# Data Structures in Io

Everett Boyer

April 20, 2017

## 1 Introduction

This paper reports on a Masters project that implements linked lists, binary trees, and important manipulations of these data structures in the Io language, which at first glance is unsuited to building data structures. Io was first created by Raphael Levien in 1989 with the expressed purpose to create a language with the "simplest practical programming language notation possible." [1] Martin Sandin created the Amalthea compiler for Io in 2003, based on the description of the Levien Paper in *Advanced Programming Language Design*. [2]. This paper contains a description of the Io language itself, linked lists and binary trees and their implementation in Io, the usage of Io as a programmer's first language, and ends with a discussion of other avenues of exploration for the Io language.

## 2 Io

Raphael Levien designed the Io notation with a single mechanism in mind, "the action". [1] An invocation is comprised of an operator and parameters. [1] An operator is the name of an action, and parameters can be either actions, integers or strings. Generally one of these parameters is an action to indicate the sequence of execution. Levien describes such an action parameter as something always given as the last parameter or surrounded by parentheses. [1] Finkel, however, describes the final action parameter as a continuation, "which represent the remainder of the computation to be performed after the called procedure is finished with its other work." [2]

These parameters are described as continuations for the rest of this paper. Passing a continuation as a parameter is not required, but it is necessary if the programmer wishes the program to continue processing after performing an action. The continuation indicates the sequence of actions the program should take.

```
1 write_int 20;
2 terminate
```

In this example, the operator `write_int` has two actual parameters: an integer and a continuation. The continuation parameter `terminate` is performed by `write_int` after it finishes printing the number 20. `terminate` is a predefined action that takes no parameters and does nothing. Because `terminate` does not take any parameters, it invokes no further action, so the program ends.

It is also possible to define new operators in Io, with new actions assigned to them. An example of a newly defined operator and its usage follows.

```
1 declare sampleAction: -> X Continuation;
2         write_int X;
3         Continuation.
4
5 sampleAction 2;
6 terminate
```

The action definition introduces several new conventions. `declare`, which is one of Io's only reserved words, signals the start of an action definition. `sampleAction` is the name of the action. Actions are invoked with their names and behave as a goto with parameters. In Io, actions are defined as `-><formal parameters>; body`. Line 5 invokes `sampleAction` with two actual parameters, the integer `2` and the continuation `terminate`. These actual parameters are bound to the formal parameters `X` and `Continuation`. This action invokes it's body which consists of an invocation of `write_int` with two additional actual parameters, `X` that is bound to `2` and `Continuation` that is bound to `terminate`. `write_int` then outputs 2 to standard output, and invokes its second parameter, which does nothing terminating the program.

The programmer can get the effect of storing data by holding information in formal parameters. In this example below and several others, the

arrow indicating an action is altered to include a number. `+ 4 5 -> X;` is the proper syntax, but `+ 4 5 -1->` is used for ease of reference.

```
1 + 4 5 -1-> X;
2 write_int X;
3 terminate
```

The addition operator takes three parameters: two integers and an action. The action parameter is filled by the anonymous action `->X; write_int X; terminate`, which has a single formal parameter `X`. The addition action passes `9` as the actual parameter anonymous action #1. action #1 one binds the actual parameter `9` to the formal parameter `X`. The `write_int` invocation then takes the value of `X` as its first actual parameter, which it writes to standard output, and then `write_int` invokes `terminate`. The syntax of anonymous actions looks intentionally similar to an assignment statement, so a naive reader can interpret this example as three independent statements.

Sequencing actions by means of continuations can only be useful if there is a method to execute some actions and not others, such as conditional statements in traditional imperative languages. Conditionals behave much in the same way as other actions, but they take actions as parameters and only execute a subset of them.

```
1 = X Y (TrueAction); FalseAction
```

In this case, the equivalence action takes four parameters: two integers, an action to perform if they are equal, and an action to perform if not. As conditionals can terminate recursive loops or create branching paths like in other languages. The major difference with Io is the lack of boolean types and syntax. The less than and greater than operators work identically.

