Algeng Developer Documentation

Tymofii Reizin

September 2, 2023

Contents

1 *Variable.cs* file **2**
1.1 *Variable* class . . . 2

2 *Operator.cs* file **2**
2.1 *UnaryOperator* abstract class . . . 2
2.1.1 *Negation* class . . . 2
2.2 *BinaryOperator* abstract class . . . 2
2.2.1 *Addition* class . . . 2
2.2.2 *Multiplication* class . . . 2

3 *Function.cs* file **2**
3.1 *Function* class . . . 2

4 *Expression.cs* file **3**
4.1 *Expression* abstract class . . . 3
4.1.1 *ConstantExpression* class . . . 3
4.1.2 *VariableExpression* class . . . 3
4.1.3 *FunctionExpression* class . . . 3
4.1.4 *UnaryOperation* class . . . 3
4.1.5 *BinaryOperation* class . . . 4

5 Statement.cs file **4**
5.1 *Statement* class . . . 4

6 Parser.cs file **4**
6.1 *CodeStructure* enum . . . 4
6.2 *Parser* class . . . 5

7 *Program.cs* file **6**
7.1 *Program* class . . . 6

1 *Variable.cs* file

1.1 *Variable* class

A class that represents a single variable. It has a single field *value* that keeps the current value of the variable, and a single property *Value* that allows to get and set *value*. It also has a constructor that takes an initial value of the variable (with default value equal to 0).

2 *Operator.cs* file

File that contains classes for different operators, each should have a member function *Apply*, with corresponding number of arguments, that can be used to apply the operator.

2.1 *UnaryOperator* abstract class

An abstract class for unary operators.

2.1.1 *Negation* class

A class that represents an unary operator that negates a number.

2.2 *BinaryOperator* abstract class

An abstract class for binary operators.

2.2.1 *Addition* class

A class that represents a binary operator that adds two numbers.

2.2.2 *Multiplication* class

A class that represents a binary operator that multiplies two numbers.

3 *Function.cs* file

3.1 *Function* class

A class that represents a single function. It has two fields:

- *_statements* that keeps the statements that compose the body of the function (in order they should be evaluated if the function is called).
- *_returnExpression* keeps the return expression of the function, it is evaluated and returned after executing all the statements from the body.

It has three member functions:

- *AddStatement* that takes a statement and adds it to the body of the function.
- *SetReturn* that takes an expression and sets it as the return statement of the function.
- *Call* that can be used to call the function. It takes lists of variables and functions as arguments, members of which can be used in the execution of the function.

4 *Expression.cs* file

4.1 *Expression* abstract class

A class that represents a single expression. It has a single abstract function *Evaluate*, that takes lists of variables and functions that can be used in the evaluation of the expression, and return the number it evaluates to.

4.1.1 *ConstantExpression* class

A class that represents a constant expression, that is a single number. It has a single field *_constant* that holds the value of the expression. It also has a constructor that can set *_constant*.

4.1.2 *VariableExpression* class

A class that represents a variable expression, that is an expression containing a variable. It has a single field *_variableId* that holds the id of the variable. It also has a constructor that can set the *_variableId*.

4.1.3 *FunctionExpression* class

A class that represents a function expression, that is an expression containing a function call. It has two fields:

- *_functionId* that holds the id of the function called.
- *_arguments* that holds the list of arguments (each is an expression) that the function was called with.

It also has a constructor that can be used to set both its fields.

4.1.4 *UnaryOperation* class

A class that represents an unary operation, that is an expression containing an unary operator applied to a single argument. It has two fields:

- *_operator* that holds the operator.

- *_expression* that holds the expression operator has to be applied to.

It also has a constructor that can be used to set both its fields.

4.1.5 *BinaryOperation* class

A class that represents a binary operation, that is an expression containing a binary operator applied to two arguments. It has three fields:

- *_operator* that holds the operator.
- *_argument* that holds the first expression operator has to be applied to.
- *_argument* that holds the second expression operator has to be applied to.

It also has a constructor that can be used to set all its fields.

5 Statement.cs file

5.1 *Statement* class

A class that represents a statement. It has two fields:

- *_variableId* that holds the id of the variable the result of the evaluation of the expression should be saved to.
- *_expression* the holds the expression.

It has a member function *Execute* that takes lists of variables and functions, and can be used to execute the statement, that is evaluated the expression, and then assign the result to the variable.

6 Parser.cs file

6.1 *CodeStructure* enum

An enumeration that represents the code structure, it has 5 possible values:

1. *Expression* represents an expression.
2. *Statement* represents a statement.
3. *Function* represents a function declaration.
4. *InsideFunctionExpression* represents an expression inside a function.
5. *InsideFunctionStatement* represents a statements inside a function.

6.2 *Parser* class

A class that represents a parser of the code. It is used to parse the code the user inputs. It has 4 fields:

- *_variableNameMap* a dictionary that maps names of variables to their ids.
- *_functionNameMap* a dictionary that maps names of functions to their ids.
- *_insideFunctionVariableNameMap* a dictionary that maps names of variables inside a function to their ids. It it used for variables when the code inside a function is being parsed, since they are separate from the outside variables.
- *_isInFunction* is a boolean variable that keeps if the code is currently inside a function.

It has 9 member functions:

- *GetLineType* that receives a line of code and returns a *CodeStructure* it represents.
- *GetFunctionId* is used to get a function id by it's name, possibly creating a new one, if it hasn't been defined before.
- *GetVariableId* is used to get a variable id by it's name, it also decides that depending on *_isInFunction*, possibly creating a new one, if it hasn't been defined before.
- *ParseAccumulated* is used to parse accumulated values, while parsing an expression. The code accumualtes numbers, names of variables, and name of functions and expressions which are their arguments, and then passes that to this function, to create an expression.
- *CompressStacks* receives expression stack and operator stack, and applies operators to the expressions, to get new expressions, until stacks have enough elements to do that.
- *ParseExpression*(private) a function that receives a string of code and index up to which the parsing has been done, and scans characters one by one, possibly calling itself recursively if there are any brackets, or function arguments, to compute them first, and then use like a single expression.
- *ParseExpression*(public) a function that receives a string of code, does some preprocessing, and call the private version.
- *ParseStatement* a function that receives a string of code and parses it into a statement.
- *ParseFunction* a function that receives a string of code and parses it into a function declaration.

7 *Program.cs* file

7.1 *Program* class

The class for the program. It has a single function *Main* that runs the program. It reads lines one by one, then parses them via *Parser* class, and then executes them.