CPS 2000 Assignment Documentation

Table of Contents

Structure of Program 2
Main Program 3
Task 1 – Table-driven Lexer 4
Overview of Lexer's *getNextToken()* 4
Constructing the Transition Table 5
Reserved Keywords 7
Testing 7
Lexer Limitations and Changes 9
Task 2 – Hand-crafted LL(k) parser 10
The Parser class 10
Handling LL(1) and LL(2) 12
The Abstract Syntax Tree 12
Testing 14
Task 3 – AST XML Generation Pass 15
Overview 15
Testing 15
Task 4 – Semantic Analysis pass 17
Overview 17
The Scope Struct 17
The Symbol Table Class 17
Type checking 18
Function returns 18
Conditionals 18
Testing 18
..... 18
Task 5 – Interpreter Execution Pass 21
Overview 21
The Symbol Table 21
Visit Classes 21
Testing 21

Structure of Program

Programming language: C++ (CMake 3.18, CLion IDE)

File structure:

- *miniLang.txt*

  Text file used to write MiniLang code for compiler to run.

- *Main.cpp*

  Main file which loads MiniLang code through all tasks of the assignment.

- *Lexer.hpp*

  Contains Token class and Lexer classes which is used for Task 1.

- *LexerTable.hpp*

  Contains DFA used for Lexer as well as enum values for character list used and token list used.

- *Lexer.cpp*

  Implementation for both the Token and Lexer Classes.

- *ASTClasses.hpp*

  Implementation of class nodes to represent EBNF structure via AST classes, starting from ASTNode. Also contains enum values for operator symbols (used when storing operators in classes).

- *Parser.hpp*

  Contains Parser classes used for Task 2.

- *Parser.cpp*

  Implementation of Parser class, parser class also used for initializations of Tasks 3 – 5.

- *VisitorDesign.hpp*

  Contains class for Visitor Design Pattern with overriding classes for XML Generation Pass (Task 3), Semantic Analysis Pass (Task 4) and finally the Interpreter Execution Pass (Task 5).

- *XMLVisitorDesign.hpp*

  Implementation of XML Generation Pass class.

- *SymTable.hpp*

  Contains Symbol table class, struct for maintaining a scope and a typedef for a map which stores each scope. (Used in Task 4 and 5)

- *SymTable.cpp*

  Implementation of Symbol table class.

- *InterpreterPass.cpp*

  Implementation of Interpreter Execution Pass class.

Main Program

'Main.cpp' contains calls made by the compiler throughout each task. When testing the lexer a utility function was created (currently commented at line 14) which runs the lexer on its own and prints each token with its respective identifier. When using this utility all other calls from parser to interpreter would need to be commented for use (lines 16 till 19).

```
int main() {

    String myfile = ("../miniLang.txt");
    std::vector<String> filecontents = readToString( pathName: myfile);

    std::shared_ptr<Parser> myParse = std::make_shared<Parser>( file: filecontents);
    //myParse->Testlexer(); //Used to test Lexer implementation.

    myParse->BeginParse(); //Parsing phase
    myParse->XMLPass(); //XML phase
    myParse->SemanticPass(); //Semantic Analysis phase
    myParse->InterpretPass(); //Interpreter phase

    return 0;
}
```

Figure 1 Compiler configured to run from lexer till interpreter

```
int main() {

    String myfile = ("../miniLang.txt");
    std::vector<String> filecontents = readToString( pathName: myfile);

    std::shared_ptr<Parser> myParse = std::make_shared<Parser>( file: filecontents);
    myParse->Testlexer(); //Used to test Lexer implementation.

    //myParse->BeginParse(); //Parsing phase
    //myParse->XMLPass(); //XML phase
    //myParse->SemanticPass(); //Semantic Analysis phase
    //myParse->InterpretPass(); //Interpreter phase

    return 0;
}
```

Figure 2 Compiler configured to just print tokens created by the Lexer

Task 1 – Table-driven Lexer

The Lexer class

The following is a table of the list of functions/storage created for the Lexer class:

`Lexer(std:: vector<String> input);`	Constructor who prints input to screen, and stores each line of input into vector buffer 'myData'
`~Lexer();`	Default destructor
`std::map<String, MyTokens> Reserved;`	Storage of each reserved token as a string, token pair.
`Token getNextToken();`	Main skeleton scanner of the Lexer.
`bool isAccepting(int state);`	Checks if current state is an accepting state.
`std::vector<String> myData;`	Stores input of miniLang.txt line by line in a vector.
`void setReserved();`	Initializes each reserved keyword as a map of string, token pair.
`MyTokens checkReserved(const String word);`	Called to check if an Identify token is a reserved keyword.
`int currentLine;`	Stores the current line during *getNextToken().*
`int stringPtr;`	Points to current character in *getNextToken().*

The following is a table of the list of functions/storage created for the Token class:

`Token();`	Default constructor.
`~Token();`	Default destructor.
`std::vector<MyTokens> type;`	Stores a particular token name as an enum value. (Used during parser phase)
`std::vector<String> word;`	Stores the actual token. (Used during parser phase)
`bool end = false;`	Kept to check if *getNextToken();* has reached end of file
`void printResult();`	Utility function which prints each token in the form {Line / Actual Token / Token Name}
`int currLine;`	Stores the current line during *getNextToken().*
`String getTypeName(MyTokens type);`	Conversion from String to Token enum value.

