The Fibonnacci Sequence

```haskell
-- nth number in the Fibonnacci sequence
fib n = fibl
  where
    (fibl, fib2) = fibs n
-- (nth, (n+1)th) in Fib. seq.
fibs 0 = (1,1)
fibs n = (f2, f1 + f2)
  where
    (f1,f2) = fibs (n-1)
```

The F sequence where F = Fibonaccii

```haskell
-- file: fib.hs
fib n = fibl
  where
    (fibl, fib2) = fibs n
  fibs m
    | m <= 1 = (1,1)
    | 0 < m = (f2, f1 + f2)
      where
        (f1,f2) = fibs (m-1),
```

Nested Definitions

"Local" definitions can be made using the where keyword:

```haskell
maxOf3 :: Int -> Int -> Int -> Int
maxOf3 x y z = maxOf2 u z
  where
    u = maxOf2 x y
```

which is equivalent to:

```haskell
maxOf3 x y z = maxOf2 (maxOf2 x y) z
```

... Or

```haskell
fib n = fibl
  where
    (fibl, fib2) = fibs n
fibs n
  | n <= 0 = (1,1)
  | 0 < n = (f2, f1 + f2)
    where
      (f1,f2) = fibs (n-1)
```

```haskell
fib n = f1
  where
    (f1, f2) = fibs n
  fibs n
    | n <= 1 = (1,1)
    | 0 < n = (f2, f1 + f2)
      where
        (f1,f2) = fibs (n-1),
```
Local is Hidden

```
Main> :l fib.hs
...
Main> fibs 5
ERROR - Undefined variable "fibs"
Main>
```

Summary

Key topics:
- Local definitions (where)

Next: Lists

Lists

- bread
- milk
- butter
- onions
- orange juice
- t-bone steaks
- chocolate
- Attendance Sheets
- League tables
- top tens
- the Fibonacci sequence
- ...

Lists in Haskell

A list is a sequence of values, in a particular order

Lists in Haskell

A list is a sequence of values, in a particular order

Lists are homogeneous:
- all the elements in a list (in Haskell) have the same type

Lists in Haskell

A list is a sequence of values, in a particular order

Lists are homogeneous:
- all the elements in a list (in Haskell) have the same type

Lists are dynamic:
- unlike arrays in imperative languages, the length of a list is not constrained (and Haskell allows infinite lists)
### Notations for Lists

Haskell has three notations for lists:

- **constructors:**
  
  - `1 : 1 : 2 : 3 : 5 : 8 : []`

- "square brackets" notation:
  
  - `[1, 1, 2, 3, 5, 8]`

- strings (only for lists of chars):
  
  - `['h', 'e', 'l', 'l', 'o'] = "hello"`

### Functions on Lists

**Length:**

<table>
<thead>
<tr>
<th>Prelude &gt; <code>length [1, 1, 2, 3, 5, 8]</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prelude &gt; <code>length &quot;&quot;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

**Concatenation (append):**

- `[: [1, 1, 2, 3] ++ [5, 8]`  
  - `[1, 1, 2, 3, 5, 8]`

- `[] ++ [1] ++ [2]`  
  - `[1, 2]`

### More Functions on Lists

**Heads and tails**

<table>
<thead>
<tr>
<th>Prelude &gt; <code>head [1, 2, 4]</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prelude &gt; <code>tail [1, 2, 4]</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[2, 4]</code></td>
</tr>
</tbody>
</table>

**Heads and tails**

<table>
<thead>
<tr>
<th>Prelude &gt; <code>head []</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR: ...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prelude &gt; <code>tail []</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR: ...</td>
</tr>
</tbody>
</table>

**Heads and tails**

<table>
<thead>
<tr>
<th>Prelude &gt; <code>head [1, 2, 4]</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prelude &gt; <code>tail [1, 2, 4]</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[2, 4]</code></td>
</tr>
</tbody>
</table>

**Heads and tails**

<table>
<thead>
<tr>
<th>Prelude &gt; <code>head []</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR: ...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prelude &gt; <code>tail []</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR: ...</td>
</tr>
</tbody>
</table>
Yet More Functions on Lists

Prelude> take 3 "catflap"
"cat"
Prelude>

Prelude> drop 2 ['i','n','t','e','n','t']
"tent"
Prelude>

Prelude> reverse 'p'::'o'::'o'::'l':[]
"loop"
Prelude>

Prelude> zip [1,2,3,4,5] "cat"
[(1,'c'),(2,'a'),(3,'t')]

