Kona

Language Summary and Example Programs

Version 0.2.1

Robert Perrone

This document is part example of, and part instructions for a new programming language called Kona, names after the dog pictured above. This is a language project for the class Theory of Programming Languages at Marist College.

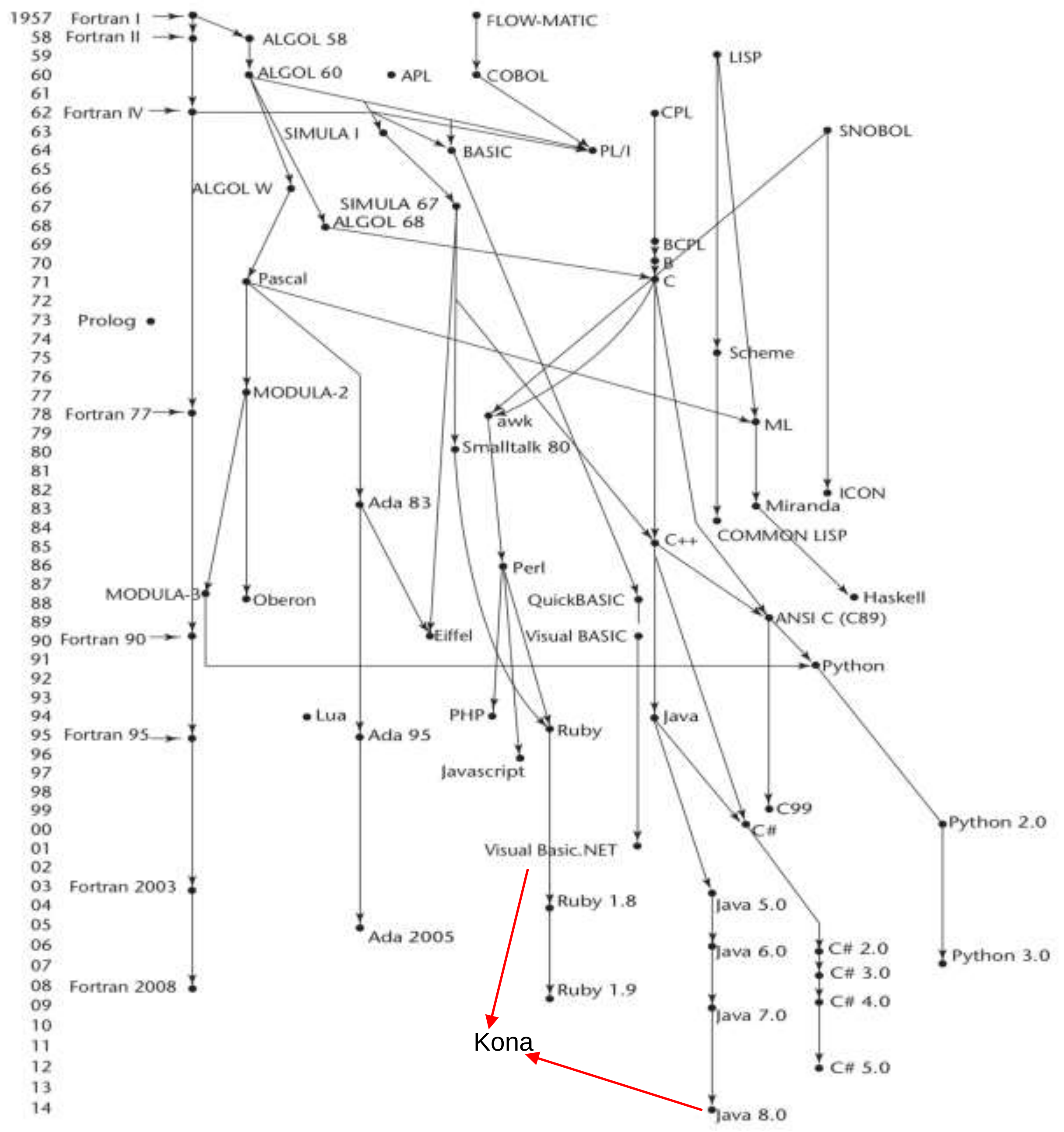
Introduction

Kona (pronounced “KOH – nuh”) is a procedural, object-oriented, and (strongly) type-safe programming language. Based on Visual BASIC and Java, but differing in the following ways:

1. While this language is mainly procedural, there will be some functional aspects such as mapping, which was added in some later versions in Java, although the syntax here is different.
2. Kona is case-sensitive, and keywords are intentionally upper or lower -case.
3. The syntax is mostly a mesh between the two languages and then there are some new functions and keywords that are not in either. Such as:
    a. `bark` to print
    b. `speak` to get user input
    c. `fetch` instead of return
    d. `elif` instead of else if
    e. `Park` and `Home` are used as access modifiers instead of Public and Private, respectively
    f. `.len` instead of len() or .length()
    g. `.UpCase` and `.LowCase` instead of .toUpperCase() or UCase() and .toLowerCase() or LCase()
    h. `Str`, `chr`, and `bool` for respective type declarations. `ball` is in place of the object type
        i. Type declarations are all lowercase
        ii. Type casting is the same keyword, but uppercase. Used like Str(), Chr(), Int(), etc.
    i. `stack` types can use `push` and `pop` built-in functions
    j. `queue` types can use `enqueue` and `dequeue` built-in functions
    k. `goodDog` and `badDog` replace true and false

1.1 Genealogy

Where does your language fit into the programming language genealogy? Add your language to this diagram to highlight your language and its ancestry.



1.2 Hello world

You are morally obligated to write the "Hello World" program in your language here.

```
Class HelloWorld {
    Sub main() {
        //bark is the new system.out.println(), and Console.WriteLn()
        bark("Hello World!\n")
        //adding new line after is done manually (not already in the function)
    }
}
```

1.3 Program structure

The key organizational concepts in Kona are as follows:

1. There must be a class declared to be able to run, with the filename being the same name as the class.
2. Declaring `Park` or `Home` access modifiers is not required, but rather strongly recommended.
3. Sub does not have a return type, while Function must have a return type.
4. Semicolons are not required to end statements, so statements are separated by new lines and { }.

```
Adopt kona.Collections
Park Class Stack {//Make own stack, tho Kona has a built in "stack" type declaration
    Park Entry top
    Park Sub push(data As ball) {
        top = new entry(top, data)
    }
    Park Function ball pop() { //ball (obj) is return type
        if (top == null) {
            throw null
        } else {
            ball result = top.data
            top = top.next
            fetch result //return ball (obj) result
        }
    }
    Park Class Entry {
        Park Entry next
        Park ball data
        Park Sub entry(next As Entry, data As ball) {
            this.next = next
            this.data = data
        }
    }
}
```

This is an example where a class named ***Stack*** has imported a package called ***kona.Collections*** using the keyword ***Adopt***. The fully qualified name of this class is ***kona.Collections.Stack***. The class contains several members: a field named top, two methods named ***Push*** and ***Pop***, and a nested class named ***Entry***. The ***Entry*** class further contains three members: a field named next, a field named data, and a constructor.

1.4 Types and variables

There are two kinds of types in Kona: ***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. See Section 3 for details.

1.5 Statements Differing from Visual BASIC and Java

For space purposes, assume all of the examples are preceded by `Park Class Main() {` *and followed by* `}`.

