Name: **Jamie Grech**
ID: **123499M**
Course: **B.Sc. Information Technology (Hons.) (Artificial Intelligence)**

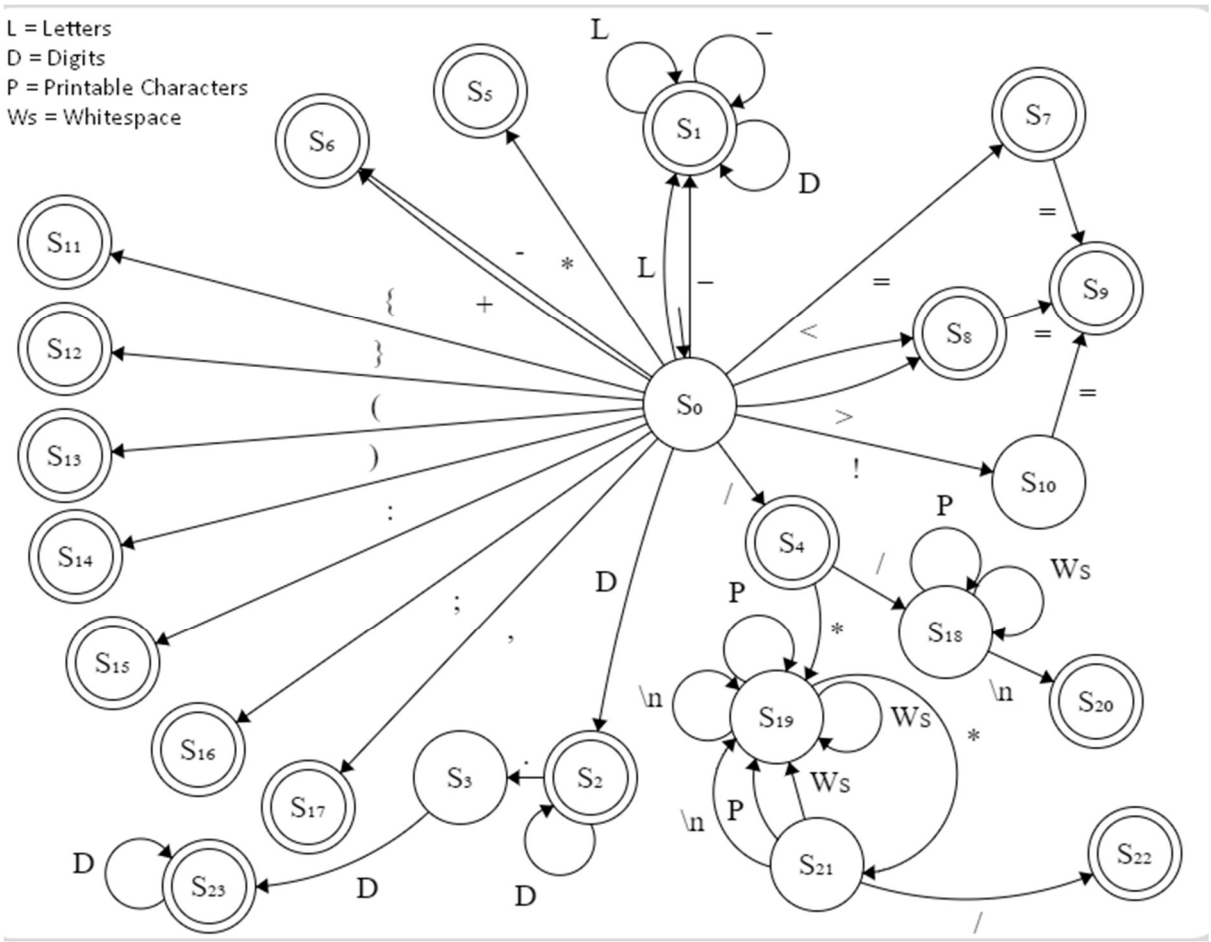
Faculty of Information and Communication Technology
Department of Artificial Intelligence

Study-unit: **Compiler Theory and Practice**
Code: **CPS2000**

Contents

Task 1 - A table-driven lexer

The role of the lexer in the compilation process is to convert the input program into a series of tokens, each having a type and a value. A table-driven lexer does this with the use of a transition table to define behaviour from one character to another. A classification table is also used to group certain characters together for cases where a large group of different characters each result in the same behaviour, such as grouping all alphabetic characters together or all printable characters together. The DFA below is the DFA I have designed based on the token types I have determined to be ideal for lexical analysis of the MiniLang language.



The class Token is used to represent these tokens and has two main properties: type of type TOK_TYPE which is an enum whose values can be seen on the next page and value which is simply a string of the lexeme the token is representing.

