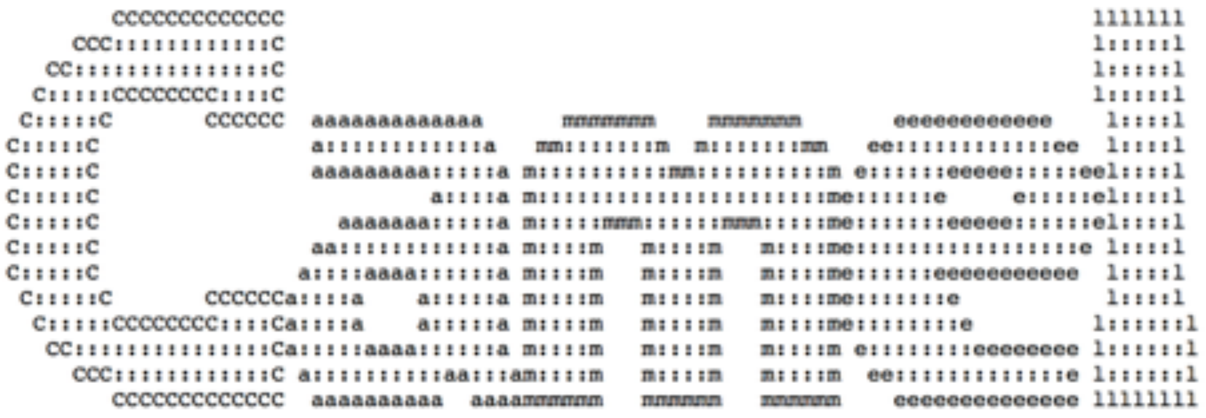

Camel Language

WHAT IS CAMEL LANGUAGE?

Camel is an interpreted language that was build using FLEX, Bison and the C language. And by interpreted we mean that it runs totally in runtime, allocating and deallocating memory space on the go with all its variables of generic type.

Camel is designed to be similar to Python in how easy its syntax is, and how powerful it can be by allowing the programmer to execute complex algorithms with the minimum number of lines and commands needed.

LANGUAGE SYNTAX

Literals

Camel Language supports the following types:

- Integers: such as

  ```
  3 , -4
  ```

- Real Numbers:

  ```
  3.5 , -0.12
  ```

- Strings:

  ```
   “Hello world!”
  ```

- Variables:

  Variables in Camel Language are dynamic, they can be of any type and change their type on assignment. As in the example:

  ```
  def a = 3;
  print a;
  a = 3.5;
  print a;
  a = “Hello world!”
  print a;
  ```

  The output would be “

  ```
  3
  3.5
  Hello world!
  ```

  “

  Here variable a was first defined with the integer value 3. Then it was reassigned with the real number 3.5. After that it was reassigned with the string “Hello world!”.

Operators

To see operators with strings, please check the strings section.

1. Unary operators

Unary operators are operators that are prefixed to an expression. Camel Language only supports the following prefix operators:

-operand;

This causes the value of the operant to be negated (multiplied by -1). String operands can't be negated.

Examples:

```
def joe = 5;

print -joe;
```

This outputs the number -5;

2. Normal operators

There are several normal operators which return the result defined for each:

expression 1 **+** *expression 2*
The result of this is the sum of the two expressions.

expression 1 **-** *expression 2*
The result of this is the value of *expression2* subtracted from *expression1*.

expression 1 ***** *expression 2*
The result of this is the multiplication of the two expressions.

expression 1 **/** *expression 2*
The result of this is the result of dividing *expression 1* by *expression 2*.

expression 1 ***mod*** *expression 2*
The result of this is the remainder of dividing *expression 1* by *expression 2*.

expression 1 **^** *expression 2*
The result of this is the exponent of *expression 1* to the power *expression 2*.