Statement	**Example**

Expression statement, print statement,
Declare string, character, int, float, and boolean.
	`Sub Main() {`
	`    int i`
	`    i = 123`
	`    chr c = 'a'`
	`    str s = "hello"`
	`    float f = 1.0`
	`    bool b = true`
	`    bark(i + " " + c + " " + s) //print statement`
	`    i++`
	`    bark(i + " " + f + " " + b) //concatenation`
	`}`

`if` statement	`Sub Main(args As str[]) {`
	`    if (args.len == 0) {`
	`        bark("No arguments")`
	`    } elif (args.len == 1) {`
	`        bark("There is one argument.")`
	`    } else {`
	`        bark("There is more than one argument.")`
	`    }`
	`}`

`switch` statements	`Sub Main(expression As str[]) ) {`
	`    select (expression) {`
	`        case 1:`
	`            //code block`
	`            bark("1")`
	`            break`
	`        case 2:`
	`            //code block`
	`            bark ("2")`
	`            break`
	`        case else:`
	`            //code block`
	`            bark("Else")`

```
    }
}
```

`for` loop

```
Sub Main() {
    for (int i = 0 until 10 by 1) {
        bark(i)
    }
}
```

`while` loop
AND
Array Declaration
type[*len*] var = {x, y, z}
//can be used with any type

```
Sub Main(args As str[]) {
    str s = "kona the pup"
    chr[s.len] chrArr = {} //empty array where s.len is the size of the character array
    int j = 0
    while (j < s.len) {
        chrArr[j] = chr[j]
        j++
    }
    bark(chrArr)
}
```

`speak` statement

```
Sub Main(args As str[]) {
    str input = speak("What is your name?")
    bark("Hi " + input + "!")
}
```

Declaring a `Stack` and using `push` and `pop`.

Declaring a `Queue` and using `enqueue` and `dequeue`.

```
Sub Main(args As str[]) {
    stack myStack //Declare a stack
    myStack.push("Woof!")
    myStack.push("Bark!")
    myStack.push("Grrrrrrr")
    myStack.pop()
    queue myQueue //Declare Queue
    myQueue.enqueue(1)
    myQueue.enqueue(2)
    myQueue.enqueue(4)
    myQueue.enqueue(8)
    myQueue.dequeue()
}
```

`map` function

```
Sub Main(args As str[]) {
     bark(map(fun(n -> n + 1) to [1,2,3,4]))
     //map function to list
}
```

2. Lexical structure

2.1 Programs

A Kona ***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 Kona programming language where it differs from Visual BASIC and Java.

2.2.1 Lexical grammar where different from Visual BASIC and Java

Write your BNF grammar productions here.

<Assignment Operator>	→	=
<Mathematical Operator>	→	+ | * | / | - | % | += | -= | *= | /=
<Comparison Operator>	→	== | != | < | > | <= | >=
<Logical Operator>	→	&& | || |
<Keyword>	→	<Language Defined>
	→	<Variable Defined>
<Begin Block>	→	{
<End Block>	→	}
<Alphabet>	→	a | b | c | … | x | y | z
	→	A | B | C | … | X | Y | Z
<Number>	→	0 | 1 | 2 | … | 7 | 8 | 9
<Single-line Comment>	→	//
<Delimited Comment>	→	/* … */

2.2.2 Syntactic ("parse") grammar where different from Visual BASIC and Java

Write your BNF grammar productions here.

<Class Declaration>	$\rightarrow$	<Access Modifier> Class <Identifier>
<Sub Declaration>	$\rightarrow$	<Access Modifier> Sub <Identifier> <Parameter List>
<Function Declaration>	$\rightarrow$	<Access Modifier> <Return Type> Function <Identifier> <Parameter List>
<Parameter List>	$\rightarrow$	<Parameter> <Parameter List>
	$\rightarrow$	<Parameter>
<Parameter>	$\rightarrow$	<Identifier> As <Data Type>

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 characters `//` 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 do not nest.

