Demystifying Haskell

An in-depth examination of the Fibonacci sequence.

Andrew Rademacher

January 11, 2014

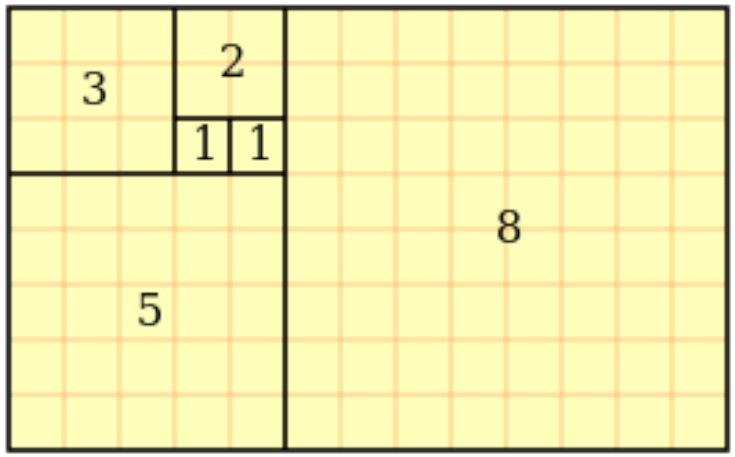
Objectives

- 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 require, but programming experience is helpful.

The Fibonacci Sequence

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



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

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

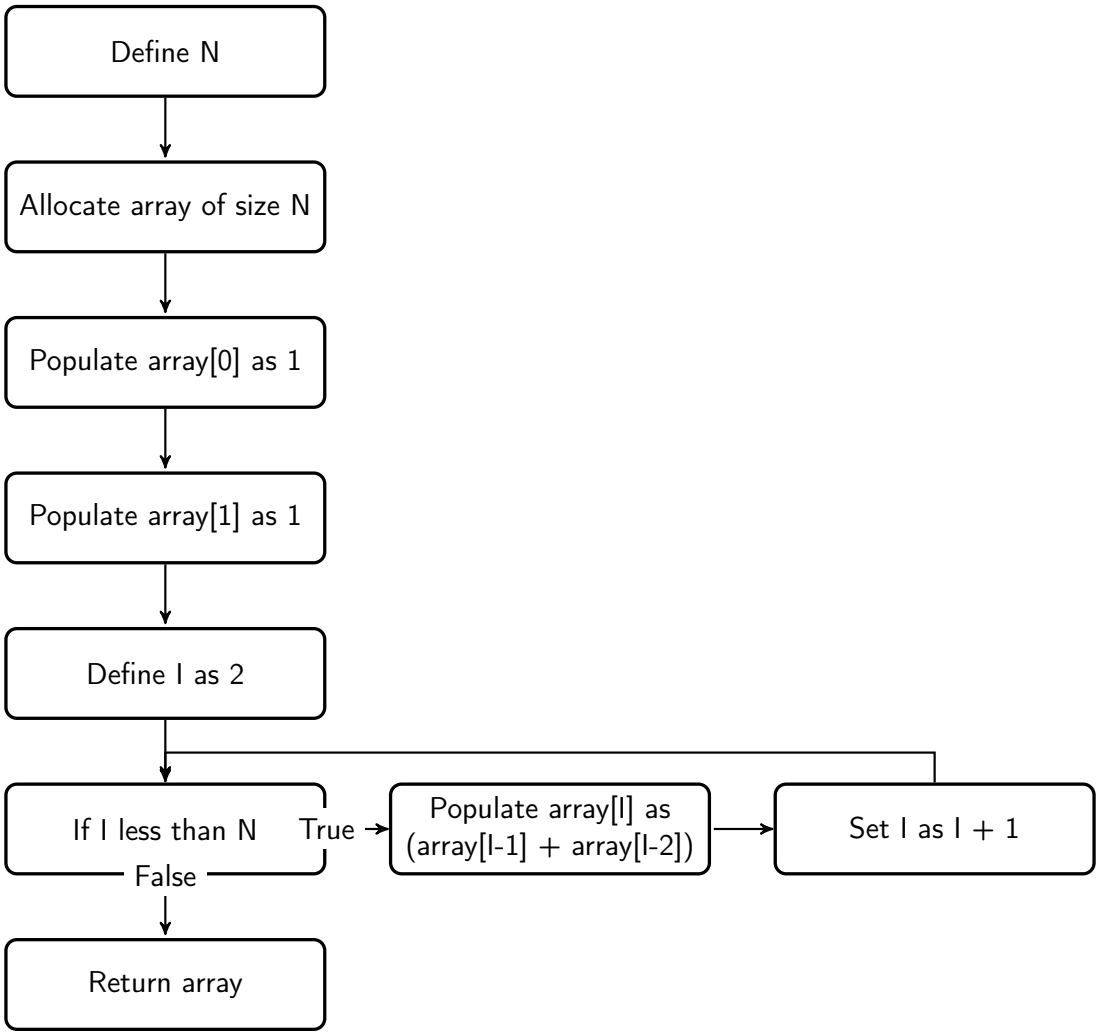
Traditional JavaScript Implementation

```
1  function getFibs (n) {
2      var fibs = [1, 1];
3      for (var i = 2; i < n; i++)
4          fibs.push(fibs[fibs.length - 1] +
               fibs[fibs.length - 2]);
5
6      return fibs;
7  }
```

