Computer Science NEA

William Dowers, CN: 7060

May 5, 2023

Contents

1 Nomenclature

- `AST` - Abstract Syntax Tree, a tree representation of a specific piece of code.
- `Token` - A discrete unit that is understood by the parser. It is generated from a `lexeme` and also stores information such as position in source file, token type (e.g. number, identifier, reserved keyword etc)
- `LL(1)` - A type of parser that implements left-recursive, leftmost derivation with 1 lookahead character (meaning it does not need to backtrack)
- `Leftmost derivation` - It takes the first derivation of a production rule reading from right-to-left
- `Lookahead character` - The character that is peeked from a stream without being consumed.
- `Lexeme` - A sequence of characters in the source file that is converted into a `Token`
- `Grammar` - A set of production rules that defines a language
- `Nonterminal` - Something within a production rule that can be further expanded by another production rule.
- `Terminal` - Something within a production rule that cannot be further expanded by another production rule.
- `Lexical analyser` - The first stage of the translator that converts the source file into a stream of Tokens.
- `Syntax analyser` - Implemented by the parser, it is the second stage of the translator that generates an `AST` from the Token stream from the `Lexical analyser`.
- `Parser` - The stage that implements the syntax and semantic analysis stages of the interpreter.
- `Semantic analyser` The stage responsible for evaluating the semantics of the AST and validating it.

2 Analysis

2.1 Introduction

Currently, GCSE and A-level computer science students for the AQA exam board are required to use AQA's pseudo-code specification to write simple programs and procedures in their exams. For many, this is the first time encountering a programming language, and some may find it overwhelming, particularly with the lack of availability of troubleshooting, necessitating the cumbersome and error-prone act of tracing the program by hand. My proposal is to make an interpreter for the AQA pseudo-code specification that works through the command line, either by executing raw text files or my starting up a REPL[1]. It will be written in C++ as this allows for the low-level memory control required for optimisation which will be particularly useful for the REPL mode.

The entire process will be pipelined, with the source file being passed into a parser that interacts with a lexical analyser[2] to generate a token[3] stream from lexeme[4] interpretation. The parser stage will use a predictive parser[5] to generate an abstract syntax tree[6] (AST hereafter) that will act as an intermediate internal representation of the program before execution. I intend for this to operate through a CLI[7] as this will prove the easiest to incorporate into other projects.

The creation of compilers and interpreters has been, for the most part, standardised since the need for programming languages. Therefore, plentiful scholarly articles, books, websites and tools are available to aid in their creation. I have been making use of books: *Crafting Interpreters* [2] and *Compilers: Principles, Techniques and Tools* [3] in order to learn the standards for writing interpreters and compilers to design an appropriate and functional solution to this problem. In order to maximise the workload for myself I will not be making use of tools such as parser generators[8]. Additionally, the AQA pseudocode has a rigid specification available on their website [1] and so the objectives of this project will largely be implementing the features listed in the specification.

From my research, there are three main ways to execute a program from source. The first of which is the typical compiler, that takes in the source file and through several intermediate stages generates machine code that can be executed, typically in the form of some executable binary. This typically only works on one machine as the instruction set is specific to that hardware. The second method is known as transpiling and involves taking the source file, generating an intermediate representation, and then creating another source file in another language for which a compiler or interpreter exists and running that in the backend. This is typically relatively easy to implement as the heavy lifting and machine-specific optimisations are performed by the already existing translator and are good for when a language needs a translator quickly or for conversion between programming languages. The third is by an interpreter, which are much like compilers in that they take in the source code and generate an intermediate representation of it, however, they instead 'interpret' the intermediate representation and execute it then. This means the translator has to be run each time the program needs to be executed. Since the required optimisations of a compiler are currently beyond my capabilities, I settled with an interpreter.

[1] read, execute, print, loop

[2] a lexical analyser, or lexer, generates a token stream by interpreting the source file

[3] the internal, unique representation for a lexeme, typically consisting of a token type and attribute value

[4] a single, typically whitespace-separated, word in the input stream

[5] predictive parsing is a type of recursive descent parsing, a top-down parser constructing a parse tree from the top-down through a recursive implementation without backtracking

[6] an AST is an abstract representation of the entire program through a tree data structures.

[7] command-line interface

[8] a parser generator is a tool that takes a grammar and generates a corresponding parser

The interpreter can be broken down into 3 stages, the first of which is referred to as lexing, or lexical analysis. Attempting to execute a language from its source file would be very challenging, as each discrete unit within the language must be identified both individually and in relation to all other discrete units. The most obvious solution is to break the source file into an array of these discrete units, which are referred to as Tokens. A token consists of a type, such as `identifier` or `FOR`, and a lexeme. A lexeme is a string that shows how that token appears in the source file, so all tokens with type `FOR` have the lexeme `FOR`. Tokens will also store the line number in which they appear to display future error messages as this information would be lost when converting from the character stream (the source file) to the token stream after lexical analysis. The complete language of accepted tokens can be modelled by a regular language.

The second stage is referred to as parsing. In this stage, an intermediate representation of the program is built from the output of the lexing stage (the token stream). It generates an array of `Statement` objects that each execute parts of the program in different ways. `Statement` objects also contain `Expression` objects that evaluate to some literal. Note that all `Statement` objects can only be executed and all `Expression` objects can only be evaluated. Both objects are implemented through polymorphism in this implementation to enforce the overriding of certain common procedures, like `Statement::execute()` and `Expression::evaluate()`. Since both `Statement` and `Expression` objects can grow dynamically and recursively, they are implemented as a tree structure where the children exist as aggregates of the base `Expression` class to the parent, avoiding memory allocation problems since the size of the parent would not be known at compile time. The purpose of the parser is simply to construct this intermediate representation, known as an abstract syntax tree, or AST, not to interpret it.

The third and final stage of this is called the interpreter. This takes in the list of statements from the parser and calls `Statement::execute()` on each one. The different behaviours of each statement are contained within each implementation of the `Statement::execute()` method. When executing statements, expressions are typically evaluated, which are evaluated through a post-order traversal of the expression's tree structure, returning a single literal, which may be of type`NumericLiteral`, `StringLiteral`, or `BooleanLiteral`. Fundamentally, the `Statement` objects inform the interpreter on how to manipulate and transform the data contained in the `Expression` objects.

2.2 Objectives

2.2.1 How the solution to the problem will be achieved

1. Have the user run the executable via the CLI, passing in a source file as a command-line argument.
2. The source file will be passed to a lexer that generates a token stream.
3. Have the parser interface with a lexical analyser object through a *GetNextToken()* function.
4. Make the lexical analyser search for tokens using regular expressions (again, using the C++ STL) and generate a token.
5. Have the lexical analyser check to see if the token it generates is a reserved keyword or exists already on the symbol table [9].
6. Have the lexical analyser pass the generated token to the Parser by reference.
7. The Parser will then use recursive-descent parsing to generate an AST.
8. The interpreter will later use the AST to evaluate the semantics of the program and execute it (this is an interpreter so there is no need to generate a compiled binary, avoiding the need for code generation or machine-independent and machine-dependent code optimisations).

2.2.2 Features of the language to be implemented

1. Variable Assignment

```
foo <- "bar"
```

2. Standard arithmetic operations: `+`, `-`, `*`, `/`,
3. Relational operators: `<=`, `>=`, `<`, `>`, `=`, `!=`,
4. `WHILE` loops

```
WHILE <Expression> DO
    <Statement>
ENDWHILE
```

[9] internally there will be no way to distinguish between the two at this point - reserved keywords will be pushed to the symbol table upon its creation and each reserved keyword will have a special case invoked in order to evaluate them during semantic analysis

5. `FOR` loops

```
FOR <iterator> <- <From> TO <End> STEP <increment> DO
    <Statement>
ENDFOR
```

6. `IF` statements

```
IF <Expression> THEN
    <Statement>
ELSE IF <Expression> THEN
    <Statement>
...
ELSE
    <Statement>
ENDIF
```

7. `PRINT` statements

```
    PRINT <Expression>
```

8. `USERINPUT` statements

9. Subroutines (and return statements)

```
SUBROUTINE <identifier>(<params>)
    <Statement>
ENDSUBROUTINE
```

10. String concatenation

11. Implicit type conversions, e.g. `"hello"+5` becomes `"hello5"`

12. Explicit type conversions, e.g. `STRING_TO_NUMERIC()`

3 Documented Design

3.1 An Overview

The process of executing from a source file without compilation can be broken down into three discrete stages. The first, called lexical analysis, implemented by a lexical analyser or lexer, takes an input character stream and outputs a token stream. This is then passed onto the syntax analysis stage, implemented by a parser that takes the token stream and generates an intermediate representation of this code, which will be an AST in this implementation. The parser will catch any syntax errors and exit the program, displaying its error type and location to the user. Code may then be optimised and machine-dependent optimisation may be performed, although I will not be implementing these last two as this is an interpreter, not a compiler, and therefore optimisation would not be necessary. An interpreter then executes each statement and evaluates each expression through a post-order traversal to the root, which is the ASTs return value. It repeats this until all statements have been executed.

Modelling the interpreter through diagrams is challenging as such a machine fundamentally has little state beyond the lexical analyser (lexer). The behaviour of the parser can be represented through a series of EBNF production rules but an attempt to use a flowchart for a machine that computes so dynamically would prove to be near impossible.

$$\begin{tikzcd} \text{Lexing} \arrow[d] \\ \text{Parsing} \arrow[d] \\ \text{Interpreting} \end{tikzcd}$$

3.2 Lexing

Lexing takes the source file and converts it into a stream of token objects, which are individual units of information that can be parsed. In their most primitive forms, tokens contain a type and an (optional) value. For example,

```
FOR i <- 1 TO 10 STEP 5
    PRINT  "hello world"
    PRINT "iteration: "
    PRINT i
ENDFOR
```

would be broken into:

```
    <FOR> <identifier, i> <ASSIGN> <number, 1> <TO> <STEP> <5> <NEWLINE>
    <PRINT> <string, "hello world"> <NEWLINE>
    <PRINT> <string, "iteration: "> <NEWLINE>
    <PRINT> <identifier, i> <NEWLINE>
    <ENDFOR> <EOF>
```

where each `<...>` encapsulates a Token object with its associated fields. It should be noted that no syntax or semantic errors are caught in this stage, only invalid character sequences, as it is merely concerned with converting the code to a form that the parser can understand.

A lexer object will be created on launch to aid in the abstraction and separation of functionality throughout the project. A class diagram for the lexer object can be seen below.

Lexer
-mSrc: filestream -mKeywords: hashMap[string, TOKEN_TYPE] -mLine: int
+Lexer(filepath: string): void +GetNextToken(): Token +GetAllTokens(): Token[] -InitKeywords(): void

Token
-mTag: TOKEN_TYPE -mValue: void* -mLexeme: string -mLine: int+
+Token(tag: TOKEN_TYPE, line: int): void +GetValue(): void* +GetType(): TOKEN_TYPE +GetLine(): int +GetLexeme(): string

The '-' and '+' in the class diagram represent private and public access respectively for fields and methods. `mSrc` is the file stream object (and as this is to be written in C++ it will be an `std::ifstream` object) that will be read from. I will be using this file stream object rather than reading the entire file contents to a string and then manipulating that as the programs to be run on this may be quite large in size and may quickly overfill the programs allocated space in memory, requiring buffering to be used. This is beyond the scope of this project and it, therefore, made more sense to use the C++ standard template library instead. The field `mKeywords` is a hashmap[10] that contains strings as the keys and `TOKEN_TYPE`s as the values. When the identifier state is accepted by the FSA, the lexer first checks to see if it is a reserved keyword. If it is, the hashmap creates a 'hit' and returns the appropriate `TOKEN_TYPE` for that reserved keyword e.g. `TOKEN_TYPE::IF`. Otherwise, the lexer returns a new token of type `TOKEN_TYPE::IDENTIFIER` and sets the appropriate value (which is the lexeme itself in this case).

