COMS W4115 Language Reference Manual

Team 11

3/21/12

Abdullah Al-Syed, aza2105

Jenee Benjamin, jlb2229

Patrick De La Garza, pmd2130

Donald Pomeroy, dep2127

Varun Ravishankar, vr2263

Contents

Part I
Program Definition

The program consists of import statements, node definitions, which contain function and variable definitions, the main consists of code not within the body of a node definition. The main method is defined as any code (with the exception of import statements) outside of node definitions, it does not necessarily have to be written at the bottom of the file. However, it is good style to put all of the main code at the end of the file following the node definitions. Yet, if there are import statements, they must be provided before any other any code.

The basic structure of the program is as follows:

```
<<import statements>>
<<node definitions>>
<<main method>>
```

Part II
Lexical Conventions

1 Whitespace

Tokens must be separated by whitespace, which can include spaces or tabs. Newlines act as separators for statements, so they cannot generally be used as whitespace.

2 Comments

is a single line comment.
/* is the beginning of a multi-line comment.
*/ is the end of the multi-line comment.

```
Ex: # this is a single line comment
/*this is a
multi-line comment*/
```

3 Identifiers

Identifiers consist of strings of numbers, letters, underscores. The first character of an identifier cannot be a number.

4 Reserved Words

The following words are reserved:

Keywords
true
false
from
import
node
end
cond
public
private
do
while
for
loop
in
until
not
or
xor
break
continue
is (equality)
assert
unless
if
else
mod
null
some
none

The following are reserved but have no function in the language. They may be added in the future.

Reserved Keywords
fork
def
goto
try
catch
finally
with
lambda

5 Types and Literals

All variables in Tandem are dynamically typed. The programmer does not need to specify types when he or she uses a variable, but does need to have a value assigned to a variable before it can be used. Use of variables that have not been declared will throw an error.

5.1 Integer

An integer, or int, is declared as follows: an identifier followed by a single equals sign, followed by zero or more digits.

Example:

```
a = 42
```

5.2 Double

A double is declared as follows: an identifier followed by a single equals sign, followed by zero or more digits, or the same followed by a single decimal point . followed by one or more digits.

Example:

```
a = 45.01
```

5.3 Boolean

A boolean is declared as follows, an identifier followed by a single equals sign followed by *true* or $false$. All values in Tandem are true, except for $false$ and *null*, which represents an undefined value. You should not use *null* to represent empty values; future versions of Tandem will introduce a *none* type to represent empty values.

Example:

```
a = true
```

5.4 Strings

A string is declared as follows, an identifier followed by a single equals sign followed by one double quotation mark followed by zero or more characters followed by a single double quotation mark.

Example:

```
a = "hello world"
```

Strings supports most of the Java escape sequences, and a few other special escape sequences.

Escape Sequence	Use
\x	Equivalent to the character x by itself, unless x is a line terminator or one of the special characters $abefnrstv$. This syntax is useful to escape the special meaning of the \,#, ′, and ” characters.
\t	The TAB character (ASCII code 9).
\s	The Space character (ASCII code 32).
\a	The BEL character (ASCII code 7). Rings the console bell.
\b	The Backspace character (ASCII code 8).
\e	The ESC character (ASCII code 27).
\n	The Newline character (ASCII code 10).
\r	The Carriage Return character (ASCII code 13).
\f	The Form Feed character (ASCII code 12).
\v	The vertical tab character (ASCII code 11).

5.5 List

A list is declared as follows, an identifier followed by a single equals sign followed by a square bracket, followed by any number of literals separated by commas.

Example:

```
a=[1,2,3,4,5]
```

5.6 Set

A set is declared as follows, an identifier followed by a single equals sign followed by a curly bracket , followed by any number of unique literals separated by commas, followed by a curly bracket.

Example:

```
a = {1,2,3,4,5}
```

5.7 Hash

A hash is declared as follows, an identifier followed by a single equals sign followed by a curly bracket, followed by any number of unique literals followed by a fat comma (=>), followed by any literal, each separated commas.