Overview of Lexer's *getNextToken()*

When constructing the Lexer a table driver approach was used with the skeleton scanner being split into its respective four stages.

These stages are:

1) Initialization
2) Scanning loop
3) Rollback loop
4) Final Section

This method was kept close to the method shown during lectures, by simulating a DFA with the use of a Transition Table. This Table acted as the simulation of the actual DFA. The lexer

will first start from the initial state of the table S0 and maintaining a stack being initialized by a bad sate (being represented as -1). In addition to the initialization stage, a counter is also kept which maintains the current line the lexer is in when going through tokens (used for error checking).

The scanner loop functions by iterating character by character and checking if we are currently in an error state or at the end of the line, the stack is manipulated during this stage in such a way to keep track if we have landed in a final state. Final states are checked by calling a helper function isAccepting(int state) which contains a list of all final states within the DFA. If we are in a final state, everything in the current stack is popped (cleared) and the final state is pushed onto the stack.

The rollback loop then starts by looping till either the top of the stack contains either a bad state or an accepting state, the inner loop functions by iteratively truncating the lexeme and popping the stack. Since the lexeme is being truncated, for better error handling a duplicate untouched lexeme is stored which is displayed during error checking.

The final stage is reporting back to the parser, if the current stack is set as the bad state an error is reported back to the user stating what lexeme and in what line isn't valid. If a final state is successfully found the Lexer proceeds to check if the token is an identifier, this can mean that the token is also a reserved keyword (such as integer, bool etc) thus, checkReserved(const String word) is used if conversion is also needed. In addition, if the final state is a space token this either stage starts again from the next character since spaces aren't considered by the Lexer.

Constructing the Transition Table

When deciding what the DFA of the program will look like two important factors had to be taken:

1) What the actual accepted characters are.
2) What tokens are to be considered.

For ease of reading enum classes were created for these two factors. To better understand which tokens/characters these would be a DFA was drawn which is derived from the specifications of the EBNF of what the final DFA should look like and thus what all the characters and tokens are. The following is the DFA considered, the DFA has been spilt into parts due to its scale but, every state emanates from the same starting node S0.

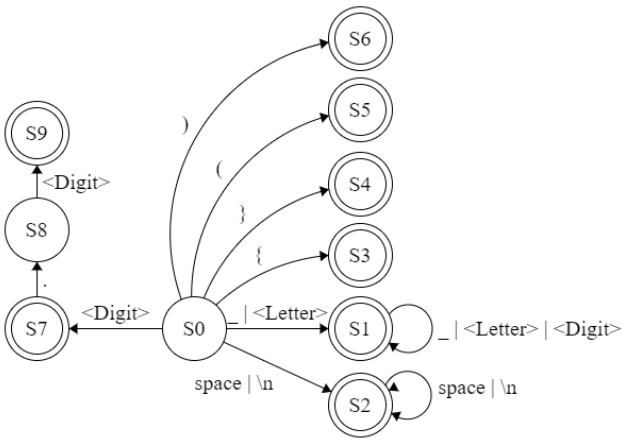

Figure 3 States S1 - S9

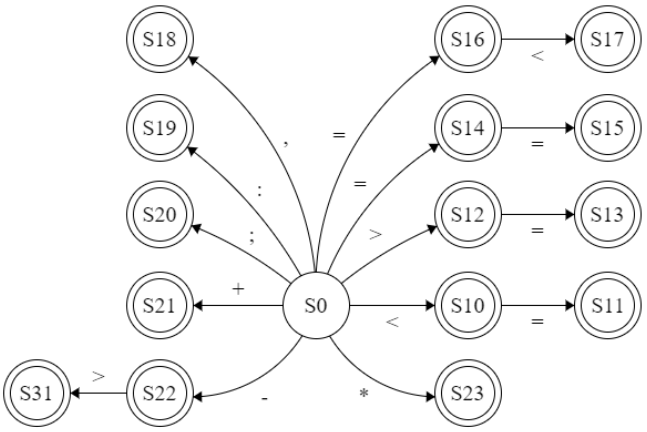

Figure 4 States S10 -S23, S31

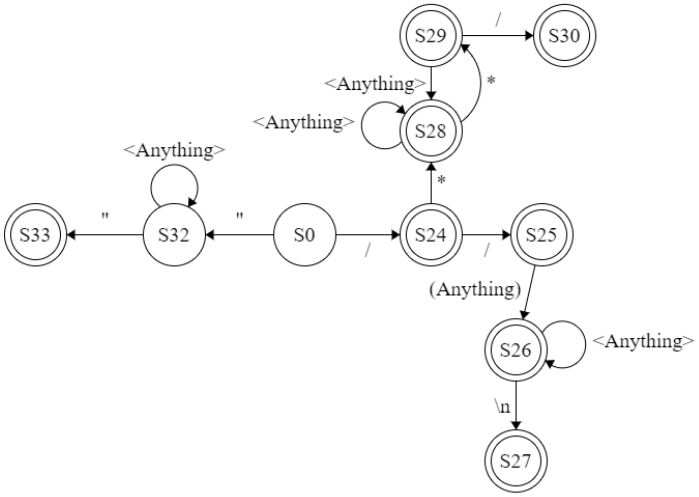

Figure 5 States S24-S30, S32- S33

The following is a list of tokens (tTOKEN) corresponding to each state:

S0	tSTART	S8	tDD	S16	tNOT	S24	tSLASH	S32	tSLITB
S1	tIDENTIFY	S9	tFLOAT	S17	tNOTE	S25	tCOMMENTB	S33	tSLIT
S2	tSPACE	S10	tLESSTHAN	S18	tCOMMA	S26	tCOMMENT	S34	ERROR
S3	tLCURLY	S11	tLORE	S19	tCOLON	S27	tCOMMENTNL		
S4	tRCURLY	S12	tGREATER	S20	tSCOLON	S28	tECOMMENTTB		
S5	tLBRACKET	S13	tGORE	S21	tADD	S29	tECOMMENTSTAR		
S6	tRBRACKET	S14	tEQUAL	S22	tSUBTRACT	S30	tECOMMENT		
S7	tDIGIT	S15	tRELA	S23	tSTAR	S31	tARROW		

Reserved Keywords

As already discussed, keyword are initialized by setReserved() and is used to transform any identifier matching a reserved keyword to the actual word. The following are the list of reserved words considered by the Lexer:

“bool” -> tBOOLEAN	“return” -> tRETURN	“and” -> tAND
“int” -> tINT	“if” -> tIF	
“float” -> tFLOAT	“else” -> tELSE	
“char” -> tCHAR	“for” -> tFOR	
“true” -> tTRUE	“while” -> tWHILE	
“false” -> tFALSE	“fn” -> tFUNCTION	
“let” -> tLET	“not” -> tN	
“print” -> tPRINT	“or” -> tOR	

Testing

As already mentioned, when developing the parser a utility function was created “void Parser::Testlexer()” which in conjunction with “printResult” is used to print every token in the format {Line | Actual Token | Token Name} the following are some results obtained when testing the Lexer.

```
Lexer input:

fn printLoop (n:int) -> int{
    for(let i:int=0; i<n; i=i+1)
    {
        print i;
    }
    return 0;
}

let y:int = printLoop(5);
```

Figure 6 Sample Input program

```
Lexer output:

<1: fn :tFUNCTION>
<1: printLoop :tIDENTIFY>
<1: ( :tLBRACKET>
<1: n :tIDENTIFY>
<1: : :tCOLON>
<1: int :tINT>
<1: ) :tRBRACKET>
<1: -> :tARROW>
<1: int :tINT>
<1: {  :tRCURLY>
<2: for :tFOR>
<2: ( :tLBRACKET>
<2: let :tLET>
<2: i :tIDENTIFY>
<2: : :tCOLON>
<2: int :tINT>
<2: = :tEQUAL>
<2: 0 :tDIGIT>
<2: ; :tSCOLON>
<2: i :tIDENTIFY>
<2: < :tLESSTHAN>
<2: n :tIDENTIFY>
<2: ; :tSCOLON>
<2: i :tIDENTIFY>
<2: = :tEQUAL>
<2: i :tIDENTIFY>
<2: + :tADD>
<2: 1 :tDIGIT>
<2: )  :tRBRACKET>
<3: {  :tRCURLY>
<4: print :tPRINT>
<4: i :tIDENTIFY>
<4: ;  :tSCOLON>
<5: }  :tLCURLY>
<6: return :tRETURN>
<6: 0 :tDIGIT>
<6: ;  :tSCOLON>
<7: }  :tLCURLY>
<9: let :tLET>
<9: y :tIDENTIFY>
<9: : :tCOLON>
<9: int :tINT>
<9: = :tEQUAL>
<9: printLoop :tIDENTIFY>
<9: ( :tLBRACKET>
<9: 5 :tDIGIT>
<9: ) :tRBRACKET>
<9: ;  :tSCOLON>
```

Figure 7 Output for figure 6

```
Lexer input:

let x:float = 20.23;
let y:float = 70.2;

if(x>y){
    print x;
}
else{
    x = y;
}
```

Figure 8 Sample input Program

```
Lexer output:

<1: let :tLET>
<1: x :tIDENTIFY>
<1: : :tCOLON>
<1: float :tFLOAT>
<1: = :tEQUAL>
<1: 20.23 :tFLOAT>
<1: ;  :tSCOLON>
<2: let :tLET>
<2: y :tIDENTIFY>
<2: : :tCOLON>
<2: float :tFLOAT>
<2: = :tEQUAL>
<2: 70.2 :tFLOAT>
<2: ;  :tSCOLON>
<4: if :tIF>
<4: ( :tLBRACKET>
<4: x :tIDENTIFY>
<4: > :tGREATER>
<4: y :tIDENTIFY>
<4: ) :tRBRACKET>
<4: {  :tRCURLY>
<5: print :tPRINT>
<5: x :tIDENTIFY>
<5: ;  :tSCOLON>
<6: }  :tLCURLY>
<7: else :tELSE>
<7: {  :tRCURLY>
<8: x :tIDENTIFY>
<8: = :tEQUAL>
<8: y :tIDENTIFY>
<8: ;  :tSCOLON>
<9: }  :tLCURLY>
```

Figure 9 Output for figure 8

```
Lexer input:

let x:&float = 20.23;

Lexer output:

<1: let :tLET>
<1: x :tIDENTIFY>
Error at line: 1, & is not a valid lexeme
```

Figure 10 Sample Lexer input and error being presented as expected

Lexer Limitations and Changes

Starting with the changes taken, instead of implementing a character type it was decided that instead strings are to be implemented, due to improved features strings can provide. For limitations, with regards to implementation while the groundwork was made for comments and extended comments by having them be present in the Transition Table, the full implementation of these where not completed.