The primary functionality of the lexical analysis stage is contained within the `Lexer::GetNextToken()` function. It uses the file stream and iterates over it until it finds a character that could match a valid token. The lexical analyser only examines one character at a time. When finding a valid character, it peeks ahead the furthest it can before the lexeme would no longer be a valid token, and takes the valid lexeme it peeked furthest ahead for, consuming all the characters and returning a new token object with type set appropriately and lexeme attached. This process can be modelled by a finite state machine and the accompanying state transition diagram can be seen beneath the source code below. An example of how the lexical analyser would process the input stream $\textvisiblespace < -$ can be demonstrated below.

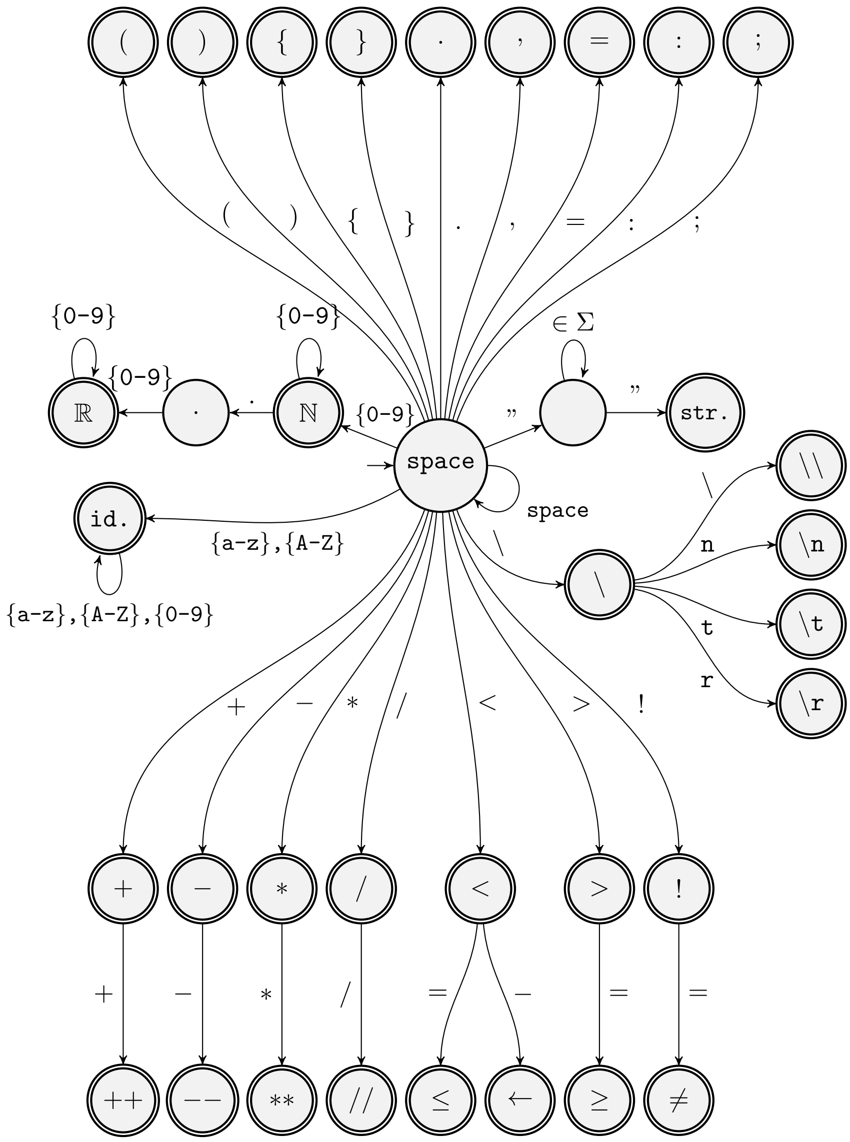
[10] a hashmap is a dictionary of key-value pairs with O(1) lookup time

pos.		[0]	[1]	⋯
input stream	␣	<	−	
	↑			
		[0]	[1]	⋯
	␣	<	−	
		↑		
		[0]	[1]	
	␣	<	−	
			↑	⋯

In the example above, the character currently being examined is '␣'[11], which is ignored. The next is "<", and so the lexical analyser moves into the $<$ state. While this is an accepting state, the analyser then checks to see if the following character in the input stream is $=$ or $-$. In this case, it is a $-$, which causes it to move to the $<-$ state. This is an accepting state and there are no longer valid lexemes with $<-$ as a prefix. The `Lexer::GetNextToken()` function, therefore, returns a new `Token` object of type `Assignment`.

Below shows the FSA representation of the `Lexer::GetNextToken()` function. The starting state is visible at the centre, the state transitions (edges) represent the input characters from the file stream and the double-circled states are accepting states that would generate a valid token. The lexer does not return a new token immediately upon hitting an accepting state, as there may be another state beyond that. It first checks all further possible inputs before returning a valid token.

[11] ␣is just a space character but has this symbol used here in order to be visible



3.3 Parsing

In this stage, the token stream is transformed into an AST guided by the grammars that define the language. In the previous stage, regular expressions can be used to describe completely the ordering of characters that would be accepted by the lexical analysis stage. In this stage, however, we must allow for recursion, necessitating the use of context-free grammars to model the translation of the token stream due to the weaker expressivity of regular expressions being insufficient for an FSA[12]. Below shows an example of an input character stream, the corresponding token stream from the lexical analysis stage and the AST we intend to generate. Note that whitespace is ignored in the lexical analysis stage and is present in the token stream for readability. The context-free grammars that describe the language must satisfy two main conditions in order for an LL(1)[13] parser to be

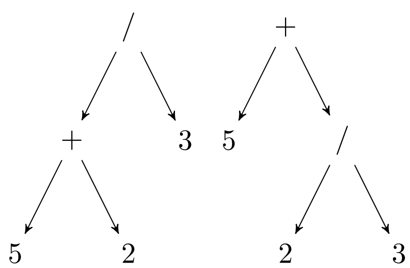
[12]Finite-state automata

[13]LL(1) parsers are a category of recursive descent parsers that follow have the principles of operations of: being left-recursive, following the leftmost derivation, whilst using up to one lookahead character

feasible. *Compilers: Principles, Techniques, and Tools* details algorithms for converting the invalid production rules into ones that can be used with an LL(1) parser which I have followed. These production rules are listed below in EBNF[14] and are the standard for expressions in many language parser implementations.

```
    expression ::= equality
    equality ::= comparison (==|!=) comparison)
    comparison ::= term ((">"|"<"|"<="|">=") term)*
    term ::= factor (("+"|"-") factor)*
    factor ::= unary | "-" unary
    unary ::= ("-" | "!") unary | literal
    literal ::= NUMERICLITERAL | STRINGLITERAL | TRUE | FALSE | NULL | "("
   expression ")"
```

This type of parsing, LL(1) have two requirements for the production rules: they must not be left-recursive or be ambiguous. This means that the leftmost derivation (the first derivation tried as the first 'L' stands for left-to-right) must not directly or indirectly include the production rule itself in it. This would mean that the parser would recursively run forever until it reached a stack overflow. The production rules listed satisfy this requirement. Ambiguity in the context of these grammars means that there are multiple derivation paths throughout a series of production rules that satisfy an expression. For example: $5 + 2/3$ could be interpreted in two ways:

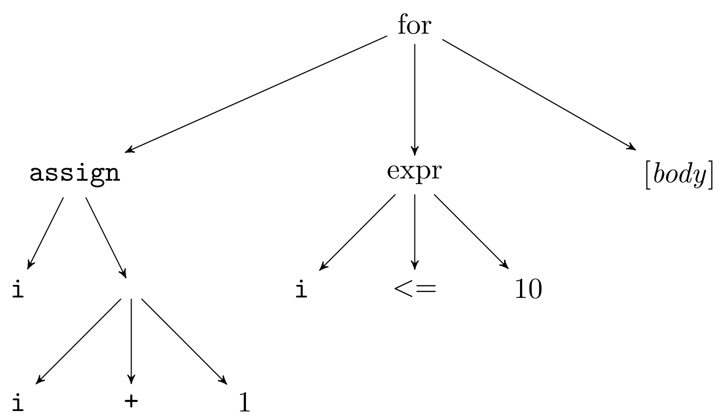


The first of which evaluates to 1.333... while the second evaluates to 5.666.... This means that the expression would not only be syntactically ambiguous but also semantically ambiguous when left in its infix form. This would not be possible to parse in an LL(1) parser as-is and necessitates the elimination of ambiguity in the production rules, which can be seen in the listed grammars.

[14]BNF with the extended operations in regex

The AST generated the following simple code snippet by the parser can be seen below;

```
FOR i <- TO 10 DO
    [body]
ENDFOR
```



Expressions all evaluate through the `Expression::evaluate()` function to a literal type.

Statement objects describe how the interpreter should evaluate the expressions and implement the behaviour of the program. Like expression, they will be implemented through polymorphism according to the following EBNF production rules.

```
    stmt ::= func_declaration | var_declaration | if | while | for | print | return
    func_declarataion ::= SUBROUTINE IDENTIFIER RIGHT_PAREN IDENTIFIER | ( IDENTIFIER ',')+ LEFT_PAREN block ENDSURBROUTINE
    var_declaration ::= IDENTIFIER ASSIGN expr
    block ::= (stmt)*
    if ::=  IF expr THEN block
            (ELSE IF expr THEN block)*
            (ELSE block)?
            ENDIF
    while ::= WHILE expr DO block ENDWHILE
    for ::= FOR IDENTIFIER ASSIGN expr TO expr (STEP expr)? DO block ENDWHILE
    print ::= PRINT expr
    return ::= RETURN expr
```

Each terminal for the `stmt` production rule in the EBNF above is implemented as a subclass of `Statement`. Like `Expression`, since `Statement` objects can grow dynamically and recursively, all `Statement` and `Expression` children of `Statement` nodes are stored as aggregate pointers within the objects. This is because C++ is a statically-typed language meaning the size of objects must be known at compile time and the size of a dynamically growing structure cannot be known trivially at compile time. Furthermore, the fragmentation of the program representation and ASTs throughout main memory as opposed to storing it contiguously makes the implementation of the interpreter more efficient as a large block of memory does not have to be available or freed, assuming it is possible.

Again, like `Expression`, the visitor pattern shall be used to allow the `Interpreter` to be able to distinguish between different `Statement` subclass types whilst only holding a pointer to the subclass object of type `Statement*` without violating the Liskov substitution principle[4] by, for example, attempting to perform a `std::dynamic_cast` [15] to each subclass until one is successful.

3.4 Interpreting

Interpreting the AST is a relatively simple process. A post-order traversal is performed from the root of the tree until it is collapsed into a single value, which is what the program exits with. The interpreter implements to visitor design pattern as the type of each child node may not be known external to the child itself - all `expression` nodes derive from `Expression` and are stored as pointers within classes. This implementation of the visitor design pattern consists of the following:

1. The interpreter calls the `accept()` method of the node, which is declared in the `Expression` base class, passing in a pointer to itself.
2. The dynamically-dispatched implementation of that class is called through a vtable-lookup
3. The node calls a specific method in the interpreter by dereferencing the provided pointer that evaluates that specific subclass and returns a new Expression node.