Traditional Java Implementation

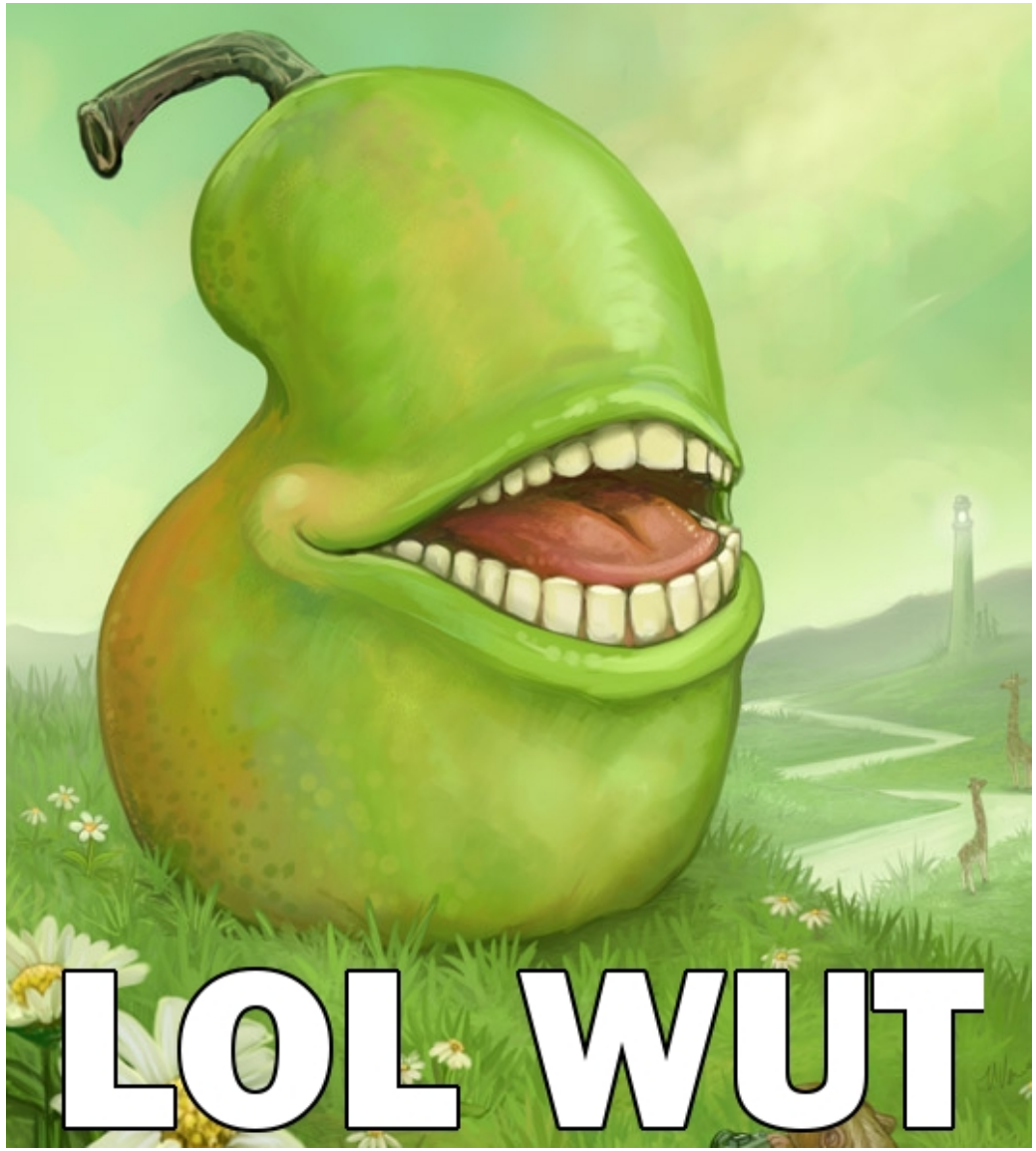
```
1  public static BigInteger[] getFibs(int n) {
2      BigInteger[] fibs = new BigInteger[n];
3      fibs[0] = BigInteger.valueOf(1);
4      fibs[1] = BigInteger.valueOf(1);
5      for (int i = 2; i < n; i++)
6          fibs[i] = fibs[i - 1].add(fibs[i - 2]);
7
8      return fibs;
9  }
```

