Knowledge Graph Language
Reference Manual

October 26, 2015

Name	UNI	Role
Bingyan Hu	bh2447	Project Manager
Cheng Huang	ch2994	Language Guru
Ruoxin Jiang	rj2394	System Architect
Nicholas Mariconda	nm2812	Verification & Validation

Contents

1 Introduction

1.1 Motivation

Almost everything in the world is connected together through some complex web of relationships. As such, building, expressing and traversing graphs is one of the most essential applications of computer science. However, it is common knowledge that implementing graphs in traditional languages is no trivial task. Many past projects have addressed this problem by designing graph-based languages that make building graphs easier. However, such projects were limited by having single, fixed relationships between nodes. Often, algorithms that operate on real-world data – such as machine learning and information retrieval algorithms – are too obfuscated to be represented by a graph with one-dimensional relationships.

1.2 Proposed Uses

Knowledge Graph Language (KGL) is a domain-specific graphing language that supports multiple user-defined relationships between nodes. Edges, nodes, and graphs are built-in types of the language; however, two nodes can be connected by multiple edges, with each edge being identified by a unidirectional, user-defined relationship. KGL reaps many of the benefits of a graphing domain-specific language – users can build, express and traverse complex graphs succinctly – while also providing a means for users to query their graphs directly. This is the main thrust of the language – by providing the users with a mechanism for defining their own relationships between nodes, they can extract a more robust collection of data through graph queries.

2 Lexical Conventions

2.1 Comments

The characters ## starts a single-line comment, which terminates at the end of the line. The characters /# introduces a multi-line comment, which terminates with characters #/. Comments do not nest.

2.2 Identifiers

An identifier is a combination of letters , numbers and underscores _. The first character must be a letter or an underscore. Upper and lower case letters are different.

2.3 Keywords

The following is a list of reserved keywords in the language. They cannot be used as variable or function names:

int	float	char	boolean	string
graph	node	list	set	dict
for	while	continue	break	in
if	elif	else	true	false
func	return	void	null	

2.4 Literals

2.4.1 Integer Literal

An integer literal consists of an optional minus sign, followed by a sequence of digits. If the digit sequence has more than one digit, the first may not be zero. The literals are interpreted as base-10 (decimal) numbers. Examples: `76 -65 43445 0 -9090`

2.4.2 Float Literal

A float literal consists of a signed integer part, a decimal part and a fraction part. The integer and fraction parts both consist of a sequence of digits. Either the integer part, or the fraction part (not both) may be missing. Examples: `-1.36 67.0 .67 8.9`

2.4.3 Boolean Literal

A boolean literal has two values: `true`, `false`

2.4.4 Char Literal

A char literal is a single character enclosed in single quotes. The following escape sequences may be used: \n, \t, \\. Examples: `'a' ' ' 'H' '7'`

2.4.5 String Literal

A string literal consists of a sequence of characters enclosed in double quotes. The following characters can be escaped inside of strings with a backslash: \n, \t, \\. Examples: `"hello" "" "367"`

2.5 Punctuations

Colon :
separator of a key : value pair in dictionary

Semicolon ;
end of statement
separator of edge list in graph

Parenthesis ()
expression precedence
conditional parameters
function arguments

Brackets [ ]
node access
list/dictionary access

Curly braces {}
statement blocks
function body

Angle brackets <>
element type of derived types (list/set)

Comma ,
separator of function arguments
separator of elements in list and key value pairs in dictionary

Edge brackets –( )–>
edge expression

List brackets [| |]
list expression

Set/Dict bracket (| |)
set/dictionary expression

Graph brackets {| |}
: graph expression

Quotes ' '
: character literal declaration

Double quotes " "
: string literal declaration

2.6 Operators

The following table lists the precedence and associativity of the operators. Operators are listed top to bottom, in descending precedence.