Task 2 – Hand-crafted LL(k) parser

The Parser class

The following is a table of the list of functions/storage created for the parser class:

`Parser(std::vector<String> file);`	Start instantiation of Lexer and receive LL(1) and LL(2)
`~Parser();`	Default Destructor
`void Testlexer();`	Utility function used to test Lexer in Task 1
`void getNextToken();`	Parsers getNextToken() used to call Lexers token method to store LL(1) and LL(2)
`void XMLPass();`	Calls Task 3
`void SemanticPass();`	Calls Task 4
`void InterpretPass();`	Calls Task 5
`std::vector<std::shared_ptr<ASTStatement>> program;`	Stores the AST tree as a vector of shared pointers within program
`bool checkRelation(MyTokens Token);`	Checks if a relational operator is passed
`OP getRelation(MyTokens Token);`	Conversion between operator types
`bool checkAdditive(MyTokens Token);`	Checks if an additive operator is passed
`OP getAdditive(MyTokens Token);`	Conversion between operator types
`bool checkMultiplicative(MyTokens Token);`	Checks if a multiplicative operator is passed
`OP getMultiplicative(MyTokens Token);`	Conversion between operator types
`bool checkType(MyTokens Token);`	Checks if a valid type is passed
`Lexer* myObj;`	Instance of Lexer class to call getNextToken() from lexer to pass to parser's getNextToken()
`Token llOne;`	Stores LL(1)
`Token llTwo;`	Stores LL(2)

Additionally, to build the AST tree the following parsing functions where created which correspond to each EBNF grammar:

(Note: Formal Param and Formal Params are combined in the same method)

`std::shared_ptr<ASTProgram> BeginParse();`	〈Program〉 ::= { 〈Statement〉 }
`std::shared_ptr<ASTStatement> stateParsing();`	〈Statement〉 ::= 〈VariableDecl〉 ';' | 〈Assignment〉 ';' | 〈PrintStatement〉 ';' | 〈IfStatement〉 | 〈ForStatement〉 | 〈WhileStatement〉 | 〈RtrnStatement〉 ';'

	| 〈FunctionDecl〉 | 〈Block〉
`std::shared_ptr<ASTVariableDecl> variableDeclParsing();`	〈VariableDecl〉 ::= 'let' 〈Identifier〉 ':' 〈Type〉 '=' 〈Expression〉
`std::shared_ptr<ASTAssign> assignParsing();`	〈Assignment〉 ::= 〈Identifier〉 '=' 〈Expression〉
`std::shared_ptr<ASTBlock> blockParsing();`	〈Block〉 ::= '{' { 〈Statement〉 } '}'
`std::shared_ptr<ASTIdentify> identifyParsing();`	〈Identifier〉 ::= ('_' | 〈Letter〉) { '_' | 〈Letter〉 | 〈Digit〉 }
`std::shared_ptr<ASTPrintStatement> printParsing();`	〈PrintStatement〉 ::= 'print' 〈Expression〉
`std::shared_ptr<ASTReturnStatement> returnParsing();`	〈RtrnStatement〉 ::= 'return' 〈Expression〉
`std::shared_ptr<ASTIfStatement> ifParsing();`	〈IfStatement〉 ::= 'if' '(' 〈Expression〉 ')' 〈Block〉 ['else' 〈Block〉]
`std::shared_ptr<ASTWhileStatement> whileParsing();`	〈WhileStatement〉 ::= 'while' '(' 〈Expression〉 ')' 〈Block〉
`std::shared_ptr<ASTForStatement> forParsing();`	〈ForStatement〉 ::= 'for' '(' [〈VariableDecl〉] ';' 〈Expression〉 ';' [〈Assignment〉] ')' 〈Block〉
`std::shared_ptr<ASTExpression> simpleexpressionParsing();`	〈SimpleExpr〉 ::= 〈Term〉 { 〈AdditiveOp〉 〈Term〉 }
`std::shared_ptr<ASTExpression> subExpressionParsing();`	〈SubExpression〉 ::= '(' 〈Expression〉 ')'
`std::shared_ptr<ASTExpression> termParsing();`	〈Term〉 ::= 〈Factor〉 { 〈MultiplicativeOp〉 〈Factor〉 }
`std::shared_ptr<ASTExpression> factorParsing();`	〈Factor〉 ::= 〈Literal〉 | 〈Identifier〉 | 〈FunctionCall〉 | 〈SubExpression〉 | 〈Unary〉
`std::shared_ptr<ASTExpression> expressionParsing();`	〈Expression〉 ::= 〈SimpleExpr〉 { 〈RelationalOp〉 〈SimpleExpr〉 }
`std::shared_ptr<ASTUnary> unaryParsing();`	〈Unary〉 ::= ('-' | 'not') 〈Expression〉
`std::shared_ptr<ASTLiteral> literalParsing()`	〈Literal〉 ::= 〈BooleanLiteral〉 | 〈IntegerLiteral〉 | 〈FloatLiteral〉 | 〈CharLiteral〉
`std::shared_ptr<ASTFunctionDecl> functionParsing();`	〈FunctionDecl〉 ::= 'fn' 〈Identifier〉 '(' [〈FormalParams〉] ')' '->' 〈Type〉 〈Block〉
`std::shared_ptr<ASTFormalParam> formalParsing();`	〈FormalParams〉 ::= 〈FormalParam〉 { ',' 〈FormalParam〉 } && 〈FormalParam〉 ::=

