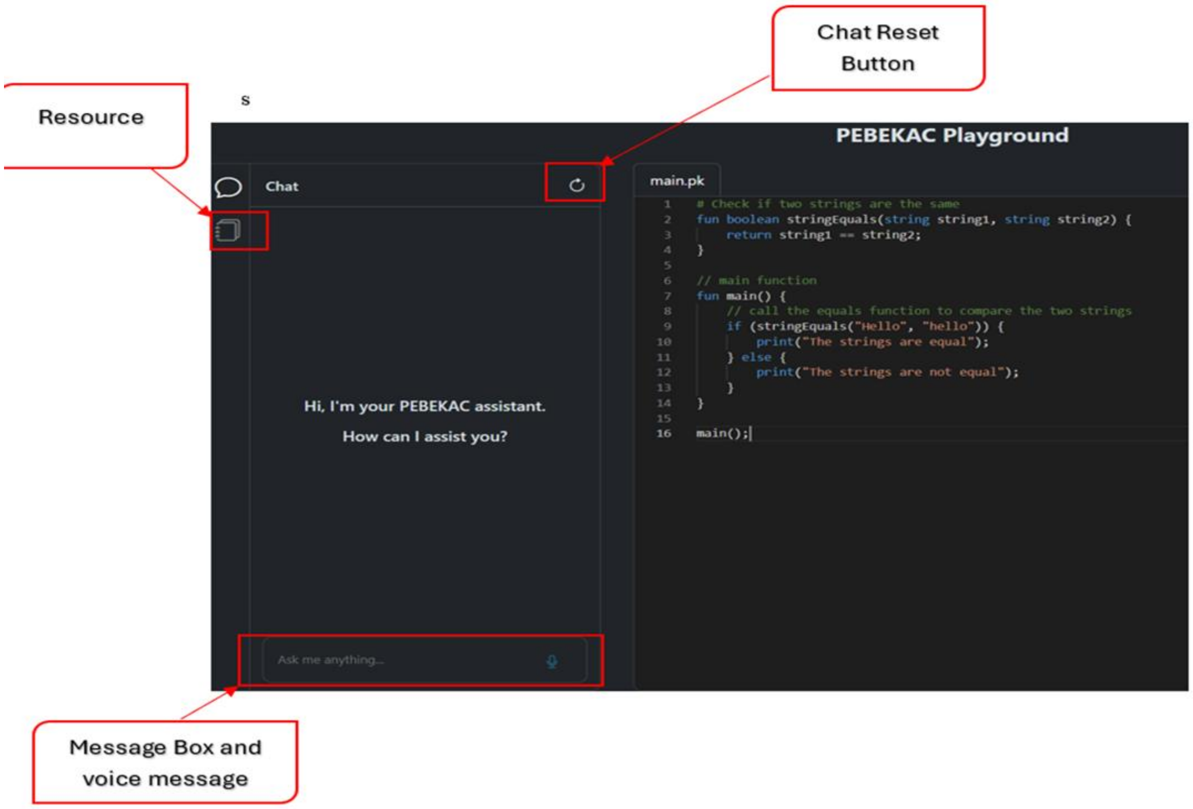
Table of Contents

Introduction

Welcome to the PEBEKAC Online Playground, your interactive coding and chat assistant. This guide will walk you through the features and functionalities of the PEBEKAC Playground software.

PEBEKAC is a high-level programming language integrated with an interactive large language model. It assists users by providing insights and suggestions for solving complex programming problems, thereby enhancing code efficiency, reducing errors, and accelerating development time.

User Interface



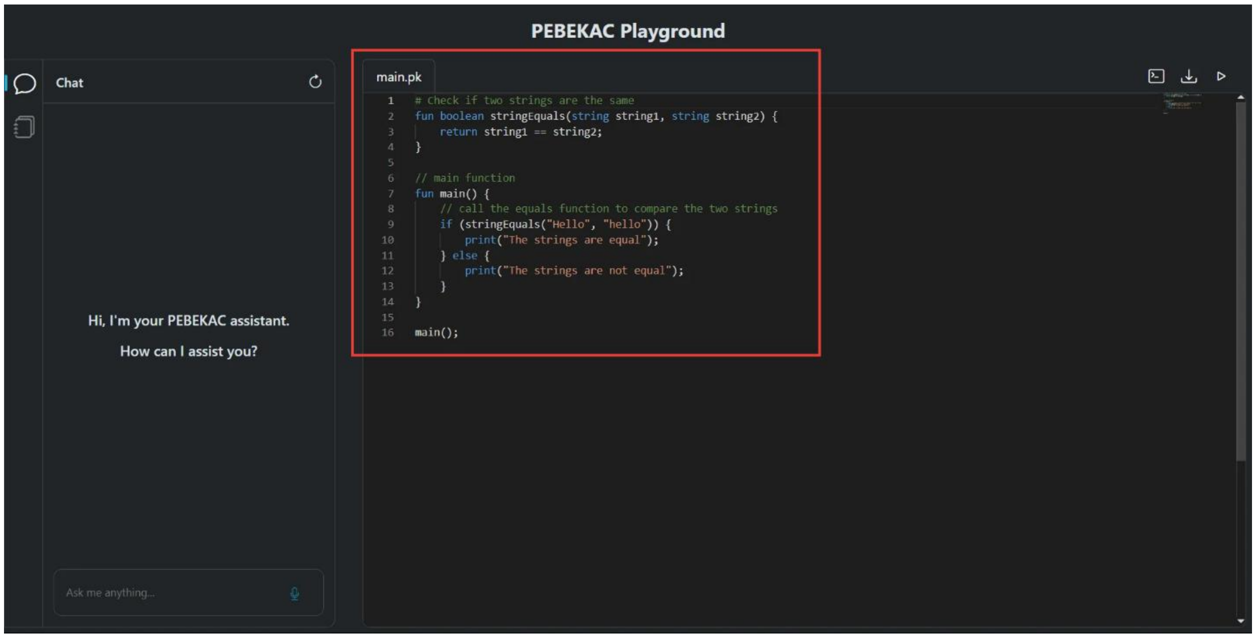
Hi, I'm your PEBEKAC assistant.

