1 Slides

1.1 Week 1

1.1.1 FP Marketing

What is a functional programming language?

- Basic notion of computation: the application of functions to arguments.
- Basic idea of program: writing function definitions
- Functional languages are declarative: more emphasis on *what* rather than *how*.

Defining Haskell values

- Function definitions are written as equations

-
  ```
  double x = x + x
  quadruple x = double (double x)
  ```
- compute the length of a list

  ```
  -- length, if empty, is zero
  length [] = 0
  -- length, if not empty, is one plus length of its "tail"
  length (x:xs) = 1 + length xs
  ```

  recursion is the natural way to describe repeated computation
- Haskell can infer types itself (Type Inference)

Type Polymorphism

- What is the type of `length`?

  ```
  > length [1,2,3]
  3
  > length ['a','b','c','d']
  4
  > length [[],[1,2],[3,2,1],[],[6,7,8]]
  5
  ```

- `length` works for lists of elements of arbitrary type `length :: [a] -> Int` Here '`a`' denotes a type variable, so the above reads as "`length` takes a list of (arbitrary) type `a` and returns an `Int`".
- A similar notion to "generics" in O-O languages, but builtin without fuss.

Laziness

- What's wrong with the following (recursive) definition ?

  ```
  from n =  n : (from (n+1))
  ```

  Nothing ! It just generates an infinite list of ascending numbers, starting from `n`.
- `take n list` — return first `n` elements of `list`.
- What is `take 10 (from 1)` ?

  ```
  > take 10 (from 1)
  [1,2,3,4,5,6,7,8,9,10]
  ```

- Haskell is a *lazy* language, so values are evaluated only when needed.

Program Compactness

- Sorting the empty list gives the empty list:

  ```
  qsort [] = []
  qsort (x:xs)
    = qsort [y | y <- xs, y < x]
      ++ [x]
      ++ qsort [z | z <- xs, z >= x]
  ```

- We have used Haskell list comprehensions `[y | y <- xs, y < x ]` "build list of `y`s, where `y` is drawn from `xs`, such that `y < x`"
- Try that in Java !

Stop! My Head hurts!

- Haskell is powerful, and quite different to most mainstream languages
- It allows very powerful programs to be written in a concise manner
- Functional languages originally developed for theorem provers and rewrite systems
- Very popular now for:
  - software static checkers, e.g., Facebook's `infer` (fbinfer.com)
  - quantitative analysis in financial services
  - Domain-Specific Languages (DSLs)
  - Front-end language handling and transformation.

1.1.2 Basics

Haskell Language Structure

- Haskell is built on top of a simple functional language (Haskell "Core")
- Haskell Core is itself built on top of an extended form of the λ-calculus, that has value, types, primitive operations, and pattern-matching added on.
- A lot of "syntactic sugar" is added
- A large collection of standard types and functions are predefined and automatically loaded (the Haskell "Prelude")
- There are a vast number of libraries that are also available
- See `www.haskell.org`

Patterns in Mathematics

In mathematics we often characterise something by laws it obeys, and these laws often look like patterns or templates:

$$\begin{array}{rcl} 0! & = & 1 \\ n! & = & n \times (n-1)!, \qquad n > 0 \end{array}$$

$$\begin{array}{rcl} len(\langle\rangle) & = & 0 \\ len(\ell_1 \frown \ell_2) & = & len(\ell_1) + len(\ell_2) \end{array}$$

Here $\langle\rangle$ denotes an empty list, and $\frown$ joins two lists together.

Pattern matching is inspired by this (but with some pragmatic differences).

Factorial! as Patterns

Math:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \end{aligned}$$

Haskell (without patterns):

```
factorial_nop n  =  if n==0 then 1 else n * factorial_nop (n-1)
```

Haskell (with patterns):

```
factorial 0  =  1
factorial n  =  n * factorial (n-1)
```

Formal argument `0` is shorthand saying check the argument to see if it is zero. If so, do my righthand side.

