Lessons from the two-three tree Homework

CS 420/520

Andrew P. Black

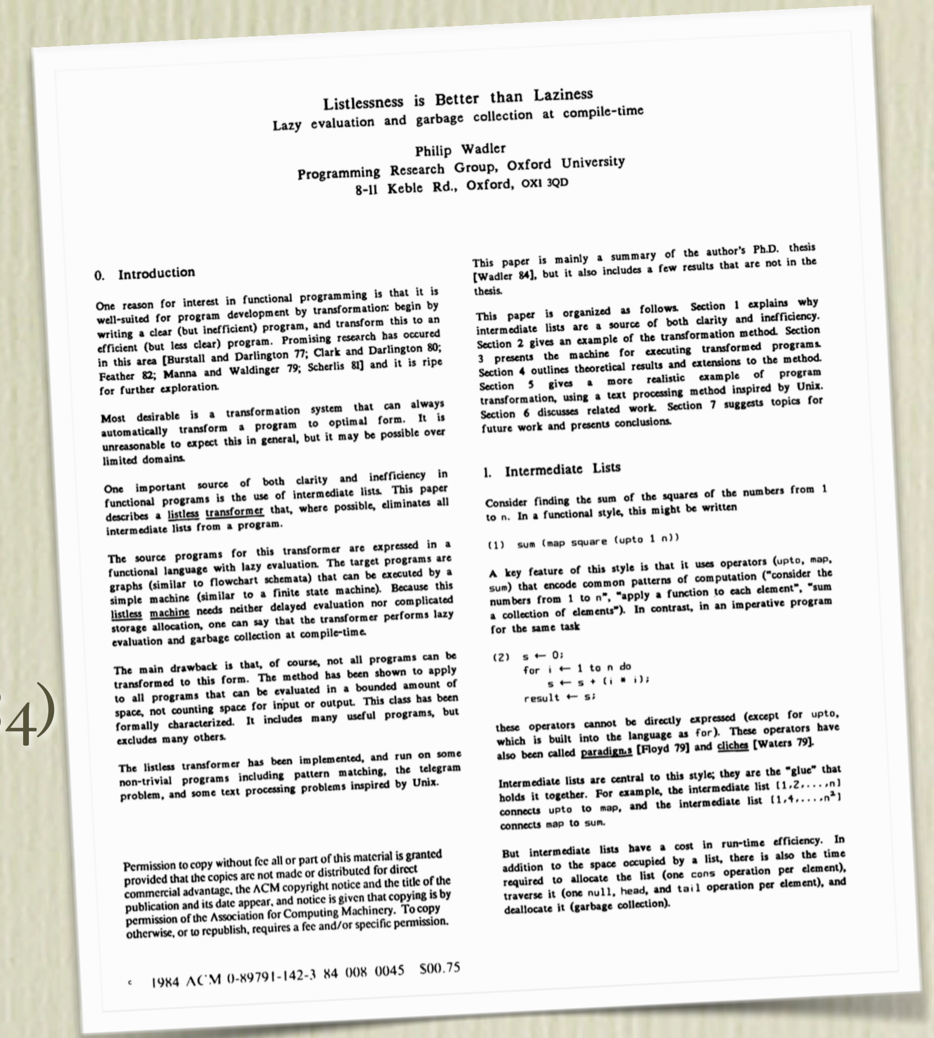
Portland State University

Goals

- See multiple objects implementing the same interface
- See blocks being used as arguments
  - `replaceMeBy` and `absorb` blocks
  - continuation block as argument to `sort3`
- *Listlessness* as a programming pattern
  - iterators deliver their results one-by-one
  - *Listlessness is Better than Laziness* (Wadler, 1984)

Goals

- See multiple objects implementing the same interface
- See blocks being used as arguments
  - `replaceMeBy` and `absorb` blocks
  - continuation block as argument to `sort3`
- *Listlessness* as a programming pattern
  - iterators deliver their results one-by-one
  - *Listlessness is Better than Laziness* (Wadler, 1984)



Listlessness is Better than Laziness
Lazy evaluation and garbage collection at compile-time

Philip Wadler
Programming Research Group, Oxford University
8-11 Keble Rd., Oxford, OX1 3QD

0. Introduction

One reason for interest in functional programming is that it is well-suited for program development by transformation: begin by writing a clear (but inefficient) program, and transform this to an efficient (but less clear) program. Promising research has occured in this area [Burstall and Darlington 77; Clark and Darlington 80; Feather 82; Manna and Waldinger 79; Scherlis 81] and it is ripe for further exploration.