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
    integer-literal
    real-literal
    character-literal
    string-literal
    operator-or-punctuator
```

2.4.1 Keywords different from Visual BASIC or Java

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:	*Removed keywords:*
Bark	*System.out.println()/Console.WriteLine()*
Speak	*Input*
Park	*Public*
Home	*Private*
Elif	
Fetch	*return*
until (in the context of a for loop)	
Adopt	*Import*
str	*String*
chr	*char*
bool	*boolean*
ball	*Object*
stack	
push	
pop	
queue	
enqueue	
dequeue	
.len	*.length/len()*
compareStr()	*.compareTo()*
.UpCase	*.toUpper()/UCase()*
.LowCase	*.toLower()/ LCase()*
	do
	goto
	default

3. Type System

Kona uses a **strong static** type system. (*Be sure to explain their details and document them with Type Inference diagrams.*) Strong typing means that type errors are caught and expressed to the programmer during compilation. Static typing means early binding compile-time type checking.

3.1 Type Rules

The type rules for Kona are as follows:

Assignment with Scope Context: $\dfrac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 = e_2 : T}$	**Comparisons with Scope:** $\dfrac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 == e_2 : bool}$	$\dfrac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 != e_2 : bool}$
$\dfrac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 > e_2 : bool}$	$\dfrac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 < e_2 : bool}$	**Addition with Scope Context:** $\dfrac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 + e_2 : T}$
String Literals: $\dfrac{S \vdash e_1 : str \quad S \vdash e_2 : str}{S \vdash e_1 \bullet e_2 : str}$	**Integer Literals:** $\dfrac{S \vdash e_1 : int \quad S \vdash e_2 : int}{S \vdash e_1 + e_2 : int}$	**Key:** *S* is the Scope $\vdash$ is a turnstile (right tack) e_1/ e_2 are Expressions 1 & 2

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

3.2 Value types (different from Visual BASIC and Java)

bool – can have the value of either goodDog or badDog, which represents true and false.

Ex: bool behavior = goodBoy

char – is a Unicode value, and can also be turned into an Ascii int value by using the function Asc().

Ex: chr stick = "s" \n int stickAsciiVal = Asc(stick)

str – is an array of characters and its indexes can be retrieved.

Ex: str stuffedBear = "Teddy Bear" \n bark(stuffedBear[6:])

3.3 Reference types (differing from Visual BASIC and Java)

ball – behaves the same way an object type does in java.

array – acts the same way as in java, although syntax is simplified.

Ex: str[4] konasFavThings = {"shirts", "stuffed animals", "cats", "treats"}

4. Example Programs

Illustrate your new language with six (6) example programs that demonstrate its use; especially what's new and improved over current languages as well as Visual BASIC and Java, on which you based this. Please include in your examples Caesar cipher encryption and decryption programs like those of our earlier class projects.

1. Caesar Cipher encrypt

```
Park Class caesarCipherEn() {
    Park Sub Main() {
        bark("Caesar Cipher:")
        str testStr = speak("Enter a string to encrypt: ") //encrypt input
        int shiftAmount = Int(speak("Enter the shift amount to encrypt the string: "))
        str encryptedStr = encrypt(testStr, shiftAmount)
        bark("Your encrypted string is \"" + encryptedStr + "\"")
    }
    Function str encrypt(testStr As str, shift As int) {
        str encryptedStr
        int e = 0

        for (e until testStr.len by 1) {//default is by 1, but just to specifiy
            chr letter = testStr[e]
            int asciiVal = Asc(letter) //Asc() turns character into ascii value
            if ((asciiVal >= 65) && (asciiVal <= 90)) {
                if ((asciiVal == 65) && (shift < 0)) {
                    encryptedStr += Chr(asciiVal + shift + 26)
                } elif ((asciiVal == 90) && (shift > 0)) {
                    encryptedStr += Chr(asciiVal + shift - 26)
                } else {
                    encryptedStr += Chr(asciiVal + shift)
                }
            } else {
                encryptedStr += letter
            }
        }
        fetch encryptedStr
    }
}
```

2. Caesar Cipher decrypt