How can I assist you?

```
# Check if two strings are the same
fun boolean stringEquals(string string1, string string2) {
    return string1 == string2;
}

// main function
fun main() {
    // call the equals function to compare the two strings
    if (stringEquals("Hello", "hello")) {
        print("The strings are equal");
    } else {
        print("The strings are not equal");
    }
}

main();
```

On the left side of the user interface, you'll find:

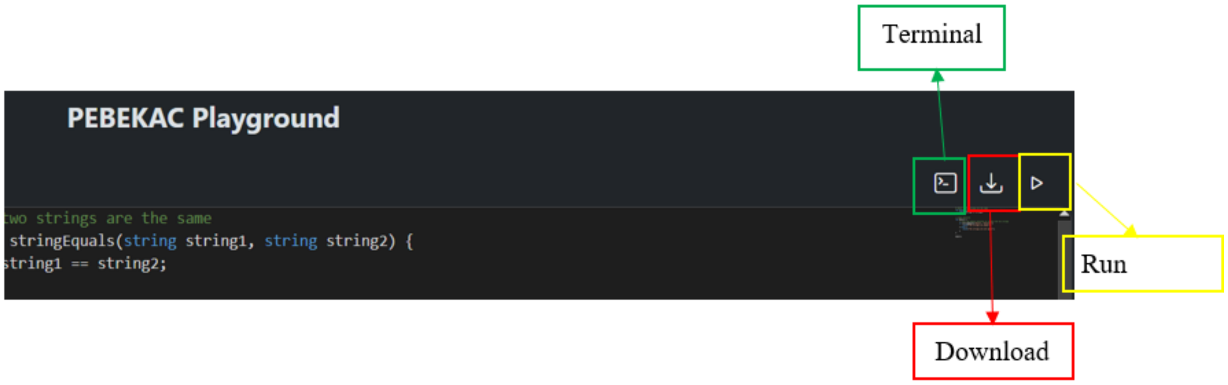
- **AI-Powered PEBEKAC Assistant**: Start a conversation by typing or voicing your message in the chat interface. You'll receive an immediate response from your PEBEKAC assistant.
- **Chat Reset**: You can reset the chat by clicking the icon at the top left of the chat interface.
- **Resources**: You can view documentations, such as tutorials and user manuals.



```
# Check if two strings are the same
fun boolean stringEquals(string string1, string string2) {
    return string1 == string2;
}

// main function
fun main() {
    // call the equals function to compare the two strings
    if (stringEquals("Hello", "hello")) {
        print("The strings are equal");
    } else {
        print("The strings are not equal");
    }
}

main();
```

On the right side, you'll find the **Code Editor**. This is where you can view, write, and edit your code. The editor features:

1. **Syntax Highlighting**: Sample code is displayed with syntax highlights to enhance your understanding of the language. You can define functions, declare variables and use conditional statements to control the flow of your program.
2. **Line Numbers**: Each line of code is numbered for easy reference.

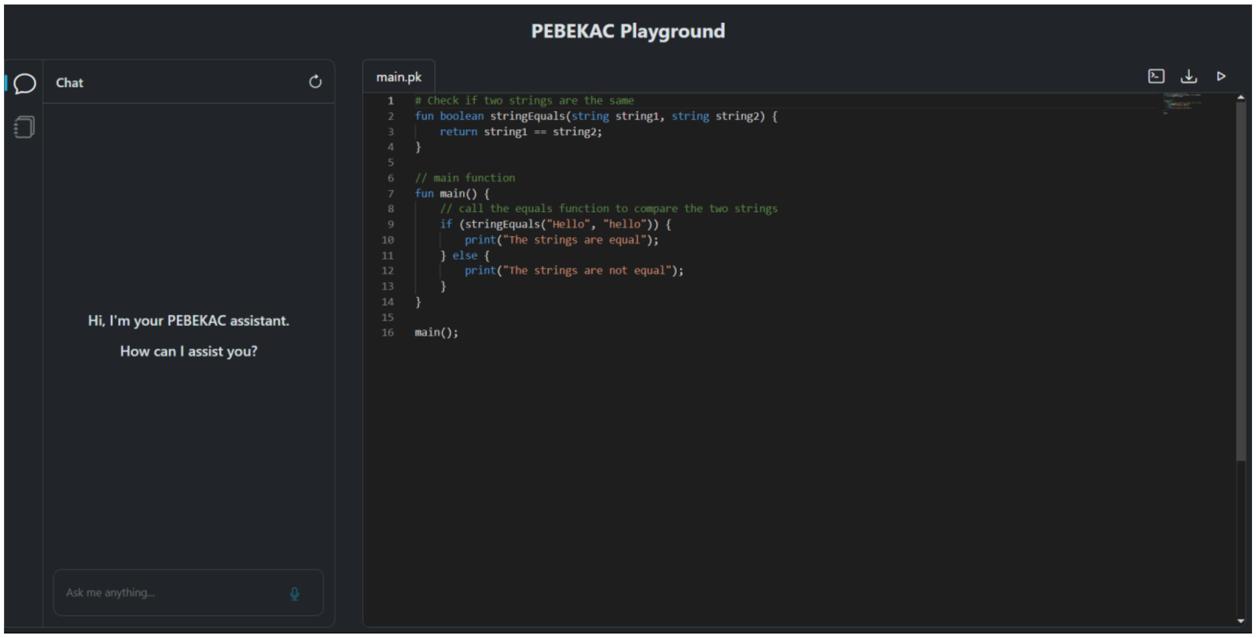


```
two strings are the same
 stringEquals(string string1, string string2) {
tring1 == string2;
```

