University of Malta – Faculty of ICT

CPS2000
Compiler Theory and Practice

Course Assignment 2021 – Part 1

Mark Mifsud
0382200L

GitHub Repository: https://github.com/markmifsud2000/CPS2000

Table of Contents

Project Structure

Language Used: C++

Environment: Windows 10

Files Included

Task 1

Token/Token.h	Contains the class definition for Token, as well as the TokenType enumeration.
Token/Token.cpp	Implements the Token class.
Lexer/Lexer.h	Contains the class definition for Lexer, as well as the CharacterType enumeration.
Lexer/LexerTable.h	Contains the initialisation of the DFA transition table used by the lexer.
Lexer/Lexer.cpp	Implements the Lexer class.

Task 2

AST/AST.h	Contains the class definitions for all abstract syntax tree class, as well as the enumerations for VariableType and Operator.
AST/AST.cpp	Implements all abstract syntax tree classes.
Parser/Parser.h	Contains the class definition for Parser.
Parser/Parser.cpp	Implements the Parser class.

Task 3

Visitor/Visitor.h	Contains the abstract class definition for Visitor. Used for the visitor design pattern.
Visitor/XMLVisitor.h	Contains the class definition for XMLVisitor, inheriting from Visitor.
Visitor/XMLVisitor.cpp	Implements the XMLVisitor class.

Task 4

SymbolTable/SymbolTable.h	Contains the class definition for SymbolTable. Also defines the Symbol struct.

SymbolTable/SymbolTable.cpp	Implements the SymbolTable class.
Visitor/SemanticVisitor.h	Contains the class definition for SemanticVisitor, inheriting from visitor.
Visitor.SemanticVisitor.cpp	Implements the SemanticVisitor class.

Task 5

Visitor/InterpreterVisitor.h	Contains the class definition for InterpreterVisitor, inheriting from visitor.
Visitor/InterpreterVisitor.cpp	Implements the InterpreterVisitor class.

Other Files

CMakeLists.txt	CMake file used for compiling the executable file.
main.cpp	Compiles and runs a given TeaLang program.
TestProgram.txt	A TeaLang program used during testing.

Running the Program

The program can be compiled using the included CMakeLists.txt file.

In order to run the program the *TeaLang.exe* file should be run from the command prompt. The file location of the TeaLang program file must be passed as a parameter.

The input program will then pass through the lexical, syntax and semantic analysis phases before being executed by the interpreter. An '*AST.xml*' file will also be produced, containing the structure of the syntax tree generated.

Task 1 – Table-Driven Lexer

During the lexical phase, a series of tokens is generated from the input program, where each token represents a substring of the program (a lexeme).

Tokens

In the current implementation, each token is represented by an instance of the Token class, which stores the type of token and, if necessary, and additional information about it, such as the value of an integer literal. The line number of the token in the program is also stored, this will be used throughout compilation when reporting errors to the user.

An enumeration will be used to store the token's type. The following types of tokens can be generated by the lexer.

tREJECTED	*tIDENTIFER*	*tGREATERTHANEQAUL*	*tELSE*
tSPACE	*tAND*	*tEQUALASSIGN*	*tFOR*
tBOOLEAN	*tOR*	*tEQUALRELATIONAL*	*tIF*
tFLOAT	*tNOT*	*tNOTEQUAL*	*tLET*
tINTEGER	*tASTERISK*	*tCOMMA*	*tPRINT*
tSTRING	*tFSLASH*	*tSEMICOLON*	*tRETURN*
tFLOATLITERAL	*tPLUS*	*tLBRACKET*	*tWHILE*
tINTEGERLITERAL	*tMINUS*	*tRBRACKET*	*tEND*
tSTRINGLITERAL	*tLESSTHAN*	*tLCURLY*	
tTRUE	*tLESSTHANEQUAL*	*tRCURLY*	
tFALSE	*tGREATERTHAN*	*tCOMMENT*	

The above list includes some special token types. The lexer generates a *tREJECTED* token whenever an unrecognised lexeme is encountered, the program should then report an error to the user.

tSPACE and *tCOMMENT* tokens are generated when the lexer encounters any whitespace or comments respectively. These should be ignored by the lexer and won't be reported. Both single and multi-line comments are handled in the same way.

The *tEND* token will be reported once the lexer reaches the end of the input program.

toString() and *getTypeName()* methods have also been implemented in the token class. These are used to help during testing and debugging, and improve error reporting.

The Lexer

The lexer class will be used to generate tokens from the input string. In order to determine whether a token should be accepted or not, we will create a deterministic finite-state automaton (DFSA) which only accepts the *TeaLang* language. This DFSA can then be represented as a transition table which will be used by the lexer.

For each character that the lexer reads, the appropriate DFSA transition will be performed using the table. If the transition does not exist in the DFSA, the lexer will stop reading characters and attempt to create the token. If, at some point, the lexer read an accepting state, we can rollback to this state and accept the lexeme. The token's type will be determined depending on which state was accepted.

An enumeration will be used to store the type of character read. Each entry in the enumeration will represent a column in the transition table, while each row represents a DFSA state.

cSTART	Initial Input	cCOMMA	,
cUNKNOWN	Unrecognised Character	cPOINT	.
cNEWLINE	\n	cUNDERSCORE	_
cSPACE	int isspace(int c)	cQUOTATION	"
cDIGIT	int isdigit(int c)	cCOLON	:
cASTERISK	*	cSEMICOLON	;
cFSLASH	/	cLBRACKET	(
cPLUS	+	cRBRACKET	)
cMINUS	-	cLCURLY	{
cLESSTHAN	<	cRCURLY	}
cGREATERTHAN	>	cLETTER	int isalpha(int c)
cEQUALS	=	cPRINTABLE	int isprint(int c)
cEXCLAMATION	!		