Anonymous actions can act as data containers and actions can provide values as parameters to their own continuations as shown below. Storing data in actions acts very similarly to a return statement, however functionally what is happening is the continuation, which I call the client, is being called with an actual parameter provided by the anonymous action's formal parameter.

```
1 declare addTen: -> X NClient;
2         + X 10 -1-> Y;
3         NClient Y.
4
5 addTen 1 -2-> X;
6 write_int X;
7 terminate
```

Action #2 takes a formal integer parameter `X`. The actual parameter is provided by the `AddTen` operation, as the final line of its body is to invoke `NClient`, which is bound to action #2 with the passing `Y` as a parameter, which is provided by `+ X 10`. The above outputs `11` as expected and terminates.

Io does not enforce typing, though the lack of enforcement may cause errors to go unnoticed. Strict typing is useful for the programmer when understanding and debugging code. In the code examples in this paper I note the types of parameters and actions in an attempt to expand understanding. Io's allowance of action parameters to have similar notation to non-action parameters can create a lot of confusion, which vigilant noting of types avoids. I denote types by strings such as `A:[MyInt:int, MyString:string, MyAction:[], MyContinuation:C]`. The braces indicate actions, and the strings inside represent formal parameters and their types. Basic types are "int" and "string", continuations are "c".

# 3 Data Structures

I chose for this paper to focus on linked lists and binary trees, because they are very simple but still present a challenge to the inexperienced Io programmer. Other data structures like arrays don't work as well with Io, as Io is like other functional programming languages, where arrays and assignment and general do not fit. They could be implemented, but at first glance it would be an implementation that was built upon linked lists and trees, losing the advantages of an array.

## 3.1 Linked Lists

I based my code on the linked-list implementation suggested by *Advanced Programming Language Design*. [2] One creates a linked list by initializing an `emptyList` as an end node, and then creating the linked list from that by appending new integers to the beginning of the linked list.

### 3.1.1 Creation

Linked lists introduce a recursive type: `list: [[], int, list]` This notation means that a list is an action that takes two parameters an action, and an action that itself takes as parameters an integer and a list. Creation of linked lists requires two actions: `cons`, named after the Lisp constructor, and `emptyList`, which creates an empty list. I also implement `writeList`, because it's a good early proof of success for linked list creation.

```
1  //      invalidList: []
2  declare invalidList: ->;
3          print_string "Invalid List";
4          terminate.
5
6  //      cons: [Number:int, AList:list,
7  //                         LClient:[list]]
8  //      Null: []
9  //      NotNull: [int, list]
10 declare cons: -> Number AList LClient;
11         LClient -1-> Null NotNull;
12         NotNull Number MyList.
13
14 // emptyList: [Null:[],NotNull:[int, list]]
15 declare emptyList: -> Null NotNull;
16         Null.
17
18 //      writeList: [AList: list, Continuation:c]
19 //      Rest: [[], [int, list]] (List)
20 //      invalidList: []
21 //      First: int
22 //      MAX_INT: int
23 declare writeList: -> AList Continuation;
24         AList (invalidList) -2-> First Rest;
25         = First MAX_INT Continuation;
26         print_int First;
27         writeList Rest;
28         Continuation.
29
30 cons 1 emptyList -3-> aList;
31 cons 2 List -4-> aList;
32 writeList aList;
33 terminate.
```

Lists are represented as an action that takes two formal action parameters. Those two actions are: ”what the list does if it is empty” and ”what the list does if it is not”. Generally in my code the action taken if empty is an error action, and the action taken if not is an unraveling action that

separates an integer from the list. The first line of the `writeList` action, line 24, demonstrates this behavior. If the list is empty the action `invalidList` is executed, else the list provides itself as an actual parameter to the action #2. An integer list is comprised of an integer followed by another integer list, so the list provides its current integer as the formal parameter `First` and the integer list `Rest`. If `First` is equal to `MAX_INT` the list is empty, and the `Continuation` is executed. Otherwise, the action prints the integer stored in `First`, and then calls `writeList` on the rest of the list. To create a list line 30 invokes the `cons` action with the formal parameters `Number:1 AList:emptyList LClient:action #3`. `cons` invokes `LClient` #3 passing it action #1 as an actual parameter. Action #3 invokes the `NotNull` continuation with actual parameters `Number` and `AList`.

`EmptyList` takes as its two parameters two actions `Null` and `NotNull` and then it does the `Null` action.