Additional features include:

- **Terminal Button**: This is where the output for your executed code will appear. The terminal also has a set a few commands such as help, cls and clear.
- **Save Button**: Clicking the save button initiates the download process, saving the current code to a .pk file, to the default download location on your device.

- **Dark Theme**: The chat interface offers a sleek dark theme for comfortable use in low-light conditions.
- **Run Button**: The “Run” button executes the code written in the editor.



```
# Check if two strings are the same
fun boolean stringEquals(string string1, string string2) {
    return string1 == string2;
}

// main function
fun main() {
    // call the equals function to compare the two strings
    if (stringEquals("Hello", "hello")) {
        print("The strings are equal");
    } else {
        print("The strings are not equal");
    }
}

main();
```

Explore our website at https://pebekac.azurewebsites.net/ to staring code in PEBEKAC.

Language Overview

PEBEKAC is a high-level programming language, an acronym for “Problem Exists Between Keyboard and Chair”. It’s a general-purpose language, with its syntax inspired by Python and Kotlin. Designed for readability and ease of writing, PEBEKAC combines elements of imperative, procedural, and functional programming paradigms. This documentation serves as a guide for users to understand the syntax and usage of PEBEKAC. Please note that the language is case-sensitive, meaning the case of letters matter.

Key Features

- **Unit and Constructs**: PEBEKAC emphasizes static binding, allowing type checking during compile time rather than just at runtime. This approach offers several benefits:
    - **Readability**: Static binding enhances code readability by catching type-related errors early.
    - **Maintainability**: By identifying issues before execution, it contributes to better code maintenance.
    - **Runtime Stability**: Static binding reduces the chances of unexpected runtime crashes.
- **Typing**: PEBEKAC enforces strong typing and static typing:
    - **Static Typing**: Variables must be declared with specific data types during compilation.
    - **Strongly Typed**: Type conversions are explicit, ensuring type safety.

Syntax

PEBEKAC follows a structured syntax defined by a set of grammar rules. These rules govern the formation of statements, expressions, declarations, and control structures within the language. The grammar is specified using a combination of production rules, symbols, and terminals.

The PEBEKAC language uses curly braces '{}'. These statements can include function declarations, variable assignments, control structures, print and return statements, etc. Each statement is terminated by a semi-colon ';'.

Function Declaration

A function in the PEBEKAC language is declared using the fun keyword followed by an optional return type, function name, optional parameter list (argument), and a block of statements enclosed in curly braces.

Syntax: 'fun' <return_type>? <identifier> '(' <params> ')' '{' <stmt_list> '}'

The parameter list can take these forms:

1. <general_type> <identifier> ',' <param>
2. <general_type> <identifier>
3. <list> <identifier> ',' <param>
4. <list> <identifier>
5. <array_type> <identifier> ',' <param>
6. <array_type> <identifier>

Function Call

Syntax: <identifier> '(' <arg_list> ')'

Please note that an argument list can take one or more expressions!

Control Structures

PEBEKAC supports common control structures for decision-making and iteration.

- **Selection Control**: if, if-else, and switch statements
    - **If control structure syntax**

    - 'if' '(' <expression> ')' '{' <stmt_list> '}'
    - 'if' '(' <expression> ')' '{' <stmt_list> '}' <else_stmt>
  - **If else control structure syntax**
    - 'else' '{' <stmt_list> '}'
    - 'else' <if_stmt>
- **Iteration Control**: for and while loops
  - **For control structure syntax**
    - 'for' '(' <variable_declaration> <expression> ';' <expression> ')' '{' <stmt_list> '}'
    - 'for' '(' <assignment> <expression> ';' <expression> ')' '{' <stmt_list> '}'
  - **While control structure syntax**
    - 'while' '(' <expression> ')' '{' <stmt_list> '}'

Data Types

PEBEKAC supports the following primitive data types:

- **Int**: the int keyword is used to declare Integer values (e.g., 10, -5).
- **Float**: the float keyword is used to declare floating-point numbers (e.g., 3.14, -0.75).
- **String**: the string keyword is used to declare texts, which can be enclosed in either double or single quotes (e.g., “hello”, ‘PEBEKAC’).
- **Double**: the double is a keyword is used to declare high precision of floating-point numbers (e.g., 3.14, -0.75).
- **Boolean**: the boolean keyword is used to declare boolean values (e.g., true, false).