Most desirable is a transformation system that can always automatically transform a program to optimal form. It is unreasonable to expect this in general, but it may be possible over limited domains.

One important source of both clarity and inefficiency in functional programs is the use of intermediate lists. This paper describes a listless transformer that, where possible, eliminates all intermediate lists from a program.

The source programs for this transformer are expressed in a functional language with lazy evaluation. The target programs are graphs (similar to flowchart schemata) that can be executed by a simple machine (similar to a finite state machine). Because this listless machine needs neither delayed evaluation nor complicated storage allocation, one can say that the transformer performs lazy evaluation and garbage collection at compile-time.

The main drawback is that, of course, not all programs can be transformed to this form. The method has been shown to apply to all programs that can be evaluated in a bounded amount of space, not counting space for input or output. This class has been formally characterized. It includes many useful programs, but excludes many others.

The listless transformer has been implemented, and run on some non-trivial programs including pattern matching, the telegram problem, and some text processing problems inspired by Unix.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1984 ACM 0-89791-142-3 84 008 0045 $00.75

This paper is mainly a summary of the author's Ph.D. thesis [Wadler 84], but it also includes a few results that are not in the thesis.

This paper is organized as follows. Section 1 explains why intermediate lists are a source of both clarity and inefficiency. Section 2 gives an example of the transformation method. Section 3 presents the machine for executing transformed programs. Section 4 outlines theoretical results and extensions to the method. Section 5 gives a more realistic example of program transformation, using a text processing method inspired by Unix. Section 6 discusses related work. Section 7 suggests topics for future work and presents conclusions.

1. Intermediate Lists

Consider finding the sum of the squares of the numbers from 1 to n. In a functional style, this might be written

```
(1)  sum (map square (upto 1 n))
```

A key feature of this style is that it uses operators (upto, map, sum) that encode common patterns of computation ("consider the numbers from 1 to n", "apply a function to each element", "sum a collection of elements"). In contrast, in an imperative program for the same task

```
(2)  s ← 0;
     for i ← 1 to n do
         s ← s + (i * i);
     result ← s;
```

these operators cannot be directly expressed (except for upto, which is built into the language as for). These operators have also been called paradigms [Floyd 79] and cliches [Waters 79].

Intermediate lists are central to this style; they are the "glue" that holds it together. For example, the intermediate list [1,2,...,n] connects upto to map, and the intermediate list [1,4,...,n²] connects map to sum.

But intermediate lists have a cost in run-time efficiency. In addition to the space occupied by a list, there is also the time required to allocate the list (one cons operation per element), traverse it (one null, head, and tail operation per element), and deallocate it (garbage collection).

Portland State UNIVERSITY

- Program to an Interface, not to an Implementation
  - The implementation was given; all you had to do was figure out the interface
- Reading tests and documentation to discover the interface
  - Resolving ambiguities:
    - writing tests, asking questions
    - spotting bugs or inconsistencies

Using multiple Classes

Using multiple Classes

- A student wrote*:

> I had experience coding a 2-3 tree in CS 163. Back in those days, I struggled for many days to deal with insert and remove. I wrote a 2-page method to add a new node to tree. I used an *if-then-else* statement to find out if the current node was empty, contained one value, or contained two. And then another nested *if* inside each branch to see if we needed to add left/middle/right, or go left/middle/right. That was a mess. I could imagine how hard it would be for a person to comprehend the code.
>
> Using OOP to implement it makes life easier. We don't need to find out which kind of node we are in: we already know. We also already know when we should change to another kind of node, and which it should be. All we need do is implement a specific case in each class, and then let the objects do their jobs.