The character types *cSPACE*, *cDIGIT*, *cLETTER* and *cPRINTABLE* all represent groups of characters, whitespaces, digits, letters and all other printable characters respectively. These will be determined using C++'s built in functions isspace, isdigit, isalpha and isprint respectively.

The new-line character will be recognised separately from other whitespace characters. This will be used to determine the end of single line comments, but in all other cases it will perform the same transitions as a whitespace in the DFSA. Recognising the line number separately also allows the lexer to track the current line number of the input program. This line number will be stored with the token, and can be used by other parts of the compiler to report errors to the user.

The Lexer Class

The lexer class has been implemented as follows.

Lexer()	Default constructor.
Lexer(string* program)	Constructs a lexer, setting the input program to the given string.
void loadProgram(string* program)	Sets the input program to the given string. Can be called after using the default constructor, or if more than one program is being compiled.
Token getNextToken()	Generates the next token from the input program and returns it. This is the main way other classes can interact with the lexer.
void initialiseReservedWords()	Initialises a mapping of reserved words (as strings) to their corresponding token types. Called by the constructor.
TokenType checkReservedWord(string lexeme)	Checks if a given lexeme is a reserved word in the language. If it is, return it's type, otherwise, the lexeme is just an identifier.

string program	The input program. When a token is generated, this string is replaced by a substring of itself, removing the used lexeme each time.
int lineNum	The current line number in the input program. Used for error reporting.
map<string, TokenType> reserved	Maps lexemes of reserved words to their corresponding token types.

The Transition Table

The transition table has been implemented as a static 2-dimensional integer array defined in the LexerTable.h file. Each row represents a state in the DFSA, and each column represents a character type. Each entry in the table represents a transition in the DFSA, and contains an integer value corresponding to the next state.

In order to determine whether or not a state is accepting, another 1-dimensional array of *TokenTypes* in LexerTable.h is used. The number of entries in this array is equal to the number of states in the DFSA, where state i corresponds to the i^{th} entry in the

array. If the state is not accepting, then its corresponding entry will contain the *tREJECTED* token type. Otherwise, if the state is accepting, the entry will contain the type of token that has been accepted (eg, *tINTEGERLITERAL*).

Recognising Reserved Words

All reserved words will initially be read by the lexer as just identifiers (tIDENTIFIER). In order to determine whether something is a reserved word, or just an identifier, a map will be maintained by the lexer, where the key values are the reserved words as strings, and the corresponding value is the token's type.

When an identifier is encountered, a lookup will be performed on the map. If the lexeme exists in the map, then it must be a reserved word. If it does not, then it is just an identifier.

Although it is possible to recognise all the reserved words using a DFSA, this would require each possible letter to be recognised independently, significantly increasing the size and complexity of the transition table. This is the reason why we have chosen to use a map instead.

The TeaLang language includes the following reserved words.

"bool"	tBOOLEAN	"not"	tNOT
"float"	tFLOAT	"else"	tELSE
"int"	tINTEGER	"for"	tFOR
"string"	tSTRING	"if"	tIF
"true"	tTRUE	"let"	tLET
"false"	tFALSE	"print"	tPRINT
"and"	tAND	"return"	tRETURN
"or"	tOR	"while"	tWHILE

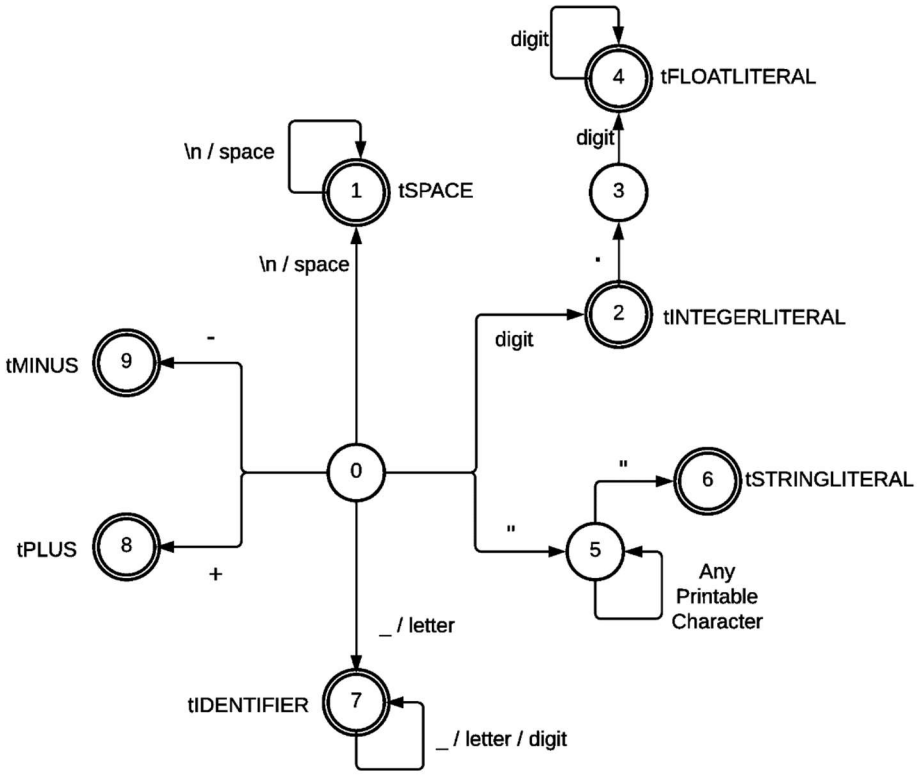
Deterministic Finite State Automaton

The deterministic finite-state automaton used by the lexer to accept the TeaLang language are listed below. State 0 represents the initial state.

