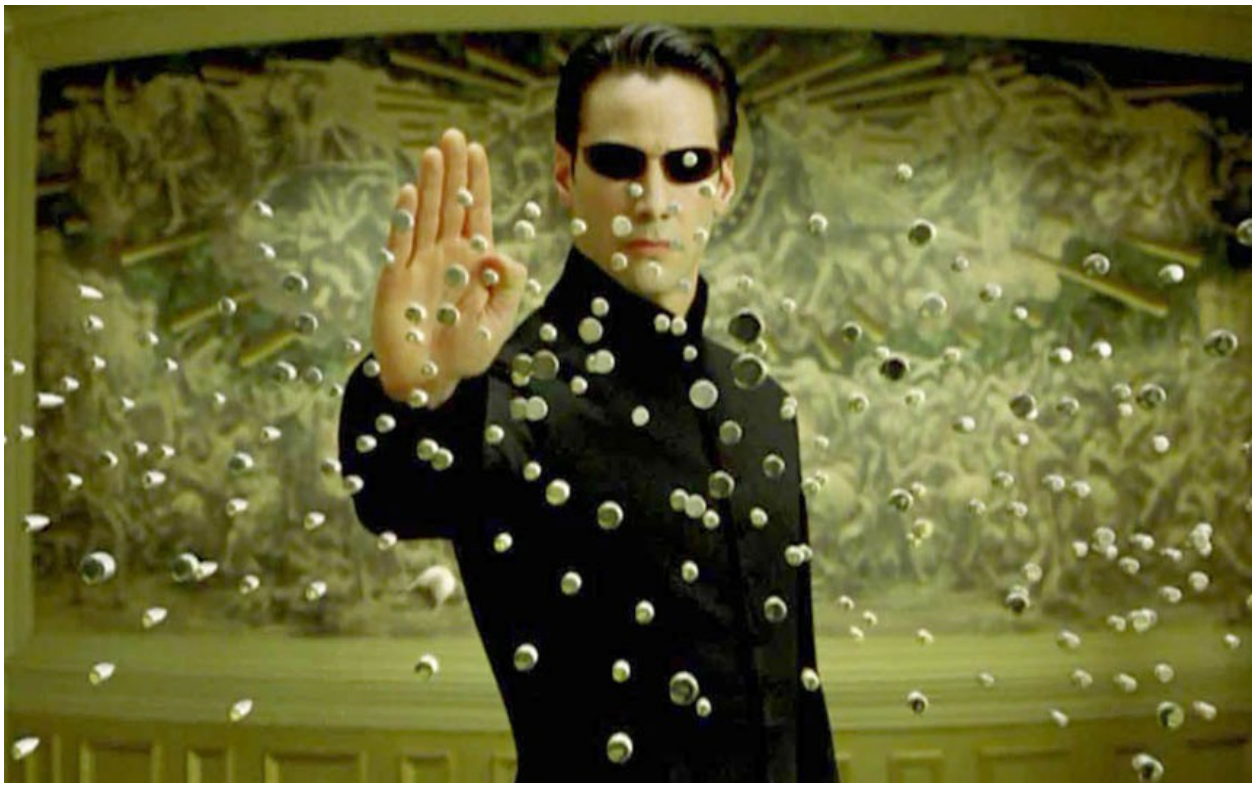
n2n reference manual



nick falba (nrf2118), bill liu (jl4347), eli aider (ea2621), johan mena (jmm2371)
Programming Languages and Translators, Fall 2014
October 27, 2014

1. Introduction

In today's world there is an escalating interest in relationships. Be it the connection of people on a social networking platform, of family members in a family tree, of variables in a mathematical equation, or the connection of trains in a city subway system, we are constantly trying to find networks of people and things and analyze how they interrelate. Most often, relationships are implemented in a programming setting via the graph data structure containing a set of nodes and edges defining connections between the nodes. In standard programming languages, however, graphs can be tedious to create and manipulate, requiring the creation of separate classes for nodes and edges, and burdening the programmer to keep track of their graphs manually. Finding and analyzing relationships between nodes of a network can be challenging.

n2n is a language that provides a higher level abstraction than procedural languages to perform common operations on graph data structures like create, read, and manipulate them through the use of relationships.

2. Syntax

2.1. Statements

A statement is anything that can make up a line of n2n code.