	〈Identifier〉 ':' 〈Type〉
`std::shared_ptr<ASTFunctionCall> functionCallParsing();`	〈FunctionCall〉 ::= 〈Identifier〉 '(' [〈ActualParams〉] ')'

Handling LL(1) and LL(2)

When dealing with function calls and identifier parsing, an LL(2) is needed to verify if after a factor if a bracket is present or not. Thus LL(k) for this EBNF is k = 2. To achieve this, the parser calls for tokens from the lexer, in such a way that it always maintains an LL(1) token and an LL(2) token. LL(2) continues a look ahead till the end of file is reached thus signaling that no more tokens can't be obtained.

```
void Parser::getNextToken() //always look ahead one, use lexers getNextToken() to update parsers getNextToken();
{
    this->llOne = llTwo;

    if(!this->llTwo.end) //if at end of file dont update ll(2)
        this->llTwo = myObj->getNextToken();
    if(this->llTwo.end)
        this->EoF=true;

}
```

Figure 11 LL(k) implemented by always keeping a look ahead

The Abstract Syntax Tree

For this implementation, the (AST) was coded as a tree, being stored as a vector of shared pointers which point to a particular node's child. A class hierarchy was used to represent each AST which allows for storage for operators, types, identifier names etc. as well as allowing the visitor design patter to traverse these classes. Shared Pointers are used for ease of deletion, since no loops will be found within the tree, we can simply delete the starting ASTNode and the rest of the tree will henceforth be deleted, preventing any memory leaks from occurring.

The following is the structure of the hierarchy for each AST class starting from ASTNode:

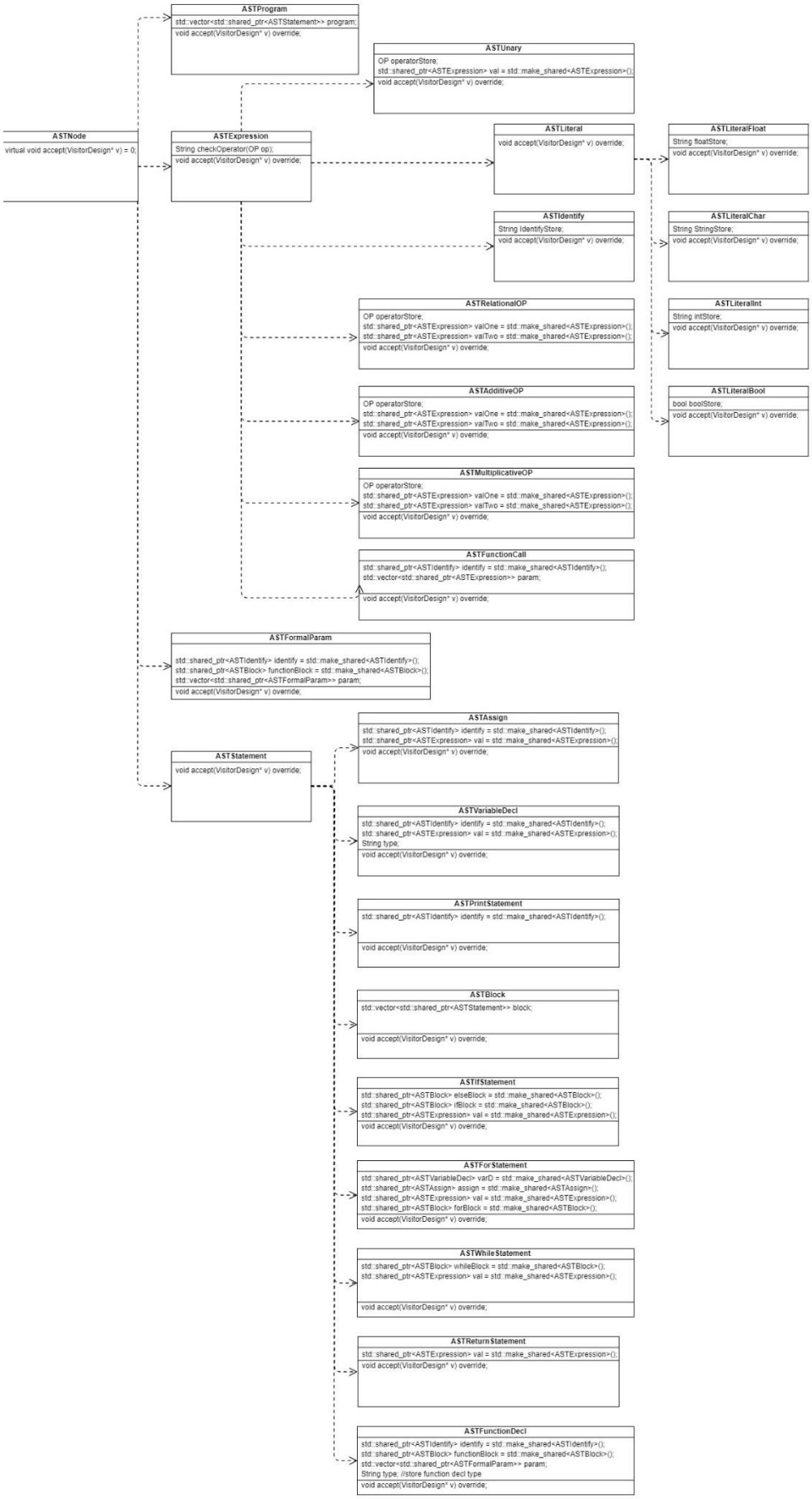

Figure 12 AST Tree class hierarchy

Testing

For testing, the XML pattern is used to check if the parser is building the tree correctly, thus for this section testing shall be carried to detect any syntactically incorrect programs.

```
Lexer input:

let x:int = 0;
print x

terminate called after throwing an instance of 'std::runtime_error'
  what():  Error at line 2 ; Expected
```

Figure 13 ; missing in line 2, identified correctly

```
Lexer input:

fn printLoop (n:12) -> int{
    for(let i:int=0; i<n; i=i+1)
    {
        print i;
    }
    return 0;
}

let y:int = printLoop(5);

terminate called after throwing an instance of 'std::runtime_error'
  what():  Error at line 1, receieved unknown type 12
```

Figure 14 12 passed instead of type; identified correctly

```
Lexer input:

let x:int = 20;
let y:int = 10;

if(x>y)
    print("X is greater than y");
 }

terminate called after throwing an instance of 'std::runtime_error'
  what():  Error at line 5, left curly bracket expected, received unexpected print
```

Figure 15 { not written after if; identified correctly

Task 3 – AST XML Generation Pass

Overview

After the AST visitor tree has been constructed, multiple visits can be taken to traverse the tree. This was achieved by using the Visitor design pattern that passes around a pointer through the tree and traversing it in the order it was instantiated in. The XML starts by being called in the main file and is called from the parser to be able to access the tree.

```
void Parser::XMLPass() //Starts the XML Pass on program
{
    auto xml = new XMLVisitorDesign();

    std::cout << "XML pass:" << std::endl;
    std::cout << "" << std::endl;

    std::cout << "<Program>" << std::endl;
    for(const std::shared_ptr<ASTStatement>& s : this->program)
    {
        s->accept( v: xml);
    }
    std::cout << "</Program>\n" << std::endl;

    delete xml;
}
```

Figure 16 XML pattern start

As denoted in figure 16 a pointer is created and is sent around each statement of the program till the end is reached and the pointer is deleted. The visitor design pattern is located in ‘VisitorDesign.hpp’ and ‘VisitorDesign’ inherits to the class ‘XMLVisitorDesing’, a integer is stored withing this class to keep track of the level of indent taken by the pattern denoted as ‘visited’, this is increased/decreased based on what hierarchy we are currently in and is used by the utility function ‘printindent()’ to print a tab for the length of visited. To integrate the pattern with the AST tree itself, a virtual acceptor was created in each AST class which accepts a VisitorDesign pointer. After calling accept, visit is then used to travers a particular node.

Testing

The following are a sample examples of syntactically correct code to showcase some XML generation.

```
Lexer input:

let x:float = 2.4;

XML pass:

<Program>
      <VariableDecl type=float>
         <Identify>x</Identify>
         <LiteralFloat>2.4</Literalfloat>
      </VariableDecl>
</Program>
```

Figure 17 Sample code with its respective XML generation

```
Lexer input:

fn AverageOfThree(x:float , y:float , z:float) -> float {
    return( x + y + z )/3.0;
}

XML pass:

<Program>
      <FunctionDecl type=float>
         <Identify>AverageOfThree</Identify>
         <FunctionDecl type=float>
            <Identify>x</Identify>
         </FunctionDecl>
         <FunctionDecl type=float>
            <Identify>y</Identify>
         </FunctionDecl>
         <FunctionDecl type=float>
            <Identify>z</Identify>
         </FunctionDecl>
         <Block>
            <Return>
               <Multiplicative type= "/" >
                  <Additive type= "+" >
                     <Identify>x</Identify>
                     <Additive type= "+" >
                        <Identify>y</Identify>
                        <Identify>z</Identify>
                     </Additive>
                  </Additive>
                  <LiteralFloat>3.0</Literalfloat>
               </Multiplicative>
            </Return>
         </Block>
      </FunctionDecl>
</Program>
```

Figure 18 Sample code with is respective XML generation

Task 4 – Semantic Analysis pass

Overview

Once the parsing of the AST Tree is complete, the visitor design pattern can once again be utilized to perform semantic analyses. Semantic Analyses needs to take care of performing type checking aswell as scope checking. To achieve these a symbol table was created and utilized throughout for this task. The symbol table functions by maintaining a stack which maintains scopes that are pushed and pop appropriately with the flow of code.

The stack itself is a vector of scopes that houses a key value pair with identifier names and a scope struct respectively. A scope struct was created to store all the relevant information needed such as variable values (Utilized in Task 5), function names and variable types. The stack was given three main features to function like a regular stack such as pushing, popping and peeking.