The interpreter also contains a stack that consists of environments which are added upon hitting a new scope. The environments will also contain symbol tables which are implemented as a hashmap of strings to `Literal` objects. For simplicity, all function calls are inlined [16]. Below is the class outline for the interpreter. Each of the "Visit" methods are called from the `Expression::accept()` method inside the `Interpreter::evaluate()` method such that they match the appropriate type of the `Expression` node.

Interpreter
+evaluate(expr: Expression*): Expression* +VisitGroupExpression(expr: GroupExpression*): Expression* +VisitBinaryExpression(expr: BinaryExpression*): Expression* +VisitUnaryExpression(expr: UnaryExpression*): Expression* +VisitBooleanLiteralExpression(expr: BooleanLiteral*): Expression +VisitNumericLiteralExpression(expr: NumericLiteral*): Expression +VisitStringLiteralExpression(expr: StringLiteral*): Expression +interpret(node: Expression*): void

[15] in the c++ stl

[16] Semantically, this means the function body has the parameter references changed for the specific call's arguments and placed at the function call with a new scope surrounding it

4 Technical Solution

4.1 Tokens

```cpp
#pragma once


enum class TOKEN_TYPE
{
    IF=0, ELSE, ELSEIF, ENDIF,
    FOR, ENDFOR, WHILE, ENDWHILE, TO, STEP,
    RETURN, CONTINUE, BREAK,
    AND, OR, NOT, TRUE, FALSE,
    SUBROUTINE, ENDSUBROUTINE, CLASS, ENDCLASS, PRINT, INPUT,
    LEFT_PAREN, RIGHT_PAREN, LEFT_BRACE, RIGHT_BRACE, LEFT_SQUARE, RIGHT_SQUARE,
    PLUS, MINUS, STAR, SLASH,
  STAR_STAR, PLUS_PLUS, MINUS_MINUS, EQUAL_EQUAL, SLASH_SLASH,
    DOT, COMMA, COLON, SEMICOLON, EQUAL,
    EXCLAMATION, NOT_EQUAL,
    ASSIGNMENT,
    GREATER_THAN, GREATER_EQ_THAN,
    LESS_THAN, LESS_EQ_THAN,
    IDENTIFIER, STRING, INTEGER, REAL, CHAR,
    NEWLINE,
    END_OF_FILE, ERROR, SPACE, THEN, DO
};

class Token
{
public:
    Token(TOKEN_TYPE tag, std::string lexeme);
  Token(TOKEN_TYPE tag);

    Token() {};
    Token(const Token& t1);


    TOKEN_TYPE GetType() const;

    std::string GetLexeme();
    int GetLine() const;

    friend std::ostream& operator<<(std::ostream& os, const Token& token);

private:
    TOKEN_TYPE mType;
    std::string mLexeme;
};
```

Listing 1: token.h

`TOKEN_TYPE` is an enumerable type that stores every possible token type in the language. A token consists of a tag of type `TOKEN_TYPE` and string (for non-keyword identifiers) for the lexeme. It does little more than hold data with some helper methods and abstraction through enumerable types.

```cpp
#include <iostream>
#include <string>

#include "token.h"

Token::Token(TOKEN_TYPE type, std::string lexeme) {
    mType = type;
    mLexeme = lexeme;
}


Token::Token(TOKEN_TYPE type) { mType = type; }

Token::Token(const Token& t1) {
    mType = t1.mType;
    mLexeme = t1.mLexeme;
}


TOKEN_TYPE Token::GetType() const { return mType; }
```

Listing 2: token.cpp

4.2 Environments

Environments store local variables in a hashtable of strings to `Value` objects, where `Value` objects are dynamic variables making this interpretation of the AQA pseudocode a dynamically-typed language.

```cpp
#pragma once



#include <unordered_map>
#include "token.h"

class Environment;

typedef std::unordered_map<std::string, class Value> map;

enum class VALUE_TYPE {
    BOOLEAN, NUMBER, STRING, FUNCTION
};

union value {
    double number;
    std::string* string;
    bool boolean;
    Function* function;
};

class Value {
public:
    Value(double val);
    Value(std::string val);
    Value(bool val);
```

```
    Value(Function* fun);
    Value() {};
    ~Value();

    VALUE_TYPE valueType;
    value   literal;

};

class Environment {
public:


    void Declare(std::string name, double val);
    void Declare(std::string name, bool val);
    void Declare(std::string name, std::string val);
    void Declare(std::string name, Function* fun);

    bool Assign(std::string name, double val);
    bool Assign(std::string name, bool val);
    bool Assign(std::string name, std::string val);
    bool Assign(std::string name, Function* fun);

    void SetParent(Environment* par);
    Environment();
    Environment(Environment* par);


    Value Get(std::string name);



private:
    Environment* parent;
    map variables;
};
```

Listing 3: environment.h

`parent` is a pointer to the scope surrounding this one. When a variable specified fails to be found within this scope, it checks its parents moving up until it hits the global scope. Since this is dynamically typed, would-be failed assignments are treated as variable declarations. As expected, `Declare()` declares a new object on the current scope, `Assign()` redefines one, although only `Declare()` is externally used due to variable semantics in this implementation.

```
#include <string>
#include <unordered_map>
#include <vector>

class Function;
class Expression;
class Interpreter;
class Visitor;

#include "token.h"
#include "nodes.h"
#include "environment.h"
```

```cpp
#include "interpreter.h"
#include "visitor.h"
#include "callable.h"

Value::Value(double val) {
    valueType = VALUE_TYPE::NUMBER;
    literal.number = val;
}


Value::Value(std::string val) {
    valueType = VALUE_TYPE::STRING;
    literal.string = new std::string(val);
}

Value::Value(bool val) {
    valueType = VALUE_TYPE::BOOLEAN;
    literal.boolean = val;
}

Value::Value(Function* fun) {
    valueType = VALUE_TYPE::FUNCTION;
    literal.function = fun;
}


Value::~Value() {
    if (valueType == VALUE_TYPE::STRING) {
        delete literal.string;
    }
}

void Environment::SetParent(Environment* par) {
    parent = par;
}

Environment::Environment() {
    parent = nullptr;
}
Environment::Environment(Environment* par) {
    parent = par;
}

void Environment::Declare(std::string name, double val) {
    if (!Assign(name, val)) {
        variables[name] = val;
    }
}

void Environment::Declare(std::string name, bool val) {
    if (!Assign(name, val)) {
        variables[name] = val;
    }
}

void Environment::Declare(std::string name, std::string val) {
    if (!Assign(name, val)) {
```

```cpp
        variables[name] = val;
    }
}

void Environment::Declare(std::string name, Function* fun) {
    if (!Assign(name, fun)) {
        variables[name] = fun;
    }
}


bool Environment::Assign(std::string name, double val) {
    if (variables.contains(name)) {
        variables[name] = Value(val);
        return true;
    }
    if (parent != nullptr) {
        parent->Assign(name, val);
    }
}

bool Environment::Assign(std::string name, bool val) {
    if (variables.contains(name)) {
        variables[name] = Value(val);
        return true;
    }
    if (parent != nullptr) {
        return parent->Assign(name, val);
    }

}

bool Environment::Assign(std::string name, std::string val) {
    if (variables.contains(name)) {
        variables[name] = Value(val);
        return true;
    }
    if (parent != nullptr) {
        return parent->Assign(name, val);
    }

}

bool Environment::Assign(std::string name, Function* fun) {
    if (variables.contains(name)) {
        variables[name] = Value(fun);
        return true;
    }
    if (parent != nullptr) {
        return parent->Assign(name, fun);
    }
}

Value Environment::Get(std::string name)  {
        if (variables.contains(name)) {
            return variables[name];
        }
```

```

        if (parent != nullptr) {
            return parent->Get(name);
        }

    }
```

Listing 4: environment.cpp

4.3 Lexing

```
#include <unordered_map>
#include <fstream>
#pragma once


class Lexer
{
public:
  Lexer(const std::string &filePath);
  ~Lexer();
  Token GetNextToken();
  void GetAllTokens(std::vector<Token>& tokens);
private:
  void InitKeyWords();

private:
  std::ifstream mSrc;
  std::unordered_map<std::string, TOKEN_TYPE> mKeyWords;
  int mCurrent;
};
```

Listing 5: lexer.h

The `mSrc` field in the `Lexer` class holds the source file which is read from. `mKeyWords` holds reserved keywords in a hashtable of string and `TOKEN_TYPE` pairs. The `mCurrent` holds the current cursor position as the file is read from. Upon construction, the hashtable is set up and `mSrc` is initialised as below.