The distinction of `Null` and `NotNull` as actions was a very difficult meaning to parse, as initially these are labeled much like any other formal parameter. This confusion caused difficulty in programming later actions because it was difficult to ascertain the end of a list without pseudodata. If a programmer does not wish to implement pseudodata, than when unwrapping a list as in this excerpt `AList (invalidList) -> First Rest;` the `invalidList` actual parameter should be replaced with the action to perform if the list is null, generally an appropriate continuation. The action to perform if the list is not null is the anonymous action beginning with `-> First Rest;`. I use pseudodata because it allows me to detect if two lists are empty at the same time, instead of just detecting if the first list is empty. `MAX_INT` is used as pseudodata indicating the end of the list via another action which I do not cover, called `initializeList`.

### 3.1.2 Sorted Insertion

```
1  //      insert: [Number:int, myList:list,
2  //                       LClient:[list]]
3  //      List(2,3,4,5)/Rest: list
4  //      invalidList: []
5  //      First: int
6  //      MAX_INT: int
7  declare insert: -> Number List LClient;
8          List (invalidList) -1-> First Rest;
9          < Number First (cons Number List -3-> List5;
10                 LClient List5);
11         insert Number Rest -4-> List2;
12         cons First List2 -5-> List3;
13         LClient List3.
```

The sorted insertion operation takes as formal parameters an integer, a list, and an action `LClient`. In this context an `LClient` is an action that takes a list as a formal parameter. In line ten, if the list is not empty, insert invokes action #1 with the components of the list as its parameters. Action #1's formal parameters are the integer `First` and the list `Rest`. First action #1 checks to see if the number to be inserted is less than the integer `First`, and if so it inserts the number at the head of the list, and executes the `LClient` with the result as a parameter. The `MAX_INT` pseudo-data at the end of the list ensures that the list cannot be empty. If the number is not less than the first one `insert` makes a recursive call to insert the number into the list `Rest`. The formal parameter `List2` is bound to the output of the recursive call. Action #4 then invokes the `cons` action to place the `First` integer to top of the `List2` and then executes the `LClient` with the list as an actual parameter.

### 3.1.3 Strict List Comparison

```
1  //      compareList: [List1: list, List2: list,
2  //                    List3: list, Path1: [c],
3  //                    Path2: [c], Path3: [c],
4  //                    Continuation: c]
5  //      Rest(1,2): list
6  //      invalidList: []
7  //      First(1,2): int
8  //      MAX_INT: int
9  declare compareList: -> List1 List2 Path1 Path2 Path3
10                        Continuation;
11     List1 (invalidList) -> First1 Rest1;
12     List2 (invalidList) -> First2 Rest2;
13     = First1 First2 (= First1 MAX_INT
14                        (path3 Continuation);
15                        compareList Rest1 Rest2 Path1
16                        Path2 Path3 Continuation);
17     = First2 MAX_INT (Path1 Continuation);
18     = First1 MAX_INT (Path2 Continuation);
19     > First1 First2 (Path1 Continuation);
20     Path2 Continuation.
```

List comparison compares two integer lists for lexicographic ordering. The algorithm compares each integer that comprises them one at a time in order. If at any step one list has a larger integer, that list is the larger list. If one list completes before the other list, the shorter list is the smaller list. The lists are only equal if both have the same numbers in the same order.

The action `compareList` takes six formal parameters: two lists, three actions, and a continuation. The three actions represent the different outcomes: `Path1` if the second list is larger, `Path2` if the first list is larger, and `Path3` if they are equivalent. After extracting the `First` integers of each list, the action compares them. If the integers are equal then it checks to see if the integers are equal to the pseudodata, if so it executes the `Path3` action with the `Continuation` as an actual parameter. If the numbers are not equivalent to the pseudodata, indicating that the lists are equivalent so far, the action recursively calls itself. The action then checks if either list is complete, indicating that one list is larger than another, and

if either is it runs the appropriate path. After these branches it executes whichever path is appropriate, given that the numbers are unequal. This action steps through both lists in tandem. The list comparison action is one of the actions, which requires knowledge if either List has finished without interrupting the program's sequence, and in my implementation requires pseudodata.