2.1.1. Identifier declarations

Identifiers store data values. They can either be of a primitive type or a complex type. The type of the identifier determines the meaning of the values in the identifier's storage, thus the type must be explicitly declared when specifying an identifier, as follows:

let *identifier_name*: *type*

The "let" keyword before the identifier's name brings that identifier into existence. The name must be followed by a colon, then by its type.

2.1.2. Conditional statements

Conditional statements are composed of three parts: a required "if"; zero, one, or more than one "elif" parts; and an optional "else" part.

1. i*f (expression) {*
   statements
   }

2. i*f (expression) {*
   statements
   } elif (expression) {
   statements
   } elif (expression) {
   statements
   } elif (expression) {
   statements
   }

3. i*f (expression) {*
   statements
   } elif (expression) {
   statements
   } else {
   statements
   }

Note that *elif* statements are always optional.

In the first case, the compiler will execute *statements* if the result of evaluating *expression* is the boolean value of true.

In the second case, the compiler will execute the first set of statements for which the preceding *expression* evaluates to a boolean value of true.

The third case is the same as the second case, except it will always evaluate the *statement*s inside the else part of the conditional if no other *expression* evaluate to true before that.

2.1.3. Return Statements

To return a function to its caller use the keyword *return* as follows:

return *expression*

This will return the result of evaluating *expression*.

2.1.4. Function declarations

A function is a callable piece of code that can be reused. It is declared by specifying the name, parameters, and return type of the function as follows:

fn *name* (*parameter_list*) -> *return_type* { *statements* }

parameter_list is a comma-separated list of either zero, one, or more variables of either primitive type, a complex type, or another function.

Functions may contain and return other functions within the *statements* block.

3. Expressions

Expressions are any line or lines of n2n code that can eventually be resolved to a single value. All expressions are also statements.

3.1. Identifier Assignment

Once an identifier has been declared, one can assign or reassign its value simply by using its name followed by the assignment operator, followed by an expression that evaluates to the identifier's type:

identifier_name = *expr*

One can also declare and initialize an identifier all in one line using the following syntax:

let *identifier_name*: *type* = *expr*

Where *expr* evaluates to a value of type *type.* This expressions resolves to the value of evaluating *expr.*

3.2. Function call

After a function has been declared, it can be called by placing parentheses after the name. If the function takes any arguments, those should be included and separated by commas.

fn *ohai* (*x: String*) -> *Void* { print(x) } ; function declaration
ohai("n2n!") ; function call

The value returned can be either a primitive type, a complex type, Void or another function.

3.3. Unary Operators

3.3.1. - operator

-1

Returns the negative version of an integer or a double. Only works on integers and doubles.

3.3.2. ! operator

! expression

Returns true if expression evaluates to false and false otherwise.

3.4. Binary Operators

3.4.1. Arithmetic operators

```
5 + 5
10  / expression
```

The +, -, *, /, % operators perform the corresponding arithmetic operations on its operands, which should be either integers or doubles. The % operator performs the modulus operation.

3.4.2. Relational operators

```
expression > 5
10 == a
```

The >, <, ==, !=, =>, <= operators perform numerical comparison between two numbers and return a boolean value depending on the result of the expression.

3.4.3. Logical Operators

```
first && second
first  | |  second
```

n2n supports two logical operators: && (or “and” operator) and || or (“or” operator). These operate on two expressions that can be evaluate to a boolean value (*first* and *second*, in the example above). The && operator returns true if both *first* and *second* evaluate to true, false otherwise. The || operator returns false if either *first* or *second* return false, and it returns true if either *first* or *second* return true.

3.4.4. Concatenation

```
“hello ” ^ “troll” ; “hello troll”
```

Strings can be concatenated using the ^ operator. This operator returns a new string in which the string on the right is appended to the string on the left.

4. Types

Identifiers in n2n are labeled by type. The identifier's type determines how the compiler interprets the data in the identifier's storage. The identifier lifetime depends upon the scope in which the identifier is defined. A function sets a scope for a variable so that any variables declared in a function only exist within that function and are discarded on return. Any variable defined outside of a function definition is global in scope and is not discarded until the end of the program. If a local variable shares its name with a global variable, the local variable (within the function) takes precedence.