The `GetNextToken()` function follows the FSA described earlier, attempting to reach an accepting state, and returning the next token. The `GetAllTokens()` function iteratively calls the `GetNextToken()` function and appends the returned tokens to a queue structure that is implemented through a dynamic array (`std::vector`). `InitTokens()` is a helper method that initialises the reserved words hashtable. Keywords are detected by checking to see if the hashtable returns a hit when passing in the lexeme of any identifier. This satisfies **Objective 2.2.1.2**, **Objective 2.2.1.4**, **Objective 2.2.1.5**, and **Objective 2.2.1.6**.

```
Token Lexer::GetNextToken() {
    if (mSrc.eof()) { return Token(TOKEN_TYPE::END_OF_FILE); }

    switch(mSrc.get()) {
        case ' ': return GetNextToken();
        case '(': return Token(TOKEN_TYPE::LEFT_PAREN);
        case ')': return Token(TOKEN_TYPE::RIGHT_PAREN);
        case '{': return Token(TOKEN_TYPE::LEFT_BRACE);
        case '}': return Token(TOKEN_TYPE::RIGHT_BRACE);
```

```
        case '[': return Token(TOKEN_TYPE::LEFT_SQUARE);
        case ']': return Token(TOKEN_TYPE::RIGHT_SQUARE);
        case '.': return Token(TOKEN_TYPE::DOT);
        case ',': return Token(TOKEN_TYPE::COMMA);
        case ':': return Token(TOKEN_TYPE::COLON);
        case ';': return Token(TOKEN_TYPE::SEMICOLON);
        case '=': return Token(TOKEN_TYPE::EQUAL);
        case '\n':
            return Token(TOKEN_TYPE::NEWLINE);

        case '+':
            if (mSrc.peek() == '+') { mSrc.ignore(); return Token(TOKEN_TYPE::PLUS_PLUS); }
            else { return Token(TOKEN_TYPE::PLUS); }
        case '-':
            if (mSrc.peek() == '-') { mSrc.ignore(); return Token(TOKEN_TYPE::MINUS_MINUS); }
            else { return Token(TOKEN_TYPE::MINUS); }
        case '*':
            if (mSrc.peek() == '*') { mSrc.ignore(); return Token(TOKEN_TYPE::STAR_STAR); }
            else { return Token(TOKEN_TYPE::STAR); }
        case '/':
            if (mSrc.peek() == '/') { mSrc.ignore(); return Token(TOKEN_TYPE::SLASH_SLASH); }
            else { return Token(TOKEN_TYPE::SLASH); }

        case '<':
            if (mSrc.peek() == '-') { mSrc.ignore(); return Token(TOKEN_TYPE::ASSIGNMENT); }
            else if (mSrc.peek() == '=') { mSrc.ignore(); return Token(TOKEN_TYPE::LESS_EQ_THAN); }
            else { return Token(TOKEN_TYPE::LESS_THAN); }
        case '>':
            if (mSrc.peek() == '=') { mSrc.ignore(); return Token(TOKEN_TYPE::GREATER_EQ_THAN); }
            else { return Token(TOKEN_TYPE::GREATER_THAN); }
        case '!':
            if (mSrc.peek() == '=') { mSrc.ignore(); return Token(TOKEN_TYPE::NOT_EQUAL); }
            else { return Token(TOKEN_TYPE::NOT); }




        case '"':
        {
            std::string lexeme;
            while (mSrc.peek() != '"') { lexeme.push_back(mSrc.get()); }
            mSrc.ignore();
            return Token(TOKEN_TYPE::STRING, lexeme);
        }

        default:
            mSrc.unget();
            if (isdigit(mSrc.peek())) {
                std::string lexeme;
```

```
                while (isdigit(mSrc.peek())) { lexeme.push_back(mSrc.get()); }
                if (mSrc.peek() != '.') { return Token(TOKEN_TYPE::INTEGER, lexeme); }
                mSrc.ignore();
                lexeme.push_back('.');
                while (isdigit(mSrc.peek())) { lexeme.push_back(mSrc.get()); }
                return Token(TOKEN_TYPE::REAL, lexeme);
            }

            if (isalpha(mSrc.peek())) {
                std::string lexeme;
                while (isalnum(mSrc.peek())) { lexeme.push_back(mSrc.get()); }
                Token(TOKEN_TYPE::IDENTIFIER, lexeme);
                if (mKeyWords.contains(lexeme)) { return Token(mKeyWords[lexeme]); }
                return Token(TOKEN_TYPE::IDENTIFIER, lexeme);
            }
            return Token(TOKEN_TYPE::END_OF_FILE);
    };

}



void Lexer::GetAllTokens(std::vector<Token>& tokens) {
    while (!mSrc.eof() && !mSrc.fail() )
        tokens.push_back(GetNextToken());
    if (tokens.back().GetType() != TOKEN_TYPE::END_OF_FILE) {
        tokens.push_back(TOKEN_TYPE::END_OF_FILE);
    }
}

void Lexer::InitKeyWords() {
    mKeyWords["IF"] = TOKEN_TYPE::IF;
    mKeyWords["ELSE"] = TOKEN_TYPE::ELSE;
    mKeyWords["ELSEIF"] = TOKEN_TYPE::ELSEIF;
    mKeyWords["ENDIF"] = TOKEN_TYPE::ENDIF;
    mKeyWords["THEN"] = TOKEN_TYPE::THEN;

    mKeyWords["FOR"] = TOKEN_TYPE::FOR;
    mKeyWords["STEP"] = TOKEN_TYPE::STEP;
    mKeyWords["TO"] = TOKEN_TYPE::TO;
    mKeyWords["ENDFOR"] = TOKEN_TYPE::ENDFOR;
    mKeyWords["DO"] = TOKEN_TYPE::DO;

    mKeyWords["WHILE"] = TOKEN_TYPE::WHILE;
    mKeyWords["ENDWHILE"] = TOKEN_TYPE::ENDWHILE;

    mKeyWords["RETURN"] = TOKEN_TYPE::RETURN;
    mKeyWords["CONTINUE"] = TOKEN_TYPE::CONTINUE;
    mKeyWords["BREAK"] = TOKEN_TYPE::BREAK;

    mKeyWords["AND"] = TOKEN_TYPE::AND;
    mKeyWords["OR"] = TOKEN_TYPE::OR;

    mKeyWords["NOT"] = TOKEN_TYPE::NOT;
    mKeyWords["TRUE"] = TOKEN_TYPE::TRUE;
```

```cpp
    mKeyWords["FALSE"] = TOKEN_TYPE::FALSE;

    mKeyWords["SUBROUTINE"] = TOKEN_TYPE::SUBROUTINE;
    mKeyWords["ENDSUBROUTINE"] = TOKEN_TYPE::ENDSUBROUTINE;

    mKeyWords["CLASS"] = TOKEN_TYPE::CLASS;
    mKeyWords["ENDCLASS"] = TOKEN_TYPE::ENDCLASS;

    mKeyWords["PRINT"] = TOKEN_TYPE::PRINT;
    mKeyWords["INPUT"] = TOKEN_TYPE::INPUT;
}
```

Listing 6: lexer.cpp

4.4 AST implementation

```cpp
#pragma once

class Expression;
class Printer;

class Statement {
public:
    Statement();
    virtual Statement* accept(Visitor* Interpreter)=0;
};

class PrintStatement : public Statement {
public:
    PrintStatement(Expression* expression);

    Statement* accept(Visitor* interpreter) override;
    Expression* expr;
};

class ExpressionStatement : public Statement {
public:
    ExpressionStatement(Expression* expression);

    Statement* accept(Visitor* interpreter) override;
    Expression* expr;
};

class VariableDeclarationStatement : public Statement {
public:
    VariableDeclarationStatement(Token identifier, Expression* value);

    Statement* accept(Visitor* interpreter) override;
    Token name;
    Expression* val;
};

class FunctionDeclarationStatement : public Statement {
public:
    FunctionDeclarationStatement(Token name, std::vector<Token> params, std::
   vector<Statement*> bdy);
```

```cpp

    Statement* accept(Visitor* interpreter) override;
    Token identifier;
    std::vector<Token> parameters;
    std::vector<Statement*> body;
};


class ReturnStatement : public Statement {
public:
    ReturnStatement(Expression* expr);
    Statement* accept(Visitor* interpreter) override;

    Expression* returnValue;
};

class BlockStatement : public Statement {
public:
    BlockStatement(std::vector<Statement*> stmts);
    Statement* accept(Visitor* interpreter) override;

    std::vector<Statement*> statements;
};

class IfStatement : public Statement {
public:
    IfStatement(Expression* cond, Statement* brnch, Statement* elseBrnch);

    Statement* accept(Visitor* interpreter) override;

    Expression *condition;
    Statement *branch, *elseBranch;
};

class WhileStatement : public Statement {
public:
    WhileStatement(Expression* cond, Statement* bdy);

    Statement* accept(Visitor* interpreter) override;

    Expression* condition;
    Statement* body;
};

class ForStatement : public Statement {
public:
    ForStatement(Token it, Expression* begin, Expression* finish, Statement* bdy);

    Statement* accept(Visitor* interpreter) override;

    Token iterator;
    Expression* start;
    Expression* end;
    Statement* body;
};
```

```cpp
class Expression : public Statement {
public:
    Expression();
    virtual Expression* accept(Visitor* interpreter)=0;
};

class BooleanLiteral : public Expression {
public:
    BooleanLiteral(bool mVal);
    Expression* accept(Visitor* interpreter);

    bool val;
};

class StringLiteral : public Expression {
public:
    StringLiteral(const std::string& _val);
    Expression* accept(Visitor* interpreter) override;
    std::string val;
};


class NumericLiteral : public Expression {
public:
    NumericLiteral(Token _num);
    NumericLiteral(double _val);
    Expression* accept(Visitor* interpreter) override;

    double val;
};

class GroupExpression : public Expression {
public:
    GroupExpression(Expression* _expr);
    ~GroupExpression() ;

    Expression* accept(Visitor* interpreter) override;

    Expression* expr;
};

class UnaryExpression : public Expression {
public:
    UnaryExpression(Token _operation, Expression* _expr);
    ~UnaryExpression();
    Expression* accept(Visitor* interpreter) override;

    Token operation;
    Expression* expr;
};

class CallExpression : public Expression {
public:
    CallExpression(Expression* callee, std::vector<Expression*> args);

    Expression* accept(Visitor* interpreter) override;

```

```
    Expression* callee;
    std::vector<Expression*> args;
};

class BinaryExpression : public Expression {
public:
    BinaryExpression(Expression* _left, Token _operation, Expression* _right);
    ~BinaryExpression();
    Expression* accept(Visitor* interpreter) override;


    Expression* left, *right;
    Token operation;
};

class VariableExpression : public Expression {
public:
    VariableExpression(Token identifier);

    Expression* accept(Visitor* interpreter) override;

    Token name;
};

class VariableAssignmentExpression : public Expression {
public:
    VariableAssignmentExpression(Token identifier, Expression* expr);
    Expression* accept(Visitor* interpreter) override;

    Token name;
    Expression* val;
};
```

Listing 7: nodes.h

Each terminal of the context-free grammars that describe the language has a class that derives from either `Statement` or `Expression`. They then implement the tree structure by containing either immediate values or pointers to other nodes that can later be traversed.

```
#include <string>
#include <iostream>
#include <memory>
#include <vector>

class Statement;
class Expression;
class Visitor;
class Interpreter;
class Function;

#include "token.h"
#include "nodes.h"
#include "interpreter.h"
#include "callable.h"


Expression::Expression() {};

```

```cpp



BooleanLiteral::BooleanLiteral(bool _val) { val = _val; }
Expression* BooleanLiteral::accept(Visitor* interpreter) { return interpreter->
    VisitBooleanLiteralExpression(this); }



StringLiteral::StringLiteral(const std::string& _val) { val = _val; }
Expression* StringLiteral::accept(Visitor* interpreter) { return interpreter->
    VisitStringLiteralExpression(this); }



NumericLiteral::NumericLiteral(Token _num) { val = (std::stod(_num.GetLexeme()));
    }
NumericLiteral::NumericLiteral(double _val) { val = _val; }
Expression* NumericLiteral::accept(Visitor* interpreter) { return interpreter->
    VisitNumericLiteralExpression(this); }



GroupExpression::GroupExpression(Expression* _expr) { expr = _expr; }
GroupExpression::~GroupExpression() { delete expr; }
Expression* GroupExpression::accept(Visitor* interpreter) { return interpreter->
    VisitGroupExpression(this); }



UnaryExpression::UnaryExpression(Token _operation, Expression* _expr) {
    operation = _operation;
    expr = _expr;
}
UnaryExpression::~UnaryExpression() {
    delete expr;
}

Expression* UnaryExpression::accept(Visitor* interpreter) { return interpreter->
    VisitUnaryExpression(this); }

CallExpression::CallExpression(Expression* _callee, std::vector<Expression*>
    arguments) {
    callee = _callee;
    args = arguments;
}

Expression* CallExpression::accept(Visitor* interpreter) { return interpreter->
    VisitCallExpression(this); }


BinaryExpression::BinaryExpression(Expression* _left, Token _operation, Expression
    * _right) {
    left = _left;
    operation = _operation;
    right = _right;
}
```

```cpp
BinaryExpression::~BinaryExpression() {
    delete left;
    delete right;
}

Expression* BinaryExpression::accept(Visitor* interpreter) { return interpreter->
    VisitBinaryExpression(this); }


Statement::Statement() {};

PrintStatement::PrintStatement(Expression* expression) { expr = expression; }

Statement* PrintStatement::accept(Visitor* interpreter) {
    interpreter->VisitPrintStatement(this);
    return nullptr;
}

ExpressionStatement::ExpressionStatement(Expression* expression) { expr =
    expression; }

Statement* ExpressionStatement::accept(Visitor* interpreter) {
    interpreter->VisitExpressionStatement(this);
    return nullptr;
}


VariableDeclarationStatement::VariableDeclarationStatement(Token identifier,
    Expression* value) {
    name = identifier;
    val = value;
}

Statement* VariableDeclarationStatement::accept(Visitor* interpreter) {
    interpreter->VisitVariableDeclarationStatement(this);
    return nullptr;
}

FunctionDeclarationStatement::FunctionDeclarationStatement(Token name, std::vector
    <Token> params, std::vector<Statement*> bdy) {
    identifier = name;
    parameters = params;
    body = bdy;
}

Statement* FunctionDeclarationStatement::accept(Visitor* interpreter) {
    interpreter->VisitFunctionDeclarationStatement(this);
    return nullptr;
}

ReturnStatement::ReturnStatement(Expression* value) { returnValue = value;}

Statement* ReturnStatement::accept(Visitor* interpreter) {
    interpreter->VisitReturnStatement(this);
    return nullptr;
}