Example:

a = {1=>2, 2=>3, 4=>5}

5.8 Binary Literal

A binary is declared as follows, an identifier followed by a single equals sign followed a 0, the letter b, and one or more 0s or 1s.

Example:

a = 0b10010

5.9 Hexadecimal Literal

A hexadecimal is declared as follows, an identifier followed by a single equals sign followed a 0, followed an X, followed by one or more of these character, $0, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f$ (case insensitive to the letters).

Example:

a = 0xFFFFFFFF

6 Operators

6.1 Operators, associativities, and precedences

Operator(s)	Associativity	Operation
{...}	N	Hash, set, and hash and set comprehensions
[...]	N	List, list comprehension
(...)	L	expression
x.*attr*	L	attribute reference
x[*i*]	L	Indexing
|	L	Pipeline operator
**	R	Exponentiation
!	R	bitwise complement; Boolean NOT
−+	R	unary minus; unary plus
* / %	L	Multiplication; division (true), modulo
+−	L	Addition (or concatenation); subtraction, set difference
<<>>	L	Bitwise shift-left; bitwise shift-right
/\	L	Bitwise AND, set intersection
^	L	Bitwise XOR, set symmetric difference
\/	L	Bitwise OR, set union
<<= >= >	L	magnitude comparison, set subset and superset, value equality operators
==! =	N	Equality testing, comparison
&&	L	Boolean AND
||	L	.. N Range creation (inclusive)
x..y	N	Range creation (inclusive)
= ** = * = / = % = + = − = <<= >>= && = || = \/ = /\ = ^ =	R	Assignment
x *mod* y	L	Modulo (low precedence)
x *is* y, x *is not* y	N	Identity tests
x *in* y, x *not in* y	N	Membership
not x	R	Boolean NOT (low precedence)
x *and* y	L	Boolean AND (low precedence)
x *xor* y	L	Boolean XOR (low precedence)
x *or* y	L	Boolean OR (low precedence)

6.2 Mathematical Operators

6.2.1 Plus Operator (+)

The plus operator is a binary operator that can perform addition on doubles. On strings and lists, the plus operator acts as concatenation. If the identifier on one side of the plus expression is a string the type on the other side is automatically converted to a string and they are concatenated.

Example:

```
a = 45 + 45.1 #a is 90.1
a = hello + world # a is hello world
a = 45 + hello # a is 45hello
```

6.2.2 Minus Operator (−)

The minus operator is a binary operator that can perform subtraction on doubles. On sets the minus operator performs the set difference operation.

Example:

```
a = 45.1 45 # a is .1
a = [1,2,3,4,5] [1,2,3] # a is [4,5]
```

6.2.3 Multiplication Operator (∗)

The multiplication operator is a binary operator that can perform multiplication on numbers (doubles, binary, hex). On strings and lists, this performs a repetition of elements.

Example:

```
a = 45*2 # a = 90
"hello" * 3 # "hellohellohello"
```

6.2.4 Division Operator (/)

The division operator is a binary operator that can perform division on numbers.

Example:

```
a = 45/2 # a is 22.5
```

6.2.5 Modulus Operator (mod, %)

This operator is a binary operator that has 2 symbols, mod and %, which perform the same function. It performs the modulus function on doubles, using the mathematical style of Ruby, allowing for non-integer values.

Example:

```
a = 3.1%3.0 # a is .1
```

6.2.6 Exponentiation Operator (**)

This operator is a binary operator that raises the left hand to the right hand power, on numbers.

Example:

```
a = 2 ** 3 # a is 2 to the 3rd power, 8
```

6.3 Bitwise Operators

6.3.1 Bitwise complement ( )

This operator performs two's complement on doubles, hexadecimals, and bytes.

Example:

```
a = ~0xFFFFFF
```

6.3.2 Bitwise and (/\)

This operator is a binary operator that performs bitwise and on all numeric types. On sets, this performs the set intersection operation.

