iota

Small but powerful … kind of

11 December 2020

Team Genesis

Shuhan Dong

Sean Kohler

Joseph McDonough

Ryan Pepe

Table of Contents

ChangeLog

20 October - Completed “Overview” section - Shuhan, Sean, Joseph, Ryan

14 November- Completed “Formal Syntax” section - Sean, Joseph, Ryan

18 November - Created top, interp, lang, data-structures files. Made some updates to grammar section including arbno and separated-lists - Sean, Joseph, Ryan

22 November - Made some updates to grammar and added intDiv, and, or, not operations. Have working code for const-exp, true-exp, false-exp, and-exp, or-exp, not-exp, sub-exp, div-exp, intDiv-exp, and isZero?-exp - Sean, Joseph, Ryan

23 November - Created test cases for working functions - Ryan

23 November - Completed semantics for all the functions that are or have planned implementation with exceptions for while, let, and proc - Joseph

3 December - Implemented statements and the store file from simple-statements. Added functionality to let-exp, while-exp, if-exp, if-else-exp - Sean, Joseph, Ryan

5 December - Fixed environments, store, data-structures, and interp such that while loops and let-exps can work. Added “test cases” to top.scm for later addition. Added proc-exp and call-exp such that users can create functions - Joseph

6 December - Added remaining relational operators as well as more test cases. Added strings. Added missing grammars in this document and adjusted the ones that were changed in the program so they match - Joseph

Overview

Purpose

iota is a language intended for entry-level usage and academic purposes on the Internet of Things (IoT). Ultimately designed to introduce students to the interfaces necessary for IoT devices to work and communicate amongst each other. iota is inspired by JavaScript and Java, such that its syntax resembles both in certain aspects. Similar to those languages it relates to, it has a functional paradigm.

Value Types

The subsequent value types are equipped in iota:

- Number
- String
- List
- Boolean
- Function

Control Mechanisms

The lexical binding of new variables is allowed through the use of `let` and existing environment variables can be modified with `set`

- e.g. `let` _x = ...
- e.g. `set` _x = ...

Branching in iota is handled through one or more

- e.g. `if` Cond `...` `elif` Cond `...` `else` `...`

Flow control and iterative control can be achieved through the use of loops, via `while` expressions

- e.g. `while` Cond ...

Single functions are permitted in iota. Anonymous functions can be denoted through proc expressions and the => symbol. Functions do have the ability to be called from inside one another. There is the ability to curry functions to provide a multi-argument-esque feature.

- e.g. `proc x` => {...}

Operators & Built-In Functions

Numerous operations and built in functions are built-in to the iota language

Number Operands

- Subtraction (`sub`)
- Division (`div`)
- Integer Division (`intDiv`)

Number Relational Operators

- Less Than (<)
- Greater Than (>)
- Less Than or Equal To (<=)
- Greater Than or Equal To (>=)
- Equal To (==)
- Not Equal To (`!=`)
- Check if Zero (`isZero?`)

Boolean Operators

- and (`&&`)
- or (`||`)
- not (!)
- Check if Null (`isNull?`)

List Operations

- list creation (`[]`)
- first element (`car`)
- remaining elements (`cdr`)

Type System

To avoid errors in run-time, iota provides type safety prior to run-time. That is type errors would be caught and handled before the program could run and disregard the rules of this language. As a result, variables cannot change on their own during execution.

Formal Syntax

Lexical Specification

The symbol *Space* is understood to be whitespace.

Digit ::= [0-9]

RadixPoint ::= .

