Konig – Language Reference Manual

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

1. Introduction

Graphs have an important role in a number of applications including networks, data processing, databases and everything in between. The Konig programming language is aimed at making the creation and manipulation of graphs easier and more enjoyable.

Konig is an imperative, statically typed language. The language's syntax is similar to C, but with the addition of a number of operators and functions specific to graph theory. Furthermore, Konig uses a syntax for generic types similar to Java.

The 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.

2. Lexical Conventions

The following conventions are used to specify the context free grammar behind Konig:

Label	**Description**
`ID`	An identifier. It can be either a variable name or a function name
`EXP`	An expression, which returns a value
`STMT`	A statement, which does not return a value
`if, and, for ...`	Any lowercase word is understood to represent the corresponding reserved keyword

2.1 Identifiers

Identifiers in Konig are made up of any lowercase or uppercase ASCII letter, any decimal number, or the underscore character. However identifiers must start with a lowercase or uppercase ASCII letter.

```
// valid identifiers
int one_two = 0;
int two33 = 0;
int OneTwoTree = 0;
```

```
// invalid identifiers
int 1two = 0;
int _two = 0;
int !@# = 0;
```

Identifiers therefore match the regular expression `[a-zA-Z][a-zA-Z0-9_]*`
Grammar: `TYPE ID = EXP`

2.2 Comments

Comments are specified with a double forward-slash. Multiline comments use a forward slash paired with an asterisk as a starting token, and an asterisk paired with a slash as an end token.

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

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

Grammar: `TYPE ID = new TYPE{EXP, EXP, …}`

2.3 Separators

Konig uses semicolons to separate expressions, curly braces to separate blocks of expressions, and parentheses to isolate expressions that take precedence, and override the default preference order.

```
int x = 0; // here ';' is acting as a separator
x = x + 1;

if (x) { // here '{' signifies the start of a code block
     x = 2;
} // here '}' signifies the end of a code block

x = (x + 1) * (x + 2) // here '(' and ')' override precedence
```

Grammar: `{ STMT; STMT; STMT ... }`

2.4 Literals

2.4.1 Boolean Literals

The boolean literal is represented in Konig by the keywords `true` and `false`.

```
bool x = true;
```

2.4.2 Integer Literals

The integer literal is represented in Konig by an arbitrary length sequence of digits each between 0 and 9.

```
int x = 1234;
```

2.4.3 Float Literals

The floating point literal is represented in Konig as either pair of integers joined by a period, accompanied by an optional exponent indicated as following:

```
// these are all equivalent floating point literals
float x = 12.34;
float y = 1.234e+1;
float z = 123.4e-1;
float w = 1.234e1;
float j = 1234.e-2;
float k = .1234e+2;
```

Floating point literals therefore match the following regular expression:

```
[0-9]\.[0-9]*([eE][\+-]?[0-9]+)?
```

2.4.4 String Literals

A string literal in Konig is a shorthand for a list of characters. The string literal is defined by an arbitrary sequence of characters contained within double quotes.

```
list<char> x = "Hello World";
```

Grammar:

```
TYPE =
    int
  | bool
  | float
  | char
  | list<TYPE>
```

3. Data 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 and float. 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.

3.1 Primitives

Data Type	**Description**	**Example**
int	A 4 byte integer type	int x = 3;
bool	A 1 bit boolean type	bool x = false;
char	A 1 byte ASCII character	list<char> x = "Hello world";
float	An 8 byte floating-point type	float x = 1.234;
void	A reference to a null-like type	float x = void;

3.2 Lists

The list in Konig is identified by the keyword list followed by the type of its contents in angle brackets. All elements of a list must be the same type as declared at initialization.

```
list<int> x = [1, 2, 3];
```

Lists can be accessed and modified through the square bracket syntax:

```
int y = x[2]; // returns 3
x[2] = 4;
```

Grammar:

```
TYPE IND = [EXP, EXP, … ]
IND[EXP] = EXP
```

3.2.1 List Functions

A number of functions are built into the standard library to manipulate lists. Each function never modifies the list directly, and instead returns a copy of the list.

Function signature	**Description**
`ko list<T> append(list<T> lst, T elem)`	Appends `elem` to the end of the list, and returns a new list
`ko list<T> pop(list<T> lst)`	Removes the last element of the list and returns a new list
`ko int length(list<T>)`	Returns the length of the list

3.3 Nodes

A single node in Konig is identified by the keyword `node`. When initialized, a node can be passed an optional data member, which will default to `void` if not specified. Nodes are initialized with the keyword `new`.

```
node x = new node{"hello world"};
```

A node's data member can be accessed by:

```
list<char> y = x.val; // returns "hello world"
```

3.4 Graphs

A graph in Konig is identified by the keyword `graph`. When initialized, a graph does not take any arguments.Graphs are initialized with the keyword `new`.

```
graph x = new graph{};
```

3.5 Edges

Edges cannot be directly initialized by the user. However, an edge object will be returned by the operations that manipulate edges.