```
Park Class caesarCipherDn() {
    Park Sub Main() {
        bark("Caesar Cipher:")
        str encryptedStr = speak("Enter a string to decrypt: ") //decrypt input
        int shiftAmount = Int(speak("Enter the shift amount used to encrypt: "))
        str decryptedStr = decrypt(encryptedStr, -shiftAmount) // - of shiftAmount
        bark("Your decrypted string is \"" + decryptedStr + "\"")
    }
    Function str decrypt(encryptedStr As str, shift As int) {
        str decryptedStr
        int d = 0

        for (d until encryptedStr.len by 1) {//default is by 1, but just to specifiy
            chr letter = encryptedStr[d]
            int asciiVal = Asc(letter) //Asc() turns character into ascii value
            if ((asciiVal >= 65) && (asciiVal <= 90)) {
                if ((asciiVal == 65) && (shift < 0)) {
                    decryptedStr += Chr(asciiVal + shift + 26)
                } elif ((asciiVal == 90) && (shift > 0)) {
                    decryptedStr += Chr(asciiVal + shift - 26)
                } else {
                    decryptedStr += Chr(asciiVal + shift)
                }
            } else {
                decryptedStr += letter
            }
        }
        fetch decryptedStr
    }
}
```

3. Factorial

```
Class Factorial {
    Park Sub Main() {
        int num = speak("Enter a number to find it's factorial: ")
        int factorialNum = fact(num)
        bark(num + "! = " + factorialNum)
    }
    Function int fact(n As int) {
        if (n == 0) {
            fetch 1
        } else {
            fetch (n * fact(n - 1))
        }
    }
}
```

4. Merge Sort

```
Park Class MergeSort {
    Park Function int mergeSort(array As str[]) {
        int n = array.len
        int mid = n / 2
        this.divide(array, n)
    }
    Park Sub divide(array As str[], n As int) {
        if (n == 1) {
            fetch array
        } else {
            int mid = n / 2
            str[mid] left = {}
            str[n - mid] right = {} //n-mid to account for odd numbers
            //Divide array into left and right arrays
            int j = 0
            for (int i = 0 until n by 1) {
                if (i < mid) {
                    left[i] = array[i]
                } else {
                    right[j] = array[i]
                    j += 1
                }
            }
            divide(left, mid)
            divide(right, n - mid)
            merge(left, right, array, mid, n - mid)
        }
    }
    Park Sub merge(left As str[], right As str[], array As str[], nLeft As int, nRight
 As int) {
        int arrayInt = 0
        int leftInt = 0
        int rightInt = 0
        //Only merge if in order, while loops to check
        while ((leftInt < nLeft) && (rightInt < nRight)) {
            if (compareStr(left[leftInt] < right[rightInt])) {
                array[arrayInt++] = left[leftInt++]
                comparisons++
            } else {
                array[arrayInt++] = right[rightInt++]
                comparisons++
            }
        }
        //while left is less than the size of left array
        while (leftInt < nLeft) {
            array[arrayInt++] = left[leftInt++]
        }
        //While right is less than the size of the right array
        while (rightInt < nRight) {
            array[arrayInt++] = right[rightInt++]
        }
    }
}
```

5. Selection Sort

```
Park Class SelectionSort {
    Park Sub selectionSort(array As str[]) {
        int n = array.len
        //Go through each item in the string array to compare
        for (int i = 0 until n by 1) {
            //set min index
            int smallPos = i
            //set min string based off of min index
            str smallString = array[i]
            //Compare minString to the next item in the array
            for (int j = (i + 1) until n by 1) {
                if (compareStr(array[j] < smallString)) {
                    smallPos = j
                    smallString = array[j]
                }
            }
            if (smallPos != i) {
                str temp = array[smallPos]
                array[smallPos] = array[i]
                array[i] = temp
            }
        }
    }
}
```

6. Linked List

```
Park Class LinkedList {
    Home node head = null

    Park Function node search(int k) {
        node currentNode = this.head
        while ((currentNode != null) && (currentNode.id != k)) {
            currentNode = currentNode.next
        }
        fetch currentNode
    }
    Park Sub insert(node n) {
        n.next = this.head
        if (this.head != null) {
            this.head.prev = n
        }
        this.head = n
        n.prev = null
    }
    Park Sub delete(node n) {
        if (n.prev != null) {
            n.prev.next = n.next
        } else {
            this.head = n.next
        }
        if (n.next != null) {
            n.next.prev = n.prev
        }
    }
}
```