The token types identified by the lexer are as follows.

```
enum TOK_TYPE{
        TOK_TYPE_DECL = 1,
        TOK_BOOL_LIT = 2,
        TOK_INT_LIT = 3,
        TOK_FLOAT_LIT = 4,
        TOK_IDENTIFIER = 5,
        TOK_MULT_OP = 6,
        TOK_ADD_OP = 7,
        TOK_REL_OP = 8,
        TOK_UNARY_OP = 9,
        TOK_ASSIGNMENT_OP = 10,
        TOK_VAR_DECL = 11,
        TOK_PRINT = 12,
        TOK_RETURN = 13,
        TOK_IF = 14,
        TOK_FOR = 15,
        TOK_FUNC_DECL = 16,
        TOK_OPEN_SCOPE = 17,
        TOK_CLOSE_SCOPE = 18,
        TOK_OPEN_PARENTHESES = 19,
        TOK_CLOSE_PARENTHESES = 20,
        TOK_COLON = 21,
        TOK_SEMICOLON = 22,
        TOK_COMMA = 23,
        TOK_ELSE = 24,
        TOK_LINE_COMMENT = 25,
        TOK_BLOCK_COMMENT = 26,
        TOK_EOF = 27,
        TOK_SYNTAX_ERR = -1
}; //defining possible token type states
```

Not all of these token types are identified directly from the transition table, keywords are identified by a ‘classifyIdentifier’ function which can be seen below. This function takes a string classified as an identifier as input and compares it to all the keywords in the language and returns the matching token type if one of these keywords is identified. Otherwise it returns it as a normal identifier token.

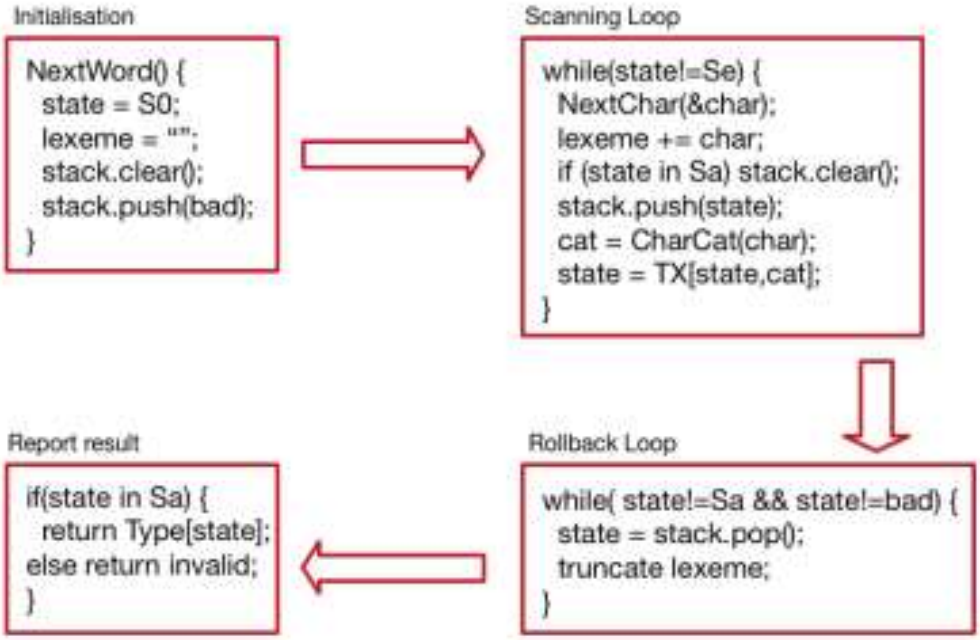
```
if(identifier == "float" || identifier == "int" || identifier == "bool") return TOK_TYPE_DECL;
if(identifier == "true" || identifier == "false") return TOK_BOOL_LIT;
if(identifier == "and") return TOK_MULT_OP;
if(identifier == "or") return TOK_ADD_OP;
if(identifier == "not") return TOK_UNARY_OP;
if(identifier == "var") return TOK_VAR_DECL;
if(identifier == "print") return TOK_PRINT;
if(identifier == "return") return TOK_RETURN;
if(identifier == "if") return TOK_IF;
if(identifier == "else") return TOK_ELSE;
if(identifier == "for") return TOK_FOR;
if(identifier == "fn") return TOK_FUNC_DECL;
return TOK_IDENTIFIER;
```

With the previously attached DFA figure in mind, the map below shows what token type each state represents.

```
unordered_map<int, TOK_TYPE> final_states = {
        {1, TOK_IDENTIFIER},
        {2, TOK_INT_LIT},
        {4, TOK_MULT_OP},
        {5, TOK_MULT_OP},
        {6, TOK_ADD_OP},
        {7, TOK_ASSIGNMENT_OP},
        {8, TOK_REL_OP},
        {9, TOK_REL_OP},
        {11, TOK_OPEN_SCOPE},
        {12, TOK_CLOSE_SCOPE},
        {13, TOK_OPEN_PARENTHESES},
        {14, TOK_CLOSE_PARENTHESES},
        {15, TOK_COLON},
        {16, TOK_SEMICOLON},
        {17, TOK_COMMA},
        {20, TOK_LINE_COMMENT},
        {22, TOK_BLOCK_COMMENT},
        {23, TOK_FLOAT_LIT}
};
```

The functions used by the lexer are as follows:

- The constructor Lexer(string file_address) takes the address of the input text file and copies it to a program string, while also counting the amount of lines in the program and storing it in the total_lines variable. This is used to identify when the end of the file is reached when the lexer is running in order to return and EOF token.
- void clearStack() which clears the stack of previously visited states when reading the current token. This function is executed every time a final state is visited when reading a token.
- char nextChar() reads and returns the current character in the input program, increments the line cursor if it is a \n character and increments the char cursor to point to the next character.
- TOK_TYPE classifyIdentifier(string identifier) takes a string which was classified as an identifier token as input and checks if it matches one of the keyword tokens and returns the respective TOK_TYPE if so. Otherwise it simply returns TOK_IDENTIFIER.
- int getNextState(char next) uses the classification table to identify what type of character the next character is then uses that information to return the index of the respective state which should be moved to from the current state given that character.
- Token getNextToken() reads the next token from the input program using the aforementioned functions by keeping a stack of the states visited while reading the current token. When a character is encountered while reading a token which causes an error transition, the token is rolled back character by character until the last lexeme which resulted in an accepting state is reached, then the respective token is returned. This implementation is based on the pseudocode from the Lexical Analysis notes covered in class, attached on the following page.


Initialisation
NextWord() {
state = S0;
lexeme = "";
stack.clear();
stack.push(bad);
}
Scanning Loop
while(state!=Se) {
NextChar(&char);
lexeme += char;
if (state in Sa) stack.clear();
stack.push(state);
cat = CharCat(char);
state = TX[state,cat];
}
Rollback Loop
while(state!=Sa && state!=bad) {
state = stack.pop();
truncate lexeme;
}
Report result
if(state in Sa) {
return Type[state];
else return invalid;
}

Task 2 - Hand-crafted recursive descent parser

The parser is used to read the tokens as produced by the lexer and build the structure of the program in the form of an abstract syntax tree. The structure for the abstract syntax tree is defined by a series of classes which are all subclasses of abstract classes ASTFactorNode, ASTExpressionNode, ASTStatementNode or ASTOp. These superclasses are in turn subclasses of the abstract superclass ASTNode. The AST is built by the buildAST() method of the parser as shown below which returns the AST as vector of ASTNodes.

```
vector<ASTNode *> *Parser::buildAST() {
    vector<ASTNode*> *AST = new vector<ASTNode*>;
    nextToken = lexer->getNextToken();
    moveToNext();
    ASTNode *currentNode;
    while(currentToken.getType() != TOK_EOF){
        cout << "Parsing node: " << currentToken.getValue() << endl;
        currentNode = parse();
        if(currentNode != nullptr){
            AST->push_back(currentNode);
        }else{
            cout << "Parsing error occured, parser terminating." << endl;
            return nullptr;
        }
    }
    return AST;
}
```

As can be seen from the above code snippet, the parser makes use of two Token variables, nextToken and currentToken, which are updated by the moveToNext() function.

```
void Parser::moveToNext() {
    currentToken = nextToken;
    nextToken = lexer->getNextToken();
}
```

The currentToken is the most often used variable, whilst nextToken is used in situations where the parser requires knowledge of the token after the one currently being parsed in order to determine the behaviour. An example of this would be when the parser is supposed to parse a factor from the current token which is an identifier. The parser would then check if the next token is an open parenthesis (‘(‘) in which case it would parse it as a function call, and parse it as a regular identifier otherwise. The parse() method checks the current token type and attempts to parse a statement accordingly. If the current token type doesn’t match the beginning of any statement, the parse function returns a nullptr thus causing the parser to terminate. This convention is used throughout the parser implementation where any unexpected behaviour results in a nullptr being returned and any nodes being returned as a nullptr also cause the nodes they are meant to be a part of, if any, to be returned as nullptr, thus causing termination of the parser. Whenever a nulltpr is returned a short message is outputted describing why to allow the user to better trace where the error in their code is.

The aforementioned parse() method can be seen below.

```
ASTStatementNode *Parser::parse() {
    switch(currentToken.getType()){
        case TOK_VAR_DECL: return parseVariableDeclStatement();
        case TOK_PRINT: return parsePrintStatement();
        case TOK_RETURN: return parseReturnStatement();
        case TOK_IF: return parseIfStatement();
        case TOK_FOR: return parseForStatement();
        case TOK_IDENTIFIER: return parseAssignmentStatement();
        case TOK_FUNC_DECL: return parseFunctionDeclStatement();
        case TOK_OPEN_SCOPE: return parseBlockStatement();
        default: return nullptr;
    }
}
```

Each of the parse___Statement methods then uses other parse methods such as parseExpression() to parse the relevant nodes needed to make up the particular statement.

As can be seen from the first code snippet for the parser, detailing the buildAST method, the program is parsed through statement by statement using the parse method and a number of other methods (one method for every ASTNode non-abstract subclass, two methods for parseActualParams, parsaeFormalParams and parseBlockStatement specifically) until either one statement is returned as a nullptr or the EOF token is reached, in which case the parser will return the vector of nodes parsed. This vector can then be used by the visitors to perform their respective functionality.