Precedence	Operator	Description	Associativity
1	()	Function call	
1	[ ]	list/dictionary/graph access	
2	!	Logical NOT	
3	* / %	Multiplication, division, remainder	
4	+ -	Addition, Subtraction	
5	>>= <<=	Relational greater, greater equal, less, less equal	Left-to-right
6	== !=	Relational equal, not equal	
7	in	Membership	
8	&&	Logical AND	
9	||	Logical OR	
10	=		Right-to-left

2.7 Whitespace

Symbols that are considered to be whitespace are blank, tab, and newline characters.

3 Types

3.1 Primitive Types

int	a signed 32-bit integer
float	a signed single precision floating point type, with a size of 32 bits
char	an 8-byte data type used to store ASCII characters
boolean	a 1-byte data type that only accepts true or false
string	a sequence of chars

3.2 Graph-related Types

3.2.1 Node

Nodes are building blocks of a graph. Each node in a graph contains an unique id (different from other node ids in the same graph) and a dictionary of user-defined attributes. A node can only exist in a graph; there is no isolated node. A node in a graph could only be created when a new edge(relationship) of this node is added in the graph.

The `node` type is only a pointer to a real node in a graph. It can either point to a node in graph or be `null`. A variable of type `node` can access and modify the real node it points to in graph.

For example:

```
node a = g["Nick"];      ## a points to the node with id "Nick" in graph g
a["age"];                ## returns the value of key "age" in the node's attribute dictionary
a["age"] = 22;           ## sets the value of key "age" to 22 in the node's attribute dictionary
```

3.2.2 Graph

A graph is defined by a set of nodes and their unidirectional relationships between each other. An unidirectional relationship between two nodes is also called an edge, defined as such:

```
source_id--(label)-->target_id
## the relationship between the source node and the target node is of value label

"Nike"--("knows")-->"Mike"
## represents the "knows" relationship from node "Nike" to node "Mike"
```

3.3 Derived Types

Besides the primitive types, `graph` and `node`, there is a conceptually infinite class of derived types constructed from any type:

list<vtype >	A list is an ordered sequence of elements of a given type (vtype)
set<vtype >	A set is a collection of elements of a given type (vtype) without duplicates.
dict	A dictionary is a collection of key value pairs. Keys are of type string, values of any type. Duplicated keys are not permitted.

3.4 Void

void can only be used as function return types to indicate that a function has no return value.

4 Expressions

4.1 Primary expressions

Primary expressions are literals, identifiers, graph expressions, list expression, set expressions, dict expressions or expressions in parentheses.

```
primary-expression:
    literal
    identifier
    null
    set-expression
    list-expression
    dict-expression
    graph-expression
    (expression)
```

4.1.1 Literals

Literals are integer literals, float literals, boolean literals, char literals and string literals (see Section 2.4).

4.1.2 Set/List/Dictionary expressions

A set expression is an empty set or a non-empty collection of expressions.

```
set-expression:
    set(| |)
    (| expression-list |)
```

A list expression is an possibly-empty ordered sequence of expressions.

```
list-expression:
    [| expression-list-opt |]
```

A dictionary expression is possibly-empty collection of key/value pairs. Each key/value pair contains two expressions, the key and the value, separated by a colon.

```
dict-expression:
    (| kv-pairs-opt |)

kv-pair:
    expression : expression
```

4.1.3 Graph expressions

A graph expression is a possibly-empty collection of edges(relationships). Each edge contains three expressions: the id of source node, the label of this relationship, and the id of target node.

```
graph-expression:
    (| edge-list-opt |)

edge:
    expression--(expression)-->expression
```

4.2 Postfix expressions

Postfix expressions are primary expressions, function calls, and list/dict/graph references

```
postfix-expression:
    primary-expression
    postfix-expression (argument_list_opt}
    postfix-expression [expression]
```

4.2.1 References

A posfix expression followed by an expression in square brackets is a postfix expression denoting a reference to list element/graph node /dictionary value/ node attribute.

Reference to	Syntax	Description
list element	list_var[3]	the 4th element in list list_var
graph node	graph_var["Nick"]	the node with id "Nick" in graph graph_var
dictionary value	dict_var["k"]	the value of key "k" in dictionary dict_var
node attribute	node_var["age"]	the value of attribute "age" in node node_var

4.2.2 Function calls

A function call is an function identifier, followed by parentheses with a possibly empty, comma-separated argument list.

