CPS2000 - Compiler Theory & Practise

Assignment Part 1

B.Sc Computer Science

Jacques Vella Critien - 97500L

Contents

1 Task1: Table-driven lexer

For the first task, we were asked to develop a table-driven lexer for the SmallLang language by simulating the DFA transition function of the SmallLang micro-syntax, which should be able to report any lexical errors in the input program. This

⟨*Letter*⟩ ::= [A-Za-z]
⟨*Digit*⟩ ::= [0-9]
⟨*Type*⟩ ::= '`float`' | '`int`' | '`bool`'
⟨*Auto*⟩ ::= '`auto`'
⟨*BooleanLiteral*⟩ ::= '`true`' | '`false`'
⟨*IntegerLiteral*⟩ ::= ⟨*Digit*⟩ { ⟨*Digit*⟩ }
⟨*FloatLiteral*⟩ ::= ⟨*Digit*⟩ { ⟨*Digit*⟩ } '.' ⟨*Digit*⟩ { ⟨*Digit*⟩ }
⟨*Literal*⟩ ::= ⟨*BooleanLiteral*⟩
| ⟨*IntegerLiteral*⟩
| ⟨*FloatLiteral*⟩
⟨*Identifier*⟩ ::= ('_' | ⟨*Letter*⟩) { '_' | ⟨*Letter*⟩ | ⟨*Digit*⟩ }
⟨*MultiplicativeOp*⟩ ::= '`*`' | '`/`' | '`and`'
⟨*AdditiveOp*⟩ ::= '`+`' | '`-`' | '`or`'
⟨*RelationalOp*⟩ ::= '`<`' | '`>`' | '`==`' | '`<>`' | '`<=`' | '`>=`'
⟨*ActualParams*⟩ ::= ⟨*Expression*⟩ { '`,`' ⟨*Expression*⟩ }
⟨*FunctionCall*⟩ ::= ⟨*Identifier*⟩ '`(`' [⟨*ActualParams*⟩] '`)`'
⟨*SubExpression*⟩ ::= '`(`' ⟨*Expression*⟩ '`)`'
⟨*Unary*⟩ ::= ('`-`' | '`not`') ⟨*Expression*⟩
⟨*Factor*⟩ ::= ⟨*Literal*⟩
| ⟨*Identifier*⟩
| ⟨*FunctionCall*⟩
| ⟨*SubExpression*⟩
| ⟨*Unary*⟩
⟨*Term*⟩ ::= ⟨*Factor*⟩ { ⟨*MultiplicativeOp*⟩ ⟨*Factor*⟩ }
⟨*SimpleExpression*⟩ ::= ⟨*Term*⟩ { ⟨*AdditiveOp*⟩ ⟨*Term*⟩ }
⟨*Expression*⟩ ::= ⟨*SimpleExpression*⟩ { ⟨*RelationalOp*⟩ ⟨*SimpleExpression*⟩ }
⟨*Assignment*⟩ ::= ⟨*Identifier*⟩ '`=`' ⟨*Expression*⟩
⟨*VariableDecl*⟩ ::= '`let`' ⟨*Identifier*⟩ '`:`' (⟨*Type*⟩ | ⟨*Auto*⟩) '`=`' ⟨*Expression*⟩
⟨*PrintStatement*⟩ ::= '`print`' ⟨*Expression*⟩
⟨*RtrnStatement*⟩ ::= '`return`' ⟨*Expression*⟩
⟨*IfStatement*⟩ ::= '`if`' '`(`' ⟨*Expression*⟩ '`)`' ⟨*Block*⟩ ['`else`' ⟨*Block*⟩]
⟨*ForStatement*⟩ ::= '`for`' '`(`' [⟨*VariableDecl*⟩] '`;`' ⟨*Expression*⟩ '`;`' [⟨*Assignment*⟩] '`)`' ⟨*Block*⟩
⟨*WhileStatement*⟩ ::= '`while`' '`(`' ⟨*Expression*⟩ '`)`' ⟨*Block*⟩
⟨*FormalParam*⟩ ::= ⟨*Identifier*⟩ '`:`' ⟨*Type*⟩
⟨*FormalParams*⟩ ::= ⟨*FormalParam*⟩ { '`,`' ⟨*FormalParam*⟩ }
⟨*FunctionDecl*⟩ ::= '`ff`' ⟨*Identifier*⟩ '`(`' [⟨*FormalParams*⟩] '`)`' '`:`' (⟨*Type*⟩ | ⟨*Auto*⟩) ⟨*Block*⟩
⟨*Statement*⟩ ::= ⟨*VariableDecl*⟩ '`;`'
| ⟨*Assignment*⟩ '`;`'
| ⟨*PrintStatement*⟩ '`;`'
| ⟨*IfStatement*⟩
| ⟨*ForStatement*⟩
| ⟨*WhileStatement*⟩
| ⟨*RtrnStatement*⟩ '`;`'
| ⟨*FunctionDecl*⟩
| ⟨*Block*⟩
⟨*Block*⟩ ::= '`{`' { ⟨*Statement*⟩ } '`}`'
⟨*Program*⟩ ::= { ⟨*Statement*⟩ }

Figure 1: SmallLang micro-syntax

1.1 Deterministic finite automaton

I started off by designing a DFA bit by bit, starting from the most simple classifiers such as digits. I continued building onto it by creating transitions to all states, which represent possible tokens. The DFA generated can be seen in the figure below and this was used to help in determinign whether

an can be classified into a token or not. Then, from this automaton, three tables could be formed which are known as the Transition table, The Classifier table and the Token Type table.

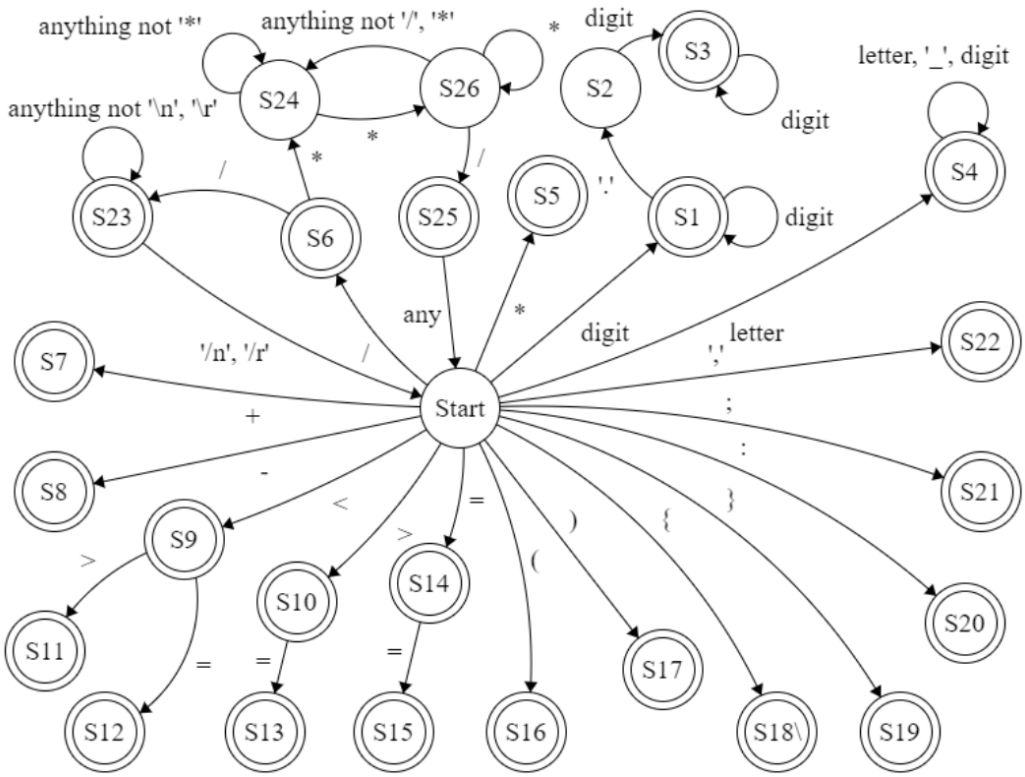


Figure 2: Deterministic finite automaton

1.2 Tables

1.2.1 Classifier Table

This table shows relates the specific characters of input to the classifiers. Classifiers can also be called categories and these are then used to match them to states to create transitions as you can see in the next section. This table can be seen in the figure below.

0-9	.	a-Z	_	*	/	+	-	<	>	=
DIGIT	DOT	LETTER	_	*	/	+	-	<	>	=

(	)	{	}	:	;	,	\n, \r	space	\uFFF	other
(	)	{	}	:	;	,	NEWLINE	SPACE	EOF	OTHER

Figure 3: Classifier Table

1.2.2 Type Token Table

This table shows how each state is linked to a classifier. When the state is not an accepted state, it is shown as a classifier of type invalid as you can see in the figure below.

START	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13
invalid	integer	invalid	float	identifier	*	/	+	-	<	>	<>	<=	>=

S14	S15	S16	S17	S18	S19	S20	S21	S22	S23	S24	S25	S26	S27
=	==	(	)	{	}	:	;	,	//	invalid	*/	invalid	EOF

Figure 4: Type Token Table

1.2.3 Transition Table

This table shows the transitions from a state to another state when given a classifier. These are based on the automaton found in the figure above.

	DIGIT	DOT	LETTER	_	*	/	+	-	<	>	=	(	)	{	}	:	;	,	NEWLINE	SPACE	EOF	OTHER
START	S1	BAD	S4	S4	S5	S6	S7	S8	S9	S10	S14	S16	S17	S18	S19	S20	S21	S22	START	START	S27	BAD
S1	S1	S2	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S2	S3	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S3	S3	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S4	S4	BAD	S4	S4	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S5	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S6	BAD	BAD	BAD	BAD	S24	S23	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S7	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S8	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S9	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	S11	S12	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S10	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	S13	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S11	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S12	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S13	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S14	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	S15	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S15	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S16	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S17	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S18	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S19	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S20	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S21	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S22	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S23	S23	S23	S23	S23	S23	S23	S23	S23	S23	S23	S23	S23	S23	S23	S23	S23	S23	S23	START	S23	S27	S23
S24	S24	S24	S24	S24	S26	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	BAD	S24
S25	S1	BAD	S4	S4	S5	S6	S7	S8	S9	S10	S14	S16	S17	S18	S19	S20	S21	S22	START	START	BAD	BAD
S26	S24	S24	S24	S24	S24	S25	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	S24	BAD	S26
S27	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD

Figure 5: Transition Table

1.3 Solution

1.3.1 TypeToken.java

This is an enum to hold the different types of tokens and it includes the following:

- INTEGER_LITERAL
- FLOAT_LITERAL
- BOOLEAN_LITERAL
- IDENTIFIER
- TYPE

- IDENTIFIER
- AUTO
- IF
- FOR
- WHILE
- LET
- PRINT
- RETURN
- FF
- MULTIPLICATIVE_OP
- ADDITIVE_OP
- RELATIONAL_OP
- EQUAL_SIGN
- BRACKET_OPEN
- BRACKET_CLOSE
- CURLY_OPEN
- CURLY_CLOSE
- SEMI_COLON
- COLON
- COMMA
- COMMENT_1LINE
- COMMENT_MULTI_CLOSE
- EOF
- NOT

1.3.2 Category.java

This is an enum to hold the different types of categories or classifiers:

- LETTER
- DIGIT
- DOT
- MULTIPLY_OPERAND
- DIVIDE_OPERAND

- ADDITION_OPERAND
- SUBTRACT_OPERAND
- EQUAL_SIGN
- GT_OPERAND
- ST_OPERAND
- BRACKET_OPEN
- BRACKET_CLOSE
- CURLY_OPEN
- CURLY_CLOSE
- SEMI_COLON
- COLON
- COMMA
- UNDERSCORE
- NEWLINE
- SPACE
- EOF
- OTHER

1.3.3 State.java

This is an enum to hold the different types of states.
The states are: **START, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, BAD, SE.**

1.3.4 Transition.java

This is a class which represents a transition, in fact, this class contains two members, a State holding the starting state of the transition and a Category which shows the path it needs to take. The values for these can easily be followed through from the DFA depicted in section 1.1. This class also has implementations of the overriden functions hashCode() and equals() to be able to compare these transitions later on.

