23/10/2017

Korgoth Programming Language

(Calculus Problem Solving Language)

Table of Contents

1. Introduction

Korgoth is a basic mathematical programming language, which solves the problem that the user codes. Korgoth is not only simple but also functional and it aims to solve the mathematical problems and identify variety of useful mathematical functions and expressions. The purpose of this language is to ease the solving of complex expressions of basic mathematical problems via computer. For example, when the user will enter the line below

```
show( 'The result is ' + C(4,2) + 2! - sin(30) );
```

Korgoth will identify it as: the combination of 4 chooses 2 plus factorial of 2 subtract the value of sin when it is 30 degrees. Then it will calculate the answer and display it as following by merging the string and the result:

```
The result is 7.5
```

Thus, the user will be able to get the answer easily.

2. Truth Values

Truth values in Korgoth are defined with `boolean` and can have a value of `true` or `false`. We decided to stick to the original truth values as in Java or C++ etc. Since the programming software is used to `boolean`, it would be easier to read and write.

3. Constants

Since Korgoth is a basic calculus problem solver, it has two main constants with defined values. They are pi and natural e. We decided to use the notations `PI` and `e` to make it readable.

4. Connectives

Korgoth has almost the same notations for the connectives as in any other programming language. However, there are also some differences. For example, in Korgoth ‘and’ is literally written as `and`, ‘or’ is literally written as `or`, as following

```
a and b
```

```
a or b
```

From our points of view, it is more readable and writable than original && and ||.

Another difference is the notation of ‘not’. We decided to use the notation of ‘~’, like in the hardware programming languages, ‘not’ is declared with tilde. The usage of tilde is as following,

`a ~= b` (a is not equal to b)

4.1 Basic Characters

The basic characters as addition, multiplication etc. are declared with same characters as in any other programming language. We did not see any advantage in changing them, because they are already obvious, readable and writable.

4.2 Comment Declaration

We have changed the declaration of the comments in Korgoth. For example, in java comments are defined with ‘//’ or ‘/*…*/’ if it has multiple lines. In Korgoth, we decided to use ‘”””’ notation, since comments seem to be the explanations or additions of the programmer, thus the additional ideas could be written in quotes as following

```
“This method returns the subtraction of two numbers”
```

The comments can also be multiple lined; in that case, the same notation will be used. Thus the rule for the comment is, if you open the comment with “”, then close it with “”. Until the closing “”, it will be defined as a comment. For example,

```
“This method
returns the subtraction of
two numbers”
```

5. Variables

Since Korgoth is going to be a dynamically typed programming language, variable types are not indicated in the code. The declaration of the variables will be as following

```
integerNumber = 1;
floatNumber   = 2.0123;
name          = ‘Mehin’;
isRunning     = true;
```

In our opinion, it would be more writable for the programmers to write a code without thinking about int, string or any other data types. At the same time, it would be more readable for the users, since they would not think about what string or float means.

6. Predicates – Class and Method Declarations

Since Korgoth has only one aim which is to calculate the value. Therefore, there are no class declarations. The execution always starts with the main function called `doMath`.

Declaration of the fucntion will be as following

```
doMath[
“expressions” ]
```

Since there are no pre-defined variables in Korgoth Programming Language, methods does not require the identifications as `public, private` or `int, double, void` but only `function` identifier. Below, there is an example method which returns boolean.

Let us say, we want to create a new `’Sum’` method that takes two inputs called `number1` and `number2`. Sums them, and returns true if the sum is calculated.

```
function sum( number1, number2)[
answer = number1 + number2;
calculated = true;
return calculated;
]
```

We decided to use “[ ]” notations for opening and ending, rather than “{ }” notations for two reasons. First, we already use “{ }” notations for defining sets in Korgoth and

secondly, we thought it would be unique for our language. From the user's point of view, nothing would have differ, because both notations are very similar.