The parseExpression, parseSimpleExpression and parseTerm methods make recursive calls in order to parse the second ASTExpressionNode as opposed to parseExpression calling parseSimpleExpression, parseSimpleExpression calling parseTerm and parseTerm calling parseFactor respectively. This is done so that expressions with subsequent operators of the same time can be parsed correctly. This can be seen in the below code snippets

```
ASTExpressionNode *Parser::parseExpression() {
    ASTExpressionNode *simpleExpression = parseSimpleExpression();
    if(currentToken.getType() == TOK_REL_OP){
        ASTRelOp *relOp = parseRelOp();
        ASTExpressionNode *simpleExpression2 = parseExpression();
```

```
ASTExpressionNode *Parser::parseSimpleExpression() {
    ASTExpressionNode *term = parseTerm();
    if(currentToken.getType() == TOK_ADD_OP){
        ASTAddOp *addOp = parseAddOp();
        ASTExpressionNode *term2 = parseSimpleExpression();
```

```
ASTExpressionNode *Parser::parseTerm() {
    ASTExpressionNode *factorNode = parseFactor();
    if(currentToken.getType() == TOK_MULT_OP){
        ASTMultOp *multOp = parseMultOp();
        ASTExpressionNode *factorNode2 = parseTerm();
```

Task 3 - AST XML Generation Pass

The abstract class Visitor was created as a superclass for the classes XMLGenerationVisitor, SemanticAnalysisVisitor and InterpreterExecutionVisitor, which will be used for this task, Task 4 and Task 5 respectively. This class contains virtual visit methods for all non-abstract ASTNode subclasses as can be seen below.

```
class Visitor {
public:
    virtual void visit(ASTBool *node) = 0;
    virtual void visit(ASTFloat *node) = 0;
    virtual void visit(ASTIdentifierNode *node) = 0;
    virtual void visit(ASTFunctionCall *node) = 0;
    virtual void visit(ASTInt *node) = 0;
    virtual void visit(ASTSubExpression *node) = 0;
    virtual void visit(ASTUnaryNode *node) = 0;
    virtual void visit(ASTActualParams *node) = 0;
    virtual void visit(ASTAddOp *node) = 0;
    virtual void visit(ASTExpression *node) = 0;
    virtual void visit(ASTSimpleExpression *node) = 0;
    virtual void visit(ASTTerm *node) = 0;
    virtual void visit(ASTFormalParam *node) = 0;
    virtual void visit(ASTFormalParams *node) = 0;
    virtual void visit(ASTMultOp *node) = 0;
    virtual void visit(ASTRelOp *node) = 0;
    virtual void visit(ASTType *node) = 0;
    virtual void visit(ASTUnaryOp *node) = 0;
    virtual void visit(ASTAssignmentStatement *node) = 0;
    virtual void visit(ASTBlockStatement *node) = 0;
    virtual void visit(ASTForStatement *node) = 0;
    virtual void visit(ASTFunctionDeclStatement *node) = 0;
    virtual void visit(ASTIfStatement *node) = 0;
    virtual void visit(ASTPrintStatement *node) = 0;
    virtual void visit(ASTReturnStatement *node) = 0;
    virtual void visit(ASTVariableDeclStatement *node) = 0;
};
```

The XMLGenerationVisitor overrides each of these functions and also has a string indent used to store the \t characters according to the level of the current node being outputted in the AST, and a constructor which simply initializes the indent string to an empty string. The visit function for the most basic nodes such as factors and operators simply consist of a string printing the value of the node enclosed within the respective xml tag as can be seen below

```
void XMLGenerationVisitor::visit(ASTBool *node) {
    cout << indent << "<Bool>" << boolalpha << node->boolValue << "</Bool>" << endl;
}
```

In this code snippet the boolalpha format flag is used to display the Boolean value as ‘true’ or ‘false’ instead of 1 or 0.

```
void XMLGenerationVisitor::visit(ASTFunctionCall *node) {
    cout << indent << "<FunctionCall>" << endl;
    indent += "\t";
    node->identifierNode->Accept(this);
    node->actualParams->Accept(this);
    indent.pop_back();
    cout << indent << "</FunctionCall>" << endl;
}
```

For nodes like the function call node shown above which are made up of other nodes, The XML Tag for the node is first opened, then an indent layer is added, the Accept methods of its component nodes are called then the indent added indent layer is removed and the XML Tag is closed. The accept function of every node simply calls the visitor (which is passed as a parameter to the function) to perform its visit on for that node.

```
void XMLGenerationVisitor::visit(ASTActualParams *node) {
    cout << indent << "<ActualParams>" << endl;
    indent += "\t";
    for (ASTExpressionNode *param : *node->parameters) {
        param->Accept(this);
    }
    indent.pop_back();
    cout << indent << "</ActualParams>" << endl;
}
```