1.3.5 Token.java

This is a class which represents a token and contains two members, **name (type)** and **attribute (lexeme)** which hold the token type and the identifier for that token. Apart from a constructor, this class contains the following methods.

- **getType()**: Getter for type
- **getAttribute()**: Getter for attribute
- **equals()**: Method to check if two tokens are equal

1.3.6 Keyword.java

This is a class which extends the Token class and in which all the keywords in the SmallLang syntax are declared. This contains **INT, FLOAT, BOOL, AUTO, IF, ELSE, FOR, WHILE, LET, RETURN, FF, PRINT, FALSE, TRUE, NOT**.

1.3.7 Operand.java

This is a class which extends the Token class and in which all the operands in the SmallLang syntax are declared. This contains:

- MULTIPLY (*)
- DIVIDE (/)
- SUBTRACT (-)
- ADDITION(+)
- AND_OP (and)
- OR_OP (op)
- GT (>)
- LT (<)
- GTE (>=)
- LTE (<=)
- EQUAL (==)
- NOTEQUAL (<>)

1.3.8 Lexer.java

This class contains the methods needed from the parser to obtain the next token. Firstly, this class contains these global variables which are used throughout the class:

- **keywords**: This is of type hashmap where the key is a string and the value is an object of type Token. This holds all the keywords and operands defined in the aforementioned name classes, Keyword and Operand.
- **stack**: This is a stack to hold states as the tokenization is happening.
- **acceptableStates**: This is of type hashmap where the key is a State and the value is an object of type TypeToken. This holds the acceptable states and their related tokentypes for example, S1 $\rightarrow$ INTEGER_LITERAL.
- **currentLine**: This is of type int and is used to hold the current line in which the lexer has arrived in order to make it easier when reporting back to the user in case of errors.

Moreover, this class contains the methods explained below:

1. **Lexer (constructor)**: This constructor takes a parameter **filename** which is a path from the resources. Inside this function, some operations are performed to open the file into a global buffered reader. Finally, this function calls the functions **setAcceptableStates()** and **setTransitionTable()** to initialise the lexer.

2. **setTransitionTable()**: This function populates the transition table hashmap explained above to have all the transitions given in the automaton and transition table found in parts 1.1 and 1.2.3 respectively.

3. **setAcceptableStates()**: This function populates the acceptable states hashmap explained above to have all the acceptable states as in the automaton and Type Token table.

4. **getCurrentLine()**: This function would return the current line at which the reader is at.

5. **rollback()**: This function rollbacks the reading from a file back to the marked position.

6. **charCat()**: This function uses if statments and a switch to return the ctaegory of a particular character. This is based on the Classifier Table in section 1.2.1.

7. **getTokens()**: This function returns an arraylist of tokens from a file input. This is used for testing.

8. **compareTokens()**: This functon accepst two arraylists of tokens and compares them. If they are equal, true is returned, otherwise false is returned. This is also used in testing

9. **nextToken()**: This method is called by the parser to return the next token and it uses all the aforementioned functions. This function was implemented and divided into 4 steps as follows:

    9.1. Initialisation: In this first step, the method sets the previous and current state to start, initialises a lexeme string, clears the stack and pushes the SE State into the stack.

    9.2. Scanning Loop: In this step, the method loops until the state is the bad one (SE). In this loop, a character is read, the current state is checked whether it is in the list of acceptable states and if so, the stack is cleared, the current state is pushed onto the stack and the character's category is found by calling **charCat()**. Then, there are some checks for comments, and if they match, the loop continues without adding to the lexeme. However, if it is not a comment, the character obtained is added to the lexeme.

    9.3. Rollback Loop: After the above step, there is another loop to go on until the state is acceptable. In this loop, the state is popped and the lexeme is trimmed, resulting in an exception if there is nothing more to trim. **Rollback()** is called before doing another round to move the character pointer in the file back.

    9.4. Result Reporting: After all the above, the method finally reports the result. However, it first checks whether the token type is an identifier because if so, a check for a keyword is made and in this case, the keyword's Token is returned. Otherwise, a new token is returned with the tokenType associated to the current state from the **acceptableStates** hashmap together with the lexeme as attribute.

1.4 Testing

In order to test my implementation of this table driven lexer, I prepared some files with SmallLang code snippets, in which I made sure to target all possible types of different tokens that can be returned by the lexer. Then I wrote unit tests, which can be found in the file name **'LexerTest.java'**. These tests get a list of tokens using the **getTokens()** function explained above and compares them to a hardcoded arraylist of expected tokens to be returned by the lexer using the **compareTokens()** function. As you can see in the image below, the files containing the source files to be tested are stored in the resources folder and are named according to the part of the SmallLang syntax to be tested.

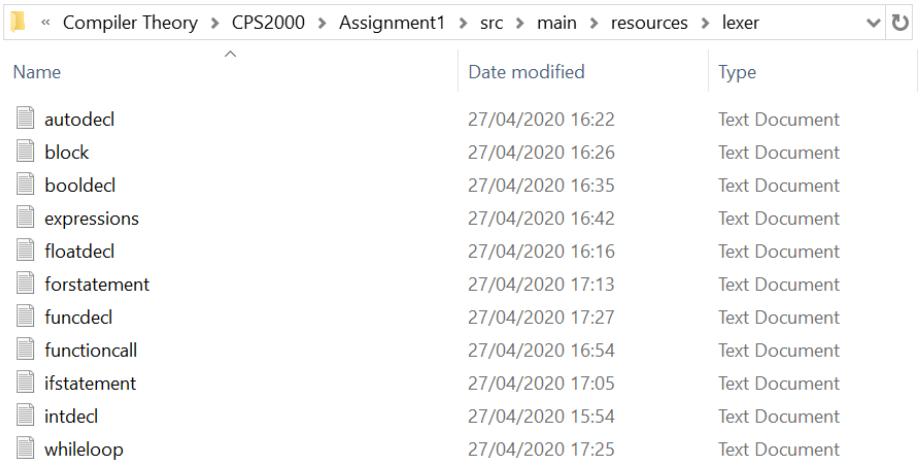


Figure 6: Structure for test source files

2 Task 2 - Hand-crafted LL parser

In this task, we were required to develop a hand-crafted predictive parser for the SmallLang language. This should interact with the aforementioned lexer through the function **nextToken()**. Moreover, this parser should be able to report any syntax errors back to the input program, while a successful parse produces an abstract syntax tree (AST) showing the structure of the input program. Furthermore, to help in the classes listed below, a **Type** enumeration with options **INT, FLOAT, BOOL** and **AUTO** was developed

2.1 AST Classes

In order to produce an abstract syntax tree, several classes were developed to be able to represent the input program in an AST. These classes can be fined in another package named **node** inside the **parser** package.

2.1.1 ASTNode.java

This is an interface with an accept method to be used later in tasks 3, 4 and 5. All the other classes implement this class.

2.1.2 ASTExpression.java

This is a class which implements the **ASTNode** interface. This is extended by other various classes as you can see further below in this section.

2.1.3 ASTStatement.java

Similarly to the class above, this is a class which implements the **ASTNode** interface and which is also extended by other classes

2.1.4 ASTProgram

This is a class which starts every program in the SmallLang language. In fact, this class only has 1 member, **statements**, which is an arraylist of objects of the type **ASTStatement**. Naturally, this includes a getter method for this member variable.

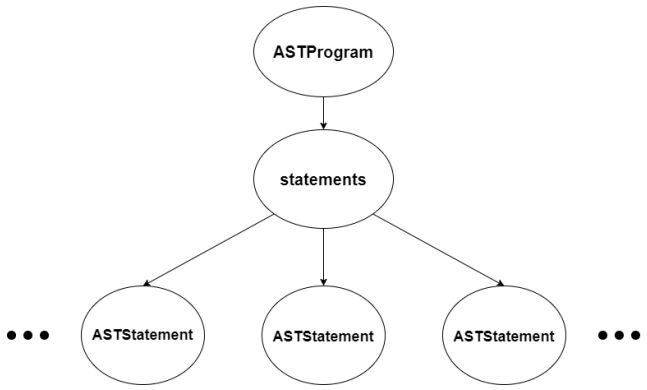


Figure 7: ASTProgram node

2.1.5 ASTIntegerLiteral

This is a class which extends the **ASTExpression** and represents an integer literal. It has one member which is **value**. Moreover, it has a constructor method and a getter for this value.

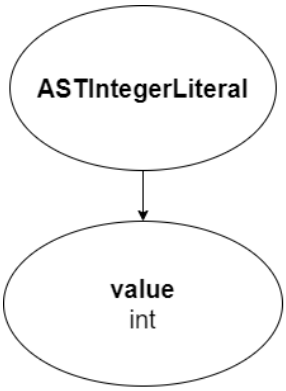


Figure 8: ASTIntegerLiteral node

2.1.6 ASTFloatLiteral

This is a class which extends the **ASTExpression** and represents a float literal. Similarly to above, it has one member which is **value**, a constructor method and a getter for this value.

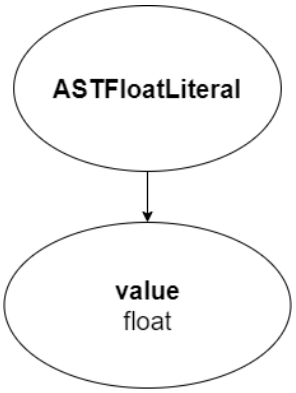


Figure 9: ASTFloatLiteral node

2.1.7 ASTBooleanLiteral

This is a class which extends the **ASTExpression** and represents a bool literal. Similarly to above, it has one member which is **value**, a constructor method and a getter for this value.

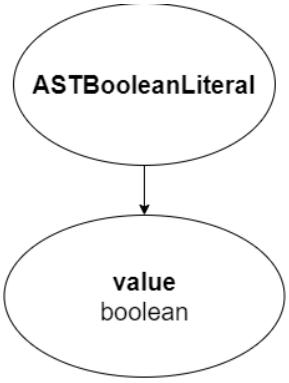


Figure 10: ASTBooleanLiteral node

2.1.8 ASTIdentifier

This is a class which extends the **ASTExpression** and represents an identifier. This class has the following 2 members:

1. **value**: Its type is String and it is used to hold the variable name
2. **type**: Its of type Type (enumeration) and it is used to hold the type of the identifier.

In addition, this has getters for each member and a setter for the type to be used in case the identifier is of type auto so that it could be set to the expression's type as I will be explaining later.

