokenizer then begins by extracting each character one by one from the character ArrayList, whitespaces are tabs are ignored and end of line characters indicate that the line number counter should be incremented. The character is then concatenated to the lexeme String and the current state is updated based on the old state and the new character. The method to get the current state will be discussed later. If the current state is one of the final states stored in the set described previously then the stack of visited states is cleared. The current state is then pushed onto the visited states stack. This is repeated until the next character leads to an error state or until all the characters in the ArrayList have been read.

The rollback loop is then implemented which pops a state from the visited states stack, truncates the lexeme and updates the current state to the state at the top of the stack until the current state is one of the final states or until the visited states stack is empty.

Once the rollback loop is exited, if the current state is still not a final state then a lexical error is reported and execution ends. However, if the current state is a final state then a Token is created and depending on the state the token Identifier is set to the correct one. Additionally, if the state is at state 1 or state 17, the states assigned to words and punctuation, the token identifier is picked depending on the value of the lexeme because the state number is not enough to identify which type of token. For example, if the current state is 1 and the lexeme is "ff" then the token identifier should not be set to *TOK_IDENTIFIER* but it should be set to *TOK_FF*.

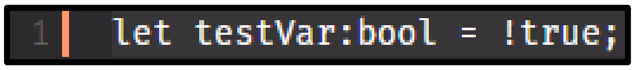
The method "getState(currentState, currentChar)" is used to return the next state based on the current state and the current char. The method returns the appropriate value from the 2D array which stores the transition table. Another method called "charToIndex" converts characters to the appropriate index of that character in the transition table.

Finally the Lexer also includes the "getNextToken()" method which is used by the Parser to retrieve the program as a list of Tokens. This method appends an end of file token to the list of tokens returned so that the parser will be able to identify when the last token has been reached and does not return comment tokens since these do not need to be parsed.

Testing

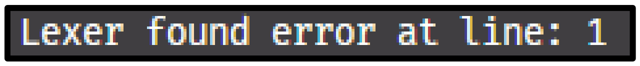
The lexer was tested by creating sample SmallLang programs with and without lexical errors and testing if the errors are detected and if correct programs are allowed.

Test 1



```
let testVar:bool = !true;
```

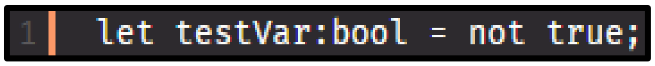
The above SmallLang code should result in a lexical error since the character '!' is not defined as an operator in SmallLang. Given this code the lexer produces the following output:



```
Lexer found error at line: 1
```

The lexer correctly finds an error and reports it at the correct line number.

Test 2



```
let testVar:bool = not true;
```

The above SmallLang does not result in a lexical error because "not" is a valid operator according to the language's rules.

Task II

The Parser

The purpose of the parser is to test that the program is syntactically correct using the sequence of tokens generated by the lexer. A successful parse of the program should generate an Abstract Syntax Tree (AST). Therefore, the first step for building the parser was to design and build all the nodes required to build the AST. A node was created for every statement and factor defined in the SmallLang EBNF. Below is a UML Class Diagram for the AST:

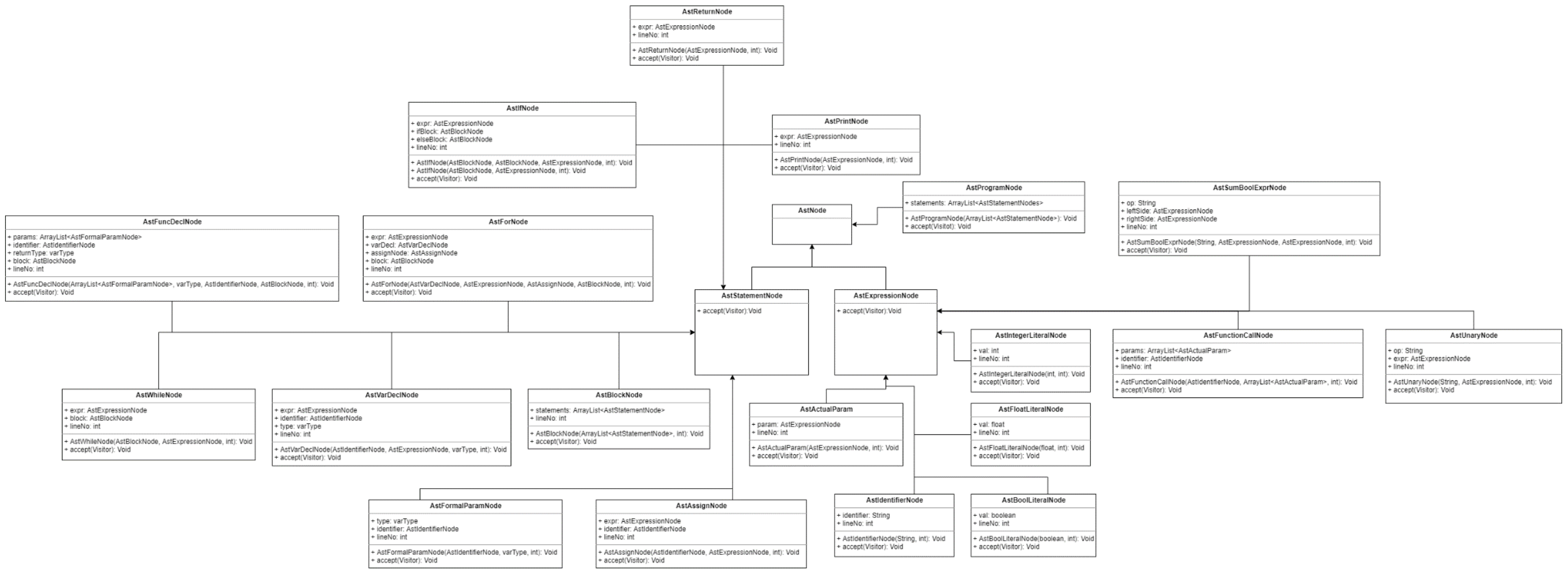

AstReturnNode
+ expr: AstExpressionNode
+ lineNo: int
+ AstReturnNode(AstExpressionNode, int): Void
+ accept(Visitor): Void
AstIfNode
+ expr: AstExpressionNode
+ ifBlock: AstBlockNode
+ elseBlock: AstBlockNode
+ lineNo: int
+ AstIfNode(AstBlockNode, AstBlockNode, AstExpressionNode, int): Void
+ AstIfNode(AstBlockNode, AstExpressionNode, int): Void
+ accept(Visitor): Void
AstPrintNode
+ expr: AstExpressionNode
+ lineNo: int
+ AstPrintNode(AstExpressionNode, int): Void
+ accept(Visitor): Void
AstProgramNode
+ statements: ArrayList<AstStatementNodes>
+ AstProgramNode(ArrayList<AstStatementNode>): Void
+ accept(Visitot): Void
AstSumBoolExprNode
+ op: String
+ leftSide: AstExpressionNode
+ rightSide: AstExpressionNode
+ lineNo: int
+ AstSumBoolExprNode(String, AstExpressionNode, AstExpressionNode, int): Void
+ accept(Visitor): Void
AstNode
AstFuncDeclNode
+ params: ArrayList<AstFormalParamNode>
+ identifier: AstIdentifierNode
+ returnType: varType
+ block: AstBlockNode
+ lineNo: int
+ AstFuncDeclNode(ArrayList<AstFormalParamNode>, varType, AstIdentifierNode, AstBlockNode, int): Void
+ accept(Visitor): Void
AstForNode
+ expr: AstExpressionNode
+ varDecl: AstVarDeclNode
+ assignNode: AstAssignNode
+ block: AstBlockNode
+ lineNo: int
+ AstForNode(AstVarDeclNode, AstExpressionNode, AstAssignNode, AstBlockNode, int): Void
+ accept(Visitor): Void
AstStatementNode
+ accept(Visitor):Void
AstExpressionNode
+ accept(Visitor):Void
AstIntegerLiteralNode
+ val: int
+ lineNo: int
+ AstIntegerLiteralNode(int, int): Void
+ accept(Visitor): Void
AstFunctionCallNode
+ params: ArrayList<AstActualParam>
+ identifier: AstIdentifierNode
+ lineNo: int
+ AstFunctionCallNode(AstIdentifierNode, ArrayList<AstActualParam>, int): Void
+ accept(Visitor): Void
AstUnaryNode
+ op: String
+ expr: AstExpressionNode
+ lineNo: int
+ AstUnaryNode(String, AstExpressionNode, int): Void
+ accept(Visitor): Void
AstWhileNode
+ expr: AstExpressionNode
+ block: AstBlockNode
+ lineNo: int
+ AstWhileNode(AstBlockNode, AstExpressionNode, int): Void
+ accept(Visitor): Void
AstVarDeclNode
+ expr: AstExpressionNode
+ identifier: AstIdentifierNode
+ type: varType
+ lineNo: int
+ AstVarDeclNode(AstIdentifierNode, AstExpressionNode, varType, int): Void
+ accept(Visitor): Void
AstBlockNode
+ statements: ArrayList<AstStatementNode>
+ lineNo: int
+ AstBlockNode(ArrayList<AstStatementNode>, int): Void
+ accept(Visitor): Void
AstActualParam
+ param: AstExpressionNode
+ lineNo: int
+ AstActualParam(AstExpressionNode, int): Void
+ accept(Visitor): Void
AstFloatLiteralNode
+ val: float
+ lineNo: int
+ AstFloatLiteralNode(float, int): Void
+ accept(Visitor): Void
AstFormalParamNode
+ type: varType
+ identifier: AstIdentifierNode
+ lineNo: int
+ AstFormalParamNode(AstIdentifierNode, varType, int): Void
+ accept(Visitor): Void
AstAssignNode
+ expr: AstExpressionNode
+ identifier: AstIdentifierNode
+ lineNo: int
+ AstAssignNode(AstIdentifierNode, AstExpressionNode, int): Void
+ accept(Visitor): Void
AstIdentifierNode
+ identifier: String
+ lineNo: int
+ AstIdentifierNode(String, int): Void
+ accept(Visitor): Void
AstBoolLiteralNode
+ val: boolean
+ lineNo: int
+ AstBoolLiteralNode(boolean, int): Void
+ accept(Visitor): Void

