

LETS_GO+

Language Design and Example Programs

Version 1.0.0



1. Introduction

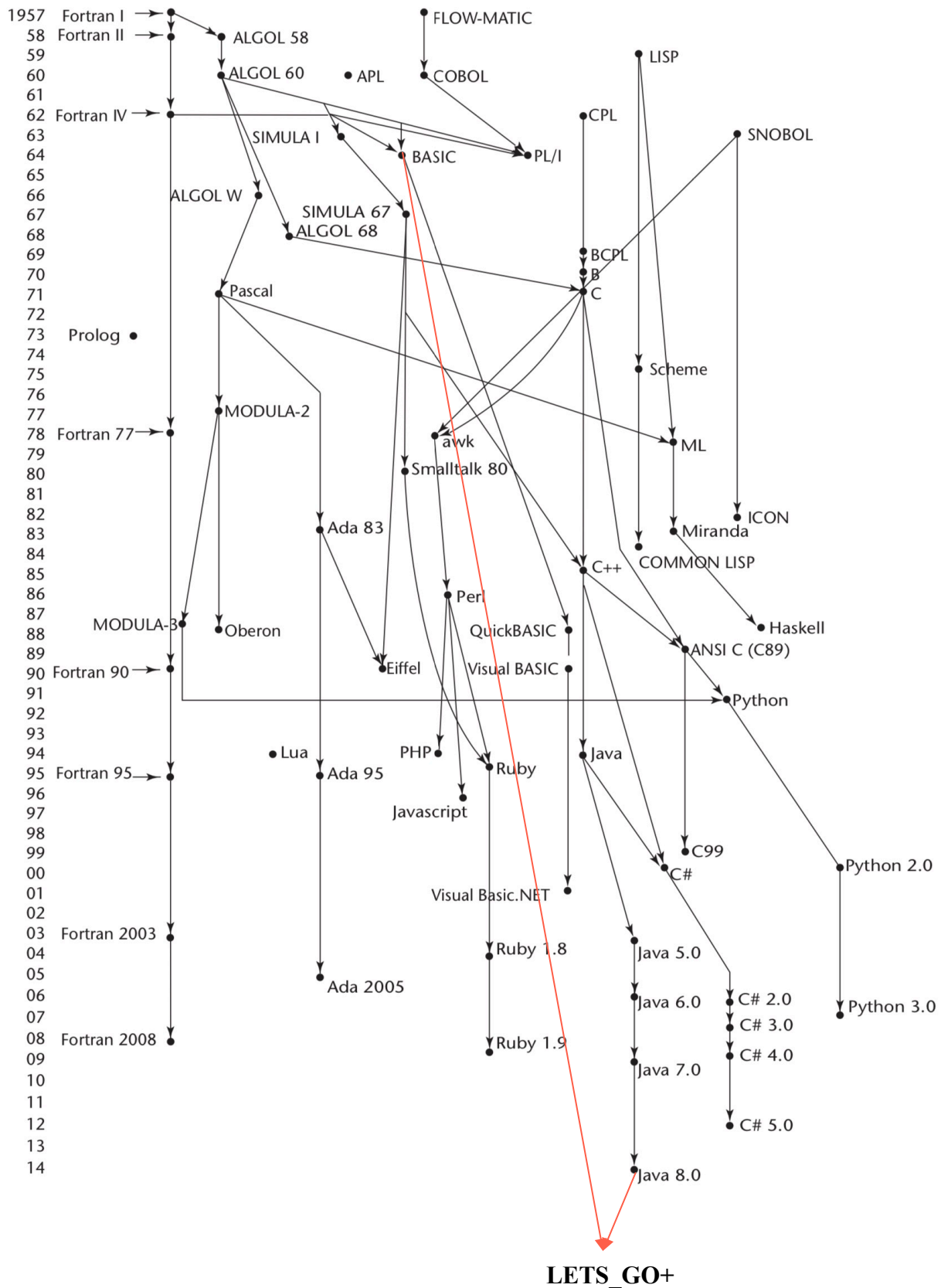
LETS_GO+ (pronounced "LETS GO Plus") is a simple, modern, object-oriented, and (strongly) type-safe programming language. During my programming journey, I really liked Java and Basic and wanted to combine both to see what new programming language can come about. And what we get is LETS_GO+. A futuristic programming language that has many built in features that help beginner programmers learn and become confident in their programming skills. This is a great first language to learn as it is beginner friendly and includes many built-in features that can help enhance your programming skills. So, LETS_GO and learn some code.