Example:

```
a = 0xFFFFF /\ 0xAAAAA
a = [1,2,3,4] /\ [3,4,5,6]
```

6.3.3 Bitwise xor (^)

This operator is a binary operator that performs bitwise xor on all numeric types. On sets, this performs the set symmetric difference operation.

Example:

```
a = 0xFFFFF ^ 0xAAAAA
a = {1,2,3,4} ^ {3,4,5,6}
```

6.3.4 Bitwise or (\/)

This operator is a binary operator that performs bitwise or on all numeric types. On sets, this performs the set union operations.

Example:

```
a = 0xFFFFF \/ 0xAAAAA
a = [1,2,3,4] \/ {3,4,5,6}
```

6.3.5 Bitwise shift-left (<<)

This operator shift-lefts doubles, hexadecimals, and bytes, discarding the bits shifted out and shifting in zeros on the right. The operand on the right determines how many bits are shifted.

Example:

```
a = 0xFFFFFF << 1
```

6.3.6 Bitwise shift-right (>>):

This operator shift right doubles, hexadecimals, and bytes the sign bit is shifted in on the left, thus preserving the sign of the operand. Further on, while shifting right, the empty spaces will be filled up with a copy of the most significant bit (MSB). The operand on the right determines how many bits are shifted.

Example:

```
a = 0xFFFFFF >> 1
```

6.4 Logical Operators

6.4.1 Boolean and (&&, *and*)

And will evaluate to true only when both the left operand and the right operand are true. There is a difference in precedence between the *and* keyword and the && symbol; *and* has a lower precedence.

Example:

```
a = true
b = true
a && b #evaluates to true
```

6.4.2 Boolean xor (*xor*)

Xor will to true only when one and only one of the operands on either side of the expression is true.

Example:

```
a = true
b = false
a xor b #evaluates to true
```

6.4.3 Boolean or (*or*, ||)

Or evaluates to true when at least one of the operands on either side of the expression is true. There is a difference in precedence between the keyword *or* and the || symbol; *or* has a lower precedence.

Example:

```
a = true
b = false
a or b #evaluates to true
```

6.4.4 Equals (== , *is*)

Equals returns true when the value of the left operand is equal to the value of the right operand. For numbers, it is mathematical equality, for strings, if all the characters are the same. There is a difference in precedence between the keyword is and the == symbol: *is* has a lower precedence.

Example:

```
a = 1
b = 1
a == b # evaluates to true
```

6.4.5 Not equals (*is not*, ! =)

Not equals returns when the value of the right operand is not equal to the value of the left operand. For numbers, it is mathematical inequality, for strings, if all the characters are not the same. There is a difference in precedence between the keyword *is not* and the ! = symbol.

Not (*not*, !) : Not *null* is true, not $false$ is true, and the complement of everything else is false.

Example:

```
a = 1
!a # evaluates to false
b = null
not b # evaluates to true
```

6.5 Assignment Operators

6.5.1 Equals (=)

The equals operator is a assignment operator that sets the value of the identifier on the left side to the value of the identifier on the right side. When an indexing operator is used, this can set elements with a list or a hash.

Example:

```
a = 4 # a is 4
b = 5 # b is 5
a = b # a is now 5
```

6.5.2 Exponentiation Equals (** =)

The exponentiation equals is an assignment operator that sets the value of the identifier on the left side to the value of the identifier on the left side raised to the value of the identifier on the right.

Example:

```
a **= b # equivalent to a = a**b
```

6.5.3 Divide Equals (/ =)

The divides equals is an assignment operator that sets the value of the identifier on the left side to the value of the identifier on the left side divided by the value of the identifier on the right.

Example:

```
a /= b # equivalent to a = a/b
```

6.5.4 Modulus Equals (% =)

The modulus equals is an assignment operator that sets the value of the identifier on the left side to the value of the identifier on the left side modulus the value of the identifier on the right.