Figure 4: AST Nodes Class Diagram

A mix of inheritance and composition is used to describe nodes. The parent node is "AstNode"class which has three direct children, "AstStatementNode", "AstExpressionNode" and "AstProgramNode". All statements extend from "AstStatementNode" and all expressions from "AstExpressionNode", composition is then used to create objects of other nodes within a node.

For example, the "AstForNode" extends from "AstStatementNode" and includes five variables:

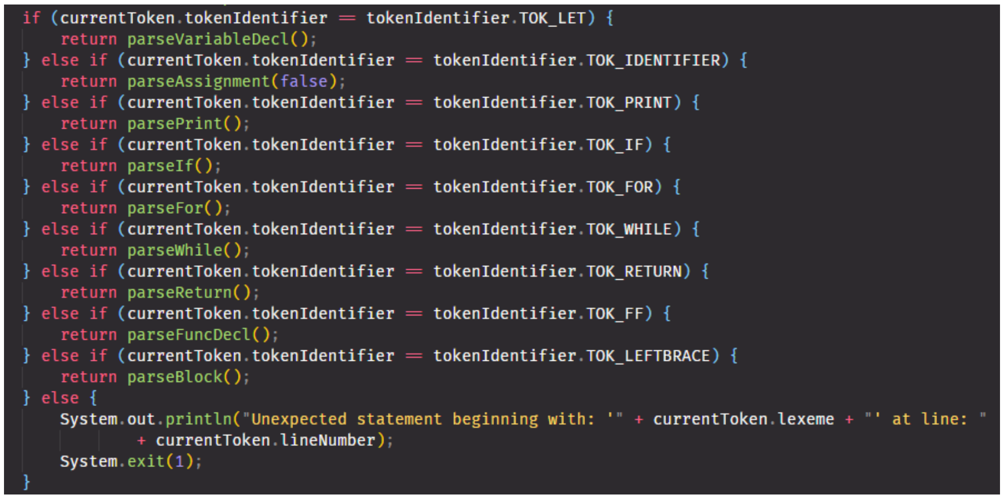
- "expr" of type AstExpressionNode. In a for loop this is used as the loop condition.
- "varDecl" of type AstVarDeclNode. This stores the variable declaration in the loop header.
- "assignNode" of type AstAssignNode. This stores the variable assignment in the loop header.
- "block" of type AstBlockNode. This stores the statements in the block of the for loop.
- "lineNo" of type int. This stores the line number of the for loop header to be used for error reporting.

All the nodes include a constructor and an "accept(Visitor)" method for use in Tasks 3-5.

Implementation

The next step to building the parser was to create a Parser Class, this class handles the parsing process. The process begins in the "parse()" method which is used to parse the whole program and return an AstProgramNode which is a node with an ArrayList of AstStatementNodes. The method starts by requesting the sequence of tokens from the lexer and storing them in an ArrayList. A method called "changeCurrentToken()" was created which sets the current token to the next element in the tokens ArrayList and the next token to the token following that one. Parsing then begins by calling the "parseStatement()" method because according to the rules of the language a block is made up of a list of statements, once this method returns an AstStatementNode it is added to a list of statements, this is repeated until the end of file token is reached. An AstProgramNode is then built with that list of statements and is returned by the "parse()" method.

Parsing of statements is done by delegating a parse method to every different type of statement, this method then returns the appropriate AstNode, for example the "parseIf()" method returns an AstIfNode. The parseStatement method mentioned above is used to call the correct parse method depending on the first token of the statement.



```java
if (currentToken.tokenIdentifier == tokenIdentifier.TOK_LET) {
    return parseVariableDecl();
} else if (currentToken.tokenIdentifier == tokenIdentifier.TOK_IDENTIFIER) {
    return parseAssignment(false);
} else if (currentToken.tokenIdentifier == tokenIdentifier.TOK_PRINT) {
    return parsePrint();
} else if (currentToken.tokenIdentifier == tokenIdentifier.TOK_IF) {
    return parseIf();
} else if (currentToken.tokenIdentifier == tokenIdentifier.TOK_FOR) {
    return parseFor();
} else if (currentToken.tokenIdentifier == tokenIdentifier.TOK_WHILE) {
    return parseWhile();
} else if (currentToken.tokenIdentifier == tokenIdentifier.TOK_RETURN) {
    return parseReturn();
} else if (currentToken.tokenIdentifier == tokenIdentifier.TOK_FF) {
    return parseFuncDecl();
} else if (currentToken.tokenIdentifier == tokenIdentifier.TOK_LEFTBRACE) {
    return parseBlock();
} else {
    System.out.println("Unexpected statement beginning with: '" + currentToken.lexeme + "' at line: "
            + currentToken.lineNumber);
    System.exit(1);
}
```