Number ::= *Digit+* | *Digit+ RadixPoint Digit+*

LowerAlpha ::= a | b | … | z

UpperAlpha ::= A | B | … | Z

Alpha ::= *LowerAlpha* | *UpperAlpha*

Char ::= *LowerAlpha* | *UpperAlpha* | *Number* | *Space* | *Punctuation*

String ::= ‘#*Char**’

List ::= car | cdr

Boolean ::= [True , False]

Keyword ::= let | set | while | if | iff | elif | else | True | False | log | nlog

Special ::= , | (|) | [|] | { | } | => | ; | *RadixPoint*

Punctuation ::= *Space* | . | ! | ? | : | ; | , | (|) | - | _ | [|] | & | | | = | < | > | / | { | } | ~ | # |

VarChar ::= Alpha | *Digit*

NameVar ::= VarChar+

OpVar ::= + | - | * | / | // | < | <= | > | >= | == | !=

Var::= _ *NameVar*

Grammatical Specification

Program ::= { *Statement* }

a-program (stmt)

Statement ::= iff *Expression* { *Statement* } elif *Expression* { *Statement* }*

if-stmt(exp1 stmt1 exps* stmts*)

::= if *Expression* { *Statement* } elif *Expression* { *Statement* }* else {*Statement* }

if-else-stmt (exp1 stmt1 exps* stmts* stmt2)

::= set *Var* = *Expression* ;

set-stmt (var exp1)

::= let *Var*$^{+(,)}$ in *Statement*

let-stmt(vars$^+$ stmt1)

::= while *Expression* { *Statement* }

while-stmt (exp1 stmt1)

::= log *Expression*

log-stmt(exp1)

::= nlog *Expression*

newLog-stmt(exp1)

::= { *Statement*$^{+(;)}$ }

multi-stmt (stmts$^+$)

Expression ::= Literal

::= [Literal$^{+(,)}$]

list-exp(exp)

::= emptyList

emptyList-exp()

::= sub (*Expression* , *Expression*)

sub-exp(rand1 rand2)

::= div (*Expression* , *Expression*)

div-exp (rand1 rand2)

::= intDiv (*Expression* , *Expression*)

intDiv-exp (rand1 rand2)

::= isZero? (*Expression*)

zero?-exp (rand)

::= isNull? (*Expression*)

null-exp(rand)

::= &&(*Expression, Expression*)

and-exp (exp1 exp2)

::= ||(*Expression, Expression*)

or-exp (exp1 exp2)

::= !(*Expression*)

not-exp (exp1)

::= <(*Expression*)

lessThan-exp(rand1 rand2)

::= <=(*Expression*)

lessThanEqual-exp(rand1 rand2)

::= >(*Expression*)

greaterThan-exp(rand1 rand2)

::= >=(*Expression*)

greaterThanEqual-exp(rand1 rand2)

::= ==(*Expression*)

equalTo-exp(rand1 rand2)

::= !=(*Expression*)

notEqualTo-exp(rand1 rand2)

::= *Var* => { *Expression* }

proc-exp(var body)

::= -> *Expression* (*Expression*)

call-exp (rator rand)

Literal ::= *Number*

const-exp (num)

::= *Var*

var-exp (var)

::= ‘ str ‘

str-exp (str)

::= True

true-exp()

::= False

false-exp()

Operational Semantics

Variables & Expressed Values

const-exp

$$\frac{}{\text{(value-of (const-exp } \texttt{num}\text{) } \textit{env}\text{) = (num-val } \texttt{num}\text{)}}$$

---

var-exp

$$\frac{}{\text{(value-of (var-exp } \texttt{var}\text{) } \textit{env}\text{) = (deref (}\textit{env}\text{) } \texttt{var}\text{)}}$$

---

str-exp

$$\frac{}{\text{(value-of (str-exp } \texttt{str}\text{) } \textit{env}\text{) = (string-val } \texttt{var}\text{)}}$$

---

true-exp

$$\frac{}{\text{(value-of (true-exp } \texttt{bool}\text{) } \textit{env}\text{) = (bool-val } \texttt{\#t}\text{)}}$$

---

false-exp

$$\frac{}{\text{(value-of (false-exp } \texttt{bool}\text{) } \textit{env}\text{) = (bool-val } \texttt{\#f}\text{)}}$$

---

emptyList-exp

$$\frac{}{\text{(value-of (emptyList-exp) } \textit{env}\text{) = (list-val } \texttt{'()}\text{)}}$$