4.1. Identifier types

4.1.1 Primitive types

- Integers are represented in 32-bit 2's complement representation
- Doubles are represented in 64-bit according to the IEEE standard
- Booleans are represented as either true or false, which in n2n, cannot be reduced.
- Strings are a sequence of ASCII characters chosen from a

4.1.2 Complex types: Data, Graph, Node, Rel

- *Data* is a collection of named primitive types intended to hold a group of related values (like C's structs or Ocaml's types) and is used to represent information contained in a *Node* or *Rel.* Examples:

  Defining a Data type:

  ```
  let data_example:Data = {
          let bool_field:Bool
          let string_filed:String
  }
  ```

  Data literals can then be made for that specific type and stored in a node. Example of the above would be:

  data_example{ bool_field = true, string_field = "test" }

- *Nodes* are the building blocks of graphs. Each node contains a set of named fields, either primtive or Data, describing the attributes of what the node represents. They can be instantiated and bound to an id as follows:

  let node1:Node = [data_example{ bool_field = true, string_field = "text" }, ...]

The ellipsis (...) above indicates that many Data types or primitive types can be stored in a node. The above is an example of a node being declared and instantiated explicitly. A node can also be declared and instantiated implicitly using the Rel and Graph instantiation syntax (see below). Node literals have the form:

[data_object{...}, data_object2{...}, primitive_type1:field_type,...]

- *Rel* is the representation of a directed edge between graph nodes. Directed means that the relationship from node1 and node2 might be different than the relationship from node2 to node1. Rels can contain one piece of information, either a single Data type or a single primitive type. This single piece of information is named.

  Rel literals have the form:

  [source_node, rel_data, destination_node]

  So a rel is defined by two nodes and a directed edge containing data about the relationship from source_node to destination_node. If source_node or destination_node are not valid in the scope where they're being referenced, empty nodes are created and bound to those identifiers. This is an example of a node being declared implicitly. Rels can be directly instantiated and bound to an identifier with the syntax:

  let rel_example:Rel = [source_node, rel_data, destination_node]

  n2n supports multiple relationships between nodes.

- *Graph* is the main data type in the language and is represented in memory as an adjacency matrix. Graph is a collection of nodes and the data representing the relationships (Rels, above) between them.

  To fill a graph, zero or more rel and/or node objects, as defined above, are declared within curly brackets following the graph's assignment operator. Node and Rel values that are to be inserted into a graph may exist prior to the assignment of the *graph* in which they are placed or they can be assigned within the *graph's* declaration.

  For example:

  let graph1:Graph = {
  let Rel_name1: Rel = [existing_node1, rel1_data, existing_node2]

```
    let Rel_name2: Rel =[node3, rel2_data, node4]
   existing_node_5
}
```

Or:

```
let graph2:Graph = {
        let Rel_name1: Rel =[[node_data_object_name{ data_field =
        expr }], rel_data{ rel_data_field = expr }, [node2_data_obj_name{
        data_field = expr }]]
        let node5:Node = [data_type{data_type_field1 = value, …}]
}
```

In the first example, existing_node1 and existing_node2 are nodes that have been declared and instantiated explicitly as shown above. node3 and node4 are being implcitly declared and instantiated as empty nodes. These node types can be referenced later with the identifiers node3 and node4, within the same scope as the graph is being defined. existing_node_5 is added to the graph but has no relationships coming from it. It is added as an orphan node, where relationships can be added later.

For graph2, it consists of only one relationship and one node, explicitly created and instantiated as well as added to the graph and bound to identifiers using the assignment operator (see Expressions below), which can be used to modify these objects later within the same scope as the graph that is being instantiated.

4.1.3 Lists

- *List* maintains an ordered collection of elements and is used in our language for iterating over a graph's nodes, relationships, or a subset of these. Lists are immutable in n2n.

5. Lexical conventions

An n2n program, written in the ASCII character set consists of one or more graph objects, the nodes it contains, the complex relationships between those nodes, and/or functions that manipulate or make use of the graph. It contains various types of tokens such as expression operators, keywords, built-in function names, identifiers, and other separators. All identifiers, keywords, and functions names are case sensitive. In general blanks, tabs, and comments are ignored except as they serve to separate tokens. New lines indicate the end of a statement.

5.1 Comments

Comments are introduced and terminated by the character ";":

; This is a comment ;

5.2 Identifiers

An identifier is a sequence of letters and digits with no spaces between them. The first character must be alphabetic. Underscore characters may be included in the identifier. Convention is that multi-word identifiers are separated by underscores ('_') to improve readability.

5.3 Keywords

The following are reserved for use as keywords and may not be used otherwise:

Int	if	false
String	else	return
Bool	elif	each
Double	let	filter
Graph	fn	in
Node	Void	map
Rel	Null	reduce
Data	true	find_many

5.4 Constants

5.4.1 An integer constant is a sequence of digits. A minus sign ('-') indicates the integer has a negative value. n2n requires decimal integer values.

