Demystifying Haskell

An in-depth examination of the Fibonacci sequence.

Andrew Rademacher

July 15, 2014

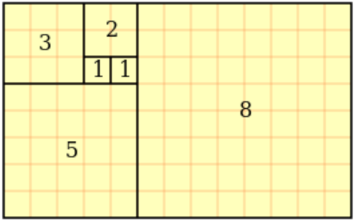
Objectives

- Demonstrate the extreme value of Haskell
- Bust myths about the "difficulty" of using Haskell
- Encourage further research and interest in Haskell

Prior knowledge in functional programming is not required, but programming experience is helpful.

The Fibonacci Sequence

- Sequence is infinite
- Sequence is self-referencing
- Values grow exponentially
- Values are always positive



$$\{1, 1, 2, 3, 5, 8...\}$$

$$F_n = F_{n-1} + F_{n-2}$$

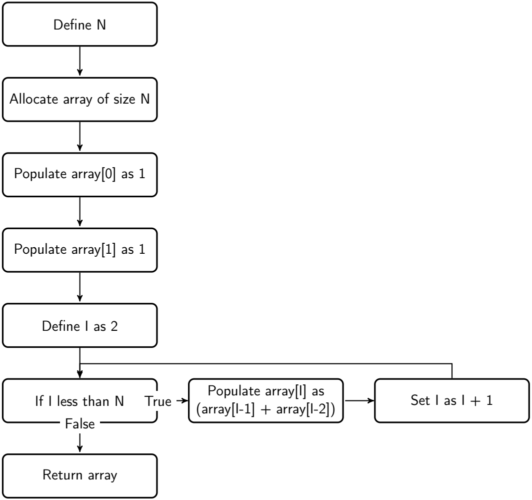
Traditional JavaScript Implementation

```
function getFibs (n) {
    var fibs = [1, 1];
    for (var i = 2; i < n; i++)
        fibs.push(fibs[fibs.length - 1] +
            fibs[fibs.length - 2]);

    return fibs;
}
```

Traditional Java Implementation

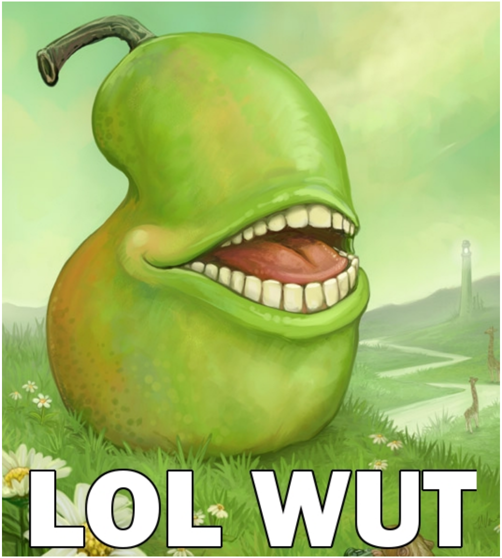
```java
public static BigInteger[] getFibs(int n) {
    BigInteger[] fibs = new BigInteger[n];
    fibs[0] = BigInteger.valueOf(1);
    fibs[1] = BigInteger.valueOf(1);
    for (int i = 2; i < n; i++)
        fibs[i] = fibs[i - 1].add(fibs[i - 2]);

    return fibs;
}
```

Imperative Process



New Fangled Haskell Implementation

```
fibs = 1 : 1 : [ a + b | (a, b) <- zip fibs (tail
    fibs) ]
```


LOL WUT

Values != Variables

- Haskell has no variables
- Values can be bound to a name
- Values can only be assigned once
- Values never change

```
fibs = 1 // Valid
    assignment
fibs = 2 // Invalid
    assignment
```

Lists

- Lists are singly-linked
- Lists are homogeneous
- Lists are immutable
- Lists use head insertion
- Two lists can share structure