Formal argument `n` says, take the argument, and refer to it in my righthand side as `n`.

Lists in Haskell

Lists of things are a very common datatype in functional languages similar to Haskell.

There is a standard approach to constructing lists:

- An empty list using: `[]`
- Given a term `x` and a list `xs` we can construct a list consisting of `x` followed by `xs` as follows: `x:xs`

- So the list 1,2,3 can be built as `1:2:3:[]` Brackets show how it is built up: `1:(2:(3:[]))`
- Syntactic Sugar:
  - We can write `[1,2,3]` as a shorthand for the above list
  - Lists can contain characters: `['H','e','l','l','o']` For character lists we have more shorthand: `"Hello"`
- Here `[]` and `:` are list *constructors* For historical reasons, "`:`" is pronounced "cons".

Length with Patterns

Math:

$$\begin{array}{rcl} len(\langle\rangle) & = & 0 \\ len(\ell_1 \frown \ell_2) & = & len(\ell_1) + len(\ell_2) \\ len(\langle \_ \rangle) & = & 1 \\ len(\langle \_ \rangle \frown \ell) & = & 1 + len(\ell) \end{array}$$

Haskell:

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

The key idea in pattern-matching is that the syntax used to build values, can also be used to look at a value, determine how it was built, and extract out the individual sub-parts if required.

Compact "Truth Tables"

Patterns can be used to give an elegant expression to certain functions, for instance we can define a function over two boolean arguments like this:

```
myand True True = True
myand _    _    = False
```

Here the underscore pattern “`_`” is a wildcrd that matches anything.

Key point: patterns are matched in order, and the first one to succeed is used.

1.1.3 Types

Types

- Haskell is strongly typed — every expression/value has a well-defined type: `myExpr :: MyType` Read: “Value `myExpr` has type `MyType`”
- Haskell supports *type-inference:* we don’t have to declare types of functions in advance. The compiler can figure them out automatically.
- Haskell’s type system is *polymorphic*, which allows the use of arbitrary types in places where knowing the precise type is not necessary.
- This is just like *generics* in Java or C++ – think of `List<T>`, `Vector<T>`, etc.

More about Types

- Some Literals have simple pre-determined types. `’a’ :: Char` `"ab"’ :: String`
- Numeric literals are more complicated `1 :: ?` Depending on context, `1` could be an integer, or floating point number.
- This is common with many other languages where notation for numbers (and arithmetic operations) are often “overloaded”.
- Haskell has a standard powerful way of handling overloading (the `class` mechanism).

Atomic Types

We have some Atomic types builtin to Haskell:

`()` the unit type which has only one value, also written as `()`.

`Bool` boolean values, of which there are just two: `True` and `False` .

`Ordering` comparison outcomes, with three values: `LT`, `EQ`, and `GT`.

`Char` character values, representing Unicode characters.

`Int` fixed-precision integer type with at least the range $[-2^{29} \ldots 2^{29} - 1]$

`Integer` infinite-precision integer type

`Float` floating point number of precision at least that of IEEE single-precision

`Double` floating point number of precision at least that of IEEE double-precision

Composite Types

- A composite type is a type built on top of another type (or types).
- Haskell has three kinds of composite types:

  Functions have a type that indicates the type(s) of its input(s), and the type of its output.

  Tuples gather values of other types together in a convenient package

  Algebraic are data types allow to define types whose values can have more than one form.
- The types used within the above composite types can themselves be composite, *including function types*.

Tuples

- We can create pairs, triples, n-tuples of values in an ad-hoc way: `(1,2)`, `(1,'a',"Hi!")`
- The type of the pair `(42,'z')`, where `42` has type `Int` (say), is `(Int,Char)` e.g `(42,'z') :: (Int,Char)`
- In general, the type of an n-tuple `(val_1,val_2,...,val_n)` is `(type_1,type_2,...,type_n)` where `val_i :: type_i`, for `i` in $1 \ldots n$.
- We can use tuples as patterns in function definitions: `sumPair (a,b) = a + b`

Algebraic Data Types

We will discuss these later, but you have already seen one example of an algebraic data type!

