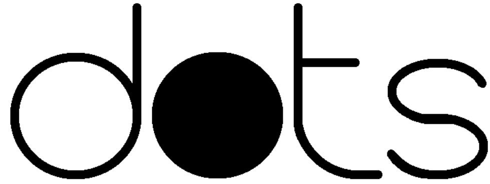

A graph language.

Hosanna Fuller (hjf2106) — Manager
Rachel Gordon (rcg2130) — Language Guru
Yumeng Liao (yl2908) — Tester
Adam Incera (aji2112) — System Architect

September 2015

Contents

1 Lexical Elements **3**
1.1 Identifiers . . . 3
1.2 Keywords . . . 3
1.3 Literals . . . 3
1.3.1 String literals . . . 3
1.3.2 Number literals . . . 3
1.3.3 Separators . . . 3
1.4 Whitespace . . . 4

2 Data Types **4**
2.1 Primitive Types . . . 4
2.1.1 num . . . 4
2.1.2 string . . . 4
2.1.3 bool . . . 4
2.2 Node and Graph Objects . . . 5

3 Expressions and Operators **5**
3.1 Number and String Operators . . . 6
3.2 Node Operators . . . 6
3.3 Graph Operators . . . 8
3.4 Subscript Notation . . . 9
3.4.1 Subscript . . . 9
3.5 Operator Precendence . . . 9
3.6 Member Access . . . 9
3.7 Control Flow Statements . . . 9
3.8 Loop Statements . . . 10
3.8.1 While Loops . . . 10
3.8.2 For Loops . . . 10
3.9 Other Statements . . . 11
3.9.1 Break . . . 11

3.9.2 Continue . . . 11
3.9.3 Return . . . 11

4 Functions **12**
4.1 Function Declaration and Definition . . . 12
4.2 Return Statements . . . 12
4.3 Parameter List . . . 12
4.4 Calling Functions . . . 13
4.5 Variable Length Parameter Lists . . . 13
4.6 Built-in Functions . . . 14

5 Program Structure and Scope **14**
5.1 Program Structure . . . 14
5.2 Scope . . . 14

6 Sample Program **15**

1 Lexical Elements

1.1 Identifiers

Identifiers are strings used to name variables and functions. The first character of an identifier must be a letter. Identifiers can contain lowercase letters, uppercase letters, digits, and the underscore character '_'. Identifiers are case-sensitive.

1.2 Keywords

The following keywords are part of dots itself, and as such cannot be used as identifiers:

```
return break continue def for while if else in dict list num bool string graph node
INF true false null
```

1.3 Literals

Literals are raw numeric or string values. The compiler will automatically determine the type of a literal, unless it is explicitly cast.

1.3.1 String literals

String literals are sequences of characters wrapped in double quotes. They may include any characters but some characters must be represented by escape sequences:

\\	backslash
\"	double quote
\n	newline
\r	carriage return
\t	horizontal tab
\v	vertical tab

For an example, see section 2.1.2 on the string type.

1.3.2 Number literals

Number constants consist of an optional negative sign, an optional sequence of digits, an optional decimal point, and another optional sequence of digits. The second sequence of digits requires the presence of the decimal point. Either the first or second sequence of digits must appear (both is also valid).

1.3.3 Separators

Separators separate tokens. Whitespace characters (' ', '\n', '\t', etc.) are separators, but not tokens. The other separators are single-character tokens. They are:

```
( ) [ ] { } ; , . < >
```

1.4 Whitespace

In d.o.t.s. whitespace is defined as any of the special characters `\r`, `\n`, and `\t` as well as the character obtained by pressing the spacebar. Whitespace is generally ignored and used as a token delimiter, except for within string literals.

2 Data Types

2.1 Primitive Types

2.1.1 num

The `num` data type represents all numbers in d.o.t.s. There is no distinction between the traditional data types of `int` and `float` from C, which means for example that there is no difference between the values `5` and `5.0`. The comparative ordering of nums is the same as that of numbers in mathematics.

```
num x = 5;
num y = 5.0;
num z = x;

num q = 3.14159;

num a, b, c;
```

Listing 1: Declaration of "num" types.