```

```
VariableExpression::VariableExpression(Token identifier) { name = identifier; }
Expression* VariableExpression::accept(Visitor* interpreter) { return interpreter
    ->VisitVariableExpression(this); }


VariableAssignmentExpression::VariableAssignmentExpression(Token identifier,
    Expression* value) {
    name = identifier;
    val = value;
}

Expression* VariableAssignmentExpression::accept(Visitor* interpreter) { return
    interpreter->VisitVariableAssignmentExpression(this); }


BlockStatement::BlockStatement(std::vector<Statement*> stmts) { statements = stmts
    ; }

Statement* BlockStatement::accept(Visitor* interpreter) {
    interpreter->VisitBlockStatement(this);
    return nullptr;
}


IfStatement::IfStatement(Expression* cond, Statement* brnch, Statement* elseBrnch)
     {
    condition = cond;
    branch = brnch;
    elseBranch = elseBrnch;
}

Statement* IfStatement::accept(Visitor* interpreter) {
    interpreter->VisitIfStatement(this);
    return nullptr;
}


WhileStatement::WhileStatement(Expression* cond, Statement* bdy) {
    condition = cond;
    body = bdy;
}

Statement* WhileStatement::accept(Visitor* interpreter) {
    interpreter->VisitWhileStatement(this);
    return nullptr;
}

ForStatement::ForStatement(Token it, Expression* begin, Expression* finish,
    Statement* bdy) {
    iterator = it;
    start = begin;
    end = finish;
    body = bdy;
}

Statement* ForStatement::accept(Visitor* interpreter) {
    interpreter->VisitForStatement(this);
```

```cpp
    return nullptr;
}
```

Listing 8: nodes.cpp

The `accept()` method present in each node class is used to dynamically determine the type of node at runtime. This is vital to the implementation of the visitor pattern mentioned earlier whereby the interpreter calls `accept()` on the `Expression` or `Statement` class passing in a pointer to itself and the appropriate definition for whatever type the node is called the correct visit method in the interpreter.

4.5 Parsing

The parser is very similar to the lexer in the sense that it builds a structure based on the sequence of input units it sees, with the difference being that each nonterminal is broken into a class and the generated structure can dynamically and recursively grow. This satisfies **Objective 2.2.1.7**.

```cpp
#pragma once

class Parser {
public:
    Parser(std::vector<Token>& tokens);
    std::vector<Statement*> Parse();
private:
    Expression* expression();
    Expression* equality();
    Expression* assignment();
    Expression* comparison();
    Expression* term();
    Expression* factor();
    Expression* unary();
    Expression* call();
    Expression* finishCall(Expression* callee);
    Expression* primary();

    Statement* declaration();
    Statement* statement();
    Statement* variableDeclaration();
    Statement* functionDeclaration();
    Statement* returnStatement();
    Statement* printStatement();
    Statement* expressionStatement();
    Statement* blockStatement(TOKEN_TYPE endingToken);
    Statement* ifStatement();
    Statement* whileStatement();
    Statement* forStatement();

    Token previous();
    Token get();
    Token peek();
    void advance();

    void enforce(Token token);
    bool eof();
    bool end_of_file;
    std::vector<Token> mTokens;
```

```
    int mCurrent;
};
```

Listing 9: parser.h

`mTokens` holds the queue of tokens, `mCurrent` holds the pointer to the front of the queue, `eof()` returns true if the end of file has been reached, `enforce()` generates a syntax error if the next token is not of a specified token type, `get()` gets the next token, `advance()` increments `mCurrent`, `peek()` returns the next token without incrementing `mCurrent`, and `previous()` returns the last token consumed. This satisfies **Objective 2.2.1.3**.

```



#include <vector>
#include <memory>
#include <iostream>
#include <string>

class Expression;
class Visitor;
class Interpreter;

#include "token.h"
#include "nodes.h"
#include "parser.h"

Parser::Parser(std::vector<Token>& tokens) {
    mTokens = tokens;
    mCurrent = 0;
}

std::vector<Statement*> Parser::Parse() {
    std::vector<Statement*> statements;
    while (!eof()) {
        statements.push_back(declaration());
        enforce(TOKEN_TYPE::NEWLINE);
    }
    return statements;
}

Statement* Parser::declaration() {
    if (peek().GetType() == TOKEN_TYPE::SUBROUTINE) {
        return functionDeclaration();
    }
    if (peek().GetType() == TOKEN_TYPE::IDENTIFIER && mTokens[mCurrent+1].GetType
    () == TOKEN_TYPE::ASSIGNMENT) {
        return variableDeclaration();
    }
    return statement();
}

Statement* Parser::variableDeclaration() {
    Token name = get();
    advance();
    Expression* val = expression();
    return new VariableDeclarationStatement(name, val);
```

```
}

Statement* Parser::functionDeclaration() {
    advance();
    Token identifier = get();
    enforce(TOKEN_TYPE::LEFT_PAREN);
    std::vector<Token> parameters;
    while (peek().GetType() != TOKEN_TYPE::RIGHT_PAREN) {
        if (parameters.size() > 127) {
            throw std::invalid_argument("Cannot have more than 127 parameters");
        }
        Token param = get();
        parameters.push_back(param);
        if (peek().GetType() != TOKEN_TYPE::RIGHT_PAREN) {
            enforce(TOKEN_TYPE::COMMA);
        }
    }
    enforce(TOKEN_TYPE::RIGHT_PAREN);

    enforce(TOKEN_TYPE::NEWLINE);
    std::vector<Statement*> body = static_cast<BlockStatement*>(blockStatement(
   TOKEN_TYPE::ENDSUBROUTINE))->statements;
    return new FunctionDeclarationStatement(identifier, parameters, body);
}

Statement* Parser::returnStatement() {
    Expression* expr = nullptr;
    if (peek().GetType() != TOKEN_TYPE::NEWLINE) {
        expr = expression();
    }
    return new ReturnStatement(expr);
}

Statement* Parser::statement() {
    while (peek().GetType() == TOKEN_TYPE::NEWLINE) {
        advance();
    }
    if (peek().GetType() == TOKEN_TYPE::IF) {
        advance();
        return ifStatement();
    }
    if (peek().GetType() == TOKEN_TYPE::WHILE) {
        advance();
        return whileStatement();
    }
    if (peek().GetType() == TOKEN_TYPE::PRINT) {
        advance();
        return printStatement();
    }
    if (peek().GetType() == TOKEN_TYPE::RETURN) {
        advance();
        return returnStatement();
    }
    /*if (peek().GetType() == TOKEN_TYPE::FOR) {
        advance();
        return forStatement();
    }*/
```

```
    if (peek().GetType() == TOKEN_TYPE::LEFT_BRACE) {
        advance(); advance();
        Statement* stmt = blockStatement(TOKEN_TYPE::RIGHT_BRACE);
        return stmt;
    }
    return expressionStatement();
}

Statement* Parser::ifStatement() {
    Expression* condition = expression();
    enforce(TOKEN_TYPE::THEN);
    enforce(TOKEN_TYPE::NEWLINE);

    std::vector<Statement*> branch;
    while ((peek().GetType() != TOKEN_TYPE::ELSE || peek().GetType() != TOKEN_TYPE::ENDIF) && !eof()) {
        branch.push_back(declaration());
        if (mTokens[mCurrent+1].GetType() != TOKEN_TYPE::END_OF_FILE) {
            enforce(TOKEN_TYPE::NEWLINE);
        }
    }

    Statement* elseBranch = nullptr;
    if (peek().GetType() == TOKEN_TYPE::ELSE) {
        advance();
        elseBranch = blockStatement(TOKEN_TYPE::ENDIF);
    }
    if (!eof()) { enforce(TOKEN_TYPE::NEWLINE); }
    return new IfStatement(condition, new BlockStatement(branch), elseBranch);
}

Statement* Parser::whileStatement() {
    Expression* condition = expression();
    enforce(TOKEN_TYPE::DO);
    enforce(TOKEN_TYPE::NEWLINE);
    Statement* body = blockStatement(TOKEN_TYPE::ENDWHILE);
    if (!eof()) { enforce(TOKEN_TYPE::NEWLINE); }
    return new WhileStatement(condition, body);
}

Statement* Parser::forStatement() {
    Token identifier = get();
    enforce(TOKEN_TYPE::ASSIGNMENT);
    Expression* from = expression();
    enforce(TOKEN_TYPE::TO);
    Expression* to = expression();
    enforce(TOKEN_TYPE::DO);
    enforce(TOKEN_TYPE::NEWLINE);
    Statement* body = blockStatement(TOKEN_TYPE::ENDFOR);
    return new ForStatement(identifier, from, to, body);
}


Statement* Parser::blockStatement(TOKEN_TYPE endingToken) {
    std::vector<Statement*> stmts;
    while (peek().GetType() != endingToken && !eof()) {
        stmts.push_back(declaration());
```

```
        if (mTokens[mCurrent+1].GetType() != TOKEN_TYPE::END_OF_FILE) {
            enforce(TOKEN_TYPE::NEWLINE);
        }
    }
    enforce(endingToken);
    return new BlockStatement(stmts);

}

Statement* Parser::printStatement() {
    Expression* expr = expression();
    return new PrintStatement(expr);
}

Statement* Parser::expressionStatement() {
    Expression* expr = expression();
    return expr;
}





Expression* Parser::expression() {
    Expression* expr = assignment();

    return expr;
}

Expression* Parser::assignment() {
    Expression* expr = equality();
    if (peek().GetType() == TOKEN_TYPE::ASSIGNMENT) {
        Token equals = previous();
        advance();
        Expression* value = assignment();

        if (dynamic_cast<VariableExpression*>(expr) != nullptr) {
            Token name = ((VariableExpression*)expr)->name;
            return new VariableAssignmentExpression(name, expr);
        }

    }
    return expr;
}

Expression* Parser::equality() {
    Expression* expr = comparison();

    while (peek().GetType() == TOKEN_TYPE::NOT_EQUAL || peek().GetType() ==
    TOKEN_TYPE::EQUAL) {
        Token operation = get();
        Expression* right = comparison();
        expr = new BinaryExpression(expr, operation, right);
    }
    return expr;
}

```

```
Expression* Parser::comparison() {
    Expression* expr = term();
    while (peek().GetType() == TOKEN_TYPE::LESS_EQ_THAN     ||
        peek().GetType() == TOKEN_TYPE::LESS_THAN        ||
        peek().GetType() == TOKEN_TYPE::GREATER_THAN     ||
        peek().GetType() == TOKEN_TYPE::GREATER_EQ_THAN) {
        Token operation = get();
        Expression* right = term();
        expr = new BinaryExpression(expr, operation, right);
    }
    return expr;
}

Expression* Parser::term() {
    Expression* expr = factor();
    while (peek().GetType() == TOKEN_TYPE::PLUS || peek().GetType() == TOKEN_TYPE
    ::MINUS) {

        Token operation = get();
        Expression* right = factor();
        expr = new BinaryExpression(expr, operation, right);
    }
    return expr;
}

Expression* Parser::factor() {
    Expression* expr = unary();
    while (peek().GetType() == TOKEN_TYPE::SLASH || peek().GetType() == TOKEN_TYPE
    ::STAR) {
        Token operation = get();
        Expression* right = unary();
        expr = new BinaryExpression(expr, operation, right);
    }
    return expr;
}

Expression* Parser::unary() {
    if (peek().GetType() == TOKEN_TYPE::EXCLAMATION || peek().GetType() ==
    TOKEN_TYPE::MINUS) {
        Token operation = get();
        Expression* right = unary();
        return new UnaryExpression(operation, right);
    }
    return call();
}

Expression* Parser::call() {
    Expression* expr = primary();
    while (true) {
        if (peek().GetType() == TOKEN_TYPE::LEFT_PAREN) {
            expr = finishCall(expr);
        }
        else {
            break;
        }
    }