Haskell lists are just such an abstract data type, with data that can have one of two forms:

- Either empty `[]`,
- or non-empty `(x:xs)`

Function Types

- A function type consists of the input type, followed by a right-arrow and then the output type `myFun :: MyInputType -> MyOutputType` (Analagous to writing $f : A \to B$ in math)
- For example, a function `sumInt` that adds up a list of `Integers`. `sumInt :: [Integer] -> Integer`
- Consider a rounding function that converts a floating point number to an fixed-width integer: `round :: Double -> Int`

Functions Are First-Class Citizens!

- In Haskell, functions are treated just like any other kind of value.
- They are not considered as distinct from "concrete" values like numbers, strings, lists, tuples
- We can pass functions into functions as input: `funTakingFunInput :: (a -> b) -> c`
- We can return functions from functions as results `funReturningFunction :: a -> (b -> c)`
- We can do both at ones `funFromFunToFun :: (a -> b) -> (c -> d)`
- We will see how this is done shortly (PS: it's not hard to do!)

Inferring Function Types

- The following two rules are enough to figure out function types for any Haskell function definition:

  FunDef Given a function declaration like `f x = e`, if `e` has type `b`, and (the use of) `x` (in `e`) has type `a`, then `f` must have type `a -> b`.

  FunUse Given a function application `f v`, if `f` has type `a -> b`, then `v` must have type `a`, and `f v` will have type `b`.

1.1.4 Guards

Definition by "Guards"

Often we want to have different equations for different cases.

Mathematically we sometimes write something like this:

$$signum(x) = \begin{cases} 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$$

Signum using Guards

Math:

$$signum(x) = \begin{cases} 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$$

Haskell:

```
signum x | x < 0  = -1
         | x == 0 =  0
         | x > 0  =  1
```

We are using a notation called *Guarded Conditionals*

Each predicate acts like a guard that determines when execution can pass through.

If all else fails...

Each guard is tested in turn, and the first one to match selects an alternative. This means that it is OK to have a guard that would always be true, as long as it is the *last* alternative.

So the previous definition could have been written like this:

```
signum x | x < 0   =  -1
         | x == 0  =   0
         | True    =   1
```

For readability the name `otherwise` is allowed as a synonym for `True`:

```
signum x | x < 0      =  -1
         | x == 0     =   0
         | otherwise  =   1
```

Now you understand leap-years!

We can use guards to select special cases in functions. This function is `True` when the year number is a leap year:

```
leapyear :: Int -> Bool
leapyear y | mod y 400 == 0  =  True   -- 2000 was
           | mod y 100 == 0  =  False  -- 1900 wasn't
           | mod y 4 == 0    =  True   -- 2020 was
           | otherwise       =  False  -- 2021 isn't
```

Patterns and Guards

Guards and patterns can be combined:

```
startswith _ [] = False
startswith c (x:xs) | x == c    = True
                    | otherwise = False
```

First the patterns are matched; when an equation is found the guards are evaluated in order in the usual way.

If no guard matches then we return to the pattern matching stage and try to find another equation.

Behaviour of `startwith`

The second argument of `startswith` is a list of things, while the first argument must have the same type as the things in the list. The first line matches the case when the second list argument is empty, and ignores the first argument, returning `False`

```
> startswith 1 []
False
```

If the second list argument is not empty, then the second pattern match succeeds, and we proceed to compare the first list element (here called `x`), with the first argument (here called `c`).

```
> startswith 42 [42,41,40]
True
```

We can also avoid using the guards: the following replacement for lines two and three of the function definition have the same effect:

```
startswith c (x:xs)  =  c == x
```

Factorial! as Patterns (again!)