* text corrected for grammar

Using multiple Classes

- A student wrote*:

> I had experience coding a 2-3 tree in CS 163. Back in those days, I struggled for many days to deal with insert and remove. I wrote a 2-page method to add a new node to tree. I used an *if-then-else* statement to find out if the current node was empty, contained one value, or contained two. And then another nested *if* inside each branch to see if we needed to add left/middle/right, or go left/middle/right. That was a mess. I could imagine how hard it would be for a person to comprehend the code.
>
> Using OOP to implement it makes life easier. We don't need to find out which kind of node we are in: we already know. We also already know when we should change to another kind of node, and which it should be. All we need do is implement a specific case in each class, and then let the objects do their jobs.

-

* text corrected for grammar

Using multiple Classes

- A student wrote*:

I had experience coding a 2-3 tree in CS 163. Back in those days, I struggled for many days to deal with insert and remove. I wrote a 2-page method to add a new node to tree. I used an *if-then-else* statement to find out if the current node was empty, contained one value, or contained two. And then another nested *if* inside each branch to see if we needed to add left/middle/right, or go left/middle/right. That was a mess. I could imagine how hard it would be for a person to comprehend the code.

Using OOP to implement it makes life easier. We don't need to find out which kind of node we are in: we already know. We also already know when we should change to another kind of node, and which it should be. All we need do is implement a specific case in each class, and then let the objects do their jobs.

- Multiple *token* classes in the *glob* homework

Portland State University

* text corrected for grammar

Using multiple Classes

- A student wrote*:

> I had experience coding a 2-3 tree in CS 163. Back in those days, I struggled for many days to deal with insert and remove. I wrote a 2-page method to add a new node to tree. I used an *if-then-else* statement to find out if the current node was empty, contained one value, or contained two. And then another nested *if* inside each branch to see if we needed to add left/middle/right, or go left/middle/right. That was a mess. I could imagine how hard it would be for a person to comprehend the code.
>
> Using OOP to implement it makes life easier. We don't need to find out which kind of node we are in: we already know. We also already know when we should change to another kind of node, and which it should be. All we need do is implement a specific case in each class, and then let the objects do their jobs.

-
    - Multiple *token* classes in the *glob* homework
    - Many different kinds of *component* on a canvas

* text corrected for grammar

Using multiple Classes

- A student wrote*:

I had experience coding a 2-3 tree in CS 163. Back in those days, I struggled for many days to deal with insert and remove. I wrote a 2-page method to add a new node to tree. I used an *if-then-else* statement to find out if the current node was empty, contained one value, or contained two. And then another nested *if* inside each branch to see if we needed to add left/middle/right, or go left/middle/right. That was a mess. I could imagine how hard it would be for a person to comprehend the code.

Using OOP to implement it makes life easier. We don't need to find out which kind of node we are in: we already know. We also already know when we should change to another kind of node, and which it should be. All we need do is implement a specific case in each class, and then let the objects do their jobs.

-
    - Multiple *token* classes in the *glob* homework
    - Many different kinds of *component* on a canvas
    - Many different test cases in a test suite

* text corrected for grammar

- # When using the state pattern

Use More Objects

- A student writes:

I was pretty happy with my code before I posted a question to the class forum about the behavior of extra symbols inside brackets. At that point I had only one "bracketParseState", rather than the "leftBracketParseState" and "bracketCharsParseState" I ended up with. I chose to raise an error any time one of the other symbols appeared inside brackets, if a left bracket appeared without a right bracket, or if a right bracket appeared before a left bracket.

After reading the discussion on the forum, I switched to the two state implementation, where one is used when a left bracket first appears, and the second one is used to fill the brackets with characters. Now the combinations of symbols described above are all treated as plain characters rather than raising errors. I'm happier with this version of the code.