How LETS_GO+ differs from Java and Basic:

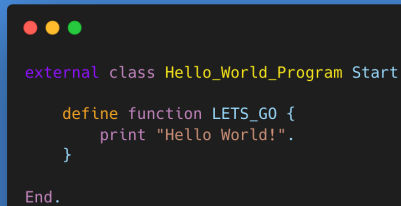
1. The original Basic, is not object oriented, and so this implementation of LETS_GO+ supports object-oriented programming. Object orientated programming is supported in Java, but we have a simpler way of defining and using them. Object oriented programming is tremendously helpful, especially when it comes to learning the basics of programming and then moving on to more complex topics. LETS_GO+ makes it very simple to use and understand, so don't worry you're in good hands.
2. There is no main function at the beginning of your files just like in Java, you begin with a LETS_GO method. This LETS_GO serves as the entry point. As soon as you type LETS_GO, you can begin coding. Without this, you cannot do anything other than define objects in a class.
3. There are no line numbers included from Basic, because these lead to confusion, and we are trying to make this as nice and easy to use as possible. which means we do not support GOTO.
4. we only support 3 primitive data types. They are number, string, and Boolean. These are the 3 most important, and beginner friendly we can take from both Basic and Java. In addition, users can define and use custom class types (e.g., student_info) for object-oriented structures.
5. we only support 3 data structures. They are Array, HashMap, and Tree. These are the 3 most important, and beginner friendly we can take from both Basic and Java. Arrays can be dynamically or statically set from initialization. HashMap and Trees are more advanced data structures, but there are built in capabilities for these as well.
6. In early Basic, there was no such thing as private, public, etc. In Java there is, and we feel that we need to support at least private and public modifiers. In LETS_GO+, there are called internal (private) and external (public).
7. we liked Basic's simple printing style and so we took that instead of Java's. We also liked Basic's begin and end for methods, and so we adopted the start and end to the beginning of class declarations.

8. Every statement ends with a period (.). In both Java and Basic everything ended with a semi-colon, but we feel that because we are accustomed to ending sentences with periods, it feels more natural for beginners to end their statements with periods as well.

1.1.Genealogy



1.2. Hello world



```
external class Hello_World_Program Start
    define function LETSGO {
        print "Hello World!".
    }
End.
```

1.3. Program structure

The key organizational concepts in LETS_GO+ are as follows:

1. Every function you define must start with a *define function function_name {...}*
2. Every file must start with a define function LETSGO, that serves as the entry point for the file. Unless you are building an object.
3. You must also specify at the start of the file, the name of the class, and whether it is external or internal. External meaning it is visible everywhere, internal meaning it is visible only locally. *external class name_of_program Start ... End.* The start marks the starting of the class and the end, marks the end of the class.
4. Once you define the function itself, you must surround the code inside of it with curly braces. We believe that it makes it easy and simpler to see.
5. When creating a class, in java you usually have to define the getters and setter methods, but with LETS_GO+, you don't have to worry about declaring those. There is a built-in system that recognizes a class and automatically creates the getters and setters for you based off of the attributes. So you don't have to worry about that. See example down below. You do need the constructor though, as this constructor is going to be used to create those getters and setters.
6. A constructor for a class start off like this: *constructor constructor_name(...){...}*. The parameters that it takes are the variables that you define as attributes for the class. And since we are extremely type safe, you must in the parameters specify the type of the parameters.
7. When initializing your constructor, you are to use this structure,
attribute:attribute_name = parameter_name.
8. You must also initialize any variables you want to use, if they are not constructor attributes.
9. All variable declarations use the (:=) sign, instead of =, because we saved the = for comparison, as that is what we are used to.