In cases like the ActualParams node where one of the node's components is a vector of other nodes, the vector is looped through and the Accept method of each node is called as can be seen in the above snippet.

```
void XMLGenerationVisitor::visit(ASTForStatement *node) {
    cout << indent << "<ForStatement>" << endl;
    indent += "\t";
    if (node->variableDeclStatement->type->value != "") node->variableDeclStatement->Accept(this);
    node->expression->Accept(this);
    if (node->assignmentStatement->identifierNode->value != "") node->assignmentStatement->Accept(this);
    node->forBlock->Accept(this);
    cout << indent << "</ForStatement>" << endl;
}

void XMLGenerationVisitor::visit(ASTFunctionDeclStatement *node) {
    cout << indent << "<FunctionDeclStatement>" << endl;
    indent += "\t";
    node->identifierNode->Accept(this);
    if (!node->formalParams->formalParams->empty()) node->formalParams->Accept(this);
    node->type->Accept(this);
    node->blockStatement->Accept(this);
    indent.pop_back();
    cout << indent << "</FunctionDeclStatement>" << endl;
}
```

For nodes which have optional components such as the ForStatement nodes with the optional variable declaration and assignment statements in their arguments and the FunctionDeclStatement nodes with the optional parameters, if statements are used so as to not output the respective nodes if they are not present.

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

Upon visiting the function declaration node for the above code snippet from a program in the MiniLang language, the XMLGenerationVisitor output would be as follows.

```
<FunctionDeclStatement>
        <IdentifierNode>Square</IdentifierNode>
        <FormalParams>
                <FormalParam>
                        <IdentifierNode>x</IdentifierNode>
                        <Type>float</Type>
                </FormalParam>
        </FormalParams>
        <Type>float</Type>
        <BlockStatement>
                <ReturnStatement>
                        <Term>
                                <IdentifierNode>x</IdentifierNode>
                                <MultOp>*</MultOp>
                                <IdentifierNode>x</IdentifierNode>
                        </Term>
                </ReturnStatement>
        </BlockStatement>
</FunctionDeclStatement>
```

Task 4 - Semantic Analysis Pass

4.1 – The Value Class

The first step of the semantic analysis process was creating a class (Value) to allow us to map identifier names to different types of values (int, float and bool) in the same map. This class makes use of an enum TYPE with values INT, FLOAT and BOOL to store the declared type of the associated variable. The properties of the value class can be seen in the code snippet below

```
enum TYPE{INT = 1, FLOAT = 2, BOOL = 3};

class Value{
public:
    TYPE identifierType;
    int *intval;
    float *floatval;
    bool *boolval;
    vector<Value*> *funcparams;
    Value(int intval);
    Value(float floatval);
    Value(bool boolval);
    Value(string type);
    Value(string type, vector<Value*> *funcparams);
    ~Value();

};
```

The first 3 constructors take an int, float, or bool as a parameter, set the identifierType to the respective Type and set the respective value pointer to the value passed as a parameter, while initializing all the other pointers as nullptrs, as can be seen below.

```
Value::Value(int intval) {
    identifierType = INT;
    *this->intval = intval;
    boolval = nullptr;
    floatval = nullptr;
    funcparams = nullptr;
}
```

Pointers were used as a failsafe for situations where an attempt could possibly be made to access the value of a non-matching type or to access the value of a parameter of a declared function. Pointers allow these to be set to nullptrs instead of undeclared values and thus make it easier to work around them and predict behaviour in such fringe cases.

4.2 – The Symbol Table

To represent the symbol table, I decided to make use of a vector of maps, which map a string (the variable/function identifier) to an object of type Value explained above which stores the type of the variable/function and possibly a value later on in the interpreter execution pass. The methods I created for manipulation of the symbol table are as follows:

```
SymbolTable();

~SymbolTable();

void openscope();

void closescope();

int lookup(string identifier);

int insertvar(string identifier, Value *value);

int assignvar(string identifier, Value *value);

bool checkValidFunctionCall(string identifier, vector<Value*> *parameters);

Value *getValue(string identifier);
```

- The openscope method creates a new map and adds it to the end of the vector, representing a new layer/scope
- The closescope method pops the last layer/scope from the vector.
- The lookup method checks if an identifier is already represented in the symbol table and returns the index in the vector of the layer where the identifier is represented. If the identifier is not already represented in the table, this method returns -1
- The insertvar first uses lookup to check if the identifier parameter is already represented. If not, it adds it to the last scope in the table mapped to the Value passed as a parameter and returns 1. If the identifier is already represented it returns -1
- The assignvar method again uses lookup to check if the identifier parameter is already represented in the table. If it is, it then checks if the type of the variable matches that of the variable passed as a parameter. If so it updates the value of the variable in the table and returns 1. If the variable is not already in the table or does not match the type of the variable passed as a parameter, it returns -1.
- checkValidFunctionCall takes an identifier which represents a function and a vector of Values containing the types of the parameters of the function, and checks if a function with matching name and parameter types exists in the table. If so it returns true, otherwise returns false. This is used to validate the format of a function call with the original function declaration.
- getValue returns the Value object mapped to a given identifier. If no variable exists mapped to that identifier it returns a nullptr.
- The constructor of the symbol table initializes the vector of maps and opens the first scope.