4.3 Operator expressions

Unary Operator

```
unary-expression:
        postfix-expression
        ! expression
```

Multiplicative Operators

```
multiplicative-expression:
        unary-expression
        multiplicative-expression * mulplicative-expression
        multiplicative-expression \ mulplicative-expression
        multiplicative-expression % mulplicative-expression
```

Additive Operators

```
additive-expression:
        multiplicative-expression
        additive-expression + additive-expression
        additive-expression - additive-expression
```

Relational Operators

```
relational-expression:
        additive-expression
        relational-expression < relational-expression
        relational-expression > relational-expression
        relational-expression <= relational-expression
        relational-expression => relational-expression
```

Equality Operators

```
equality-expression:
        relational-expression
        equality-expression == equality-expression
        equality-expression != equality-expression
```

Logical AND Operator

```
logical-AND-expression:
        equality-expression
        logical-AND-expression && logical-AND-expression
```

Logical OR Operator

```
logical-OR-expression:
        logical-AND-expression
        logical-OR-expression || logical-OR-expression
```

Assignment expressions

```
assignment-expression:
        logical-OR-expression
        assignment-expression = assignment-expression
```

4.3.1 List, Set and Dict

Only the following operators have meanings for list-expression, set-expression and dict-expression: ! + - == != =

Operand	Operator	Operand	Description
	!	list/set/dict-expression	true if list/set/dict is empty, false otherwise
list-expression	+	list-expression	concatenation of two lists
set-expression	+	set-expression	union of two sets
set-expression	-	set-expression	set of elements in the first set but not in the second
dict-expression	+	dict-expression	union of two dictionaries
dict-expression	-	set-expression	dict of key/value pairs in the dict whose key not in the
list/set/dict-expression	==	list/set/dict-expression	true if they contain the same elements, false otherwise
list/set/dict expression	!=	list/set/dict expression	true if they do not contain the same elements, true othe

4.3.2 Graph and Node

Only the following operators have meanings for graph-expression and node: ! + - == != =

Operand	Operator	Operand	Description
	!	node	true if node is null, false otherwise
node	+	dict-expression	union of the node's attribute dictionary and the dict
node	-	set-expression	dict of key/value pairs in the node's attribute dict whose key not in the set
node	==	node	true if they point to the same node in the same graph
node	!=	node	false if they point to the same node in the same graph
	!	graph-expression	true if graph is empty, false otherwise
graph-expression	+	graph-expression	union of two graphs
graph-expression	-	graph-expression	graph of edges in the first but not in the second
graph-expression	==	graph-expression	true if they contain the same nodes and relationships
graph-expression	!=	graph-expression	false if they contain the same nodes and relationships

Examples:

```
## add nodes and relationships to a graph
graph g = {||};
g = g + {|"Nick"--("friends")-->"Mike"|};  ## adds two new nodes and a relationship to g
g = g + {|"Nick"--("knows")-->"Jack"|};    ## adds a new relationship to g

## add an attribute to node
```

```
string key = "age"; int value = 22;
g["Nick"] = g["Nick"]+[key:value];

## remove an attribute from node
g["Nick"] = g["Nick"] - (| key |);

## remove nodes / relationship from graph
g = g - {|"Nick"--("knows")-->"Jack"|};
## node "Jake" which is connected to no other nodes in g is deleted from the graph as well.
```

5 Declarations

Declarations defines the types of identifiers and initializes their values.

```
declaration:
    type-specifier identifier
    type-specifier identifier = initializer
```

5.1 Type Specifiers & Initializer

The type-specifiers are

```
type-specifier:                 initializer:
    void                            N/A (void only used for func return type)
    int                             expression evaluated to int
    float                           expression evaluated to float
    boolean                         expression evaluated to boolean
    char                            expression evaluated to char
    string                          expression evaluated to string
    graph                           graph-expression
    node                            reference to node in graph
    dict                            dict-expression
    set<type-specifier>             set-expression
    list<type-specifier>            list-expression
```