All three figures form part of the same DFSA, however they have been split up for the sake of clarity and readability.

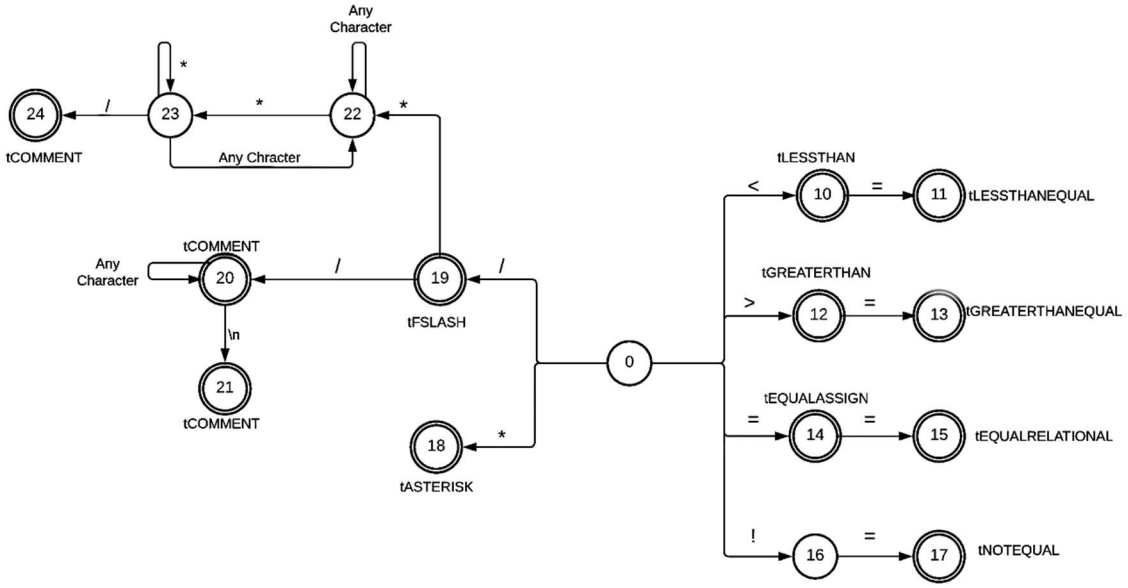
Tokens Accepted:

tSPACE	*tINTEGERLITERAL*	*tFLOATLITERAL*	*tSTRINGLITERAL*
tIDENTIFIER	*tPLUS*	*tMINUS*	



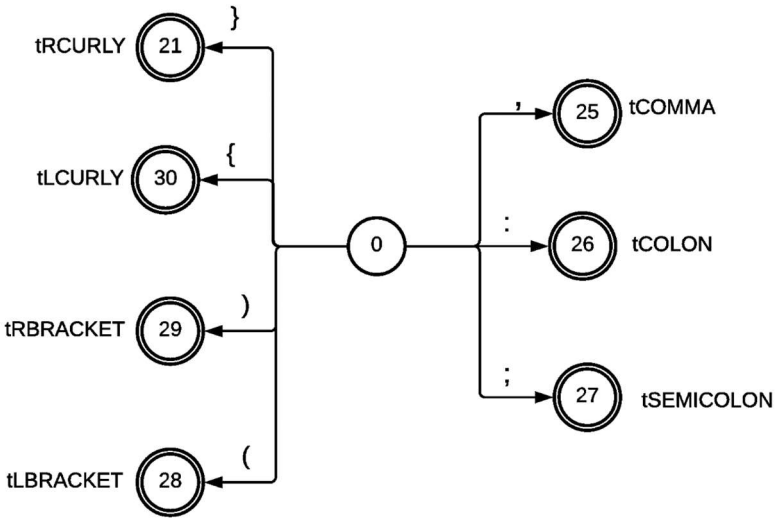
Tokens Accepted:

tLESSTHAN	*tLESSTHANEQUAL*	*tGREATERTHAN*	*tGREATERTHANEQUAL*
tEQUALASSIGN	*tEQUALRELATIONAL*	*tNOTEQUAL*	*tASTERISK*
tFSLASH	*tCOMMENT*		



Tokens Accepted:

tCOMMA	*tCOLON*	*tSEMICOLON*	*rLBRACKET*
tRBRACKET	*tLCURLY*	*tRCURLY*	



Transition Table

State/Character	cStart	cUnknown	cNewLine	cSpace	cDigit	cAsterisk	cFSlash	cPlus	cMinus	cLessThan	cGreaterThan	cEquals	cExclamation	cComma	cPoint	cUnderscore	cQuotation	cColon	cSemicolon	cLBracket	cRBracket	cLCurly	cRCurly	cLetter	cPrintable
State 00	E	E	1	1	2	18	19	8	9	10	12	14	16	25	E	7	5	26	27	28	29	30	31	7	E
State 01	E	E	1	1	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
State 02	E	E	E	E	2	E	E	E	E	E	E	E	E	E	3	E	E	E	E	E	E	E	E	E	E
State 03	E	E	E	E	4	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
State 04	E	E	E	E	4	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
State 05	E	E	5	5	5	5	5	5	5	5	5	5	5	5	5	5	6	5	5	5	5	5	5	5	5
State 06	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
State 07	E	E	E	E	7	E	E	E	E	E	E	E	E	E	E	7	E	E	E	E	E	E	E	7	E
State 08	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
State 09	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
State 10	E	E	E	E	E	E	E	E	E	E	E	11	E	E	E	E	E	E	E	E	E	E	E	E	E
State 11	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
State 12	E	E	E	E	E	E	E	E	E	E	E	13	E	E	E	E	E	E	E	E	E	E	E	E	E
State 13	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
State 14	E	E	E	E	E	E	E	E	E	E	E	14	E	E	E	E	E	E	E	E	E	E	E	E	E
State 15	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
State 16	E	E	E	E	E	E	E	E	E	E	E	17	E	E	E	E	E	E	E	E	E	E	E	E	E
State 17	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
State 18	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
State 19	E	E	E	E	E	22	20	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
State 20	E	20	21	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20
State 21	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
State 22	E	22	22	22	22	23	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22
State 23	E	22	22	22	22	23	24	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22
State 24	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
State 25	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
State 26	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
State 27	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
State 28	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
State 29	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
State 30	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
State 31	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
State E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E