Data Type Assignment

1. <data_type> <identifier> ‘=’ <expression> ‘;’
2. <data_type> <identifier> ‘=’ <function_call> ‘;’
3. <data_type> <identifier> ‘=’ <null> ‘;’

Data Structures

PEBEKAC provides an array and list data types for storing collections of elements:

Arrays

- **intArray**; the intArray keyword is used to declare a list of elements that are of the int data type
- **floatArray**: the floatArray keyword is used to declare a list of elements that are of the float data type
- **stringArray**: the stringArray keyword is used to declare a list of elements that are of the string data type
- **doubleArray**: the doubleArray keyword is used to declare a list of elements that are of the double data type
- **list**: the list keyword is used to declare a list of elements that are of any of the primitive data types, except Boolean

Syntax

- **Declaration**: <array_type> <identifier> '[' ']' ';'
- **Element Access**: <identifier> '[' <expression> ']' ';'
- **Assignment**:
    1. <array_type> <identifier> '=' <null> ';'
    2. <array_type> <identifier> '=' '[' <expression> ']' ';'
    3. <array_type> <identifier> '=' <function_call> ';'

List

- **List**: List is a keyword in PEBEKAC language used to declare elements of the corresponding primitive types.

Syntax

- **Declaration**: <array_type> <identifier> '{' '}' ';'
- **Element Access**: <identifier> '[' <expression> ']' ';'
- **Assignment**:
    1. <list> <identifier> '=' <null> ';'
    2. <list> <identifier> '=' '[' <expression> ']' ';'
    3. <list> <identifier> '=' <function_call> ';'

Length function

PEBEKAC provides the len function to determine the number of characters/items in an object.

Syntax: <len> '(' <identifier> ')' ';'

Identifiers

In PEBEKAC, an identifier is a name that identifies either a unique object.

Identifier assignment:

- <identifier> '=' <expression> ';'
- <identifier> '=' <function_call> ';'
- <identifier> '=' <null> ';'
- <identifier> '=' <len> '(' <identifier> ')' ';'

Variables

Variables in PEBEKAC are declared using the data type followed by an identifier and a semi-colon.

- **Declaration for Primitive data types**: <data_type> <identifier> ';'
- **Declaration for Array**: <array_type> <identifier> '[' ']' ';'
- **Declaration for List**: <list> <identifier> '{' '}' ';'

Operators

PEBEKAC allows various operations involving arithmetic, logical, comparison, assignments, and relational operators. Parentheses can be used to specify the order of operations.

- **Arithmetic Operators**:
    - +: Plus sign used to add two or more expressions/operands.
      Syntax: <expression> + <expression>
    - - : Subtraction sign used to subtract two or more expressions/operands.
      Syntax: <expression> - <expression>
    - * : Multiplication sign used to multiply two or more expressions/operands.
      Syntax: expression * expression
    - / : Division sign used to divide two or more expressions/operands.
      Syntax: expression / expression

  - % : Modulus sign used to find the modulus of two or more expressions/operands.
    Syntax: expression % expression
- **Assignment Operators**:
  - = : Equal operator is used to assign the operand on the right to the one on the left.
    Syntax: <expression> = <expression>
  - += : Addition assignment is used to perform addition and assigns the result to the left operand.

    Syntax: <expression> += <expression>
  - -= : Subtraction assignment is used to perform subtraction assign the result to the left operand.

    Syntax: <expression> -= <expression>
  - *= : Multiplication assignment is used to perform multiplication and assign the result to the left operand.

    Syntax: <expression> *= <expression>
  - /= : Division assignment is used to perform division and assign the result to the left operand.

    Syntax: <expression> /= <expression>
  - %= : Modulus assignment is used to perform modulus and assign the result to the left operand.

    Syntax: <expression> %= <expression>
- **Unary Operators**:
  - -<digit>: Unary minus operator negates the value of the operand.
- **Comparison Operators**:
  - == : The equality operator returns true if both operands are equal, otherwise it returns false.

Syntax: <expression> == <expression>

  - != : The inequality operator returns true if the operands are not equal, otherwise it returns false.

    Syntax: <expression> != <expression>

  - < : Less than operator returns true if the left operand is less than the right operand, and false otherwise.

    Syntax: <expression> < <expression>

  - > : The greater than operator returns true if the left operand is greater than the right operand, and false otherwise.

    Syntax: <expression> > <expression>

  - <= : The less than or equal operator returns true if the left operand is less than or equal to the right operand, and false otherwise.

    Syntax: <expression> <= <expression>

  - >= : The greater than or equal operator returns true if the left operand is greater than or equal to the right operand, and false otherwise.

    Syntax: <expression> >= <expression>

- **Logical Operators**:

  - && : Logical AND indicates whether both operands are true.