Imperative Process



New Fangled Haskell Implementation

```
1  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

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

Lists

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

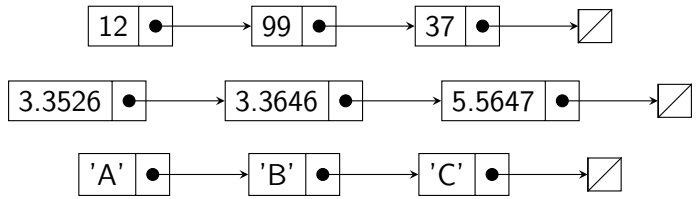


Figure: Haskell Lists

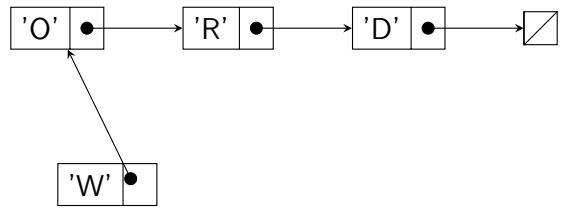


Figure: List Insertion

Basic List Operations

```
1  listA = [1, 2, 3, 4, 5]          // Literal
2  listB = [6, 7, 8]                // Literal
3
4  listC = listA ++ listB           // Concatenation
5  // listC = [1, 2, 3, 4, 5, 6, 7, 8]
6
7  listD = 0 : listC                // Insert
8  // listD = [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

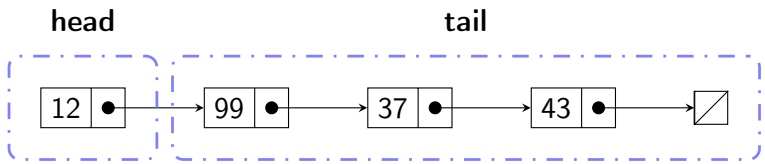
List Functions

head

The first element in a list.

tail

All remaining elements in a list.