Figure 5: parseStatement() method

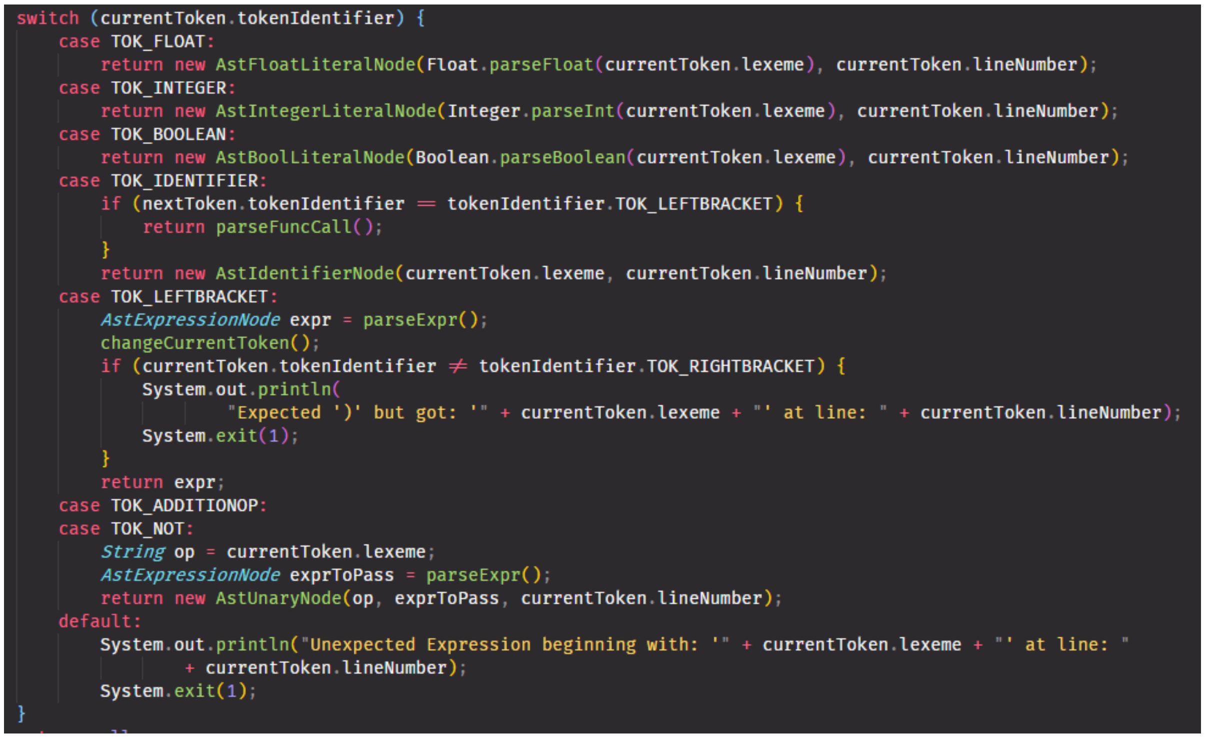
As seen in figure 5 if the statement begins with the incorrect token a syntax error is raised, otherwise the parse method for the correct statement is called. Most of the parse methods follow a similar pattern of checking if the current token is the expected one and raising an error if it is not, therefore only the parseVariableDecl(), parseBlock() and parseExpr() methods will be explained bellow.

The parsing of variable declarations begins by checking that the token following the "let" keyword is an identifier. If an identifier is found an object of type AstIdentifierNode is created to store this identifier, naturally if an identifier is not found a syntax error is reported and execution ends. Then the method checks that the next token is a colon token and if it is it then the variable type is parsed, similarly if everything is as expected it checks that the next token is an equals token before then calling the parseExpr() method to parse the expression. Finally, the method checks if the statement ends in a semicolon before creating and returning an object of type AstVarDeclNode which stores the identifier, variable type, expression and line number.

The method parseBlock() is called multiple times throughout the Parser class, for example it is called from the parseFor() and parseIf() methods to parse the blocks of those statements. It is also called if a statement begins with an opening brace. This method is similar to the parse() method since it constructs a list of AstStatementNodes by calling parseStatement() until a closing brace is reached. Then an AstBlockNode object is created which stores the list of statements in the block and the line number and it is returned.

The parseExpr() method is responsible for parsing all expressions. This method relies on another 3 methods to work, this because during parsing it follows the same flow used in the EBNF. So when parseExpr() is called it calls a method parseSimpleExpr() which then calls parseTerm() which finally calls parseFactor(). In parseExpr() after calling parseSimpleExpr() the next token is checked and if it is a relational operator token then parseExpr() is called again because according to the EBNF another simple expression should be expected. Similarly in parseSimpleExpr() after calling parseTerm() the next token is checked and if it is a addition operator then according to the EBNF rules another term should be expected therefore parseSimpleExpr() is called again. The same thing is done in the parseFactor() method where if the next token is a multiplication operator then another Factor should be expected.

The parseFactor() (Figure 6) method is then in charge of deciding which factor AstNode it should return. If the current token is a literal then the appropriate AstLiteralNode is returned, if the current token is an identifier then either a function call is parsed (this decision is made by checking if the token after the identifier is an opening bracket) or an AstIdentifierNode is returned, if the current token is an opening bracket then another Expression is expected therefore parseExpr() is called and if the current token is a '-' or a "not" then an AstUnaryNode is returned.



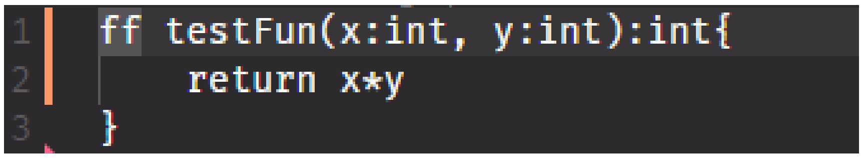
```
switch (currentToken.tokenIdentifier) {
    case TOK_FLOAT:
        return new AstFloatLiteralNode(Float.parseFloat(currentToken.lexeme), currentToken.lineNumber);
    case TOK_INTEGER:
        return new AstIntegerLiteralNode(Integer.parseInt(currentToken.lexeme), currentToken.lineNumber);
    case TOK_BOOLEAN:
        return new AstBoolLiteralNode(Boolean.parseBoolean(currentToken.lexeme), currentToken.lineNumber);
    case TOK_IDENTIFIER:
        if (nextToken.tokenIdentifier == tokenIdentifier.TOK_LEFTBRACKET) {
            return parseFuncCall();
        }
        return new AstIdentifierNode(currentToken.lexeme, currentToken.lineNumber);
    case TOK_LEFTBRACKET:
        AstExpressionNode expr = parseExpr();
        changeCurrentToken();
        if (currentToken.tokenIdentifier != tokenIdentifier.TOK_RIGHTBRACKET) {
            System.out.println(
                    "Expected ')' but got: '" + currentToken.lexeme + "' at line: " + currentToken.lineNumber);
            System.exit(1);
        }
        return expr;
    case TOK_ADDITIONOP:
    case TOK_NOT:
        String op = currentToken.lexeme;
        AstExpressionNode exprToPass = parseExpr();
        return new AstUnaryNode(op, exprToPass, currentToken.lineNumber);
    default:
        System.out.println("Unexpected Expression beginning with: '" + currentToken.lexeme + "' at line: "
                + currentToken.lineNumber);
        System.exit(1);
}
```

Figure 6: parseFactor()

Testing

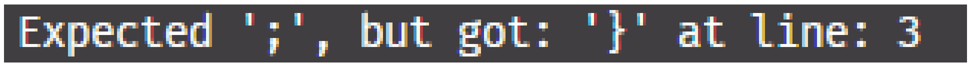
Testing of the parser was conducted similarly to testing of the lexer, programs with syntax errors and programs without syntax errors were both tested.

Test 1



```
ff testFun(x:int, y:int):int{
    return x*y
}
```

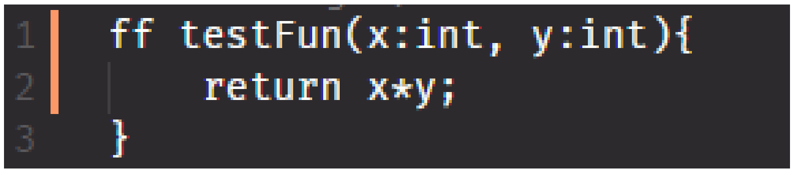
The above code is missing a semi-colon after the return statement. The parser outputs the following error:



```
Expected ';', but got: '}' at line: 3
```

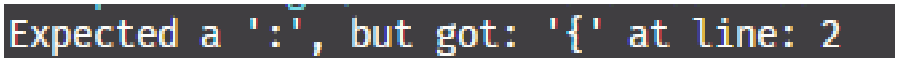
It correctly outputs what the expected token was and what it got at the correct line number.

Test 2



```
ff testFun(x:int, y:int){
    return x*y;
}
```

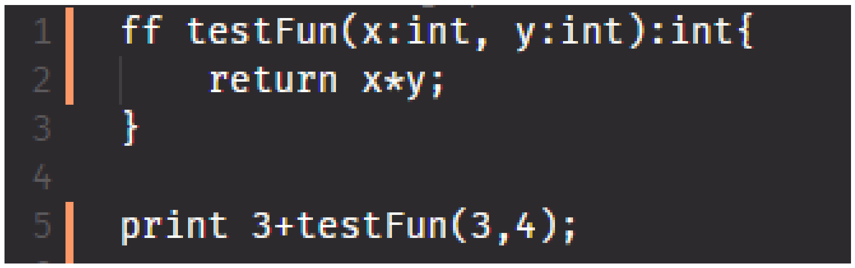
The above function declaration is missing a return type. The parser outputs the following error:



```
Expected a ':', but got: '{' at line: 2
```

The is correctly reported by the parser.