    Syntax: <expression> && <expression>

  - || : Logical OR indicates true if and only if one or more of its operands is true.
    Syntax: <expression> || <expression>

  - ! : Logical NOT operator negates the operand.

    Syntax: ! <expression>

- **Conditional Operator**:

    - ?: : Ternary operator is a concise way of checking if a condition is true or false. Syntax: condition ? true : false

- **Array and List Operators**:
    - Element access: array[index], List[index]
    - Array/List length: len(array), len(List)
- **String Operators**:
    - Concatenation: string + string
    - String comparison: string == string, string > string, string < string

Return Statement

PEBEKAC provides the return function to end the execution of a function and returns control to the calling function. Use the return syntax followed by a semi-colon.

Syntax: 'return' <expression> ';'

Break Statement

PEBEKAC provides the break statement that allows you to terminate and exit a loop using the break keyword followed by a semi-colon.

Syntax: 'break' ';'

Displaying output

PEBEKAC provides the print function to display output to the console. Use the print keyword followed by an expression or function call, then a semi-colon.

Syntax:

'print' '(' <expression> (',' <function_call> ',' <expression>)? ')' ';'
'print' '(' <function_call> ')' ';'

Accepting input

PEBEKAC doesn’t support the acceptance of inputs.

String Literal

Syntax: " <identifier> " OR ' <identifier> '

Comments

PEBEKAC supports single-line comments by using // or # and multiple line comments using /* */.

Comment Syntax:

1. // Identifier
2. # identifier
3. /* identifier */

Reserved Words and Keywords

In PEBEKAC, it's essential to be aware of reserved words and keywords that cannot be used as identifiers. These reserved words are integral to the language's syntax and functionality, serving specific purposes such as defining control structures, data types, and function declarations. Attempting to use these reserved words as identifiers for variables, functions, or other constructs will result in syntax errors. Therefore, when naming identifiers such as variables or functions, ensure they do not match any of the reserved words listed in the language specification. By adhering to this guideline, you can avoid potential conflicts and write clean, error-free PEBEKAC code.

Reserved words

• def	• if	• const
• raise	• for	• int
• None	• while	• float
• del	• do	• double
• import	• else	• string
• return	• break	• private
• elif	• true	• public
• in	• false	• intList
• try	• print	• floatList
• and	• nonlocal	• stringList
• is	• yield	• doubleList
• as	• not	• intArray
• except	• class	• floatArray
• lambda	• form	• stringArray
• with	• or	• doubleArray
• assert	• continue	• boolean
• finally	• global	• void
• null	• pass	• len

Keywords

• fun	• intList	• int
• return	• floatList	• float
• if	• stringList	• double
• else	• doubleList	• string
• for	• intArray	• boolean
• while	• floatArray	• true
• void	• stringArray	• false
• break	• doubleArray	
• null		

Tokens and Regular Expressions

Token	**Regular Expression**	**Token**	**Regular Expression**
t_PLUS	r ‘\+’	t_COLON	r ‘ : ’
t_MINUS	r ‘-’	t_ASSIGN	r ‘ = ’
t_MULTIPLY	r ‘*’	t_INCREMENT	r ‘\+ \+’
t_DIVIDE	r ‘/’	t_DECREMENT	r ‘ -- ’
t_LPAREN	r ‘\(’	t_POW	r ‘* *’
t_RPAREN	r ‘\)’	t_AT	r ‘@’
t_MODULUS	r ‘%’	t_HASH	r ‘\#’
t_EQUAL	r ‘==’	t_QUESTION	r ‘\?’
t_NOTEQUAL	r ‘!=’	t_BACKSLASH	r ‘\ \’
t_LESSTHAN	r ‘<’	t_SLASH	r ‘ / ’
t_GREATERTHAN	r ‘>’	t_APOSTROPHE	r ‘\ ‘ ’
t_LESSTHANEQUAL	r ‘<=’	t_DOUBLEQUOTE	r ‘\ “ ’
t_GREATERTHANEQUAL	r ‘>=’	t_PIPE	r ‘\ |’
t_AND	r ‘&&’	t_PLUSASSIGN	r ‘\+=’
t_OR	r ‘\ | \ |’	t_MINUSASSIGN	r ‘\-=’
t_NOT	r ‘!’	t_TIMEASSIGN	r ‘*=’
t_SEMICOLON	r ‘ ; ’	t_DIVIDEASSIGN	r ‘ /=’
t_LBRACE	r ‘{’	t_MODASSIGN	r ‘%=’
t_RBRACE	r ‘}’	t_Arrow	R '->’
t_LBRACKET	r ‘\[’	t_ignore	‘ \t’
t_RBRACKET	r ‘\]’		
t_COMMA	r ‘ , ’		
t_DOT	r ‘ . ’		