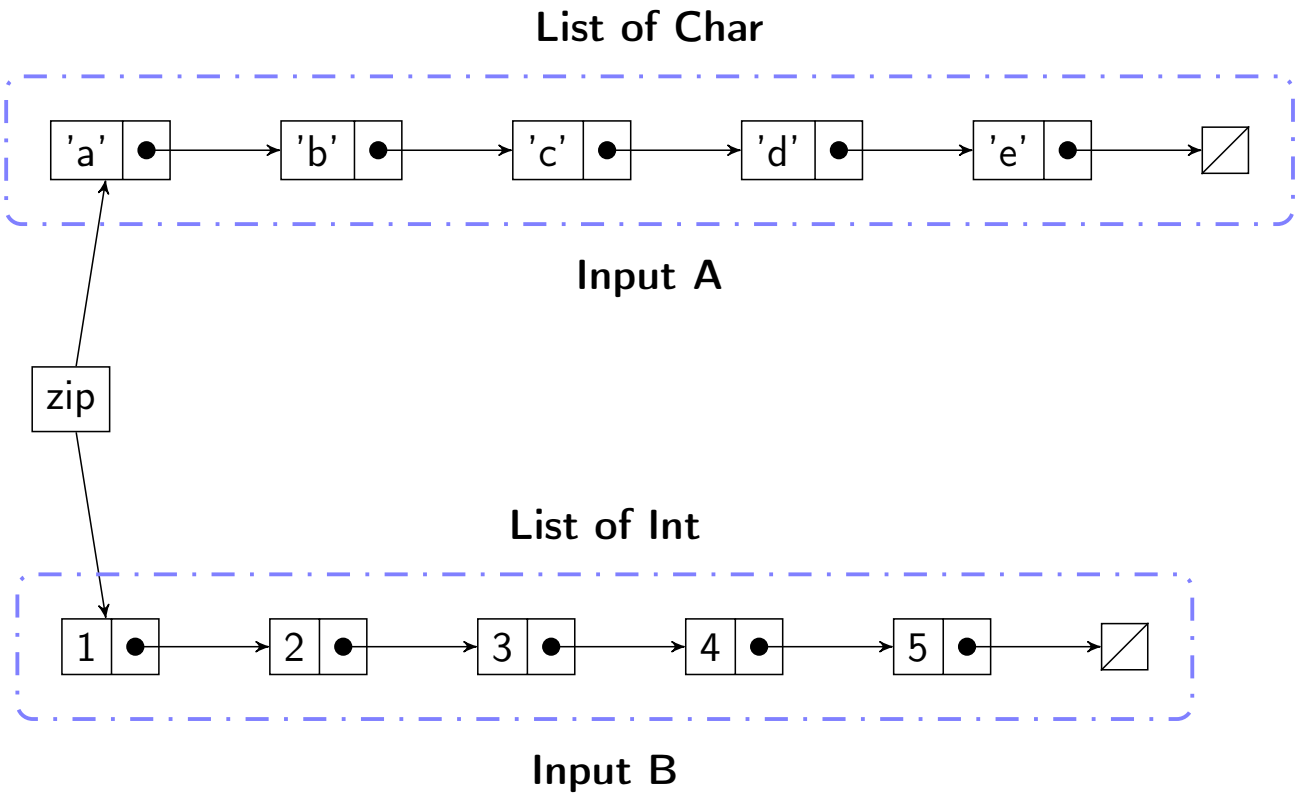

```
1  listA = [1, 2, 3, 4]
2
3  headA = head listA
4  //  headA = 1
5  tailA = tail listA
6  //  tailA = [2, 3, 4]
```