This time we do factorial properly, with a check for negative numbers.

Math:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)!, \qquad n > 0 \end{aligned}$$

Haskell:

```
factorial 0           =  1
factorial n | n > 0  =  n * factorial (n-1)
```

1.1.5 Function Notation (I)

Function Notation (I)

Consider the definition and application/use of a function in mathematics:

$$f(x) \;\widehat{=}\; x+1 \qquad\qquad\qquad\qquad f(42)$$

In a C-like language we might write:

```
int f (int x) { return (x+1) }               f(42)
```

In Haskell we could write:

```
f1(x) = x+1                                            f1(42)
```

Usually, however, in Haskell, we write:

```
f2 x = x+1                                             f2 42
```

Function Notation (II)

Lets add a few more arguments:

$$g(x,y,z) \;\widehat{=}\; x+y+z \qquad\qquad\qquad\qquad g(42,57,99)$$

In a C-like language we might write:

```
int g (int x,y,z) { return (x+y+z) }      g(42,57,99)
```

In Haskell we could write:

```
g1(x,y,z) = x+y+z                                g1(42,57,99)
```

Usually, however, in Haskell, we write:

```
g2 x y z = x+y+z                                 g2 42 57 99
```

Function Notation (III)

Why does Haskell have this strange function notation?

Reason 1 Because, defining and using functions is so common that the notation should be as lightweight as possible.

Reason 2 With more than one argument, the Haskell notation proves to be surprisingly flexible (and powerful!)

We'll learn about this flexibility and power later.

Function Notation (IV)

As far as Haskell is concerned, `f1 x` and `f2(x)` are the same.

However, `g1(x,y,z)` and `g2 x y z` are not:

- Their types are different:

  ```
  g1 :: Num a => (a,a,a) -> a
  g2 :: Num a => a -> a -> a -> a
  ```

  Function `g1` takes a triple of numbers and returns a number, whereas, `g2` takes a number, another number and another number, and returns a number. (`Num a =>` says that `a` must be a numeric type).

- The implementation of `g2` is faster and uses less memory than that of `g1`.

1.1.6 Patterns

Pattern Matching

- Inputs: a pattern, and a Haskell expression
- Output: either a "fail", or a "success", with a binding from each variable to the part of the expression it matched.

Example:

pattern `('c':zs)`,

matches expression `'c':('a':('t':[]))`,

with `zs` bound to `'a':('t':[])`

Patterns in Haskell

- We can build patterns from atomic values, variables, and certain kinds of "constructions".
- An atomic value, such as `3`, `'a'` or `"abc"` can only match itself
- A variable, or the wildcard `_`, will match anything.
- A "construction" is either:
  1. a tuple such as `(a,b)` or `(a,b,c)`, etc.,
  2. a list built using `[]` or `:`,
  3. or a user-defined datatype.

  A construction pattern matches if all its sub-components match.

Pattern Sequences

- Function definition equations may have a sequence of patterns (e.g., the `and` function example.)
- Each pattern is matched against the corresponding expression, and all such matches must succeed. *One* binding is returned for all of the matches
- Any given variable may only occur once in any pattern sequence (it can be reused in a different equation.).

Pattern Examples

- Expect three arbitrary arguments (of the appropriate type!)

```
myfun x y z = whatever -- using x y z
```

- Illegal — if we want first two arguments to be the same then we need to use a conditional (somehow).

```
myfun x x z = whatever         -- Can’t use x twice here !
myfun x y z | x ==y = whatever -- much better !
```

- First argument must be zero, second is arbitrary, and third is a non-empty list.

```
myfun 0 y (z:zs) = whatever -- using y z zs
```

- First argument must be zero, second is arbitrary, and third is a non-empty list, whose first element is character ‘c’

```
myfun 0 y (’c’:zs) = whatever -- using y zs
```

- First argument must be zero, second is arbitrary, and third is a non-empty list, whose tail is a singleton.

```
myfun 0 y (z:[z’]) = whatever -- using y z z’
```

