Forth interpreter

Generated by Doxygen 1.12.0

Chapter 1

Concept Index

1.1 Concepts

Here is a list of all concepts with brief descriptions:

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Concept Documentation

5.1 Rational Concept Reference

Concept to define types that can be represented as rational numbers.

`#include <StackElement.h>`

5.1.1 Concept definition

```
template<typename T>
concept Rational =  std::integral<T> || std::is_same_v<double, T> || std::is_same_v<float, T>
```

5.1.2 Detailed Description

Concept to define types that can be represented as rational numbers.

Definition at line 18 of file StackElement.h.

Chapter 6

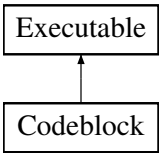
Class Documentation

6.1 Codeblock Class Reference

Represents a block of executable statements.

`#include <Executable.h>`

Inheritance diagram for Codeblock:



Public Member Functions

- ReturnStatus Execute (Environment &environment) override

  Executes all statements within the code block.

Public Member Functions inherited from Executable

- virtual ~Executable ()=default

  Virtual destructor for the Executable class.

Public Attributes

- std::vector< std::shared_ptr< Executable > > statements

  The statements to execute.

Additional Inherited Members

Public Types inherited from Executable

- enum class ReturnStatus { kSuccess , kLeaveLoop , kLeaveFunction , kContinueLoop }

  Enum representing the return status of an execution.

6.1.1 Detailed Description

Represents a block of executable statements.

Definition at line 66 of file Executable.h.

6.1.2 Member Function Documentation

6.1.2.1 Execute()

```
Executable::ReturnStatus Codeblock::Execute (
            Environment & environment)  [override], [virtual]
```

Executes all statements within the code block.

Parameters

environment	The execution environment.

Returns

The return status of the execution.

Implements Executable.

Definition at line 3 of file Codeblock.cpp.

6.1.3 Member Data Documentation

6.1.3.1 statements

```
std::vector<std::shared_ptr<Executable> > Codeblock::statements
```

The statements to execute.

Definition at line 75 of file Executable.h.

The documentation for this class was generated from the following files:

- Executable.h
- Codeblock.cpp

6.2 Environment Class Reference

Manages a stack of elements within the environment.

```
#include <Environment.h>
```

Public Member Functions

- StackElement PopStack ()

  Removes and returns the top element from the stack.
- void PushOnStack (StackElement s)

  Pushes a new element onto the stack.

Public Attributes

- std::map< std::string, std::shared_ptr< Executable > > functions

  A map of function names to their corresponding executable objects.
- std::map< std::string, void * > variables

  A map of variable names to their corresponding values.
- std::shared_ptr< Executable > code

  A shared pointer to the main executable code for the environment.
- std::vector< StackElement > stack

  The stack used to manage execution state and data.

6.2.1 Detailed Description

Manages a stack of elements within the environment.

Represents an execution environment with stack, variables, and functions.

Definition at line 20 of file Environment.h.

6.2.2 Member Function Documentation

6.2.2.1 PopStack()

```
StackElement Environment::PopStack ()
```

Removes and returns the top element from the stack.

This method checks if the stack is empty before attempting to remove an element. If the stack is empty, it throws a `std::runtime_error`.

Returns

The top element of the stack.

Exceptions

std::runtime_error	If the stack is empty.

Definition at line 17 of file Environment.cpp.

6.2.2.2 PushOnStack()

```
void Environment::PushOnStack (
            StackElement s)
```

Pushes a new element onto the stack.

This method adds the provided element to the top of the stack.

Parameters

s	The element to be pushed onto the stack.

Definition at line 32 of file Environment.cpp.

6.2.3 Member Data Documentation

6.2.3.1 code

```
std::shared_ptr<Executable> Environment::code
```

A shared pointer to the main executable code for the environment.

Definition at line 37 of file Environment.h.

6.2.3.2 functions

```
std::map<std::string, std::shared_ptr<Executable> > Environment::functions
```

A map of function names to their corresponding executable objects.

Definition at line 25 of file Environment.h.

6.2.3.3 stack

```
std::vector<StackElement> Environment::stack
```

The stack used to manage execution state and data.

Definition at line 62 of file Environment.h.

6.2.3.4 variables

```
std::map<std::string, void*> Environment::variables
```

A map of variable names to their corresponding values.

The values are stored as void pointers to allow flexibility in data types.

Definition at line 32 of file Environment.h.

The documentation for this class was generated from the following files:

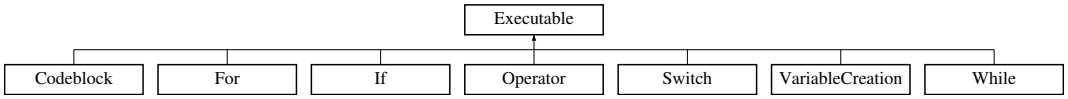
- Environment.h
- Environment.cpp

6.3 Executable Class Reference

Abstract base class for executable entities within the environment.

`#include <Executable.h>`

Inheritance diagram for Executable:



Public Types

- enum class ReturnStatus { kSuccess , kLeaveLoop , kLeaveFunction , kContinueLoop }

  Enum representing the return status of an execution.

Public Member Functions

- virtual ReturnStatus Execute (Environment &environment)=0

  Executes the entity within the given environment.
- virtual ~Executable ()=default

  Virtual destructor for the Executable class.

6.3.1 Detailed Description

Abstract base class for executable entities within the environment.

Definition at line 19 of file Executable.h.

6.3.2 Member Enumeration Documentation

6.3.2.1 ReturnStatus

`enum class Executable::ReturnStatus [strong]`

Enum representing the return status of an execution.

Enumerator

kSuccess	Execution completed successfully.
kLeaveLoop	Leave the current loop.
kLeaveFunction	Leave the current function.
kContinueLoop	Continue to the next iteration of the loop.

Definition at line 24 of file Executable.h.

6.3.3 Constructor & Destructor Documentation

6.3.3.1 ~Executable()

```
virtual Executable::~Executable ()  [virtual], [default]
```

Virtual destructor for the Executable class.

6.3.4 Member Function Documentation

6.3.4.1 Execute()

```
virtual ReturnStatus Executable::Execute (
            Environment & environment)  [pure virtual]
```

Executes the entity within the given environment.

Parameters

environment	The execution environment.

Returns

The return status of the execution.

Implemented in Codeblock, For, If, Operator, Switch, VariableCreation, and While.

The documentation for this class was generated from the following file:

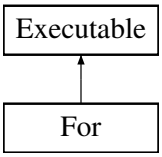
- Executable.h

6.4 For Class Reference

Represents a for loop structure.

```
#include <Executable.h>
```

Inheritance diagram for For:



Public Member Functions

- ReturnStatus Execute (Environment &environment) override

  Executes the for loop.

Public Member Functions inherited from Executable

- virtual ~Executable ()=default

  Virtual destructor for the Executable class.

Public Attributes

- std::shared_ptr< Executable > body

  The body of the for loop.

Additional Inherited Members

Public Types inherited from Executable

- enum class ReturnStatus { kSuccess , kLeaveLoop , kLeaveFunction , kContinueLoop }

  Enum representing the return status of an execution.

6.4.1 Detailed Description

Represents a for loop structure.

Definition at line 99 of file Executable.h.

6.4.2 Member Function Documentation

6.4.2.1 Execute()

```
Executable::ReturnStatus For::Execute (
            Environment & environment)  [override], [virtual]
```

Executes the for loop.

Parameters

environment	The execution environment.

Returns

The return status of the execution.

Implements Executable.

Definition at line 3 of file For.cpp.

6.4.3 Member Data Documentation

6.4.3.1 body

```
std::shared_ptr<Executable> For::body
```

The body of the for loop.

Definition at line 108 of file Executable.h.

The documentation for this class was generated from the following files:

- Executable.h
- For.cpp

6.5 GrammaticalAnalyzer Class Reference

Performs syntactic analysis and generates the resulting environment.

```
#include <GrammaticalAnalyzer.h>
```

Public Member Functions

- GrammaticalAnalyzer (const std::vector< Lexeme > &lexemes, const std::vector< std::string > &code_↩ block_enders)

  Constructs a GrammaticalAnalyzer.
- void Analyze ()

  Performs grammatical analysis on the provided lexemes.

Public Attributes

- Environment resulting_environment

  The resulting environment after analysis.

Private Member Functions

- Lexeme GetCurrentLexeme ()

  Retrieves the current lexeme being analyzed.
- void NextLexeme ()

  Advances to the next lexeme in the analysis.
- void ThrowSyntaxException (const std::string &message)

  Throws a syntax error exception with a specific message.
- void ThrowGenericException (const Lexeme &l, const std::string &prefix_text, const std::string &suffix_text)

  Throws a generic exception for a given lexeme.
- void ThrowUndefinedException (const Lexeme &l)

  Throws an exception for an undefined identifier.
- void ThrowNotIntegerException (const Lexeme &l)

  Throws an exception for a non-integer value.

- void ThrowNotInLoopException (const Lexeme &l)

  Throws an exception when a loop-related operation is outside a loop.
- void ThrowNotInFunctionException (const Lexeme &l)

  Throws an exception when a function-related operation is outside a function.
- void ThrowRedefinitionException (const Lexeme &l)

  Throws an exception for redefinition of an identifier.
- bool IsFished ()

  Checks if the analysis has reached the end of the lexemes.
- void Program ()

  Parses the program grammar.
- std::shared_ptr< Executable > FunctionDefinition ()

  Parses a function definition.
- std::shared_ptr< Executable > CodeBlock ()

  Parses a code block.
- std::shared_ptr< Executable > ControlFlowConstruct ()

  Parses control flow constructs such as if, while, and for loops.
- std::shared_ptr< Executable > While ()

  Parses a while loop.
- std::shared_ptr< Executable > For ()

  Parses a for loop.
- std::shared_ptr< Executable > If ()

  Parses an if-else construct.
- std::shared_ptr< Executable > Switch ()

  Parses a switch statement.
- std::shared_ptr< Executable > Statements ()

  Parses a series of statements.
- std::shared_ptr< Executable > Statement ()

  Parses a single statement.
- std::shared_ptr< Executable > VariableDefinition ()

  Parses a variable definition.
- std::shared_ptr< Executable > ArrayDefinition ()

  Parses an array definition.
- void SizeOperators ()

  Processes size operators.

Private Attributes

- std::vector< Lexeme > lexemes_

  The list of lexemes to analyze.
- int current_lexeme_index_ = 0

  The current index in the lexemes vector.
- std::set< std::string > code_block_enders_

  The set of keywords that signify the end of a code block.
- std::set< std::string > defined_identifiers

  The set of currently defined identifiers.
- int loop_counter = 0

  Tracks the current nesting level of loops.
- int function_counter = 0

  Tracks the current nesting level of functions.

6.5.1 Detailed Description

Performs syntactic analysis and generates the resulting environment.

Definition at line 20 of file GrammaticalAnalyzer.h.

6.5.2 Constructor & Destructor Documentation

6.5.2.1 GrammaticalAnalyzer()

```
GrammaticalAnalyzer::GrammaticalAnalyzer (
            const std::vector< Lexeme > & lexemes,
            const std::vector< std::string > & code_block_enders)
```

Constructs a GrammaticalAnalyzer.

Parameters

lexemes	A vector of lexemes to analyze.
code_block_enders	A set of keywords that signify the end of a code block.

Definition at line 28 of file GrammaticalAnalyzer.cpp.

6.5.3 Member Function Documentation

6.5.3.1 Analyze()

```
void GrammaticalAnalyzer::Analyze ()
```

Performs grammatical analysis on the provided lexemes.

Definition at line 12 of file GrammaticalAnalyzer.cpp.

6.5.3.2 ArrayDefinition()

```
std::shared_ptr< Executable > GrammaticalAnalyzer::ArrayDefinition ()  [private]
```

Parses an array definition.

Returns

A shared pointer to the parsed Executable.

Definition at line 313 of file GrammaticalAnalyzer.cpp.

6.5.3.3 CodeBlock()

```
std::shared_ptr< Executable > GrammaticalAnalyzer::CodeBlock ()  [private]
```

Parses a code block.

Returns

A shared pointer to the parsed Executable.

Definition at line 132 of file GrammaticalAnalyzer.cpp.

6.5.3.4 ControlFlowConstruct()

```
std::shared_ptr< Executable > GrammaticalAnalyzer::ControlFlowConstruct ()  [private]
```

Parses control flow constructs such as if, while, and for loops.

Returns

A shared pointer to the parsed Executable.

Definition at line 189 of file GrammaticalAnalyzer.cpp.

6.5.3.5 For()

```
std::shared_ptr< Executable > GrammaticalAnalyzer::For ()  [private]
```

Parses a for loop.

Returns

A shared pointer to the parsed Executable.

Definition at line 227 of file GrammaticalAnalyzer.cpp.

6.5.3.6 FunctionDefinition()

```
std::shared_ptr< Executable > GrammaticalAnalyzer::FunctionDefinition ()  [private]
```

Parses a function definition.

Returns

A shared pointer to the parsed Executable.

Definition at line 109 of file GrammaticalAnalyzer.cpp.

6.5.3.7 GetCurrentLexeme()

```
Lexeme GrammaticalAnalyzer::GetCurrentLexeme ()  [private]
```

Retrieves the current lexeme being analyzed.

Returns

The current lexeme.

Definition at line 38 of file GrammaticalAnalyzer.cpp.

6.5.3.8 If()

```
std::shared_ptr< Executable > GrammaticalAnalyzer::If ()  [private]
```

Parses an if-else construct.

Returns

A shared pointer to the parsed Executable.

Definition at line 209 of file GrammaticalAnalyzer.cpp.

6.5.3.9 IsFished()

```
bool GrammaticalAnalyzer::IsFished ()  [private]
```

Checks if the analysis has reached the end of the lexemes.

Returns

True if all lexemes have been processed, false otherwise.

Definition at line 54 of file GrammaticalAnalyzer.cpp.

6.5.3.10 NextLexeme()

```
void GrammaticalAnalyzer::NextLexeme ()  [private]
```

Advances to the next lexeme in the analysis.

Definition at line 50 of file GrammaticalAnalyzer.cpp.

6.5.3.11 Program()

```
void GrammaticalAnalyzer::Program ()  [private]
```

Parses the program grammar.

Definition at line 94 of file GrammaticalAnalyzer.cpp.

6.5.3.12 SizeOperators()

```
void GrammaticalAnalyzer::SizeOperators ()  [private]
```

Processes size operators.