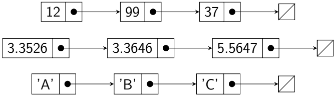


Figure : Haskell Lists

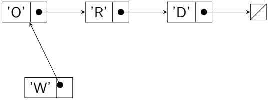


Figure : List Insertion

Basic List Operations

```
listA = [1, 2, 3, 4, 5]          // Literal
listB = [6, 7, 8]                // Literal

listC = listA ++ listB           // Concatenation
// listC = [1, 2, 3, 4, 5, 6, 7, 8]

listD = 0 : listC                // Insert
// listD = [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

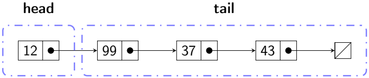
List Functions

head

The first element in a list.

tail

All remaining elements in a list.



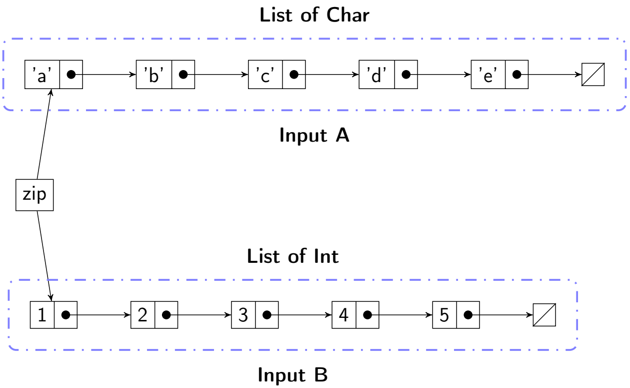
```
listA = [1, 2, 3, 4]

headA = head listA
//  headA = 1
tailA = tail listA
//  tailA = [2, 3, 4]
```

Tuples

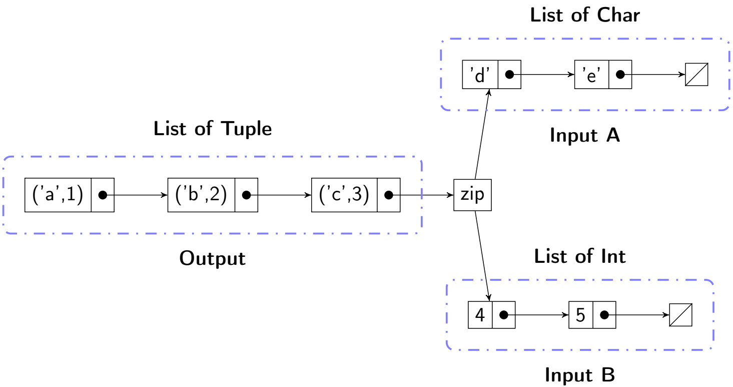
- Tuples are atomic (like an int or boolean).
- Tuples are heterogeneous.
- Tuples are immutable.
- Tuples allow random access.
- Tuples are usually small (less than 4 elements).

```
coord =
    (23.235, -345.345)
name  =
    ("John", "Edward",
        "Doe")
person =
    ("Jane Doe", 24,
        Female)
complex =
    ("SGF", [2,3,4],
        [2.3,2.5])
```

Zip (Lists and Tuples)

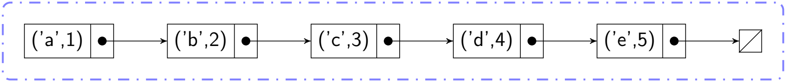


Zip (Lists and Tuples)



Zip (Lists and Tuples)

List of Tuple



('a',1) ('b',2) ('c',3) ('d',4) ('e',5)

Output

Zip (Lists and Tuples)

```
// zip :: [a] -> [b] -> [(a,b)]

lLetters = [’a’, ’b’, ’c’, ’d’]
lNumbers = [1, 2, 3, 4]

lZipped = zip lLetters lNumbers
// lZipped = [(’a’,1), (’b’,2), (’c’,3), (’d’,4)]

lReversed = zip lNumbers lLetters
// lReversed = [(1,’a’), (2,’b’), (3,’c’), (4,’d’)]
```

List Comprehension

```
powers = [ 2^x | x <- [1, 2, 3, 4, 5] ]
// powers = [2, 4, 8, 16, 32]
```

| "for every"

<- "that is an element of"

Lazy Evaluation (Haskell's secret sauce).