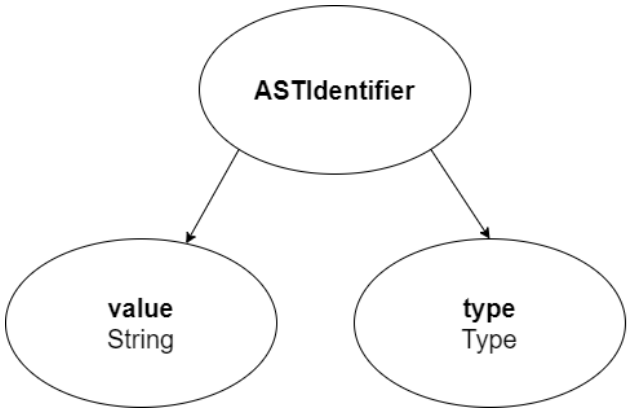


Figure 11: ASTIdentifier node

2.1.9 ASTBinExpression

Once again apart from extending **ASTExpression**, this is a class which represents a binary expression. In fact, this class has the following 3 members:

1. **left**: Its type is ASTExpression and it is used to hold the leftmost expression
2. **operand**: Its of type String and it is used to hold the operator.
3. **right**: Its type is ASTExpression and it is used to hold the rightmost expression

Similar to other classes, it also has getters for all members.

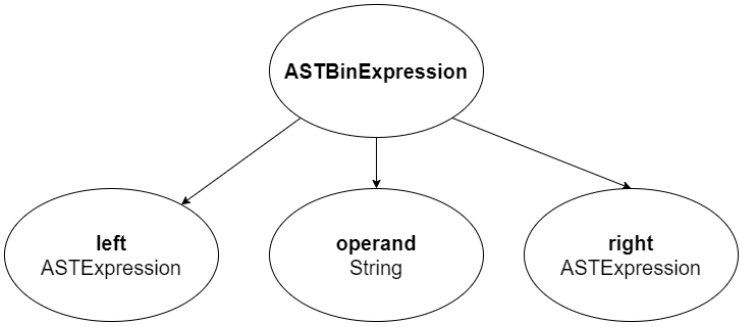


Figure 12: ASTBinExpression node

2.1.10 ASTUnary

This class extends **ASTExpression** and represents a unary operator '-' or 'not'. Moreover, this class has two members.

1. **lexeme**: Its type is String and it is used to hold whether it is a '-' or a 'not'
2. **expression**: Its type is ASTExpression and it is used to hold the expression after the unary operator

It also implements getters for these members.

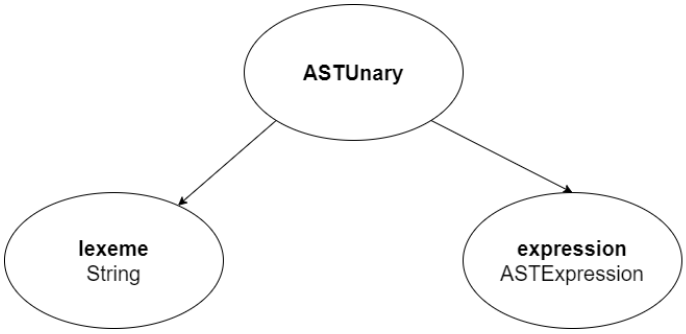


Figure 13: ASTUnary node

2.1.11 ASTActualParams

This is a class representing actual parameters, in fact it only consists of an arraylist of type **AST-Expression** as a member named **expressions** and its getter.

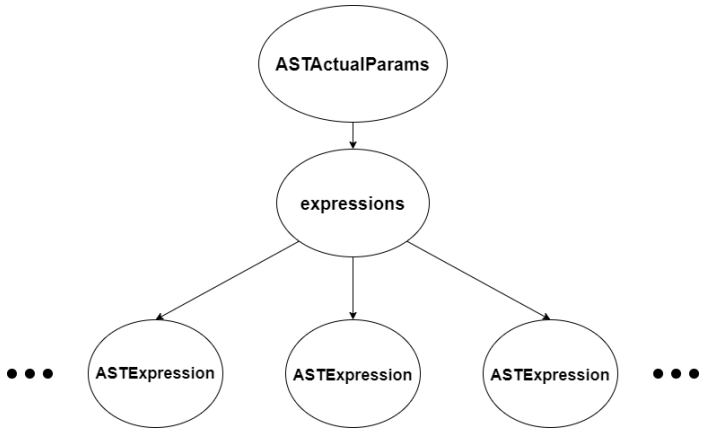


Figure 14: ASTActualParams node

2.1.12 ASTFunctionCall

This class also extends **ASTExpression** but represents a function call. This class has the following two members:

1. **identifier**: Its type is ASTIdentifier and it is used to hold the identifier of the function
2. **params**: Its type is ASTActualParams and it is used to hold the actual parameters passed

Just like in the other classes, we also have the getters for the members.

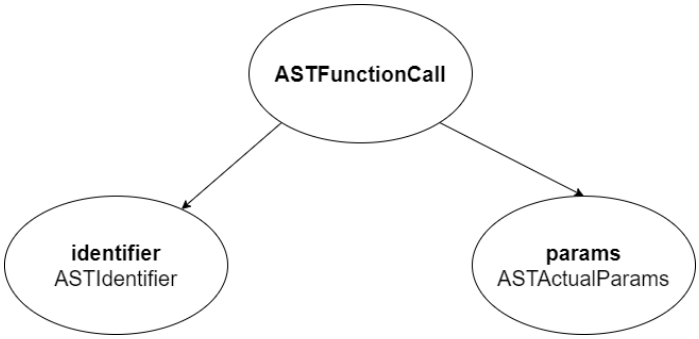


Figure 15: ASTFunctionCall node

2.1.13 ASTAssignment

This class extends **ASTStatement** and represents an assignment. This class has the following two members:

1. **identifier**: Its type is ASTIdentifier and it is used to hold the identifier of the variable being assigned a value
2. **expression**: Its type is ASTExpression and it is used to hold the expression for the assignment

Just like in the other classes, we also have the getters for the members.

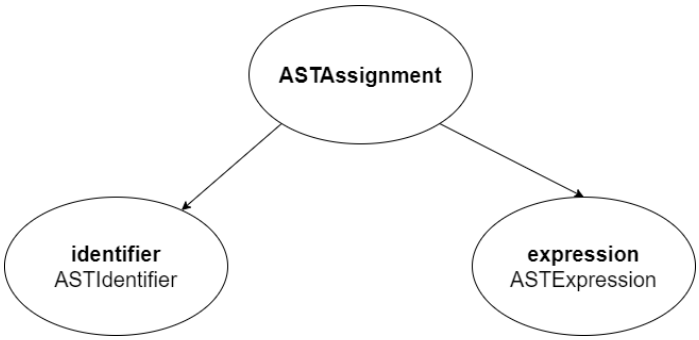


Figure 16: ASTAssignment node

2.1.14 ASTBlock

This class extends **ASTStatement** and represents a block. It only consists of a single member named **statements** which is an arraylist of type ASTStatement and its getter.

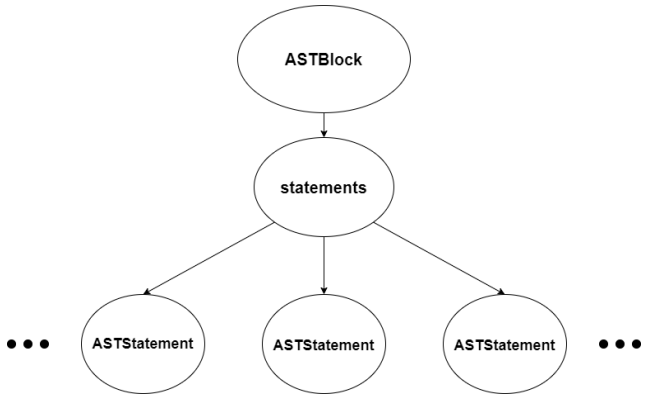


Figure 17: ASTBlock node

2.1.15 ASTVariableDecl

This class extends **ASTStatement** and represents a variable declaration. This class has the following two members:

1. **identifier**: Its type is ASTIdentifier and it is used to hold the identifier of the variable to declare
2. **expression**: Its type is ASTExpression and it is used to hold the expression for the declaration

Moreover, this class also includes the getters for these members.

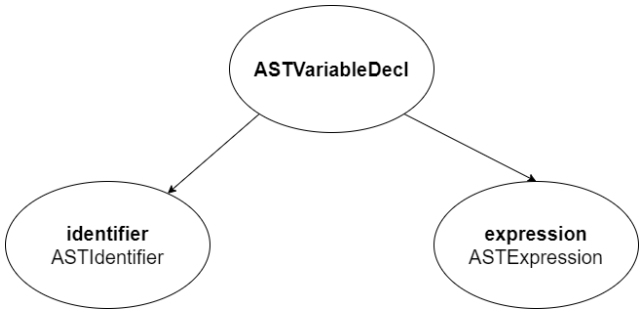


Figure 18: ASTVariableDecl node

2.1.16 ASTPrint

This class extends **ASTStatement** and represents a print statement. It consists of a member named **expression** of type ASTExpression which holds the expression to print. One can also find the getter for this expression.

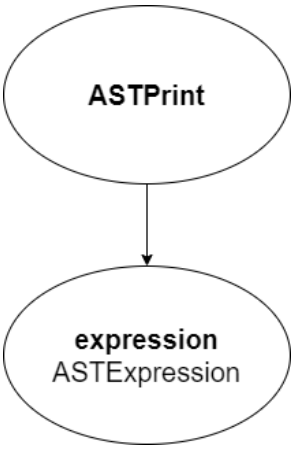


Figure 19: ASTPrint node

2.1.17 ASTReturn

This class extends **ASTStatement** and represents a return statement. It consists of a member named **expression** of type ASTExpression which holds the expression to return. One can also find the getter for this expression.

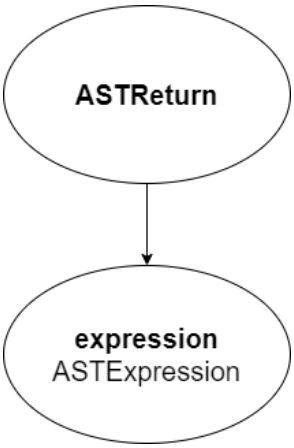


Figure 20: ASTReturn node

2.1.18 ASTIf

This class extends **ASTStatement** and represents a conditional statement. It consists of the following 3 members:

1. **expression**: Its type is ASTExpression and it is used to hold the expression to test as condition
2. **block**: Its type is ASTBlock and it is used to hold the block to execute in case the expression is true
3. **elseBlock**: Its type is ASTBlock and it is used to hold the block to execute in case the expression is false. This member can be null if there is no else condition.

This class also contains getters for these members.

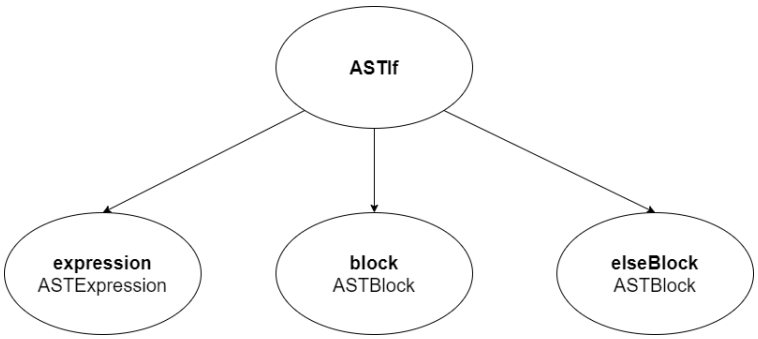


Figure 21: ASTIf node

2.1.19 ASTWhile

This class extends **ASTStatement** and represents a while loop. It consists of the following 2 members:

1. **expression**: Its type is ASTExpression and it is used to hold the expression to test as condition
2. **block**: Its type is ASTBlock and it is used to hold the block to execute in case the expression is truee

This class also contains getters for these members.