```

```
    return expr;
}

Expression* Parser::finishCall(Expression* callee) {
    std::vector<Expression*> args;
    advance();
    while (peek().GetType() != TOKEN_TYPE::RIGHT_PAREN) {
        if (args.size()>127) {
            throw std::inavlid_argument("Cannot have over 127 args")
        }
        args.push_back(expression());
        if (peek().GetType() != TOKEN_TYPE::RIGHT_PAREN) {
            enforce(TOKEN_TYPE::COMMA);
        }
    }

    enforce(TOKEN_TYPE::RIGHT_PAREN);
    return new CallExpression(callee, args);
}

Expression* Parser::primary() {
    std::cout<<static_cast<int>(peek().GetType())<<std::endl;
    switch (peek().GetType())
    {
    case TOKEN_TYPE::FALSE:
        advance();
        return new BooleanLiteral(false);
    case TOKEN_TYPE::TRUE:
        advance();
        return new BooleanLiteral(true);
    case TOKEN_TYPE::IDENTIFIER:
        return new VariableExpression(get());
    case TOKEN_TYPE::INTEGER:
        return new NumericLiteral(Token(get()));
    case TOKEN_TYPE::REAL:
        return new NumericLiteral(Token(get()));
    case TOKEN_TYPE::STRING:
        return new StringLiteral(get().GetLexeme());
    case TOKEN_TYPE::LEFT_PAREN:
        advance();
        Expression* expr = expression();
        enforce(Token(TOKEN_TYPE::RIGHT_PAREN));
        return new GroupExpression(expr);
    }
    return nullptr;
}

Token Parser::previous() { return mTokens[mCurrent-1]; }
Token Parser::get() {
    if (!eof()) {
        mCurrent++;
        return previous();
    }
    return Token(TOKEN_TYPE::END_OF_FILE);
}
Token Parser::peek() { return mTokens[mCurrent]; }
void Parser::advance() { mCurrent++; }
```

```
void Parser::enforce(Token token) {

    if (mTokens[mCurrent].GetType() != token.GetType()) {
        std::cout<<"Syntax Error: Expected: "<<static_cast<int>(token.GetType())<<
    std::endl;
    }
    advance();
}
bool Parser::eof() {
    if (mCurrent >= mTokens.size()-1) { return true; }
    return false;
}
```

Listing 10: parser.cpp

4.6 Visitor interface

There are two classes that implement the visitor pattern. Therefore, in order to be able to use the same `accept()` method from the nodes classes, I used an interface from which the two classes derive. The `accept()` method will take in a pointer of type `Visitor` instead and use polymorphism through dynamic dispatch again to determine the right visit method to call.

```
#pragma once

#include "nodes.h"

class Visitor{
public:
    virtual NumericLiteral* VisitNumericLiteralExpression(NumericLiteral*
    numericLiteral) = 0;
    virtual StringLiteral* VisitStringLiteralExpression(StringLiteral*
    stringLiteral) = 0;
    virtual BooleanLiteral* VisitBooleanLiteralExpression(BooleanLiteral*
    booleanLiteral) = 0;

    virtual Expression* VisitGroupExpression(GroupExpression* groupExpression) =
    0;
    virtual Expression* VisitUnaryExpression(UnaryExpression* unaryExpression) =
    0;
    virtual Expression* VisitCallExpression(CallExpression* callExpression) = 0;
    virtual Expression* VisitBinaryExpression(BinaryExpression* binaryExpression)
    = 0;

    virtual Expression* VisitVariableExpression(VariableExpression*
    variableExpression) = 0;
    virtual Expression* VisitVariableAssignmentExpression(
    VariableAssignmentExpression* variableAssignmentExpression) = 0;

    virtual void VisitExpressionStatement(ExpressionStatement* expressionStatement
    ) = 0;
    virtual void VisitPrintStatement(PrintStatement* printStatement) = 0;
    virtual void VisitVariableDeclarationStatement(VariableDeclarationStatement*
    variableStatement) = 0;
    virtual void VisitFunctionDeclarationStatement(FunctionDeclarationStatement*
    functionDeclaration) = 0;
    virtual void VisitReturnStatement(ReturnStatement* returnStatement) = 0;
```

```
    virtual void VisitBlockStatement(BlockStatement* blockStatement) = 0;
    virtual void VisitIfStatement(IfStatement* ifStatement) = 0;
    virtual void VisitWhileStatement(WhileStatement* whileStatement) = 0;
    virtual void VisitForStatement(ForStatement* forStatement) = 0;


};
```

Listing 11: Visitor.h

In C++, the `virtual` keyword indicates that the method has no definition in the base class. The = 0 following the class definition indicates that the methods must be overridden and defined in all derived classes.

4.7 Interpreting

```
#pragma once

#include "nodes.h"
#include "visitor.h"
#include "ASTprinter.h"
#include "environment.h"

class ReturnException : public std::exception {
public:
    ReturnException(Expression* expr);

    Expression* value;
};


class Interpreter : public Visitor {
public:
    Interpreter();
    void Interpret(std::vector<Statement*> statements);


    NumericLiteral* VisitNumericLiteralExpression(NumericLiteral* numericLiteral)
    override;
    StringLiteral* VisitStringLiteralExpression(StringLiteral* stringLiteral)
    override;
    BooleanLiteral* VisitBooleanLiteralExpression(BooleanLiteral* booleanLiteral)
    override;

    Expression* VisitGroupExpression(GroupExpression* groupExpression) override;
    Expression* VisitUnaryExpression(UnaryExpression* unaryExpression) override;
    Expression* VisitCallExpression(CallExpression* callExpression) override;
    Expression* VisitBinaryExpression(BinaryExpression* binaryExpression) override
    ;

    Expression* VisitVariableExpression(VariableExpression* variableExpression)
    override;
    Expression* VisitVariableAssignmentExpression(VariableAssignmentExpression*
    variableAssignmentExpression) override;

    void VisitExpressionStatement(ExpressionStatement* expressionStatement)
    override;
```

```
    void VisitPrintStatement(PrintStatement* printStatement) override;
    void VisitVariableDeclarationStatement(VariableDeclarationStatement*
    variableStatement) override;
    void VisitFunctionDeclarationStatement(FunctionDeclarationStatement*
    functionDeclarationStatement) override;
    void VisitReturnStatement(ReturnStatement* returnStatement) override;
    void VisitBlockStatement(BlockStatement* blockStatement) override;
    void VisitIfStatement(IfStatement* ifStatement) override;
    void VisitWhileStatement(WhileStatement* whileStatement) override;
    void VisitForStatement(ForStatement* forStatement) override;


    Expression* evaluate(Expression* expr);
    void execute(Statement* stmt);
    void executeBlock(std::vector<Statement*> stmts, Environment* env);

public:
    Environment* globals;


private:
    Printer printer;
    Environment* environment;


};
```