### 3.1.4 mapCar

`mapCar` gets its name from a Lisp function that executes a function over each element in a list.

```
1  //      mapCar: myList:[myList:list,
2  //                      AnAction: [int[int]],
3  //                      LClient: [list]]
4  //      Rest: list
5  //      invalidList: []
6  //      First/Result: int
7  //      MAX_INT: int
8  //      New(Rest/List): list
9  declare mapCar: -> List AnAction LClient;
10     List (invalidList) -> First Rest;
11     = First MAX_INT (LClient List);
12     AnAction First -> Result;
13     mapCar Rest Function -> NewRest;
14     Cons Result NewRest -> NewList;
15     LClient NewList.
```

`mapCar`'s formal parameters are a list, an action to be performed on each element in the list, and an action that takes a List as a formal parameter. After unwrapping the list and checking pseudodata equality. the action is executed on the `First` integer, with the result being placed on the list output from the `mapCar` function on `Rest`. This list is than provided as an actual parameter to the LClient action. The `AnAction` parameter can be bound to any action with type `[int,[int]]`.

### 3.1.5 map2Car

`map2Car` is much like `mapCar`: instead it takes two lists as parameters, and applies a binary action to each element in tandem, resulting in a list to be used as a parameter to a `LClient`.

```
1  //      Map2Car: [List1: list, List2: list,
2  //                      AnAction: [int, int, [int]],
3  //                      LClient: [list]]
4  //      (''/New/Result)List(3/4): List
5  //      invalidList: []
6  //      First(1/2): int
7  //      Rest(1/2): list
8  //      result: int
9  declare Map2Car: -> List1 List2 AnAction LClient;
10    List1 invalidList -> First1 Rest1;
11    List2 invalidList -> First2 Rest2;
12    = First1 MAX_INT (= First1 First2 (initializeList
13                          -> List3; LClient List3);
14                          print_string "WARNING: Lists
15                          of unequal Length";
16                          initializeList -> List4;
17                          LClient List4);
18    = First2 MAX_INT (print_string "WARNING:
19                          Lists of unequal Length";
20                          initializeList -> List4;
21                          LClient List4);
22    AnAction First1 First2 -> Result;
23    map2Car Rest1 Rest2 AnAction -> NewList;
24    cons Result NewList -> ResultList;
25    LClient resultList.
```

`map2Car` after it unravels the lists checks equality with the pseudodata. If both lists are empty then it executes the `LClient` with an empty list to be filled with the results of its callers. If one list is longer than the other, `map2Car` prints a warning to standard output and executes the `LClient` with the the list made so far in an attempt to fail gracefully. `map2Car` then calls the action on the two elements and then recursively calls itself. The new list is then constructed and the `LClient` has the new list as a pa-

rameter. An action that works well with `map2Car` must have type `[int, int, [int]]` such as the addition operator `+`.

## 3.2 Binary Trees

Binary trees share many similarities with linked lists, however, instead of having a second action as a formal parameter that only takes two formal parameters, binary tree's second formal action parameter takes three formal parameters. One formal parameter represents the value in the node, the second formal parameter represents left subtree, and the last formal parameter represents the right subtree. The recursive type of tree is `[[], [int, tree, tree]]` with data contained in each node. For ease of searching, pseudodata marks the leaves, as the pseudodata in the linked list implementation marks the list's end.

### 3.2.1 Creation

```
1  //      writeTree: [ATree: tree, Spaces:string,
2  //                  Continuation:c]
3  //      invalidTree: []
4  //      Value: int
5  //      LeftTree/RightTree: tree
6  //      MAX_INT: int
7  //      NewSpaces: string
8  declare writeTree: -> ATree Spaces Continuation;
9      ATree invalidTree ->
10                          Value LeftTree RightTree;
11     = Value MAX_INT (Continuation);
12     //The carrot operator concatenates strings
13     ^ "   " Spaces -> NewSpaces;
14     writeTree LeftTree NewSpaces;
15     //The print_string_ operator does not print a
16     //newline.
17     print_string_ Spaces;
18     print_int Value;
19     writeTree RightTree NewSpaces;
20     Continuation.
21
22 //      consTree: [Number:int, LeftTree:tree,
23 //                  RightTree: tree,
24 //                  TClient:[tree]]
25 //      Null: []
26 //      NotNull: [int,tree,tree]
27 declare consTree: -> Number LeftTree
28                          RightTree TClient;
29     TClient -> Null NotNull;
30     NotNull Number LeftTree RightTree.
31
32 //      emptyTree: [Null:[],
33 //                  NotNull:[int, tree, tree]]
34 declare emptyTree: -> Null NotNull;
35     Null.
```