Test 4



```
ff testFun(x:int, y:int):int{
    return x*y;
}

print 3+testFun(3,4);
```

The above code is syntactically correct, and the parser did not report any errors when parsing it.

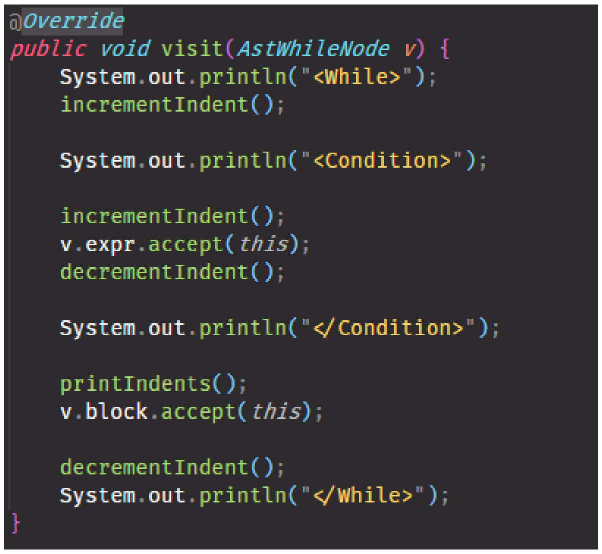
Task III

XML Generation

The purpose of this task is to implement a Visitor class which can build the AST tree produced by the parser in XML. A visitor Interface was made with visit methods for every AstNode in the AST, accept methods were also added to the nodes. The XML Visitor class implements this interface.

Implementation

The XMLVisitor class has a visit method for every type of node because it implements the Visitor Interface. Each method prints the appropriate XML tags according to the type of node.



```java
@Override
public void visit(AstWhileNode v) {
    System.out.println("<While>");
    incrementIndent();

    System.out.println("<Condition>");

    incrementIndent();
    v.expr.accept(this);
    decrementIndent();

    System.out.println("</Condition>");

    printIndents();
    v.block.accept(this);

    decrementIndent();
    System.out.println("</While>");
}
```

Figure 7: visit method for the AstWhileNode

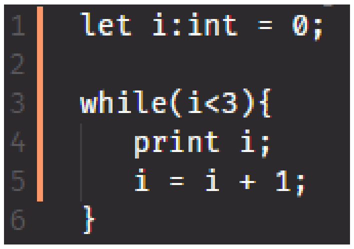
Figure 7 displays the visit method for the AstWhileNode, the while tag is printed and an indent is added. The XMLVisitor is passed to the while loop condition and to the while block. Methods such as incrementIndent(), decrementIndent() and printIndents() had to be added to keep track of the indentation level when printing the AST in XML.

All the different AstNodes follow a similar pattern where the correct tags are printed and any variables which are other nodes are passed the XMLVisitor class.

Testing

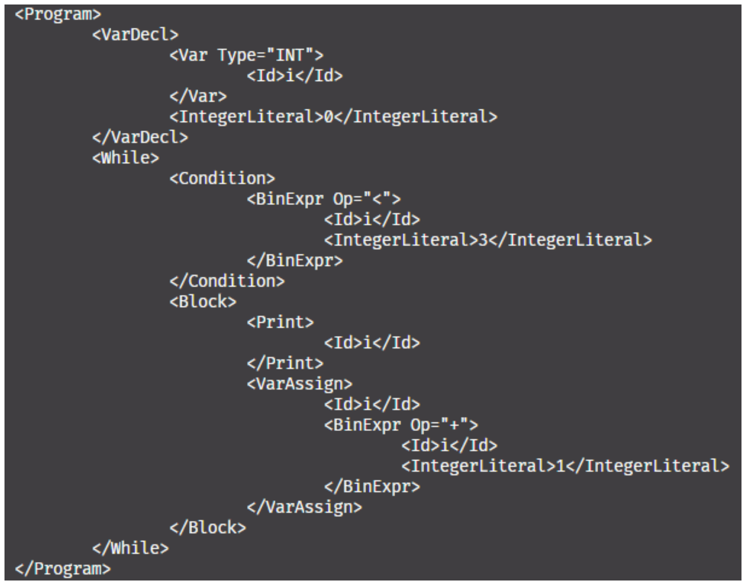
For testing sound programs were parsed and then printed by the XML Visitor.

Test 1



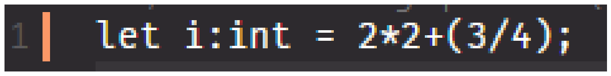
```
let i:int = 0;

while(i<3){
    print i;
    i = i + 1;
}
```

The XML tree generated from this SmallLang program is:



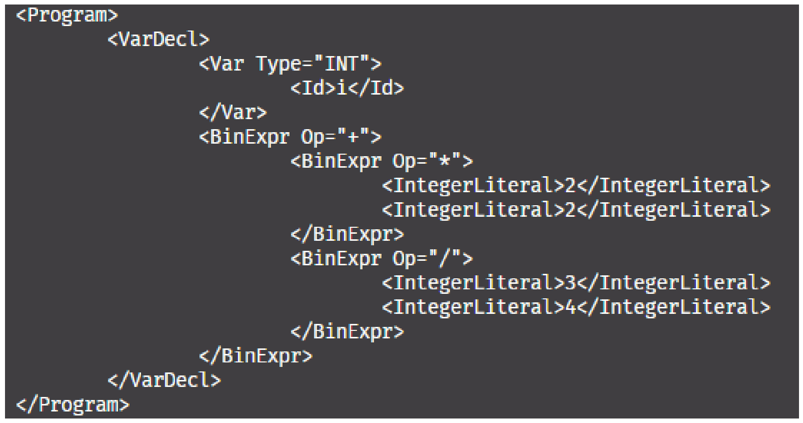
```
<Program>
        <VarDecl>
                <Var Type="INT">
                        <Id>i</Id>
                </Var>
                <IntegerLiteral>0</IntegerLiteral>
        </VarDecl>
        <While>
                <Condition>
                        <BinExpr Op="<">
                                <Id>i</Id>
                                <IntegerLiteral>3</IntegerLiteral>
                        </BinExpr>
                </Condition>
                <Block>
                        <Print>
                                <Id>i</Id>
                        </Print>
                        <VarAssign>
                                <Id>i</Id>
                                <BinExpr Op="+">
                                        <Id>i</Id>
                                        <IntegerLiteral>1</IntegerLiteral>
                                </BinExpr>
                        </VarAssign>
                </Block>
        </While>
</Program>
```

Test 2



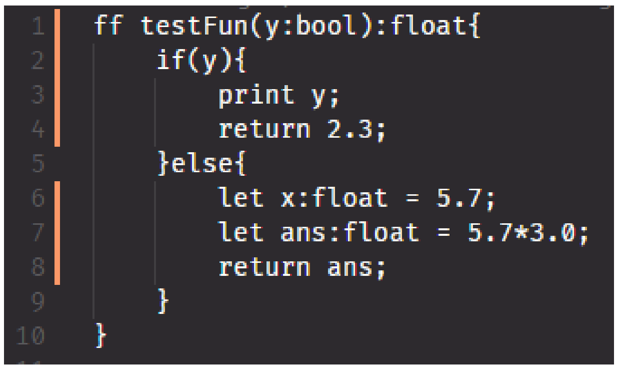
```
let i:int = 2*2+(3/4);
```

The XML tree generated is:



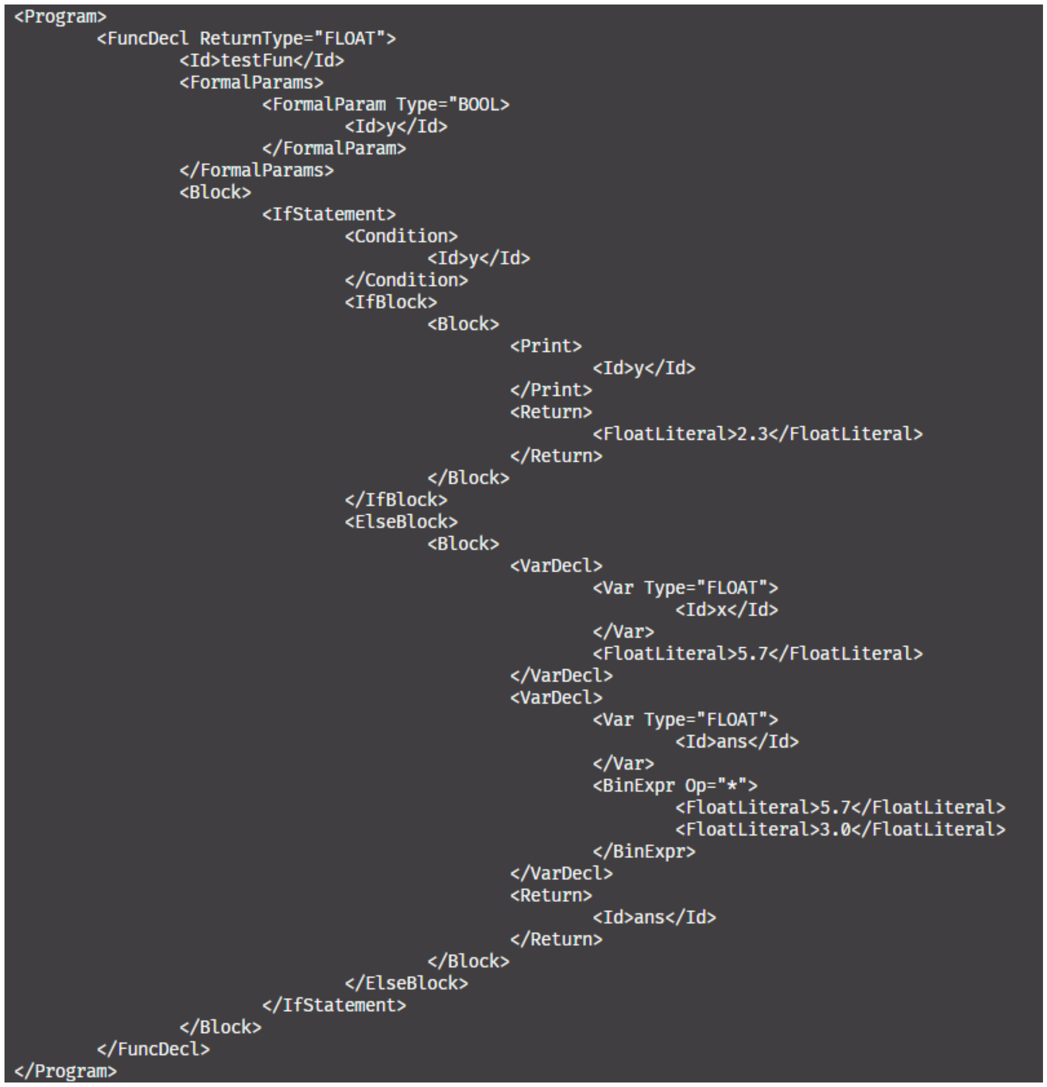
```
<Program>
        <VarDecl>
                <Var Type="INT">
                        <Id>i</Id>
                </Var>
                <BinExpr Op="+">
                        <BinExpr Op="*">
                                <IntegerLiteral>2</IntegerLiteral>
                                <IntegerLiteral>2</IntegerLiteral>
                        </BinExpr>
                        <BinExpr Op="/">
                                <IntegerLiteral>3</IntegerLiteral>
                                <IntegerLiteral>4</IntegerLiteral>
                        </BinExpr>
                </BinExpr>
        </VarDecl>
</Program>
```

Test 3



```
ff testFun(y:bool):float{
    if(y){
        print y;
        return 2.3;
    }else{
        let x:float = 5.7;
        let ans:float = 5.7*3.0;
        return ans;
    }
}
```

The XML tree Generated is:



```
<Program>
        <FuncDecl ReturnType="FLOAT">
                <Id>testFun</Id>
                <FormalParams>
                        <FormalParam Type="BOOL>
                                <Id>y</Id>
                        </FormalParam>
                </FormalParams>
                <Block>
                        <IfStatement>
                                <Condition>
                                        <Id>y</Id>
                                </Condition>
                                <IfBlock>
                                        <Block>
                                                <Print>
                                                        <Id>y</Id>
                                                </Print>
                                                <Return>
                                                        <FloatLiteral>2.3</FloatLiteral>
                                                </Return>
                                        </Block>
                                </IfBlock>
                                <ElseBlock>
                                        <Block>
                                                <VarDecl>
                                                        <Var Type="FLOAT">
                                                                <Id>x</Id>
                                                        </Var>
                                                        <FloatLiteral>5.7</FloatLiteral>
                                                </VarDecl>
                                                <VarDecl>
                                                        <Var Type="FLOAT">
                                                                <Id>ans</Id>
                                                        </Var>
                                                        <BinExpr Op="*">
                                                                <FloatLiteral>5.7</FloatLiteral>
                                                                <FloatLiteral>3.0</FloatLiteral>
                                                        </BinExpr>
                                                </VarDecl>
                                                <Return>
                                                        <Id>ans</Id>
                                                </Return>
                                        </Block>
                                </ElseBlock>
                        </IfStatement>
                </Block>
        </FuncDecl>
</Program>
```

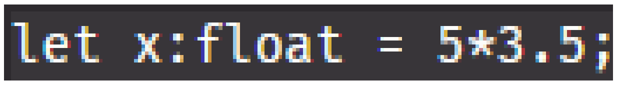
Task IV

Semantic Analysis

The aim of this task was to perform type and declaration checking, a new visitor class for semantic analysis which includes the visit methods for all the nodes and a symbol table to keep track of the variables and functions declared and their respective types. Every scope has a separate symbol table to be able to differentiate which variables have been declared in different scopes. Some rules had to be added on to the ones given in the specification, these are:

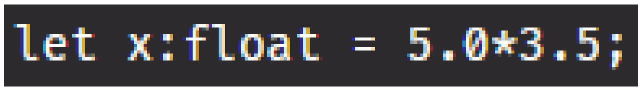
Int & Float Literals

The semantic analysis implemented does not allow for variables of type int and float to be used together in expressions. For example, the following statement is not allowed:



```
let x:float = 5*3.5;
```

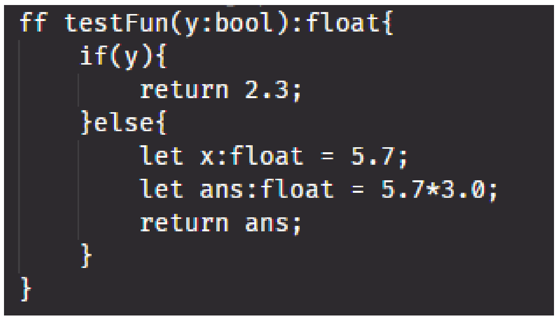
The expression includes an integer, '5', so for the expression to be valid it should be written as:



```
let x:float = 5.0*3.5;
```

Functions & Returns

All functions declared must have an accessible return within their block. The function below is valid because it is guaranteed to return:

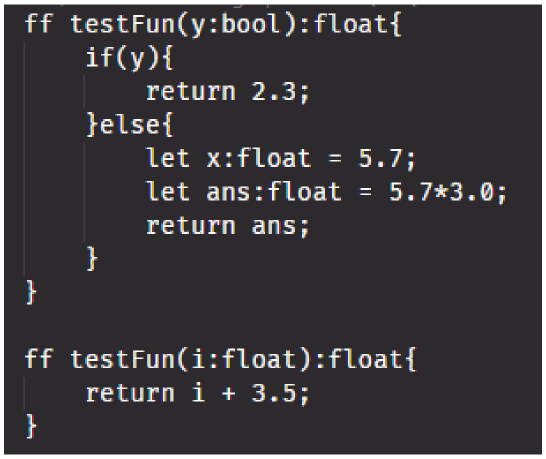


```
ff testFun(y:bool):float{
    if(y){
        return 2.3;
    }else{
        let x:float = 5.7;
        let ans:float = 5.7*3.0;
        return ans;
    }
}
```

However, if the return statement is removed from the else block than the function becomes invalid and an error would be raised.