Listing 12: Interpreter.h

Since this derives from `Visitor`, all the pure virtual functions have been declared as overridden. The `evaluate()` function takes in an AST and reduces it to a single node that can be interpreted. The `execute()` function takes in a `statement` and executes it according to the rules defined in the visit methods. `executeBlock()` executes a series of statements. `environment` stores the current environment that holds all local variables and `printer` is used as an interface stdout. Each visit method is called from the dynamically selected definition of `accept()`. They execute or evaluate parts of the program at a time - i.e. the `visitWhileStatement()` method executes the `WhileStatement` object passed into it. This satisfies **Objective 2.2.1.8**.

```
#include <string>
#include <iostream>
#include <vector>
#include <cmath>
#include <stdexcept>

class Statement;
class Expression;
class Visitor;
class Interpreter;
class Printer;
class Environment;
class Value;
class Function;

#include "token.h"
#include "nodes.h"
#include "visitor.h"
#include "interpreter.h"
```

```cpp
#include "ASTprinter.h"
#include "environment.h"
#include "callable.h"



ReturnException::ReturnException(Expression* expr) { value = expr; }

Interpreter::Interpreter() {
    globals = new Environment();
    environment = globals;
}

void Interpreter::Interpret(std::vector<Statement*> statements) {
    environment = new Environment();
    for (Statement* stmt : statements) {
        execute(stmt);
    }
}


NumericLiteral* Interpreter::VisitNumericLiteralExpression(NumericLiteral* numericLiteral) { return numericLiteral; }
StringLiteral* Interpreter::VisitStringLiteralExpression(StringLiteral* stringLiteral) { return stringLiteral; }
BooleanLiteral* Interpreter::VisitBooleanLiteralExpression(BooleanLiteral* booleanLiteral) { return booleanLiteral; }

Expression* Interpreter::VisitGroupExpression(GroupExpression* groupExpression) { return groupExpression->expr; }

Expression* Interpreter::VisitUnaryExpression(UnaryExpression* unaryExpression) {
    Expression* right = evaluate(unaryExpression->expr);
    if(unaryExpression->operation.GetType() == TOKEN_TYPE::MINUS) {
        return new NumericLiteral(-1*((NumericLiteral*)right)->val);
    }
    return new NumericLiteral(((NumericLiteral*)right)->val);
}

Expression* Interpreter::VisitCallExpression(CallExpression* callExpression) {
    Expression* callee = evaluate(callExpression->callee);

    std::vector<Expression*> arguments;
    for (Expression* arg : callExpression->args) {
        arguments.push_back(evaluate(arg));
    }

    if (reinterpret_cast<Function*>(callee) == nullptr) {
        throw std::invalid_argument("Can only call subroutines");
    }



    Function* subroutine = reinterpret_cast<Function*>(callee);

    if (arguments.size() != subroutine->Arity()) {
        throw std::runtime_error("Expected " + std::to_string(subroutine->Arity())
```

```
        + " arguments but " + std::to_string(arguments.size()) + " were given");
    }

    return subroutine->call(this, arguments);
}





Expression* Interpreter::VisitBinaryExpression(BinaryExpression* binaryExpression) {
    Expression* left = evaluate(binaryExpression->left);
    Expression* right = evaluate(binaryExpression->right);
    switch(binaryExpression->operation.GetType()) {
        case TOKEN_TYPE::PLUS:
            if (dynamic_cast<NumericLiteral*>(left) != nullptr && dynamic_cast<NumericLiteral*>(right) != nullptr) {
                return new NumericLiteral(((NumericLiteral*)left)->val + ((NumericLiteral*)right)->val);
            }
            if (dynamic_cast<StringLiteral*>(left) != nullptr && dynamic_cast<NumericLiteral*>(right) != nullptr) {
                double rght = ((NumericLiteral*)right)->val;
                if (std::floor(rght) == rght) { return new StringLiteral(((StringLiteral*)left)->val.append(std::to_string(int(rght)))); }
                else { return new StringLiteral(((StringLiteral*)left)->val + (std::to_string(rght))); }
            }
            if (dynamic_cast<StringLiteral*>(left) != nullptr && dynamic_cast<StringLiteral*>(right) != nullptr) {
                return new StringLiteral(((StringLiteral*)left)->val + (((StringLiteral*)right)->val));
            }
            throw std::invalid_argument("Operands of '+' must evaluate to: (NumericLiteral, NumericLiteral), (StringLiteral, StringLiteral), (StringLiteral, NumericLiteral)");
        case TOKEN_TYPE::MINUS:
            if (dynamic_cast<NumericLiteral*>(left) == nullptr || dynamic_cast<NumericLiteral*>(right) == nullptr) { throw std::invalid_argument("Operands of '-' must evaluate to type NumericLiteral"); }
            return new NumericLiteral(((NumericLiteral*)left)->val - ((NumericLiteral*)right)->val);
        case TOKEN_TYPE::STAR:
            if (dynamic_cast<NumericLiteral*>(left) == nullptr || dynamic_cast<NumericLiteral*>(right) == nullptr) { throw std::invalid_argument("Operands of '*' must evaluate to type NumericLiteral"); }
            return new NumericLiteral(((NumericLiteral*)left)->val * ((NumericLiteral*)right)->val);
        case TOKEN_TYPE::SLASH:
            if (dynamic_cast<NumericLiteral*>(left) == nullptr || dynamic_cast<NumericLiteral*>(right) == nullptr) { throw std::invalid_argument("Operands of '/' must evaluate to type NumericLiteral"); }
            return new NumericLiteral(((NumericLiteral*)left)->val /  ((NumericLiteral*)right)->val);
        case TOKEN_TYPE::GREATER_THAN:
            if (dynamic_cast<NumericLiteral*>(left) == nullptr || dynamic_cast<
```

```
NumericLiteral*>(right) == nullptr) { throw std::invalid_argument("Operands of '>' must evaluate to type NumericLiteral"); }
            return new BooleanLiteral(((NumericLiteral*)left)->val > ((NumericLiteral*)right)->val);
        case TOKEN_TYPE::LESS_THAN:
            if (dynamic_cast<NumericLiteral*>(left) == nullptr || dynamic_cast<NumericLiteral*>(right) == nullptr) { throw std::invalid_argument("Operands of '<' must evaluate to type NumericLiteral"); }
            return new BooleanLiteral(((NumericLiteral*)left)->val < ((NumericLiteral*)right)->val);
        case TOKEN_TYPE::GREATER_EQ_THAN:
            if (dynamic_cast<NumericLiteral*>(left) == nullptr || dynamic_cast<NumericLiteral*>(right) == nullptr) { throw std::invalid_argument("Operands of '>=' must evaluate to type NumericLiteral"); }
            return new BooleanLiteral(((NumericLiteral*)left)->val >= ((NumericLiteral*)right)->val);
        case TOKEN_TYPE::LESS_EQ_THAN:
            if (dynamic_cast<NumericLiteral*>(left) == nullptr || dynamic_cast<NumericLiteral*>(right) == nullptr) { throw std::invalid_argument("Operands of '<=' must evaluate to type NumericLiteral"); }
            return new BooleanLiteral(((NumericLiteral*)left)->val <= ((NumericLiteral*)right)->val);
        case TOKEN_TYPE::NOT_EQUAL:
            if (dynamic_cast<NumericLiteral*>(left) == nullptr || dynamic_cast<NumericLiteral*>(right) == nullptr) { throw std::invalid_argument("Operands of '!=' must evaluate to type NumericLiteral"); }
            return new BooleanLiteral(((NumericLiteral*)left)->val != ((NumericLiteral*)right)->val);
        case TOKEN_TYPE::EQUAL:
            if (dynamic_cast<NumericLiteral*>(left) == nullptr || dynamic_cast<NumericLiteral*>(right) == nullptr) { throw std::invalid_argument("Operands of '=' must evaluate to type NumericLiteral"); }
            return new BooleanLiteral(((NumericLiteral*)left)->val == ((NumericLiteral*)right)->val);
        default:
            return nullptr;
    }
}

void Interpreter::VisitBlockStatement(BlockStatement* blockStatement) {
    executeBlock(blockStatement->statements, new Environment(environment));
}

Expression* Interpreter::VisitVariableExpression(VariableExpression* variableExpression) {
    Value val = environment->Get(variableExpression->name.GetLexeme());
    switch (val.valueType) {
        case VALUE_TYPE::BOOLEAN:
            return new BooleanLiteral(val.literal.boolean);
        case VALUE_TYPE::STRING:
            return new StringLiteral(*(val.literal.string));
        case VALUE_TYPE::NUMBER:
            return new NumericLiteral(val.literal.number);
        case VALUE_TYPE::FUNCTION:
            return reinterpret_cast<Expression*>(val.literal.function);
    }
}
```

```cpp

Expression* Interpreter::VisitVariableAssignmentExpression(
    VariableAssignmentExpression* variableAssignmentExpression) {
    Expression* value = evaluate(variableAssignmentExpression->val);
    if (dynamic_cast<NumericLiteral*>(value) != nullptr) {
        environment->Assign(variableAssignmentExpression->name.GetLexeme(),
    dynamic_cast<NumericLiteral*>(value)->val);
    }
    else if (dynamic_cast<StringLiteral*>(value) != nullptr) {
        environment->Assign(variableAssignmentExpression->name.GetLexeme(),
    dynamic_cast<StringLiteral*>(value)->val);
    }
    else if (dynamic_cast<BooleanLiteral*>(value) != nullptr) {
        environment->Assign(variableAssignmentExpression->name.GetLexeme(),
    dynamic_cast<BooleanLiteral*>(value)->val);
    }
    return value;
}


void Interpreter::VisitExpressionStatement(ExpressionStatement*
    expressionStatement) {
    Expression* expr = evaluate(expressionStatement->expr);
}
void Interpreter::VisitPrintStatement(PrintStatement* printStatement) {
    Expression* expr = evaluate(printStatement->expr);
    printer.Print(expr);
}

void Interpreter::VisitVariableDeclarationStatement(VariableDeclarationStatement*
    variableStatement) {
    Expression* value = evaluate(variableStatement->val);
    if (dynamic_cast<NumericLiteral*>(value) != nullptr) {
        environment->Declare(variableStatement->name.GetLexeme(), dynamic_cast<
    NumericLiteral*>(value)->val);
    }
    else if (dynamic_cast<StringLiteral*>(value) != nullptr) {
        environment->Declare(variableStatement->name.GetLexeme(), dynamic_cast<
    StringLiteral*>(value)->val);
    }
    else if (dynamic_cast<BooleanLiteral*>(value) != nullptr) {
        environment->Declare(variableStatement->name.GetLexeme(), dynamic_cast<
    BooleanLiteral*>(value)->val);
    }
    else {

    }
}


void Interpreter::VisitFunctionDeclarationStatement(FunctionDeclarationStatement*
    functionDeclarationStatement) {
    Function* function = new Function(functionDeclarationStatement);
    environment->Declare(functionDeclarationStatement->identifier.GetLexeme(),
    function);
}

```

```
void Interpreter::VisitReturnStatement(ReturnStatement* returnStatement) {
    Expression* value = nullptr;
    if (returnStatement->returnValue != nullptr) {
        value = evaluate(returnStatement->returnValue);
    }
    throw ReturnException(value);
}

void Interpreter::VisitIfStatement(IfStatement* ifStatement) {
    if (dynamic_cast<BooleanLiteral*>(evaluate(ifStatement->condition))->val) {
        execute(ifStatement->branch);
    }
    else if (dynamic_cast<BooleanLiteral*>(evaluate(ifStatement->condition))->val
    == false && ifStatement->elseBranch != nullptr) {
        execute(ifStatement->elseBranch);
    }
}

void Interpreter::VisitWhileStatement(WhileStatement* whileStatement) {
    while (dynamic_cast<BooleanLiteral*>(evaluate(whileStatement->condition))->val
    ) {
        execute(whileStatement->body);
    }
}

void Interpreter::VisitForStatement(ForStatement* forStatement) {
    Environment* previous = environment;
    environment = new Environment(environment);
    environment->Declare(forStatement->iterator.GetLexeme(), dynamic_cast<
    NumericLiteral*>(evaluate(forStatement->start))->val);
    while(environment->Get(forStatement->iterator.GetLexeme()).literal.number <
    dynamic_cast<NumericLiteral*>((forStatement->end))->val) {
        execute(forStatement->body);
        environment->Assign(forStatement->iterator.GetLexeme(), environment->Get(
    forStatement->iterator.GetLexeme()).literal.number+1);
    }
    delete environment;
    environment = previous;
}

Expression* Interpreter::evaluate(Expression* expr) {
    return expr->accept(this);
}

void Interpreter::execute(Statement* stmt) {
    stmt->accept(this);
}

void Interpreter::executeBlock(std::vector<Statement*> stmts, Environment* env) {
    Environment* current = environment;
    environment = env;
    for (Statement* stmt : stmts) {
        execute(stmt);
    }
    environment = current;
```

```
}
```

Listing 13: Interpreter.cpp

4.8 Printer

For the `PRINT` method, a class `Printer` is used that also derives from `Visitor` to evaluate and output the result of an expression. It also has diagnostic benefits, as printing a non-evaluated expression shows the tree structure of the expression without collapsing it which is beneficial during debugging.

```
#pragma once


#include "visitor.h"

class Printer : public Visitor {
public:
    void Print(Expression* expression);


    NumericLiteral* VisitNumericLiteralExpression(NumericLiteral* numericLiteral) override;
    StringLiteral* VisitStringLiteralExpression(StringLiteral* stringLiteral) override;
    BooleanLiteral* VisitBooleanLiteralExpression(BooleanLiteral* booleanLiteral) override;

    Expression* VisitGroupExpression(GroupExpression* groupExpression) override;
    Expression* VisitUnaryExpression(UnaryExpression* unaryExpression) override;
    Expression* VisitCallExpression(CallExpression* callExpression) override;
    Expression* VisitBinaryExpression(BinaryExpression* binaryExpression) override;

    Expression* VisitVariableExpression(VariableExpression* variableExpression) override;
    Expression* VisitVariableAssignmentExpression(VariableAssignmentExpression* variableAssignmentExpression) override;

    void VisitExpressionStatement(ExpressionStatement* expressionStatement) override;
    void VisitPrintStatement(PrintStatement* printStatement) override;
    void VisitVariableDeclarationStatement(VariableDeclarationStatement* variableDeclarationStatement) override;
    void VisitFunctionDeclarationStatement(FunctionDeclarationStatement* functionDeclarationStatement) override;
    void VisitReturnStatement(ReturnStatement* returnStatement) override;
    void VisitBlockStatement(BlockStatement* blockStatement) override;
    void VisitIfStatement(IfStatement* ifStatement) override;
    void VisitWhileStatement(WhileStatement* whileStatement) override;
    void VisitForStatement(ForStatement* ForStatement) override;


    Expression* print(Expression* expr);
};
```