```
1  a = 2
2  b = 3 + 6
3
4  main = do
5         putStrLn "Hello, World!"
6         putStrLn (show (a + b))
```

b The value of 3 + 6 is not evaluated immediately.

putStrLn This call prints to the screen, and requires full evaluation.

show Converts arbitrary types to a string.

- The value of b is not evaluated until line 6 when it is printed to the console.

—TO
VICTORY!
Printed by the MOD Ministry of Defence Room 73 Public Relations Office, [illegible] Test Print awaiting approval. Not for public display until further notice.
Ref: [illegible]
©BBC
bbc.co.uk/doctorwho

Begin with a List

```
fibs = [1, 1, 2, 3, 5]
```

Figure : Define the sequence explicitly.

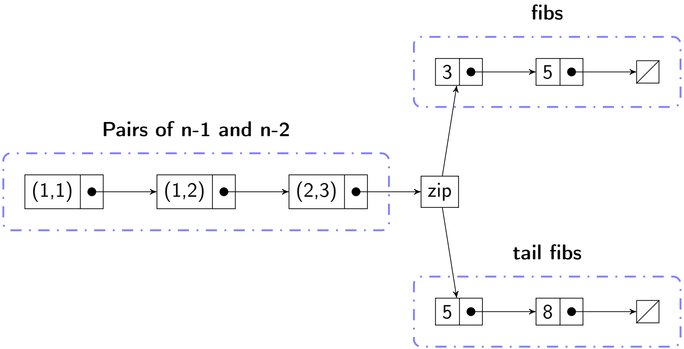
```
fibs = 1 : 1 : [2, 3, 5]
```

Figure : Define only the first two fibs, then use some programmatic definition for the rest of the sequence.

Playing with Functions

```
fibs = 1 : 1 : [2, 3, 5, 8]
// fibs = [1, 1, 2, 3, 5, 8]

t = tail fibs
// t = [1, 2, 3, 5, 8]

z = zip fibs (tail fibs)
// z = [(1,1), (1,2), (2,3), (3,5), (5,8)]
```

Playing with Functions



Building a Comprehension

```
fibs = 1 : 1 : [2, 3, 5, 8]

fibs2 = [ a + b | (a, b) <- zip fibs (tail fibs) ]
// fibs2 = [2, 3, 5, 8, 13]
```

zip The "zip fibs (tail fibs)" statement produces a list of tuples.

(a, b) We bind each tuple to the names "a" and "b".

a + b Each item in fibs2 is the sum of each tuple.

- Each element of fibs2 is the sum of "a" and "b", where "a" and "b" are elements of zipping together fibs and the tail of fibs.

Merging Fibs and Fibs2

```haskell
fibs = 1 : 1 : [ a + b | (a, b) <- zip fibs (tail
    fibs) ]
```

Properties of Fib Sequence

- Sequence is infinite.
- Sequence is self-referencing.
- Values grow exponentially.
- Values are always positive.

Properties of Haskell Implementation

- fibs is and infinite sequence.
- fibs references itself in its definition.
- Integer values in Haskell support infinite precision.
- It is impossible for a negative value to be produced.

Contact and GitHub

	Contact Me
Email	andrewrademacher@gmail.com
Twitter	@AndRademacher
GitHub	https://github.com/AndrewRademacher
LinkedIn	http://www.linkedin.com/in/andrewrademacher
	Get the Presentation
Repo	https://github.com/AndrewRademacher/haskell-labs/tree/master/fibs
Slides	https://github.com/AndrewRademacher/haskell-labs/tree/master/fibs/FibsPresentation.pdf