Here is an example of a mini program, that has a `student_info` class that holds the students information and a `student_program` class that is the entry point to the whole program, that adds students to a dynamically sized array, or displays the student array we have:

```
external class student_info Start
  constructor_attribute string first_name.
  constructor_attribute string last_name.
  constructor_attribute number id.

  constructor student_info(string first_name, string last_name, number id) {
    attribute:first_name := first_name.
    attribute:last_name := last_name.
    attribute:id := id.
  }
End.
external class student_program Start
  grab student_info class.
  string input := "".
  Array students <|student_info|> [].

  define function LETS_GO {
    input := accept_string_input(print "Would you like to add a new student or display information (Type add or display)").
    if (input = "add") {
      string first_name := "".
      string last_name := "".
      number id := 0.

      first_name := accept_string_input(print "Provide first name: ").
      last_name := accept_string_input(print "Provide last name: ").
      id := accept_number_input(print "Provide id: ").

      call addStudent(first_name, last_name, id).
    }else{
      call display_student_info(students).
    }
  }

  define function addStudent(string first_name, string last_name, number id) {
    let new_student := student_info(first_name, last_name, id).
    students.add(new_student).
  }

  define function display_student_info(Array students){
    number iterator := 0.
    for(iterator to length(students)):
      print "First name: ", students.get_first_name[iterator],
        "Last name: ", student.get_last_name[iterator],
        "ID Number: ", student.get_id[iterator].
      NEXT iterator.
  }
End.
```

1.4.Types and Variables

There are two kinds of types in LETS_GO+: *value types* and *reference types*. Variables of value types directly contain their data whereas variables of reference types store references to their data, the latter being known as objects. With reference types, it is possible for two variables to reference the same object and thus possible for operations on one variable to affect the object referenced by the other variable.

1.5.Visibility

In LETS_GO+, there are two visibility types, external and internal. External represents public visibility, meaning everything has access to it. Internal represent private visibility, meaning only functions, attributes, etc. in the same class have access to it.

1.6.Statements Differing from Java and Basic

Statement	Example
Function Definition	<pre>define function LETS_GO { print "Hello world". }</pre>
Expression Statement	<pre>external class expression_statement Start define function LETS_GO { number a := 5. string b := "hey". print a. print b. print "LETS GO". Array < number > evenNumbers [5]. evenNumbers.add(2). } End.</pre>
For-Loop Statement	<pre>external class for_loop_statement Start define function LETS_GO { number iterator := 0. Array < number > random_Numbers [5]. random_Numbers.add(56). random_Numbers.add(25). random_Numbers.add(100). random_Numbers.add(1). random_Numbers.add(250). for(iterator to length(random_Numbers)): print "Number ", iterator, " : ", random_Numbers[iterator]. Next iterator. } End.</pre>

Input Accept Statement	<pre> external class input_statement Start define function LETS_GO { number b := accept_number_input(print "Enter in a number: "). print b. string s := accept_string_input(print "Enter in your name: "). print s. boolean a := accept_boolean_input(print "True or False: LETS_GO+ is the best programming language in the world?"). print a. } End. </pre>
Constructor Attribute Statement	<pre> external class sample_Program Start constructor attribute string first_name. constructor attribute string last_name. constructor attribute number id. constructor student_info(string first_name, string last_name, number id) { attribute.first_name := first_name. attribute.last_name := last_name. attribute.id := id. } } End. </pre>

2. Lexical structure

2.1. Programs

A LETS_GO+ *program* consists of one or more *source files*. A source file is an ordered sequence of (probably Unicode) characters.

Conceptually speaking, a program is compiled using three steps:

1. Transformation, which converts a file from a particular character repertoire and encoding scheme into a sequence of Unicode characters.
2. Lexical analysis, which translates a stream of Unicode input characters into a stream of tokens.
3. Syntactic analysis, which translates the stream of tokens into executable code.

2.2. Grammars

This specification presents the syntax of the LETS_GO+ programming language where it differs from Java and Basic.

