User Manual: KorLang

hello

1. Introduction

As of the current version of KorLang, there is no custom interface for coding, debugging, and running source codes written in this language.

1.1. Programs coded using this language must be saved with the `.kl` file extension.

1.2. Prerequisites (programs needed to use this language): Python compiler (3.6.4rc1 version), PLY, korLang.py file

1.3. Steps for compilation and execution:

Install the latest version of Python (3.6.4rc1)

Install PLY

Download the `korLang.py` file

Once you are done coding, save the file with the `.kl` extension

(e.g) `helloworld.kl`

Place the `korLang.py` file and the `.kl` file in one folder

Open the terminal, then go to the directory where you placed the files

Type `python korLang.py`, then press enter

This will appear:

```
Enter filename (include .kl extension):
```

Enter the filename, in this example type `helloworld.kl`

2. Lexical Analysis

2.1. Line Structure

A program coded in KorLang language is divided into a number of logical lines.

2.1.1. Logical Lines

The end of a logical line is represented by the token TILDE (~). A logical line is constructed from one or more physical lines/code lines.

2.1.2. Comments

A comment starts with a dollar sign character ($) that is not part of a string literal, and ends at the end of a physical/code line. Comments are ignored by the syntax, they are also not treated as tokens. As of the current version of KorLang, multi-line comments are not allowed.

2.1.3. Blank Lines

A logical line that contains only spaces, tabs, and a comment, is ignored.

2.1.4. Indentation

Until python, leading whitespace (spaces and tabs) at the beginning of a logical line is not used to determine the grouping of statements. In fact, leading whitespace would not affect your code.

2.2. Keywords

2.2.1. The following words are used are reserved words or keywords of the KorLang language, and cannot be used as ordinary identifiers / names for variables. They must be spelled exactly as written:

```
if          else        while
true        false       printk
scank
```

2.3. Operators

2.3.1. The following tokens are operators:

```
+           -           *
/           ^           >
<           >=          <=
==          !=          ||
%%
```

2.4. Delimiters

2.4.1. The following tokens serve as delimiters in the KorLang language grammar:

```
(           )           [
]           {           }
,           .
```

3. Syntax Documentation

3.1. Built-in Types

3.1.1. Numeric Types

The two distinct numeric types are *integers* and *floating point* numbers. Integers have unlimited precision. Arithmetic operations may only be executed if numeric literals are of the same type. Otherwise, a type error will occur.

Integer declaration

```
variablename.oppa
```

Float declaration

```
variablename.hyung
```

3.1.1.1. Numeric Operations

These are the numeric operations supported by KorLang, in order of descending precedence.

Operation	Result	Associativity
-a	a is negated	
a ^ b	a to the power b	right
a % b	remainder of a/b	right
a * b	product of a and b	left
a / b	quotient of a and b	left
a + b	sum of a and b	left
a - b	difference of a and b	left

3.1.2. **Boolean**

Boolean literals are declared using the format:

```
variablename.noona
```

These literals may only be assigned a value of either *true* or *false*.

3.1.2.1. Boolean Operations

These are the boolean operations supported by KorLang, in order of descending priority. Boolean operations may also be used for integer and floating point types.

Operation	Result
a && b	if a is false then a, else b
a || b	if a is false then b, else a

3.1.2.2. Comparisons

Comparison operations have the same priority (higher than boolean operations). A type error will also occur when comparing objects of different types.

Operation	Meaning
<	strictly less than
>	strictly greater than
<=	less than or equal to

>=	greater than or equal to
==	equal
!=	not equal

3.1.3. **Strings**

String literals are declared using the format:

```
variablename.unnie
```

KorLang currently does not support any string methods.

3.1.4. **Arrays**

An array is a mutable collection of objects with the same data type. Arrays in KorLang start at index 0 and are dynamically allocated.

3.1.4.1. Types

To declare an array, we use brackets and include what elements we want to put in the array. We may also initialize an empty one ([ ]).

Integer array declaration

```
arrayname.nim
```

Float array declaration

```
arrayname.ah
```

String array declaration

```
arrayname.ssi
```

3.1.4.2. Array Operations

- access element

  `array[index number]` - returns element at given index
- add an element to an array

  `array + element`
- delete element from an array

  `array.delete(index number)`
- replace element

  `array[index number] = new element`
- get index

  `array.index(element)`

  returns index of given element
- get length

  `array.len()`

  returns number of elements in array

- concatenate two arrays

  `array + array`

  returns a new array

3.2. Built-in Constants

Objects (expressions, etc.) may be tested for their truth values to be used in *if* and *while* conditions as well as for Boolean operations.

True - returns 1

False - returns 0

3.3. Built-in Functions

3.3.1. Standard I/O

`printk(object 1, object 2, ... , object 3)`

outputs objects to the terminal

`scank()`

gets user input one at a time

3.3.2. Conditionals

One condition

If `statement` holds true then `body 1` will execute, else `body 2` will execute.

```
if(statement) { body 1 }~
else { body 2 }~
```

Multiple conditions

If we have n conditions, KorLang will check if `statement n` is true and execute `body n`. If it is false, it will proceed to `statement n+1` and execute `body n+1`. The default body to be executed will be what's in the else statement.

```
if(statement 1) { body 1 }~
else if(statement 2) { body 2 }~
else if(statement 3) { body 3 }~
else if(statement n) { body n }~
else { body }~
```

3.3.3. Iteratives

KorLang uses a precondition loop for its iterative statements. The `body` will execute until `statement` becomes false.

```
while (statement) {
```

```
        body
}~
```

4. Error Handling

4.1. Arithmetic Error

Raised when the second argument of a division or modulo operation is zero.

4.2. EOF Error

Raised when the input file contains no data, i.e. the interpreter hits an end-of-file condition without reading data.

4.3. Index Error

Raised when trying to access or delete an element from an array in which the index used is greater than or less than the array size.

4.4. Lookup Error

Raised when an undefined variable is used in an assignment statement or used in array operations.

4.5. Syntax Error

Raised when the parser encounters a syntax error. This includes missing a symbol from the syntax or when variables are not declared in the right form (illegal name).

4.6. Type Error

Raised when a function is applied on an object type that is not supported by KorLang. This includes printing objects that are not integers, floating point numbers, strings, boolean values, newlines, or arrays. It also occurs when we create an array, conditional statement, loop condition, boolean operation, arithmetic operation, input values with incompatible types. Note that variables should always have their corresponding `.suffix` to avoid this error.

4.7. Value Error

Raised when assigning an inappropriate value to a variable of a certain type and using inappropriate values in arithmetic operations and in getting the index of an element..