4.3 – The SemanticAnalysisVisitor

The semantic analysis visitor class has the following properties:

- An overridden method for each of the methods declared in the Visitor superclass.
- An instance of the SymbolTable class used to build the symbol table for the program as the visitor goes through it.
- A constructor which initializes the symbol table and destructor which deletes it.
- Two variables storing TYPE (as defined in the Value class) enums:
    - currentType stores the type of the last variable visited by the visitor.
    - returnType stores the declared return type of the declared function when going through the function body in order to ensure that return statements in the function match the function's return type. If a return statement is found in a function body with a type that doesn't match the return type, an error message indicating this is outputted and the program exits.
- A bool variable returnFlag which is initialized as false in the visitor constructor. This flag is set to true if a return statement is found when parsing through a function body. If this flag is still false at the end of a function body an error is outputted by the visitor indicating that a function is present without a return statement and the program exits. At the end of the visit method for function declaration statements, this flag is set back to false so that it may be reused for future functions.
- A map which maps strings to objects of type Value called parameters. This map is used to store the identifiers and types of the parameters in a function declaration so that these may then be stored in the function body's scope in the symbol table and not in the outer scope where the function declaration itself resides. This allows variable identifiers used for a parameter of a function to be reused within the scope where the function is declared, as well as to be used for parameters of other functions declared in the same scope.

The semantic errors which this visitor can detect in a program and cause it to terminate the program are as follows:

- Visiting a node of type ASTIdentifierNode which is not mapped to an existing variable in the symbol table. (Caused when a variable which is not in scope is accessed)

```
void SemanticAnalysisVisitor::visit(ASTIdentifierNode *node) {
    if(symbolTable->getValue(node->value) == nullptr){
        cout << "Identifier  " << node->value << " not declared" << endl;
        exit(EXIT_FAILURE);
```

- If a function is called without a function with a matching function/method signature existing in the symbol table (Determined using checkValidFunctionCall method mentioned in section 4.2)

```
if(symbolTable->checkValidFunctionCall(node->identifierNode->value, parametertypes))
    currentType = symbolTable->getValue(node->identifierNode->value)->identifierType;
else{
    cout << "Invalid function call" << endl;
    exit(EXIT_FAILURE);
}
```

- If a unary node with operator 'not' has an expression of type float or int or if a unary node with operator '-' has an expression of type bool.

```
if(node->unaryOp->value == "not"){
    if(currentType != BOOL){
        cout << "Error: unary operator 'not' can only be applied to bools" << endl;
        exit(EXIT_FAILURE);
    }
}else{
    if(currentType == BOOL){
        cout << "Error: unary operator '-' can only be applied to ints and floats" << endl;
        exit(EXIT_FAILURE);
    }
}
```

- If the variables on both sides of an operator in an Expression/SimpleExpression/Term are not of the same type

```
node->simpleExpression1->Accept(this);
TYPE type1 = currentType;
node->simpleExpression2->Accept(this);
TYPE type2 = currentType;

if(type1 != type2){
    cout << "Error: variables in an expression must be of the same type" << endl;
    exit(EXIT_FAILURE);
}
```

- If the less than or greater than operators are used to compare variables of type bool.

```
if(exprOp == "<" || exprOp == ">"){
    if(type1 == BOOL){
        cout << "Invalid expression: " << exprOp << " can only be applied to ints and floats" << endl;
        exit(EXIT_FAILURE);
    }
```

- If the +, -, / or * operators are used on variables of type bool.

```
if(exprOp == "+" || exprOp == "-"){
    if(type1 == BOOL){
        cout << "Invalid expression: " << exprOp << " can only be applied to ints and floats" << endl;
        exit(EXIT_FAILURE);
if(exprOp == "*"){
    if(type1 == INT) currentType = INT;//multiplication between two ints = int
    else if(type1 == FLOAT) currentType = FLOAT;//multiplication between two floats = float
    else{
        cout << "Invalid expression: " << exprOp << " can only be applied to ints and floats" << endl;
        exit(EXIT_FAILURE);
    }
}else if(exprOp == "/"){
    if(type1 == BOOL){
        cout << "Invalid expression: " << exprOp << " can only be applied to ints and floats" << endl;
        exit(EXIT_FAILURE);
```

- If the ‘or’ or ‘and’ operators are used on variables of type float or int

```
}else{
    if(type1 != BOOL){
        cout << "Invalid expression: " << exprOp << " can only be applied to bools" << endl;
        exit(EXIT_FAILURE);
    }
}else{
    if(type1 != BOOL){
        cout << "Invalid expression: " << exprOp << " can only be applied to bools" << endl;
        exit(EXIT_FAILURE);
    }
```

- If the value being assigned to a variable in an assignment statement does not match the type of the variable it is being assigned to.

```
void SemanticAnalysisVisitor::visit(ASTAssignmentStatement *node) {
    TYPE vartype, valtype;
    node->identifierNode->Accept(this);
    vartype = currentType;
    node->expression->Accept(this);
    valtype = currentType;
    if(vartype != valtype){//error occurs if the variable and the value it is assigned have different types
        cout << "Assignment statement error: Variable type does not match expression type" << endl;
        exit(EXIT_FAILURE);
    }
}
```

- If the type of variable returned by a return statement does not match the return type of the function it is a part of.

```
for(ASTStatementNode *statementNode : *node->block){
    statementNode->Accept(this);
    if(typeid(*statementNode) == typeid(ASTReturnStatement)){
        returnFlag = true;
        if(currentType != returnType){
            cout << "Return type does not match return type specified in function declaration" << endl;
            exit(EXIT_FAILURE);
        }
    }
}
```