2.2.1. Lexical grammar (tokens) where different from Java and Basic

<Assignment Operator> → :=
<Math Operators> → + | * | ÷ | -
<Remainder Operator> → Remainder (number)
<String Concatenation> → ,
<Begin Block> → Start (For classes), { (for functions)
<Close Block> → End. (For classes), } (for functions)
<Number> → +/- (0|([1-9][0-9]*)) '.' (0|([1-9][0-9]*))*
<String> → [a-z][a-Z0-9]*
<Array> → Array <|Type|> Array_Name [Size (If not set, will be dynamically sized)].
<End_of_All_Statements> → .
<Array_length> → length(Array_Name)
<Input> → accept_type_input()
<Single-Line Comments> → #
<Multi-Line Comments> → #|...|
<Creating_New_Object> → let object_name = new object_type()
<Less than, greater than operators> → <, >, ≤, ≥

2.2.2.Syntactic (“parse”) grammar where different from Java and Basic

<Class Declaration> → *<Modifier> class “class_name” Start*

<Function Definition> →
define function “function_name” (<parameters>)

<Constructor Attribute Declaration> →
constructor_attribute <type> “name”

<Constructor Attribute Setting> →
attribute: “constructor_attribute_name” := <parameter>

<Import Class> → *grab “class_name” class.*

<Array Declaration> → *Array “array_name” <|Array Type|> [].*

<Ending Class Declaration> → *End.*

<Statement Ending> → *.*

<Input Acceptance> → *accept_<Type>_input().*

<Object Instantiation> →
let <object_name> := “constructor_name” (<parameters>).

<Function Calling> → *call “function_name” <parameters> .*

<Single-Line Comments> → *# <comment>*

<Multi-Line Comments> → *#!/ <comment> /#*

2.3.Lexical analysis

2.3.1.Comments

Two forms of comments are supported: single-line comments and delimited comments. *Single-line comments* start with the character # and extend to the end of the source line. *Delimited comments* start with the characters #| and end with the characters |#. Delimited comments may span multiple lines. Comments can nest, as this may be useful in certain situations, so we made it possible for you to nest comments.

2.4.Tokens

There are several kinds of tokens: identifiers, keywords, literals, operators, and punctuators. White space and comments are not tokens, though they act as separators for tokens where needed.

Tokens:

identifier

keyword

number-literal

string-literal

Boolean-literal

Remainder-literal

String Concatenation-literal

operator-or-punctuator

2.4.1.Keywords different from Java or Basic

A **keyword** is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier except when prefaced by the @ character.

New keywords:

start, end, accept_TYPE_input, constructor_attribute, attribute,
let, internal, external, define, function, call, grab, number,
remainder

Removed keywords:

Goto, public, private, import, def, system.out.println, int, float,
mod

3.Type System

LETS_GO+ uses a *strong static* type system. LETS_GO+ is attempting to become the worlds easiest programming language to use and understand and so making this a strongly typed system, it will catch type errors and let the user know where and how to fix it. This is extremely important to help with building those basic skills as a beginner. Static typing means early binding, compile-time type checking. This is just more intuitive and easier to debug as a user.

3.1.Type Rules

The type rules for LETS_GO+ are as follows:

Arithmetic Operations:

```
s ⊢ e1: Number
s ⊢ e2: Number
Number is an integer or decimal
-----
s ⊢ e1 + e2: Number
```

```
s ⊢ e1: Number
s ⊢ e2: Number
Number is an integer or decimal
-----
s ⊢ e1 - e2: Number
```

```
s ⊢ e1: Number
s ⊢ e2: Number
Number is an integer or decimal
-----
s ⊢ e1 * e2: Number
```

```
s ⊢ e1: Number
s ⊢ e2: Number
Number is an integer or decimal
-----
s ⊢ e1 ÷ e2: Number
```