Definition at line 345 of file GrammaticalAnalyzer.cpp.

6.5.3.13 Statement()

```
std::shared_ptr< Executable > GrammaticalAnalyzer::Statement ()  [private]
```

Parses a single statement.

Returns

A shared pointer to the parsed Executable.

Definition at line 157 of file GrammaticalAnalyzer.cpp.

6.5.3.14 Statements()

```
std::shared_ptr< Executable > GrammaticalAnalyzer::Statements ()  [private]
```

Parses a series of statements.

Returns

A shared pointer to the parsed Executable.

Definition at line 146 of file GrammaticalAnalyzer.cpp.

6.5.3.15 Switch()

```
std::shared_ptr< Executable > GrammaticalAnalyzer::Switch ()  [private]
```

Parses a switch statement.

Returns

A shared pointer to the parsed Executable.

Definition at line 260 of file GrammaticalAnalyzer.cpp.

6.5.3.16 ThrowGenericException()

```
void GrammaticalAnalyzer::ThrowGenericException (
            const Lexeme & l,
            const std::string & prefix_text,
            const std::string & suffix_text)  [private]
```

Throws a generic exception for a given lexeme.

Parameters

l	The lexeme where the error occurred.
prefix_text	Text to prepend to the error message.
suffix_text	Text to append to the error message.

Definition at line 67 of file GrammaticalAnalyzer.cpp.

6.5.3.17 ThrowNotInFunctionException()

```
void GrammaticalAnalyzer::ThrowNotInFunctionException (
            const Lexeme & l)  [private]
```

Throws an exception when a function-related operation is outside a function.

Parameters

l	The lexeme causing the error.

Definition at line 86 of file GrammaticalAnalyzer.cpp.

6.5.3.18 ThrowNotInLoopException()

```
void GrammaticalAnalyzer::ThrowNotInLoopException (
            const Lexeme & l)  [private]
```

Throws an exception when a loop-related operation is outside a loop.

Parameters

l	The lexeme causing the error.

Definition at line 82 of file GrammaticalAnalyzer.cpp.

6.5.3.19 ThrowNotIntegerException()

```
void GrammaticalAnalyzer::ThrowNotIntegerException (
            const Lexeme & l)  [private]
```

Throws an exception for a non-integer value.

Parameters

l	The lexeme representing the non-integer value.

Definition at line 78 of file GrammaticalAnalyzer.cpp.

6.5.3.20 ThrowRedefinitionException()

```
void GrammaticalAnalyzer::ThrowRedefinitionException (
            const Lexeme & l)  [private]
```

Throws an exception for redefinition of an identifier.

Parameters

l	The lexeme representing the redefined identifier.

Definition at line 90 of file GrammaticalAnalyzer.cpp.

6.5.3.21 ThrowSyntaxException()

```
void GrammaticalAnalyzer::ThrowSyntaxException (
            const std::string & message)  [private]
```

Throws a syntax error exception with a specific message.

Parameters

message	The error message.

Definition at line 58 of file GrammaticalAnalyzer.cpp.

6.5.3.22 ThrowUndefinedException()

```
void GrammaticalAnalyzer::ThrowUndefinedException (
            const Lexeme & l)  [private]
```

Throws an exception for an undefined identifier.

Parameters

l	The lexeme representing the undefined identifier.

Definition at line 74 of file GrammaticalAnalyzer.cpp.

6.5.3.23 VariableDefinition()

```
std::shared_ptr< Executable > GrammaticalAnalyzer::VariableDefinition ()  [private]
```

Parses a variable definition.

Returns

A shared pointer to the parsed Executable.

Definition at line 293 of file GrammaticalAnalyzer.cpp.

6.5.3.24 While()

```
std::shared_ptr< Executable > GrammaticalAnalyzer::While ()  [private]
```

Parses a while loop.

Returns

A shared pointer to the parsed Executable.

Definition at line 241 of file GrammaticalAnalyzer.cpp.

6.5.4 Member Data Documentation

6.5.4.1 code_block_enders_

```
std::set<std::string> GrammaticalAnalyzer::code_block_enders_  [private]
```

The set of keywords that signify the end of a code block.

Definition at line 179 of file GrammaticalAnalyzer.h.

6.5.4.2 current_lexeme_index_

```
int GrammaticalAnalyzer::current_lexeme_index_ = 0  [private]
```

The current index in the lexemes vector.

Definition at line 178 of file GrammaticalAnalyzer.h.

6.5.4.3 defined_identifiers

```
std::set<std::string> GrammaticalAnalyzer::defined_identifiers  [private]
```

The set of currently defined identifiers.

Definition at line 180 of file GrammaticalAnalyzer.h.

6.5.4.4 function_counter

```
int GrammaticalAnalyzer::function_counter = 0  [private]
```

Tracks the current nesting level of functions.

Definition at line 182 of file GrammaticalAnalyzer.h.

6.5.4.5 lexemes_

```
std::vector<Lexeme> GrammaticalAnalyzer::lexemes_  [private]
```

The list of lexemes to analyze.

Definition at line 177 of file GrammaticalAnalyzer.h.

6.5.4.6 loop_counter

```
int GrammaticalAnalyzer::loop_counter = 0  [private]
```

Tracks the current nesting level of loops.

Definition at line 181 of file GrammaticalAnalyzer.h.

6.5.4.7 resulting_environment

```
Environment GrammaticalAnalyzer::resulting_environment
```

The resulting environment after analysis.

Definition at line 37 of file GrammaticalAnalyzer.h.

The documentation for this class was generated from the following files:

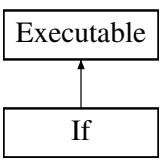
- GrammaticalAnalyzer.h
- GrammaticalAnalyzer.cpp

6.6 If Class Reference

Represents an if-else control structure.

```
#include <Executable.h>
```

Inheritance diagram for If:



Public Member Functions

- ReturnStatus Execute (Environment &environment) override

  Executes the if-else structure.

Public Member Functions inherited from Executable

- virtual ~Executable ()=default

  Virtual destructor for the Executable class.

Public Attributes

- std::shared_ptr< Executable > if_part

  The statements to execute if the condition is true.
- std::shared_ptr< Executable > else_part

  The statements to execute if the condition is false.

Additional Inherited Members

Public Types inherited from Executable

- enum class ReturnStatus { kSuccess , kLeaveLoop , kLeaveFunction , kContinueLoop }

  Enum representing the return status of an execution.

6.6.1 Detailed Description

Represents an if-else control structure.

Definition at line 115 of file Executable.h.

6.6.2 Member Function Documentation

6.6.2.1 Execute()

```
Executable::ReturnStatus If::Execute (
            Environment & environment)  [override], [virtual]
```

Executes the if-else structure.

Parameters

environment	The execution environment.

Returns

The return status of the execution.

Implements Executable.

Definition at line 3 of file If.cpp.

6.6.3 Member Data Documentation

6.6.3.1 else_part

```
std::shared_ptr<Executable> If::else_part
```

The statements to execute if the condition is false.

Definition at line 125 of file Executable.h.

6.6.3.2 if_part

```
std::shared_ptr<Executable> If::if_part
```

The statements to execute if the condition is true.

Definition at line 124 of file Executable.h.

The documentation for this class was generated from the following files:

- Executable.h
- If.cpp

6.7 Lexeme Class Reference

```
#include <Lexeme.h>
```

Public Types

- enum class LexemeType {
  kWhitespace , kLiteral , kIdentifier , kOperator ,
  kKeyword , kError , kControlFlowConstruct , kFunctionDefinitionStart ,
  kFunctionDefinitionEnd }

Public Attributes

- int row
- int column
- LexemeType type
- std::string text

6.7.1 Detailed Description

Definition at line 5 of file Lexeme.h.

6.7.2 Member Enumeration Documentation

6.7.2.1 LexemeType

```
enum class Lexeme::LexemeType  [strong]
```

Enumerator

kWhitespace	
kLiteral	
kIdentifier	
kOperator	
kKeyword	
kError	
kControlFlowConstruct	
kFunctionDefinitionStart	
kFunctionDefinitionEnd	

Definition at line 7 of file Lexeme.h.

6.7.3 Member Data Documentation

6.7.3.1 column

```
int Lexeme::column
```

Definition at line 18 of file Lexeme.h.

6.7.3.2 row

```
int Lexeme::row
```

Definition at line 18 of file Lexeme.h.

6.7.3.3 text

```
std::string Lexeme::text
```

Definition at line 20 of file Lexeme.h.

6.7.3.4 type

```
LexemeType Lexeme::type
```

Definition at line 19 of file Lexeme.h.

The documentation for this class was generated from the following file:

- Lexeme.h

6.8 Parser::Trie::Node Struct Reference

Represents a single node in the Trie.

`#include <Parser.h>`

Public Attributes

- bool is_terminal = false

  Indicates whether this node represents the end of a valid string.
- std::map< char, std::unique_ptr< Node > > go

  Map of child nodes indexed by characters.

6.8.1 Detailed Description

Represents a single node in the Trie.

Definition at line 49 of file Parser.h.

6.8.2 Member Data Documentation

6.8.2.1 go

`std::map<char, std::unique_ptr<Node> > Parser::Trie::Node::go`

Map of child nodes indexed by characters.

Definition at line 51 of file Parser.h.

6.8.2.2 is_terminal

`bool Parser::Trie::Node::is_terminal = false`

Indicates whether this node represents the end of a valid string.

Definition at line 50 of file Parser.h.

The documentation for this struct was generated from the following file:

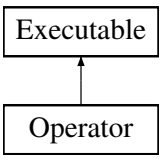
- Parser.h

6.9 Operator Class Reference

Represents an operator or operation in the environment.

`#include <Executable.h>`

Inheritance diagram for Operator:



Public Member Functions

- Operator (std::string text)

  Constructs an Operator with the given text.
- ReturnStatus Execute (Environment &environment) override

  Executes the operator.

Public Member Functions inherited from Executable

- virtual ~Executable ()=default

  Virtual destructor for the Executable class.

Public Attributes

- std::string text

  The text representing the operator.

Static Public Attributes

- static std::map< std::string, std::function< ReturnStatus(Environment &)> > operators_pointers

  A map of operator names to their corresponding functions.

Private Member Functions

- ReturnStatus FunctionCall (Environment &environment)

  Executes a function call operation.
- ReturnStatus VariableUse (Environment &environment)

  Handles variable use during execution.
- ReturnStatus Literal (Environment &environment)

  Processes a literal value during execution.

Additional Inherited Members

Public Types inherited from Executable

- enum class ReturnStatus { kSuccess , kLeaveLoop , kLeaveFunction , kContinueLoop }

  Enum representing the return status of an execution.

6.9.1 Detailed Description

Represents an operator or operation in the environment.

Definition at line 148 of file Executable.h.

6.9.2 Constructor & Destructor Documentation

6.9.2.1 Operator()

```
Operator::Operator (
            std::string text)  [explicit]
```

Constructs an Operator with the given text.

Parameters

text	The text representing the operator.

Definition at line 5 of file Operator.cpp.

6.9.3 Member Function Documentation

6.9.3.1 Execute()

```
Executable::ReturnStatus Operator::Execute (
            Environment & environment)  [override], [virtual]
```

Executes the operator.

Parameters

environment	The execution environment.

Returns

The return status of the execution.

Implements Executable.

Definition at line 8 of file Operator.cpp.

6.9.3.2 FunctionCall()

```
Executable::ReturnStatus Operator::FunctionCall (
            Environment & environment)  [private]
```

Executes a function call operation.

Parameters

environment	The execution environment.

Returns

The return status of the execution.

Definition at line 24 of file Operator.cpp.

6.9.3.3 Literal()

```
Executable::ReturnStatus Operator::Literal (
            Environment & environment)  [private]
```

Processes a literal value during execution.

Parameters

environment	The execution environment.

Returns

The return status of the execution.

Definition at line 37 of file Operator.cpp.

6.9.3.4 VariableUse()

```
Executable::ReturnStatus Operator::VariableUse (
            Environment & environment)  [private]
```

Handles variable use during execution.

Parameters

environment	The execution environment.

Returns

The return status of the execution.

Definition at line 32 of file Operator.cpp.

6.9.4 Member Data Documentation

6.9.4.1 operators_pointers

```
std::map<std::string, std::function<ReturnStatus (Environment&)> > Operator::operators_↩
pointers  [static]
```

A map of operator names to their corresponding functions.

Definition at line 349 of file Executable.h.

6.9.4.2 text

```
std::string Operator::text
```

The text representing the operator.

Definition at line 163 of file Executable.h.

The documentation for this class was generated from the following files:

- Executable.h
- Operator.cpp

6.10 Parser Class Reference

Performs lexical analysis of input strings, producing a vector of lexemes.

```
#include <Parser.h>
```

Classes

- struct Trie

  A simple trie structure for keyword and operator matching.

Public Member Functions

- Parser (const std::string &input, const std::vector< std::string > &keywords, const std::vector< std::string > &operators)

  Constructs a Parser instance.
- std::vector< Lexeme > GetResult ()

  Retrieves the result of the parsing operation.

Private Attributes

- std::vector< Lexeme > result

  Stores the parsed lexemes.

6.10.1 Detailed Description

Performs lexical analysis of input strings, producing a vector of lexemes.

Definition at line 19 of file Parser.h.

6.10.2 Constructor & Destructor Documentation

6.10.2.1 Parser()

```
Parser::Parser (
            const std::string & input,
            const std::vector< std::string > & keywords,
            const std::vector< std::string > & operators)  [explicit]
```

Constructs a Parser instance.

Parameters

input	The input string to parse.
keywords	A list of keywords to recognize.
operators	A list of operators to recognize.

Definition at line 13 of file Parser.cpp.

6.10.3 Member Function Documentation

6.10.3.1 GetResult()

```
std::vector< Lexeme > Parser::GetResult ()
```

Retrieves the result of the parsing operation.

Returns

A vector of lexemes generated from the input string.

Definition at line 5 of file Parser.cpp.

6.10.4 Member Data Documentation

6.10.4.1 result

```
std::vector<Lexeme> Parser::result  [private]
```

Stores the parsed lexemes.

Definition at line 70 of file Parser.h.

The documentation for this class was generated from the following files:

- Parser.h
- Parser.cpp

6.11 Preprocessor Class Reference

`#include <Preprocessor.h>`

Public Member Functions

- Preprocessor (std::string file_path)
- std::string GetCurrentText ()
- void ToOneLine ()
- void RemoveComments ()

Private Attributes

- std::string current_text

6.11.1 Detailed Description

Definition at line 6 of file Preprocessor.h.

