Konig Programming Language Proposal

Jessica Ling Yan (jly2121), Matteo Sandrin (ms4911),
Delilah Beverly (db3250), Lord Crawford (lrc2161)

February 1, 2021

1 Overview

The Konig language is named after the "Seven Bridges of Königsberg", a famous math problem that laid the foundations of graph theory. It also means "king" in German. The language is designed to make defining and manipulating graphs easier and more enjoyable.

Konig is an imperative, statically typed language. The language's syntax is similar to Java, but with the addition of a number of operators specific to graph theory. Furthermore, a number of arithmetic operators perform a dual function as both set operators on graphs, nodes and edges, and as regular arithmetic operators.

Konig features a set of standard library functions, which allow the user to easily traverse and apply modifications to a graph. Depth-first search and breadth-first search are implemented out of the box, and can be readily used inside any loop.

2 Language Details

2.1 Types

The Konig programming language supports several primitive data types. Some of these data types can be found in any programming language, such as int, bool, float and char. Other data types are specific to graph theory, such as node, edge and graph. The language is statically and strongly typed, so the type of each variable is explicitly specified at the time of declaration. For those data types that contain another primitive, such as a node, the type of that primitive is specified between angle brackets after the container's type, in a Java-like fashion.

Data Type	Description	Example
`int`	A 4 byte integer type	`int x = 4;`
`bool`	A 1 bit boolean type	`int x = false;`
`float`	An 8 byte floating-point type	`int x = 4.0;`
`list`	A variable-length list type	`list<int>x = list<int>{1,2,3};`
`node`	A single graph node type	`node<float>x = node{4.0};`
`edge`	A single graph edge type	`edge<int>x = edge{from, to, val};`
`graph`	A graph type	`graph x = graph{};`

2.2 Operators

Konig implements a set of operators that are specific to graph theory, such as , ∼> and >>. These operators make it easy to create and compose graphs, both directed and undirected. In addition, Konig implements all classic arithmetic operators, and a set of comparison operators.

Operator	Operands	Description
a ∼> b	a is a node b is a node	Creates a directed edge from a to b. It will fail if both a and b are not members of the same graph
a ∼ b	a is a node b is a node	Creates an undirected edge between a and b. It will fail if both a and b are not members of the same graph
a >> g	a is a node g is a graph	Adds the node a to the graph g
a + b a - b a / b a * b a++ a–	a is an int, float b is an int, float	Performs the corresponding arithmetic operation (sum, difference, float division, multiplication, increment, decrement)
a >b a <b a =>b a <= b a == b a != b	a is any type b is any type a and b have the same type	Performs the corresponding comparison operation, and returns a boolean value
a and b a or b not a	a is a bool b is a bool	Performs the corresponding boolean operation between boolean values

2.3 Keywords

The following keywords are reserved in Konig:

```
int, bool, float, char, graph, node, edge, list
for, if, else, else if, while
ko, return, true, false, and, or, not
```

2.4 Control Flow

Konig implements control flow in a standard manner:

```
if (exp) {
    ...
} else {
    ...
}

for (int i; i < 10; i++) {
    ...
}

while (exp) {
    ...
}
```

2.5 Functions

Functions in Konig are defined with the reserved keyword "ko". The function arguments are specified inside the parentheses and after the function name. The return type is specified after the closing parenthesis.

```
ko add(int x, int y) int {
    return x + y;
}
```

2.6 Comments

Comments are specified with a double forward-slash. Multiline comments use a forward slash paired with an asterisk.

```
graph g = graph{} // this is a comment

/* this is a multiline comment
and it keeps going
and going */
```

2.7 Standard Library

The Konig programming language features a rich standard library for creating and manipulating graphs. The exact scope and breadth of this library is still under discussion, but we feel confident that the following functions will be implemented:

Function signature	Description
`ko print(graph g) int`	Visualize the graph g
`ko neighbors(node n) list<node>`	Returns a list of all neighbors of the node n
`ko edges(graph g) list<edge>`	Returns a list of edges in the graph, without any ordering guarantee
`ko nodes(graph g) list<node>`	Returns a list of nodes in the graph, without any ordering guarantee
`ko isNeighbor(node a, node b) boolean`	Returns true if nodes a and b have an edge connecting them
`ko dfs(graph g, node start) list<node>`	Returns a list of nodes in the graph, in depth-first search order
`ko bfs(graph g, node start) list<node>`	Returns a list of nodes in the graph, in breadth-first search order
`ko paths(node a, node b) list<list<node>>`	Computes all the possible paths from node a to node b
`ko fullyConnect(graph g)`	Connects each node to every other node in a given graph

3 Examples

3.1 Feature Showcase

```
graph g1 = graph{}; // initialize an empty, undirected graph

node<int> n0 = node{0}; // initialize a node with value 0

n0 >> g1 // add a node to the graph g1
node n1 = node{1}; // initialize a node with value 1

node n3 = node{3}; // initialize a node with value 3
n1 ~ n2; // create an undirected edge between n1 and n2
n1 ~> n3; // create a directed edge from n1 to n2

// create an undirected edge between n1 and n2, labeled "amazing edge"
edge e1 = edge{n1, n2, true, "amazing edge"}

graph g2 = Graph(); // initialize an empty, undirected graph
node{2} >> g2;

list<node> nodes = dfs(g3, n0); // traverse g3 with depth-first search
for (int i = 0; i < length(nodes); i++) {
    print(nodes[i]); // print the value of each node
}

// We'd like to link it with a graph visualization library
print(g3);
```

3.2 Algorithm Implementation

Calculate all the possible paths between two friends in a social network.

```
list<list<char>> names = list<list<char>>{
    "John",
    "Ann",
    "Brian,
    "Monica"
};
list<list<int>> friends = list<list<int>>{
    list<int>{1, 3},    // John's friends
    list<int>{0, 2, 3}, // Ann's friends
    list<int>{1}        // Brian's friends
    list<int>{0, 1}     // Monica's friends
};
graph g = graph{};

for (int i = 0; i < length(names); i++) {
    node n = node{names[i]}; // create new node n
    n >> g; // add node n to graph g
}

for (int i = 0; i < length(friends); i++) {
    list<int> nbrs = friends[i];
    for (int j = 0; j < length(nbrs); j++) {
        if (not isNeighbor(g[i], g[nbrs[j]]) {
            g[i] ~ g[nbrs[j]]; // create an undirected edge
        }
    }
}

list<list<node>> p = paths(g[0], g[3]) // how many paths between John and Monica?
print(p);
/*
 * =>
 * list<list<node>>{
 *     list<node>{
 *         node{"John"},
 *         node{"Monica"}
 *     },
 *     list<node>{
 *         node{"John"},
 *         node{"Ann"},
 *         node{"Monica"}
 *     }
 * }
*/
```