`consTree` is similar to `cons`. The `emptyTree` action creates a tree, this time the `NotNull` action parameter is of the type `[int, tree, tree]` The `consTree` action takes two trees as parameters, to act as subtrees, a number to store in the node, and a continuation that expects a tree. The `writeTree` action takes as formal parameters: a tree, a string comprised of only spaces, and a continuation. in the `writeTree` the code checks for the presence of the pseudodata, and then concatenates the given spaces to create a `NewSpaces` string. This growing string allows for a visually helpful tree with a left to right orientation. The `NewSpaces` string is an actual parameter in lines 14 and 17 to print the left tree and right tree, with the action printing the current node in the middle. The program prints the tree in symmetric order and then executes the continuation

```
1  consTree 1 EmptyTree EmptyTree -> LeftTree;
2  consTree 3 EmptyTree EmptyTree -> RightTree;
3  consTree 2 LeftTree RightTree -> FinalTree;
4  writeTree FinalTree "";
5  terminate.
6
7  Output:
8     1
9  2
10    3
```

A difference from the linked-list syntax and the binary-tree syntax is the new method of unraveling the data structure `Tree invalidTree ->First LeftTree RightTree;` . The unraveling has more formal parameters, because of the two subtrees. Pseudodata does not need to be used to detect when a tree is empty, however in my code pseudodata marks when tree leaves are detected in multiple trees at the same time, which is more difficult to implement without pseudodata.

### 3.2.2 Sorted Insertion

```
1  //      insertTree: [Number: int, ATree: tree,
2  //                   TClient: [tree]]
3  //      invalidTree: []
4  //      Value: int
5  //      (Left/Right)Tree: tree
6  //      MAX_INT: int
7  //      NewTree: tree
8  declare insertTree: -> Number ATree TClient;
9      ATree invalidTree ->
10                          Value LeftTree RightTree;
11     = Value MAX_INT (createTree Number -> NewTree;
12                          TClient NewTree);
13     > Number Value (InsertTree Number RightTree ->
14                          RightTree; consTree Value
15                          LeftTree RightTree -> NewTree;
16                          TClient NewTree);
17     insertTree Number LeftTree -> LeftTree;
18     consTree Value LeftTree RightTree -> NewTree;
19     TClient NewTree.
```

Sorted insertion takes as parameters an integer, a tree, and a tree client. First, `insertTree` unravels the tree node and compares the integer `Value` with the pseudodata value. If the node equals the pseudodata, the action creates a new tree to append at this leaf node, and it executes the `TClient` with the new tree as an actual parameter. After checking the pseudodata, `insertTree` compares the given number with the number in the current node. If the number to be inserted is greater, a recursive call inserts that number in the `RightTree`, recreates the tree with the new right tree, and executes the `TClient` with that recreated tree as a parameter. If the number to be inserted is not greater, it is inserted in the left tree, and the final tree is recreated with that new tree.

### 3.2.3 Strict Comparison

The goal of strict comparison is to ascertain if two trees have the same layout and the same values.

```
1  //      compareTreeStrictRecursive: [Tree1:tree,
2  //                      Tree2:tree,
3  //                      NClient: [int]]
4  //      invalidTree: []
5  //      First(1/2): int
6  //      (Left/Right)Tree(1/2): tree
7  //      MAX_INT: int
8  //      Num(1/2): int
9  //      result: int
10 declare compareTreeStrictRecursive: -> Tree1 Tree2
11                      NClient;
12     Tree1 invalidTree -> First1
13                      LeftTree1 RightTree1;
14     Tree2 invalidTree -> First2
15                      LeftTree2 RightTree2;
16     = First1 First2 (= First1 MAX_INT
17                 (NClient 0);
18                 compareTreeStrictRecursive
19                 LeftTree1 LeftTree2 -> Num1;
20                 compareTreeStrictRecursive
21                 RightTree1 RightTree2 -> Num2;
22                 + Num1 Num2 -> result;
23                 > result 0 (NClient 1);
24                 NClient 0);
25     NClient 1.
26
27 //      compareTreeStrict: [Tree1:tree, Tree2:tree,
28 //                      Pathy: [c], Pathn, [c],
29 //                      Continuation:C
30 //      result: int
31 declare compareTreeStrict: -> Tree1 Tree2 Pathy Pathn
32                      Continuation;
33     compareTreeStrictRecursive Tree1 Tree2 -> result;
34     = result 0 (Pathy Continuation);
35     Pathn Continuation.
```