In Listing 1 variables `a, b, c, x, y, z`, and `q` are all of the type `num`. Variables `x`, `y`, and `z` store equivalent values. Variables `a`, `b`, and `c` are all equal to `null`.

2.1.2 string

A `string` is a sequence of 0 or more characters. Comparative ordering of strings is determined sequentially by comparing the ASCII value of each character in the two strings from left to right.

```
string a = "alpha";
string empty = "";
string char = "a";
string escape_sequence_example = "And he said, \"Look at this! I have quotes
    inside my string!\nAnd a newline too!\"";
```

Listing 2: Declaration of "string" types.

2.1.3 bool

The `bool` type is a logical value which can be either the primitive values `true` or `false` or an expression which evaluates to those literals.

```
bool t = true;
bool f = false;
```

Listing 3: Declaration of "bool" types.

2.2 Node and Graph Objects

A `node` object represents a single vertex in a Graph, and a `graph` object represents a collection of graphs (which can be empty). *Note*: A node is a graph, but a graph is not a node.

Recursive definition of graph objects:

- An empty graph is a graph.
- A node is a graph.
- A graph added to a graph is a graph.
- A graph subtracted from a graph is a graph.

A graph contains only the field `vertices`, which is a list of all node objects contained within the graph.

A node contains the fields `value, in, out`. Internally, a node is uniquely identified by an id number set by the compiler, but this distinction is invisible to the programmer. The `value` field is a string object, and simply represents some value that the node contains. One possible use of the value field is to allow users to assign a more semantic meaning to nodes (ex. setting the value to the name of a city). The `in` field is a dict mapping nodes that the current node has edges into to weights. Similarly, the `out` field is a dict mapping nodes that have edges into the current node to weights. The keys of the two dicts are nodes. An example of accessing the `in` and `out` dicts of a node can be seen in Listing 7.

Since node has an *is-a* relationship with graph, it also contains a `vertices` field, but this is set upon declaration to contain only the node itself, and cannot be altered by the user.

Nodes can be declared in two different ways. In the first, the variable can simply be declared with the `node` keyword and a variable name. This creates a basic node with an empty value, in_list, and out_list. In the second manner, a node can be declared by giving it an initial value inside parentheses after the variable name (as seen in line 2 of Listing 4). Alternatively, a declared variable can be initialized with the assignment operator "=" to any object of the type `node`.

As with all other types, graphs can be assigned at the same time as declaration or assigned later. A graph can be assigned any expression that evaluates to something of the type `graph`. There is also a special graph-creation syntax that can *only* be used at declaration time of a graph object (as seen in lines 6-9 of Listing 4). This special syntax consists of a comma-separated list of nodes and/or node operations (an example of this syntax can be found in lines 26-32 of Listing 7). Each node referenced in this type of assignment must have been previously declared.

```
node x;
node y("nyc");

graph g1;
graph g2 = g1;
graph g3 = {
   x,
   y
}; # special graph assignment syntax only available at declaration time
```

Listing 4: Declaration of "node" and "graph" objects.

3 Expressions and Operators

A compound **expression** has at least one operand and at least one operator. Operators have types including constants, variables, and functions that return values. For example:

```
3 * 4;
3 * 4 + 2;
((3 * 4) + 4) / (6*8));
```

Listing 5: demonstration of expressions and subexpressions

3.1 Number and String Operators

Operators are special symbols that are used to perform actions on operands. An operand can be a function, variable, or literal. **Numerical operators** are either arithmetic or comparative operators that operate on the `num` data type.

Arithmetic Operators
Arithmetic Operators function exactly like C with primitive data types such as literals, variables and functions. d.o.t.s supports the addition, subtraction, multiplication, division and modular divison operators. Similar to C, integer division truncates the resulting value `5 / 3 = 1`. Modular division returns the remainder of a given division. Ex. `5 % 3 = 2`

Comparative Operators
These operators relate two operands to each other. Some sample comparisons include:

```
#operators: >,<,==,!=, <=, >=,||,&&

3 == 4; # returns false
(x && 3) || z; # if x and z are both false, returns false; else returns true
(foo && x)
```

Listing 6: Declaration of "string" types.