Example:

```
a %= b # equivalent to a = a%b
```

6.5.5 Plus Equals (+ =)

The plus equals is an assignment operator that sets the value of the identifier on the left side to the value of the identifier on the left side plus the value of the identifier on the right.

Example:

```
a += b # equivalent to a = a+b
```

6.5.6 Minus Equals (− =)

The minus equals is an assignment operator that sets the value of the identifier on the left side to the value of the identifier on the left side minus the value of the identifier on the right.

Example:

```
a -= b # equivalent to a = a-b
```

6.5.7 Bitwise left shift Equals (<<=)

The bitwise left shift equals is an assignment operator that sets the value of the identifier on the left side to the value of the operator on the right side left shifted.

Example:

```
a <<= b # equivalent to a = <<b
```

6.5.8 Bitwise right shift Equals (>>=)

The bitwise right shift equals is an assignment operator that sets the value of the identifier on the left side to the value of the operator on the right side right shifted.

Example:

```
a >>= b # equivalent to a = >>b
```

6.5.9 And equals (&& =)

The 'and-equals' is an assignment operator that sets the value of the identifier on the left side to the value of the identifier on the left anded with the the value of the identifier on the right.

Example:

```
a &&= b # equivalent to a = a&&b
```

6.5.10 Or equals (|| =)

The or equals is an assignment operator that sets the value of the identifier on the left side to the value of the identifier on the left or'ed with the the value of the identifier on the right.

Example:

```
a ||= b # equivalent to a = a||b
```

6.5.11 Bitwise or equals (\/ =)

The bitwise or equals is an assignment operator that sets the value of the identifier on the left side to the value of the identifier on the left bitwise or'ed with the the value of the identifier on the right.

Example:

```
a \/= b # equivalent to a = a\/b
```

6.5.12 Bitwise and equals (/\ =)

The bitwise and equals is an assignment operator that sets the value of the identifier on the left side to the value of the identifier on the left bitwise and'ed with the the value of the identifier on the right.

Example:

```
a /\= b # equivalent to a = a/\b
```

6.5.13 Bitwise xor equals (^ =)

The bitwise xor equals is an assignment operator that sets the value of the identifier on the left side to the value of the identifier on the left bitwise xor'ed with the the value of the identifier on the right.

Example:

```
a ^= b # equivalent to a = a^b
```

6.6 Relational Operators

6.6.1 Less than (<)

The less than sign is a relational operator that returns true if the left operand is smaller than the right operand, for numbers, and if the left operand is lexicographically before the right operand for strings. Otherwise, the expression returns false. On sets, this checks if the left operand is a true subset of the right operand: that is, $set < other$ means $set <= other$ and $set != other$.

6.6.2 Less than or equal to (<=)

The less than or equal to sign is a relational operator that returns true if the left operand is smaller than or equal to the right operand for numbers, or if the left operand is lexicographically before or equal to the right operand for strings. Otherwise, the expression returns false. On sets, this checks if every element in the left operand is in the right operand.

6.6.3 Greater than (>)

The greater than sign is a relational operator that returns true if the left operand is greater than the right operand, for numbers, and if the left operand is lexicographically after the right operand for strings. Otherwise, the expression returns false. On sets, this checks if the left operand is a true superset of the right operand: that is, $set > other$ means $set >= other$ and $set != other$.

6.6.4 Greater than or equal to (>=)

The greater than or equal to sign is a relational operator that returns true if the left operand is greater than or equal to the right operand for numbers, or if the left operand is lexicographically after or equal to the right operand for strings. Otherwise, the expression returns false. On sets, this checks if every element in the right operand is in the left operand.

6.7 Indexing

For the index of list, a positive index from 0 to the arrays size - 1, inclusive, returns the corresponding element of the list. Negative values are also allowed, and indices go from -size to -1. Anything outside of these ranges returns a null value. Indexing starts from 0. Index values are truncated to integers. To assign to an element in the list, the list identifier followed by a square bracketed index is followed by an assignment statement. The list can be extended by passing a positive number larger than the size as the bracketed index in the assignment. The list will be padded with null values.