Mathematical Operations

sub-exp

(value-of `rand1` *env*) = (num-val `num1`)
(value-of `rand2` *env*) = (num-val `num2`)
(value-of `(- rand1, rand2)))= val1`

(value-of (sub-exp `rand1 rand2`) *env*) = `val1`

---

div-exp

(value-of `rand1` *env*) = (num-val `rand1`)
(value-of `rand2` *env*) = (num-val `rand2`)
(value-of `(/ rand1, rand2))= val1`

(value-of (div-exp `rand1 rand2`) *env*) = `val1`

---

intDiv-exp

(value-of `rand1` *env*) = (num-val `rand1`)
(value-of `rand2` *env*) = (num-val `rand2`)
(value-of (`floor (/ rand1, rand2)))= val1`

(value-of (intDiv-exp `rand1 rand2`) *env*) = `val1`

Boolean Operations

isZero?-exp

(value-of `rand` *env*) = (num-val `num`)

(value-of (isZero?-exp `rand`) *env*) = (bool-val (= `num` 0))

---

isNull?-exp

(value-of `rand` *env*) = (list-val `lst`)

(value-of (isNull?-exp `rand`) *env*) = (bool-val (`null? lst`))

---

and-exp

(value-of `exp1` *env*) = (bool-val `bool1`)
(value-of `exp2` *env*) = (bool-val `bool2`)
(`bool1 = #t`)
(`bool2 = #t`)

(value-of (and-exp `exp1 exp2`) *env*) = (bool-val `#t`)

(value-of `exp1` *env*) = (bool-val `bool1`)
(value-of `exp2` *env*) = (bool-val `bool2`)
(`bool1 = #f`)

(value-of (and-exp `exp1 exp2`) *env*) = (bool-val `#f`)

(value-of `exp1` *env*) = (bool-val `bool1`)
(value-of `exp2` *env*) = (bool-val `bool2`)
(`bool2 = #f`)

(value-of (and-exp `exp1 exp2`) *env*) = (bool-val `#f`)

---

or-exp

(value-of `exp1` *env*) = (bool-val `bool1`)
(value-of `exp2` *env*) = (bool-val `bool2`)
(`bool1 = #t`)

(value-of (or-exp `exp1 exp2`) *env*) = (bool-val `#t`)

(value-of `exp1` *env*) = (bool-val `bool1`)
(value-of `exp2` *env*) = (bool-val `bool2`)
(`bool2 = #t`)

(value-of (or-exp `exp1` `exp2`) *env*) = (bool-val `#t`)

(value-of `exp1` *env*) = (bool-val `bool1`)
(value-of `exp2` *env*) = (bool-val `bool2`)
(`bool1 = #f`)
(`bool2 = #f`)

(value-of (or-exp `exp1` `exp2`) *env*) = (bool-val `#f`)

---

not-exp

(value-of `exp1` *env*) = (bool-val `bool1`)
(`bool1 = #t`)

(value-of (not-exp `exp1`) *env*) = (bool-val `#f`)

(value-of `exp1` *env*) = (bool-val `bool1`)
(`bool1 = #f`)

(value-of (not-exp `exp1`) *env*) = (bool-val `#t`)

---

lessThan-exp

(value-of `rand1` *env*) = (num-val `num1`)
(value-of `rand2` *env*) = (num-val `num2`)
(< `num1 num2`)

(value-of (lessThan-exp `rand1 rand2`) *env*) = (bool-val `#t`)

(value-of `rand1` *env*) = (num-val `num1`)
(value-of `rand2` *env*) = (num-val `num2`)
(`>= num1 num2`)

$$\text{(value-of (lessThan-exp } \texttt{rand1 rand2}) \ \mathit{env}) = \text{(bool-val } \texttt{\#f})$$