- Dictionary itself!
  - hash-table implementation
  - search-tree implementation

Objects have *Two* Interfaces

1. Interface to ***use*** the object:

```
type Dictionary = interface {
    at(_)put(_); keys; iterator; do(_); … }
```

2. Interface to ***create*** the object:

```
type DictionaryFactory = interface {
    dictionary(_); dictionary.withAll(_);
    dictionary <<; dictionary.with(_);
    dictionary.empty }
```

Assignment wasn't explicit about this; most students missed its importance.

- To *test* a dictionary, you have to *create* a dictionary

Portland State UNIVERSITY

Tests/Specs *Communicate*

```
type Collection⟦T⟧ = type {
    iterator -> Iterator⟦T⟧
    // Returns an iterator over my elements.  It is an error to modify self while iterating
    // over it. Note: all other methods can be defined using iterator. Iterating over a
    // dictionary yields its values.

    …

type Dictionary⟦K, T⟧ = Collection⟦T⟧ & interface {
    …
    keys -> Collection⟦K⟧      // returns my keys as a lazy sequence in arbitrary order
    values -> Collection⟦T⟧    // returns my values as a lazy sequence in arbitrary order
    bindings -> Enumerable⟦ Binding⟦K, T⟧ ⟧   // returns my bindings as a lazy sequence
```

My tests tell much the same story:

```
test_small_iterator: <set{3::three, 4::four, 2::two, 1::one, 5::five}>
        should be <set{"five", "three", "two", "one", "four"}>
```

Portland State University

Simple Methods

- Compare

```
method ≠(someOtherDictionary) {
  if (self == someOtherDictionary) then {
     return false
  } else {
     return true
  }
}
```

to

```
method ≠(other) { (self == other).not }
```

- Does `other` have to be a dictionary?

Shop, don't Build

Portland State University

Shop, don’t Build

- Consider

Portland State University

Shop, don't Build

- Consider

```
method ++ (t) {
     def newTree = self.copy
     def iter = t.iterator
     var current
     (1 .. iter.zipper.size).do { i →
        current := iter.zipper.at(i)
        (1 .. current.bindingList.size).do { j →
          newTree.at(current.bindingList.at(j).key)
                   put (current.bindingList.at(j).value)
        }
     }
     newTree
}
```

Shop, don't Build

- Consider

```
method ++ (t) {
      def newTree = self.copy
      def iter = t.iterator
      var current
      (1 .. iter.zipper.size).do { i →
          current := iter.zipper.at(i)
          (1 .. current.bindingList.size).do { j →
             newTree.at(current.bindingList.at(j).key)
                          put (current.bindingList.at(j).value)
          }
      }
      newTree
}
```

- Train wreck!
    - This will work *only* when `t` has an `iterator` with a `zipper` method that is itself a collection

Portland State UNIVERSITY

- Better to reuse the implementation from *collectionsPrelude* `dictionary`:

```
method ++ (other:Collection⟦T⟧) {
     // answers a new dictionary containing all my keys and
     // the keys of other; if other contains one of my keys,
     // other's value overrides mine

     def newDict = self.copy
     other.keysAndValuesDo {k, v ->
        newDict.at(k) put(v)
     }
     return newDict
}
```

- This works for any `other` that understands `keysAndValuesDo(_)`
- Many of the methods in the dictionary implementation *could be* factored out into a reusable trait.