Testing

In order to test the lexer, a number of input TeaLang programs were created. Each program was passed to the lexer as a string, and the tokens were generated using the *getNextToken()* method. This method was called using a loop until the *tEND* token was received.

```
1. Lexer l(&program);
2. Token t = l.getNextToken();
3. while (t.type != tEND && t.type != tREJECTED) {
4.     cout << t.toString() << endl;
5.     t = l.getNextToken();
6. }
```

Some of the results obtained are listed below.

Input Program:

```
1. print y; //6.25
```

Tokens Generated:

```
1. < 1 : print >
2. < 1 : Identifier, y >
3. < 1 : ; >
```

Input Program:

```
1. /* Hello World
```

Error Generated:

```
terminate called after throwing an instance of 'std::runtime_error'
  what():  Line 1: Unexpected /

Process finished with exit code 3
```

Input Program:

```
1. let str:string = "Hello" + "World";
2. print str;
3. let x:float = 2.4;
4. let y:float = Square (2.5);
```

Tokens Generated:

```
1. < 1 : let >
2. < 1 : Identifier, str >
3. < 1 : : >
4. < 1 : string >
5. < 1 : = >
6. < 1 : StringLiteral, "Hello" >
7. < 1 : + >
8. < 1 : StringLiteral, "World" >
9. < 1 : ; >
10. < 2 : print >
11. < 2 : Identifier, str >
12. < 2 : ; >
13. < 3 : let >
14. < 3 : Identifier, x >
15. < 3 : : >
16. < 3 : float >
17. < 3 : = >
18. < 3 : FloatLiteral, 2.4 >
19. < 3 : ; >
20. < 4 : let >
21. < 4 : Identifier, y >
22. < 4 : : >
23. < 4 : float >
24. < 4 : = >
25. < 4 : Identifier, Square >
26. < 4 : ( >
27. < 4 : FloatLiteral, 2.5 >
28. < 4 : ) >
29. < 4 : ; >
```

Input Program:

```
1. int printLoop (n:int) {
2.     for (let i:int = 0; i<n; i=i+1) {
3.         print i;
4.     }
5.     return 0;
6. }
```

Tokens Generated:

```
1. < 1 : int >
2. < 1 : Identifier, printLoop >
3. < 1 : ( >
4. < 1 : Identifier, n >
5. < 1 : : >
6. < 1 : int >
7. < 1 : ) >
8. < 1 : { >
9. < 2 : for >
10. < 2 : ( >
11. < 2 : let >
12. < 2 : Identifier, i >
13. < 2 : : >
14. < 2 : int >
15. < 2 : = >
16. < 2 : IntegerLiteral, 0 >
17. < 2 : ; >
18. < 2 : Identifier, i >
19. < 2 : < >
20. < 2 : Identifier, n >
21. < 2 : ; >
22. < 2 : Identifier, i >
23. < 2 : = >
24. < 2 : Identifier, i >
25. < 2 : + >
26. < 2 : IntegerLiteral, 1 >
27. < 2 : ) >
28. < 2 : { >
29. < 3 : print >
30. < 3 : Identifier, i >
31. < 3 : ; >
32. < 4 : } >
33. < 5 : return >
34. < 5 : IntegerLiteral, 0 >
35. < 5 : ; >
36. < 6 : } >
```

Task 2 – Hand-Crafter LL(k) Parser

Once we are able to generate tokens, we must now verify that the list of tokens generated actually form the structure of a program according to the TeaLang specification. This is called syntax analysis and will be handled by the parser.

The Parser will accept tokens from the lexer and then use these tokens to build an abstract syntax tree. If the parser does not throw any syntax errors, the resulting abstract tree generated should represent the final structure of the program.