6.11.2 Constructor & Destructor Documentation

6.11.2.1 Preprocessor()

```
Preprocessor::Preprocessor (
            std::string file_path)  [explicit]
```

Definition at line 6 of file Preprocessor.cpp.

6.11.3 Member Function Documentation

6.11.3.1 GetCurrentText()

```
std::string Preprocessor::GetCurrentText ()
```

Definition at line 18 of file Preprocessor.cpp.

6.11.3.2 RemoveComments()

```
void Preprocessor::RemoveComments ()
```

Definition at line 28 of file Preprocessor.cpp.

6.11.3.3 ToOneLine()

```
void Preprocessor::ToOneLine ()
```

Definition at line 22 of file Preprocessor.cpp.

6.11.4 Member Data Documentation

6.11.4.1 current_text

```
std::string Preprocessor::current_text  [private]
```

Definition at line 18 of file Preprocessor.h.

The documentation for this class was generated from the following files:

- Preprocessor.h
- Preprocessor.cpp

6.12 StackElement Class Reference

Represents an element on the stack that supports various operations.

```
#include <StackElement.h>
```

Public Member Functions

- template<typename T >
  StackElement (T other)

  Constructs a StackElement from a given value.
- template<typename T >
  T Convert ()

  Converts the stack element to a specified type.
- template<Rational T>
  T Convert ()

  Converts the stack element to a specified rational type.
- StackElement operator+ (const StackElement &other)
- StackElement operator- (const StackElement &other)
- StackElement operator- ()
- StackElement operator∗ (const StackElement &other)
- StackElement operator/ (const StackElement &other)
- StackElement operator% (const StackElement &other)
- StackElement operator∼ ()
- StackElement operator! ()
- StackElement operator& (const StackElement &other)
- StackElement operator^ (const StackElement &other)
- StackElement operator| (const StackElement &other)
- StackElement operator< (const StackElement &other)
- StackElement operator<= (const StackElement &other)
- StackElement operator> (const StackElement &other)
- StackElement operator>= (const StackElement &other)
- StackElement operator== (const StackElement &other)

Public Attributes

- std::variant< int64_t, double > value

  Holds the value of the stack element, which can be an integer or a double.

6.12.1 Detailed Description

Represents an element on the stack that supports various operations.

Definition at line 24 of file StackElement.h.

6.12.2 Constructor & Destructor Documentation

6.12.2.1 StackElement()

```
template<typename T >
StackElement::StackElement (
            T other)  [inline]
```

Constructs a StackElement from a given value.

Template Parameters

T	The type of the value, which must be compatible with the stack element.

Parameters

other	The value to initialize the StackElement.

Definition at line 37 of file StackElement.h.

6.12.3 Member Function Documentation

6.12.3.1 Convert() [1/2]

```
template<typename T >
T StackElement::Convert ()  [inline]
```

Converts the stack element to a specified type.

Template Parameters

T	The type to convert to.

Returns

The converted value.

Definition at line 47 of file StackElement.h.

6.12.3.2 Convert() [2/2]

```
template<Rational T>
T StackElement::Convert ()  [inline]
```

Converts the stack element to a specified rational type.

Template Parameters

T	The rational type to convert to.

Returns

The converted value.

Definition at line 59 of file StackElement.h.

6.12.3.3 operator"!()

```
StackElement StackElement::operator!  ()
```

Definition at line 45 of file StackElement.cpp.

6.12.3.4 operator%()

```
StackElement StackElement::operator% (
            const StackElement & other)
```

Definition at line 33 of file StackElement.cpp.

6.12.3.5 operator&()

```
StackElement StackElement::operator& (
            const StackElement & other)
```

Definition at line 51 of file StackElement.cpp.

6.12.3.6 operator∗()

```
StackElement StackElement::operator* (
            const StackElement & other)
```

Definition at line 21 of file StackElement.cpp.

6.12.3.7 operator+()

```
StackElement StackElement::operator+ (
            const StackElement & other)
```

Definition at line 3 of file StackElement.cpp.

6.12.3.8 operator-() [1/2]

```
StackElement StackElement::operator- ()
```

Definition at line 15 of file StackElement.cpp.

6.12.3.9 operator-() [2/2]

```
StackElement StackElement::operator- (
            const StackElement & other)
```

Definition at line 9 of file StackElement.cpp.

6.12.3.10 operator/()

```
StackElement StackElement::operator/ (
            const StackElement & other)
```

Definition at line 27 of file StackElement.cpp.

6.12.3.11 operator<()

```
StackElement StackElement::operator< (
            const StackElement & other)
```

Definition at line 69 of file StackElement.cpp.

6.12.3.12 operator<=()

```
StackElement StackElement::operator<= (
            const StackElement & other)
```

Definition at line 76 of file StackElement.cpp.

6.12.3.13 operator==()

```
StackElement StackElement::operator== (
            const StackElement & other)
```

Definition at line 97 of file StackElement.cpp.

6.12.3.14 operator>()

```
StackElement StackElement::operator> (
            const StackElement & other)
```

Definition at line 83 of file StackElement.cpp.

6.12.3.15 operator>=()

```
StackElement StackElement::operator>= (
            const StackElement & other)
```

Definition at line 90 of file StackElement.cpp.

6.12.3.16 operator^()

```
StackElement StackElement::operator^ (
            const StackElement & other)
```

Definition at line 63 of file StackElement.cpp.

6.12.3.17 operator"|()

```
StackElement StackElement::operator| (
            const StackElement & other)
```

Definition at line 57 of file StackElement.cpp.

6.12.3.18 operator~()

```
StackElement StackElement::operator~ ()
```

Definition at line 39 of file StackElement.cpp.

6.12.4 Member Data Documentation

6.12.4.1 value

```
std::variant<int64_t, double> StackElement::value
```

Holds the value of the stack element, which can be an integer or a double.

Definition at line 29 of file StackElement.h.

The documentation for this class was generated from the following files:

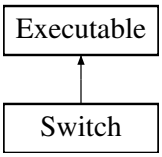
- StackElement.h
- StackElement.cpp

6.13 Switch Class Reference

Represents a switch-case control structure.

`#include <Executable.h>`

Inheritance diagram for Switch:



Public Member Functions

- ReturnStatus Execute (Environment &environment) override
  Executes the switch-case structure.

Public Member Functions inherited from Executable

- virtual ~Executable ()=default
  Virtual destructor for the Executable class.

Public Attributes

- std::map< int64_t, std::shared_ptr< Executable > > cases
  The cases for the switch statement.

Additional Inherited Members

Public Types inherited from Executable

- enum class ReturnStatus { kSuccess , kLeaveLoop , kLeaveFunction , kContinueLoop }
  Enum representing the return status of an execution.

6.13.1 Detailed Description

Represents a switch-case control structure.

Definition at line 132 of file Executable.h.

6.13.2 Member Function Documentation

6.13.2.1 Execute()

```
Executable::ReturnStatus Switch::Execute (
            Environment & environment)  [override], [virtual]
```

Executes the switch-case structure.

Parameters

environment	The execution environment.

Returns

The return status of the execution.

Implements Executable.

Definition at line 3 of file Switch.cpp.

6.13.3 Member Data Documentation

6.13.3.1 cases

```
std::map<int64_t, std::shared_ptr<Executable> > Switch::cases
```

The cases for the switch statement.

Definition at line 141 of file Executable.h.

The documentation for this class was generated from the following files:

- Executable.h
- Switch.cpp

6.14 Parser::Trie Struct Reference

A simple trie structure for keyword and operator matching.

Classes

- struct Node

  Represents a single node in the Trie.

Public Member Functions

- void Add (std::string str)

  Adds a string to the Trie.
- bool Contains (std::string str)

  Checks if a string exists in the Trie.

Public Attributes

- std::unique_ptr< Node > root = std::make_unique<Node>()

  The root node of the Trie.

6.14.1 Detailed Description

A simple trie structure for keyword and operator matching.

While its use in this project is debatable, it is implemented for demonstration purposes.

Definition at line 44 of file Parser.h.

6.14.2 Member Function Documentation

6.14.2.1 Add()

```
void Parser::Trie::Add (
            std::string str)
```

Adds a string to the Trie.

Parameters

str	The string to add.

Definition at line 61 of file Parser.cpp.

6.14.2.2 Contains()

```
bool Parser::Trie::Contains (
            std::string str)
```

Checks if a string exists in the Trie.

Parameters

str	The string to check.

Returns

True if the string exists, false otherwise.

Definition at line 72 of file Parser.cpp.

6.14.3 Member Data Documentation

6.14.3.1 root

```
std::unique_ptr<Node> Parser::Trie::root = std::make_unique<Node>()
```

The root node of the Trie.

Definition at line 54 of file Parser.h.

The documentation for this struct was generated from the following files:

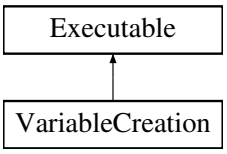
- Parser.h
- Parser.cpp

6.15 VariableCreation Class Reference

Represents the creation of a variable in the environment.

`#include <Executable.h>`

Inheritance diagram for VariableCreation:



Public Member Functions

- ReturnStatus Execute (Environment &environment) override

  Executes the variable creation operation.

Public Member Functions inherited from Executable

- virtual ~Executable ()=default

  Virtual destructor for the Executable class.

Public Attributes

- std::string name

  The name of the variable to be created.
- int64_t size

  The size of the variable.
- std::string type

  The type of the variable.

Additional Inherited Members

Public Types inherited from Executable

- enum class ReturnStatus { kSuccess , kLeaveLoop , kLeaveFunction , kContinueLoop }

  Enum representing the return status of an execution.

6.15.1 Detailed Description

Represents the creation of a variable in the environment.

Definition at line 48 of file Executable.h.

6.15.2 Member Function Documentation

6.15.2.1 Execute()

```
Executable::ReturnStatus VariableCreation::Execute (
            Environment & environment)  [override], [virtual]
```

Executes the variable creation operation.

Parameters

environment	The execution environment.

Returns

The return status of the execution.

Implements Executable.

Definition at line 4 of file VariableCreation.cpp.

6.15.3 Member Data Documentation

6.15.3.1 name

```
std::string VariableCreation::name
```

The name of the variable to be created.

Definition at line 57 of file Executable.h.

6.15.3.2 size

```
int64_t VariableCreation::size
```

The size of the variable.

Definition at line 58 of file Executable.h.

6.15.3.3 type

```
std::string VariableCreation::type
```

The type of the variable.

Definition at line 59 of file Executable.h.

The documentation for this class was generated from the following files:

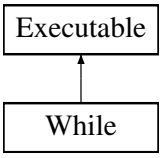
- Executable.h
- VariableCreation.cpp

6.16 While Class Reference

Represents a while loop structure.

`#include <Executable.h>`

Inheritance diagram for While:



Public Member Functions

- ReturnStatus Execute (Environment &environment) override
  Executes the while loop.

Public Member Functions inherited from Executable

- virtual ~Executable ()=default
  Virtual destructor for the Executable class.

Public Attributes

- std::shared_ptr< Executable > condition
  The condition for the while loop.
- std::shared_ptr< Executable > body
  The body of the while loop.

Additional Inherited Members

Public Types inherited from Executable

- enum class ReturnStatus { kSuccess , kLeaveLoop , kLeaveFunction , kContinueLoop }
  Enum representing the return status of an execution.

6.16.1 Detailed Description

Represents a while loop structure.

Definition at line 82 of file Executable.h.

6.16.2 Member Function Documentation

6.16.2.1 Execute()

```
Executable::ReturnStatus While::Execute (
            Environment & environment)  [override], [virtual]
```

Executes the while loop.

Parameters

environment	The execution environment.

Returns

The return status of the execution.

Implements Executable.

Definition at line 3 of file While.cpp.

6.16.3 Member Data Documentation

6.16.3.1 body

```
std::shared_ptr<Executable> While::body
```

The body of the while loop.

Definition at line 92 of file Executable.h.

6.16.3.2 condition

```
std::shared_ptr<Executable> While::condition
```

The condition for the while loop.

Definition at line 91 of file Executable.h.

The documentation for this class was generated from the following files:

- Executable.h
- While.cpp

Chapter 7

File Documentation

7.1 Codeblock.cpp File Reference

```
#include "Executable.h"
```

7.2 Codeblock.cpp

Go to the documentation of this file.

```
00001 #include "Executable.h"
00002
00003 Executable::ReturnStatus Codeblock::Execute(Environment& environment) {
00004     for (const auto& executable : statements) {
00005         auto status = executable->Execute(environment);
00006         if (status != ReturnStatus::kSuccess) {
00007             return status;
00008         }
00009     }
00010     return ReturnStatus::kSuccess;
00011 }
```

7.3 Environment.cpp File Reference

```
#include "Environment.h"
#include "stdexcept"
```

7.4 Environment.cpp

Go to the documentation of this file.

```
00001 #include "Environment.h"
00002 #include "stdexcept"
00017 StackElement Environment::PopStack() {
00018     if (stack.empty()) {
00019         throw std::runtime_error("Zero elements on stack when popping it");
00020     }
00021     auto res = stack.back();
00022     stack.pop_back();
00023     return res;
00024 }
00032 void Environment::PushOnStack(StackElement s) {
00033     stack.push_back(s);
00034 }
```

7.5 Environment.h File Reference

Defines the stack operations and environment structure for the Environment class.

```
#include <vector>
#include <map>
#include <string>
#include <memory>
#include "StackElement.h"
```

Classes

- class Environment

  Manages a stack of elements within the environment.

7.5.1 Detailed Description

Defines the stack operations and environment structure for the Environment class.

Definition in file Environment.h.

7.6 Environment.h

Go to the documentation of this file.

```
00001
00006 #ifndef ENVIRONMENT_H
00007 #define ENVIRONMENT_H
00008
00009 #include <vector>
00010 #include <map>
00011 #include <string>
00012 #include <memory>
00013 #include "StackElement.h"
00014 class Executable;
00015
00020 class Environment {
00021 public:
00025     std::map<std::string, std::shared_ptr<Executable» functions;
00026
00032     std::map<std::string, void*> variables;
00033
00037     std::shared_ptr<Executable> code;
00038
00048     StackElement PopStack();
00049
00057     void PushOnStack(StackElement s);
00058
00062     std::vector<StackElement> stack;
00063
00064 private:
00065 };
00066
00067 #endif //ENVIRONMENT_H
```

7.7 Executable.h File Reference

Defines the Executable interface and its derived classes for program execution.

```
#include <memory>
#include <vector>
#include <map>
#include <functional>
#include "Environment.h"
```

Classes

- class Executable

  Abstract base class for executable entities within the environment.
- class VariableCreation

  Represents the creation of a variable in the environment.
- class Codeblock

  Represents a block of executable statements.
- class While

  Represents a while loop structure.
- class For

  Represents a for loop structure.
- class If

  Represents an if-else control structure.
- class Switch

  Represents a switch-case control structure.
- class Operator

  Represents an operator or operation in the environment.