Tuples

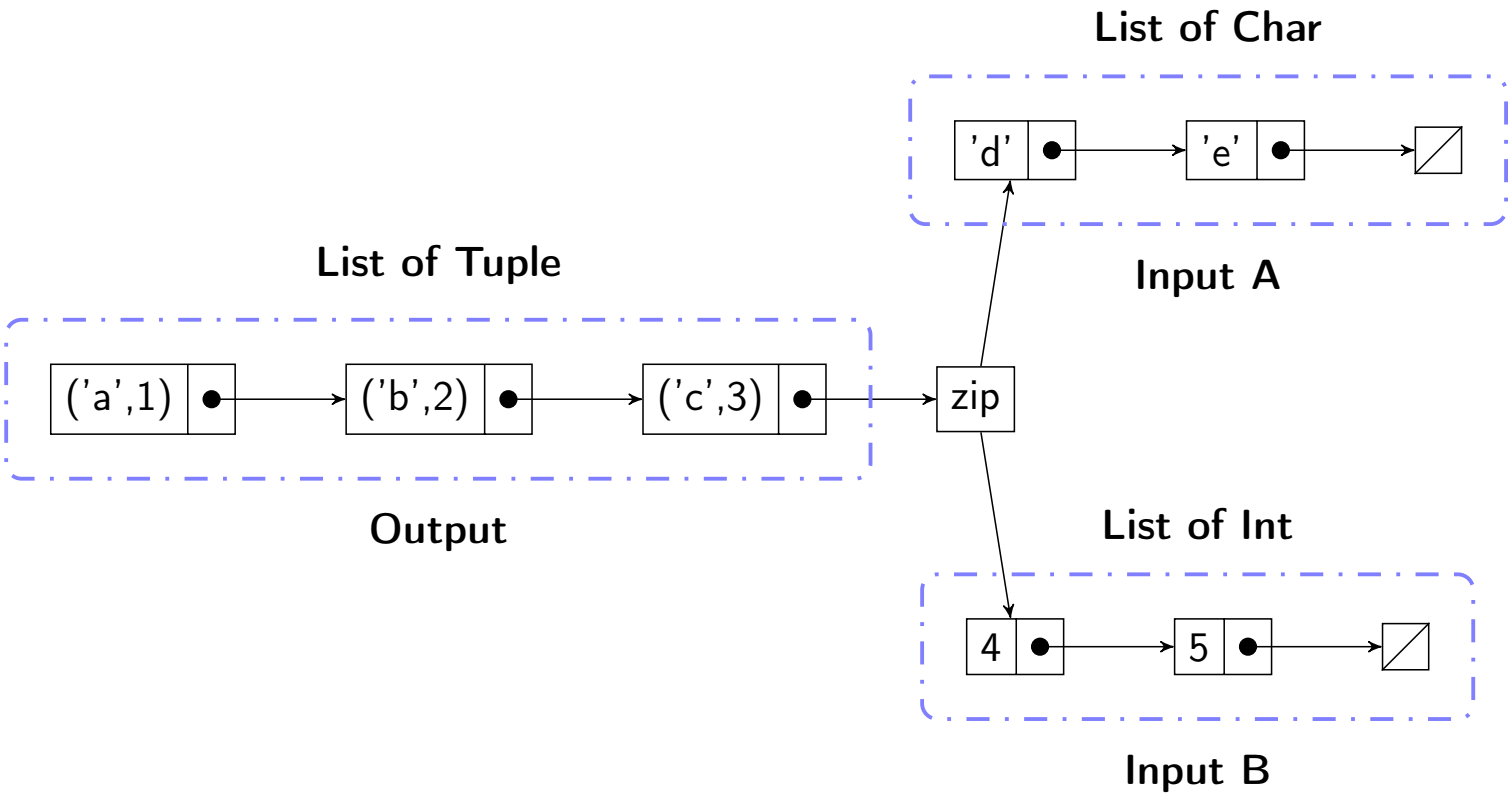
- Tuples are atomic (like an int or boolean).
- Tuples are heterogeneous.
- Tuples are immutable.
- Tuples allow random access.
- Tuples are usually small (less than 4 elements).

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

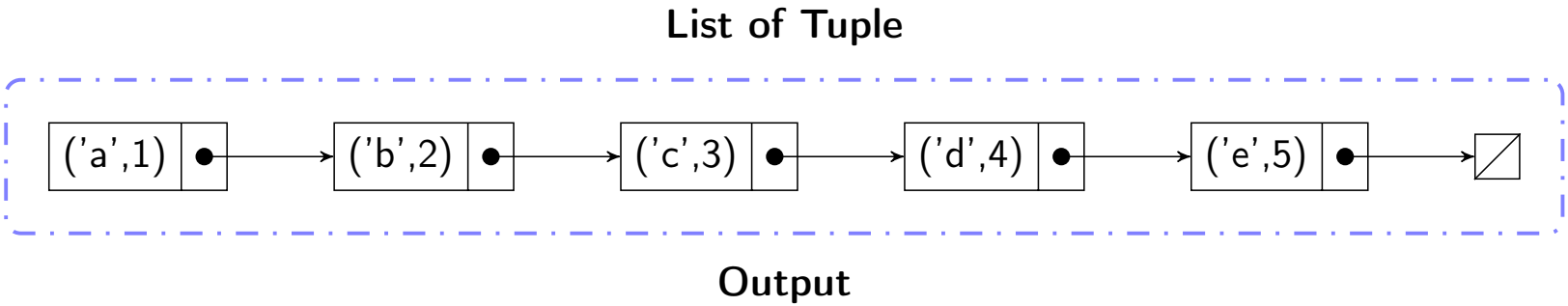
Zip (Lists and Tuples)



Zip (Lists and Tuples)



Zip (Lists and Tuples)



Zip (Lists and Tuples)

```
1  // zip :: [a] -> [b] -> [(a,b)]
2
3  lLetters = ['a', 'b', 'c', 'd']
4  lNumbers = [1, 2, 3, 4]
5
6  lZipped = zip lLetters lNumbers
7  // lZipped = [('a',1), ('b',2), ('c',3), ('d',4)]
8
9  lReversed = zip lNumbers lLetters
10 // lReversed = [(1,'a'), (2,'b'), (3,'c'), (4,'d')]
```

List Comprehension

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

bbc.co.uk/doctorwho

Begin with a List

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

Figure: Define the sequence explicitly.