- If the conditional expression of a for statement doesn't give a bool value.

```
void SemanticAnalysisVisitor::visit(ASTForStatement *node) {
    TYPE expressiontype;
    if(node->variableDeclStatement->identifierNode->value != "") node->variableDeclStatement->Accept(this);//c
    node->expression->Accept(this);
    expressiontype = currentType;
    if(node->assignmentStatement->identifierNode->value != "") node->assignmentStatement->Accept(this);//check
    if(expressiontype != BOOL){
        cout << "Expression in for statement arguments must be of type bool" << endl;
        exit(EXIT_FAILURE);
    }
    //since the variable declaration and assignment in for statements are optional no further checks are done
}
```

- If a function is read all the way through and no return statement is found.

```
node->blockStatement->Accept(this);
if(!returnFlag){
    cout << "Function has no return statement" << endl;
    exit(EXIT_FAILURE);
}
```

- If the condition of an if statement doesn't give a bool value.

```
void SemanticAnalysisVisitor::visit(ASTIfStatement *node) {
    node->expression->Accept(this);
    if(currentType != BOOL){
        cout << "Conditional expression for if statement must be a bool" << endl;
        exit(EXIT_FAILURE);
    }
```

- If a declared variable is initialized with a value that doesn't match the type it is declared as or if a variable with the same identifier already exists in the scope.

```
void SemanticAnalysisVisitor::visit(ASTVariableDeclStatement *node) {
    TYPE vartype;
    if(node->type->value == "int") vartype = INT;
    else if(node->type->value == "float") vartype = FLOAT;
    else vartype = BOOL;
    node->expression->Accept(this);
    if(currentType != vartype){
        cout << "Expression type does not match declared identifier type" << endl;
        exit(EXIT_FAILURE);
    }
    if(symbolTable->insertvar(node->identifierNode->value, new Value(currentType)) == -1){
        cout << "Duplicate declaration of variable " << node->identifierNode->value << endl;
        exit(EXIT_FAILURE);
    }
}
```

Task 5 - Interpreter Execution Pass

The class InterpreterExecutionVisitor has the exact same properties as the SemanticAnalysisVisitor with an added property in the form of a stack of Value objects called values. This stack is used to store variable values used throughout the program such as the results of expressions and allow them to be used between different methods of the class. An example of this would be evaluating the expression node of a print statement and then pushing its result to the top of the stack so that it may be accessed by the print statement, printed and popped from the stack. When a literal is visited, it's value should be pushed to the top of the stack.

