# Kraken Programming Guide

Jack SPARROW

November 26, 2015

## 1 Compiling

Kraken compilation currently only supports building the compiler from source. You can clone the repository from a terminal using:

```
git clone https://github.com/Limvot/kraken.git
```

Once you have the repository, run the following commands:

```
mkdir build %Create a build directory
cd build
cmake ..    %Requires cmake to build the compiler
make        %Create the compiler executable
```

This will create a kraken executable, which is how we will call the compiler. Kraken supports several ways of invoking the compiler. These include:

```
kraken source.krak
kraken source.krak outputExe
kraken grammarFile.kgm source.krak outputExe
```

The grammar file is a file specific to the compiler, and should be included in the github repository. When you run the compile command, a new directory with the name of the outputExe you specified will be created. In this directory is a shell script, which will compile the created C file into a binary executable. This binary exectuable can then be run as a normal C executable.

## 2 Variables

Kraken has automatic type deduction. This is sort of like the duck typing of Python. The difference is that variables cannot change types. In this way, it is much more like an implicit ”auto” keyword in C++. Unlike C++, semicolons are optional after declarations.

### 2.1 Variable Declaration

```
var A: int;          //A is unitialized int
var B = 1;           //B is integer
var C = 2.0;         //C is double
var D: float = 3.14 //D is double
```

### 2.2 Primitive Types

The primitive types found in kraken are:

a. int

b. float

c. double

d. char

e. bool

f. void

## 3 Functions

```
fun FunctionName(arg1 : arg1_type, arg2 : arg2_type) : returnType {
  var result = arg1 + arg2;
  return result;
}
```

Functions are declared using the **fun** keyword. If you pass in variables as shown, there will be passed by value, not by reference. Therefore if you pass a variable in, it will not be modified outside the function.

## 4 Input and Output

In order to print to a terminal or file, the **io** library must be imported. There are a few different functions you can use to print to the terminal. The print() function will print out to the terminal without a newline character. Like java, there is a println() function that will print whatever you pass in, as well as a newline. There are also functions that can print colors in a unix terminal. The color will continue when you print until you call the function Reset().

a. BoldRed()

b. BoldGreen()

c. BoldYellow()

d. BoldBlue()

e. BoldMagneta()

f. BoldCyan()

```
io::print(3.2); //print without a newline character
io::println(varA); //print variable A with a newline character
io::BoldRed();
io::println("This line is printed Red");
io::Reset();
io::println("This line is printed black");
```

You can also use kraken to read and write to files. The functions are as follows:

```
//returns true if file exists
var ifExists = io::file_exists("/usr/bin/clang");

//read file into string
var fileString =  io::read_file("~/Documents/file.txt");

//write a string to the file
io::write_file("/",SteamString);

//read file into vector of chars
var charVec = io::read_file_binary("~/Documents/file2.txt");

//write a vector of chars to a file
io::write_file_binary("/",md5checkSum);
```

# 5 Memory Management

## 5.1 Pointers

Pointers in kraken work like they do in C. The notation is the * symbol. This is a dereference operator. This means that it operates on a pointer, and gives the variable pointed to. For instance:

```
var B: *int = 4;  //B is a pointer to an integer 4
*B = 3;           //B is now equal to 3
print(B);         //would print an address, like "0xFFA3"
```

## 5.2 References

References are a way to create "automatic" pointers. If a function takes in a reference, the variable is passed by reference, instead of by value. This means that no copy of the variable is made, and any changes made to the variable in the function will remain after the end of the function. References also allow left-handed assignment. This means that an array indexed on the left hand of an equal sign can have its value changed.

```
fun RefFunction(arg1: ref int): ref int{
  return arg1 + 1;
}

var a = 6;
var b = RefFunction(a);
println(a); //a is now equal to 6
println(b); //b is now equal to 6
RefFunction(b) = 15;
println(b); //b is now equal to 15
```

### 5.3 Dynamic Memory Allocation

In order to allocate memory on the heap instead of the stack, dynamic memory allocation must be used. The data must be explicitly allocated with the **new** keyword, and deleted with the **delete** keyword. The size in both instances must be provided.

```
var data = new<int>(8); //Allocate 8 integers on the heap
delete(data,8);         //Free the memory when its no longer used.
```

## 6 Classes

### 6.1 Constructors

### 6.2 Operator Overloading

### 6.3 Inheritance

## 7 Templates

Section T

# 8 Standard Library

## 8.1 Import Statements

## 8.2 Vector

## 8.3 String

## 8.4 Regex

## 8.5 Util

## 8.6 Data Structures

### 8.6.1 Stack

### 8.6.2 Queue

### 8.6.3 Set

### 8.6.4 Map

# 9 Understanding Kraken Errors

Section error

# 10 C Passthrough