12
-1
1024

5.4.2 A floating constant consists of an integer part, a decimal point, and a fraction part. Either the integer part or the fraction part but not both may be missing. Both are comprised of a series of digits.

1.3
.2
1.

5.4.3 String constants are one or more ASCII characters surrounded by double quotes. Certain characters may be escaped within a string as follows:

“ \n” -> “ \\n “
“ \t “-> “ \\t “
“ \\\\ “ -> “ \\“
“ \” “ -> “ “ “

6. Built-in Functions

6.1. Graph and Node Operators

An existing node or relationship can be added to a Graph using the ^+ operator as in: *Graph_name* ^+ *Node_name/Rel_name*. Similarly, the ^- operator is used to delete a node in a Graph.

For Graph_name ^- Rel_name, it just removes the particular relationship from the "Graph".

Whereas for Graph_name ^- Node_name, it removes the node and all the relationships associated with it.

To add or remove Data from a Node, use the [+] and [-] operators respectively. For adding data, set the field values within curly braces.

```
let node_example: Node = [ let data: data_type ]

let data_to_add: Data = {
        let field1: data_type
}
```

node_example [+] *data_to_add{field1 = value}*

;

The result after insertion would be:

```
node_example = [ data, data_to_add{ field1 } ]
```

;

Data deletion is just deleting a spacific data from the node object, for example:

```
node_example [-] data
```

;

The result after deletion would be:

```
node_example = [ data_to_add{ field1 } ]
```

6.2. The *find_many* Function

The *find_many* method takes in a set of criteria and returns a list of *Node* objects associated with the specified criteria. The criteria may be any of the following: (For clarification purposes, examples are given below from the following set of data: There are Nodes with Data types *movies* and *persons* and Rel of type "*acted_in*" and instantiation Node named *Amber*)

```
let actor: Data = {
        let name: String
        let age: Int
}

let Amber: Node = [actor{ age = 27 }]
```

```
let Keanu: Node = [actor{ name = “Keanu”, age = 27}]

let acted_in: Data = {
        let role: String = “Nancy”
}

let movie1: Node = [ let movie_name: String = “Matrix” ]
let movie2: Node = [ let movie_name: String = “Matrix” ]

 let graph: Graph = {
     [Amber acted_In movie1]
     [Keanu acted_In{role =”Neo”} movie1]
     movie2
 }
```

- find_many(node_data_literal): This returns a list of all nodes with fields that match the given node_data_literal. Some examples:

```
find_many( [actor]) ; returns a list containing Amber and Keanu nodes;
find_many( [actor{ age = 27}]) ; returns the same list as above;
find_many( [actor{name = “Keanu”}]) ; returns a list containing Keanu ;
```

- find_many(node_data_literal, rel_data): This returns a list of nodes that are pointed to by the nodes that match node_data_literal with relationship rel_data.

```
find_many( [actor{name = “Keanu”}], acted_in{role = “Matrix”}) ; returns movie1.
```

- find_many(node_data_literal1, rel_data, node_data_literal2): This returns a list of nodes that are defined by node_data_literal1 pointing to node_data_literal2, with relationship rel_data.

ex: returns all *persons* that have *acted_in* any *movie*

```
find_many( [actor], acted_In, [movie_name] ) ; returns Amber and Keanu nodes ;
```

For the node_literals, you could also put in a named_node variable. In these cases, the set of nodes that match these arguments would be one, the node that the name is bound to. The function then tries to return only those nodes that match the rel_data pattern from that specific node. An example:

```
find_many(Keanu, acted_in{role = “Neo”}) ; returns movie1;
find_many(Amber, acted_in) ;returns movie1;
find_many([actor], acted_in, movie1) ;returns Amber and Neo;
```

7. Standard Library functions

7.1. each(collection, function) applies ‘function’ to every element in ‘collection’ and returns the original ‘collection’

7.2. map(collection, function) returns a list of the results of running ‘function’ on each element of ‘collection’

7.3. reduce(collection, function, init) combines all elements of ‘collection’ by applying ‘function’ to two elements of the collection at a time, starting from an initial value of ‘init’

7.4. filter(collection, predicate) yields every element in the collection to a predicate and returns a new list of the elements for which predicate evaluates to true

8. Program Execution

8.1. Start symbol

The symbol “$” indicates the start point of an n2n program. All statements after the dollar sign will be executed sequentially until the end of file is reached.

```
$
statement 1
statement 2
statement 3
```