Example:

```
a = [1,2,3,4]
a[2] = 1 # a is now [1,2,1,4]
a[-1] # returns 4
# a[-5] is an error
```

For hashes, indexing is done by the key. If the key exists in the hash, the hash returns the corresponding value. Otherwise, the hash returns *null*.

Example:

```
b = {"hello" => 1, "world" => 2}
b["hello"] => 1
b[0] # returns null

b["goodbye"] = 3

# b is now {"hello" => 1, "goodbye" => 3, "world" => 2}.
# Notice hashes do not have a guaranteed ordering.
```

Sets do not have an index, because there is no guaranteed position that each element in the set has. You can still iterate through a set, however, and can save each element in a list, at which point you can access the elements by their indices.

Part III
Scoping

7 Variable Scoping

Variables declared inside of a node are not accessible to the code within the function definitions. We are using lexical scoping, so the code within a sub-node is not accessible to the body of the primary node, and the code within the node is not accessible to the main.

Example:

```
node n()
        x = 3
        y = 2

        node f()
                x = 5 #a different x variable
                return x #returns 5
        end

        return x /* returns 3, if the return was missing
                                the return would be the value of y */
end
```

Example:

```
private node n(x)
        public node f(a) #This is not valid syntax
                a+1
        end

        node g(b) # This is assumed to be private
                b+1
        end

        f x | g /* returns the result of the node g acting
                                on the result of f acting on x */
end
```

8 Node Scoping

To access a node in another file the import declaration must be used. To access sub nodes of a node, the dot operator is used. Private nodes cannot be accessed. Nodes must be declared before they are accessed.

Example:

```
from "sort.td" import Quicksort, Mergesort.implementation1
/* imports the implementation1 node from the Mergsort node,
        contained in the file sort.td */

from "sort.td" import *
/* The above imports all public nodes and their public
        sub-nodes from the file sort.td */
```

Part IV
Type Conversions

9 Numeric and String Conversions

Converting an int to a double is valid.
Example:

```
a = 45.1
b = 45
b = a + b # Valid assignment; b = 90.1
```

Converting an int or double to a string is valid.
Example:

```
a = 45.1
b =
b = a + b # b is a string containing 45.1
```

10 Boolean Conversions

Converting a boolean to a string is valid.
Example:

```
a = true
b =
b = a + b #b is a string reading true
```

Converting a string to a boolean is only valid if the string literal is 'true' or 'false' (case insensitive).
Example:

```
a = true
b = false
b = a # b is now true
```

Part V
Statements and Expressions

11 Statements

The two types of statements are Node Statements and Main Statements.

11.1 Node Statements

A node is the basic unit of the language. Nodes can contain expressions and nodes. A node is declared using the node keyword followed by comma separated parameters in parentheses. The node code ends with an 'end' keyword, the grammar determines which 'end' keyword within the node is the proper closing *end* keyword. If no return is explicitly declared, the last expression before the closing end is considered the return value.

Example:

```
node node1(a, b, c)
        if a < 2
                return x = 5
        else

        end #this end closes the if else expression

        if b > 2
                return y = 1 # declares a variable and returns it
        else

        end

        node inner_node(d) # declaring an inner node
                return d+1
        end #ending an inner node

        z = inner_node c /*c is being passed a parameter to inner_node
        /* if the returns are not specified, z is the return value, implicitly.
           The last non-node line's value is the default return value. */
end # this is the end that closes node1
```

11.2 Main Statements

Main statements can be Loop Statements, Assignment Statements, or Expressions.

11.2.1 Loop Statements

There are for-loops, while-loops, do-while-loops, Until-loops, and generic loops.

The *break* keyword is used to break out of the current loop. *continue* is used to proceed to the next iteration. Every loop is ended by the keyword *end*.

Generic Loops: Generic loops are done as follows:

```
loop
        # do things
        # possibly (hopefully) break under some condition
end
```

For-loops: For loops are used to loop through all the items of a Set, List, Hash, or Range.