The Scope Struct

`String myval`	Stores values within a scope (Utilized in Task 5)
`std::vector<ASTFunctionDecl*> myFuncs;`	Stores a vector of function names within a scope
`String type;`	Stores a type within a scope (Used for type checking)

The Symbol Table Class

- *Scope is typdef as <String, Scope>*

`SymTable();`	Default constructor
`~SymTable();`	Default destructor
`void push();`	Push scope on top of a stack
`void pop();`	Pop scope from the stack
`Scopes* top();`	Peek the top of scope stack
`void declaration(ASTFunctionDecl* node);`	Declares a new function by pushing on top of the scope stack
`void declaration(ASTVariableDecl* node);`	Declares a new variable by pushing it on top of the scope stack
`void declaration(ASTFormalParam* node);`	Declares a new function parameter by pushing it on top of the stack
`void declaration(const std::shared_ptr<ASTFormalParam>& node);`	Special function param method needed when dealing with a shared pointer instead of regular
`bool checkDeclaration(const String& var);`	Checks if variables have already been declared
`bool checkDeclaration(const String& var, std::vector<String>* type);`	Checks if function declarations have already been declared
`std::map<String, Scope>::iterator checkVariableType(const String& var);`	Goes through the stack and attempts to return the type of a particular variable
`bool hasReturn{};`	Flag to check if a function has a return type
`std::vector<Scopes> myStack`	Stack of a map of scopes {String | Type}

Type checking

When performing type checking a strict rule is kept by MiniLang that does not perform type casting, thus if an integer is assigned to a float for example, Semantic Analysis would report this as an error.

Function returns

Semantic analysis needs to ensure that functions always have a return type, to enable this a flag is maintained which is set to true only when a return node is visited. If the function block is traversed and the return flag is still false, an appropriate error is reported. Returns also checks if the return type is also valid.

Conditionals

For the conditionals IF, FOR and WHILE, the conditional statement is checked to ensure that this matches if a conditional statement is indeed a Boolean/Relational Operator.

Testing

When ever the semantic analyzer is tested using a valid program no errors are detected thus the following will be testing if the analyzer can detect any semantically incorrect programs.

```
Lexer input:

print x;

XML pass:

<Program>
      <Print>
         <Identify>x</Identify>
      </Print>
</Program>

Semantic pass:

terminate called after throwing an instance of 'std::runtime_error'
  what():  Declaration error: x has not been declared

Process finished with exit code 3
```

Figure 19 x was not declared; correctly identified

```
Lexer input:

let x:int = 5 + "20";

XML pass:

<Program>
      <VariableDecl type=int>
         <Identify>x</Identify>
         <Additive type= "+" >
            <LiteralInt>5</LiteralInt>
            <LiteralString>20</LiteralString>
         </Additive>
      </VariableDecl>
</Program>

Semantic pass:

terminate called after throwing an instance of 'std::runtime_error'
  what():  TypeMatch error: int and string do not match

Process finished with exit code 3
```

Figure 20 Attempting addition of int and string; correctly identified

```
Lexer input:

fn addNums (n:int, m:int) -> int{
  return n+m;
}

fn addNums (n:int, m:int) -> int{
  return n+m;
}

let x:int = addNums(5,10);
print(x);

Semantic pass:

terminate called after throwing an instance of 'std::runtime_error'
  what():  Declaration error: addNums has already been declared

Process finished with exit code 3
```

Figure 21 function addNums present twice; correctly identified

```
Lexer input:

if("true"){
print("Works accordingly");
}

XML pass:

<Program>
      <If>
<LiteralString>true</LiteralString>
         <Block>
            <Print>
               <LiteralString>Works accordingly</LiteralString>
            </Print>
         </Block>
      </If>
</Program>

Semantic pass:

terminate called after throwing an instance of 'std::runtime_error'
  what():  Relational error: expected Relation, string given.

Process finished with exit code 3
```

Figure 22 String passed as a relational operator; correctly identified

```
Lexer input:

fn printLoop (n:int) -> int{
    for(let i:int=0; i<n; i=i+1)
    {
        print i;
    }
    return 0;
}

let y:float = printLoop(5.5);

Semantic pass:

terminate called after throwing an instance of 'std::runtime_error'
  what():  Function: printLoop has not been defined with type: float
```

Figure 23 attempting to call printLoop with float param; correctly identified

Task 5 – Interpreter Execution Pass

Overview

Once the code has been properly fully validated by the Lexer, Parser and Semantic Analysis the interpreter can now go through the code to generate interpreted results from print statements. Since the interpreter builds on what has already been established in Task 3 and Task 4, the exact same layout was taken. A new class deriving from VisitorDesign was written that stores the following information and are used when printing the values to the user:

`float actualFloat{};`	Stores the actual float value
`int actualInt{};`	Stores the actual int value
`String actualString;`	Stores the actual string value
`bool actualBool{};`	Stores the actual Boolean value

The Symbol Table

The symbol table is once again used, and the following assignment functions were added to class SymTable to allow storage of variables within scopes:

```
void assignment(const String& var, const String& actualValue)
void assignment(const String& var, float actualValue);
void assignment(const String& var, int actualValue);
void assignment(const String& var, bool actualValue);
```

Within struct Scope ‘String myval’ has also been added to store values in scopes when assigned. All types are type casted to string for storage and these are type casted back to their original types when ever they are used again.