---

lessThanEqual-exp

$$\frac{\begin{array}{l}\text{(value-of } \texttt{rand1} \ \mathit{env}) = \text{(num-val } \texttt{num1})\\ \text{(value-of } \texttt{rand2} \ \mathit{env}) = \text{(num-val } \texttt{num2})\\ (\texttt{<= num1 num2})\end{array}}{\text{(value-of (lessThanEqual-exp } \texttt{rand1 rand2}) \ \mathit{env}) = \text{(bool-val } \texttt{\#t})}$$

$$\frac{\begin{array}{l}\text{(value-of } \texttt{rand1} \ \mathit{env}) = \text{(num-val } \texttt{num1})\\ \text{(value-of } \texttt{rand2} \ \mathit{env}) = \text{(num-val } \texttt{num2})\\ (\texttt{> num1 num2})\end{array}}{\text{(value-of (lessThanEqual-exp } \texttt{rand1 rand2}) \ \mathit{env}) = \text{(bool-val } \texttt{\#f})}$$

---

greaterThan-exp

$$\frac{\begin{array}{l}\text{(value-of } \texttt{rand1} \ \mathit{env}) = \text{(num-val } \texttt{num1})\\ \text{(value-of } \texttt{rand2} \ \mathit{env}) = \text{(num-val } \texttt{num2})\\ (\texttt{> num1 num2})\end{array}}{\text{(value-of (greaterThan-exp } \texttt{rand1 rand2}) \ \mathit{env}) = \text{(bool-val } \texttt{\#t})}$$

$$\frac{\begin{array}{l}\text{(value-of } \texttt{rand1} \ \mathit{env}) = \text{(num-val } \texttt{num1})\\ \text{(value-of } \texttt{rand2} \ \mathit{env}) = \text{(num-val } \texttt{num2})\\ (\texttt{<= num1 num2})\end{array}}{\text{(value-of (greaterThan-exp } \texttt{rand1 rand2}) \ \mathit{env}) = \text{(bool-val } \texttt{\#f})}$$

---

greaterThanEqual-exp

$$\frac{\begin{array}{l}\text{(value-of } \texttt{rand1} \ \mathit{env}) = \text{(num-val } \texttt{num1})\\ \text{(value-of } \texttt{rand2} \ \mathit{env}) = \text{(num-val } \texttt{num2})\\ (\texttt{>= num1 num2})\end{array}}{\text{(value-of (greaterThanEqual-exp } \texttt{rand1 rand2}) \ \mathit{env}) = \text{(bool-val } \texttt{\#t})}$$

$$\begin{array}{l}\text{(value-of } \texttt{rand1} \ \mathit{env}) = \text{(num-val } \texttt{num1})\\ \text{(value-of } \texttt{rand2} \ \mathit{env}) = \text{(num-val } \texttt{num2})\end{array}$$

(< num1 num2)

(value-of (greaterThanEqual-exp rand1 rand2) *env*) = (bool-val #f)

---

equalTo-exp

(value-of rand1 *env*) = (num-val num1)
(value-of rand2 *env*) = (num-val num2)
(eq? num1 num2)

(value-of (equalTo-exp rand1 rand2) *env*) = (bool-val #t)

(value-of rand1 *env*) = (num-val num1)
(value-of rand2 *env*) = (num-val num2)
(eq? num1 num2)

(value-of (equalTo-exp rand1 rand2) *env*) = (bool-val #f)

---

notEqualTo-exp

(value-of rand1 *env*) = (num-val num1)
(value-of rand2 *env*) = (num-val num2)
(eq? num1 num2)

(value-of (notEqualTo-exp rand1 rand2) *env*) = (bool-val #t)

(value-of rand1 *env*) = (num-val num1)
(value-of rand2 *env*) = (num-val num2)
(eq? num1 num2)

(value-of (notEqualTo-exp rand1 rand2) *env*) = (bool-val #f)