Example:

```
for item in group
        # do stuff here
end
```

While-Loops: While loops are used to loop as long as condition holds. Example:

```
while condition
        # do stuff
end
```

A do-while-loop is shown below:

```
do
        #do things
while condition
end
```

Until-Loops: Until-loops loop until a condition is met:

```
Until condition
        # do stuff
end
```

12 Expressions

Expressions are statements that return a value. There are conditional expressions, pipeline expressions, and the expressions containing the operators in Section 6.

12.1 Pipeline Expressions

A pipeline is an expression that contains a node id followed by space separated parameters followed by the pipeline (|) symbol, followed by the any number of node ids separated by pipelines. Also, you can have a pipeline of one node, consisting of a single node that is followed by space separated parameters. As a matter of good style and to avoid any errors, just follow the simple rule:

NO NODE CODE IN THE PIPELINE!

By this we mean that you do not put conditionals and expressions in the pipeline, only nodes, literals, ids for basic types. Also note that a value piped from one node to another is a copy of the return value of the preceding node so as to preserve thread safety.

Example:

```
node1 1 2 3 | node2 | node3 /* This means the return of node1
taking parameters 1,2, and 3 will be given as parameters to node2,
the results of node2 will be given as parameters to node3 */

a = 5
node1 a 2 3 | node2 # good

a = 5
node1 a+1 2 3 | node2 # Error NO NODE CODE IN THE PIPELINE!
node1 if a < 6 | node2 else | node3 # Error NO NODE CODE IN THE PIPELINE!
```

12.2 Conditional Expressions

Conditional expressions are used to evaluated different statements and return a value based on some condition(s). The three conditional expression types are *cond*, *if*, and *unless*. Conditionals return the last line executed unless the *return* keyword is specified.

12.2.1 If-expressions

If-expressions evaluate a certain portion of code if a certain condition is met; if the condition is not met, it will evaluate the other. If expressions must have an else portion.

Example:

```
if condition
        # do stuff
else
        # do some other stuff
end
```

12.2.2 Unless-expressions

Unless-expressions will evaluate a block of code unless conditions is met:

```
unless condition
        # do stuff
end
```

12.2.3 Cond-expressions

Cond-expressions evaluate blocks of code given their respective conditions hold. If a condition is met, its block of code is executed and the other conditions are skipped.

Example:

```
cond
        condition1
                # Execute this code and return if condition1 is met
                #do not deal with the other two
        end

        condition2
                #return this one
        end

        condition3
                #return this one
        end
end
```

Part VI
Imports and System Functions

13 Import Syntax

Imports allow you to access nodes form other files or libraries. You can import every node in a file in one of these two ways:

```
import filename
from filename import *
```

To import specific nodes do the following:
from filename import specific1, specific2

14 System Functions

The system functions (nodes) are print, read, and write. Additionally, math and random nodes are provided.

14.1 Print:

The print node takes a value and prints it to standard out. Example:

```
print 1 # this prints 1
print 'gamma' # this prints the string 'gamma'
list | QuickSort | print # prints a sorted list
```

14.2 Read

Read reads in a file and tokenizes it by line by default. The only parameter is the file name.

Example:

```
book = read('Starship Troopers.txt') # reads in the text

for line in book
        #do something with the information
end
```

14.3 Write

Write takes a file name (does not have to be existent yet) for the first parameter, 'w' (indicating write) or 'a' (indicating append) as the second element, and an array of lines to add as the last element.

Example:

```
book = read('Starship Troopers.txt') # read it in
book2 = read('War of the Worlds.txt')# read the other in

write('Starship Troopers Backup.txt', 'w', book) # back up Starship Troopers
write('War of the Worlds Backup.txt', 'w', book2) # back up Starship Troopers
write('War of the Worlds.txt', 'a', book ) # append Starship Troopers to War of
```

14.4 Math Functions

There are many math System functions included (refer to the Ruby documentation): (Math.abs, Math.sqrt, Math.log, Math.pi, Math.inf, Math.e, Math.sin, Math.cos, Math.tan, Math.asin, Math.acos, Math.atan, Math.sinh, Math.cosh, Math.tanh, Math.gamma, Math.floor, Math.ceil).