The Abstract Syntax Tree

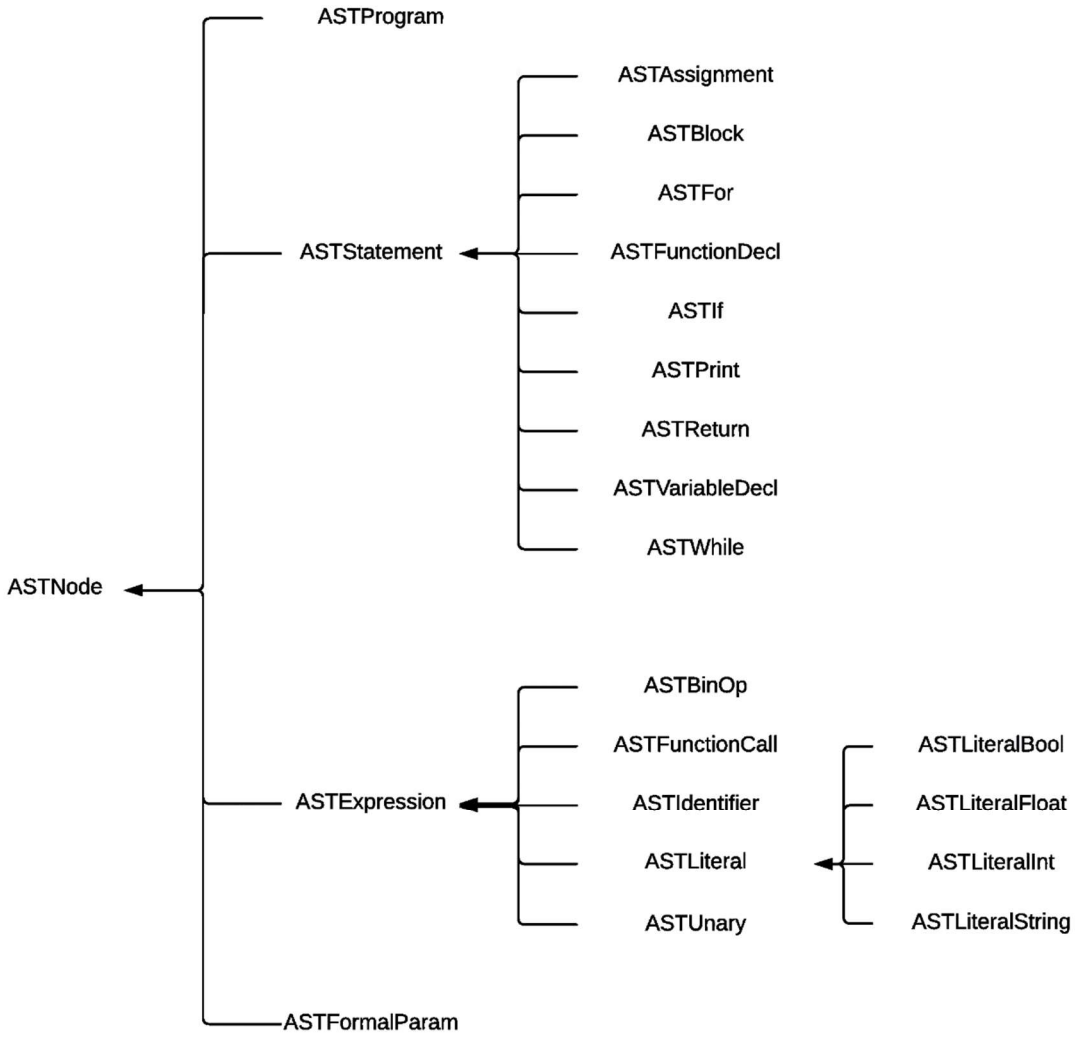
The abstract syntax tree (AST) is generated after the parser parses a correctly structured program. Each node in the AST represents a sub-structure in the TeaLang language. This includes non-terminal nodes (eg Block, For, etc.) which contain other nodes as their children, and terminal nodes (eg LiteralString, Identifier, etc.) which will only contain a value.

Once the tree has been successfully built, it can then be traversed in order to perform operations on the program, such as XML generation and semantic analysis, which will be discussed later.

AST Classes

Each type of node that we have in the program is represented by a separate class. All classes contain a constructor which creates the node with all values initialised, and stores the line number at which the node was created, which can then be used when outputting error messages to the user.

While implementing the AST, a class hierarchy was used, with every AST node inheriting from an abstract *ASTNode* parent class. Additionally, the abstract classes *ASTStatement*, *ASTExpression* and *ASTLiteral* have also been created, and allow the use of polymorphism in cases such as $< Program > = \{< Statement >\}$ (Taken from the *TeaLang* Specification). A diagram of the class hierarchy used is shown below.



The difference between each of the implemented classes lies in the type of data that they store, either as child nodes or directly as values. A table of these values is included below. Where a node can contain multiple children of a single type, such as $< Program > = \{< Statement >\}$, C++'s vector will be used.

ASTNode	int lineNum
ASTProgram	vector<ASTStatement*> program
ASTAssignment	ASTIdentifier* identifier ASTExpression* value
ASTBinOp	ASTExpression* lExpression Operator op ASTExpression* rExpression

ASTBlock	vector<ASTStatement> block
ASTFor	ASTVariableDecl* declaration ASTExpression* conditional ASTAssignment* assignment ASTBlock* block
ASTFormalParam	ASTIdentifier* identifier VariableType type
ASTFunctionCall	ASTIdentifier* identifier vector<ASTExpression*> param
ASTFunctionDecl	VariableType returnType ASTIdentifier* identifier vector<ASTFormalParam> parameters ASTBlock* block
ASTIdentifier	string identifier
ASTIf	ASTExpression* conditional ASTBlock* ifBlock ASTBlock* elseBlock
ASTLiteralBool	bool b
AST LiteralFloat	float f
AST LiteralInt	int i
ASTLiteralString	string s
ASTPrint	ASTExpression* expression
ASTReturn	ASTExpression* returnValue
ASTUnary	Operator op ASTExpression* expression
ASTVariableDecl	ASTIdentifier* identifier VariableType type ASTExpression* value
ASTWhile	ASTExpression* conditional ASTBlock* block

In the above class definitions, two enumerations where used, *VariableType* and *Operator*. *VariableType* will be used to represent the data types stored in nodes, such as the data type of a new variable declared or the return type of a function declaration. *Operator* will be used during binary and unary operations to indicate the type of operation that actually needs to be performed.

VariableType:

INCOMPATIBLE	*BOOL*	*FLOAT*
INT	*STRING*	

INCOMPATIBLE is used during semantic analysis to check when a data type conversion is not possible.