Conditional Control Operations

if-stmt

(value-of `exp1` *env*) = (bool-val `#t`)
(result-of `stmt1` *env*) = `val1`

(result-of (if-stmt `exp1 stmt1 exps* stmts*`) *env*) = `val1`

(value-of `exp1` *env*) = (bool-val `#f`)
(value-of cdr(`exps*`)) = `val2`
(value-of(null? `val2`)) = (bool-val `#f`)
(value-of cdr(`stmts*`)) = `val3`

(result-of (if-stmt `exp1 stmt1 exps* stmts*`) *env*) =
    (value-of (if-exp car(`val2`) car(`val3`) cdr (`val2`) cdr (`val3`)) *env*)

(value-of `exp1` *env*) = (bool-val `#f`)
(value-of (cdr(`exps*`)) = `val2`
(value-of(null? `val2`)) = (bool-val `#t`)

---

if-else-stmt

(value-of `exp1` *env*) = (bool-val `#t`)
(result-of `stmt1` *env*) = `val1`

(result-of (if-else-stmt `exp1 stmt1 exps* stmts* stmt2`) *env*) = `val1`

(value-of `exp1` *env*) = (bool-val `#f`)
(value-of cdr(`exps*`)) = `val2`
(value-of(null? `val2`)) = (bool-val `#f`)
(value-of cdr(`stmts*`)) = `val3`

(result-of (if-else-stmt `exp1 stmt1 exps* stmts* stmt2`) *env*) =
    (result-of (if-else-stmt car(`val2`) car(`val3`) cdr(`val2`) cdr (`val3`) `stmt2`) *env*)

(value-of `exp1` *env*) = (bool-val `#f)`
(value-of (cdr(`exps*)`) = `val2`
(value-of(null? `val2))` = (bool-val `#t`)
(result-of `stmt2)` = `val3`

(result-of (if-else-stmt `exp1 stmt1 exps* stmts* stmt2`) *env*) = `val3`

---

while-stmt

(value-of `exp1` *env* `sto0`) = (bool-val `#f`, `sto1`)

(result-of (while-stmt `exp1 stmt1`) *env* `sto0`) = `sto1`

(value-of `exp1` *env* `sto0`) = (bool-val `#t`, `sto1`)
(result-of body env `sto1`) = `sto2`

(result-of (while-stmt `exp1 stmt1`) *env* `sto0`) = (result-of (while-stmt `exp1`
`stmt1`) *env* `sto2`)

Miscellaneous Functions

set-stmt

(value-of `exp1` *env*) = `val1`
(apply-env `var val1` *env*) = *env*

(result-of (set-stmt `var exp1`) *env*) = `val2`

---

let-stmt

(value-of `(car vars`$^+$`)` *env*) = undefined-val
(apply-env `var val1` *env*) = *env*
(value-of(null? `(cdr vars`$^+$`))`) = (bool-val `#f`)

(result-of (let-stmt `vars`$^+$ `stmt1`) *env*) =
(result-of (let-stmt (`cdr vars`$^+$`)` `stmt1`) *env*)

(value-of `(car vars`$^+$`)` *env*) = undefined-val
(apply-env `var val1` *env*) = *env*
(value-of(null? `(cdr vars`$^+$`))`) = (bool-val `#t`)
(result-of (let-stmt `vars`$^+$ `stmt1`) *env*) = `val2`

(result-of (let-stmt `vars`$^+$ `stmt1`) *env*) = `val2`

---

proc-exp

(value-of `var` *env*) = `val1`
(value-of `body` *env*) = `val2`

(value-of (proc-exp `var body`) *env*) = (proc-val = `val1`, `val2`)

---

call-exp

(value-of `rator` *env*) = (proc-val `val1`)
(value-of `rand` *env*) = `val2`

(value-of (call-exp `rator rand`) *env*) = (apply-procedure `val1 val2`)