14.5 Random Functions

A Random node is provided that returns a random integer within some provided range, can return a double between 0 and 1, or can pick an element from a sequence (list, hash, set, string, or range) at random.

Part VII
Future Language Extensions

In the future, extensions will be added to Tandem to provide first-class functions, forking nondeterministically, add exceptions, complex numbers, comprehensions, and add option types.

15 First-class functions

This feature would allow users to pass nodes as parameters and to create anonymous functions. Users would be able to use higher-order functions to create functions that return other functions, and these functions could possibly close over the variables in scope at the time to allow for closures. This would make recursive functions more powerful.

16 Fork

Fork will work exactly like cond, except all of the conditions that evaluate to true will result in a new thread being spawned and the corresponding pipeline being run. This will allow for nondeterministic code. While there are always thread safety issues, all data passed between nodes is immutable, and therefore should not cause invalid reference errors, or be altered unexpectedly.

17 Exceptions

This feature would add an exceptions system, as well as provide for functionality with the *try*, *catch*, and *finally* as done in Java. Users could create exception nodes, throw and catch exceptions, and then guarantee code to be run after the exception is handled. Also, a *with* keyword would be provided for, as in Python, that would automatically close up files after reading or writing so users would not have to do all reading and writing within a *try/catch* statement.

18 Complex Numbers

This feature would add complex number literals to the language, allowing for a consistent syntax for most mathematical operations.