All comparitive operators function as they do in C as long as the operands are primitive data types, variables or functions. The result of a comparison operation is a boolean.

3.2 Node Operators

The node operators outlined in Table 1 below are all binary operators which take a node object on the left-hand and right-hand sides of the operator. (Basic Form: *`node_var node_op node_var`*)

Operator	Explanation
--	Add undirected edge with no weights between the 2
-->	Add directed edge from left node to right node with no weight
--[*num*]	Add an undirected edge between 2 nodes with weight *num* in both directions
-->[*num*]	Add a directed edge from the left node to the right node with weight *num*
[*num*]--[*num*]	Add edge from left to right with the weight in the right set of brackets, and an edge from right to left with the weight in the left set of brackets
==	Returns whether the internal ids of 2 nodes match
!=	Returns whether the internal ids of 2 nodes do not match

Table 1: Node Operators

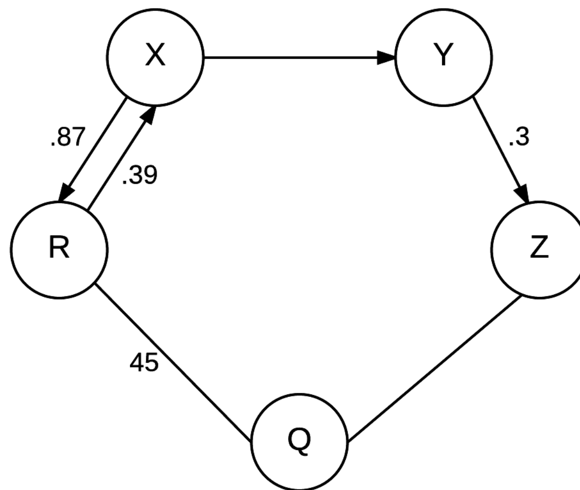

Figure 1: Example Graph showing nodes with different weights and edges.

```
node X, Y, Z, Q, R;

X --> Y;
Y -->[.3] Z;
Z -- Q;
Q --[45] R;
R [.87]--[.39] X;

R == Q; # returns false
R != Q; # returns true

\* accessing edge lists: *\
```

```
X.out[Y]; # == null
Y.out[Z]; # == .3
R.in[X]; # == .87

node alt = Z;
Y.out[alt] == Y.out[Z]; # returns true

\* Alternate Graph Creation:
 * adds the nodes to the graph G, while
 * at the same time it adds edges and weights
 * between the nodes
 *\
node x, y, z, q, r;
graph G = {
   x --> y,
   y -->[.3] z,
   z -- q,
   q --[45] r,
   r [.87]--[.39] x
};
```

Listing 7: Shows the use of node operators that creates the graph in Figure 2.

3.3 Graph Operators

Operator	Explanation
-	Returns a new graph object with the left-hand side graph without any connections dictated on the right-hand side
+	Returns a graph that contains all of the graphs in the left-hand and right-hand graph
+=	Adds the graph on the right-hand side of the operator to the graph on the left-hand side.
-=	Removes the graph on the right-hand side of the operator from the graph on the left-hand side.
==	Returns whether the two graphs contain the same nodes.

Table 2: Graph Operators

There are two logical operators: == and `!=`. The first evaluates to `true` if the left and right-hand operands are equal, while the second evaluates to `true` if the left and right-hand operands are *not* equal.

The logical operators == and `!=` are the only logical operators which function on graph operands. Two graph objects are equal if their `vertices` fields contain the same set of `node` objects.

```
node x, y, z;
graph g = {x, y}; # g.vertices: x, y
g = g + z; # g.vertices: x, y, z
g = g - z; # g.vertices: x, y

graph g2 = {x, y};
```

```
g2 == g; # return true
```

Listing 8: Shows the use of graph operators.

3.4 Subscript Notation

3.4.1 Subscript

A subscript is indicated by an id followed by an argument enclosed in brackets. The only two data types which can be subscripted are lists and dicts.

Dict objects are subscripted using a key. The notation `D[i]`, where D is a `dict` returns the value stored in the dictionary at the key i, where i is evaluated if it it's a variable. The type of the key used in a dict subscript must be of the same type as that paritcular dict object's key type.