Operator:

MULT	*DIVIDE*	*PLUS*	*MINUS*
LESSTHAN	*LESSTHANEQUAL*	*GREATERTHAN*	*GREATERTHANEQUAL*
EQUALS	*NOTEQUALS*	*AND*	*OR*
NOT			

The Parser

The Parser is responsible for accepting tokens from the lexer and using these to generate an AST. While creating the AST, the parser will ensure that the program's syntax is correct.

The parser interacts with the lexer by calling the *Lexer::getNextToken()* method, which generates a token from the input program. Since a lookahead parser has been implemented, these tokens will be stored and used to determine what type of structure is being parsed. A method *Parser::getNextToken()* has also been implemented to update the tokens stored by Parser.

In most cases, 1 token is enough to determine which type of structure is being parsed. The only exception is when the parser encounters an <*Identifier*> token. In this case, a second token is needed to determine whether we must parse a function call, or only an identifier.

$$ASTIdentifier = < Identifier >$$

$$ASTFunctionCall = < Identifier >' \ '(' \ < Expression > \ \{',' < Expression >\} \ ')'$$

Once the parser has been constructed, the user can call the *parseProgram()* method, which, after parsing the program, will return an *ASTProgram** node which represents the structure of the entire program.

When a parse function is called, the parser will check each token in the appropriate derivation. If the token is only used for program structure, and does not contain any actual information, such as ';' (*tSEMICOLON*) or '(' (*tLBRACKET*), this can be discarded. If the token contains any data, such as tIDENTIFIER, or if another derivation is expected, the parser will call the appropriate parse function which will return an appropriate AST class. If an unexpected token is encountered, the parser will throw an error.

Parser Class

The Parser class has been implemented as follows.

Parser()	Default constructor.
Parser(string* program)	Constructs a parser, setting the input program to the given string.
void loadProgram(string* program)	Sets the input program to the given string. Can be called after using the default constructor, or if more than one program is being compiled.
Token getNextToken()	Gets the next token from the lexer. The previous lookahead token is replaced by the new one.

Lexer lexer	The lexer used to generate tokens.
Token next	The next token to be read. Used when deciding which derivation to use.
Token nextnext	The second token to be read. Used when two lookahead tokens are required.

Parse Functions

ASTProgram*	parseProgram()
ASTStatement*	parseStatement()
ASTExpression*	parseFactor()
ASTLiteral*	parseLiteral()
ASTAssignment*	parseAssignment()
ASTBlock*	parseBlock()
ASTExpression*	parseExpression()
ASTFor*	parseFor()

ASTFormalParam*	parseFormalParam()
ASTFunctionCall*	parseFunctionCall()
ASTFunctionDecl*	parseFunctionDecl()
ASTIdentifier*	parseIdentifier()
ASTIf*	parseIf()
ASTPrint*	parsePrint()
ASTReturn*	parseReturn()
ASTExpression*	parseSimpleExpresion()
ASTExpression*	parseSubExpression()
ASTExpression*	parseTerm()
ASTUnary*	parseUnary()
ASTVariableDecl*	parseVariableDecl()
ASTWhile*	parseWhile()

Testing

Whenever the parser is tested with a valid program, no errors were reported. We will ensure that the AST has been built correctly during Task 3, when we can generate an XML file representing the structure of the AST.

During this section, we will make sure that the parser can detect any syntactically incorrect programs.

Input Program:

```
1. /* Test Program */
2. let x:x = 10;
```

Parser Output:

```
terminate called after throwing an instance of 'std::runtime_error'
  what():  Line 2: Expected type, found Identifier
```

Input Program:

```
1. int printLoop (n:int) {
2.     for (let i:int = 0; i<n; i=i+1) {
3.         print i
4.     }
5.     return 0;
6. }
```

Parser Output:

```
terminate called after throwing an instance of 'std::runtime_error'
  what():  Line 4: Expected ';', found }
```

Input Program:

```
1. bool XGreaterThanY (x:float, y:float) {
2.     let ans : bool = true;
3.     if (y>x) {ans = false;}
4.     return;
5. }
```

Parser Output:

```
terminate called after throwing an instance of 'std::runtime_error'
  what():  Line 4: Unexpected ;
```

Task 3 – AST XML Generation Pass

Once we have produced the AST, we can now traverse it performing various operations. One such operation is generating an XML representation of the program.

The XML Visitor

In order to generate the XML file, the visitor design pattern will be used.

An abstract *Visitor* class was created, with a single method, *visit*. This *visit* method is then overloaded so that it can accept any type of *ASTNode* as a parameter. An *accept* function was also added to each AST class, which accepts a visitor class and calls the appropriate *visit* method.

The *XMLVisitor* class inherits from *Visitor*, implementing each instance of the *visit* method.

When a node in the AST is visited, an XML tag will be added to the output describing the visited node. The visitor will then be passed onto any child nodes so that their structure can also be added to the output file. When the visitor returns from the child nodes, the original tag can then be closed.

In order to maintain indentation throughout, an integer value will be stored representing the number of indents to add before each line. When a new tag is opened, it will increase the number of indents, and decrease it when the tag is closed. An *addIndent()* method can then be called at the start of each line.

Testing

In order to test out XML generator, we will parse a number of syntactically correct programs and inspect the XML file generated.