7. Mathematical Expressions and Functions

As it was mentioned above, Korgoth is the basic calculus problem solver. Thus, it should have the default functions for making the user's life easier. For example, consider that you have to create a variable that will be a value of sine when it is 66 degrees. It would be very difficult to create such a variable in one line in any programming language. Usually, for creating such a variable and getting the value, the class needs to call at least one method and execute the answer. However, in Korgoth it can be implemented in one line as following,

`a = sin(66);` (a will be equal to the value of sine when it is 66 degrees).

We think, this is very writable and readable, since in real life we write it the same.

In Korgoth there are also arrays, but they are called sets and are defined with {} notations. Once the array is created, they are very easily unified. As an example,

```
A = {1, 2, 3};
B = {4, 5, 6};
C = A U B; "It means C = {1, 2, 3, 4, 5, 6}
```

Sets in Korgoth Language can also contain other variable types such as string or float. For example,

```
A = {1, 2.5, 3.66, 'a'};
B = {4, 5.75, 'Example'};
```

Normally, we would had to use the merge method to merge two arrays in other languages. However, in Korgoth the notation ‘U’ as in real math notations, easily connects two sets. Similarly notation ‘n’ intersects them.

```
A = {1, 3, 5, 6};

SETB = {3, 5};

SETC = A n SETB; “SETC = {3, 5}”
```

In addition, it also has the vector idea which is defined with <> notations as following,

```
v1 = <-3, 3.5>;

v2 = v1 x <-1, 12>;
```

Note: The set names are always uppercase strings, whereas vector names are declared with v + any number. In addition, our set corresponds to the array.

8. Selection

The ‘if’ statements are written and work in the same logic as in any other programming language.

Examples:

```
if(a == b) [
     show(‘The values are equal’);
]
else if(a ~= b) [
     show(‘The values are not equal’);
]
else [
     show(‘Undefined’);
```

```
]
```

9. Looping Statements

Looping statements work in the same logic as in any other programming language.

For example:

```
while(a ~= b)[
    a = a + 1; ]
for(a = 0; a < 10; a = a + 1)[
    show('Hello World!'); ]
```

10. Input/Output Statements

There are both input and output methods in Korgoth. In order to get the input from the user, `take()` method is used. For outputting anything, `show()` method is used as following,

```
take(number);
show('Hello World!');
show(b); "b is an integer value"
```

take() method corresponds to `cin`, show() method corresponds to `cout` method of C++. We think, take and show method names could be easily written and read.

11. Reserved Words

Korgoth has two main reserved words, which are return and function. `return` is used to return any definition, variable or expression out of function and `function` is reserved word for declaring an operation. For example:

```
function getIncrement(number) [
      return number++;
```

]

12. Complete BNF of Korgoth

12.1 Declarations

<method declaration> ::= <method header><method body>

<method header> ::= <method declarator>

<method declarator> ::= <function identifier> <identifier>(<formal parameter list>?)

<function identifier> ::= function

<method body> ::= [<method expressions>?<return statement>?]

12.2 Types

<boolean literal> ::= true | false

<set type> ::= {<set body>};

<set body> ::= <token>+,?

<vector type> ::= < <vector body> >;

<vector body> ::= <token>+,?

12.3 Constant Variables

<constant variables> ::= <pi expression> | <natural e>

<pi expression> ::= <pi>

<pi> ::= pi

pi ::= 3.141592653589

<natural e expression> ::= e

e ::= 2.718281828459

12.4 Connective and Arithmetical expressions

<connective expressions> ::= <assignment expression> | <and expression> |

<or expression> | <equality expression> | <not equality expression>

<assignment expression> ::= =

<not expression> ::= ~

<and expression> ::= and

<or expression> ::= or

<equality expression> ::= ==

<not equality expression> ::= ~=

<less expression> ::= <

<greater expression> ::= >

<less and equal expression> ::= <=

<greater and equal expression> ::= >=

<left parenthesis expression> ::= \(

<right parenthesis expression> ::= \)

<left curly brace expression> ::= \{

<right curly brace expression> ::= \}

<comma expression> ::= ,

<semicolon expression> ::= ;

<addition declaration> ::= \+

<subtraction declaration> ::= \-

<multiplication declaration> ::= *

<division declaration> ::= \/

<modulus declaration> ::= \%

<exponent declaration> ::= \^

<factorial declaration> ::= !