`compareTreeStrict` does not compare trees in lexicographic ordering, or compare trees based only on the values of the nodes. A comparison

operation that compares trees lexicographically is possible, but the behavior of strict comparison is much easier to define. `compareTreeStrict` calls `compareTreeStrictRecursive` and determines if the value used on the `NClient` denote equivalence or not. `compareTreeStrict` takes as formal parameters two trees, and three actions: One to be performed if equivalent, one if not, and a continuation. `compareTreeStrictRecursive` unravels to two trees, compares the value, and executes the `NClient` with a number based on if the value is equivalent to the pseudodata in one or both trees, an equal value in both trees or a different value in both trees.

### 3.2.4 mapCar

```
1  //      mapCarTree: [ATree: tree, AnAction: [int],
2  //                   TClient: [tree]]
3  //      First: int
4  //      (Left/Right/NewLeft/NewRight/End)Tree: tree
5  //      Result: int
6  declare mapCarTree: -> ATree AnAction TClient;
7      ATree invalidTree -> First
8                        LeftTree RightTree;
9      = First MAX_INT (TClient ATree);
10     AnAction First -> Result;
11     mapCarTree LeftTree myAction -> NewLeftTree;
12     mapCarTree RightTree myAction -> NewRightTree;
13     consTree Result NewLeftTree NewRightTree ->
14                       EndTree;
15     TClient EndTree.
```

`mapCarTree`, named after the Lisp function, takes a tree and an action and executes `TClient` with a parameter of the result of running the action on each element in the tree. After `mapCarTree` checks the pseudodata it runs the action on both the left and right tree, and then the recombined tree is a parameter for the `TClient` action.

# 4 Io as a first language for instruction

Upon first seeing and working with Io I would not have suggested it as a first programming language, because its behavior is so different from other languages. Io has some advantages in its esoteric behavior, however. The lack of enforced typing and the spartan notation both create an easy-to-write language similar to other common first languages such as Python. The question lies in if those savings in understanding notation make up for the new time spent in understanding code behavior added to the time required to move from Io to a more traditional programming language. It is possible that the additional time spent understanding code behavior could be desirable because the confusing aspects of Io are mostly sequencing and odd but consistent notational behaviors, and not purely notational in nature, allowing programmers to focus on program logic and not remembering an odd quirk of notation. Let's look at some code examples and imagine how difficult it would be to understand them from a beginner's viewpoint.

## 4.1 Hello, World!

```
1 print_string "Hello, World!";
2 terminate
```

"Hello, World!" has long been an example of a first program in any programming language, and it is simple in Io. There is not much here that can be misunderstood, and the function of this code excerpt is much the same as any other language; though without the more difficult notation that some other versions of "Hello, World!" require.

## 4.2 Action Definition

```
1 declare: -> Add Num1 Num2 NClient;
2         + Num1 Num2 -> Result;
3         NClient Result.
4
5 add 2 3 -> Result;
6 print_int Result;
7 terminate
```

Actions, like functions in other languages, are also relatively easy to code. The inclusion of a reserved word, `declare`, and the specific notation isn't any more difficult than other language's syntax. It's also quite a bit simpler than languages such as Java or C. If we consider predeclared functions, usage can be confusing; addition, long an easy first step in programming, is now obfuscated behind a different syntax with what a person would be accustomed to. `2 + 3` as in many common languages is something even those who do not think of themselves as coders can parse. `+ 2 3` introduces confusion where beginners may want to only see relief. I recommend renaming the single character operators to be verbs the programmer would make themselves. "add 2 3" doesn't bring to mind the mathematical notation, and brings a linguistic similarity to Io that would probably prove helpful to beginners. It also doesn't make addition or subtraction an exception to how actions normally work.