List Constructors I

A list is either:

- empty
- or
- an element of a given type together with a list of elements of the same type

in BNF: \[ a ::= [] \mid a : [a] \]

List Constructors II

"[]" and "::" are constructors for lists:

- they are primitive operators (i.e., not evaluated)
- all lists are built from these operators (and elements of the "parameter type")
Constructors and Pattern-Matching

Constructors can be used in Pattern-Matching:

```
isempty [] = True
isempty anythingElse = False
```

```
head [] = error "head []"
head (x:xs) = x
```

```
length [] = 0
length (x : xs) = 1 + length xs
```

---

Evaluating Recursive Functions

```
length [] = 0
length (x : xs) = 1 + length xs
```

```
length [1 : 2 : 4 : []]

⇒ [x ← 1 , xs ← 2 : 4 : []]
```

```
1 + length (2 : 4 : [])

⇒ [x ← 2 , xs ← 4 : []]
```

```
1 + 1 + length (4 : [])
```

---

Evaluating Recursive Functions

```
length [] = 0
length (x : xs) = 1 + length xs
```

```
length [1 : 2 : 4 : []]

⇒ [x ← 1 , xs ← 2 : 4 : []]
```

```
1 + length (2 : 4 : [])

⇒ [x ← 2 , xs ← 4 : []]
```

```
1 + 1 + length (4 : [])
```

---

Evaluating Recursive Functions

```
length [] = 0
length (x : xs) = 1 + length xs
```

```
length [1 : 2 : 4 : []]

⇒ [x ← 1 , xs ← 2 : 4 : []]
```

```
1 + length (2 : 4 : [])

⇒ [x ← 2 , xs ← 4 : []]
```

```
1 + 1 + length (4 : [])
```

---

Evaluating Recursive Functions

```
length [] = 0
length (x : xs) = 1 + length xs
```

```
length [1 : 2 : 4 : []]

⇒ [x ← 1 , xs ← 2 : 4 : []]
```

```
1 + length (2 : 4 : [])

⇒ [x ← 2 , xs ← 4 : []]
```

```
1 + 1 + length (4 : [])
```
Evaluating Recursive Functions

\[
\text{length} \ [\ ] = 0 \\
\text{length} \ (x : x\ s) = 1 + \text{length} \ x\ s
\]

\[
\text{length} \ (1 : 2 : 4 : [\ ]) \\
\Rightarrow \ [x \leftarrow 1 , \ x\ s \leftarrow 2 : 4 : [\ ]] \\
1 + \text{length} \ (2 : 4 : [\ ]) \\
\Rightarrow \ [x \leftarrow 2 , \ x\ s \leftarrow 4 : [\ ]] \\
1 + 1 + \text{length} \ (4 : [\ ]) \\
\Rightarrow \ [x \leftarrow 4 , \ x\ s \leftarrow [\ ]] \\
1 + 1 + 1 + \text{length} \ [\ ] \\
\Rightarrow \ 1 + 1 + 1 + 0
\]

Evaluating Recursive Functions

\[
\text{length} \ [\ ] = 0 \\
\text{length} \ (x : x\ s) = 1 + (\text{length} \ x\ s)
\]

\[
\text{length} \ (1 : 2 : 4 : [\ ]) \\
\Rightarrow \ [x \leftarrow 1 , \ x\ s \leftarrow 2 : 4 : [\ ]] \\
1 + \text{length} \ (2 : 4 : [\ ]) \\
\Rightarrow \ [x \leftarrow 2 , \ x\ s \leftarrow 4 : [\ ]] \\
1 + 1 + \text{length} \ (4 : [\ ]) \\
\Rightarrow \ [x \leftarrow 4 , \ x\ s \leftarrow [\ ]] \\
1 + 1 + 1 + \text{length} \ [\ ] \\
\Rightarrow \ [\ ]
\]

length without Pattern-Matching

length can be defined without pattern-matching, but the definition is more difficult to read:

\[
\text{length} \ x\ s \\
\begin{cases} 
  \text{xs} = [] & = 0 \\
  \text{otherwise} & = 1 + \text{length} \ (\text{tail} \ x\ s)
\end{cases}
\]

Summary

Key topics:
- Lists (3 notations)
- Constructors
- Pattern-matching
- Recursion

Next: Recursion and List Comprehensions