Comparisons:

```
s ⊢ e1: T
s ⊢ e2: T
T is a primitive type
-----
s ⊢ e1 = e2: Boolean
```

```
s ⊢ e1: T
s ⊢ e2: T
T is a primitive type
-----
s ⊢ e1 ≠ e2: Boolean
```

```
s ⊢ e1: T
s ⊢ e2: T
T is a primitive type
-----
s ⊢ e1 < e2: Boolean
```

```
s ⊢ e1: T
s ⊢ e2: T
T is a primitive type
-----
s ⊢ e1 > e2: Boolean
```

```
s ⊢ e1: T
s ⊢ e2: T
T is a primitive type
-----
s ⊢ e1 ≤ e2: Boolean
```

```
s ⊢ e1: T
s ⊢ e2: T
T is a primitive type
-----
s ⊢ e1 ≥ e2: Boolean
```

String Concatenation:

```
s ⊢ e1: String
s ⊢ e2: String
String also includes single characters
-----
s ⊢ e1 , e2: String
```

Assignment:

```
s ⊢ e1: T
s ⊢ e2: T
T is a primitive type
-----
s ⊢ e1 := e2: T
```

LETS_GO+ types are divided into two main categories: *Value types* and *Reference types*.

3.2.Value types (different from Java and Basic)

- Number: Just like any other number we think about. This also includes negative numbers, decimals etc. We didn't want two separate identifiers for each.
- Booleans: Represents a logical value either true or false.
- Strings: Represents a sequence of characters or single character.

These types are passed by value, meaning the variable holds the actual data.

3.3.Reference types (differing from Java and Basic)

- Object creation: When you create your own class in LETS_GO+, you instantiate your attributes with a constructor. These objects are then stored and passed by reference, so changes to the object affect all references to it. This is because object creation is generally complex in nature, and to help separate this with the simple value types we have, we decided that by passing any object creation by reference is the best way to go about this.

4.Example Programs

Caesar Cipher
with LETS_GO
method that
takes in
input and
then either
goes to
Encrypt or
Decrypt:

```
external class Caesar_Cipher_Program Start

define function LETS_GO {
  # Declaring variable
  string input := "".

  # Asking for input
  print "Caesar Cipher Program".
  input := accept_string_input(print "Would you like to encrypt or decrypt a string (Enter encrypt or decrypt)").
  if(input = "encrypt"){
    # Ask for string and shift amount for it
    string word_to_encrypt := "".
    number shift_amount := 0.

    word_to_encrypt := accept_string_input(print "Please enter string to encrypt: ").
    shift_amount := accept_number_input(print "Please enter a shift amount: ").

    # Call function encrypt
    call encrypt(word_to_encrypt, shift_amount).
  }else{
    # Ask for string and shift amount for it
    string word_to_decrypt := "".
    number shift_amount := 0.

    word_to_decrypt := accept_string_input(print "Please enter string to decrypt: ").
    shift_amount := accept_number_input(print "Please enter a shift amount: ").

    # Call function decrypt
    call decrypt(word_to_decrypt, shift_amount).
  }
  print "End of Caesar Cipher Program".
}

define function encrypt(string word_to_encrypt, number shift_amount) {
  # Variable Declarations
  string encrypted_string := "".
  number i := 0.
  number k := 0.
  string alphabet := "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
  string letter := "".
  number index_of_shifted := 0.

  for(i to length(string_to_encrypt)):
    letter := string_to_encrypt[i].
    if(letter = " "){
      encrypted_string := encrypted_string + letter.
    }else{
      for(k to length(alphabet)):
        if(letter = alphabet[k]){
          index_of_shifted := shift_amount + k.
          if(index_of_shifted ≥ length(alphabet)){
            index_of_shifted := index_of_shifted remainder 26
          }
          letter := alphabet[index_of_shifted].
          encrypted_string := encrypted_string + letter.
        }
        NEXT k.
      }
    }
    NEXT i.
  print "Encrypted String: ", encrypted_string.
}

define function decrypt(string word_to_decrypt, number shift_amount) {
  # Variable Declarations
  string decrypted_string := "".
  number i := 0.
  number k := 0.
  string alphabet := "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
  string letter := "".
  number index_of_shifted := 0.

  for(i to length(string_to_decrypt)):
    letter := string_to_decrypt[i].
    if(letter = " "){
      decrypted_string := decrypted_string + letter.
    }else{
      for(k to length(alphabet)):
        if(letter = alphabet[k]){
          index_of_shifted := k - shift_amount.
          if(index_of_shifted ≤ 1){
            index_of_shifted := index_of_shifted remainder 26
          }
          letter := alphabet[index_of_shifted].
          decrypted_string := decrypted_string + letter.
        }
        NEXT k.
      }
    }
    NEXT i.
  print "Decrypted String: ", decrypted_string.
}

End.
```