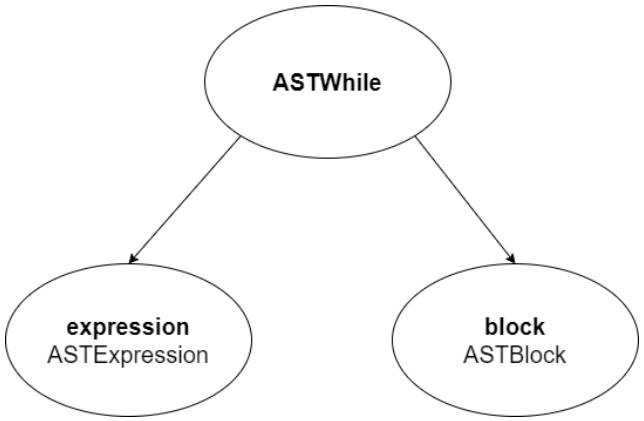


Figure 22: ASTWhile node

2.1.20 ASTFor

This class extends **ASTStatement** and represents a for loop. It consists of the following 4 members.

1. **variableDecl**: Its type is ASTVariableDecl and it is used to hold the variable declaration which is declared in the for loop. This can be left empty and it will be null
2. **expression**: Its type is ASTExpression and it is used to hold the expression to test as condition
3. **assignment**: Its type is ASTAssignment and it is used to hold the assignment which is declared in the for loop. This can be left empty and it will be null

4. **block**: Its type is ASTBlock and it is used to hold the block which is performed if the expression is true

This class also contains all the getters for these members.

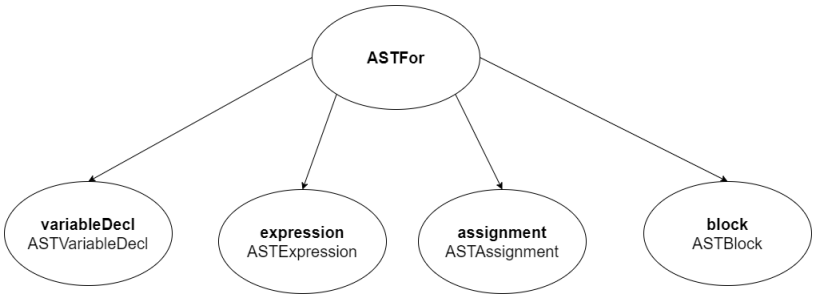


Figure 23: ASTFor node

2.1.21 ASTFormalParam

This class represents a formal parameter and contains only one member named **identifier** which is of type Identifier. It is an exact identifier but the type cannot be set to auto. This is checked in the constructor and if it is done, an IncorrectTypeException would be thrown. This class also has a getter for this member

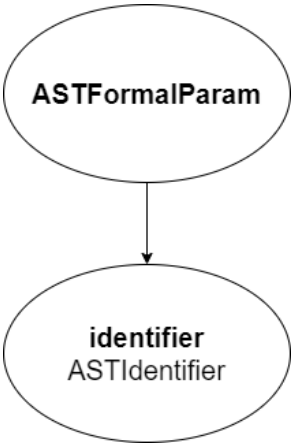


Figure 24: ASTFormalParam node

2.1.22 ASTFormalParams

This class represents formal parameters and contains only one member named **formalParams** which is an arraylist of ASTFormalParam. This is used by the ASTFunctionDecl class to hold its formal parameters. This class also has a getter for this member.

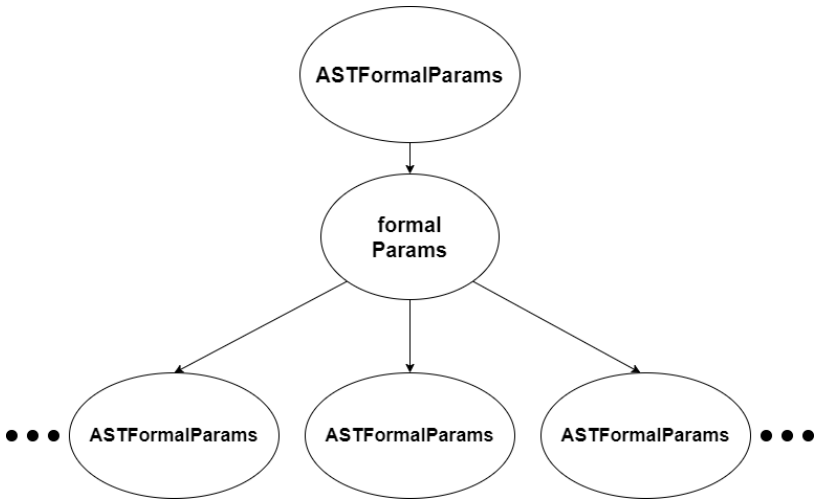


Figure 25: ASTFormalParams node

2.1.23 ASTFunctionDecl

This class extends **ASTStatement** and represents a function declaration. This function contains the following 3 members and its getters:

1. **identifier**: Its type is ASTIdentifier and it is used to hold the identifier for this new function
2. **formalParams**: Its type is ASTFormalParams and it is used to hold the formalParams for this function
3. **block**: Its type is ASTBlock and it is used to hold the statements which will be executed whenever this function is called

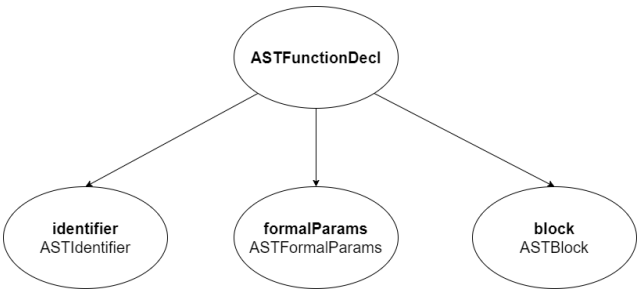


Figure 26: ASTFunctionDecl node

2.2 Solution

2.2.1 InvalidSyntaxException

This is an exception class which extends the Exception class and it is thrown when an invalid syntax is detected by the parser. The line is also passed with the message to help the user find the error in the input program.

2.2.2 Parser.java

This is the class in which one can find methods for parsing the input program. As explained before, this works by communicating with the Lexer to request a new token after it is ready from parsing the previous one.

This class has the following two member variables:

- **lexer**: This is an object of type Lexer which is used to link a lexer to the parser. This is used by the parser to request the next token through **nextToken()**
- **currentToken**: This is an object of type Token and is used to hold the current token in operation.

Below, I will be explaining each method and its function in relation to how the parser works. These methods implement the micro-syntax shown in Figure 1.

1. **Parser**: This is a constructor and it initialises the parser by setting the lexer and getting the next token from it.
2. **absorb(TypeToken)** : This is a method which checks whether the current token is of the correct type. The passed argument is the expected token and is called from the functions which I will be explaining below. If the token matches, it means syntax is valid and a new token is requested from the lexer, otherwise an InvalidSyntaxException is thrown.
3. **literal()**: This function checks the current token and according to its type which can be either an INTEGER_LITERAL, FLOAT_LITERAL or a BOOLEAN_LITERAL, it absorbs this token and then returns a new AST node according to the token type.
4. **identifier()**: This function is used to parse an identifier. It first absorbs a token of type IDENTIFIER and then returns a new ASTIdentifier node with the identifier name.
5. **type()**: This function is used to parse a type. It absorbs the token of type TYPE and then returns the token being absorbed.
6. **actualParams()**: This function is used to parse actual parameters. Since the syntax for a micro syntax is '<EXPRESSION> ',' <EXPRESSION>', I first initialise an array list of type ASTExpression to hold expressions and check if the next token is a ')', meaning that there are no parameters. If so, return an empty ASTActualParams object, otherwise call **expression()** to get the expression node, add it to the array list. As long as the next token type is a COMMA, I absorb the comma, get and store another expression and keep on doing so until the next token type is not a COMMA. Finally, an ASTActualParams node is returned with the collected expressions.
7. **functionCall(ASTIdentifier)**: This function is used to parse a function call and it takes an ASTIdentifier as a parameter. Since the syntax for a micro syntax is '<EXPRESSION>'(' [<ACTUALPARAMS>] ')', I first absorb an opening bracket token. Then, I check if the next token is a closing bracket, because if so, it means that there are no actual parameters, hence, and empty ASTActualParams node is stored. If not, it means that there are parameters and **actualParams()** is called to store them. Finally, the closing bracket is absorbed and a new ASTFunctionCall is returned with the identifier passed and the actual parameters collected.
8. **unary()**: This function is used to parse a unary expression. Since the syntax for a micro syntax is '<EXPRESSION>('-' , 'not') <EXPRESSION>, I first check whether the starting token is an ADDITIVE_OP ('-') or a NOT and absorb it. Then **expression** is called and a new ASTUnary is returned with the attribute of the ADDITIVE_OP ('-') or a NOT token and the expression.

9. **subExpression()**: This function is used to parse a sub expression. According to the micro-syntax, a sub expression takes the form of '(' <EXPRESSION>')'. Therefore, this function first absorbs an open bracket, then calls **expression()** and stores it and then absorbs the closing bracket. Finally the expression() of type ASTExpression is returned.

10. **factor()**: This function is used to parse a factor. According to the micro-syntax, a factor can either be a LITERAL, IDENTIFIER, FUNCTIONCALL, SUBEXPRESSION or a UNARY. Therefore, this function checks for the type of token and if the token is of type:

    - **INTEGER_LITERAL, FLOAT_LITERAL, BOOLEAN_LITERAL**: it calls **literal()** and returns it.
    - **IDENTIFIER, FUNCTIONCALL**: this can be either an actual identifier or a function call since both start with an identifier. To check which is which, I first call **identifier()** to absorb the identifier token and store the return in an ASTIdentifier object and then I check if the next token is an '('. If it is, it must be a function call, hence, I call **functionCall()** with the identifier and return it. Otherwise, it must be an identifier so I simply return the identifier stored.
    - **BRACKET_OPEN**: it calls and returns **subExpression()** because if the first token is an opening bracket it must be a sub-expression.
    - **NOT, SUBTRACT_OP**: if not any of the above, it calls and returns **unary()** because that is the only option left from a factor.

11. **term()**: This function is used to parse a term. According to the micro-syntax, a term takes the form of <FACTOR><MULTIPLICATIVE OP (*, and)> <FACTOR>. Therefore, this function calls **factor()** and stores it. Then there is a while loop which goes on as long as the current token is of type MULTIPLICATIVE_OP. In this loop the MULTIPLICATIVE_OP token is absorbed and the rightmost expression is obtained by calling **factor()**. The node initialised in the beginning is set to a new ASTBinExpression with the first expression as the current node, the operand taken from the MULTIPLICATE_OP token and the rightmost expression obtained. Finally the expression node is returned.

12. **simpleExpression()**: This function is used to parse a simple expression. According to the micro-syntax, a term takes the form of <TERM><ADDITIVEOP(+, -, 'or')> <TERM>. Therefore, this function calls **term()** and stores it. Then there is a while loop which goes on as long as the current token is of type ADDITIVE_OP. In this loop the ADDITIVE_OP token is absorbed and the rightmost expression is obtained by calling **term()**. The node initialised in the beginning is set to a new ASTBinExpression with the first expression as the current node, the operand taken from the ADDITIVE_OP token and the rightmost expression obtained. Finally the expression node is returned.