List objects are subscripted using an index. The notation `L[i]` returns the value stored in the list at position/index *i*. *i* must evaluate to an integer.

3.5 Operator Precendence

Operator precedence follows traditional mathematical order of operations: parenthesis, exponent, multiply, divide, addition, subtraction

3.6 Member Access

The "." character, when not part of a *num* literal, allows access to members of a data type or variable

```
node x, y, z, q, r;
graph G = {
   x --> y,
   y -->[.3] z,
   z -- q,
   q --[45] r,
   r [.87]--[.39] x
};

x.in; # returns {r: 0.39}

x.in[r] # dict subscript notation; returns 0.39
```

3.7 Control Flow Statements

In regular expression terms, `if / else if* / else?` conditional statements are supported, where $\star$ represents the Kleene star operator, and "?" means 0 or 1 instances. Semantically, this means you can have `if {}` statements, `if {} else` statements, `if {} else if {} ....`, and `if {} else if {} .... else {}` statements. Within each "`{}`" is a statement (which could be the empty statment or a block of expressions, etc.).

```
if condition {
 \* execute when condition is true *\
}
else {
```

```
  \* if condition is false go here *\
}

if condition {
  \* execute when condition is true *\
}
else if another_condition {
  \* if anothe_condition is true go here *\
}
else if yet_another_condition {
  \* only reached if another_condition is false
  go here if yet_another_condition is true *\
}
else {
  \* execute when all conditions are false *\
}

if condition {
  \* execute if condition is true *\
}
\* continue executing subsequent expression statements *\
```

3.8 Loop Statements

d.o.t.s. supports `while` and `for` loops.

3.8.1 While Loops

`while` statements execute repeatedly the code in the scoped block (delimited by braces following the Boolean expression), while the boolean expression `condition` evaluates to true.

```
while condition {
  \* code *\
}
```

3.8.2 For Loops

`for` statements can only be used on the iterable types `list` and `dict`. A `for` statements iterates through all elements in `iterable_var`, assigning the current element to `var_name`. For dicts: the data type of `var_name` is automatically set to the type of the dict's keys, and the dict's keys are iterated over. For lists, the data type of `var_name` is automatically set to the type of the list, and the elements of the list are iterated over.

```
for var_name in iterable_var {
  \* code *\
}

\* common usage examples *\
for node_var in graph_var {
  \* do something with each node *\
}

```

```
for edge_var in n.out {
  \* do something with each outbound edge from node n *\
}
```

3.9 Other Statements

3.9.1 Break

Used in a loop to exit out of the loop immediately. Continue executing expressions after the loop block as expected.

```
num i = 0;
while i < 5 {
  i = i + 1;
  if i == 3 {
    break;
  }
  print(i, ' ');
}
\* Prints 1 2 *\

print(i, ' ');
\* Prints 2 *\
```

3.9.2 Continue

Used in a loop to skip the rest of the loop block and execute the next iteration of the loop.

```
num i = 0;
while i < 5 {
  i = i + 1;
  if i == 2 {
    continue;
  }
  print(i, ' ');
}
\* Prints 1 3 4 5 *\
```

3.9.3 Return

Used to exit from a function and dictate the output argument. The output argument must be the same type as specified in the function header. If not, the compiler will throw an error.

```
def num foo(num n) {
  if n > 2 {
    return 2;
  }
  return 1;
}

num out1 = foo(10)
print(out1);
```

```
10 \* Prints 2 *\
11
12 num out2 = foo(1)
13 print(out2);
14 \* Prints 1 *\
```

4 Functions

4.1 Function Declaration and Definition

Before a function can be used, it must be declared and defined. Functions are declared using the `def` keyword, followed by the data type the function will return, followed by the function name, followed by a list of parameters enclosed in parentheses. The function must then be immediately defined within a set of curly braces immediately follwing the parentheses of the parameter list.

```
1 \*
2  * Outline of function declaration and definition.
3  * ``return_type'' would be a data type.
4  *\
5 def return_type function_name () {
6   \* function implementation code *\
7 }
```

Listing 9: Function declaration and definition.