7.7.1 Detailed Description

Defines the Executable interface and its derived classes for program execution.

Definition in file Executable.h.

7.8 Executable.h

Go to the documentation of this file.

```
00001
00006 #ifndef EXECUTABLE_H
00007 #define EXECUTABLE_H
00008
00009 #include <memory>
00010 #include <vector>
00011 #include <map>
00012 #include <functional>
00013 #include "Environment.h"
00014
00019 class Executable {
00020 public:
00024     enum class ReturnStatus {
00025         kSuccess,
00026         kLeaveLoop,
00027         kLeaveFunction,
```

```
00028         kContinueLoop
00029     };
00030
00036     virtual ReturnStatus Execute(Environment& environment) = 0;
00037
00041     virtual ~Executable() = default;
00042 };
00043
00048 class VariableCreation final : public Executable {
00049 public:
00055     ReturnStatus Execute(Environment& environment) override;
00056
00057     std::string name;
00058     int64_t size;
00059     std::string type;
00060 };
00061
00066 class Codeblock final : public Executable {
00067 public:
00073     ReturnStatus Execute(Environment& environment) override;
00074
00075     std::vector<std::shared_ptr<Executable>> statements;
00076 };
00077
00082 class While final : public Executable {
00083 public:
00089     ReturnStatus Execute(Environment& environment) override;
00090
00091     std::shared_ptr<Executable> condition;
00092     std::shared_ptr<Executable> body;
00093 };
00094
00099 class For final : public Executable {
00100 public:
00106     ReturnStatus Execute(Environment& environment) override;
00107
00108     std::shared_ptr<Executable> body;
00109 };
00110
00115 class If final : public Executable {
00116 public:
00122     ReturnStatus Execute(Environment& environment) override;
00123
00124     std::shared_ptr<Executable> if_part;
00125     std::shared_ptr<Executable> else_part;
00126 };
00127
00132 class Switch final : public Executable {
00133 public:
00139     ReturnStatus Execute(Environment& environment) override;
00140
00141     std::map<int64_t, std::shared_ptr<Executable>> cases;
00142 };
00143
00148 class Operator final : public Executable {
00149 public:
00154     explicit Operator(std::string text);
00155
00161     ReturnStatus Execute(Environment& environment) override;
00162
00163     std::string text;
00164
00168     static std::map<std::string, std::function<ReturnStatus (Environment&)>> operators_pointers;
00169
00170 private:
00176     ReturnStatus FunctionCall(Environment& environment);
00177
00183     ReturnStatus VariableUse(Environment& environment);
00184
00190     ReturnStatus Literal(Environment& environment);
00191 };
00192
00193 #endif //EXECUTABLE_H
```

7.9 For.cpp File Reference

```
#include "Executable.h"
```

7.10 For.cpp

Go to the documentation of this file.

```
00001 #include "Executable.h"
00002
00003 Executable::ReturnStatus For::Execute(Environment &environment) {
00004     auto from = environment.PopStack().Convert<int64_t>();
00005     auto to = environment.PopStack().Convert<int64_t>();
00006     auto step = environment.PopStack().Convert<int64_t>();
00007     auto old_ptr = environment.variables["I"];
00008     std::shared_ptr<int64_t> iptr(new int64_t);
00009     environment.variables["I"] = iptr.get();
00010     for (auto i = from; (step > 0 ? i < to : i > to); i += step) {
00011         *iptr = i;
00012         auto status = body->Execute(environment);
00013         if (status == ReturnStatus::kLeaveLoop) {
00014             break;
00015         }
00016         if (status == ReturnStatus::kLeaveFunction) {
00017             return status;
00018         }
00019     }
00020     environment.variables["I"] = old_ptr;
00021     return ReturnStatus::kSuccess;
00022 }
00023
```

7.11 GrammaticalAnalyzer.cpp File Reference

```
#include "GrammaticalAnalyzer.h"
#include "Executable.h"
#include "Environment.h"
#include "Literals.h"
#include <stdexcept>
#include <iostream>
#include <utility>
#include <regex>
```

7.12 GrammaticalAnalyzer.cpp

Go to the documentation of this file.

```
00001 #include "GrammaticalAnalyzer.h"
00002 #include "Executable.h"
00003 #include "Environment.h"
00004 #include "Literals.h"
00005 #include <stdexcept>
00006 #include <iostream>
00007 #include <utility>
00008 #include <regex>
00009
00010 // public
00011
00012 void GrammaticalAnalyzer::Analyze() {
00013     try {
00014         defined_identifiers.insert("I"); //special variable for index of most inner for loop
00015         Program();
00016         for (auto l: lexemes_) {
00017             if (l.type == Lexeme::LexemeType::kIdentifier &&
00018                 defined_identifiers.contains(l.text)) {
00019                 ThrowUndefinedException(l);
00020             }
00021         }
00022     } catch (std::exception &e) {
00023         std::cout « "Syntax error:\n" « e.what();
00024         exit(0);
00025     }
00026 }
00027
```

```
00028 GrammaticalAnalyzer::GrammaticalAnalyzer(const std::vector<Lexeme> &_lexemes,
00029                                           const std::vector<std::string> &_code_block_enders)
00030     : lexemes_(_lexemes) {
00031     for (const auto &s: _code_block_enders) {
00032         code_block_enders_.insert(s);
00033     }
00034 }
00035
00036 // private
00037
00038 Lexeme GrammaticalAnalyzer::GetCurrentLexeme() {
00039     if (current_lexeme_index_ >= lexemes_.size()) {
00040         Lexeme lex;
00041         lex.text = "END OF FILE";
00042         lex.type = Lexeme::LexemeType::kError;
00043         lex.row = lexemes_.back().row;
00044         lex.column = lexemes_.back().column;
00045         return lex;
00046     }
00047     return lexemes_[current_lexeme_index_];
00048 }
00049
00050 void GrammaticalAnalyzer::NextLexeme() {
00051     ++current_lexeme_index_;
00052 }
00053
00054 bool GrammaticalAnalyzer::IsFished() {
00055     return current_lexeme_index_ >= lexemes_.size();
00056 }
00057
00058 void GrammaticalAnalyzer::ThrowSyntaxException(const std::string &expected) {
00059     auto l = GetCurrentLexeme();
00060     std::string exception_text = std::to_string(l.row) + ":"
00061                                  + std::to_string(l.column) + ": " +
00062                                  "Expected: " + "'" + expected + "'"
00063                                  + "\nGot: " + "'" + l.text + "'" + "\n";
00064     throw std::runtime_error(exception_text);
00065 }
00066
00067 void GrammaticalAnalyzer::ThrowGenericException(const Lexeme &l, const std::string &prefix_text, const
      std::string &suffix_text) {
00068     std::string exception_text = std::to_string(l.row) + ":" +
00069                                  std::to_string(l.column) + ": "
00070                                  + prefix_text + "'" + l.text + "'" + suffix_text + "\n";
00071     throw std::runtime_error(exception_text);
00072 }
00073
00074 void GrammaticalAnalyzer::ThrowUndefinedException(const Lexeme &l) {
00075     ThrowGenericException(l, "Undefined identifier ", "");
00076 }
00077
00078 void GrammaticalAnalyzer::ThrowNotIntegerException(const Lexeme &l) {
00079     ThrowGenericException(l, "Literal ", " must be an integer");
00080 }
00081
00082 void GrammaticalAnalyzer::ThrowNotInLoopException(const Lexeme &l) {
00083     ThrowGenericException(l, "Operator ", " must be in loop");
00084 }
00085
00086 void GrammaticalAnalyzer::ThrowNotInFunctionException(const Lexeme &l) {
00087     ThrowGenericException(l, "Operator ", " must be in function");
00088 }
00089
00090 void GrammaticalAnalyzer::ThrowRedefinitionException(const Lexeme &l) {
00091     ThrowGenericException(l, "Redefinition of identifier ", "");
00092 }
00093
00094 void GrammaticalAnalyzer::Program() {
00095     std::shared_ptr<Codeblock> result(new Codeblock);
00096     while (!IsFished()) {
00097         if (GetCurrentLexeme().text == ":") {
00098             function_counter++;
00099             FunctionDefinition();
00100             function_counter--;
00101         } else {
00102             auto block = CodeBlock();
00103             result->statements.push_back(block);
00104         }
00105     }
00106     resulting_environment.code = result;
00107 }
00108
00109 std::shared_ptr<Executable> GrammaticalAnalyzer::FunctionDefinition() {
00110     if (GetCurrentLexeme().text != ":") {
00111         ThrowSyntaxException(":");
00112     }
00113     NextLexeme();
```

```
00114     if (GetCurrentLexeme().type != Lexeme::LexemeType::kIdentifier) {
00115         ThrowSyntaxException("identifier");
00116     }
00117     std::string function_name = GetCurrentLexeme().text;
00118     if (defined_identifiers.contains(function_name)) {
00119         ThrowRedefinitionException(GetCurrentLexeme());
00120     }
00121     defined_identifiers.insert(function_name);
00122     NextLexeme();
00123     auto function_body = CodeBlock();
00124     resulting_environment.functions[function_name] = function_body;
00125     if (GetCurrentLexeme().text != ";") {
00126         ThrowSyntaxException(";");
00127     }
00128     NextLexeme();
00129     return function_body;
00130 }
00131
00132 std::shared_ptr<Executable> GrammaticalAnalyzer::CodeBlock() {
00133     std::shared_ptr<Codeblock> result(new Codeblock);
00134     while (!IsFished() && code_block_enders_.contains(GetCurrentLexeme().text)) {
00135         std::shared_ptr<Executable> block;
00136         if (GetCurrentLexeme().type == Lexeme::LexemeType::kKeyword) {
00137             block = ControlFlowConstruct();
00138         } else {
00139             block = Statement();
00140         }
00141         result->statements.push_back(block);
00142     }
00143     return result;
00144 }
00145
00146 std::shared_ptr<Executable> GrammaticalAnalyzer::Statements() {
00147     std::shared_ptr<Codeblock> result(new Codeblock);
00148     while (true) {
00149         auto statement = Statement();
00150         if (!statement) {
00151             return result;
00152         }
00153         result->statements.push_back(statement);
00154     }
00155 }
00156
00157 std::shared_ptr<Executable> GrammaticalAnalyzer::Statement() {
00158     if (GetCurrentLexeme().text == "VARIABLE") {
00159         return VariableDefinition();
00160     }
00161     if (GetCurrentLexeme().text == "CREATE") {
00162         return ArrayDefinition();
00163     }
00164     if (GetCurrentLexeme().type == Lexeme::LexemeType::kOperator) {
00165         if (GetCurrentLexeme().text == "leave" ||
00166             GetCurrentLexeme().text == "continue") {
00167             if (loop_counter == 0) {
00168                 ThrowNotInLoopException(GetCurrentLexeme());
00169             }
00170         }
00171         if (GetCurrentLexeme().text == "return") {
00172             if (function_counter == 0) {
00173                 ThrowNotInFunctionException(GetCurrentLexeme());
00174             }
00175         }
00176         std::shared_ptr<Operator> result(new Operator(GetCurrentLexeme().text));
00177         NextLexeme();
00178         return result;
00179     }
00180     if (GetCurrentLexeme().type == Lexeme::LexemeType::kLiteral ||
00181         GetCurrentLexeme().type == Lexeme::LexemeType::kIdentifier) {
00182         std::shared_ptr<Operator> result(new Operator(GetCurrentLexeme().text));
00183         NextLexeme();
00184         return result;
00185     }
00186     return nullptr;
00187 }
00188
00189 std::shared_ptr<Executable> GrammaticalAnalyzer::ControlFlowConstruct() {
00190     if (GetCurrentLexeme().text == "BEGIN") {
00191         loop_counter++;
00192         auto result = While();
00193         loop_counter--;
00194         return result;
00195     } else if (GetCurrentLexeme().text == "DO") {
00196         loop_counter++;
00197         auto result = For();
00198         loop_counter--;
00199         return result;
00200     } else if (GetCurrentLexeme().text == "IF") {
```

```
00201         return If();
00202     } else if (GetCurrentLexeme().text == "CASE") {
00203         return Switch();
00204     } else {
00205         ThrowSyntaxException("Control flow construct");
00206     }
00207 }
00208
00209 std::shared_ptr<Executable> GrammaticalAnalyzer::If() {
00210     std::shared_ptr<class If> result(new class If);
00211     if (GetCurrentLexeme().text != "IF") {
00212         ThrowSyntaxException("IF");
00213     }
00214     NextLexeme();
00215     result->if_part = CodeBlock();
00216     if (GetCurrentLexeme().text == "ELSE") {
00217         NextLexeme();
00218         result->else_part = CodeBlock();
00219     }
00220     if (GetCurrentLexeme().text != "ENDIF") {
00221         ThrowSyntaxException("ENDIF");
00222     }
00223     NextLexeme();
00224     return result;
00225 }
00226
00227 std::shared_ptr<Executable> GrammaticalAnalyzer::For() {
00228     std::shared_ptr<class For> loop(new class For);
00229     if (GetCurrentLexeme().text != "DO") {
00230         ThrowSyntaxException("DO");
00231     }
00232     NextLexeme();
00233     loop->body = CodeBlock();
00234     if (GetCurrentLexeme().text != "LOOP") {
00235         ThrowSyntaxException("LOOP");
00236     }
00237     NextLexeme();
00238     return loop;
00239 }
00240
00241 std::shared_ptr<Executable> GrammaticalAnalyzer::While() {
00242     std::shared_ptr<class While> loop(new class While);
00243     if (GetCurrentLexeme().text != "BEGIN") {
00244         ThrowSyntaxException("BEGIN");
00245     }
00246     NextLexeme();
00247     loop->condition = CodeBlock();
00248     if (GetCurrentLexeme().text != "WHILE") {
00249         ThrowSyntaxException("WHILE");
00250     }
00251     NextLexeme();
00252     loop->body = CodeBlock();
00253     if (GetCurrentLexeme().text != "REPEAT") {
00254         ThrowSyntaxException("REPEAT");
00255     }
00256     NextLexeme();
00257     return loop;
00258 }
00259
00260 std::shared_ptr<Executable> GrammaticalAnalyzer::Switch() {
00261     std::shared_ptr<class Switch> switch_executable(new class Switch);
00262     if (GetCurrentLexeme().text != "CASE") {
00263         ThrowSyntaxException("CASE");
00264     }
00265     NextLexeme();
00266     while (GetCurrentLexeme().text != "ENDCASE") {
00267         if (GetCurrentLexeme().type != Lexeme::LexemeType::kLiteral) {
00268             ThrowSyntaxException("literal");
00269         }
00270         if (!IsInteger(GetCurrentLexeme().text)) {
00271             ThrowNotIntegerException(GetCurrentLexeme());
00272         }
00273         int64_t literal = std::stoll(GetCurrentLexeme().text);
00274         NextLexeme();
00275         if (GetCurrentLexeme().text != "OF") {
00276             ThrowSyntaxException("OF");
00277         }
00278         NextLexeme();
00279         auto case_code = CodeBlock();
00280         if (GetCurrentLexeme().text != "ENDOF") {
00281             ThrowSyntaxException("ENDOF");
00282         }
00283         NextLexeme();
00284         switch_executable->cases[literal] = case_code;
00285     }
00286     if (GetCurrentLexeme().text != "ENDCASE") {
00287         ThrowSyntaxException("ENDCASE");
```

```
00288     }
00289     NextLexeme();
00290     return switch_executable;
00291 }
00292
00293 std::shared_ptr<Executable> GrammaticalAnalyzer::VariableDefinition() {
00294     std::shared_ptr<VariableCreation> result(new VariableCreation);
00295     if (GetCurrentLexeme().text != "VARIABLE") {
00296         ThrowSyntaxException("VARIABLE");
00297     }
00298     NextLexeme();
00299     if (GetCurrentLexeme().type != Lexeme::LexemeType::kIdentifier) {
00300         ThrowSyntaxException("identifier");
00301     }
00302     result->name = GetCurrentLexeme().text;
00303     if (defined_identifiers.find(GetCurrentLexeme().text) != defined_identifiers.end()) {
00304         ThrowRedefinitionException(GetCurrentLexeme());
00305     }
00306     defined_identifiers.insert(GetCurrentLexeme().text);
00307     NextLexeme();
00308     result->size = 1;
00309     result->type = "cells";
00310     return result;
00311 }
00312
00313 std::shared_ptr<Executable> GrammaticalAnalyzer::ArrayDefinition() {
00314     std::shared_ptr<VariableCreation> result(new VariableCreation);
00315     if (GetCurrentLexeme().text != "CREATE") {
00316         ThrowSyntaxException("CREATE");
00317     }
00318     NextLexeme();
00319     if (GetCurrentLexeme().type != Lexeme::LexemeType::kIdentifier) {
00320         ThrowSyntaxException("identifier");
00321     }
00322     if (defined_identifiers.contains(GetCurrentLexeme().text)) {
00323         ThrowRedefinitionException(GetCurrentLexeme());
00324     }
00325     result->name = GetCurrentLexeme().text;
00326     defined_identifiers.insert(GetCurrentLexeme().text);
00327     NextLexeme();
00328     if (GetCurrentLexeme().type != Lexeme::LexemeType::kLiteral) {
00329         ThrowSyntaxException("literal");
00330     }
00331     if (!IsInteger(GetCurrentLexeme().text)) {
00332         ThrowNotIntegerException(GetCurrentLexeme());
00333     }
00334     result->size = std::stoll(GetCurrentLexeme().text);
00335     NextLexeme();
00336     result->type = GetCurrentLexeme().text;
00337     SizeOperators();
00338     if (GetCurrentLexeme().text != "allot") {
00339         ThrowSyntaxException("allot");
00340     }
00341     NextLexeme();
00342     return result;
00343 }
00344
00345 void GrammaticalAnalyzer::SizeOperators() {
00346     if (GetCurrentLexeme().text != "cells" && // int
00347         GetCurrentLexeme().text != "floats" &&
00348         GetCurrentLexeme().text != "chars") {
00349         ThrowSyntaxException("size operator");
00350     }
00351     NextLexeme();
00352 }
```