13. **expression()**: This function is used to parse a simple expression. According to the micro-syntax, a term takes the form of <SIMPLEEXPR><RELATIONALOP(¡, ¿, ==, ¡¿, ¡=, ¿=)> <SIMPLEEXPR>. Therefore, this function calls **simpleExpression()** and stores it. Then there is a while loop which goes on as long as the current token is of type RELATIONAL_OP. In this loop the RELATIONAL_OP token is absorbed and the rightmost expression is obtained by calling **simpleExpression()**. The node initialised in the beginning is set to a new ASTBinExpression with the first expression as the current node, the operand taken from the RELATIONAL_OP token and the rightmost expression obtained. Finally the expression node is returned.

14. **assignment()**: This function is used to parse an assignment. According to the micro-syntax, a term takes the form of <IDENTIFIER>'=' <EXPRESSION>. This function starts by

checking if the current token is an identifier, because in the case of a for loop without an assignment, an empty ASTAssignment node has to be returned. Otherwise, the identifier is obtained by calling **identifier()**, the EQUAL_SIGN token is absorbed and the expression is obtained by calling **expression()**. Finally, a new ASTAssignment object is returned with the identifier and expression obtained.

15. **variableDeclaration()**: This function is used to parse a variable declaration. According to the micro-syntax, a term takes the form of 'let' <IDENTIFIER>':' (<TYPE>— <AUTO>) '='<EXPRESSION>. This function starts by checking if the current token is an identifier, because in the case of a for loop without an assignment, an empty ASTVariableDecl node has to be returned. Otherwise, the LET token is absorbed, the identifier is obtained by calling **identifier()** and a COLON token is absorbed. Then, we check whether it is an TYPE or AUTO and absorb it and then we set the type of the identifier. The last two steps involve absorbing the EQUAL_SIGN token and getting the expression by calling **expression()** before returning an ASTVariableDecl object with the identifier and expression.

16. **formalParam()**: This function is used to parse a variable declaration. According to the micro-syntax, a term takes the form of <IDENTIFIER>':' <TYPE>. This function starts by obtaining an identifier by calling **identifier()**, absorbing a COLON and then obtaining the type by calling **type()**. Then, then we set the type of the identifier before returning an ASTFormalParam object with the identifier.

17. **formalParams()**: This function is used to parse formal parameters. Since the syntax for formal parameters is '<FORMALPARAM> ',' <FORMALPARAM>', I first initialise an array list of type ASTFormalParam to hold params and check if the next token is a ')', meaning that there are no parameters. If so, return an empty ASTFormalParams object, otherwise call **formalParam()** to get the parameter node, add it to the array list. As long as the next token type is a COMMA, I absorb the comma, get and store another parameter and keep on doing so until the next token type is not a COMMA. Finally, an ASTFormalParams node is returned with the collected expressions.

18. **functionDeclaration()**: This function is used to parse formal parameters. The syntax for a function declaration is 'ff"<IDENTIFIER>'(' [<FORMALPARAMS>] ')' ':' (<TYPE>— <AUTO>) '='<EXPRESSION>. I first absorb the FF token, then I obtain an identifier by calling **identifier()** and then absorb a '(' token. The next steps include obtaining formal parameters by calling 'formalParams()', absorbing a COLON token, absorbing a toke of type AUTO or TYPE and then set the identifier's type accordingly. Finally, I obtain the block by calling **block()** and return a new ASTFunctionDecl object with the identifier, formal parameters and block obtained.

19. **printStatement()**: This function is used to parse a print statement. The syntax for a print is 'print"<EXPRESSION>, therefore a PRINT token is absorbed and an expression is obtained by calling **expression()**. Finally a new ASTPrint object with the expression is returned.

20. **returnStatement()**: This function is used to parse a return statement. The syntax for a return is 'return"<EXPRESSION>, therefore a RETURN token is absorbed and an expression is obtained by calling **expression()**. Finally a new ASTREturn object with the expression is returned.

21. **returnStatement()**: This function is used to parse a return statement. The syntax for a return is 'return"<EXPRESSION>, therefore a RETURN token is absorbed and an expression is obtained by calling **expression()**. Finally a new ASTREturn object with the expression is returned.

22. **ifStatement()**: This function is used to parse an if statement. The syntax for a an if statement is 'if' '(' <EXPRESSION>')' <BLOCK>['else' <BLOCK>]. This is done by absorbing an IF token, then absorbing a '(' token, obtaining an expression by calling **expression()**, absorbing a ')' and obtaining a block by calling **block()**. The next step is checking if there is an ELSE token. If there is, the ELSE token is absorbed and another block is obtained by recalling **block()**, otherwise the else block is left as null. Finally an ASTIf object is returned with the expression and the two objects of ASTBlock.

23. **forStatement()**: This function is used to parse a for statement. The syntax for a for statement is 'for' '(' [<VARIABLEDECL>] ';' <EXPRESSION>';' [<ASSIGNMENT>] ')' <BLOCK>. To do this I first absorb a FOR token, then absorb a '(' token, then call **variableDeclaration()** to obtain a variable declaration, then absorb a SEMI_COLON, call **expression()** to obtain an expression, absorb another SEMI_COLON token and call **assignment()** to obtain an assignment. Finally a ')' token is absorbed and the block for the loop is obtained by calling **block()**. A new ASTFor object with variable declaration, expression, assignment and the block is returned

24. **whileStatement()**: This function is used to parse a while statement. The syntax for a while statement is 'while' '(' [<EXPRESSION>] ')' <BLOCK>. To do this I first absorb a WHILE token, then absorb a '(' token, then call **expression()** to obtain an expression, then absorb a ')' token and the block for the loop is obtained by calling **block()**. A new ASTWhile object with the expression and block is returned.

25. **block()**: This function is used to parse a block. The syntax for a block is '{' { <STATEMENT>} '}'. To do this I first absorb a '{' token and then initialise an arraylist of ASTStatement. Then there is a while loop which goes on until a '}' token is found. This loop gets a statement by calling **statement()** and adds it to the arraylist. Finally a '}' token is absorbed and an ASTBlock object is returned with the statements.

26. **statement()**: This function is used to parse a statement. According to the micro-syntax, a factor can either be a VARIABLEDECL ';', ASSIGNMENT ';', PRINTSTMNT ';', IFSTMNT, FORSTMNT, WHILESTMNT, RETURNSTMNT ';', FUNCTIONDECL, BLOCK. This is done by first intialising a toReturn variable of the type ASTStatement. Therefore, this function checks for the type of token and if the token is of type:

    - **LET**: it calls **variableDeclaration()**, sets it to the variable and absorbs a SEMI_COLON token.
    - **IDENTIFIER**: it calls **assignment()**, sets it to the variable and absorbs a SEMI_COLON token.
    - **PRINT**: it calls **printStatement()**, sets it to the variable and absorbs a SEMI_COLON token.
    - **IF**: it calls **ifStatement()** and sets it to the variable.
    - **FOR**: it calls **forStatement()** and sets it to the variable.
    - **WHILE**: it calls **whileStatement()** and sets it to the variable.
    - **RETURN**: it calls **returnStatement()**, sets it to the variable and absorbs a SEMI_COLON token.
    - **FF**: it calls **functionDeclaration()** and sets it to the variable.
    - **CURLY_OPEN**: it calls **block()** and sets it to the variable.
    - **EOF**: returns null;

After this, the toReturn variable is returned.

27. **program()**: This function is used to parse a program. The syntax for a program is { <STATEMENT>} . To do this I am declaring an arraylist of ASTStatement and get a statement by calling **statement()**. Then there is a while loop until statement is not null in which the statement is added to the arraylist and a new statement is obtained. finally a new ASTProgram object with the statements is returned.

28. **parse()**: This function starts the parsing by calling **program()**.

29. **getTypeEnum(String)**: This function returns the type in enum format when given in String.

2.3 Testing

The parser was tested by creating several integration tests to make sure that syntax errors are outputted when they should be. I also made sure to obtain 100% coverage so that all possible branches and cases are covered.

Such tests include the following

1.
```
//incorrect declaration
let x: ? int = 3;
```

2.
```
/* try */
let x: float = 1234.a;
```

3.
```
//no correct formal param
ff wjiw(d): int
{
}
```

4.
```
//no brackets
if 1==1
{
        print 1;
}
```

5.
```
// no semi colon
print 1
```

6.
```
//incorrect type
let var : wrong = 5;
```

7.
```
//while no bracket
while 1 == 1
{
        print 1;
}
```

8.
```
//no ending block
{
print 1;
```

9.
```
//incorrect assignment
x 2;
```

10.
```
//formal params auto
ff square(x: auto)
{
          return x*x;
}
```

11.
```
// Function definition for Power
ff Pow(x : float , n : int ) : auto
{
    let y : float = 1.0 ; // Declare y and set it to 1.0
    if( n>0 )
    {
        for (; n>0 ; n=n\minus1)
        {
            y = y*x; //Assignment y = y*x;
        }
    }
    else
    {
        for (; n<0 ; n=n+1)
        {
            y = y/x; //Assignment y = y/x;
        }
    }
    return y ; // return y as the result
}

//this throws an incorrect type exception since a \\function cannot be assigned a v
Pow =3;
```

12.
```
// Function definition for Power
ff Pow(x : float , n : int ) : auto
{
    let y : float = 1.0 ; // Declare y and set it to 1.0
    if( n>0 )
    {
        for (; n>0 ; n=n\minus1)
        {
            y = y*x; //Assignment y = y*x;
        }
    }
    else
    {
        for (; n<0 ; n=n+1)
        {
            y = y/x; //Assignment y = y/x;
        }
```

```
    }
    return y ; // return y as the result
}

//this throws an incorrect type exception since a function cannot be printed
print Pow;
```

These tests all gave out an InvalidSyntaxException as expected.

3 Task 3 - AST XML Generation Pass

For this task we had to implement a visitor class to output a properly indented XML representation of the abstract syntax tree returned by the parser. I started this task by creating an interface **Visitor.java** so that then I could extend it by a new class **VisitorXMLGenerator.java**. The visitor class contains void methods named visit for each AST class created and explained in section 2.1.

3.1 Solution

3.1.1 VisitorXMLGenerator.java

This class is used to visit each AST class and print its XML representation with indentation. To be able to do this, I am keeping a member variable of type int called indent which holds the current number of tabulations, which is incremented and decrement by specific visit methods as will be explained below.

The below are the methods implemented and their function:

1. **getIndent()**: This method is used to return a String of tabulations according to the value stored in the member variable named **indent**.

2. **visit(ASTActualParams)**: This method starts by getting the indentation as a string. Then, after incrementing indent, we get the ASTActualParams node's expressions and if they are empty we print the string specifying that its empty. Otherwise, the opening tag of actual params is printed, **accept()** is called for each expression and the closing tag of actual params is printed before decrementing the indent once again.

3. **visit(ASTAssignment)**: This method starts by getting the indentation as a string. Then, after incrementing indent, we get the identifier and expression of the ASTAssignment node. If the assignment is null, the assignment as an empty tag is printed. Otherwise, the opening tag of assignment is printed, **accept()** is called for the identifier and the expression and the closing tag of the assignment representation is printed before decrementing the indent once again.

4. **visit(ASTBinExpression)**: This method starts by getting the indentation as a string. Then, after incrementing indent, we get the operand, left and right expression of the ASTBinExpression node. The opening tag of expression is printed with the operand, **accept()** is called for the left and right expressions and the closing tag of the assignment representation is printed before decrementing the indent once again.

5. **visit(ASTBlock)**: This method starts by getting the indentation as a string. Then, after incrementing indent, we get the statements of the ASTBlock node. If the statements' size is 0, the assignment as an empty tag is printed. Otherwise, The opening tag of block is printed,

accept() is called for every statement and the closing tag of the block representation is printed before decrementing the indent once again.