Grammar

```
<program>
    : <stmt_list>
    ;

<stmt_list>
    : <stmt> <stmt_list>
    | <stmt>
    ;

<stmt>
    : <fun_declaration>
    | <print_stmt>
    | <len_stmt> ';'
    | <function_call> ';'
    | <return_stmt>
    | <variable_declaration>
    | <assignment>
    | <compound_assignment> ';'
    | <control_structure>
    | <break_stmt>
    | <comment>
    ;

<fun_declaration>
    : 'fun' <return_type>? <identifier> '(' <params> ')' '{' <stmt_list> '}'
    ;

<params>
    : <param>
    ;

<param>
    : <general_type> <identifier> ',' <param>
    | <general_type> <identifier>
    | <list> <identifier> ',' <param>
    | <list> <identifier>
    | <array_type> <identifier> ',' <param>
    | <array_type> <identifier>
    ;
```

```
<len_stmt>
    : 'len' '(' <identifier> ')'
    ;

<print_stmt>
    : 'print' '(' <expression> (',' <function_call> ',' <expression>)? ')' ';'
    | 'print' '(' <function_call> ')' ';'
    ;

<function_call>
    : <identifier> '(' <arg_list> ')'
    ;

<arg_list>
    : <expression> ',' <arg_list>
    | <expression>
    ;

<return_stmt>
    : 'return' <expression> ';'
    ;

<variable_declaration>
    : <general_type> <identifier> ';'
    | <list> <identifier> '{' '}' ';'
    | <array_type> <identifier> '[' ']' ';'
    ;

<assignment>
    : <general_type> <identifier> '=' <expression> ';'
    | <general_type> <identifier> '=' <function_call> ';'
    | <general_type> <identifier> '=' <null> ';'
    | <list> <identifier> '=' '{' <expression> '}' ';'
    | <list> <identifier> '=' <function_call> ';'
    | <list> <identifier> '=' <null> ';'
    | <array_type> <identifier> '=' '[' <expression> ']' ';'
    | <array_type> <identifier> '=' <function_call> ';'
    | <array_type> <identifier> '=' <null> ';'
    | <identifier> '=' <expression> ';'
    | <identifier> '=' <function_call> ';'
    | <identifier> '=' <null> ';'
    | <identifier> <assignment_sign> <function_call> ';'
```

```
      | <identifier> '=' <len_stmt> ';'
      ;

<control_structure>
      : <if_stmt>
      | <for_stmt>
      | <while_stmt>
      ;

<break_stmt>
      : 'break' ';'
      ;

<comment>
      : '//' <identifier>
      | '#' <identifier>
      | '/*' <identifier> '*/'
      ;

<return_type>
      : <general_type>
      | <array_type>
      | <list>
      | 'void'
      ;

<if_stmt>
      : 'if' '(' <expression> ')' '{' <stmt_list> '}'
      | 'if' '(' <expression> ')' '{' <stmt_list> '}' <else_stmt>
      | <expression> '?' <expression> ':' <expression> ';'
      ;

<else_stmt>
      : 'else' '{' <stmt_list> '}'
      | 'else' <if_stmt>
      ;

<for_stmt>
      : 'for' '(' <variable_declaration> <expression> ';' <expression> ')' '{' <stmt_list> '}'
      | 'for' '(' <assignment> <expression> ';' <expression> ')' '{' <stmt_list> '}'
      ;
```

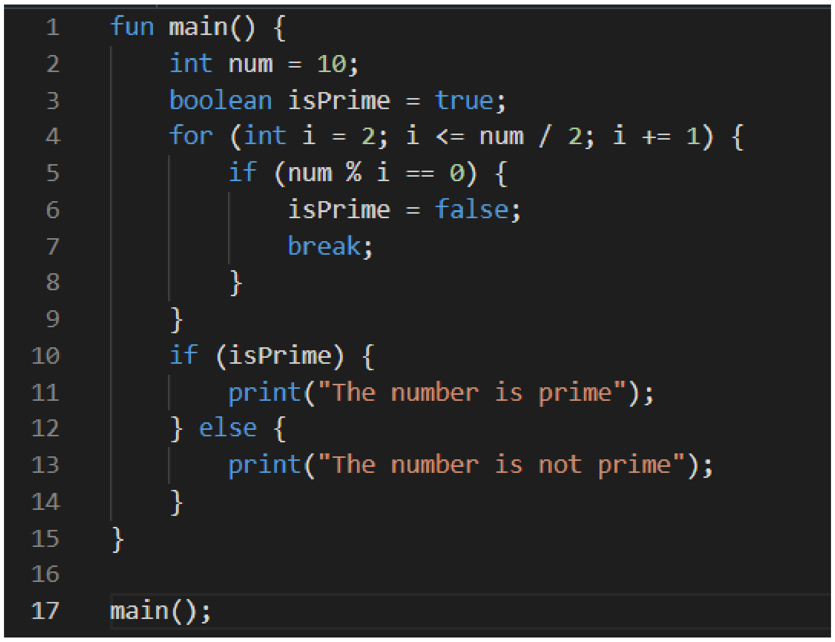
```
<while_stmt>
    : 'while' '(' <expression> ')' '{' <stmt_list> '}'
    ;

<expression>
    : <expression> '+' <expression>
    | <expression> '-' <expression>
    | <expression> '*' <expression>
    | <expression> '/' <expression>
    | <expression> '%' <expression>
    | <expression> '&&' <expression>
    | <expression> '||' <expression>
    | <expression> '==' <expression>
    | <expression> '!=' <expression>
    | <expression> '<' <expression>
    | <expression> '>' <expression>
    | <expression> '<=' <expression>
    | <expression> '>=' <expression>
    | <expression> ',' <expression>
    | <expression> '**' <expression>
    | '!' <expression>
    | '(' <expression> ')'
    | <identifier>
    | <int>
    | <float>
    | <string_literal>
    | <boolean>
    | <element_access>
    | <function_call>
    | <compound_assignment>
    | <len_stmt>
    | <null>

<compound_assignment>
    : <expression> <assignment_sign> <expression>
    | <identifier> <assignment_sign> <expression>
    ;
```

```
<assignment_sign>
    : '+='
    | '-='
    | '*='
    | '/='
    | '%='
    ;

<int>
    : '-'?[0-9]+
    ;

<float>
    : '-'?[0-9]+'.'[0-9]+
    ;

<string>
    : [a-zA-Z]
    ;

<identifier>
    : ('_')?(<string>|<int>)+
    ;

<boolean>
    : 'true'
    | 'false'
    ;

<null>
    : 'null'
    ;

<string_literal>
    : '"'<identifier>'"'
    | "'" <identifier> "'"
    ;

<general_type>
    : 'int'
    | 'float'
    | 'double'
    | 'string'
    | 'boolean'
    ;

<array_type>
    : 'intArray'
    | 'floatArray'
    | 'stringArray'
    | 'doubleArray'
    ;

<list>
    : 'list'
    ;

<element_access>
    : <identifier> '[' <expression> ']'
    ;
```