Input Program:

```
1. print y;
```

XML Generated:

```
<Program>
        <print>
                <Id>y</Id>
        </print>
</Program>
```

Input Program:

```
1. bool XGreaterThanY (x:float, y:float) {
2.     let ans : bool = true;
3.     if (y>x) {ans = false;}
4.     return ans;
5. }
```

XML Generated:

```
<Program>
	<Function return="bool">
		<Id>XGreaterThanY</Id>
		<Params>
			<Param type="float">
				<Id>x</Id>
			</Param>
			<Param type="float">
				<Id>y</Id>
			</Param>
		</Params>
		<Block>
			<VariableDecl type="bool">
				<Id>ans</Id>
				<Literal type="bool">true</Literal>
			</VariableDecl>
			<If>
				<Condition>
					<BinOp op=">">
						<Id>y</Id>
						<Id>x</Id>
					</BinOp>
				</Condition>
				<Block>
					<Assignment>
						<Id>ans</Id>
						<Literal
type="bool">false</Literal>
					</Assignment>
				</Block>
			</If>
			<return>
				<Id>ans</Id>
			</return>
		</Block>
	</Function>
</Program>
```

Task 4 – Semantic Analysis Pass

Using the visitor design pattern, we can also perform semantic analysis on the AST. Once the parser has confirmed that the structure of the program is correct, we can now perform a number of checks to make sure all values are correct.

The Symbol Table

In order to keep track of any variables and functions declared during the program, we must implement a symbol table. Each symbol in the table will either represent a variable or a function, and will store its data type. During execution, the symbol will also store its current value.

The symbol table will act similarly to a stack, whenever the program enters a new scope, we will push onto the stack. Once we leave the scope, we will then pop form the stack. This ensures that variables are no longer visible once they reach the end of their scope. It also allows for variables to be shadowed, if a matching identifier is found in an inner scope, there is no need to check any outer scopes. The outermost scope will act as the global scope and the top of the stack will act as the current scope.

However, the symbol table is not actually implemented as a stack. Since we need to be able to check every scope when searching for a variable, using C++'s stack is not possible since only the innermost scope would be accessible. Instead we will use a vector and implement our own *push()*, *pop()* and *top()* methods.

In order to store all the symbols within a scope, we will use a map. The identifier string will be used as a key value, and the Symbol will be the value. When searching for a symbol in the table, a map lookup will be performed. The innermost scope will be searched first before moving outwards until a matching symbol is found. Declaring a variable or function involves adding the key and value pair to the map in the current scope.

When handling functions, there are additional considerations that must be made. Since there may be multiple functions with the same identifier but different parameters, we must implement some method of differentiating between them. We will store each of these functions in the same symbol, with the same identifier. This allows us to use the same lookup functions as before, but if we find a function, we must also check whether the parameter list matches. When declaring the function, we will also store a pointer to it's *ASTFunctionDecl* node, as this will allow us to actually call it's code during execution.

Symbol Struct

The Symbol struct has been implemented as follows.

VariableType type	The data type of the identifier/type returned by the function.
int lineNum	The line at which the symbol is declared.
vector<ASTFunctionDecl*>	The list of functions that have the same identifier. Used during function overloading.
ASTLiteral* value	The stored by the identifier. Only used during execution.

SymbolTable Class

The SymbolTable class has been implemented as follows.

Scope is a typedef for *map<string, Symbol>.*

SymbolTable()	Default constructor.
bool isDeclared(string* program) **bool isDeclared(string* program, vector<VariableType>* types)**	Checks if a given identifier has been declared in the program. For a function, the parameter types must also be given.
bool isDeclaredScope(string id)	Check if an identifier is declared, but only check the current scope. Allows for shadowing of variables declared in outer scopes.
void declare(ASTVariableDecl* node) **void declare(ASTFunctionDecl* node)** **void declare(ASTFormalParam* node)**	Declares a variable/function in the current scope.
VariableType getType(string s)	Get the data type of a variable/function.
ASTFunctionDecl* getFunction(string s, vector<VariableType>* types)	Get the function declaration node with a given identifier and parameter types.
Scope::iterator findSymbol(string id)	Find a key/value pair from the map.
void assign(string id, bool value) **void assign(string id, float value)** **void assign(string id, int value)** **void assign(string id, string value)**	Assign a new value to a variable in the symbol table.
void push()	Push a new Scope onto the stack.
void pop()	Pop the top Scope from the stack.
Scope* top()	Get the top (innermost) Scope from the stack.

vector<Scope> stack	The stack used to manage scopes.

The Semantic Visitor

Similar to the XML visitor, the semantic visitor will visit each node in the abstract syntax tree and carry out an operation. However, instead of generating an XML file, we will instead check a number of semantic rules to make sure that the program is valid. The *SemanticVisitor* class will implement all the *visit()* functions from the *Visitor* class.

The semantic visitor will make use of the symbol table whenever an identifier is encountered. If a declaration statement is encountered, the variable/function will be declared in the symbol table. If the identifier is already declared in the current scope, the symbol table will throw an error to the user. When an identifier, assignment or function call is encountered, the identifier will be looked up in the symbol table to ensure that it has already been declared.

When traversing the AST, we need to keep track of the data type returned by any expressions encountered. A *returnedType* variable will be used to track this. After visiting an expression or literal node, this value will be updated according to the data type evaluated. When the visit function returns, we can then check this variable to ensure that any data types match.

Semantic Rules

Automatic Type Casting

During execution, it is sometimes possible to automatically convert values from one data type to another (eg. int to float). In order to check this, whenever a certain data type is required, we will call the *doTypesMatch(expected, actual)* function, which will return true if a type cast is possible.

In the current implementation, booleans, floats and integers can be converted interchangeably (although sometimes with a loss of data), but strings cannot be converted into anything.

Operator Types

Whenever an operator is encountered, we must check if the operator type is compatible with the given operands, and if so, what data type will be returned. The opReturnType(lType, op, rType) will be used, and will return the data type that would be returned by the expression.

When we encounter a multiplication or division, we can just convert the values to floats to carry out the operation. The value can then be cast back to the original type afterwards.