6. **visit(ASTBooleanLiteral)**: This method starts by getting the indentation as a string. Then, the xml representation of the ASTBooleanLiteral is printed with the value.

7. **visit(ASTfloatLiteral)**: This method starts by getting the indentation as a string. Then, the xml representation of the ASTfloatLiteral is printed with the value.

8. **visit(ASTFor)**: This method starts by getting the indentation as a string. Then, after incrementing indent, we get the variable declaration, expression, assignment and block of the ASTFor node. The opening tag of the for loop is printed, **accept()** is called for the variable declaration, expression, assignment and block and the closing tag of the for representation is printed before decrementing the indent once again.

9. **visit(ASTFormalParam)**: This method starts by getting the indentation as a string. Then, after incrementing indent, we get the identifier of the ASTFormalParam node. The opening tag of the for loop is printed, **accept()** is called for the identifier and the closing tag of the for representation is printed before decrementing the indent once again.

10. **visit(ASTFormalParams)**: This method starts by getting the indentation as a string. Then, after incrementing indent, we get the formal parameters of the ASTFormalParams node. If the formal parameters' size is 0, the formal parameters as an empty tag is printed. Otherwise, The opening tag of formal parameters is printed, **accept()** is called for every formal parameter and the closing tag of the block representation is printed before decrementing the indent once again.

11. **visit(ASTFunctionCall)**: This method starts by getting the indentation as a string. Then, after incrementing indent, we get the identifier and parameters of the ASTFunctionCall node. The opening tag of the function call is printed, **accept()** is called for the identifier and parameters and the closing tag of the block representation is printed before decrementing the indent once again.

12. **visit(ASTFunctionDecl)**: This method starts by getting the indentation as a string. Then, after incrementing indent, we get the identifier, formal parameters and the block of the ASTFunctionDecl node. The opening tag of the function call is printed, **accept()** is called for the identifier, the formal parameters and the block and the closing tag of the block representation is printed before decrementing the indent once again.

13. **visit(ASTIdentifier)**: This method starts by getting the indentation as a string. Then, we get the type of the ASTIdentifier node. The opening tag of the function call is printed with the type together with the closing tag.

14. **visit(ASTIf)**: This method starts by getting the indentation as a string. Then, after incrementing indent, we get the expression, block and else block of the ASTIf node. The opening tag of the function call is printed, **accept()** is called for the expression, block and else block and the closing tag of the block representation is printed before decrementing the indent once again.

15. **visit(ASTIntegerLiteral)**: This method starts by getting the indentation as a string. Then, the xml representation of the ASTIntegerLiteral is printed with the value.

16. **visit(ASTPrint)**: This method starts by getting the indentation as a string. Then, after incrementing indent, we get the expression of the ASTPrint node. The opening tag of the function call is printed, **accept()** is called for the expression and the closing tag of the print representation is printed before decrementing the indent once again.

17. **visit(ASTProgram)**: This method starts by getting the indentation as a string. Then, after incrementing indent, we get the statements of the ASTProgram node. The opening tag of the function call is printed, **accept()** is called for every statement and the closing tag of the program representation is printed before decrementing the indent once again.

18. **visit(ASTReturn)**: This method starts by getting the indentation as a string. Then, after incrementing indent, we get the expression of the ASTReturn node. The opening tag of the function call is printed, **accept()** is called for the expression and the closing tag of the return representation is printed before decrementing the indent once again.

19. **visit(ASTUnary)**: This method starts by getting the indentation as a string. Then, after incrementing indent, I get the expression and lexeme of the ASTUnary node. The opening tag of the function call is printed with the type of unary, **accept()** is called for the expression and the closing tag of the unary representation is printed before decrementing the indent once again.

20. **visit(ASTVariableDecl)**: This method starts by getting the indentation as a string. Then, after incrementing indent, I get the identifier and expression of the ASTVariableDecl node. If the identifier is null, in case of a for loop with no declaration, an empty variable declaration tag is printed. Otherwise, The opening tag of the variable declaration is printed, **accept()** is called for the identifier and expression and the closing tag of the variable declaration representation is printed before decrementing the indent once again.

21. **visit(ASTWhile)**: This method starts by getting the indentation as a string. Then, after incrementing indent, we get the expression and block of the ASTWhile node. The opening tag of the function call is printed, **accept()** is called for the expression and block and the closing tag of the while representation is printed before decrementing the indent once again.

22. **generate()**: This function is the entry point to generate the XML representation of the input program and it directly visits the ASTProgram function.

3.1.2 Testing

In order to test my implementation of the XML representation generator, I created several tests to make sure to test all types of representations that the SmallLang language can produce. Then, I asserted that the output from this generator is as expected.

The tests conducted can be found below:

```
let i : int = 0;
let b : bool = true;

/*
Expected

<Program>
    <VarDecl>
        <Identifier Type="INT">i</Identifier>
        <IntegerLiteral>0</IntegerLiteral>
    </VarDecl>
    <VarDecl>
        <Identifier Type="BOOL">b</Identifier>
        <BooleanLiteral>true</BooleanLiteral>
    </VarDecl>
</Program>

*/
```

Figure 27: xmltest1.txt

```
print 1;

/*
Expected

<Program>
    <Print>
        <IntegerLiteral>1</IntegerLiteral>
    </Print>
</Program>
*/
```

Figure 28: xmltest2.txt

```
//for loop no assignment and declaration
let x : int =9;
for(; x < 10;){ print x; x = x+1; }

/* Expected
<Program>
    <VarDecl>
        <Identifier Type="INT">x</Identifier>
        <IntegerLiteral>9</IntegerLiteral>
    </VarDecl>
    <For>
        <VarDecl>Empty</VarDecl>
        <BinaryExpr Op="<">
            <Identifier>x</Identifier>
            <IntegerLiteral>10</IntegerLiteral>
        </BinaryExpr>
        <Assignment>Empty</Assignment>
        <Block>
            <Print>
                <Identifier>x</Identifier>
            </Print>
            <Assignment>
                <Identifier>x</Identifier>
                <BinaryExpr Op="+">
                    <Identifier>x</Identifier>
                    <IntegerLiteral>1</IntegerLiteral>
                </BinaryExpr>
            </Assignment>
        </Block>
    </For>
</Program>
*/
```

Figure 29: xmltest9.txt

```
if(1 == 1)
{
    print 1;
}
else
{
    print 0;
}

/*Expected
<Program>
    <If>
        <BinaryExpr Op="==">
            <IntegerLiteral>1</IntegerLiteral>
            <IntegerLiteral>1</IntegerLiteral>
        </BinaryExpr>
        <Block>
            <Print>
                <IntegerLiteral>1</IntegerLiteral>
            </Print>
        </Block>
        <Block>
            <Print>
                <IntegerLiteral>0</IntegerLiteral>
            </Print>
        </Block>
    </If>
</Program>
```

Figure 30: xmltest4.txt

```
//empty block
if(1 == 1)
{
}
else
{
}

/*
Expected

<Program>
    <If>
        <BinaryExpr Op="==">
            <IntegerLiteral>1</IntegerLiteral>
            <IntegerLiteral>1</IntegerLiteral>
        </BinaryExpr>
        <Block>Empty</Block>
        <Block>Empty</Block>
    </If>
</Program>
*/
```

Figure 31: xmltest5.txt

```
while(1 < 3)
{
print 1;
}

/*
expected

<Program>
    <While>
        <BinaryExpr Op="<">
            <IntegerLiteral>1</IntegerLiteral>
            <IntegerLiteral>3</IntegerLiteral>
        </BinaryExpr>
        <Block>
            <Print>
                <IntegerLiteral>1</IntegerLiteral>
            </Print>
        </Block>
    </While>
</Program>
*/
```

Figure 32: xmltest6.txt

```
//function with formal params
ff square(x: int): int
{
    return x*x;
}

/*
expected

<Program>
    <FuncDecl>
        <Identifier Type="INT">square</Identifier>
        <FormalParams>
            <FormalParam>
                <Identifier Type="INT">x</Identifier>
            </FormalParam>
        </FormalParams>
        <Block>
            <Return>
                <BinaryExpr Op="*">
                    <Identifier>x</Identifier>
                    <Identifier>x</Identifier>
                </BinaryExpr>
            </Return>
        </Block>
    </FuncDecl>
</Program>
*/
```

Figure 33: xmltest7.txt

```
//for loop
for(let x:int =0; x < 10; x = x+1)
{ print x;}
/*Expected
<Program>
    <For>
        <VarDecl>
            <Identifier Type="INT">x</Identifier>
            <IntegerLiteral>0</IntegerLiteral>
        </VarDecl>
        <BinaryExpr Op="<">
            <Identifier>x</Identifier>
            <IntegerLiteral>10</IntegerLiteral>
        </BinaryExpr>
        <Assignment>
            <Identifier>x</Identifier>
            <BinaryExpr Op="+">
                <Identifier>x</Identifier>
                <IntegerLiteral>1</IntegerLiteral>
            </BinaryExpr>
        </Assignment>
        <Block>
            <Print>
                <Identifier>x</Identifier>
            </Print>
        </Block>
    </For>
</Program>
```

Figure 34: xmltest8.txt

```
let x:float = 3.2;
x = 5.0;

/*
Expected

<Program>
    <VarDecl>
        <Identifier Type="FLOAT">x</Identifier>
        <FloatLiteral>3.2</FloatLiteral>
    </VarDecl>
    <Assignment>
        <Identifier>x</Identifier>
        <FloatLiteral>5.0</FloatLiteral>
    </Assignment>
</Program>
*/
```

Figure 35: xmltest3.txt

```
//function with no formal params
ff square(): int
{
    return 1;
}

/*
Expected

<Program>
    <FuncDecl>
        <Identifier Type="INT">square</Identifier>
        <FormalParams>Empty</FormalParams>
        <Block>
            <Return>
                <IntegerLiteral>1</IntegerLiteral>
            </Return>
        </Block>
    </FuncDecl>
</Program>
*/
```

Figure 36: xmltest10.txt

```
//function call
let x: int = square(1);

/*
Expected

<Program>
    <VarDecl>
        <Identifier Type="INT">x</Identifier>
        <FunctionCall>
            <Identifier>square</Identifier>
            <ActualParams>
                <IntegerLiteral>1</IntegerLiteral>
            </ActualParams>
        </FunctionCall>
    </VarDecl>
</Program>
*/
```

Figure 37: xmltest11.txt

```
//function call no params
let x: int = func();

/*
Expected

<Program>
    <VarDecl>
        <Identifier Type="INT">x</Identifier>
        <FunctionCall>
            <Identifier>func</Identifier>
            <ActualParams>Empty</ActualParams>
        </FunctionCall>
    </VarDecl>
</Program>
*/
```

Figure 38: xmltest12.txt

```
//variable declaration with unary
let x: int = -2;

/*
Expected

<Program>
    <VarDecl>
        <Identifier Type="INT">x</Identifier>
        <Unary Type="-">
            <IntegerLiteral>2</IntegerLiteral>
        </Unary>
    </VarDecl>
</Program>
*/
```

Figure 39: xmltest13.txt

4 Task 4 - Semantic Analysis Pass

In this task, another visitor class was needed to traverse the AST and perform type-checking to make sure variables are only declared once and variables are assigned to appropriate types. Moreover, a

proper implementation of a symbol table was required in order to handle scopes. Finally, this semantic analyser should be able to assign types to auto variables.

4.1 Solution

4.1.1 Scope.java

This is a class which represents a scope. This class has the following two member variables:

- **declarations**: This is a hashmap with key of type String, and the value of type ASTNode. This is used to hold declarations in the scope.
- **values**: This is a hashmap with key of type String, and the value of type Object. This is used to hold values for task 5.

