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 Whitespace

In d.o.t.s. white space 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`, 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.

3 Expressions and Operators

an **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 4: demonstration of expressions and subexpressions

3.1 Number and String Operators

Definition: operators are operations that are performed on operands which can be functions, variables, or data types. **numerical operators** are operators that include arithmetic and comparative operators. *Arithmetic Operators* Arithmetic Operators function exactly like C primitive data types, variables and functions.

Comparative Operators Sample operators include

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

3 == 4;
(x && 3) || z;
(foo && x++)
```

Listing 5: Declaration of "string" types.

all operators function as they do in C as long as the operands are primitive data types, variables or functions.

4 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

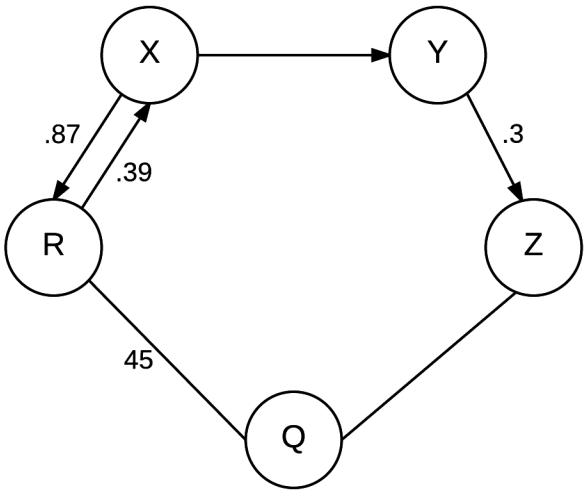

Figure 1: Example Graph showing nodes with different weights and edges.

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

4.1 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

4.2 Comparison and Logical Operators

Definition: logical operators evaluate the logical truth of two operands. These set of functions operate similarly to the logical operators

== and != are the only logical operators which function on graph operands. These operators evaluate the truth of these statements by using the Node == and != logical comparison operators. The `graph` operand will then iterate through its `nodes`. == signifies that two graphs have the same node and neighbors != means that two graph operands do not have the same nodes and neighbors

4.3 Subscript Notation

4.3.1 dict Subscript

a way of indexing specific elements within a dict the notation is "D[i]" where brackets denote the key in the dictionary you want to index

4.3.2 list Subscript

a way of indexing specific elements within a list the notation is "L[i]" where brackets denote the position in list L

4.4 Operator Precendence

follows classic programming ordering: parenthesis, exponent, multiply, divide, addition, subtraction

4.5 Member Access

"." allows you to access members of a structure or a variable

5 Statements

5.1 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. *\
```

5.2 Control Flow Statements

`if / else if* / else` and `if` conditional statements are supported, where `*` 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 *\
```

5.3 Loop Statements

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

5.3.1 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 *\
}
```

5.3.2 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 *\
}
```

5.4 Other Statements

5.4.1 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 *\
```

5.4.2 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 *\
```

5.4.3 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 *\
```

6 Functions

6.1 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 7: Function declaration and definition.

6.2 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 8), or it can explcitly `return null` (line 6 of Listing 8).

```
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 8: Return statements of functions.

6.3 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.

```
1  def null no_params () {
2    return;
3  }
4
5  def num one_param (num x) {
6    num b = x;
7    return b;
8  }
9
10 def string multi_params (string s1, num y, string s2) {
11   string statement = s1 + " " + " " + y + "s2";
12   return statement;
13 }
```

Listing 9: Parameters in function declarations.

6.4 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 10.

```
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 10: Function declaration and definition.

6.5 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 11: The built-in "print" function.

In Listing 11, 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.

7 Program Structure and Scope

7.1 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.

7.2 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 12: Example of masking

In Listing 12, the declaration of `n` in the loop scope masks the declaration of `n` from the top level scope.

8 Sample Program