Lazy Sequences, aka Streams

- Implementations are available for reuse in *collectionsprelude*

```
trait iteratorOver⟦T,R⟧ (sourceIterator: Iterator⟦T⟧)
        mappedBy (function:Function1⟦T, R⟧) -> Iterator⟦R⟧ {
    method asString { "a mapped iterator over {sourceIterator}" }
    method hasNext { sourceIterator.hasNext }
    method next { function.apply(sourceIterator.next) }
}
```

```
class lazySequenceOver⟦T,R⟧ (source: Collection⟦T⟧)
        mappedBy (function:Function1⟦T, R⟧) -> Enumerable⟦R⟧ {
    use enumerable⟦T⟧
    class iterator {
        use iteratorOver⟦T,R⟧ (source.iterator) mappedBy (function)
    }
    method size { source.size }
    method isEmpty { source.isEmpty }
    method asDebugString { "a lazy sequence mapping over {source}" }
}
```

```
method iteratorOver⟦T⟧ (sourceIterator: Iterator⟦T⟧)
        filteredBy(predicate:Predicate1⟦T⟧) -> Iterator⟦T⟧ {
    // returns a trait that supplies the iteration protocol

    var cache
    var cacheLoaded := false
    object {
        method asString { "a filtered iterator over {sourceIterator}" }
        method hasNext {
            // To determine if this iterator has a next element, we have to find
            // an acceptable element; this is then cached, for the use of next
            // If I return true, the cache is loaded.
            if (cacheLoaded) then { return true }
            while { sourceIterator.hasNext } do {
                def outerNext = sourceIterator.next
                def isAcceptable = predicate.apply(outerNext)
                if (isAcceptable) then {
                    cacheLoaded := true
                    cache := outerNext
                    return true
                }
            }
            return false
        }
        method next {
            if (hasNext) then {
                cacheLoaded := false
                return cache
            } else {
                IteratorExhausted.raise "no more elements in {self}"
            }
        }
    }
}
```

```
method iteratorOver[T] (sourceIterator: Iterator[T])
        filteredBy(predicate:Predicate1[T]) -> Iterator[T] {
    // returns a trait that supplies the iteration protocol

    var cache
    var cacheLoaded := false
    object {
        method asString { "a filtered iterator over {sourceIterator}" }
        method hasNext {
            // To determine if this iterator has a next element, we have to find
            // an acceptable element; this is then cached, for the use of next
            // If I return true, the cache is loaded.
            if (cacheLoaded) then { return true }
            while { sourceIterator.hasNext } do {
                def outerNext = sourceIterator.next
                def isAcceptable = predicate.apply(outerNext)
                if (isAcceptable) then {
                    cacheLoaded := true
                    cache := outerNext
                    return true
                }
            }
            return false
        }
        method next {
            if (hasNext) then {
                cacheLoaded := false
                return cache
            } else {
                IteratorExhausted.raise "no more elements in {self}"
            }
        }
    }
}
```

