

Comp 205: Comparative Programming Languages

Functional Programming Languages:

More Haskell

- Nested definitions
- Lists

Lecture notes, exercises, etc., can be found at:
www.csc.liv.ac.uk/~grant/Teaching/COMP205/

Nested Definitions

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

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

which is equivalent to:

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

The Fibonacci Sequence

```
-- nth number in the Fibonacci sequence
fib n = fib1
      where
        (fib1, 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)
```

... Or

```
fib n = fib1
      where
        (fib1, fib2) = fibs n

fibs n
| n <= 0 = (1,1)
| n > 0  = (f2, f1 + f2)
          where
            (f1,f2) = fibs (n-1)
```

The \mathbb{F} sequence where \mathbb{F} = Fibonacci

```
-- file: fib.hs

fib n = fib1
      where
        (fib1, fib2) = fibs n
        fibs m
          | m < 1  = (1,1)
          | 0 < m = (f2, f1 + f2)
          where
            (f1,f2) = fibs (m-1),
```

The \mathbb{F} sequence where \mathbb{F} = Fibonacci

```
-- file: fib.hs

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

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

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:

```
Prelude> length [1, 1, 2, 3, 5, 8]
6
Prelude> length ""
0
```

Concatenation (append):

```
Prelude> [1, 1, 2, 3] ++ [5, 8]
[1, 1, 2, 3, 5, 8]
Prelude> [] ++ [1] ++ [2]
[1,2]
```

More Functions on Lists

Heads and tails

```
Prelude> head [1, 2, 4]
1
Prelude>
```

More Functions on Lists

Heads and tails

```
Prelude> head [1, 2, 4]
1
Prelude> tail [1, 2, 4]
[2,4]
Prelude>
```

More Functions on Lists

Heads and tails

```
Prelude> head [1, 2, 4]
1
Prelude> tail [1, 2, 4]
[2,4]
Prelude> head []
ERROR: ...
Prelude>
```

More Functions on Lists

Heads and tails

```
Prelude> head [1, 2, 4]
1
Prelude> tail [1, 2, 4]
[2,4]
Prelude> head []
ERROR: ...
Prelude> tail []
ERROR: ...
```

Yet More Functions on Lists

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

Yet More Functions on Lists

```
Prelude> take 3 "catflap"  
"cat"  
Prelude> drop 2 ['i','n','t','e','n','t']  
"tent"  
Prelude>
```

Yet More Functions on Lists

```
Prelude> take 3 "catflap"  
"cat"  
Prelude> drop 2 ['i','n','t','e','n','t']  
"tent"  
Prelude> reverse 'p':'o':'o':'l':[]  
"loop"  
Prelude>
```

Yet More Functions on Lists

```
Prelude> take 3 "catflap"  
"cat"  
Prelude> drop 2 ['i','n','t','e','n','t']  
"tent"  
Prelude> reverse 'p':'o':'o':'l':[]  
"loop"  
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 : [])  
⇒
```

Evaluating Recursive Functions

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

```
length (1 : 2 : 4 : [])  
⇒ { x ← 1 , xs ← 2 : 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 : [])
```

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 : [] }
```

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 : [])
=> { x ← 4 , xs ← [] }
1 + 1 + 1 + length []
=>
```

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 : [])
=> { x ← 4 , xs ← [] }
1 + 1 + 1 + length []
=> {}
```

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 : [])
=> { x ← 4 , xs ← [] }
1 + 1 + 1 + length []
=> {}
1 + 1 + 1 + 0
```

length without Pattern-Matching

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

```
length xs
| xs == [] = 0
| otherwise = 1 + length (tail xs)
```

Summary

Key topics:

- Lists (3 notations)
- Constructors
- Pattern-matching
- Recursion

Next: Recursion and List Comprehensions