4.2 Return Statements

Each function must return a value that matches the declared return type using the `return` keyword. For functions with the `null` return type, indicating that nothing is returned by the function, the return statement can consist either of the keyword `return` as an expression by itself (line 2 of Listing 10), or it can explcitly `return null` (line 6 of Listing 10).

```
1 def null fnull1 () {
2   return;
3 }
4
5 def null fnull2 () {
6   return null;
7 }
8
9 def int fint () {
10   return 4;
11 }
```

Listing 10: Return statements of functions.

4.3 Parameter List

The declaration of a function must include a list of required parameters enclosed within parentheses. To define a function which requires no parameters, the contents of the parentheses can be left blank. Otherwise, each parameter requires the data type, followed by a variable name by which the parameter can be referenced within the function definition.

```
def null no_params () {
 return;
}

def num one_param (num x) {
 num b = x;
 return b;
}

def string multi_params (string s1, num y, string s2) {
 string statement = s1 + " " + " " + y + "s2";
 return statement;
}
```

Listing 11: Parameters in function declarations.

4.4 Calling Functions

The syntax for calling a function is: the name of the function, followed by a comma-separated list of values or variables to be used in paremter list enclosed within parentheses. Each value or variable passed in to a function call is mapped to the corresponding variable in the declared parameter list of the function.

A function-call expression is considered of the same type as its return type. Because of this, function-call expressions may be used as any other expression. For example a function-call expression can be used in the assignment of variables, as in line 11 of Listing 12.

```
def num increment (num n, num incr) {
 return n + incr;
}

num x = 4;

\* The following call maps ``x'' to the variable ``n'',
 * and ``2'' to the variable ``incr'' from the declaration
 * of the ``increment'' function
 *\
num y = increment(x, 2);

print("y: ", y); # prints --> ``y: 6''
```

Listing 12: Function declaration and definition.

4.5 Variable Length Parameter Lists

The *only* function in d.o.t.s. that can have a variable number of parameters is the built-in `print` function. All other functions must be declared with a defined absolute number of 0 or more parameters.

The `print` function may be called using a comma-separated list of expressions which can be evaluated as or converted to the `string` type. Each of the built-in types may be used directly as an argument to the print function.

```
string alpha = "World";
print("Hello", alpha, "\n");

```

```
node x("foo");
num n = 20;
print("The node <", x, "> has an associated num equal to:", n, "\n");
```

Listing 13: The built-in "print" function.

In Listing 13, the `print` function was called on line 2 with 3 arguments and with 5 arguments on line 6. The number of arguments passed to `print` does not matter.

4.6 Built-in Functions

d.o.t.s. comes with two pre-defined functions.

The first, `print(...)`, prints the string representation of a list of expressions to standard output.

The second, `range(num min, num max)`, returns a `list <num>` consisting of the integers from min to max-1. The arguments to `range` must be integers.

```
string alpha = "World";
print("Hello", alpha, "\n"); # prints "Hello World\n" to standard output

range(0, 4); # returns the list [0, 1, 2, 3]
```

Listing 14: The built-in "print" function.

5 Program Structure and Scope

5.1 Program Structure

A d.o.t.s. program consists of a series of function declarations and expressions. Because d.o.t.s. is a scripting language, there is no `main` function. Instead, expressions are executed in order from top to bottom. Functions must be declared and defined before use.

5.2 Scope

In d.o.t.s., variables declared in expressions not belonging to functions or `for` loops have scope outside of function definitions. Variables declared in a function are visible or available to be referenced by name within the body of the function (functional scoping). Variables declared in `for` loop expressions can be referenced by name within the loop body (lexical scoping). There is no true global scoping as programs are executed from top to bottom. Variable declarations within functions and `for` loops can mask variables declared outside.

```
node n("Hello world");
node x("Goodbye world");
node y("cool");

Graph nodes = {
  x,
  y
};

for node n in nodes {
  print(n, " ya ya ya\n");
```

```
12 }
13 \* prints
14  Goodbye world ya ya ya
15  cool ya ya ya*\
16
17 print(n);
18 \* prints Hello world *\
```

Listing 15: Example of masking

In Listing 15, the declaration of n in the loop scope masks the declaration of n from the top level scope.