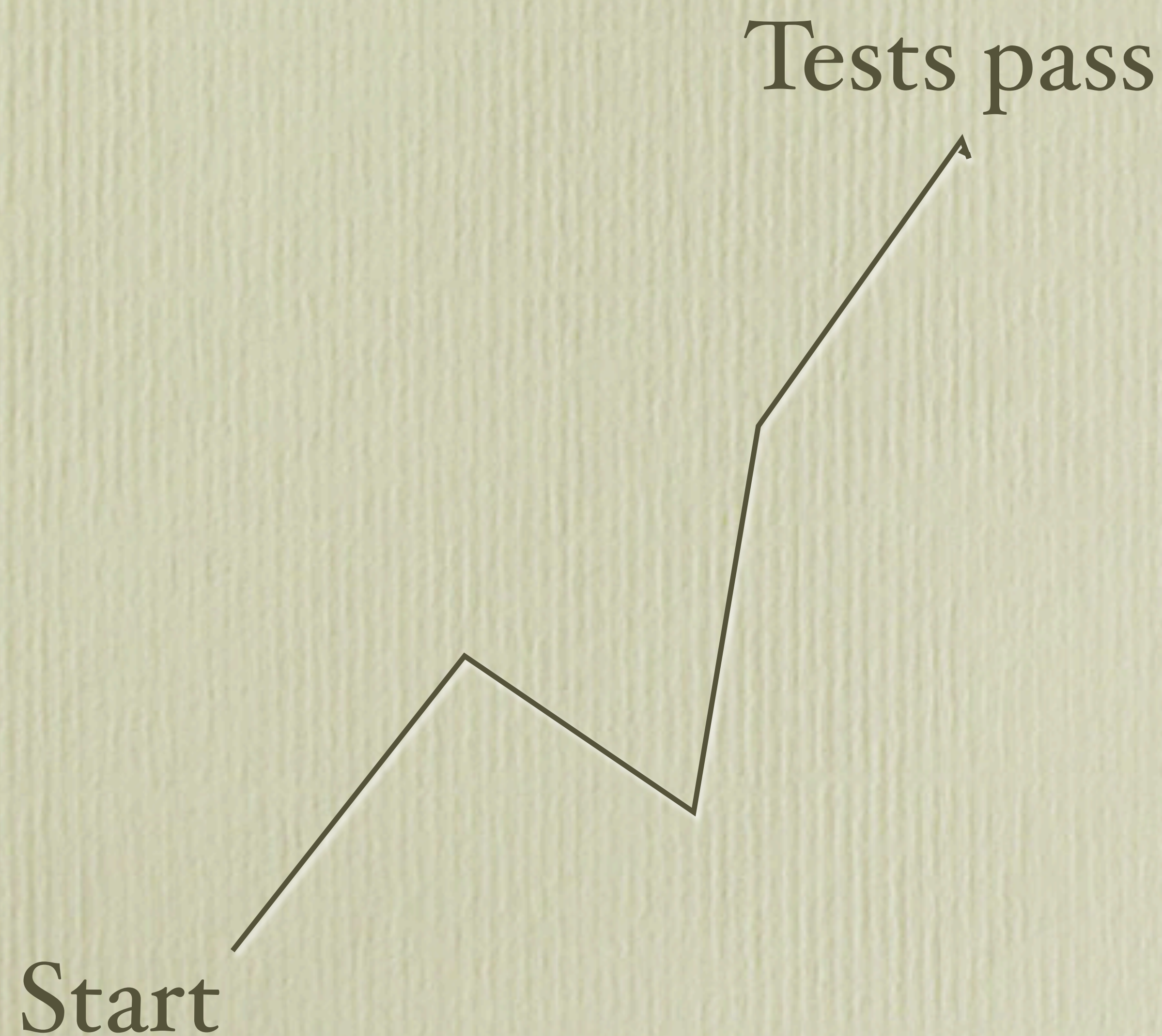
```
method iteratorOver⟦T⟧ (sourceIterator: Iterator⟦T⟧)
        filteredBy(predicate:Predicate1⟦T⟧) -> Iterator⟦T⟧ {
    // returns a trait that supplies the iteration protocol

    var cache
    var cacheLoaded := false
    object {
        method asString { "a filtered iterator over {sourceIterator}" }
        method hasNext {
            // To determine if this iterator has a next element, we have to find
            // an acceptable element; this is then cached, for the use of next
            // If I return true, the cache is loaded.
            if (cacheLoaded) then { return true }
            while { sourceIterator.hasNext } do {
                def outerNext = sourceIterator.next
                def isAcceptable = predicate.apply(outerNext)
                if (isAcceptable) then {
                    cacheLoaded := true
                    cache := outerNext
                    return true
                }
            }
            return false
        }
        method next {
            if (hasNext) then {
                cacheLoaded := false
                return cache
            } else {
                IteratorExhausted.raise "no more elements in {self}"
            }
        }
    }
}
```