19 Comprehensions

This feature would add list, set, and hash comprehensions. A microsyntax would be created to allow users to pick elements that satisfy some condition, such as in the following:

```
squares = [x**2 for x in 0..9]

# equivalent to the following
squares = []
for x in 0..9
        squares += x**2
end

# squares is now [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

20 Option and Empty Types

This would allow for *none* and *some*, to indicate whether an element was empty, or if there was some element there. This would allow for the disambiguation between the usage of *null* to represent undefined values and to represent an empty value.

Part VIII
Appendix

A Language Grammar

$$
\begin{array}{rcl}
S & \rightarrow & I\ M \mid M \\
I & \rightarrow & Imp\ I \mid Imp \\
 & & \\
Imp & \rightarrow & from\ F\ import\ Modules \mid import\ F \\
F & \rightarrow & filename \\
Modules & \rightarrow & '*' \mid NID \mid NID\ Modules \\
NID & \rightarrow & id \mid id'.'NID \\
 & & \\
M & \rightarrow & Body \mid \epsilon \\
Body & \rightarrow & Statement\ Body \mid Statement \\
 & & \\
Statement & \rightarrow & node\ id\ '('\ Params\ ')'\ M\ end \mid Expression \\
 & & \mid Assignment \mid LoopType \mid return\ Expression \\
 & & \mid assert\ Expression \\
Params & \rightarrow & ParamList \mid \epsilon \\
ParamList & \rightarrow & id \mid id\ ','\ ParamList \\
Assignment & \rightarrow & IndexAssign\ '='\ Expression \mid IndexAssign\ '+='\ Expression \\
 & & \mid IndexAssign\ '-='\ Expression \mid IndexAssign\ '*='\ Expression \\
 & & \mid IndexAssign\ '/='\ Expression \mid IndexAssign\ '\%='\ Expression \\
 & & \mid IndexAssign\ '**='\ Expression \mid IndexAssign\ '>>='\ Expression \\
 & & \mid IndexAssign\ '<<='\ Expression \mid IndexAssign\ '\hat{}='\ Expression \\
 & & \mid IndexAssign\ '/\backslash='\ Expression \mid IndexAssign\ '\backslash/='\ Expression \\
 & & \mid IndexAssign\ '\&\&='\ Expression \mid IndexAssign\ '||='\ Expression \\
IndexAssign & \rightarrow & Indexed \mid id \\
LoopType & \rightarrow & for\ id\ in\ Iterable\ LM\ end \mid while\ Expression\ LM\ end \\
 & & \mid do\ LM\ while\ Expression\ end \mid loop\ LM\ end \\
 & & \mid until\ Expression\ LM\ end \\
Iterable & \rightarrow & List \mid Set \mid Hash \mid String \mid Range \\
CondType & \rightarrow & if\ Expression\ M\ else\ M\ end \mid unless\ Expression\ M\ end \\
 & & \mid cond\ CBody\ end \mid fork\ Cbody\ end \\
CBody & \rightarrow & Cstatements \mid \epsilon
\end{array}
$$

$$
\begin{aligned}
Cstatements &\rightarrow Csentence \mid Csentence\ Cstatements \\
Csentence &\rightarrow Expression\ M\ end \\
LM &\rightarrow Lbody \mid \epsilon \\
Lbody &\rightarrow Lstatement\ Lbody \mid Lstatement \\
Lstatement &\rightarrow node\ id\ '('\ Params\ ')'\ M\ end \mid Expression \mid Assignment \\
&\quad \mid return\ Expression \mid LoopType \mid assert\ Expression \mid Break \mid continue \\
Break &\rightarrow break \mid break\ Expression \\
\\
Expression &\rightarrow ID \mid CondType \mid Expression\ Binop\ Expression \mid Unary\ Expression \mid Literal \\
&\quad \mid Indexed \mid Attr \mid '('\ Expression\ ')' \\
ID &\rightarrow id \mid id\ NParams \mid id\ Pipeline \mid id\ NParams\ Pipeline \\
NParams &\rightarrow Literal \mid id \mid Literal\ NParams \mid id\ NParams \\
Pipeline &\rightarrow '|'\ id \mid\ '|'\ id\ Pipeline \\
\\
Unary &\rightarrow '+' \mid '-' \mid '\sim' \mid ! \mid not \\
Binop &\rightarrow MathOp \mid BitwiseOp \mid RelOp \mid LogicOp \\
MathOp &\rightarrow '+' \mid '-' \mid '*' \mid '/' \mid '\%' \mid \text{mod} \mid '**' \\
BitwiseOp &\rightarrow '<<' \mid '>>' \mid '/\backslash' \mid '\backslash/' \mid '\sim' \mid '\hat{}' \\
RelOp &\rightarrow '<' \mid '<=' \mid '>=' \mid '>' \\
LogicOp &\rightarrow '\&\&' \mid '||' \mid is \mid is\ not \mid in \mid not\ in \mid '==' \mid '!=' \\
&\quad \mid and \mid xor \mid or \\
Literal &\rightarrow AlphaNum \mid List \mid Hash \mid Set \mid Range \mid true \mid false \mid null \mid none \\
Indexed &\rightarrow id\ Indexer \mid Hash\ Indexer \mid List\ Indexer \\
Indexer &\rightarrow '['\ Key\ ']' \mid '['\ Key\ ']'\ Indexer \\
Attr &\rightarrow id\ '.'\ Attributes \\
Attributes &\rightarrow id \mid id\ '.'\ Attribtues \\
List &\rightarrow '['\ Innards\ ']' \\
Set &\rightarrow '\{'\ Literal\ '\}' \\
Innards &\rightarrow Literal\ ','\ Innards \mid Literal \mid id\ ','\ Innards \mid id \\
Hash &\rightarrow '\{'\ Hinnards\ '\}' \\
Hinnards &\rightarrow Key\ '=>'\ Hvalue \mid Key\ '=>'\ Hvalue\ ','\ Hinnards \\
Key &\rightarrow id \mid Literal \\
AlphaNum &\rightarrow Numeric \mid String \mid ByteLiteral \mid HexLiteral \\
Range &\rightarrow Numeric\ RangeTail \\
RangeTail &\rightarrow '..'\ Numeric
\end{aligned}
$$