Function Overloading

Functions with the same name but different parameters are allowed, so the bellow two functions below can exist in the same scope:

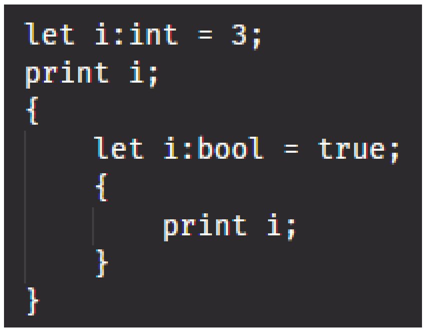


```
ff testFun(y:bool):float{
    if(y){
        return 2.3;
    }else{
        let x:float = 5.7;
        let ans:float = 5.7*3.0;
        return ans;
    }
}

ff testFun(i:float):float{
    return i + 3.5;
}
```

As might be expected functions with the same identifier and the same parameters are not allowed to be declared in the same scope.

Variable Shadowing

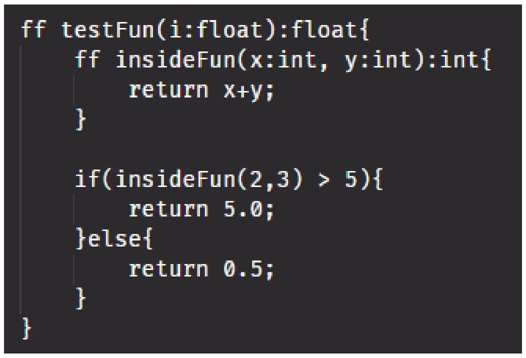
Variables with the same identifier can be declared multiple times if they exist within different scopes. Variables declared within scopes then shadow variables with the same identifier which exist in parent scopes. In the below example a variable with the identifier "i" is declared twice within different scopes. The first print statement will refer to the variable with type int whilst the second print statement will refer to the variable of type bool since it is declared in the scope closest to the print statement.



```
let i:int = 3;
print i;
{
    let i:bool = true;
    {
        print i;
    }
}
```

Functions in Functions

Functions can be declared within other functions. The function below is an example of how a function can be declared and called from within another function.

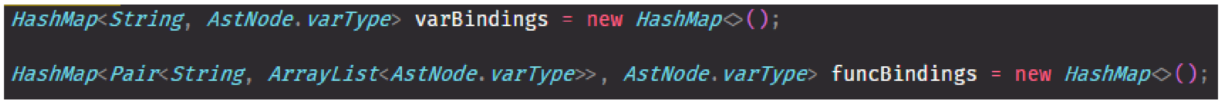


```
ff testFun(i:float):float{
    ff insideFun(x:int, y:int):int{
        return x+y;
    }

    if(insideFun(2,3) > 5){
        return 5.0;
    }else{
        return 0.5;
    }
}
```

Implementation

The first step was to build a class to define scopes, this class holds all the variable and function bindings in a symbol table. It also provides methods such as lookup and insert to be able to check if a variable or function exists within the scope and to add a variable or function binding to the scope.

The data structures used to store these bindings are below:



```
HashMap<String, AstNode.varType> varBindings = new HashMap<>();

HashMap<Pair<String, ArrayList<AstNode.varType>>, AstNode.varType> funcBindings = new HashMap<>();
```

Variable bindings are stored in a Hash map where the key is the variable identifier and the value is the type. On the other hand, function bindings are also stored in a Hash map where the key is a tuple of the function identifier and a list of the parameter types and the value is the return type. The key for function bindings was set as a tuple to allow function overloading.

The semantic analyzer class was then built, it implements the original Visitor class, so it also includes the visit methods for all the AstNodes. Below is a list of the member variables in the class which are used throughout the analysis process:

- A Stack of Scopes, this is used to keep track of all the scopes in the program.
- A variable of type Scope which stores the current Scope.
- A variable of type varType which stores the type of the current expression being analyzed, this variable is set when analyzing one of the factors in the EBNF.
- A Boolean flag to indicate when a new scope should be pushed or not.
- A stack of type "functionsAssignReturn" which is a class which stores the data of a function which is still to be checked if the return expression matches the return

type. These are stored in a stack to easily keep track of which function should be assigned a return next.

The class also includes several methods besides the visit methods which are used during the analysis process. Below is a list of these methods:

- A push() method which pushes a new Scope to the stack of scopes.
- A pop() method which pops a Scope from the stack of scopes.
- A checkForReturn() method which is a recursive method which given a statement checks if it contains a return statement. This is recursive because if the given statement is a “for”, “while”, “if” or “block” statement then the blocks of those statements must also be checked.
- A setReturnType() method which checks if the return expression type is equal to a function’s return type. This function is taken from the previously mentioned stack of “functionsAssignReturn”. This method is called after a return expression has been analyzed.
- A getIdentifierType() method which takes an identifier and sets the expression type to the type of the identifier. This method follows the variable shadowing rules mentioned previously.

Similarly to the XMLVisitor class the visit methods for the different AstNodes follow a similar pattern. Therefore, the AstBlockNode, AstFuncDeclNode, AstSumBoolExpr and the AstIfNode will be explained rather than all of them.

AstBlockNode

This visit method pushes a new scope to the stack of scopes unless the ignoreBlockPush flag is set to true. If the flag is set to true it means that a new scope has already been pushed and does not need to be pushed by this method. It then loops through all the statements in the block and passes the Semantic Analysis class to them. Once analysis of all the statements has been done a scope from the stack of scopes is popped.

AstFuncDeclNode

This visit method begins by storing all the formal parameters in an array list of types and in a map of variable bindings like the one mentioned in the Scope class. Once all the parameters have been stored the identifier of the function is stored. Then the method checks if a function with the same identifier and the same parameter types has already been declared in the current scope, if one is already declared then an error is declared and execution ends. If a function hasn’t been declared, then the return type is stored and the checkForReturn method mentioned previously is used to check if the function block includes a return.

If the function includes a return statement, then it is pushed onto the stack of functions which are to be checked if the return expression type matches the return type. Then a new scope is pushed onto the stack of scopes and the variables in the parameters are added to the variable bindings of the new scope. The ignoreBlockPush flag is set to true since a new scope has already been pushed and the block is then analyzed.

AstSumBoolExprNode

This method gets the expression type of both the left- and right-hand sides of the expression and performs type checking based on the types and the operator of the expression. The expression type is set whenever one of the factors from the EBNF is met, for example if a Boolean literal is met than the expression type is set to bool. After getting the left and right side expression types a number of checks are performed, if the left and right side expression type do not match then an error is raised. If the operator is one of the following operators; “*”, “/”, “+”, “-“, and the expression type is a bool then an error is raised because those operators are only allowed in expressions of type int or float. A similar check is performed if the operator is “and” or “or”, if the expression type is not a bool then an error is reported. Then if the operator is one of the relational operators then the expression type is set to bool since the result of that expression will be a Boolean.

AstIfNode

This visit method first passes the semantic analysis class to its condition expression and if the expression type is not Boolean an error is reported since an if condition needs to result in a Boolean. The same concept is applied to “for” and “while” loop conditions. Then the semantic analysis class is passed to its ifBlock and to its elseBlock if it has one.

Auto Type Deduction

Semantic analysis is also responsible for auto type deduction where variables given the type auto need to be assigned the correct type based on the expression they are assigned to. This deduction is may be made in two places, in variable declarations and in function returns.

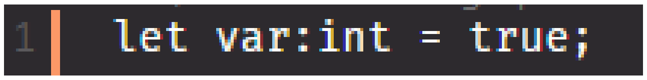
In variable declarations the deduction is made once the expression has been analyzed, the variable type is set to the type of the expression. In function returns this is done in the “setReturnType()” method previously mentioned, the return type is then set to the type of the expression in the return statement.

Testing

Testing was done by creating programs which break syntax rules to make sure that errors are reported.

