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

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 Collections

2.2.1 Lists

A list is a data structure that represents zero or more elements of a particular data type.

List Declaration:

Lists are declared by using the `list` keyword and specifying the data type of the elements inside “< >”, followed by the variable name. A list can hold elements of any data type, but all elements within a single given list must have a data type matching that of the declared type.

<u>Accessing List Elements:</u>

Lists are indexed by integers and are random-access. In addition to random access, lists also have built-in member functions for queueing and dequeueing elements. The `enqueue` member function appends the given element to the end of the list. The `dequeue` member function removes the first element of the list, and shifts the remaining elements to the left.

```
list <string> str_list;

list <num> num_list = [1, 2, 5.3, 6.23];
num_list[3]; # returns 5.3
num_list.enqueue(3.2); # num_list now equals [1, 2, 5.3, 6.23, 3.2]

\* after the following:
 * x = 1; num_list = [2, 5.3, 6.23, 3.2]
 *\
num x = num_list.dequeue();
```

Listing 4: The “list” type.

2.2.2 Dicts

A dict is a data structure that represents a mapping of keys to values.

<u>Dict Declaration:</u>

Dicts are declared by using the `dict` keyword and including a comma-separated specification of the data types of the keys and values inside “< >”, followed by the variable name. The first data type listed inside “< >” corresponds to the type of the keys, while the second data type corresponds to the type of the values. The data type of dict keys can be any type *except* `list` or `dict`. There is no restriction on the data type of the values of dict objects. Once a dict has been declared, all keys must be of the same type as that declared for the key, and all values must be of the same type as that declared for the value.

<u>Accessing Dict Elements:</u>

The value of an pairing in a dict can be accessed by specifying the dict name followed by the corresponding key value enclosed in brackets, as seen in line 2 of Listing 5.

New mappings can be assigning by subsetting a dict object with a key and setting it equal to a value using the assignment operator. If the key already exists in the dict, the old value will be overwritten with the new value. If the key did not already exist, the new key will be added to the dict along with its new value.

```
1 dict <string, val> city_map = {"toronto" : 6, "queens" : 7.8, "nyc" : 0.2};
2 city_map["toronto"]; # returns 6
3
4 city_map["L.A."] = 18;
5 \* city_map = {"toronto" : 6, "queens" : 7.8, "nyc" : 0.2, "L.A." : 18} *\
```

Listing 5: The “dict” type.

3 Expressions and Operators

4 Statements

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.

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

```

```
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. 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