Recursion and problem solving

Functional Programming in Haskell

Prof Hans Georg Schaathun

Høgskolen i Ålesund

18th January 2016

Outline

NTNU

Problem Solving

How do you eat an elephant?

Problem Solving

How do you eat an elephant?

1. Take one small piece and eat it.
2. If there is more elepant left, then repeat from start.

Functional Programming

How do you write a functional program?

Functional Programming

How do you write a functional program?

1. Write one small, useful function.
2. If your last function does not complete the program, then repeat from start.

Outline

Simple functions

— Functions can be exceedingly simple
  - `addTwo :: Integer -> Integer`
  - `addTwo a = a + 2`

— Functions can have several arguments
  - `polynomial :: Double -> Double -> Double`
  - `polynomial a b = 2*a^2 + 3*b^2 + a*b + a + 10*b - 50`

— Functions may be exceedingly messy

— Functions should be simple and comprehensible

— Ten simple functions is better than one incomprehensible one

Pattern Matching

- Function evaluation using pattern matching
  - matching actual arguments in the function call
  - ... against formal arguments in the function definition
- For instance
  - Definition: `mul a b = a*b`
  - Call: `mul 5 10`
    1. $a \leftarrow 5$
    2. $b \leftarrow 10$

Patterns with Constants

— Formal arguments need not be simple symbols
  - `funny 0 b = -b`
  - `funny a 0 = a^2`
  - `funny a b = b*a`

— The call `funny 5 10` uses the third defintion
  - First definition invalid, because 5 does not match 0
  - Second definition invalid, because 10 does not match 0
  - $a \leftarrow 5$, $b \leftarrow 10$ is OK

— The first valid pattern is used

— A common example
  - `myXOR False x = x`
  - `myXOR True x = not x`

Guards

- Pattern matching allows definition of multiple cases
  - not all case handling can be done with patterns
- `myAbs a | a < 0 = -a`
- `myAbs a | a > 0 = a`
- `myAbs a | otherwise = 0`

$$|a| = \begin{cases} -a, & a < 0, \\ a, & a > 0, \\ 0, & \text{otherwise} \end{cases}$$

- The first guard which evaluates to true is used.
- `otherwise` is an alias for True

Combining Guards in one Definition

— Usually we combine all guard in one definition

```
myAbs a | a < 0       = -a
        | a > 0       = a
        | otherwise   = 0
```

$$|a| = \begin{cases} -a, & a < 0, \\ a, & a > 0, \\ 0, & \text{otherwise} \end{cases}$$

— Note the indentation of the guard lines (Lines 2–3)

- this is necessary to let Haskell know that it is part of the same definitions as Line 1.

Local definitions

- Auxiliary definitions are often seen in mathematics

$$f(x) = \cos y + \sin y, \quad \text{where} \tag{1}$$
$$y = x^2. \tag{2}$$

- Local definitions in Haskell follow the same pattern

```
f x = cos y + sin y
      where y = x^2
```

- Local definitions can only be used in the definition where they appear
- The linebreak is optional, and can be placed elsewhere

Function types

- Functions of several parameters
  - `myAdd ::  Double -> Double -> Double`
- Why do we use arrows twice?

Function types

- Functions of several parameters
  - `myAdd :: Double -> Double -> Double`
- Why do we use arrows twice?
- Actually, `myAdd` takes one Double
  - returns a function of type `Double -> Double`
  - ... which in turn takes a second double to return the third double

Function types

- Functions of several parameters
  - **`myAdd :: Double -> Double -> Double`**
- Why do we use arrows twice?
- Actually, **`myAdd`** takes one Double
  - returns a function of type **`Double -> Double`**
  - ... which in turn takes a second double to return the third double
- Partial application is possible
  - **`myAdd 3`** is a function **`Double -> Double`**

```
*Main> :type myAdd 3
myAdd 3 :: Double -> Double
```

Outline

Modularation

- Problems are always solved in parts
- A module is a part solution
  - functional programs: functions
  - OO programming: classes (object types)
  - mathematical arguments:
    1. quantities
    2. functions
    3. concepts
- Each module must be easy to understand
  - intuitive purpose
  - comprehensible definition
- Modules may be defined in terms of other modules

Functional programming

Outline

NTNU

Recursion

- Many functions are defined in terms of themselves
- Fibonacci sequence
  - $f_0 = 1$
  - $f_1 = 1$
  - $f_i = f_{i-1} + f_{i-2}$ when $i \geq 2$
- This is called recurrence

```
f 0 = 1
f 1 = 1
f n = f (n-1) + f (n-2)
```

The Bisection Method

- Solve an equation $0 = f(x)$
- Linear and quadratic equations are simple
- For many other equations we need numeric solutions
- The bisection method is one of the simplest
- Requires a known interval (l, u) to search for a solution
  - $f(l) \cdot f(u) < 0$
- If $u - l$ is very small, then either u or l is an approximate solution

The Bisection Method

- If $u - l$ is very small,
  - then either u or l is an approximate solution
- If $u - l$ is not small enough,
  - find $m = (u + l)/2$
  - is the root in (l, m) or in (m, u)?
  - repeat recursively on half the interval

Outline

Motivation

Defining functions

Modularisation

Recursion

Summary

Summary

- Split a problem into smaller pieces
  - standard approach to problem solving
- When the subproblem is simple enough, write a function
- Combine simple functions to solve larger problems
- Often functions can call themselves recursivly
  - standard way to define functions in any paradigm
  - necessary way to get iteration in functional programming
  - common way to define mathematical functions