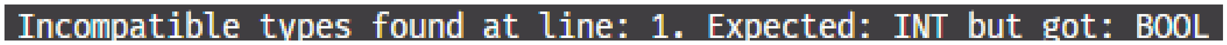
Test 1

The first test was trying to declare a variable with the wrong type.



```
let var:int = true;
```

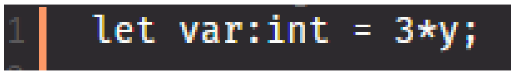
The error reported was:



```
Incompatible types found at line: 1. Expected: INT but got: BOOL
```

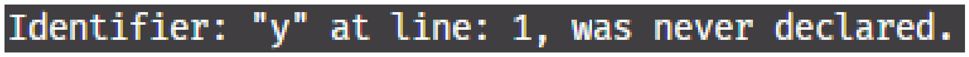
Test 2

The second test was trying to use a variable which was never declared.



```
let var:int = 3*y;
```

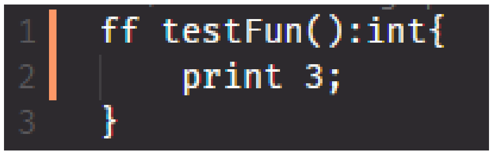
The error reported was:



```
Identifier: "y" at line: 1, was never declared.
```

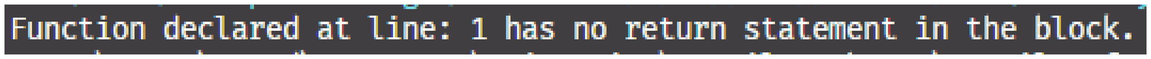
Test 3

The third test was trying to create a function without a return statement.



```
ff testFun():int{
    print 3;
}
```

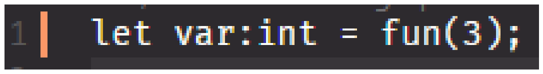
The error reported was:



```
Function declared at line: 1 has no return statement in the block.
```

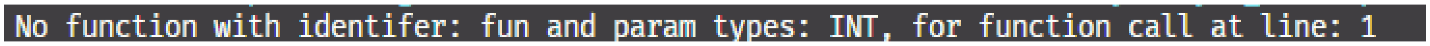
Test 4

The fourth test was trying to call a function which was never declared.



```
let var:int = fun(3);
```

The error reported was:



```
No function with identifer: fun and param types: INT, for function call at line: 1
```

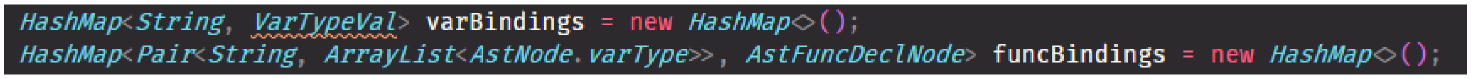
Task V

Interpreter

The goal of this task was to create an interpreter for SmallLang so after Tasks 1-3 are completed the code is run and an output is produced. Another visitor class called InterpreterVisitor and another scope class had to be implemented. Both classes resemble the semantic analysis visitor class and scope class explained in Task 4. The new scope class is different to the semantic analysis scope class because rather than storing the variable identifier and type in the symbol table the value of the variable also needs to be stored.

Implementation

A class called VarTypeVal was created which stores the type of the variable and the value, the class makes use of Generic Types so that the type of the variable value can be assigned at instantiation. Below are the modified data structures of the symbol table.

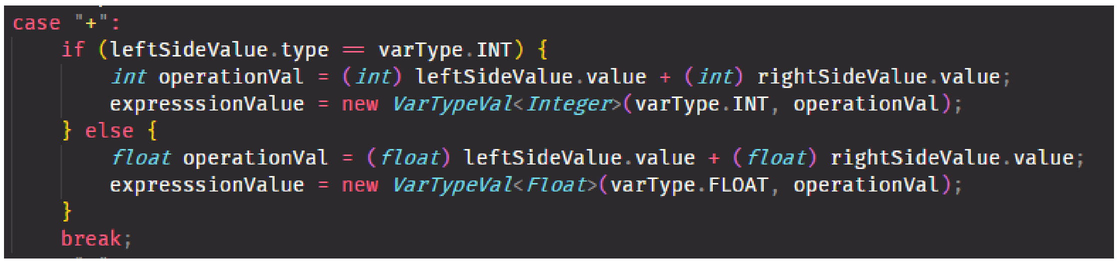


```
HashMap<String, VarTypeVal> varBindings = new HashMap<>();
HashMap<Pair<String, ArrayList<AstNode.varType>>, AstFuncDeclNode> funcBindings = new HashMap<>();
```

The member variables of the class are the same as the semantic analysis class except one more Boolean flag called "ignoreBlockPop" was added, and the expression type variable was changed to be of type VarTypeVal so that the value of an expression is also stored. Besides all the visit methods the only other method is "getIdentifierValue" which is an adaptation of the method in the semantic analysis class called "getIdentiferType", rather than retrieving the type of the identifier it retrieves the value. This also follows the previously explained rule of variable shadowing. Below are some of the visit methods explained in detail.

AstSumBoolExprNode

This visit works by retrieving the expression values of the left and right-side expressions and then performing the correct operation on the two values based on the operator of the expression. Below is a snippet of the operation performed when the operator is "+".



```
case "+":
    if (leftSideValue.type == varType.INT) {
        int operationVal = (int) leftSideValue.value + (int) rightSideValue.value;
        expresssionValue = new VarTypeVal<Integer>(varType.INT, operationVal);
    } else {
        float operationVal = (float) leftSideValue.value + (float) rightSideValue.value;
        expresssionValue = new VarTypeVal<Float>(varType.FLOAT, operationVal);
    }
    break;
```

The member variable “expressionValue” is set to value of the operation performed. In all the factors of the EBNF a similar pattern is followed where an operation is performed and then the expression value is updated to the new value. For example, in Unary expressions the operation is performed in the same way the code above works, and in Function calls the function called is evaluated given the actual parameters in the function call and the expression value is set to the return value.

AstVarDeclNode

This method gets the current scope and insert a new variable binding into the current scope. The variable binding stores both the identifier of the variable and the type and value of the variable.

AstIfNode

The AstIfNode visit method works by first evaluating the condition expression and if it is true the if block is processed but if its not the else block is.

AstForNode

This visit method pushes a new scope to the stack of scopes immediately because if a variable is declared in the for header it should be accessible in the block of the loop but not outside it. Therfore the ignoreBlockPush flag is set to true so that when the block visit method is evaluated a new scope isn’t pushed again. The ignoreBlockPop flag is also set to true so that the scope isn’t popped when the block ends, because if the loop needs the loop variable shouldn’t be discarded. Therefore, a scope is popped when the loop ends. The loop works by looping while the loop expression remains true, it is reevaluated at every iteration of the loop.

The same idea is applied to while loops too where the block is repeatedly processed until the loop condition is false.

AstPrintNode

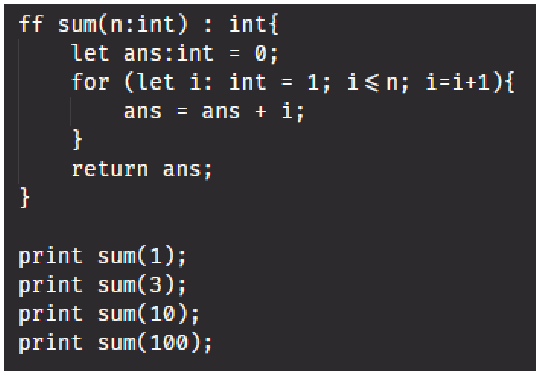
Finally, this visit method is responsible for printing out the value of expressions in a print statement. The expression is computed, and the value is then printed.

Testing

Testing for the interpreter was done by creating sample SmallLang programs and outputting their results.

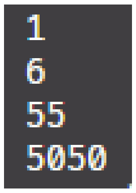
Test 1

The first test was a function which outputs the sum of numbers from 1 to n.



```
ff sum(n:int) : int{
    let ans:int = 0;
    for (let i: int = 1; i<=n; i=i+1){
        ans = ans + i;
    }
    return ans;
}

print sum(1);
print sum(3);
print sum(10);
print sum(100);
```

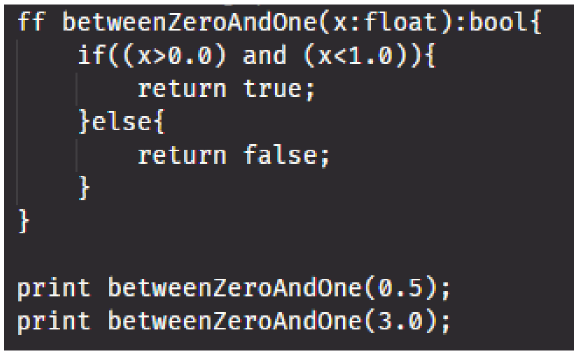
The output produced was:



```
1
6
55
5050
```