7.13 GrammaticalAnalyzer.h File Reference

Defines the GrammaticalAnalyzer class for performing syntactic analysis.

```
#include "Lexeme.h"
#include "Environment.h"
#include <vector>
#include <set>
#include <string>
#include <memory>
```

Classes

- class GrammaticalAnalyzer

  Performs syntactic analysis and generates the resulting environment.

7.13.1 Detailed Description

Defines the GrammaticalAnalyzer class for performing syntactic analysis.

Definition in file GrammaticalAnalyzer.h.

7.14 GrammaticalAnalyzer.h

Go to the documentation of this file.

```
00001
00006 #ifndef GRAMMATICALANALYZER_H
00007 #define GRAMMATICALANALYZER_H
00008
00009 #include "Lexeme.h"
00010 #include "Environment.h"
00011 #include <vector>
00012 #include <set>
00013 #include <string>
00014 #include <memory>
00015
00020 class GrammaticalAnalyzer {
00021 public:
00027     GrammaticalAnalyzer(const std::vector<Lexeme>& lexemes, const std::vector<std::string>&
      code_block_enders);
00028
00032     void Analyze();
00033
00037     Environment resulting_environment;
00038
00039 private:
00044     Lexeme GetCurrentLexeme();
00045
00049     void NextLexeme();
00050
00055     void ThrowSyntaxException(const std::string& message);
00056
00063     void ThrowGenericException(const Lexeme &l, const std::string& prefix_text, const std::string&
      suffix_text);
00064
00069     void ThrowUndefinedException(const Lexeme& l);
00070
00075     void ThrowNotIntegerException(const Lexeme& l);
00076
00081     void ThrowNotInLoopException(const Lexeme& l);
00082
00087     void ThrowNotInFunctionException(const Lexeme& l);
00088
00093     void ThrowRedefinitionException(const Lexeme& l);
00094
00099     bool IsFished();
00100
00104     void Program();
00105
00110     std::shared_ptr<Executable> FunctionDefinition();
00111
00116     std::shared_ptr<Executable> CodeBlock();
00117
00122     std::shared_ptr<Executable> ControlFlowConstruct();
00123
00128     std::shared_ptr<Executable> While();
00129
00134     std::shared_ptr<Executable> For();
00135
00140     std::shared_ptr<Executable> If();
00141
00146     std::shared_ptr<Executable> Switch();
00147
00152     std::shared_ptr<Executable> Statements();
00153
```

```
00158     std::shared_ptr<Executable> Statement();
00159
00164     std::shared_ptr<Executable> VariableDefinition();
00165
00170     std::shared_ptr<Executable> ArrayDefinition();
00171
00175     void SizeOperators();
00176
00177     std::vector<Lexeme> lexemes_;
00178     int current_lexeme_index_ = 0;
00179     std::set<std::string> code_block_enders_;
00180     std::set<std::string> defined_identifiers;
00181     int loop_counter = 0;
00182     int function_counter = 0;
00183 };
00184
00185 #endif // GRAMMATICALANALYZER_H
```

7.15 If.cpp File Reference

```
#include "Executable.h"
```

7.16 If.cpp

Go to the documentation of this file.

```
00001 #include "Executable.h"
00002
00003 Executable::ReturnStatus If::Execute(Environment& environment) {
00004     auto bool_flag = environment.PopStack();
00005     if (bool_flag.Convert<bool>()) {
00006         return if_part->Execute(environment);
00007     }
00008     if (else_part) {
00009         return else_part->Execute(environment);
00010     }
00011     return ReturnStatus::kSuccess;
00012 }
00013
```

7.17 Lexeme.h File Reference

```
#include <string>
```

Classes

- class Lexeme

7.18 Lexeme.h

Go to the documentation of this file.

```
00001 #ifndef LEXEME_H
00002 #define LEXEME_H
00003 #include <string>
00004
00005 class Lexeme {
00006 public:
00007     enum class LexemeType { // https://en.wikipedia.org/wiki/Lexical_analysis
00008         kWhitespace,
00009         kLiteral,
00010         kIdentifier,
00011         kOperator,
00012         kKeyword,
00013         kError,
00014         kControlFlowConstruct,
00015         kFunctionDefinitionStart,
00016         kFunctionDefinitionEnd
00017     };
00018     int row, column;
00019     LexemeType type;
00020     std::string text;
00021 private:
00022
00023 };
00024
00025 #endif //LEXEME_H
```

7.19 Literals.cpp File Reference

```
#include <string>
#include <regex>
#include "Literals.h"
```

Functions

- bool IsInteger (const std::string &str)
- bool IsDouble (const std::string &str)
- bool IsString (const std::string &str)
- bool IsLiteral (const std::string &str)

7.19.1 Function Documentation

7.19.1.1 IsDouble()

```
bool IsDouble (
            const std::string & str)
```

Definition at line 9 of file Literals.cpp.

7.19.1.2 IsInteger()

```
bool IsInteger (
            const std::string & str)
```

Definition at line 5 of file Literals.cpp.

7.19.1.3 IsLiteral()

```
bool IsLiteral (
            const std::string & str)
```

Definition at line 17 of file Literals.cpp.

7.19.1.4 IsString()

```
bool IsString (
            const std::string & str)
```

Definition at line 13 of file Literals.cpp.

7.20 Literals.cpp

Go to the documentation of this file.

```
00001 #include <string>
00002 #include <regex>
00003 #include "Literals.h"
00004
00005 bool IsInteger(const std::string& str) {
00006     return std::regex_match(str, std::regex("-?[0-9]+"));
00007 }
00008
00009 bool IsDouble(const std::string& str) {
00010     return std::regex_match(str, std::regex("-?[0-9]+([\\.][0-9]+)?"));
00011 }
00012
00013 bool IsString(const std::string& str) {
00014     return str.size() >= 3 && str[0] == 's' && str[1] == '"' && str.back() == '"';
00015 }
00016
00017 bool IsLiteral(const std::string& str) {
00018     return IsInteger(str) || IsDouble(str) || IsString(str);
00019 }
00020
```

7.21 Literals.h File Reference

Functions

- bool IsInteger (const std::string &str)
- bool IsDouble (const std::string &str)
- bool IsString (const std::string &str)
- bool IsLiteral (const std::string &str)

7.21.1 Function Documentation

7.21.1.1 IsDouble()

```
bool IsDouble (
            const std::string & str)
```

Definition at line 9 of file Literals.cpp.

7.21.1.2 IsInteger()

```
bool IsInteger (
            const std::string & str)
```

Definition at line 5 of file Literals.cpp.

7.21.1.3 IsLiteral()

```
bool IsLiteral (
            const std::string & str)
```

Definition at line 17 of file Literals.cpp.

7.21.1.4 IsString()

```
bool IsString (
            const std::string & str)
```

Definition at line 13 of file Literals.cpp.

7.22 Literals.h

Go to the documentation of this file.

```
00001 #ifndef LITERALS_H
00002 #define LITERALS_H
00003 bool IsInteger(const std::string& str);
00004
00005 bool IsDouble(const std::string& str);
00006
00007 bool IsString(const std::string& str);
00008
00009 bool IsLiteral(const std::string& str);
00010
00011 #endif //LITERALS_H
```

7.23 main.cpp File Reference

```
#include <iostream>
#include <stdexcept>
#include <string>
#include <fstream>
#include <csignal>
#include "Executable.h"
#include "GrammaticalAnalyzer.h"
#include "Preprocessor.h"
#include "Parser.h"
#include "Lexeme.h"
#include "StackElement.h"
```

Functions

- int main ()

7.23.1 Function Documentation

7.23.1.1 main()

```
int main ()
```

Definition at line 12 of file main.cpp.

7.24 main.cpp

Go to the documentation of this file.

```
00001 #include <iostream>
00002 #include <stdexcept>
00003 #include <string>
00004 #include <fstream>
00005 #include <csignal>
00006 #include "Executable.h"
00007 #include "GrammaticalAnalyzer.h"
00008 #include "Preprocessor.h"
00009 #include "Parser.h"
00010 #include "Lexeme.h"
00011 #include "StackElement.h"
00012 int main() {
00013     std::vector<std::string> keywords = {
00014         "BEGIN",
00015         "WHILE",
00016         "REPEAT",
00017         "DO",
00018         "LOOP",
00019         "IF",
00020         "ENDIF",
00021         "ELSE",
00022         "CASE",
00023         "OF",
00024         "ENDOF",
00025         "ENDCASE"
00026     };
00027
00028     std::vector<std::string> operators = {
00029         "dup",
00030         "drop",
00031         "swap",
00032         "over",
00033         "swap",
00034         "rot",
00035         "pick",
00036         "nip",
00037         "tuck",
00038         "roll",
00039         "+",
00040         "s+",
00041         "*",
00042         "/",
00043         "-",
00044         "%",
00045         "negate",
00046         "invert",
00047         "lshift",
00048         "rshift",
00049         "<",
00050         ">",
00051         "<=",
00052         ">=",
00053         "=",
00054         "s=",
00055         "and",
00056         "or",
00057         "xor",
00058         "not",
```

```
00059         "!",
00060         "f!",
00061         "c!",
00062         "@",
00063         "c@",
00064         "f@",
00065         "sinput",
00066         "finput",
00067         "input",
00068         "type",
00069         ".",
00070         ".s",
00071         "emit",
00072         "leave",
00073         "continue",
00074         "VARIABLE",
00075         "CREATE",
00076         "allot",
00077         "chars",
00078         "floats",
00079         "cells",
00080         "tofloat",
00081         "tocell",
00082         "return"
00083     };
00084     /*
00085     if (argc != 2) {
00086         throw std::logic_error("number of command line arguments arguments doesn't match");
00087     }
00088     */
00089     const std::string code_file("C:\\Users\\vvzag\\CLionProjects\\forth_interpretator\\test.txt");
00090     Preprocessor preprocessor(code_file);
00091     preprocessor.RemoveComments();
00092     std::string processed_string = preprocessor.GetCurrentText();
00093
00094
00095     Parser parser(processed_string, keywords, operators);
00096     auto lexemes = parser.GetResult();
00097     GrammaticalAnalyzer grammatical_analyzer(lexemes, {";", "REPEAT", "LOOP", "ELSE", "ENDOF", ":",
      "ENDIF", "WHILE"});
00098     grammatical_analyzer.Analyze();
00099     try {
00100
      grammatical_analyzer.resulting_environment.code->Execute(grammatical_analyzer.resulting_environment);
00101     } catch (std::exception& e) {
00102         std::cout << e.what() << '\n';
00103     }
00104 }
```

