d.o.t.s.
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

1.1 Identifiers

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

1.2 Keywords

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

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

1.3 Literals

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

1.3.1 String literals

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

\\	backslash
\'	single quote
\"	double quote
\n	newline
\r	carriage return
\t	horizontal tab
\v	vertical tab

1.3.2 Number literals

Number constants consist of an optional negative sign, a 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.

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. white space is defined as any of the special characters `\r\n`, `\n`, and `\t` as well as the character obtained by pressing the spacebar. The space character sometimes delimits a separation between the parts of an expression (see the rest of the manual for syntax specifics), but otherwise whitespace is generally ignored outside of string and char expressions, as expressions are separated by the semicolon and scope is defined by braces.

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`, 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";
```

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`.

```
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 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 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 5.

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 be 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 11 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 5). Each node referenced in this type of assignment must have been previously declared.

```
1 node x;
2 node y("nyc");
3
4 graph g1;
5 graph g2 = g1;
6 graph g3 = {
7    x,
8    y
9 }; # special graph assignment syntax only available at declaration time
```

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

3 Expressions and Operators

3.1 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.

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

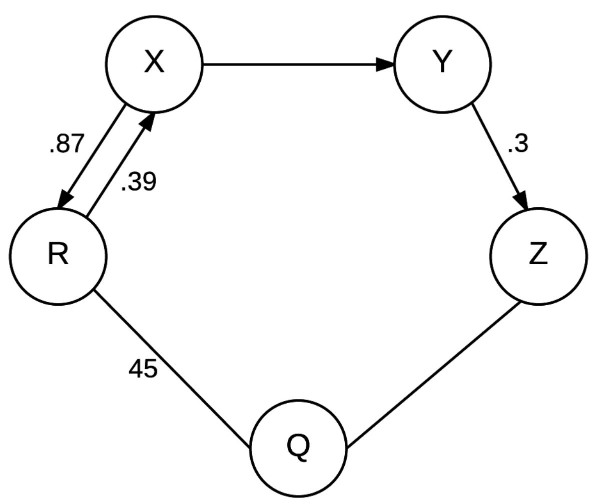

Figure 1: Example Graph showing nodes with different weights and edges.

```
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 5: Shows the use of node operators that creates the graph in Figure **??**.

3.2 Graph Operators

Hosanna's got this one.

4 Statements

4.1 Expression Statements

In d.o.t.s. expression statements are terminated by a semicolon, and are executed in the order they are read in from top to bottom. All effects from an expression statement are completed before the next one is executed.

```
\* Examples *\
node y("Hello world"); \* declaration of a node with data "Hello World" *\
num a = 5; \* declaration of number a which equals 5 *\
a = (3 + 3) / 2; \* basic arithmetic, sets a to 3 *\
print(a); \* function call, prints 3 because previous expression executed. *\
```

4.2 Control Flow Statements

`if / else if* / else` and `if` conditional statements are supported, where $\star$ represents the Kleene star operator.

```
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 *\
```

4.3 Loop Statements

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

4.3.1 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 *\
}
```

4.3.2 For Loops

`for` statements iterate through all elements in `iterable_var`, assigning the current element to `var_name`.

```
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 *\
}
```

4.4 Other Statements

4.4.1 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 *\
```

4.4.2 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 *\
```

4.4.3 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);
\* Prints 2 *\

num out2 = foo(1)
print(out2);
\* Prints 1 *\
```

5 Functions

5.1 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.

```
\*
 * Outline of function declaration and definition.
 * ``return_type'' would be a data type.
 *\
def return_type function_name () {
  \* function implementation code *\
}
```

Listing 6: Function declaration and definition.

5.2 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 7), or it can explcitly `return null` (line 6 of Listing 7).

```
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 7: Return statements of functions.

5.3 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 8: Parameters in function declarations.

5.4 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 9.

```
1  def num increment (num n, num incr) {
2    return n + incr;
3  }
4
5  num x = 4;
6
7  \* The following call maps ``x'' to the variable ``n'',
8   * and ``2'' to the variable ``incr'' from the declaration
9   * of the ``increment'' function
10  *\
11 num y = increment(x, 2);
12
13 print("y: ", y); # prints --> ``y: 6''
```

Listing 9: Function declaration and definition.

5.5 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.

```
1 string alpha = "World";
2 print("Hello", alpha, "\n");
3
4 node x("foo");
5 num n = 20;
6 print("The node <", x, "> has an associated num equal to:", n, "\n");
```

Listing 10: The built-in "print" function.

In Listing 10, 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.

6 Program Structure and Scope

6.1 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.

6.2 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");
}
\* prints
  Goodbye world ya ya ya
  cool ya ya ya*\

print(n);
\* prints Hello world *\
```

Listing 11: Example of masking

In Listing 11, the declaration of `n` in the loop scope masks the declaration of `n` from the top level scope.

7 Sample Program