Addition and Subtraction only needs to ensure that both operands are compatible with each other. The returned type will be same type as the operands.

When performing any comparison operations, we just need to ensure that both types are compatible with each other. The returned type will always be a boolean.

For boolean operations *AND* and *OR*, we will cast both operands to boolean to perform the operation. The returned type will also be a boolean.

Boolean Conditions

When encountering a conditional statement, we must ensure that the conditional statement returns a boolean value. This occurs when visiting *IF*, *FOR* and *WHILE* nodes.

Function Calls

When encountering a function call, aside from just searching for the identifier in the symbol table, we must also check if the parameter list matches an existing function declaration. Since a function may be overloaded, no automatic typecasting will take place on parameter types to ensure that the correct function is called.

Function Returns

When parsing a function declaration, we must ensure that the function includes a return type. We will use a *returnSatement* boolean variable to check this. This variable will only be set to true when a return node has been visited. Once we confirm that a return statement is present, we then only need to check that the data type returned matches the functions declared return type. The return statement must be the last statement in the function.

Testing

In order to test the semantic visitor, we will first parse an input program using the parser implemented earlier. Once the AST has been generated, we will visit the *ASTProgram* node with the *SemanticVisitor*.

Whenever the semantic analyser is tested with a valid program, no errors were reported. During this section, we will make sure that the analyser can detect any semantically incorrect programs.

Input Program:

```
1. print y;
```

Error Output:

```
terminate called after throwing an instance of 'std::runtime_error'
  what():  Line 1: Variable y has not been declared.
```

Input Program:

```
1. float Square (x:float) {
2.     return x*x;
3. }
4.
5. let x:int = Square(10, 10);
```

Error Output:

```
terminate called after throwing an instance of 'std::runtime_error'
  what():  Line 5: Function Square(int, int) is not defined.
```

Input Program:

```
1. let x:int = 10;
2. x = "Hello";
```

Error Output:

```
terminate called after throwing an instance of 'std::runtime_error'
  what():  Line 2: Variable x is of type int but found string.
```

Input Program:

```
1. let x:int = 10 + "World";
```

Error Output:

```
terminate called after throwing an instance of 'std::runtime_error'
  what():  Line 1: Types int and string are not compatible under this operation.
```

Input Program:

```
1. if ("true") {
2.     print "Hello";
3. }
```

Error Output:

```
terminate called after throwing an instance of 'std::runtime_error'
  what():  Line 1: Expected type bool, found type string.
```

Input Program:

```
1. float Square (x:float) {
2.     let ans:float = x*x;
3. }
```

Error Output:

```
terminate called after throwing an instance of 'std::runtime_error'
  what():  Line 1: Missing return statement.
```

Task 5 – Interpreter Execution Pass

Now that we have confirmed that the program is structured correctly, we can now execute it. We can once again use the visitor design pattern to execute each node in the AST generated by the parser. The symbol table will also be used to track any variables and functions.

The interpreter assumes that semantic analysis has already taken place, and assumes that all operations can be carried out.

The Interpreter

We will once again use the visitor design pattern. The interpreter visitor will visit each node in the AST executing each node in order. The *IntepreterVisitor* class will implement all the *visit()* functions from the *visitor* class.

The Symbol Table

During execution, we will once again make use of the symbol table. Whenever a variable is declared, it will be added to the symbol table, and its initial value will be assigned. When a function is declared, it will be added to the table and a pointer to its declaration node will also be stored. Whenever an identifier is encountered, we will look it up in the symbol table and return its associated value.

Returned Values

In order to temporarily store any returned values in between nodes, the variables *returnedBool, returnedFloat, returnedInt* and *returnedString* have been defined. When a value is returned, it will be stored in the appropriate variable, and *returnedType* will be set to identify which variable should be accessed.

Convert Returned Type

As described in the previous section, the interpreter can perform automatic type casting in some cases. The *convertReturnedType(type)* function will be used to convert the last returned type to a new type. The new value will not be returned by the function, but will be assigned to one of the returned value variables described above.

Function Calls

When a function is called, a lookup is performed in the symbol table to find the matching identifier with matching parameter types. Before executing the function, we must first declare the parameters as variables in the scope. When declaring the

parameters, we will take the identifier name from the formal parameter in the function declaration, and assign the value from the actual parameter in the function call. This ensures the variables match what is used in the function block.

Once all parameters have been declared, we can then visit the block in the function's declaration and continue execution there.

Testing

To test the interpreter, we will take a number of input programs and ensure that they produce the expected output. We will first perform syntax and semantic analysis on to ensure the program is correct, then we can visit the *ASTProgram* node with the *InterpreterVisitor*.

Input Program:

```
1. print "Hello World";
```

Interpreter Output:

Hello World

Input Program:

```
1. let x:int = 10/2;
2. print x;
```

Interpreter Output:

5

Input Program:

```
1. bool XGreaterThanY (x:float, y:float) {
2.     let ans : bool = true;
3.     if (y>x) {ans = false;}
4.     return ans;
5. }
6.
7. let ans:bool = XGreaterThanY(10.0, 20.0);
8. print ans;
```

Interpreter Output:

false

Input Program:

```
1. int printLoop (n:int) {
2.     for (let i:int = 0; i<n; i=i+1) {
3.         print i;
4.     }
5.     return 0;
6. }
7.
8. let n:float = printLoop(5);
```

Interpreter Output:

0

1

2

3

4

Input Program:

```
1. int output(x:int) {
2.     print "int";
3.     return 0;
4. }
5.
6. int output(x:float) {
7.     print "float";
8.     return 0;
9. }
10.
11.     let n:int = 0;
12.     n = output(10);
13.     n = output(32.2);
```

Interpreter Output:

int

float