<union declaration> ::= U

<intersection declaration> ::= n

<absolute expression> ::= \|

<increment> ::= \+\+

<decrement> ::= \-\-

12.5 Mathematical Default Functions

<combination declaration> ::= <combination identifier>([<constant variables><digit><float>]+,[<constant variables><digit><float>]+)

<combination identifier> ::= C

<permutation declaration> ::= <permutation identifier>([<constant variables><digit><float>]+,[<constant variables><digit><float>]+)

<permutation identifier> ::= P

<sin function declaration> ::= <sin identifier>(<sign>?[<constant variables><digit><float>]+)

<sin identifier> ::= sin

<cos function declaration> ::= <cos identifier>(<sign>?[<constant variables><digit><float>]+)

<cos identifier> ::= cos

<tan function declaration> ::= <tan identifier>(<sign>?[<constant variables><digit><float>]+)

<tan identifier> ::= tan

<cot function declaration> ::= <cot identifier>(<sign>?[<constant variables><digit><float>]+)

<cot identifier> ::= cot

<sec function declaration> ::= <sec identifier>(<sign>?[<constant variables><digit><float>]+)

<sec identifier> ::= sec

<csc function declaration> ::= <csc identifier>(<sign>?[<constant variables><digit><float>]+)

<csc identifier> ::= csc

<arcsin function declaration> ::= <arcsin identifier>(<sign>?[<constant variables><digit><float>]+)

<arcsin identifier> ::= arcsin

<arccos function declaration> ::= <arccos identifier>(<sign>?[<constant variables><digit><float>]+)

<arccos identifier> ::= arccos

<arctan function declaration> ::= <arctan identifier>(<sign>?[<constant variables><digit><float>]+)

<arctan identifier> ::= arctan

<arccot function declaration> ::= <arccot identifier>(<sign>?[<constant variables><digit><float>]+)

<arccot identifier> ::= arccot

<min function declaration> ::= <min identifier>(<sign>?[<constant variables><digit><float>]+, <sign>?[<constant variables><digit><float>]+)

<min identifier> ::= min

<max function declaration> ::= <max identifier>(<sign>?[<constant variables><digit><float>]+, <sign>?[<constant variables><digit><float>]+)

<max identifier> ::= max

<square root declaration> ::= <square root identification>([<constant variables><digit><float>]+)

<square root identification> ::= sqrt

<nth root declaration> ::= <nth root identification>([< constant variables ><digit><float>]+, [<constant variables ><digit><float>]+)

<nth root identification> ::= nrt

<logarithm default 10^{th} root declaration> ::= <logarithm identification>([< constant variables ><digit>]+?, [<constant variables ><digit><float>]+)

<logarithm identification> ::= log

<logarithm from e^{th} root declaration> ::= <logarithm identification>([<constant variables><digit><float>]+)

<ln identification> ::= ln

12.6 Tokens

<tokens> ::= <digit> | <float> | <identifier> |

<digit> ::= 0 | <non zero digit>

<non zero digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<float> ::= <digit>+(\).?<digit>+

<sign> ::= + | -

<identifier> ::= [a-zA-Z0-9]

11.7 Loops

<if statement> ::= if (<expression>)<statement>

<if then else statement> ::= if(<expression) <statement> else <statement>

<if then else if statement> ::= if(<expression) <statement> (else if)* <statement> else?<statement>

<statement> ::= <while statement> | <for statement> | <do while statement>

<while statement> ::= while(<expression>)<statement>

<do while statement>::= do<statement>while(<expression>).

<for statement> ::= for(<for init>; <expression>; <for update>) <statement>