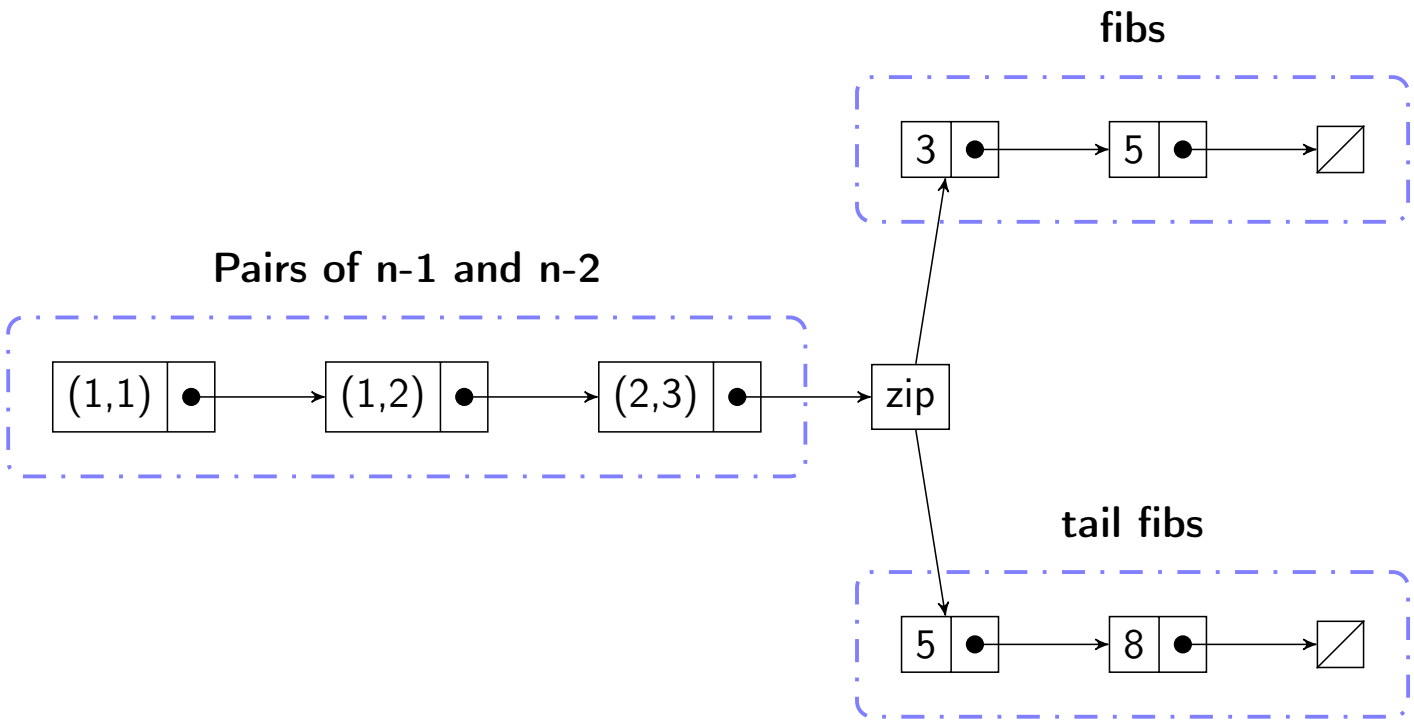
```
1 fibs = 1 : 1 : [2, 3, 5]
```

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

Playing with Functions

```
1  fibs = 1 : 1 : [2, 3, 5, 8]
2  // fibs = [1, 1, 2, 3, 5, 8]
3
4  t = tail fibs
5  // t = [1, 2, 3, 5, 8]
6
7  z = zip fibs (tail fibs)
8  // z = [(1,1), (1,2), (2,3), (3,5), (5,8)]
```

Playing with Functions



Building a Comprehension

```
1  fibs = 1 : 1 : [2, 3, 5, 8]
2
3  fibs2 = [ a + b | (a, b) <- zip fibs (tail fibs) ]
4  // 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

```
1 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

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