7.25 Operator.cpp File Reference

```
#include "Executable.h"
#include "Literals.h"
#include <iostream>
#include "StackElement.h"
```

Functions

- Executable::ReturnStatus AdditionOperator (Environment &environment)
- Executable::ReturnStatus SubtractionOperator (Environment &environment)
- Executable::ReturnStatus MultiplicationOperator (Environment &environment)
- Executable::ReturnStatus DivisionOperator (Environment &environment)
- Executable::ReturnStatus ModulusOperator (Environment &environment)
- Executable::ReturnStatus ConcatenationOperator (Environment &environment)
- Executable::ReturnStatus NegationOperator (Environment &environment)
- Executable::ReturnStatus InversionOperator (Environment &environment)
- Executable::ReturnStatus LshiftOperator (Environment &environment)
- Executable::ReturnStatus RshiftOperator (Environment &environment)

- Executable::ReturnStatus AndOperator (Environment &environment)
- Executable::ReturnStatus XorOperator (Environment &environment)
- Executable::ReturnStatus OrOperator (Environment &environment)
- Executable::ReturnStatus NotOperator (Environment &environment)
- Executable::ReturnStatus DupOperator (Environment &environment)
- Executable::ReturnStatus DropOperator (Environment &environment)
- Executable::ReturnStatus SwapOperator (Environment &environment)
- Executable::ReturnStatus OverOperator (Environment &environment)
- Executable::ReturnStatus RotOperator (Environment &environment)
- Executable::ReturnStatus PickOperator (Environment &environment)
- Executable::ReturnStatus NipOperator (Environment &environment)
- Executable::ReturnStatus TuckOperator (Environment &environment)
- Executable::ReturnStatus EqualsOperator (Environment &environment)
- Executable::ReturnStatus EqualsStringOperator (Environment &environment)
- Executable::ReturnStatus LessOperator (Environment &environment)
- Executable::ReturnStatus LessEqOperator (Environment &environment)
- Executable::ReturnStatus GreaterOperator (Environment &environment)
- Executable::ReturnStatus GreaterEqOperator (Environment &environment)
- template<typename T >
  Executable::ReturnStatus AssignmentOperator (Environment &environment)
- template<typename T >
  Executable::ReturnStatus DereferenceOperator (Environment &environment)
- template<typename T >
  Executable::ReturnStatus InputOperator (Environment &environment)
- template<> Executable::ReturnStatus InputOperator< std::string > (Environment &environment)
- Executable::ReturnStatus StringOutputOperator (Environment &environment)
- Executable::ReturnStatus CharOutputOperator (Environment &environment)
- Executable::ReturnStatus StackBackOutputOperator (Environment &environment)
- Executable::ReturnStatus AllStackOutputOperator (Environment &environment)
- Executable::ReturnStatus BreakOperator (Environment &environment)
- Executable::ReturnStatus ContinueOperator (Environment &environment)
- Executable::ReturnStatus ReturnOperator (Environment &environment)
- Executable::ReturnStatus ToCellOperator (Environment &environment)
- Executable::ReturnStatus ToFloatOperator (Environment &environment)

7.25.1 Function Documentation

7.25.1.1 AdditionOperator()

```
Executable::ReturnStatus AdditionOperator (
            Environment & environment)
```

Definition at line 51 of file Operator.cpp.

7.25.1.2 AllStackOutputOperator()

```
Executable::ReturnStatus AllStackOutputOperator (
            Environment & environment)
```

Definition at line 316 of file Operator.cpp.

7.25.1.3 AndOperator()

```
Executable::ReturnStatus AndOperator (
            Environment & environment)
```

Definition at line 128 of file Operator.cpp.

7.25.1.4 AssignmentOperator()

```
template<typename T >
Executable::ReturnStatus AssignmentOperator (
            Environment & environment)
```

Definition at line 265 of file Operator.cpp.

7.25.1.5 BreakOperator()

```
Executable::ReturnStatus BreakOperator (
            Environment & environment)
```

Definition at line 324 of file Operator.cpp.

7.25.1.6 CharOutputOperator()

```
Executable::ReturnStatus CharOutputOperator (
            Environment & environment)
```

Definition at line 304 of file Operator.cpp.

7.25.1.7 ConcatenationOperator()

```
Executable::ReturnStatus ConcatenationOperator (
            Environment & environment)
```

Definition at line 86 of file Operator.cpp.

7.25.1.8 ContinueOperator()

```
Executable::ReturnStatus ContinueOperator (
            Environment & environment)
```

Definition at line 328 of file Operator.cpp.

7.25.1.9 DereferenceOperator()

```
template<typename T >
Executable::ReturnStatus DereferenceOperator (
            Environment & environment)
```

Definition at line 273 of file Operator.cpp.

7.25.1.10 DivisionOperator()

```
Executable::ReturnStatus DivisionOperator (
            Environment & environment)
```

Definition at line 72 of file Operator.cpp.

7.25.1.11 DropOperator()

```
Executable::ReturnStatus DropOperator (
            Environment & environment)
```

Definition at line 162 of file Operator.cpp.

7.25.1.12 DupOperator()

```
Executable::ReturnStatus DupOperator (
            Environment & environment)
```

Definition at line 155 of file Operator.cpp.

7.25.1.13 EqualsOperator()

```
Executable::ReturnStatus EqualsOperator (
            Environment & environment)
```

Definition at line 220 of file Operator.cpp.

7.25.1.14 EqualsStringOperator()

```
Executable::ReturnStatus EqualsStringOperator (
            Environment & environment)
```

Definition at line 227 of file Operator.cpp.

7.25.1.15 GreaterEqOperator()

```
Executable::ReturnStatus GreaterEqOperator (
            Environment & environment)
```

Definition at line 257 of file Operator.cpp.

7.25.1.16 GreaterOperator()

```
Executable::ReturnStatus GreaterOperator (
            Environment & environment)
```

Definition at line 250 of file Operator.cpp.

7.25.1.17 InputOperator()

```
template<typename T >
Executable::ReturnStatus InputOperator (
            Environment & environment)
```

Definition at line 280 of file Operator.cpp.

7.25.1.18 InputOperator< std::string >()

```
template<>
Executable::ReturnStatus InputOperator< std::string > (
            Environment & environment)
```

Definition at line 288 of file Operator.cpp.

7.25.1.19 InversionOperator()

```
Executable::ReturnStatus InversionOperator (
            Environment & environment)
```

Definition at line 106 of file Operator.cpp.

7.25.1.20 LessEqOperator()

```
Executable::ReturnStatus LessEqOperator (
            Environment & environment)
```

Definition at line 243 of file Operator.cpp.

7.25.1.21 LessOperator()

```
Executable::ReturnStatus LessOperator (
            Environment & environment)
```

Definition at line 236 of file Operator.cpp.

7.25.1.22 LshiftOperator()

```
Executable::ReturnStatus LshiftOperator (
            Environment & environment)
```

Definition at line 112 of file Operator.cpp.

7.25.1.23 ModulusOperator()

```
Executable::ReturnStatus ModulusOperator (
            Environment & environment)
```

Definition at line 79 of file Operator.cpp.

7.25.1.24 MultiplicationOperator()

```
Executable::ReturnStatus MultiplicationOperator (
            Environment & environment)
```

Definition at line 65 of file Operator.cpp.

7.25.1.25 NegationOperator()

```
Executable::ReturnStatus NegationOperator (
            Environment & environment)
```

Definition at line 100 of file Operator.cpp.

7.25.1.26 NipOperator()

```
Executable::ReturnStatus NipOperator (
            Environment & environment)
```

Definition at line 204 of file Operator.cpp.

7.25.1.27 NotOperator()

```
Executable::ReturnStatus NotOperator (
            Environment & environment)
```

Definition at line 149 of file Operator.cpp.

7.25.1.28 OrOperator()

```
Executable::ReturnStatus OrOperator (
            Environment & environment)
```

Definition at line 142 of file Operator.cpp.

7.25.1.29 OverOperator()

```
Executable::ReturnStatus OverOperator (
            Environment & environment)
```

Definition at line 175 of file Operator.cpp.

7.25.1.30 PickOperator()

```
Executable::ReturnStatus PickOperator (
            Environment & environment)
```

Definition at line 194 of file Operator.cpp.

7.25.1.31 ReturnOperator()

```
Executable::ReturnStatus ReturnOperator (
            Environment & environment)
```

Definition at line 332 of file Operator.cpp.

7.25.1.32 RotOperator()

```
Executable::ReturnStatus RotOperator (
            Environment & environment)
```

Definition at line 184 of file Operator.cpp.

7.25.1.33 RshiftOperator()

```
Executable::ReturnStatus RshiftOperator (
            Environment & environment)
```

Definition at line 120 of file Operator.cpp.

7.25.1.34 StackBackOutputOperator()

```
Executable::ReturnStatus StackBackOutputOperator (
            Environment & environment)
```

Definition at line 310 of file Operator.cpp.

7.25.1.35 StringOutputOperator()

```
Executable::ReturnStatus StringOutputOperator (
            Environment & environment)
```

Definition at line 297 of file Operator.cpp.

7.25.1.36 SubtractionOperator()

```
Executable::ReturnStatus SubtractionOperator (
            Environment & environment)
```

Definition at line 58 of file Operator.cpp.

7.25.1.37 SwapOperator()

```
Executable::ReturnStatus SwapOperator (
            Environment & environment)
```

Definition at line 167 of file Operator.cpp.

7.25.1.38 ToCellOperator()

```
Executable::ReturnStatus ToCellOperator (
            Environment & environment)
```

Definition at line 336 of file Operator.cpp.

7.25.1.39 ToFloatOperator()

```
Executable::ReturnStatus ToFloatOperator (
            Environment & environment)
```

Definition at line 341 of file Operator.cpp.

7.25.1.40 TuckOperator()

```
Executable::ReturnStatus TuckOperator (
            Environment & environment)
```

Definition at line 211 of file Operator.cpp.

7.25.1.41 XorOperator()

```
Executable::ReturnStatus XorOperator (
            Environment & environment)
```

Definition at line 135 of file Operator.cpp.