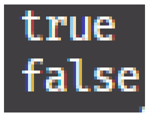
Test 2

The second test was a function which tests if a float is between 0 and 1.



```
ff betweenZeroAndOne(x:float):bool{
    if((x>0.0) and (x<1.0)){
        return true;
    }else{
        return false;
    }
}

print betweenZeroAndOne(0.5);
print betweenZeroAndOne(3.0);
```

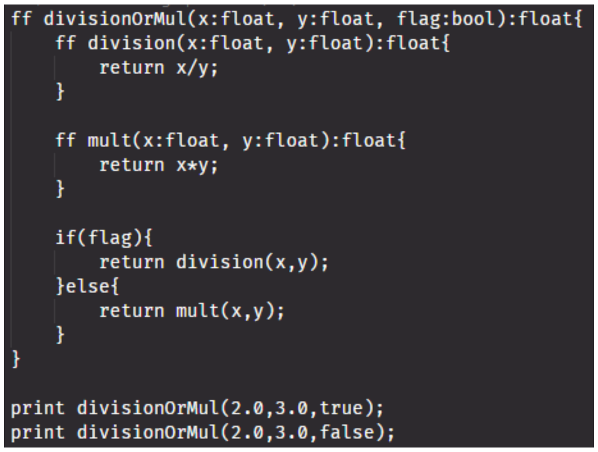
The output produced was:



```
true
false
```

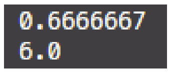
Test 3

The third test was a function which performs an operation based on a flag passed to function as a parameter. The aim of this test was to test the outputs produced when using functions in functions.



```
ff divisionOrMul(x:float, y:float, flag:bool):float{
    ff division(x:float, y:float):float{
        return x/y;
    }

    ff mult(x:float, y:float):float{
        return x*y;
    }

    if(flag){
        return division(x,y);
    }else{
        return mult(x,y);
    }
}

print divisionOrMul(2.0,3.0,true);
print divisionOrMul(2.0,3.0,false);
```

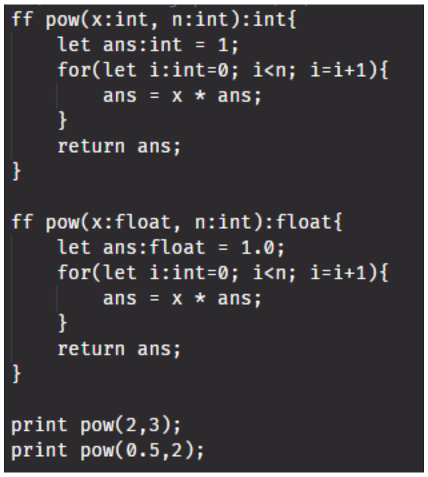
The output produced was:



```
0.6666667
6.0
```

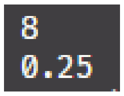
Test 4

The fourth test was a power function which supports both integers and floats using function overloading.



```
ff pow(x:int, n:int):int{
    let ans:int = 1;
    for(let i:int=0; i<n; i=i+1){
        ans = x * ans;
    }
    return ans;
}

ff pow(x:float, n:int):float{
    let ans:float = 1.0;
    for(let i:int=0; i<n; i=i+1){
        ans = x * ans;
    }
    return ans;
}

print pow(2,3);
print pow(0.5,2);
```

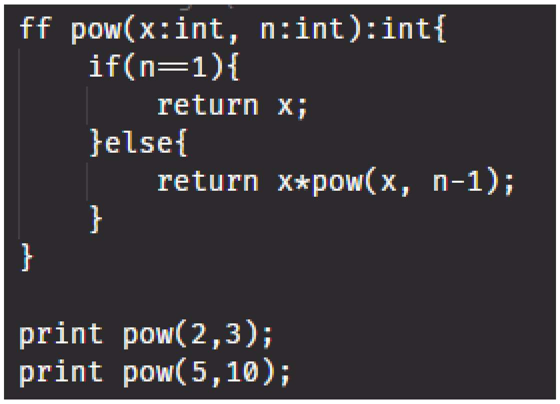
The output produced was:



```
8
0.25
```

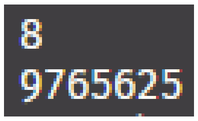
Test 5

The fifth test was the same power function however implemented using recursion.



```
ff pow(x:int, n:int):int{
    if(n==1){
        return x;
    }else{
        return x*pow(x, n-1);
    }
}

print pow(2,3);
print pow(5,10);
```

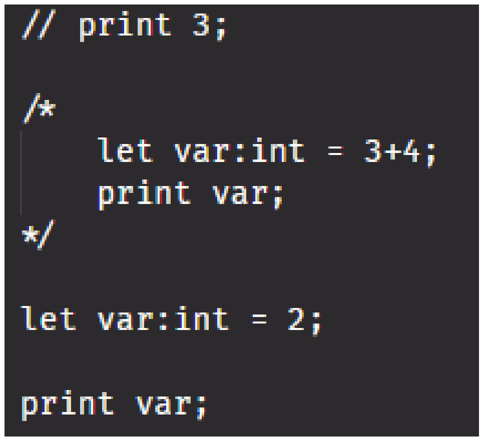
The output produced was:



```
8
9765625
```

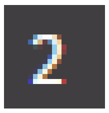
Test 6

The sixth test was a program with both inline and multiline comments.



```
// print 3;

/*
    let var:int = 3+4;
    print var;
*/

let var:int = 2;

print var;
```

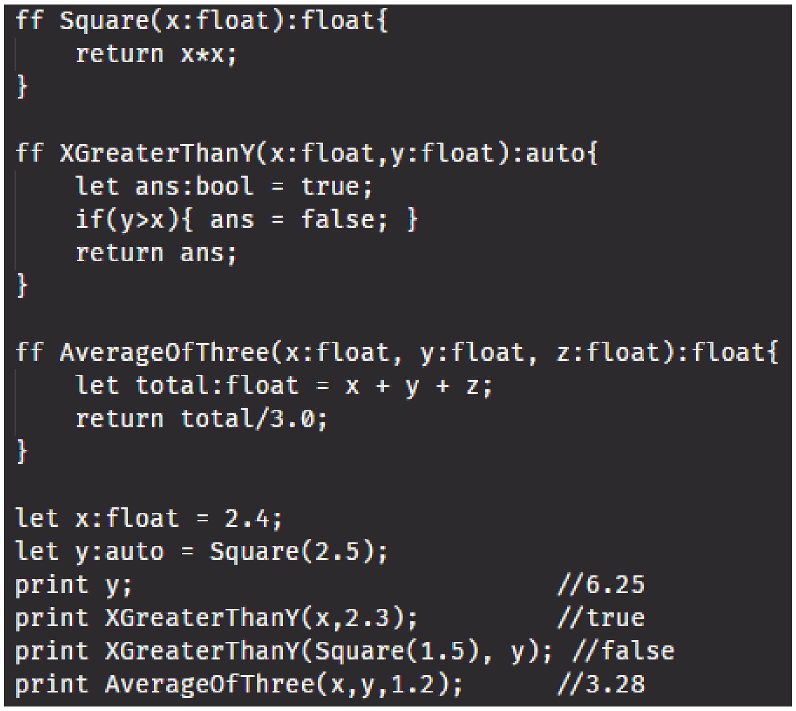
The output produced was:



```
2
```

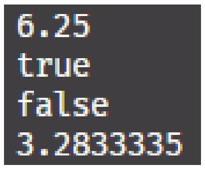
Test 7

Finally, the last test was the SmallLang program given in the specification.



```
ff Square(x:float):float{
    return x*x;
}

ff XGreaterThanY(x:float,y:float):auto{
    let ans:bool = true;
    if(y>x){ ans = false; }
    return ans;
}

ff AverageOfThree(x:float, y:float, z:float):float{
    let total:float = x + y + z;
    return total/3.0;
}

let x:float = 2.4;
let y:auto = Square(2.5);
print y;                            //6.25
print XGreaterThanY(x,2.3);         //true
print XGreaterThanY(Square(1.5), y); //false
print AverageOfThree(x,y,1.2);      //3.28
```

The output produced was:



```
6.25
true
false
3.2833335
```

Conclusion

All tasks have been implemented successfully and there are no known issues.

On the other hand, future improvements to the set of tasks include allowing the use of integers and floats in the same expressions to allow for more flexibility.

The above tasks may be run by building all the packages found within the “src” folder and by running it from the “Main.java” file.