Sample Programs

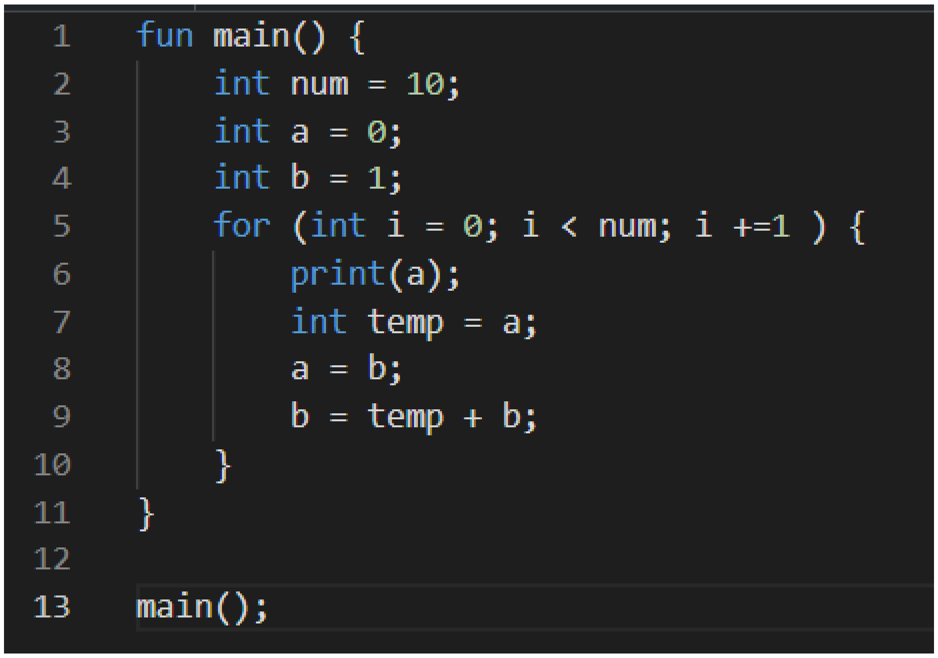
Below are sample programs written in PEBEKAC to help you get started.

Prime Number Checker



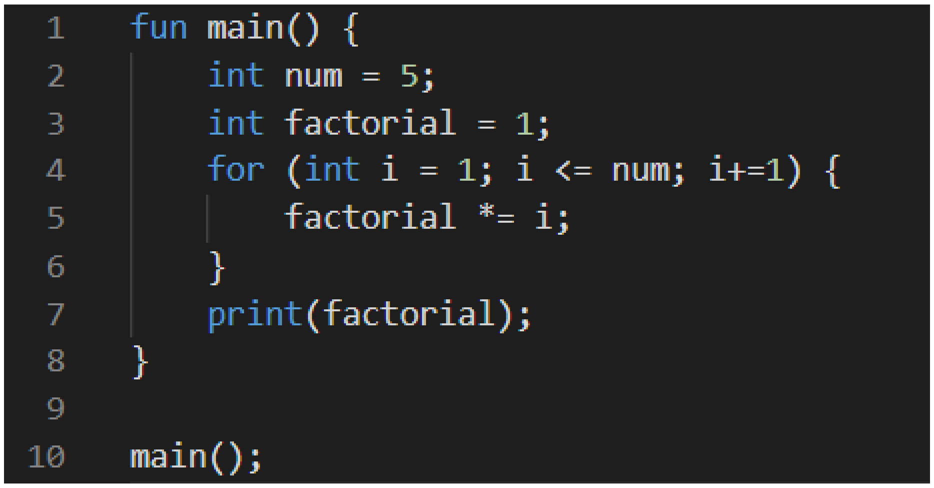
```
fun main() {
    int num = 10;
    boolean isPrime = true;
    for (int i = 2; i <= num / 2; i += 1) {
        if (num % i == 0) {
            isPrime = false;
            break;
        }
    }
    if (isPrime) {
        print("The number is prime");
    } else {
        print("The number is not prime");
    }
}

main();
```