5.2 Graph and Node

The following examples shows the declaration and initialization of graphs and nodes.

```
/# Syntax of graph declaration and initialization
graph g = {| sourceID--(edgeLabel)-->targetID;
             sourceID--(edgeLabel)-->targetID;...|};
#/

## declare a graph g1
graph g1;

## declare a graph g2 and initialize it with two relationships
graph g2 = {|
             "Nick"--("friends")-->"Mike";
             "Mike"--("knows")-->"Jake"
            |};
```

```
## declare and initialize another graph g3
string rel = "knows";
string sourceID = "Nick"; string targetID = "Mike";
graph g3 = {| sourceID--(rel)-->targetID |};

## declares a node
node n1;

## declare and initialize nodes
node n2 = g2["Nick"];
node n3 = g3[targetID];
```

5.3 List, Set and Dict

```
list<int> l1 = [||] ## declare an empty integer list
list<float> l2 = [|1.0, 2.0, 3.0 |] ## declare an float list with three elements

set<string> s = set(||) ## declare an empty string set
set<string> s = (|"one", "two", "three"|) ## declare a string list with three elements

dict d = (||) ## declare an empty dictionary
dict d = (|"name": "Nick", "age": 22, "sex":male, "married":false|) ## declare a dict with key-va
```

6 Statements

```
statement:
    block-statement
    expression-statement
    conditional-statement
    loop-statement
    jump-statement
```

6.1 Block statements

```
block-statement:
    { statement-list-opt }
```

6.2 Expression statements

Expression statements are mostly used as assignments or function calls.

```
expression-statement:
    expression;
```

6.3 Conditional statements

A conditional statement is used to express decisions.

```
conditional-statement:
    if (expression-opt) statement
```

```
    if (expression-opt) statement else statement
    if (expression-opt) statement elif-list

elif-list:
    ELIF (expression-opt) statement
    ELIF (expression-opt) statement ELSE statement
    ELIF (expression-opt) statement elif_list
```

6.4 Loop statements

Loops are control statements that specify iteration, which allow a block of code (”substatement”) to be executed multiple times. KGL supports two types of loops: the `for` loop and the `while` loop.

```
loop-statement:
    for (expression-opt; expression-opt; expression-opt) statement
    for (expression in expression) statement
    while (expression) statement
```

6.4.1 For loop

The `for` loop supports two separate usages. The first is the standard usage in which the for loop takes three expressions: an initialization expression, a test expression, and an update expression.

The second usage of the `for` loop resembles the Pythonic implementation – it executes the substatement for each element in a given collection. The first expression is a user-specified label that will serve as a reference to the current element in the collection, and the second expression is the handle for the collection being iterated through. Supported collections include sets, lists, dictionaries and graphs.

6.4.2 While loop

The `while` loop is the `for` loop stripped of the initialization and update expressions. It contains the test expression and statement. The statement is executed repeatedly until the test expression is evaluated to a boolean false. The expression is reevaluated before each iteration of the loop.

6.5 Jump Statements

```
jump-statement:
    continue;
    break;
    return expression_opt;
```

6.5.1 Continue

`continue` is used to skip some statements in the iteration loops and cause control to pass to the loop-continuation portion of the smallest enclosing such statement. For example:

```
while (expr) {
    ...
    if (expr) { continue; }
    ...
}
```

6.5.2 Break

`break` statement is used to terminate all iteration loops. For example

```
while (expr) {
    ...
    if (expr) { break; }
    ...
}
```

6.5.3 Return statements

The `return` statement terminates the execution of a function and returns control to the calling function. When `return` is followed by an expression, the value is returned to the caller of the function.

7 Built-in Functions

- `getNodes(graph g)`: returns the set of all nodes in graph g
- `getNeighors(node v)`: returns the set of all nodes connected to node v, regardless of the relationship
- `getNeighbors(node v, string label)`: returns the set of all nodes connected to the node v by the relationship 'label'
- `getLabels(node v1,node v2)`: returns the set of labels that connect two given nodes, v1 and v2