Listing 14: ASTprinter.h

While statements are never printed, since the methods for visiting statements are purely virtual in the base class, they must be declared as overridden here.

```


#include <string>
#include <iostream>
#include <vector>

class Expression;
class Visitor;
class Printer;

#include "token.h"
#include "nodes.h"
#include "visitor.h"
#include "ASTprinter.h"

void Printer::Print(Expression* expression) {
    Expression* completed = print(expression);
    std::cout<<"\n";
    return;
}


NumericLiteral* Printer::VisitNumericLiteralExpression(NumericLiteral*
    numericLiteral) { std::cout<<numericLiteral->val; return nullptr; }
StringLiteral* Printer::VisitStringLiteralExpression(StringLiteral* stringLiteral)
     { std::cout<<stringLiteral->val; return nullptr; }
BooleanLiteral* Printer::VisitBooleanLiteralExpression(BooleanLiteral*
    booleanLiteral) {
    if (booleanLiteral->val) { std::cout<< "TRUE"; }
    else { std::cout<< "FALSE"; }
    return nullptr;
}

Expression* Printer::VisitGroupExpression(GroupExpression* groupExpression) {
    std::cout<<'(';
    print(groupExpression->expr);
    std::cout<<')';
    return nullptr;
}
Expression* Printer::VisitUnaryExpression(UnaryExpression* unaryExpression) {
    if(unaryExpression->operation.GetType() == TOKEN_TYPE::MINUS) {
        std::cout<<"-(";
        Expression* right = print(unaryExpression->expr);
        std::cout<<")";
        return nullptr;
    }
    print(unaryExpression->expr);
    return nullptr;
}

Expression* Printer::VisitCallExpression(CallExpression* callExpression) {

}

```

```

Expression* Printer::VisitBinaryExpression(BinaryExpression* binaryExpression) {
    Expression* left = print(binaryExpression->left);
    switch(binaryExpression->operation.GetType()) {
        case TOKEN_TYPE::PLUS:
            std::cout<<'+'; break;
        case TOKEN_TYPE::MINUS:
            std::cout<<'-'; break;
        case TOKEN_TYPE::STAR:
            std::cout<<'*'; break;
        case TOKEN_TYPE::SLASH:
            std::cout<<'/'; break;
        case TOKEN_TYPE::GREATER_THAN:
            std::cout<<'>'; break;
        case TOKEN_TYPE::LESS_THAN:
            std::cout<<'<'; break;
        case TOKEN_TYPE::GREATER_EQ_THAN:
            std::cout<<">="; break;
        case TOKEN_TYPE::LESS_EQ_THAN:
            std::cout<<"<="; break;
        case TOKEN_TYPE::NOT_EQUAL:
            std::cout<<"!="; break;
        case TOKEN_TYPE::EQUAL:
            std::cout<<'='; break;
        default:
            break;
    }
    Expression* right = print(binaryExpression->right);
    return nullptr;
}

Expression* Printer::print(Expression* expr) {
    return expr->accept(this);
}

Expression* Printer::VisitVariableExpression(VariableExpression* expression) {
    return nullptr; }

void Printer::VisitPrintStatement(PrintStatement* printStatement) { return; }
void Printer::VisitExpressionStatement(ExpressionStatement* ExpressionStatement) {
     return; }

void Printer::VisitVariableDeclarationStatement(VariableDeclarationStatement*
    variableDeclarationStatement) { return; }
void Printer::VisitFunctionDeclarationStatement(FunctionDeclarationStatement*
    functionDeclarationStatement) { return; }
void Printer::VisitReturnStatement(ReturnStatement* returnStatement) { return; }
Expression* Printer::VisitVariableAssignmentExpression(
    VariableAssignmentExpression* variableAssignmentExpression) { return nullptr; }

void Printer::VisitBlockStatement(BlockStatement* blockStatement) { return; }

void Printer::VisitIfStatement(IfStatement* ifStatement) { return; }
void Printer::VisitWhileStatement(WhileStatement* whileStatement) { return; }
void Printer::VisitForStatement(ForStatement* ForStatement) { return; }
```

Listing 15: ASTprinter.cpp

4.9 Putting it together

```cpp
#include <iostream>
#include <vector>

class Visitor;
class Interpreter;
class Printer;
class Function;

#include "token.h"
#include "lexer.h"
#include "nodes.h"
#include "parser.h"
#include "visitor.h"
#include "interpreter.h"
#include "ASTprinter.h"
#include "environment.h"
#include "callable.h"



int main(int argc, char** argv)
{

    if (argc != 2)
    {
        std::cerr<<"Usage- ./main.exe [filename]"<<std::flush;
        return -1;
    }

    Lexer lexer(argv[1]);
    std::vector<Token> vectors;
    lexer.GetAllTokens(vectors);

    Parser parser(vectors);
    std::vector<Statement*> statements = parser.Parse();
    Interpreter interpreter;
    interpreter.Interpret(statements);
```

Listing 16: main.cpp

If the user incorrectly uses the interpreter in the command line, they are prompted to do it correctly and the program exits. A lexer object generates the token stream through the `GetAllTokens()` function. The parser generates a list of statements through the `Parser()` method and the interpreter is called through the `interpret` method, passing in the statement list. This satisfies **Objective 2.2.1.1**.

5 Testing

5.1 Introduction

Since this project is to produce an interpreter, testing is relatively simple. Each input to the interpreter has one and only one behaviour that should be produced, therefore testing each objective is as simple as modelling a piece of code that implements that specific feature, determining the expected outcome through a brief inspection and dry-run, and executing that to see if the expected outcome matches the actual outcome. Any discrepancy between the two is an error and the code should be corrected.

5.2 Variable Assignment

5.2.1 Input

```
foo <- "bar"
```

5.2.2 Expectation

A variable of `Identifier` ”foo” should be created on the current scope’s environment of type `StringLiteral` and value ”bar”.

5.2.3 Test

```
foo <- "bar"
PRINT foo
```

results in ”bar” being output to the console, satisfying **Objective 2.2.2.1**.

5.3 Standard arithmetic operators

5.3.1 Input

```
PRINT 5+3
PRINT 2-6
PRINT 4*3
PRINT 2/8
```

5.3.2 Expectation

The program should print the result of the mathematical expressions

5.3.3 Test

```
8
-4
12
0.25
```

which satisfies **Objective 2.2.2.2**.

5.4 Relational Operators

5.4.1 Input

```
PRINT 5 > 3
PRINT 3 > 5
PRINT 2 < 4
PRINT 4 < 2
PRINT 2 >= 2
PRINT 3 >= 2
PRINT 2 >= 3
PRINT 4 <= 5
PRINT 4 <= 4
PRINT 4 <= 3
PRINT 6 = 6
PRINT 6 = 7
PRINT TRUE = FALSE
PRINT TRUE != FALSE
PRINT (TRUE = FALSE) = (6 < "banana")
```

5.4.2 Test Result

```
TRUE
FALSE
TRUE
FALSE
TRUE
TRUE
FLASE
TRUE
TRUE
FALSE
TRUE
FALSE
FALSE
TRUE
TRUE
```

which satisfies **Objective 2.2.2.3**.

5.5 While loops

5.5.1 Input

```
i <- 0
WHILE i < 10 DO
    PRINT i
    i <- i + 1
ENDWHILE
```

5.5.2 Expectation

Prints the numbers 0 to 9

5.5.3 Test Result

```
0
1
2
3
4
5
6
7
8
9
```

which satisfies **Objective 2.2.2.4**.

5.6 For loops

5.6.1 Input

```
FOR i < 1 TO 10 STEP 1 DO
    PRINT i
ENDFOR
```

5.6.2 Expectation

Prints the numbers 1 to 10

5.6.3 Test Result

```
1
2
3
4
5
6
7
8
9
10
```

which satisfies **Objective 2.2.2.5**.

5.7 If statement

5.7.1 Input

```
IF TRUE THEN
    PRINT "true"
ELSE
    PRINT "not true"
ENDIF

IF FALSE THEN
    PRINT "not true"
```

```
ELSE
    PRINT "true"
ENDIF
```

5.7.2 Expectation

Prints ”true” twice

5.7.3 Test Result

```
true
true
```

which satisfies **Objective 2.2.2.6**.

5.8 Userinput

5.8.1 Input

```
in <- USERINPUT
PRINT in
```

5.8.2 Expectation

Prints whatever the user inputs

5.8.3 Test Result

with input ”out”

```
out
```

which satisfies **Objective 2.2.2.7** and **Objective 2.2.2.8**.

5.9 Subroutines

5.9.1 Input

```
SUBROUTINE pow(base, exp)
    result <- 1
    WHILE exp > 0 DO
        result <- result * base
        exp <- exp - 1
    ENDWHILE
    RETURN result
ENDSUBROUTINE

PRINT pow(2, 5)
```

5.9.2 Expectation

Prints 32

5.9.3 Test Result

```
32
```

which satisfies **Objective 2.2.2.9**.

5.10 String concatenation

5.10.1 Input

```
PRINT "hello, " + "world
```

5.10.2 Expectation

Prints "hello, world"

5.10.3 Test Result

```
hello, world
```

which satisfied **Objective 2.2.2.10**

5.11 Implicit conversions

5.11.1 Input

```
PRINT "hello" + 5
```

5.11.2 Expectation

Prints "hello5"

5.11.3 Test Result

```
hello5
```

which satisfies **Objective 2.2.2.11**.

5.12 Explicit conversions

5.12.1 Input

```
PRINT STRING_TO_NUMERIC("5") * 3
```

5.12.2 Expectation

Prints 15

5.12.3 Result

```
15
```

which satisfies **Objective 2.2.2.12**.

6 Evaluation

6.1 Self-assessment

Overall, this project meets the objectives set out in **2.2** and implements an interpreter for the AQA pseudocode specification [1] making use of an LL(1) parser. Further support could be added by extending the language with OOP[17] which would make the project more useful in particular for A-level students as they are introduced to multiple programming paradigms.

The interface of a CLI has been chosen to make the project as easy-to-use as possible but also as extendable as possible. I believe it would be relatively simple to be able to build a desktop application, for example in *Electron* or *Node.js*, and use it as a wrapper for the command-line tool, providing an environment for students to code in that has a friendlier GUI, perhaps with other tools such as debugging, syntax highlighting, and other developer tools.

The final improvement I would make would be to make the project available on the web, either by compiling it to *WebAssembly* and executing it through that or through a cloud computing service, such as *AWS*, *Azure*, or *GCP*, with the latter of the two options being favourable for simplicity and ease of maintenance.

6.2 Independent feedback

For my independent feedback I obtained the following review:

"All of the objectives were achieved and the final application seems like a very useful tool for teaching GCSE students about pseudocode. However, it might be slightly difficult for younger students to use the command line to execute their code as it's not something typically done at GCSE level. It could be beneficial if there were extra tools such as syntax highlighting or maybe even a help method which can be called to explain how certain statements can be used."

which is consistent with my view on the project outcome.

[17] Object-oriented programming

6.3 Objectives

The table below details where, in this documentation, each objective is satisfied.

Objective	met in
2.2.1.1	4.9
2.2.1.2	4.3
2.2.1.3	4.5
2.2.1.4	4.3
2.2.1.5	4.3
2.2.1.6	4.3
2.2.1.7	4.5
2.2.1.8	4.7
2.2.2.1	5.2
2.2.2.2	5.3
2.2.2.3	5.4
2.2.2.4	5.5
2.2.2.5	5.6
2.2.2.6	5.7
2.2.2.7	5.8
2.2.2.8	5.8
2.2.2.9	5.9
2.2.2.10	5.10
2.2.2.11	5.11
2.2.2.12	5.12

References

[1] [Online; accessed 28. Mar. 2023]. Feb. 2023. URL: `https://filestore.aqa.org.uk/resources/computing/AQA-8525-NG-PC.PDF`.

[2] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools*. 2nd ed. Pearson, 2006.

[3] Robert Nystrom. *Crafting Interpreters*. s.n., 2021.

[4] *Wikiwand - Liskov substitution principle*. [Online; accessed 30. Mar. 2023]. Mar. 2023. URL: `https://www.wikiwand.com/en/Liskov_substitution_principle`.