Factorial:

```
external class Factorial_Program Start

define function LETS_GO {
  # Variable declaration
  number factorial_num := 0.

  # Accepting Input
  print "Factorial Program".
  factorial_num := accept_number_input(print "Please enter a number to find the factorial of: ").
  call factorial(factorial_num).
}

define function factorial(number factorial_num) {
  # Variable declaration
  number result := 1.
  number original_num = factorial_num.

  # While statement to get us to loop through the whole number
  while(factorial_num ≥ 1):
    result := result * factorial_num.
    factorial_num := factorial_num - 1.

  print "Factorial of ", original_num, " is: ", result.
}
End.
```

Bubble Sort:

```
external class Bubble_Sort_Program Start

define function LETS_GO {
  # Variable Declarations
  Array <|number|> random_numbers [5].

  # These are the 5 numbers we are going sort
  random_numbers[0] := 25.
  random_numbers[1] := 143.
  random_numbers[2] := 2.
  random_numbers[3] := 10.
  random_numbers[4] := 52.

  call bubble_sort(random_numbers).
}

define function bubble_sort(Array random_numbers) {
  # Variable Declarations
  number i := 0.
  number k := 0.
  number temp := 0.
  number j := 0.

  for(i to length(random_numbers) - 1):
    for(k to length(random_numbers) - i - 1):
      if(random_numbers[k] > random_numbers[k+1]){
        temp := random_numbers[k].
        random_numbers[k] := random_numbers[k+1].
        random_numbers[k+1] := random_numbers[temp].
      }
    NEXT k.
  NEXT i.

  # Print out the sorted list now
  for(j to length(random_numbers)):
    print random_numbers[j].
    print " ".
  NEXT j.
}
End.
```

Fibonacci Sequence Recursive Implementation:

```
external class Fibonacci_Program Start

define function LETS_GO {
  # Variable Declarations
  number value := 0.
  number i := 0.
  number result := 0.
  print "Welcome to the Fibonacci Sequence Program".
  value := accept_number_input(print"Enter the number you would like to go up to in the fibonacci sequence: ").

  for(i to value):
    result := call fibonacci_program(number i).
    print "Sequence is: ", result.
  NEXT i.
}

define function number fibonacci(number value) {
  # Define base cases
  if(value = 0){
    return 0.
  }else if(value = 1){
    return 1.
  }

  # Recursive implementation
  if(n > 1){
    return fibonacci(value - 1) + fibonacci(value -2).
  }
}

End.
```

Number Guessing Game:

```
external class Number_Guessing_Game_Program Start

define function LETS_GO {
  print "Welcome to the Number Guessing Game!".
  call number_guessing().
}

define function number_guessing() {
  # Import the random number class
  grab random_number_class.

  # This is calling the random_number_class and grabbing a number from the range we gave it
  number tries = 0.
  number range = 25.
  number random_number := random_number_class.number in range.

  # Take some input
  number guess := 0.
  print "Lets start off by taking a guess."
  guess = accept_number_input(print"Enter your guess between (1-25): ").

  # We are going to check if the initial guess was correct
  if(guess = random_number){
    print "WOW! On the first try! 🎉 Good Job!".
    exit.
  }else{
    # We are going to have them guess until correct
    while(guess ≠ random_number){
      if (guess < random_number){
        print "Your guess is less than the actual number. Try again, you got this! 🐢".
        tries := tries + 1.
        guess := accept_number_input(print"Enter another number: ").
      }else if(guess > random_number){
        print "Your guess is greater than the actual number. Try again, you got this! 🐢".
        tries := tries + 1.
        guess := accept_number_input(print"Enter another number: ").
      }
    }

    print"After ",tries," tries, you finally got the correct number. LETS_GO! 🎉"
    exit.
  }
}

End.
```