This class contains these methods

- **Scope**: This is a simple constructor to initialise the two hashmaps
- **addDeclaration(String, ASTNode)**: This is a method to add a declaration to the declarations
- **removeDeclaration(String)**: This is a method to remove a declaration from the declarations
- **addValue(String, Object)**: This is a method to add a value to the values
- **removeValue(String)**: This is a method to remove a value from the values
- **getDeclarations()**: This is a function to get all the declarations of the scope
- **getValues()**: This is a function to get all the values of the scope
- **isDefined(String)**: This is a function to check if a declaration is defined.

4.1.2 SymbolTable.java

This is a singleton class which represents a symbol table. This class has the following four member variables:

- **symbolTable**: This is a private static variable of type SymbolTable used just to make the class singleton.
- **scopes**: This is a Stack of Scope objects which is used to hold scopes.
- **constantValue**: This is a variable of type Object used to store the value from expressions. This is to be used in task 5.
- **constantType**: This is a variable of type Type and is used to store the constant type for expressions to be able to compare the type from one visitor method to another.

This class contains these methods

- **SymbolTable**: This is a simple private constructor to initialise the singleton class
- **getSymbolTable()**: This is a method to obtain the singleton instance of the symbol table
- **getScopes()**: This is a function to return the stack of scopes
- **getCurrentScope()**: This is a method to obtain the current scope

- **popScope()**: This is a function to pop a scope from the stack
- **insertScope(Scope)**: This is a function to insert a scope at the top of the stack
- **insertDecl(String, ASTNode)**: This is a function to insert a declaration to the current scope. If this identifier is already defined an **AlreadyDeclaredException** will be thrown.
- **insertValue(String, Object)**: This is a function to insert a value in the firsts cope where it is declared.
- **insertDeclGlobal(String, ASTNode)**: This is a function to insert a declaration in the global scope.
- **lookup(String)**: This is a function to search for an identifier through the loops and return it if found.
- **getValue(String)**: This is a function to obtain a value of an identifier where it is found first
- **getGlobalScope()**: This is a function to obtain the global scope
- **getConstantValue()**: This is a function to the value of the expression
- **setConstantValue(Object)**: This is a function to set the value of the expression
- **getConstant()**: This is a function to obtain the type of the expression
- **setConstant(Type)**: This is a function to set the type of the expression
- **reset()**: This is a function to reset the symbol table for testing purposes

4.1.3 VisitorSemanticAnalysis.java

This class implements the Visitor interface and is used to visit every AST class and perform semantic analysis and throwing meaningful exceptions where needed. This class contains the following global variables:

- **symbolTable**: This is a variable to hold the symbol table.
- **functionIdentifier**: This is a variable of type ASTIdentifier to hold the current identifier for a function declaration to check the return type as I will explain further below.

All the visit methods implementations can be found below:

1. **visit(ASTActualParams)**: This method performs semantic analysis on actual parameters. This is done by getting the expressions from the ASTActualParams node and then visit each expression.
2. **visit(ASTAssignment)**:This method performs semantic analysis on an assignment statement. First, we check if the expression in the node is null in case of a for loop with no assignment and if so, this is skipped. Otherwise, the expression and identifier are obtained from the node. The next step entails looking up the identifier and if this is not found in the symbol table, an **UndeclaredException** is thrown. The expression is then visited, where the expression's result type is stored in the global. After getting actual identifier's type, we check it across the type of the expression and if these do not match, an **IncorrectTypeException** is thrown. Finally the global containing the type of the expression is set to null.

3. **visit(ASTBinExpression)**: This method performs semantic analysis on a binary expression node. To do this, the left and right expressions are obtained from the node. Then, the left expression is visited and the type of this expression is stored and the same is done afterwards, for the right expression. After this, the two stored types are checked and if they do not match, an **IncorrectTypeException** is thrown. The next steps involve getting the operand and operating a switch statement on it. Here, all types of operands are checked and it is made sure that they are being used with the correct types, if not, an **IncorrectTypeException** is thrown. Finally, the type for the binary expression return is set.

4. **visit(ASTBlock)**: This method performs semantic analysis on a block. This function starts by inserting a new scope and setting a variable named **alreadyReturn**, which is used to hold a flag whether there was already a return statement, to false. After this, the block's statements are obtained for every statement and it is checked if the alreadyReturn flag is set, if so a warning is outputted. Moreover, each statement is visited and if the statement is a return statement, alreayReturn is set to true. Finally, the scope is popped.

5. **visit(ASTBooleanLiteral)**: This method just sets the expression's constant variable type to BOOL.

6. **visit(ASTfloatLiteral)**: This method just sets the expression's constant variable type to FLOAT.

7. **visit(ASTFor)**: This method performs semantic analysis on a for statement. This is done by first getting the variable declaration from the node and visiting it, getting the expression and visiting it, getting the assignment and visiting it and finally getting and visiting the block from the ASTFor node.

8. **visit(ASTFormalParam)**: This method handles a formal parameter. This is done by getting the identifier from the ASTFormalParam node and inserting it to the declarations.

9. **visit(ASTFormalParams)**: This method handles formal parameters by getting all the formal parameters from the node and visiting them one by one.

10. **visit(ASTFunctionCall)**: This method performs semantic analysis on a function call. This is done by first getting the identifier and the actual identifier of the function being called by the **lookup()** method. If the actual identifier of the function is null, meaning it was not found, an **UndeclaredException** is thrown. Then, the expressions from the ASTFunctionCall node are obtained together with the formal parameters from the function's actual identifier. The first check is done on the size of the actual parameters and the formal parameters, and if these are not equal, an **IncorrectTypeException** is thrown. After this, for each expression in the actual parameters object, the expression is visited (where the expression type is stored in the global variable), the corresponding formal parameter's type is obtained and if the expression type stored in the global and the formal parameter's type do not match, an **IncorrectTypeException** is thrown. Finally, the global variable holding the constant type is set to the type stated in the actual function's identifier declaration.

11. **visit(ASTFunctionDecl)**: This method performs semantic analysis on a function declaration. This starts by removing any declarations named 'return' just in case they were set before. The next step is to create a new scope to hold the actual parameters of the function declaration. The identifier of the function is obtained and stored in a variable and the formal parameters are obtained and visited. What follows is getting the block from the ASTFunctionDecl node and visiting it. If after visiting the block, there is no declaration named 'return' in the symbol table, it must mean that there was no return statement and if so a **ReturnTypeMismatchException** is thrown. After this, the function declaration scope is popped

and a function's identifier is inserted.

As a point of discussion, this implementation does cover the feature to be be able to define nested function declarations, meaning other declarations inside blocks. This was only done to an extent as the lecturer advised that for the sake of this assignment, these type of programs can be used to be inexistant. This implementation does not allow a function declared inside another block to be called from outside that block because the implementation does not entail in the functions being declared in the global scope. This was not done because with the current implementation, this would pass the semantic analysis but would fail in the interpreter because as will be explained further down, the interpreter visitor method for the function declaration does not go into the block, hence, the nested function declaration is not added to the declarations, meaning when you try to call the nested function from outside the scope in which it was declared, it would not be found.

12. **visit(ASTIdentifier)**: This method performs semantic analysis on an identifier by first getting the value of the identifier. This identifier is searched in the symbol table by the **lookup()** method and if it is not found, an **UndeclaredException** is thrown. Otherwise the global variable named **constantType** which holds the type for the constant expression is set to the identifier's type/

13. **visit(ASTIf)**: This method performs semantic analysis on an if statement. This starts by getting the expression form the ASTIf node and visiting it. After this the true block and the else block are obtained and visited one by one.

14. **visit(ASTIntegerLiteral)**:This method handles formal parameters by getting all the formal parameters from the node and visiting them one by one.

15. **visit(ASTPrint)**: This method performs semantic analysis on a print statement by first getting the expression and visiting it. Finally, the constant type variable is set to null

16. **visit(ASTProgram)**: This method performs semantic analysis on a program node. This is done by creating a new scope and after getting all the statements from the node, visit them one by one. Finally, the scope is popped.

17. **visit(ASTReturn)**:This method performs semantic analysis on a return statement by first getting the expression and visiting it. Finally, the constant type returned from the expression is stored and checked by calling **checkReturnType()**. If this function does not throw any exceptions, the return type is stored as a declaration so that it could be used by the parent function declaration function.

18. **visit(ASTUnary)**: This method performs semantic analysis on a unary expression by first getting the expression and visiting it.

19. **visit(ASTVariableDecl)**: This method performs semantic analysis on a variable declaration. First, it is checked whether the expression is empty, as can happen in for loops with no variable declaration. If it is not null, the expression is obtained and visited and the identifier and the type of the identifier are obtained. If the type is auto, the type of the identifier is set to the constant type set when visiting the expression, otherwise, the type of the identifier is checked against the constant type. If these do not match, an **IncorrectTypeException** is thrown. Finally, the identifier is inserted in the scope, it is visited and the constant type is set to null.

20. **visit(ASTWhile)**: his method performs semantic analysis on a while loop statement. This is done by first getting the expression and visiting it and finally, getting the block and visiting it.

21. **analyse()**: This function is the entry point to semantically analyse the input program and it directly visits the ASTProgram function.

22. **checkReturnType(Type)**: This function is used by return visitor method to check the return type with the function's type. This is done by getting the type of the function identifier stored in the global variable name **functionIdentifier**. If the function identifier's type is auto, the function identifier's type is set to the type passed, otherwise, the function identifier's type is checked against the type passed and if they do not match, a **ReturnTypeMismatchException** is thrown.

4.1.4 Exceptions

These are the different types of exceptions thrown by the semantic analyser

1. **AlreadyDeclaredException.java**: This is an exception class thrown when a variable or a function declaration is trying to be inserted but it is already declared.

2. **UndeclaredException.java**: This is an exception class thrown when an identifier is being used but it is not yet declared.

3. **ReturnTypeMismatchException.java**: This is an exception class thrown when a return type of a function does not match with the function's identifier type.

4. **IncorrectTypeException.java**: This is an exception class thrown when actual parameters are not passed with the correct types, when trying to assign a variable to a wrong type or when assigning a bad type expression to an identifier on declaration.

4.2 Testing

Apart from the integration tests, the following tests were created to hit all the possible semantic errors that an input program can have.

1. //incorrect type exception - expected IncorrectTypeException
   let x:int = 3.2;

2. //undeclared exception - expected UndeclaredException
   let x:int = y;

3. let x:int = 1;
   //already declared - expected AlreadyDeclaredException
   let x:int =2;

4. //function with bad return type - expected ReturnTypeMismatchException
   ff bad_return_type():int
   {
   return 0.1;
   }

5. //bad assignment - expected UndeclaredException
   x = 1;

6. let x : int =1;
   //bad assignment - expected IncorrectTypeException
   x = 1.2;

7. let x : float =1.2;
   //bad assignment - expected IncorrectTypeException
   x = 1;

8. let x : bool =false;
   //bad assignment - expected IncorrectTypeException
   x = 1;

9. //expression of not same types - expected IncorrectTypeException
   let x: int = 1 + 1.2;

10. //bad expression - expected IncorrectTypeException
    let x: int = 1 and 2;

11. //bad expression - expected IncorrectTypeException
    let x: bool = true + false;

12. //function call with inexistent function - expected UndeclaredException
    let x: bool = bad_func();

13. ff plus1(x:int):int
    {
        return x+1;
    }

    //bad function call - expected IncorrectTypeException
    let x: int = plus1(1,2);

14. ff plus1(x:float):float
    {
        return x+1;

    }

    //bad function call - expected IncorrectTypeException
    let x: int = plus1(1,2);

15. ff andgate(x:bool):bool
    {
        return x and true;
    }

    //bad function call - expected IncorrectTypeException
    let x: bool = andgate(1);

16. //function no return - expected ReturnTypeMismatchException
    ff andgate(x:bool):bool
    {}

17. //bad variable decl float - expected IncorrectTypeException
    let x: float = 1;

18. //bad variable decl bool - expected IncorrectTypeException
    let x: bool = 1;

19. //bad expression - expected IncorrectTypeException
    let x: bool = true ¿ true;

20. ff square(x:int):int
    {
        return x*x;
    }

    //already declared - expected AlreadyDeclaredException
    ff square(x:int):int
    {
        return x*x;
    }

5 Task 5 - Interpreter Execution Pass

The last task for this assignment required another visitor class to traverse the AST and simulate an interpreter, executing the test program. The Scope class was modified at this point to also have a hashmap of values, while the symbol table was changed to have a constant value member variable apart from the aforementioned constant type member value. More details about these can be found in sections **4.1.1** and **4.1.2**.