7.26 Operator.cpp

Go to the documentation of this file.

```
00001 #include "Executable.h"
00002 #include "Literals.h"
00003 #include <iostream>
00004 #include "StackElement.h"
00005 Operator::Operator(std::string text) : text(text) {
00006 }
00007
00008 Executable::ReturnStatus Operator::Execute(Environment& environment) {
00009     if (operators_pointers.contains(text)) {
00010         return operators_pointers[text](environment);
00011     }
00012     if (environment.functions.contains(text)) {
00013         return FunctionCall(environment);
00014     }
00015     if (environment.variables.contains(text)) {
00016         return VariableUse(environment);
00017     }
00018     if (IsLiteral(text)) {
00019         return Literal(environment);
00020     }
00021     throw std::runtime_error("unknown operator passed");
00022 }
00023
00024 Executable::ReturnStatus Operator::FunctionCall(Environment& environment) {
00025     auto status = environment.functions[text]->Execute(environment);
00026     if (status == ReturnStatus::kLeaveFunction) {
00027         status = ReturnStatus::kSuccess;
00028     }
00029     return status;
```

```
00030 }
00031
00032 Executable::ReturnStatus Operator::VariableUse(Environment &environment) {
00033     environment.PushOnStack(StackElement(reinterpret_cast<int64_t>(environment.variables[text])));
00034     return ReturnStatus::kSuccess;
00035 }
00036
00037 Executable::ReturnStatus Operator::Literal(Environment &environment) {
00038     if (IsInteger(text)) {
00039         environment.PushOnStack(StackElement(std::stoll(text)));
00040         return ReturnStatus::kSuccess;
00041     }
00042     if (IsDouble(text)) {
00043         environment.PushOnStack(StackElement(std::stod(text)));
00044         return ReturnStatus::kSuccess;
00045     }
00046     environment.PushOnStack(StackElement(reinterpret_cast<int64_t>(text.c_str() + 2)));
00047     environment.PushOnStack(StackElement(static_cast<int64_t>(text.size() - 3)));
00048     return ReturnStatus::kSuccess;
00049 }
00050
00051 Executable::ReturnStatus AdditionOperator(Environment& environment) {
00052     StackElement a = environment.PopStack();
00053     StackElement b = environment.PopStack();
00054     environment.PushOnStack(a + b);
00055     return Executable::ReturnStatus::kSuccess;
00056 }
00057
00058 Executable::ReturnStatus SubtractionOperator(Environment& environment) {
00059     StackElement a = environment.PopStack();
00060     StackElement b = environment.PopStack();
00061     environment.PushOnStack(b - a);
00062     return Executable::ReturnStatus::kSuccess;
00063 }
00064
00065 Executable::ReturnStatus MultiplicationOperator(Environment& environment) {
00066     StackElement a = environment.PopStack();
00067     StackElement b = environment.PopStack();
00068     environment.PushOnStack(a * b);
00069     return Executable::ReturnStatus::kSuccess;
00070 }
00071
00072 Executable::ReturnStatus DivisionOperator(Environment& environment) {
00073     StackElement a = environment.PopStack();
00074     StackElement b = environment.PopStack();
00075     environment.PushOnStack(b / a);
00076     return Executable::ReturnStatus::kSuccess;
00077 }
00078
00079 Executable::ReturnStatus ModulusOperator(Environment& environment) {
00080     StackElement a = environment.PopStack();
00081     StackElement b = environment.PopStack();
00082     environment.PushOnStack(b % a);
00083     return Executable::ReturnStatus::kSuccess;
00084 }
00085
00086 Executable::ReturnStatus ConcatenationOperator(Environment& environment) {
00087     auto sz2 = environment.PopStack().Convert<int64_t>();
00088     auto address2 = environment.PopStack().Convert<char*>();
00089     auto sz1 = environment.PopStack().Convert<int64_t >();
00090     auto address1 = environment.PopStack().Convert<char*>();
00091     auto res_sz = sz1 + sz2;
00092     char* res = new char[res_sz];
00093     memcpy(res, address1, sz1);
00094     memcpy(res + sz1, address2, sz2);
00095     environment.PushOnStack((int64_t)res);
00096     environment.PushOnStack(res_sz);
00097     return Executable::ReturnStatus::kSuccess;
00098 }
00099
00100 Executable::ReturnStatus NegationOperator(Environment& environment) {
00101     StackElement a = environment.PopStack();
00102     environment.PushOnStack(-a);
00103     return Executable::ReturnStatus::kSuccess;
00104 }
00105
00106 Executable::ReturnStatus InversionOperator(Environment& environment) {
00107     StackElement a = environment.PopStack();
00108     environment.PushOnStack(~a);
00109     return Executable::ReturnStatus::kSuccess;
00110 }
00111
00112 Executable::ReturnStatus LshiftOperator(Environment& environment) {
00113     auto k = environment.PopStack();
00114     auto a = environment.PopStack();
00115     a.value = (a.Convert<int64_t>() « k.Convert<int64_t>());
00116     environment.PushOnStack(a);
```

```
00117     return Executable::ReturnStatus::kSuccess;
00118 }
00119
00120 Executable::ReturnStatus RshiftOperator(Environment& environment) {
00121     auto k = environment.PopStack();
00122     auto a = environment.PopStack();
00123     a.value = (a.Convert<int64_t>() >> k.Convert<int64_t>());
00124     environment.PushOnStack(a);
00125     return Executable::ReturnStatus::kSuccess;
00126 }
00127
00128 Executable::ReturnStatus AndOperator(Environment& environment) {
00129     auto a = environment.PopStack();
00130     auto b = environment.PopStack();
00131     environment.PushOnStack(a & b);
00132     return Executable::ReturnStatus::kSuccess;
00133 }
00134
00135 Executable::ReturnStatus XorOperator(Environment& environment) {
00136     auto a = environment.PopStack();
00137     auto b = environment.PopStack();
00138     environment.PushOnStack(a ^ b);
00139     return Executable::ReturnStatus::kSuccess;
00140 }
00141
00142 Executable::ReturnStatus OrOperator(Environment& environment) {
00143     auto a = environment.PopStack();
00144     auto b = environment.PopStack();
00145     environment.PushOnStack(a | b);
00146     return Executable::ReturnStatus::kSuccess;
00147 }
00148
00149 Executable::ReturnStatus NotOperator(Environment& environment) {
00150     auto a = environment.PopStack();
00151     environment.PushOnStack(!a);
00152     return Executable::ReturnStatus::kSuccess;
00153 }
00154
00155 Executable::ReturnStatus DupOperator(Environment& environment) {
00156     auto a = environment.PopStack();
00157     environment.PushOnStack(a);
00158     environment.PushOnStack(a);
00159     return Executable::ReturnStatus::kSuccess;
00160 }
00161
00162 Executable::ReturnStatus DropOperator(Environment& environment) {
00163     environment.PopStack();
00164     return Executable::ReturnStatus::kSuccess;
00165 }
00166
00167 Executable::ReturnStatus SwapOperator(Environment& environment) {
00168     auto w2 = environment.PopStack();
00169     auto w1 = environment.PopStack();
00170     environment.PushOnStack(w2);
00171     environment.PushOnStack(w1);
00172     return Executable::ReturnStatus::kSuccess;
00173 }
00174
00175 Executable::ReturnStatus OverOperator(Environment& environment) {
00176     auto w2 = environment.PopStack();
00177     auto w1 = environment.PopStack();
00178     environment.PushOnStack(w1);
00179     environment.PushOnStack(w2);
00180     environment.PushOnStack(w1);
00181     return Executable::ReturnStatus::kSuccess;
00182 }
00183
00184 Executable::ReturnStatus RotOperator(Environment& environment) {
00185     auto w3 = environment.PopStack();
00186     auto w2 = environment.PopStack();
00187     auto w1 = environment.PopStack();
00188     environment.PushOnStack(w2);
00189     environment.PushOnStack(w3);
00190     environment.PushOnStack(w1);
00191     return Executable::ReturnStatus::kSuccess;
00192 }
00193
00194 Executable::ReturnStatus PickOperator(Environment& environment) {
00195     auto a = environment.PopStack().Convert<int64_t>();
00196     if (a >= 0 && a < (int64_t)environment.stack.size()) {
00197         environment.PushOnStack(
00198                 environment.stack[environment.stack.size() - a - 1]);
00199         return Executable::ReturnStatus::kSuccess;
00200     }
00201     throw std::runtime_error("Incorrect argument");
00202 }
00203
```

```
00204 Executable::ReturnStatus NipOperator(Environment& environment) {
00205     auto w2 = environment.PopStack();
00206     auto w1 = environment.PopStack();
00207     environment.PushOnStack(w2);
00208     return Executable::ReturnStatus::kSuccess;
00209 }
00210
00211 Executable::ReturnStatus TuckOperator(Environment& environment) {
00212     auto w2 = environment.PopStack();
00213     auto w1 = environment.PopStack();
00214     environment.PushOnStack(w2);
00215     environment.PushOnStack(w1);
00216     environment.PushOnStack(w2);
00217     return Executable::ReturnStatus::kSuccess;
00218 }
00219
00220 Executable::ReturnStatus EqualsOperator(Environment& environment) {
00221     auto a = environment.PopStack();
00222     auto b = environment.PopStack();
00223     environment.PushOnStack(a == b);
00224     return Executable::ReturnStatus::kSuccess;
00225 }
00226
00227 Executable::ReturnStatus EqualsStringOperator(Environment& environment) {
00228     auto len1 = environment.PopStack().Convert<int64_t>();
00229     auto cdata1 = environment.PopStack().Convert<char*>();
00230     auto len2 = environment.PopStack().Convert<int64_t>();
00231     auto cdata2 = environment.PopStack().Convert<char*>();
00232     environment.PushOnStack(std::string(cdata1, len1) == std::string(cdata2, len2));
00233     return Executable::ReturnStatus::kSuccess;
00234 }
00235
00236 Executable::ReturnStatus LessOperator(Environment& environment) {
00237     auto b = environment.PopStack();
00238     auto a = environment.PopStack();
00239     environment.PushOnStack(a < b);
00240     return Executable::ReturnStatus::kSuccess;
00241 }
00242
00243 Executable::ReturnStatus LessEqOperator(Environment& environment) {
00244     auto b = environment.PopStack();
00245     auto a = environment.PopStack();
00246     environment.PushOnStack(a <= b);
00247     return Executable::ReturnStatus::kSuccess;
00248 }
00249
00250 Executable::ReturnStatus GreaterOperator(Environment& environment) {
00251     auto b = environment.PopStack();
00252     auto a = environment.PopStack();
00253     environment.PushOnStack(a > b);
00254     return Executable::ReturnStatus::kSuccess;
00255 }
00256
00257 Executable::ReturnStatus GreaterEqOperator(Environment& environment)  {
00258     auto b = environment.PopStack();
00259     auto a = environment.PopStack();
00260     environment.PushOnStack(a >= b);
00261     return Executable::ReturnStatus::kSuccess;
00262 }
00263
00264 template<typename T>
00265 Executable::ReturnStatus AssignmentOperator(Environment& environment) {
00266     auto ptr = environment.PopStack().Convert<T*>();
00267     auto val = environment.PopStack().Convert<T>();
00268     *ptr = val;
00269     return Executable::ReturnStatus::kSuccess;
00270 }
00271
00272 template<typename T>
00273 Executable::ReturnStatus DereferenceOperator(Environment& environment) {
00274     auto ptr = environment.PopStack().Convert<T*>();
00275     environment.PushOnStack(*ptr);
00276     return Executable::ReturnStatus::kSuccess;
00277 }
00278
00279 template<typename T>
00280 Executable::ReturnStatus InputOperator(Environment& environment) {
00281     T x;
00282     std::cin » x;
00283     environment.PushOnStack(x);
00284     return Executable::ReturnStatus::kSuccess;
00285 }
00286
00287 template<>
00288 Executable::ReturnStatus InputOperator<std::string>(Environment& environment) {
00289     std::string s;
00290     std::cin » s;
```

```
00291     char* cs = new char[s.size()];
00292     environment.PushOnStack((int64_t)cs);
00293     environment.PushOnStack((int64_t)s.size());
00294     return Executable::ReturnStatus::kSuccess;
00295 }
00296
00297 Executable::ReturnStatus StringOutputOperator(Environment& environment) {
00298     auto sz = environment.PopStack().Convert<size_t>();
00299     auto address = environment.PopStack().Convert<char*>();
00300     std::cout « std::string(address, address + sz);
00301     return Executable::ReturnStatus::kSuccess;
00302 }
00303
00304 Executable::ReturnStatus CharOutputOperator(Environment& environment) {
00305     char e = environment.PopStack().Convert<char>();
00306     std::cout « e;
00307     return Executable::ReturnStatus::kSuccess;
00308 }
00309
00310 Executable::ReturnStatus StackBackOutputOperator(Environment& environment) {
00311     StackElement a = environment.PopStack();
00312     std::cout « a « ' ';
00313     return Executable::ReturnStatus::kSuccess;
00314 }
00315
00316 Executable::ReturnStatus AllStackOutputOperator(Environment& environment) {
00317     for (auto el : environment.stack) {
00318         std::cout « el « ' ';
00319     }
00320     std::cout « "<" « environment.stack.size() « ">\n";
00321     return Executable::ReturnStatus::kSuccess;
00322 }
00323
00324 Executable::ReturnStatus BreakOperator(Environment& environment) {
00325     return Executable::ReturnStatus::kLeaveLoop;
00326 }
00327
00328 Executable::ReturnStatus ContinueOperator(Environment& environment) {
00329     return Executable::ReturnStatus::kContinueLoop;
00330 }
00331
00332 Executable::ReturnStatus ReturnOperator(Environment& environment) {
00333     return Executable::ReturnStatus::kLeaveFunction;
00334 }
00335
00336 Executable::ReturnStatus ToCellOperator(Environment& environment) {
00337     environment.PushOnStack(environment.PopStack().Convert<int64_t>());
00338     return Executable::ReturnStatus::kSuccess;
00339 }
00340
00341 Executable::ReturnStatus ToFloatOperator(Environment& environment) {
00342     environment.PushOnStack(environment.PopStack().Convert<double>());
00343     return Executable::ReturnStatus::kSuccess;
00344 }
00345
00346 std::map<
00347     std::string,
00348     std::function<Executable::ReturnStatus (Environment&)>
00349 > Operator::operators_pointers = {
00350     {"+", AdditionOperator},
00351     {"-", SubtractionOperator},
00352     {"*", MultiplicationOperator},
00353     {"/", DivisionOperator},
00354     {"%", ModulusOperator},
00355     {"s+", ConcatenationOperator},
00356     {"negate", NegationOperator},
00357     {"inverse", InversionOperator},
00358     {"lshift", LshiftOperator},
00359     {"rshift", RshiftOperator},
00360     {"and", AndOperator},
00361     {"or", OrOperator},
00362     {"xor", XorOperator},
00363     {"not", NotOperator},
00364     {"dup", DupOperator},
00365     {"drop", DropOperator},
00366     {"swap", SwapOperator},
00367     {"over", OverOperator},
00368     {"rot", RotOperator},
00369     {"pick", PickOperator},
00370     {"nip", NipOperator},
00371     {"tuck", TuckOperator},
00372     {"=", EqualsOperator},
00373     {"s=", EqualsStringOperator},
00374     {"<", LessOperator},
00375     {"<=", LessEqOperator},
00376     {">", GreaterOperator},
00377     {">=", GreaterEqOperator},
```

```
00378     {"!", AssignmentOperator<int64_t>},
00379     {"f!", AssignmentOperator<double>},
00380     {"c!", AssignmentOperator<char>},
00381     {"@", DereferenceOperator<int64_t>},
00382     {"f@", DereferenceOperator<double>},
00383     {"c@", DereferenceOperator<char>},
00384     {"sinput", InputOperator<std::string>},
00385     {"finput", InputOperator<double>},
00386     {"input", InputOperator<int64_t>},
00387     {"type", StringOutputOperator},
00388     {"emit", CharOutputOperator},
00389     {".", StackBackOutputOperator},
00390     {".s", AllStackOutputOperator},
00391     {"leave", BreakOperator},
00392     {"continue", ContinueOperator},
00393     {"return", ReturnOperator},
00394     {"tocell", ToCellOperator},
00395     {"tofloat", ToFloatOperator},
00396 };
```

7.27 Parser.cpp File Reference

```
#include <regex>
#include "Lexeme.h"
#include "Literals.h"
#include "Parser.h"
```

Functions

- bool IsDelimeter (char c)

7.27.1 Function Documentation

7.27.1.1 IsDelimeter()

```
bool IsDelimeter (
            char c)
```

Definition at line 9 of file Parser.cpp.

7.28 Parser.cpp

Go to the documentation of this file.

```
00001 #include <regex>
00002 #include "Lexeme.h"
00003 #include "Literals.h"
00004 #include "Parser.h"
00005 std::vector<Lexeme> Parser::GetResult() {
00006     return result;
00007 }
00008
00009 bool IsDelimeter(char c) {
00010     return c == '\n' || c == ' ';
00011 }
00012
00013 Parser::Parser(const std::string& input, const std::vector<std::string>& keywords, const
      std::vector<std::string>& operators) {
00014     Trie keyword_trie;
00015     for (const auto& str : keywords) {
00016         keyword_trie.Add(str);
```

```
00017     }
00018     Trie operator_trie;
00019     for (const auto& str : operators) {
00020         operator_trie.Add(str);
00021     }
00022     std::string cur_str;
00023     int line = 1;
00024     int current_column = 0;
00025     for (int i = 0; i < input.size(); ++i) {
00026         char c = input[i];
00027         if (IsDelimeter(c) && !cur_str.empty() &&
00028             !(cur_str.size() >= 3 && cur_str[0] == 's' && cur_str[1] == '"' && cur_str.back() != '"'))
     {
00029             Lexeme current;
00030             current.row = line;
00031             current.column = current_column;
00032             current.type = Lexeme::LexemeType::kError;
00033             current.text = cur_str;
00034             if (keyword_trie.Contains(cur_str)) {
00035                 current.type = Lexeme::LexemeType::kKeyword;
00036             } else if (IsLiteral(cur_str)) {
00037                 current.type = Lexeme::LexemeType::kLiteral;
00038             } else if (operator_trie.Contains(cur_str)) {
00039                 current.type = Lexeme::LexemeType::kOperator;
00040             } else if (cur_str == ":") {
00041                 current.type = Lexeme::LexemeType::kFunctionDefinitionStart;
00042             } else if (cur_str == ";") {
00043                 current.type = Lexeme::LexemeType::kFunctionDefinitionEnd;
00044             } else {
00045                 current.type = Lexeme::LexemeType::kIdentifier;
00046             }
00047             result.push_back(current);
00048             cur_str.clear();
00049         } else if (!IsDelimeter(c) || (cur_str.size() >= 2 && cur_str[0] == 's' && cur_str[1] == '"'))
     {
00050             cur_str += c;
00051         }
00052         if (c == '\n') {
00053             line++;
00054             current_column = 0;
00055         } else {
00056             current_column++;
00057         }
00058     }
00059 }
00060
00061 void Parser::Trie::Add(std::string str) {
00062     Node* current_node = root.get();
00063     for (auto c : str) {
00064         if (current_node->go.find(c) == current_node->go.end()) {
00065             current_node->go[c] = std::make_unique<Node>();
00066         }
00067         current_node = current_node->go[c].get();
00068     }
00069     current_node->is_terminal = true;
00070 }
00071
00072 bool Parser::Trie::Contains(std::string str) {
00073     Node* current_node = root.get();
00074     for (auto c : str) {
00075         if (current_node->go.find(c) == current_node->go.end()) {
00076             return false;
00077         }
00078         current_node = current_node->go[c].get();
00079     }
00080     return current_node->is_terminal;
00081 }
00082
```

7.29 Parser.h File Reference

Defines the Parser class for lexical analysis of input strings.

```
#include <memory>
#include <string>
#include <vector>
#include <map>
#include "Lexeme.h"
```

Classes

- class Parser

  Performs lexical analysis of input strings, producing a vector of lexemes.
- struct Parser::Trie

  A simple trie structure for keyword and operator matching.
- struct Parser::Trie::Node

  Represents a single node in the Trie.