Due to time constraints, this visitor's functionality was not successfully completed, although some methods, namely the methods for visiting expressions, print statements and literals were implemented. Snippets of these particular methods can be found below.

```
void InterpreterExecutionVisitor::visit(ASTBool *node) {
    values->push(new Value(node->boolValue));
    currentType = BOOL;
}

void InterpreterExecutionVisitor::visit(ASTFloat *node) {
    values->push(new Value(node->floatValue));
    currentType = FLOAT;
}
void InterpreterExecutionVisitor::visit(ASTInt *node) {
    values->push(new Value(node->value));
    currentType = INT;
}
void InterpreterExecutionVisitor::visit(ASTPrintStatement *node) {
    node->expression->Accept(this);
    if(values->top()->identifierType == INT) cout <<  values->top()->intval << endl;
    else if(values->top()->identifierType == FLOAT) cout << values->top()->floatval << endl;
    else cout << boolalpha << values->top()->floatval << endl;
    values->pop();
}
```

```
void InterpreterExecutionVisitor::visit(ASTExpression *node) {
    node->simpleExpression1->Accept(this);
    node->simpleExpression2->Accept(this);
    //evaluating the first expression followed by the second expression.
    //It should be noted that this means the value at the top of the stack will be the result of the second expression.
    if(node->relOp->value == "<"){
        Value *val2 = values->top();//retrieving the result of the second expression from the stack
        values->pop();//popping the result of the second expression from the stack
        Value *val1 = values->top();//retrieving the result of the first expression from the stack
        values->pop();//popping the result of the first expression from the stack
        if(val1->identifierType == INT) values->push(new Value(*val1->intval < *val2->intval));
        else values->push(new Value(*val1->floatval < *val2->floatval));
    }else if(node->relOp->value == ">"){
        Value *val2 = values->top();//retrieving the result of the second expression from the stack
        values->pop();//popping the result of the second expression from the stack
        Value *val1 = values->top();//retrieving the result of the first expression from the stack
        values->pop();//popping the result of the first expression from the stack
        if(val1->identifierType == INT) values->push(new Value(*val1->intval > *val2->intval));
        else values->push(new Value(*val1->floatval > *val2->floatval));
    }else if(node->relOp->value == "<="){
        Value *val2 = values->top();//retrieving the result of the second expression from the stack
        values->pop();//popping the result of the second expression from the stack
        Value *val1 = values->top();//retrieving the result of the first expression from the stack
        values->pop();//popping the result of the first expression from the stack
        if(val1->identifierType == INT) values->push(new Value(*val1->intval <= *val2->intval));
        else if(val1->identifierType == FLOAT) values->push(new Value(*val1->floatval <= *val2->floatval));
        else values->push(new Value(*val1->boolval <= *val2->boolval));
    }else if(node->relOp->value == ">="){
        Value *val2 = values->top();//retrieving the result of the second expression from the stack
        values->pop();//popping the result of the second expression from the stack
        Value *val1 = values->top();//retrieving the result of the first expression from the stack
        values->pop();//popping the result of the first expression from the stack
        if(val1->identifierType == INT) values->push(new Value(*val1->intval >= *val2->intval));
        else if(val1->identifierType == FLOAT) values->push(new Value(*val1->floatval >= *val2->floatval));
        else values->push(new Value(*val1->boolval >= *val2->boolval));
    }else if(node->relOp->value == "=="){
        Value *val2 = values->top();//retrieving the result of the second expression from the stack
        values->pop();//popping the result of the second expression from the stack
        Value *val1 = values->top();//retrieving the result of the first expression from the stack
        values->pop();//popping the result of the first expression from the stack
        if(val1->identifierType == INT) values->push(new Value(*val1->intval == *val2->intval));
        else if(val1->identifierType == FLOAT) values->push(new Value(*val1->floatval == *val2->floatval));
        else values->push(new Value(*val1->boolval == *val2->boolval));
    }else if(node->relOp->value == "!="){
        Value *val2 = values->top();//retrieving the result of the second expression from the stack
        values->pop();//popping the result of the second expression from the stack
        Value *val1 = values->top();//retrieving the result of the first expression from the stack
        values->pop();//popping the result of the first expression from the stack
        if(val1->identifierType == INT) values->push(new Value(*val1->intval != *val2->intval));
        else if(val1->identifierType == FLOAT) values->push(new Value(*val1->floatval != *val2->floatval));
        else values->push(new Value(*val1->boolval != *val2->boolval));
```

```
void InterpreterExecutionVisitor::visit(ASTSimpleExpression *node) {
    node->term1->Accept(this);
    node->term2->Accept(this);
    //evaluating the first expression followed by the second expression.
    //It should be noted that this means the value at the top of the stack will be the result of the second expression.
    if(node->addOp->value == "+"){
        Value *val2 = values->top();//retrieving the result of the second expression from the stack
        values->pop();//popping the result of the second expression from the stack
        Value *val1 = values->top();//retrieving the result of the first expression from the stack
        values->pop();//popping the result of the first expression from the stack
        if(val1->identifierType == INT) values->push(new Value(*val1->intval + *val2->intval));
        else values->push(new Value(*val1->floatval + *val2->floatval));
    }else if(node->addOp->value == "-"){
        Value *val2 = values->top();//retrieving the result of the second expression from the stack
        values->pop();//popping the result of the second expression from the stack
        Value *val1 = values->top();//retrieving the result of the first expression from the stack
        values->pop();//popping the result of the first expression from the stack
        if(val1->identifierType == INT) values->push(new Value(*val1->intval - *val2->intval));
        else values->push(new Value(*val1->floatval - *val2->floatval));
    }else if(node->addOp->value == "or"){
        Value *val2 = values->top();//retrieving the result of the second expression from the stack
        values->pop();//popping the result of the second expression from the stack
        Value *val1 = values->top();//retrieving the result of the first expression from the stack
        values->pop();//popping the result of the first expression from the stack
        values->push(new Value(*val1->boolval | *val2->boolval));
    }
}
void InterpreterExecutionVisitor::visit(ASTTerm *node) {
    node->factor1->Accept(this);
    node->factor2->Accept(this);
    //evaluating the first expression followed by the second expression.
    //It should be noted that this means the value at the top of the stack will be the result of the second expression.
    if(node->multOp->value == "*"){
        Value *val2 = values->top();//retrieving the result of the second expression from the stack
        values->pop();//popping the result of the second expression from the stack
        Value *val1 = values->top();//retrieving the result of the first expression from the stack
        values->pop();//popping the result of the first expression from the stack
        if(val1->identifierType == INT) values->push(new Value(*val1->intval * *val2->intval));
        else values->push(new Value(*val1->floatval / *val2->floatval));
    }else if(node->multOp->value == "/"){
        Value *val2 = values->top();//retrieving the result of the second expression from the stack
        values->pop();//popping the result of the second expression from the stack
        Value *val1 = values->top();//retrieving the result of the first expression from the stack
        values->pop();//popping the result of the first expression from the stack
        if(val1->identifierType == INT) values->push(new Value(*val1->intval / *val2->intval));
        else values->push(new Value(*val1->floatval / *val2->floatval));
    }else if(node->multOp->value == "and"){
        Value *val2 = values->top();//retrieving the result of the second expression from the stack
        values->pop();//popping the result of the second expression from the stack
        Value *val1 = values->top();//retrieving the result of the first expression from the stack
        values->pop();//popping the result of the first expression from the stack
        values->push(new Value(*val1->boolval & *val2->boolval));
    }
}
```