4. Operators

4.1 Arithmetic Operators

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

Operator	**Operands**	**Return type**	**Description**

`a @ g`
 	
`a ! g`
	`a` is a `node`
	`g` is a `graph`
		`graph`	Adds the node a to the graph g
			
			Removes the node a from the graph g

`a + b`
`a - b`
`a / b`
`a * b`
	`a` is an `int, float`
	`b` is an `int, float`
		`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` is any type
	`b` is any type
	
	`a` and `b` have the same type
		`bool`	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`
		`bool`	Performs the corresponding boolean operation between boolean values

5. Graph Semantics

5.1 Graphs

Initialize an empty graph and assign it to a variable `g1`:

```
graph g1 = new graph{};
```

Combine two already existing graphs `g1` and `g2` :

```
combineGraphs(g1, g2);
```

5.2 Nodes in Graph

Initialize a node `n0` with a value of `0`:

```
node n0 = new node{0};
```

Add a node `n0` to the graph `g1`:

```
n0 @ g1;
```

Delete node `n0` from graph `g1`:

```
n0 ! g1;
```

Return list of nodes from graph `g1`:

```
nodes(g1);
```

Return list of all nodes that can be accessed from node `n1`:

```
neighbors(n1);
```

5.3 Edges in Graph

Create an undirected edge between `n0` and `n1` with weight value 0:

```
setEdge(n0, n1, 0);
```

Create a directed edge from `n1` to `n2` with weight value 5:

```
setDirEdge(n1, n2, 5);
```

Update edge weight from `n1` to `n2`:

```
updateEdge(n1, n2, 15);
```

Delete edge from `n1` to `n2`:

```
deleteEdge(n1, n2);
```

Access edge object between `n1` and `n2` :

```
edge e = getEdge(n1, n2);
```

Get edge type, returns int 0 for undirected edges and int 1 for directed edges:

```
e.type
getEdge(n1, n2).type
```

Get edge weight:

```
e.weight
getEdge(n1, n2).weight
```

6. Keywords

The following keywords are reserved in Konig:

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

7. Control Flow

7.1 While Loop

The while statement has the form:

```
while (EXP) { STMT; STMT; ... }
```

The statements inside the code block are executed multiple times. After each execution of the code block, the expression in parenthesis is evaluated, and if returning `true` the statements inside the block are executed again.

7.2 For Loop

The for statement has the form

```
for ( EXP; EXP; EXP ) { STMT; STMT; ... }
```

The first expression specifies initialization for the loop. The second expression specifies a test, evaluated before each iteration, which terminates the loop once it evaluates to `false`. The third expression is evaluated at every iteration. It usually contains an increment.

7.3 Conditionals

There are two forms of conditional statements in Konig:

```
if ( EXP )  { STMT; STMT; ... }

if ( EXP ) {STMT; STMT; ... } else { STMT; STMT; ... }
```

In both forms, if the expression within parentheses evaluates to `true`, then the code block is executed. The second form of conditional also features a second code block, followed by the keyword else. This code block is executed if the expression evaluates to `false`.

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

Grammar:

```
ko TYPE ID ( TYPE ID, TYPE ID, ... ) { STMT; STMT; ... }
```

9. Standard Library

The Konig programming language features a rich standard library for creating and manipulating graphs.

Function signature	**Description**
ko edge setEdge(node a, node b, float weight) {}	Constructs an undirected edge between two nodes with a weight to the edge. Users can define a default weight of 0 or NULL. Returns an error if given nodes do not exist or are not in the same graph.
ko edge setDirEdge(node a, node b, float weight) {}	Constructs a directed edge from node a to node b with a weight to the edge. Users can define a default weight of 0 or NULL. Returns an error if given nodes do not exist or are not in the same graph.
ko edge getEdge(node a, node b)	Returns the edge object between node a and node b
ko edge updateEdge(node a, node b, float weight) {}	Updates the weight of the edge between node a and node b
ko edge deleteEdge (node a, node b) {}	Deletes the edge between node a and b
ko list<node> neighbors(node n) {}	Returns a list of all neighbors that node n can reach: nodes that node n has a directed edge to or an undirected edge

	with. This will be listed by order of edge weight.
`ko list<node> nodes(graph g) {}`	Returns a list of nodes in the graph, without any ordering guarantee
`ko bool isConnected(node a, node b) {}`	Returns true if nodes a and b have an edge connecting them
`ko list<char> viz(graph g) {}`	Visualize the graph g
`ko graph combineGraphs(graph g1, graph g2)`	Returns a new graph with all the nodes and connections in graph g1 coupled with that of graph g2's nodes and connections.

10. Sample Code

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

node n0 = new node{0}; // initialize a node with value 0
n0 @ g1; // add node n0 to the graph g1

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

node n2 = new node{2}; // initialize a node with value 3
n2 @ g1; // add node n1 to the graph g1

setEdge(n0, n1, 0); // create an undirected edge between n0 and n1
setDirEdge(n1, n2, 0); // create a directed edge from n1 to n2

list<node> nodes = neighbors(n0); // gets a list of neighbor nodes of n0
for (int i = 0; i < length(nodes); i++) {
    print(nodes[i]); // print the value of each node
}

viz(g1); // Links to a graph visualization library to display any graph
```