6 Sample Program

In this section, we demonstrate how two simple path-searching algorithms can be implemented using d.o.t.s.'s syntax. The end-goal of our project is to be able to implement more complex algorithms. But for the purposes of demonstration, we chose to show Dijkstra's Algorithm (Listing 16) and the Breadth-First Search algorithm (Listing 17), as their implementation make use of each collection type and control-flow statement.

```
1 \*
2  * Djikstra's algorithm: calculates shortest paths starting
3  * from the source node and returns a dict of the lowest cost
4  * to destination nodes
5  *\
6
7 def dict<node, num> relax (node u, dict<node, num> w) {
8       for node v in u.out {
9          if w[v] > u.out[v] {
10             w[v] = u.out[v];
11          }
12       }
13       return w;
14 }
15
16 def dict<node, num> dijkstra (graph G, node source){
17    dict<node, num> S, Q;
18
19    for node n in G:
20       Q[n] = INF;
21    Q[source] = 0;
22
23    while !isEmpty(Q) {
24       node u = min(Q);
25       num w = Q[u];
26       Q.remove(u);
27       S[u] = w;
28       Q = relax(u, Q);
29    }
30    return S;
31 }
```

Listing 16: Dijkstra's Algorithm

```
1 \* Breadth-First-Search with source and dest*\
```

```
def list<node> bfs(graph G, node source, node dest) {
   list<node> visited = [source];
   dict<node, node> parents;

   node curr;
   bool found = false;
   while !isEmpty(visited) {
     curr = visited.dequeue();
     if curr == dest {
       found = true;
       break;
     }

     for node child in curr.out {
       if !(child in visited) {
         visited.enqueue(child);
         parents[child] = curr;
       }
     }
   }

   if !found {
     return [];
   }

   \* reconstruct path *\
   list<node> path = [curr];
   while (curr in parents && parents[curr] != null) {
     path.enqueue(parents[curr]);
     curr = parents[curr];
   }
   return path;
}
```

Listing 17: Breadth-First Search Algorithm

Now that we have defined the `bfs` and `dijkstra` functions, we can use them to find shortest paths in an actual instance of a graph.

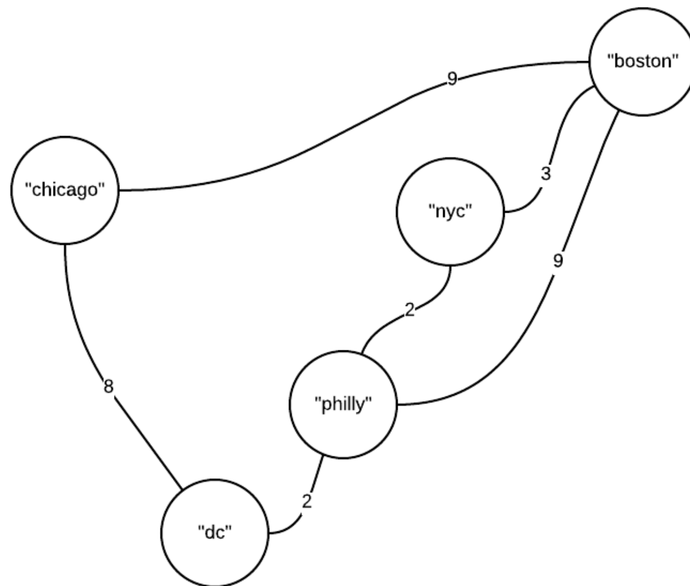

Figure 2: Visual representation of graph created in Listing 18.

```
\* Graph set-up *\
node x("dc"), y("chicago"), z("philly"), q("nyc"), r("boston");

graph g1 = {
   x --[2] z;
   z --[2] q;
   q --[3] r;
   z --[9] r;
   x --[8] y;
   y --[9] r;
};
\* end Graph set-up *\

# find the min costs from "philly" to all other cities:
dict<node, num> min_costs = dijkstra(g1, z);

# find path with minimum number of hops from "philly" to "chicago":
list<node> min_path = bfs(g1, z, y);
```

Listing 18: Using user-defined functions.