Visit Classes

Visit classes were changed to reflect the behavior to be carried out by the interpreter. AST’s loops such as for and while repeatedly call their respective visitors till a Boolean flag representing the end of a loop is reached. Example: AST For repeatedly calls visit for block, updates its inner assignment and updates any needed value updates. Whenever ASTPrintStatement is called the value stored in a particular variable or the value itself are simply print to screen. ASTRelationalOP, ASTUnary, ASTAdditiveOP and ASTMultiplicativeOP are used to perform their respective calculations depending on the type being stored within the node. Finally, ASTFunctionCall attempts to get any params being pushed to a particular function and match them to the appropriate function for execution.

Testing

To verify that the expected output is taken by the interpreter, expected cases will be taken and verified. Code is also tested by the Parser and Semantic Analysis to verify if code is also semantically correct. The following function is used in the main by the parser to start the call for each statement with the program to start the interpreter traversal:

```
void Parser::InterpretPass()
{
    auto Interpret = new InterpreterPass();

    Interpret->myTable.push();

    std::cout << "Interpreter pass:" << std::endl;
    std::cout << "" << std::endl;

    for(const std::shared_ptr<ASTStatement>& s : this->program)
    {
        s->accept( v: Interpret);
    }

    Interpret->myTable.pop();

    delete Interpret;
}
```

Figure 24 Interpeter pass start

```
Lexer input:

print("Hello world!");

XML pass:

<Program>
      <Print>
         <LiteralString>Hello world!</LiteralString>
      </Print>
</Program>

Semantic pass:

Semantic pass completed without error.

Interpreter pass:

Hello world!

Process finished with exit code 0
```

Figure 25 Expected: Hello World! is to be printed by the interpreter. Actual: Hello world! was successfully outputted.

```
Lexer input:

fn XGreaterY (x:int, y:int) -> bool{
if(x>y){
    return true;
}
else{
    return false;
}
}

print XGreaterY(5, 10);

XML pass:

<Program>
      <FunctionDecl type=bool>
         <Identify>XGreaterY</Identify>
         <FunctionDecl type=int>
            <Identify>x</Identify>
         </FunctionDecl>
         <FunctionDecl type=int>
            <Identify>y</Identify>
         </FunctionDecl>
         <Block>
            <If>
               <Relational type= ">" >
                  <Identify>x</Identify>
                  <Identify>y</Identify>
               </Relational>
               <Block>
                  <Return>
                     <LiteralBool>1</LiteralBool>
                  </Return>
               </Block>
            </If>
            <Else>
               <Block>
                  <Return>
                     <LiteralBool>0</LiteralBool>
                  </Return>
               </Block>
            </Else>
         </Block>
      </FunctionDecl>
      <Print>
         <FunctionCall>
            <Identify>XGreaterY</Identify>
            <LiteralInt>5</LiteralInt>
            <LiteralInt>10</LiteralInt>
         </FunctionCall>
      </Print>
</Program>

Semantic pass:

Semantic pass completed without error.

Interpreter pass:

False
```

Figure 26 Function call to check if 5 is greater than 10, False is printed as expected.

```
Lexer input:

fn printLoop (n:int) -> int{
    for(let i:int=0; i<n; i=i+1)
    {
        print i;
    }
    return 0;
}

let y:int = printLoop(5);

XML pass:

<Program>
      <FunctionDecl type=int>
         <Identify>printLoop</Identify>
         <FunctionDecl type=int>
            <Identify>n</Identify>
         </FunctionDecl>
         <Block>
            <For>
               <VariableDecl type=int>
                  <Identify>i</Identify>
                  <LiteralInt>0</LiteralInt>
               </VariableDecl>
               <Relational type= "<" >
                  <Identify>i</Identify>
                  <Identify>n</Identify>
               </Relational>
               <Assign>
                  <Identify>i</Identify>
                  <Additive type= "+" >
                     <Identify>i</Identify>
                     <LiteralInt>1</LiteralInt>
                  </Additive>
               </Assign>
            </For>
            <loop>
               <Block>
                  <Print>
                     <Identify>i</Identify>
                  </Print>
               </Block>
            </loop>
            <Return>
               <LiteralInt>0</LiteralInt>
            </Return>
         </Block>
      </FunctionDecl>
      <VariableDecl type=int>
         <Identify>y</Identify>
         <FunctionCall>
            <Identify>printLoop</Identify>
            <LiteralInt>5</LiteralInt>
         </FunctionCall>
      </VariableDecl>
</Program>

Semantic pass:

Semantic pass completed without error.

Interpreter pass:

0
1
2
3
4

Process finished with exit code 0
```

Figure 27 Code to check if loop correctly prints a counter from 0 to one less of the number specified. Output is as expected.

Plagiarism Declaration Form

Plagiarism is defined as "*the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines*" (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

I, the undersigned, declare that the report submitted is my work, except where acknowledged and referenced. I understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

Kian Parnis	CPS2000	20/6/2022
Student's full name	**Study-unit code**	**Date of submission**

Title of submitted work: CPS2000 Compiler Report

Student's Signature