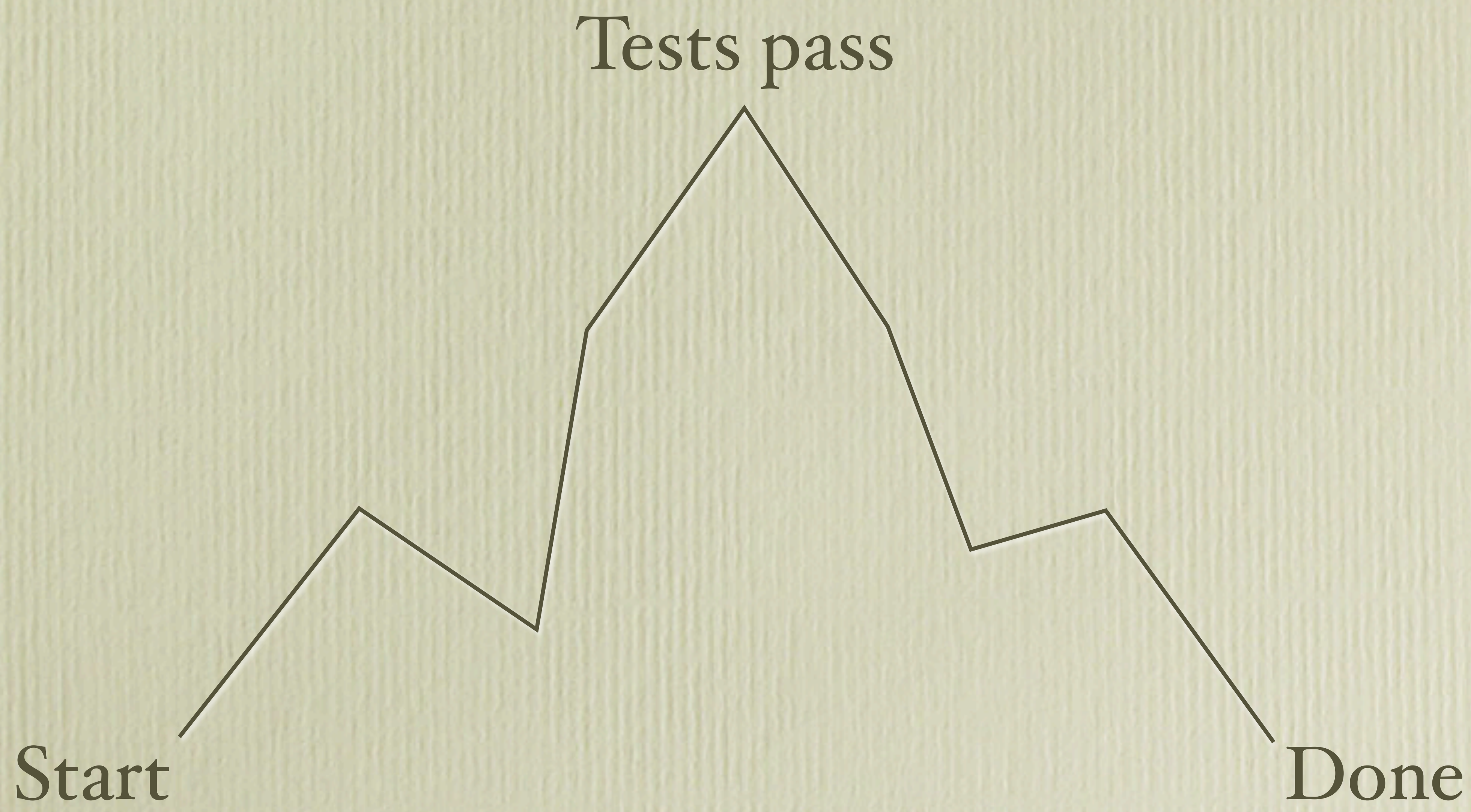
private variables

```
class lazySequenceOver⟦T⟧ (source: Collection⟦T⟧)
        filteredBy(predicate:Predicate1⟦T⟧) -> Enumerable⟦T⟧ {
    use enumerable⟦T⟧
    class iterator {
        use iteratorOver⟦T⟧ (source.iterator) filteredBy (predicate)
    }
    method asDebugString { "a lazy sequence filtering {source}" }
}
```

When are you done?



When are you done?



Iterators are tricky to implement

- but handy to use!
  - Some languages make it easier, e.g., Python:

```
def fibonacci(limit):       # The generator constructs an iterator
    a, b, c = 0, 1, 0
    while c < limit:
        yield a             # Note: yield, not return
        a, b, c = b, a+b, c+1

it = fibonacci(10)
while True:
    try:
        value = it.__next__()    ##  gets the next value; no effect.  Also next(it)
    except StopIteration:
        break
    it.__iter__()                ##  advances the iterator.  Also iter(it)
    print(value)

for v in fibonacci(10):        ## for stmt also uses iterator
    print(v)
```

Portland State University