## 4.3 Loops

```
1 declare: forLoop Num1 Continuation;
2         print_int Num1;
3         - Num1 1 -> Result;
4         forLoop Result;
5         Continuation.
6
7 forLoop 10;
8 terminate
```

Loops are another common topic to cover for first-time programmers, and the first cause of worry specific to Io's notation. Io, like all functional

programming languages, lacks loops, instead relying entirely on recursion. Even if a beginner understands and can parse recursion easily, not having different methods to create loops restricts the number of tools a beginning programmer has. Although I am certain any loop could if engineered correctly mimic recursion, and any recursion could if engineered correctly mimic loops, it is worrisome that a beginner would not have some of the positive aspects of **`for`** and **`while`** looping. I think you could teach programming without them, and for some students that might be a positive aspect of the course; but that is not true for every student, and lacking those controls is worrisome.

Another point of confusion would be the lack of returns. The current logical flow of an action from start to end is relatively easy to parse, but continuations and clients instead of returns could prove confusing to a beginner. In other languages, the return looks and acts like an end to a function. Io instead has continuations that pass results to the rest of the code. The difference is subtle, but that subtlety adds to confusion when a beginner is learning.

Arrays are another aspect of programming that is commonly taught early; they would have to be discarded entirely. I think there are some positive aspects to teaching linked lists, for example, earlier however. I imagine students think of arrays being stored like a list of numbers on a page. Facing a different type of storage early in their education might help open their eyes to other data structures before they are even introduced, particularly if arrays aren't even mentioned in the first place.

I think there are some problems with replacing all newcomer's language with Io as well. Io doesn't have the support structure that other languages have, and there isn't much current evolution from low level Io to high level Io. If you learn to program with Python; you start with the same things as every other language, as detailed above. But it is really easy with Python to go from beginning programs to creating a program with a user interface to accomplish some task, and many different types of programs are coded in Python. While I would not want a first programming language to be picked entirely on what is popular, the popularity of a language influences access to help and future work that can be easily done with a language. A student pays more attention and is more interested if they are taught something that they think is immediately useful, and sadly I don't think Io has that draw.

# 5 Future Research

Some places that a future student could take Io are implementing other data structures, while I have done some initial work into ternary and n-ary trees I am certain there is more work to be accomplished. An interesting problem to solve would be a softer comparison of trees where instead of comparing them node by node the action would compare their linked list equivalents, without transforming them of course. I would also be interested in seeing a red-black tree in this language or a self balancing tree of any type.

## 5.1 Ternary Trees

I have done some initial work on ternary trees. Ternary trees would be simple to implement if a programmer would just expand `emptyTree` and the tree unraveling function to have an extra slot for a third tree and a second number. However, that isn't generalizable; at least not beyond rewriting trees for each `n` you wished. A ten-ary tree would largely be unrecognizable for example. So I believe the best way to create a ternary tree would instead be like two linked lists in each node. The first linked list would be a collection of numbers and the second would be a collection of subtrees. The unraveling action would extract those two lists, which would have a number of values according to the value of `n`, which would need to be a parameter to the action the unraveling was taking place in. I imagine this implementation would allow for dynamic `n`-ary trees which do not have obtuse hard to read syntax.

# 6 Conclusion

The Io programming language has a unique notation that simplifies programming notation without over complicating direct understanding of most of the code written in it. There are some small aspects of the Io language that are more difficult to understand for beginners such as the reliance on recursion for loops, and the usage of continuations. Continuations in particular could confuse new programmers who could be uncertain about the mechanics of parameter passing in general, and passing an actions would only compound that problem.

# References


[1] Raphael Levien *Io: a new programming notation*. 1989, SIGPLAN Notices, Vol. 24, No. 12.

[2] Raphael Finkel, "Control Structures," in *Advanced Programming Language Design* 1st ed. Menlo Park: ADWES, 1996, ch. 2, sec. 3, pp. 50-57

[3] Everett Boyer (2017, April 13) *Io Codebase* Available: `https://drive.google.com/file/d/0B7mC3XFGKzdTV1p3VFI1YlpSX3c/view?usp=sharing`

[4] Martin Sandin. (2003, Jan. 28) *io & amalthea* Available: `https://hackage.haskell.org/package/Ganymede-0.0.0.5/src/vague%27s%20-%20amalthea.html`