Examples:

```
print 34 - 30;
```

This outputs the number 4;

```
print 2 ^ 6;
```

This outputs the number 64;

```
print 3 mod 2;
```

This outputs the number 1;

3. Boolean operators

Boolean operators are operators that return either 1 (true) or 0 (false). Camel Language supports the following boolean operators:

expression 1 **<** *expression 2*
returns 1 (true) if *expression 1* is less than *expression 2*, else returns 0 (false).

expression 1 **>** *expression 2*
returns 1 (true) if *expression 1* is greater than *expression 2*, else returns 0 (false).

expression 1 **<=** *expression 2*
returns 1 (true) if *expression 1* is less than or equal to *expression 2*, else returns 0 (false).

expression 1 **>=** *expression 2*
returns 1 (true) if *expression 1* is greater than or equal to *expression 2*, else returns 0 (false).

expression 1 **==** *expression 2*
returns 1 (true) if *expression 1* is equal to *expression 2*, else returns 0 (false).

expression 1 **<>** *expression 2*
returns 1 (true) if *expression 1* is not equal to *expression 2*, else returns 0 (false).

Examples:

```
print 34 <> 30;
```

This outputs 1 (true) because 34 does **not** equal 30 ;

```
print 34 <= 30;
```

This outputs 0 (false) because 34 is not less than or equal (it is bigger than) 30 ;

Strings

Strings in Camel Language are similar to strings in the C language. Starting with a double quotation mark and ending with a double quotation mark, such as in the example below:

```
“Hi! I am a string. Pleased to meet you. :-)”
```

1. Escape sequences

The following escape sequences allow for special characters to be put in strings and in the source code:

Escape Sequence	Name	Meaning
`\n`	New Line	Moves the cursor to the first position of the next line.
`\t`	Horizontal Tab	Moves the cursor to the next horizontal tabular position.
`\"`		Produces a double quote.

2. Operators

Some operators have special behaviour when used with strings, such as the following:

string **+** *string* ***or*** *string* **+** *expression* ***or*** *expression* **+** *string*
returns the concatenation of the string with the other operand.

string ***** *integer expression* ***or*** *integer expression * string*
repeat the string (integer expression) times.

string 1 **<** *string 2*
returns 1 (true) if *string 1* has less characters than *string 2*, else returns 0 (false).

string 1 **>** *string 2*
returns 1 (true) if *string 1* has more characters than *string 2*, else returns 0 (false).

string 1 **<=** *string 2*
returns 1 (true) if *string 1* has less or equal characters than *string 2*, else returns 0 (false).

string 1 **>=** *string 2*
returns 1 (true) if *string 1* has more or equal characters than *string 2*, else returns 0 (false).

string 1 **==** *string 2*
returns 1 (true) if *string 1* is the same as *string 2*, else returns 0 (false).

string 1 **<>** *string 2*
returns 1 (true) if *string 1*is **not** the same as *string 2*, else returns 0 (false).

Statements

The following are statements found in the current version of Camel Language:

1. Declarations

```
def variable_name [, variable_name2 . . .];
```

Defines a variable in the memory with the name variable_name. Multiples variable can also be defined on the same line such as:

```
def bart, joe, sara;
```

Values can be given to variables on declaration, such as:

```
def bart, joe = 5, sara;
```

Here variables bart and sara were created with no value, and variable joe was declared with integer value 5.

2. Control statements

Statements that change the flow of statements .

- **Conditional statements.**

```
if ( expression1 ) statement1 [ else statement2 ]
```

An if statement is a statement that executes statement 1 if expression1 was true (1).
The [else statement2] part is optional and it specifies what statement to do if expression1 was false (0).

- **Loops**

```
while ( expression1 ) statement1
```

A while loop is a loop that executes statement1 as long as expression1 is true.
It checks expression1 before executing statement1.

```
loop integer literal : statement1
```

A ‘loop’ loop is a loop that executes statement1 n times as long s the integer literal is positive. The integer literal can also be a variable of integer value.

3. Print statements

```
print expression1;
```

Prints the expression on a new line.

4. Read statements

```
read_i variable1;
```

Reads an integer value and puts it into variable1.

```
read_r variable1;
```

Reads real value (float) and puts it into variable1.

```
read_s variable1;
```

Reads a string and puts it into variable1.

5. Assigning statements

```
variable1 = variable2;
```

Puts the value of variable2 in variable1, and changes the type of variable1 accordingly.

```
variable1 = expression1;
```

Puts the value of expression1 into variable1, and changes the type of variable1 accordingly.