---

log-stmt

(value-of (log-stmt `exp1`) *env*) = `val1`

(result-of (log-stmt `exp1`) *env*) = (display `val1)`

---

nlog-stmt

(value-of (newLog-stmt `exp1`) *env*) = `val1`

(result-of (newLog-stmt `exp1`) *env*) = (newline)(display `val1)`

---

multi-stmt

(result-of (multi-stmt `(`car `stmts))` *env*) = `val1`
(value-of cdr(`stmts))` = `val2`
(value-of(null? `val2))` = (bool-val `#f`)
(extend-env `val1` *env*) = *env2*

(result-of (multi-stmt `stmts`) *env*) = (result-of (multi-stmt `(`cdr `stmts))` *env2*)

(result-of (multi-stmt `(`car `stmts))` *env*) = `val1`
(value-of cdr(`stmts))` = `val2`
(value-of(null? `val2))` = (bool-val `#t`)

(result-of (multi-stmt `stmts`) *env*) = *env2*

---

list-exp

(value-of (list-exp exp) *env*) = `val1`
(value-of (list-exp (cdr `exp))` *env*) = (list-val `val2)`

(value-of (list-exp `exp`) *env*) = (list-val (cons `val1 val2))`

Appendix: Sample Programs

```
{
 let _a, _b, _c, _d in
     {
          set _a = 3;
          set _b = 5;
          set _c = 0;
          while !(isZero?(_b))
           {
             {
               iff >(_a, _b)
                {
                  set _c = sub(_c, -1)
                 };
               set _b = sub(_b,1)
             }
            };
          set _d =
              proc _z =>
                       {
                         <=(_z, 3)
                       };
          if -> _d (_c)
           {
              log '# _a is within 3 digits of _b'
           }
          else
           {
              log '# _a was not within 3 digits of _b'
           }
       }
}

Result → # _a is within 3 digits of _b
```

```
{
 {
   set _x = 3;
   while !(isZero?(_x))
    {
      {
        log _x;                                  Result -> 321
        set _x = sub(_x,-1)
      }
    }
 }
}
```

Boolean Expressions and Operations

```
{
if isNull?([1,2])
   { log 1 }
elif isNull?([1])                    Result -> -1
   { log 0 }
else
   { log -1 }
}
```

```
{
iff isNull?([1,2])
    { log 5 }
elif isNull?(emptyList)              Result -> 0
    { log 0 };
}
```

```
{
if isNull?([1,2])
     { log 1 }
elif <(5, 3)
     { log 3 }
elif isZero?(4)                      Result -> 5
     { log 4 }
elif isZero?(0)
     { log 5 }
elif ==(1,sub(3,2))
     { log 6 }
else
     { log 7 }
}
```

```
{
log isZero?(9)                       Result -> #f
}
```

```
{
log &&(True, isZero?(0))             Result -> #t
}
```

```
{
log ||(isZero?(1), isZero?(0))        Result -> #t
}
```

```
{
log !(isZero(0))                      Result -> #f
}
```

Simple Arithmetic Expressions and Operations

```
{
log 11                                 Result -> 11
}
```

```
{
log -33                                Result -> -33
}
```

```
{
log sub(10,2)                          Result -> 8
}
```

```
{
log div(10,2)                          Result -> 5
}
```

```
{
log intDiv(10,3)                       Result -> 3
}
```

Let, Set, And Proc Expressions and Operations

```
{
  let _a, _b, _c in
   {
     set _a = 6;
     set _b = 4;                                    Result -> 6
     set _c = 13;
     log intDiv(_c, sub(_a, _b))}
   }
}
```

```
{
  {
    set _i =
        proc _y =>                                  Result -> 35
           { sub(_y,5)};
    log -> _i(40)
   }
}
```

```
{
  {
   set _i =
      proc _y =>
            { sub(_y,5)};                           Result -> 1/10
   set _v =
       proc _z =>
             { proc _a => { div(-> _i(_z),_a)} };
   log -> -> _v(6)(10)
  }
}
```

```
{
  {
     set _i =
         proc _y =>
              { sub(_y,5)};
     set _v =
         proc _z =>
              { div(-> _i(25),_z)};      Result -> 10
     log -> _v(2)
  }
}
```