Fibonacci Series



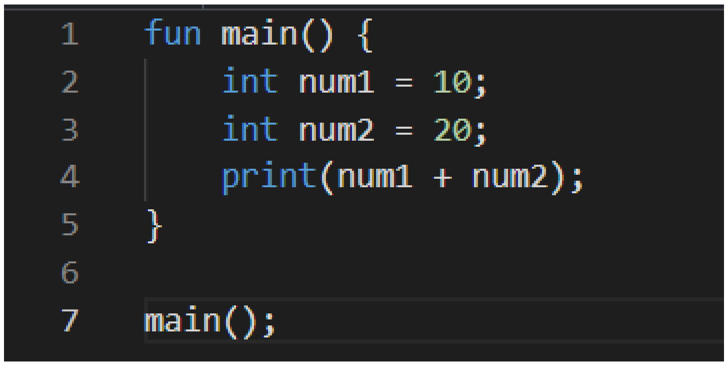
```
fun main() {
    int num = 10;
    int a = 0;
    int b = 1;
    for (int i = 0; i < num; i +=1 ) {
        print(a);
        int temp = a;
        a = b;
        b = temp + b;
    }
}

main();
```

Factorial



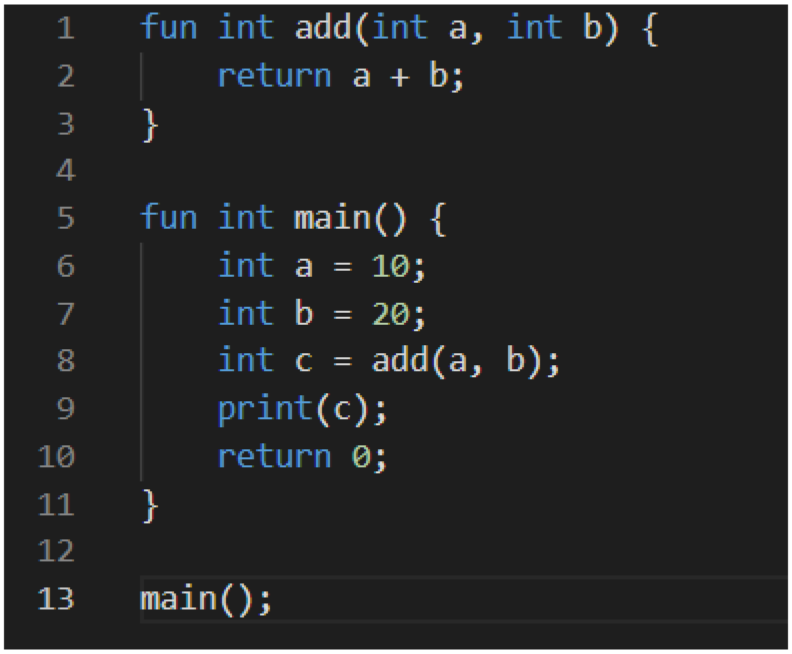
```
fun main() {
    int num = 5;
    int factorial = 1;
    for (int i = 1; i <= num; i+=1) {
        factorial *= i;
    }
    print(factorial);
}

main();
```

Sum of two numbers



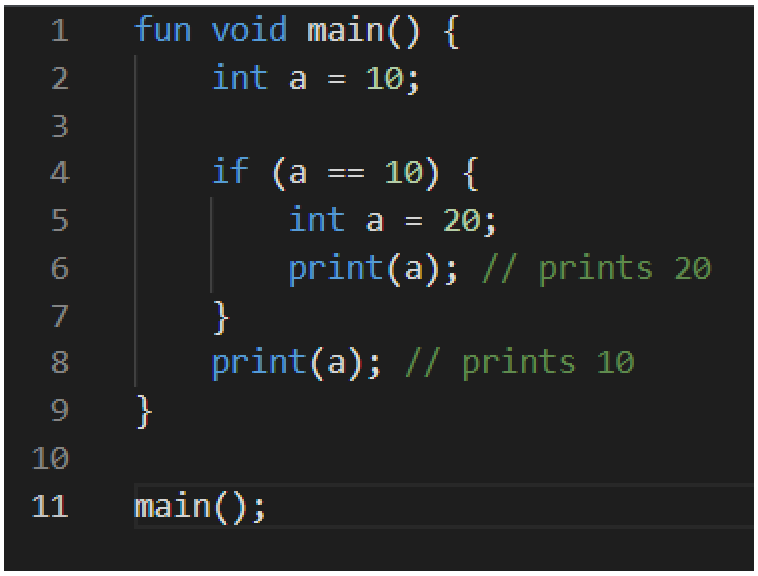
```
fun main() {
    int num1 = 10;
    int num2 = 20;
    print(num1 + num2);
}

main();
```

Demonstrating functions



```
fun int add(int a, int b) {
    return a + b;
}

fun int main() {
    int a = 10;
    int b = 20;
    int c = add(a, b);
    print(c);
    return 0;
}

main();
```

Demonstrating scopes



```
fun void main() {
    int a = 10;

    if (a == 10) {
        int a = 20;
        print(a); // prints 20
    }
    print(a); // prints 10
}

main();
```