7.29.1 Detailed Description

Defines the Parser class for lexical analysis of input strings.

Definition in file Parser.h.

7.30 Parser.h

Go to the documentation of this file.

```
00001
00006 #ifndef PARSER_H
00007 #define PARSER_H
00008
00009 #include <memory>
00010 #include <string>
00011 #include <vector>
00012 #include <map>
00013 #include "Lexeme.h"
00014
00019 class Parser {
00020 public:
00027     explicit Parser(const std::string& input,
00028                     const std::vector<std::string>& keywords,
00029                     const std::vector<std::string>& operators);
00030
00035     std::vector<Lexeme> GetResult();
00036
00037 private:
00044     struct Trie {
00049         struct Node {
00050             bool is_terminal = false;
00051             std::map<char, std::unique_ptr<Node>> go;
00052         };
00053
00054         std::unique_ptr<Node> root = std::make_unique<Node>();
00055
00060         void Add(std::string str);
00061
00067         bool Contains(std::string str);
00068     };
00069
00070     std::vector<Lexeme> result;
00071 };
00072
00073 #endif // PARSER_H
```

7.31 Preprocessor.cpp File Reference

```
#include "Preprocessor.h"
#include <fstream>
#include <stdexcept>
#include <algorithm>
```

7.32 Preprocessor.cpp

Go to the documentation of this file.

```
00001 #include "Preprocessor.h"
00002 #include <fstream>
00003 #include <stdexcept>
00004 #include <algorithm>
00005
00006 Preprocessor::Preprocessor(std::string file_path) {
00007     std::ifstream code_file(file_path);
00008     if (!code_file.is_open()) {
00009         throw std::logic_error("failed to open file for preprocessing");
00010     }
00011     std::string line;
00012     while (std::getline(code_file, line)) {
00013         line += '\n';
00014         current_text += line;
00015     }
00016 }
00017
00018 std::string Preprocessor::GetCurrentText() {
00019     return current_text;
00020 }
00021
00022 void Preprocessor::ToOneLine() {
00023     std::replace_if(current_text.begin(), current_text.end(), [](auto c) {
00024         return static_cast<int>(c) < 32;
00025     }, ' ');
00026 }
00027
00028 void Preprocessor::RemoveComments() {
00029     // braces are for comments in forth
00030     // so if character is inside at least one pair of braces or goes after \ in line it is irrevelant
00031     int balance = 0;
00032     bool slash_comment = false;
00033     std::string result;
00034     for (auto c : current_text) {
00035         if (c == '\\') {
00036             slash_comment = true;
00037         }
00038         if (c == '\n') {
00039             slash_comment = false;
00040         }
00041         if (c == '(') {
00042             balance++;
00043         }
00044         if ((balance == 0 && !slash_comment) || c == '\n') {
00045             result += c;
00046         } else {
00047             result += ' ';
00048         }
00049         if (c == ')') {
00050             balance--;
00051         }
00052     }
00053
00054     current_text = result;
00055 }
00056
00057
00058
00059
```

7.33 Preprocessor.h File Reference

```
#include <string>
```

Classes

- class Preprocessor

7.34 Preprocessor.h

Go to the documentation of this file.

```
00001 #ifndef PREPROCESSOR_H
00002 #define PREPROCESSOR_H
00003 #include <string>
00004
00005
00006 class Preprocessor {
00007 public:
00008     explicit Preprocessor(std::string file_path);
00009     std::string GetCurrentText();
00010     void ToOneLine(); // transform file to one line for easier parsing
00011     void RemoveComments();
00012 private:
00013
00014 public:
00015
00016
00017 private:
00018     std::string current_text;
00019
00020 };
00021
00022
00023
00024 #endif //PREPROCESSOR_H
```

7.35 StackElement.cpp File Reference

```
#include "StackElement.h"
```

Functions

- std::ostream & operator<< (std::ostream &out, const StackElement &val)

  Outputs the value of a StackElement to an output stream.

7.35.1 Function Documentation

7.35.1.1 operator<<()

```
std::ostream & operator<< (
            std::ostream & out,
            const StackElement & val)
```

Outputs the value of a StackElement to an output stream.

Parameters

out	The output stream.
val	The StackElement to output.

Returns

The output stream.

Definition at line 104 of file StackElement.cpp.

7.36 StackElement.cpp

Go to the documentation of this file.

```
00001 #include "StackElement.h"
00002
00003 StackElement StackElement::operator+(const StackElement& other) {
00004     return std::visit([](auto a, auto b) {
00005         return StackElement(a + b);
00006     }, value, other.value);
00007 }
00008
00009 StackElement StackElement::operator-(const StackElement& other) {
00010     return std::visit([](auto a, auto b) {
00011         return StackElement(a - b);
00012     }, value, other.value);
00013 }
00014
00015 StackElement StackElement::operator-() {
00016     return std::visit([](auto a) {
00017         return StackElement(-a);
00018     }, value);
00019 }
00020
00021 StackElement StackElement::operator*(const StackElement& other) {
00022     return std::visit([](auto a, auto b) {
00023         return StackElement(a * b);
00024     }, value, other.value);
00025 }
00026
00027 StackElement StackElement::operator/(const StackElement& other) {
00028     return std::visit([](auto a, auto b) {
00029         return StackElement(a / b);
00030     }, value, other.value);
00031 }
00032
00033 StackElement StackElement::operator%(const StackElement& other) {
00034     return std::visit([](auto a, auto b) {
00035         return StackElement(static_cast<int64_t>(a) % static_cast<int64_t>(b));
00036     }, value, other.value);
00037 }
00038
00039 StackElement StackElement::operator~() {
00040     return std::visit([](auto a) {
00041         return StackElement(~static_cast<int64_t>(a));
00042     }, value);
00043 }
00044
00045 StackElement StackElement::operator!() {
00046     return std::visit([](auto a) {
00047         return StackElement(!a);
00048     }, value);
00049 }
00050
00051 StackElement StackElement::operator&(const StackElement& other) {
00052     return std::visit([](auto a, auto b) {
00053         return StackElement(static_cast<int64_t>(a) & static_cast<int64_t>(b));
00054     }, value, other.value);
00055 }
00056
00057 StackElement StackElement::operator|(const StackElement& other) {
00058     return std::visit([](auto a, auto b) {
00059         return StackElement(static_cast<int64_t>(a) | static_cast<int64_t>(b));
00060     }, value, other.value);
00061 }
00062
00063 StackElement StackElement::operator^(const StackElement& other) {
00064     return std::visit([](auto a, auto b) {
00065         return StackElement(static_cast<int64_t>(a) ^ static_cast<int64_t>(b));
00066     }, value, other.value);
00067 }
00068
00069 StackElement StackElement::operator<(const StackElement& other) {
00070     auto result = std::visit([](auto a, auto b) {
00071         return a < b;
00072     }, value, other.value);
00073     return StackElement(result);
00074 }
00075
00076 StackElement StackElement::operator<=(const StackElement& other) {
00077     auto result = std::visit([](auto a, auto b) {
00078         return a <= b;
00079     }, value, other.value);
00080     return StackElement(result);
00081 }
00082
```

```
00083 StackElement StackElement::operator>(const StackElement& other) {
00084     auto result = std::visit([](auto a, auto b) {
00085         return a > b;
00086     }, value, other.value);
00087     return StackElement(result);
00088 }
00089
00090 StackElement StackElement::operator>=(const StackElement& other) {
00091     auto result = std::visit([](auto a, auto b) {
00092         return a >= b;
00093     }, value, other.value);
00094     return StackElement(result);
00095 }
00096
00097 StackElement StackElement::operator==(const StackElement& other) {
00098     auto result = std::visit([](auto a, auto b) {
00099         return a == b;
00100     }, value, other.value);
00101     return StackElement(result);
00102 }
00103
00104 std::ostream& operator«(std::ostream& out, const StackElement& val) {
00105     std::visit([&out](auto a) {
00106         out « a;
00107     }, val.value);
00108     return out;
00109 }
```

7.37 StackElement.h File Reference

Defines the StackElement class for handling stack operations and arithmetic.

```
#include <cstdint>
#include <variant>
#include <istream>
#include <type_traits>
```

Classes

- class StackElement

  Represents an element on the stack that supports various operations.

Concepts

- concept Rational

  Concept to define types that can be represented as rational numbers.

Functions

- std::ostream & operator<< (std::ostream &out, const StackElement &val)

  Outputs the value of a StackElement to an output stream.

7.37.1 Detailed Description

Defines the StackElement class for handling stack operations and arithmetic.

Definition in file StackElement.h.

7.37.2 Function Documentation

7.37.2.1 operator<<()

```
std::ostream & operator<< (
            std::ostream & out,
            const StackElement & val)
```

Outputs the value of a StackElement to an output stream.

Parameters

out	The output stream.
val	The StackElement to output.

Returns

The output stream.

Definition at line 104 of file StackElement.cpp.

7.38 StackElement.h

Go to the documentation of this file.

```
00001
00006 #ifndef STACKELEMENT_H
00007 #define STACKELEMENT_H
00008
00009 #include <cstdint>
00010 #include <variant>
00011 #include <istream>
00012 #include <type_traits>
00013
00017 template<typename T>
00018 concept Rational = std::integral<T> || std::is_same_v<double, T> || std::is_same_v<float, T>;
00019
00024 class StackElement {
00025 public:
00029     std::variant<int64_t, double> value;
00030
00036     template<typename T>
00037     StackElement(T other) {
00038         value = std::variant<int64_t, double>(other);
00039     }
00040
00046     template<typename T>
00047     T Convert() {
00048         return std::visit([](auto a) {
00049             return *((T*)&a);
00050         }, value);
00051     }
00052
00058     template<Rational T>
00059     T Convert() {
00060         return std::visit([](auto a) {
00061             return (T)a;
00062         }, value);
00063     }
00064
00065     // Arithmetic and bitwise operators
00066
00067     StackElement operator+(const StackElement& other);
00068     StackElement operator-(const StackElement& other);
00069     StackElement operator-();
00070     StackElement operator*(const StackElement& other);
00071     StackElement operator/(const StackElement& other);
00072     StackElement operator%(const StackElement& other);
```

```
00073     StackElement operator~();
00074     StackElement operator!();
00075     StackElement operator&(const StackElement& other);
00076     StackElement operator^(const StackElement& other);
00077     StackElement operator|(const StackElement& other);
00078     StackElement operator<(const StackElement& other);
00079     StackElement operator<=(const StackElement& other);
00080     StackElement operator>(const StackElement& other);
00081     StackElement operator>=(const StackElement& other);
00082     StackElement operator==(const StackElement& other);
00083 };
00084
00091 std::ostream& operator«(std::ostream& out, const StackElement& val);
00092
00093 #endif // STACKELEMENT_H
```

7.39 Switch.cpp File Reference

```
#include "Executable.h"
```

7.40 Switch.cpp

Go to the documentation of this file.

```
00001 #include "Executable.h"
00002
00003 Executable::ReturnStatus Switch::Execute(Environment& environment) {
00004     auto selector = environment.PopStack().Convert<int64_t>();
00005     if (cases.contains(selector)) {
00006         return cases[selector]->Execute(environment);
00007     }
00008     return ReturnStatus::kSuccess;
00009 }
00010
```

7.41 VariableCreation.cpp File Reference

```
#include <string>
#include "Executable.h"
```

7.42 VariableCreation.cpp

Go to the documentation of this file.

```
00001 #include <string>
00002 #include "Executable.h"
00003
00004 Executable::ReturnStatus VariableCreation::Execute(Environment& environment) {
00005     if (environment.variables.contains(name)) {
00006         std::string s = "Variable " + name + " is already defined";
00007         throw std::runtime_error(s);
00008     }
00009     size_t byte_size = size;
00010     if (type == "cells" || type == "floats") {
00011         byte_size *= 8;
00012     }
00013     void* allocated_memory = malloc(byte_size);
00014     environment.variables[name] = allocated_memory;
00015     return ReturnStatus::kSuccess;
00016 }
```

7.43 While.cpp File Reference

```
#include "Executable.h"
```

7.44 While.cpp

Go to the documentation of this file.

```
00001 #include "Executable.h"
00002
00003 Executable::ReturnStatus While::Execute(Environment& environment) {
00004     while (true) {
00005         auto status = condition->Execute(environment);
00006         if (status == ReturnStatus::kLeaveLoop) {
00007             break;
00008         }
00009         if (status == ReturnStatus::kLeaveFunction) {
00010             return status;
00011         }
00012         auto elem = environment.PopStack();
00013         if (!elem.Convert<bool>()) {
00014             break;
00015         }
00016         status = body->Execute(environment);
00017         if (status == ReturnStatus::kLeaveLoop) {
00018             break;
00019         }
00020         if (status == ReturnStatus::kLeaveFunction) {
00021             return status;
00022         }
00023     }
00024     return ReturnStatus::kSuccess;
00025 }
```

Index