5.1 Solution

5.1.1 VisitorInterpreter.java

This class implements the Visitor interface and is used to visit every AST class and perform interpretation for the statements.

- **symbolTable**: This is a variable to hold the symbol table.

All the visit methods implementations can be found below:

1. **visit(ASTAssignment)**:This method performs interpretation on an assignment statement. First, we check if the expression in the node is null in case of a for loop with no assignment and if so, this is skipped. Otherwise, the expression and identifier are obtained from the node. The next step entails looking up the identifier and the expression is then visited, where the expression's result type and value are stored in the global variables constantValue and constantType. Finally a new value is inserted to the hashmap of values with the value obtained from the expression with the key set to the identifier's name.

2. **visit(ASTBinExpression)**: This method performs interpretation on a binary expression node. To do this, the left and right expressions are obtained from the node. Then, the left expression is visited and the type and value of this expression are stored and the same is done afterwards, for the right expression. The next steps involve getting the operand and operating a switch statement on it. Here, for all different operands, the type stored in the global variable holding the constant type is checked and the values stored are converted either to Integer, Float or Boolean according to the type. After this, according to the operand, an operation is performed on these two values and the resulting value is set to the global symbol table variable constantValue. For division, if there is a division by 0, and Arithmetic Exception is thrown.

3. **visit(ASTBlock)**: This method performs interpretation on a block. This function starts by inserting a new scope and removing any previously declared declarations with the name 'return'. After this, the block's statements are obtained for every statement and each statement is visited and if after the visit, there is a declaration with the name 'return', it must mean that something was returned, hence we break out of the loop since we do not need to continue computing statements. Finally, the scope is popped.

4. **visit(ASTBooleanLiteral)**: This method just sets the expression's constant variable type to BOOL and sets the value to the constantValue variable.

5. **visit(ASTfloatLiteral)**: This method just sets the expression's constant variable type to FLOAT and sets the value to the constantValue variable.

6. **visit(ASTFor)**: This method performs interpretation on a for statement. This is done by first getting the variable declaration from the node and visiting it and getting the expression and visiting it. After this the value from the constantValue variable set by the expression visitor method is obtained. Then there is a while loop that goes on until the value from the expression is true. In the loop, the block from the ASTFor is obtained and visited, the assignment is obtained and visited and the expression is revisted, and the value is reset before rechecking for the loop to continue or not. So basically, this continues executing until the expression is false.

7. **visit(ASTFormalParam)**: This method handles a formal parameter. This is done by getting the identifier from the ASTFormalParam node and inserting it to the declarations.

8. **visit(ASTFunctionCall)**: This method performs interpretation on a function call. This is done by first getting the identifier and the actual identifier of the function being called by the **lookup()** method. Then, the expressions from the ASTFunctionCall node are obtained together with the formal parameters from the function's actual identifier. After this, a new scope is inserted and for each expression in the actual parameters object, the expression is visited (where the expression type and value are stored in the global variable) and the value is stored in an arraylist containing the actual parameters' values. After this, all formal parameters are looped and each one is visited, its identifier is stored and a value is stored associated to the formal parameter identifier's name inside the newly created scope. After this, the block is obtained and visited before popping the newly created scope.

9. **visit(ASTFunctionDecl)**: This method handles a function declaration and it only inserts a declaration for the function.

10. **visit(ASTIdentifier)**: This method performs interpretation on an identifier by first getting the value of the identifier. This identifier is searched in the symbol table by the **lookup()** method. Then, I first set the type to the global variable holding the constant expression's type and then I get the value of the identifier from the values hashmap and set it to the global variable holding the constant value.

11. **visit(ASTIf)**: This method performs interpretation on an if statement. This starts by getting the expression form the ASTIf node and visiting it. After this the value is obtained from the constantValue variable. If this is true, the true node is obtained and visited, otherwise, the else block is obtained and visited.

12. **visit(ASTIntegerLiteral)**:This method just sets the expression's constant variable type to INT and sets the value to the constantValue variable.

13. **visit(ASTPrint)**: This method performs interpretation on a print statement by first getting the expression and visiting it. Finally, the value from the variable holding the constant expression's value is printed.

14. **visit(ASTProgram)**: This method performs interpretation on a program node. This is done by creating a new scope and after getting all the statements from the node, visit them one by one. Finally, the scope is popped.

15. **visit(ASTReturn)**: This method performs interpretation on a return statement by first getting the expression and visiting it. Finally, the return type is stored as a declaration so that it could be used by the parent function call function.

16. **visit(ASTUnary)**: This method performs interpretation on a unary expression by first getting the expression and visiting it. After visiting it, the values stored in the constantType and constantValue are stored. If the type is int, a new value is assigned to the constantValue variable by multiplying the old value to -1, else if the type is float, a new value is assigned to the constantValue variable by multiplying the old value to -1.0 or else if the the type is boolean, the old value is negated and a new value is set to constantValue.

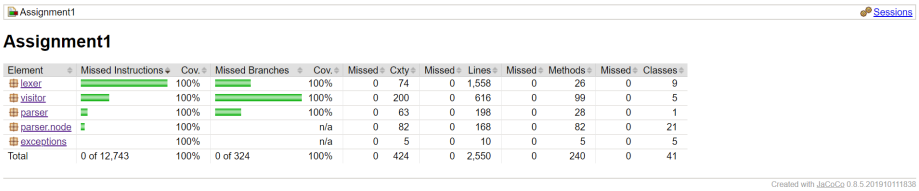
17. **visit(ASTVariableDecl)**: This method performs interpretation on a variable declaration. First, it is checked whether the expression is empty, as can happen in for loops with no variable declaration. If it is not null, the expression is obtained and visited and the identifer is also obtained. Both a declaration and a value are obtained to the symbol table and then the identifier is visited. Finally, the constant variables are set to null and the return declaration is removed in case of a function call.

18. **visit(ASTWhile)**: his method performs interpretation on a while loop statement. This is done by first getting the expression and visiting it. After, the value from the expression is used as the condition for the while loop. In the while look, the block is obtained and visited and the expression of the ASTWhile node is called again and the value for the loop condition is set to the new value for rechecking whether to revisit the block.

19. **interpret()**: This function is the entry point to interpret the input program and resets the symbol table it directly visits the ASTProgram function.

5.2 Testing

In order to test this implementation several integration tests were developed to make sure they produce expected results. This was made sure by having 100% code coverage. Some of these test can be found below.


Assignment1 Sessions

Assignment1

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
lexer		100%		100%	0	74	0	1,558	0	26	0	9
visitor		100%		100%	0	200	0	616	0	99	0	5
parser		100%		100%	0	63	0	198	0	28	0	1
parser.node		100%		n/a	0	82	0	168	0	82	0	21
exceptions		100%		n/a	0	5	0	10	0	5	0	5
Total	0 of 12,743	100%	0 of 324	100%	0	424	0	2,550	0	240	0	41

Created with JaCoCo 0.8.5.201910111838

Figure 40: Code Coverage

```
//just a comment
let i : int = 0;
let f: float = 0.1;
let b: bool = true;

/* ***** IF **** */
if(b)
{
    print 1;
}
else
{
    print 2;
}

/* ***** ELSE **** */
if(not b)
{
    print 1;
}
else
{
    print 2;
}

/*
while
*/
while(f < 0.6)
{
    print f;
    f = f+ 0.1;
}

for(let x:int=1; x<= 5; x = x + 1)
{
    i = x;
    print i;
}

if(true and true)
{
    print true;
}

//expected 1, 2, 0.1, 0.2, 0.3, 0.4, 0.5, 1, 2, 3, 4, 5, true
```

Figure 41: test1.png

```
let i : int = ((1*2)+10);
let j : int = i / 3;
let k : int = j -2;

let f : float = ((1.2*1.3)-0.04)/0.1;

let _boolean : bool = true;
_boolean = true and true;
_boolean = true and not _boolean;

print i;
print j;
print k;
print f;
print _boolean;

// expected - 12, 4, 2, 15.20001, false
```

Figure 42: test2.png

```
/* comment at start */
let x:auto =1;
{
    let x: int = 2+2;
}

ff inf(): auto
{

    let num:int = 1;
    for(;num < 10;)
    {
        print num;
    }
    return 1;
}

//auto declarations
let num: auto = 1;
let f:auto = 2.0;
let b:auto = false;

ff print1(): auto
{
    print 1;
    return 1;
}

if(x == -(-1))
{
    print x;
}

let y: int = print1();

//expected - 1, 1
```

Figure 43: test4.png

```
/* some comment */
let z:int = 4;
let x:int = 5;

ff square(x: int): int
{
    return x*x;
}

//float function
ff floatplus1(x: float): float
{
    return x+1.0;
}

//boolfunc
ff andgate(x:bool):bool
{
    return x and true;
}


ff _try(x: int, y:int):auto
{
    let z : int = square(x);
    if(z>y)
    {
        return z;
    }
    else
    {
        return y;
    }

    return 1;

}

let y: int = _try(z,x);
print y;
print floatplus1(2.0);
print andgate(true);

//expected - 16, 3.0, true
```

Figure 44: test3.png

```
let x: int =1;
let z: int = 2;

ff try(x:int, z:int) : int
{
    if( x > z)
    {
    return 1;
    }
    else
    {
    return 2;
    }

}

let y : int = try(z, x);
print y;

//expected 2
```

Figure 45: test32.png

```
// Function definition for Power
ff Pow(x : float , n : int ) : auto
{
    let y : float = 1.0 ; // Declare y and set it to 1.0
    if( n>0 )
    {
        for (; n>0 ; n=n-1)
        {
            y = y*x; //Assignment y = y*x;
        }
    }
    else
    {
        for (; n<0 ; n=n+1)
        {
            y = y/x; //Assignment y = y/x;
        }
    }
    return y ; // return y as the result
}
let x : auto = Pow(2.1, 10);
print x; //prints to console 1667.9874
```

Figure 46: test33.png

References

[1] "Compiler design lexical analysis." https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis. Accessed on 15-03-2020.

[2] "Introduction of lexical analysis." https://www.geeksforgeeks.org/introduction-of-lexical-analysis/. Accessed on 28-03-2020.

[3] "Construction of ll(1) parsing table." https://www.geeksforgeeks.org/construction-of-ll1-parsing-table/. Accessed on 03-04-2020.

[4] "Let's build a simple interpreter. part 7: Abstract syntax trees." https://ruslanspivak.com/lsbasi-part7/. Accessed on 10-03-2020.