Pattern Matching (summary)

- Pattern-matching can *succeed* or *fail*.
- If successful, a pattern match returns a (possibly empty) *binding*.
- A binding is a mapping from (pattern) variables to values.
- Examples:

Patterns	Values	Outcome
x (y:ys) 3	99 [] 3	Fail
x (y:ys) 3	99 [1,2,3] 3	Ok, $x \mapsto 99, y \mapsto 1, ys \mapsto [2,3]$
x (3:ys) 3	99 [3,2,1] 3	Ok, $x \mapsto 99, ys \mapsto [2,1]$

- Binding $x \mapsto 99, y \mapsto 1, ys \mapsto [2,3]$ can also be written as a (simultaneous) *substitution* $[99, 1, [2,3]/x, y, ys]$ where 99, 1 and $[2,3]$ replace x, y and ys respectively.

1.1.7 Function Notation (again)

Function Notation (V)

We can define and use functions whose names are either Variable Identifiers ($varid$) or Variable Operators ($varsym$)

For $varid$ names, the function definition uses “prefix” notation, where the function name appears before the arguments:

```
myfun x y  =  x+y+y                      myfun 57 42
```

For $varsym$ names, the function definition uses “infix” notation, where the function has exactly two arguments and the name appears inbetween the arguments:

```
x +++ y  =   x+y+y                       57 +++ 42
```

Function Notation (VI)

For $varid$ names, with functions having two[1] arguments, we can define and use them “infix-style” by surrounding them with backticks:

```
x ‘plus2‘ y  =  x+y+y                     57 ‘plus2‘ 42
```

For $varsym$ names, we can define and use them “prefix-style” by enclosing them in parentheses:

```
(++++) x y  =   x+y+y                     (++++) 57 42
```

We can define one way and use the other—all these are valid:

```
57 ‘myfun‘ 42            (+++) 57 42
plus2 57 42               57 ++++ 42
```

They all have type `Num a => a -> a -> a`.

[1] or more ?!?

1.1.8 Prelude Extracts

The Haskell Prelude [H2010 Sec 9]

- The "`Standard Prelude`" is a library of functions loaded automatically (by default) into any Haskell program.
- Contains most commonly used datatypes and functions
- [H2010 Sec 9] is a specification of the Prelude the actual code is compiler dependent

Prelude extracts (I)

- Infix declarations

```
infixr 9  .
infixr 8  ^, ^^, ..
infixl 7  *, /, ‘quot‘, ‘rem‘, ‘div‘, ‘mod‘
infixl 6  +, -
infixr 5  :, ++
infix  4  ==, /=, <, <=, >=, >
infixr 3  &&
infixr 2  ||
infixl 1  >>, >>=
infixr 1  =<<
infixr 0  $, $!, ‘seq‘
```

Higher precedence numbers bind tighter. Function application binds tightest of all

Prelude extracts (II)

- Numeric Functions

```
subtract  :: (Num a) => a -> a -> a
even, odd :: (Integral a) => a -> Bool
gcd       :: (Integral a) => a -> a -> a
lcm       :: (Integral a) => a -> a -> a
(^)       :: (Num a, Integral b) => a -> b -> a
(^^)      :: (Fractional a, Integral b) => a -> b -> a
```

The Num, Integral and Fractional annotations have to do with *type-classes* — see later.

Prelude extracts (III)

- Boolean Type & Functions

```
data Bool   =  False | True
(&&), (||) :: Bool -> Bool -> Bool
not        :: Bool -> Bool
otherwise  :: Bool
```

Prelude extracts (IV)

- List Functions

```
map    :: (a -> b) -> [a] -> [b]
(++)   :: [a] -> [a] -> [a]
filter :: (a -> Bool) -> [a] -> [a]
concat :: [[a]] -> [a]
head   :: [a] -> a
tail   :: [a] -> [a]
null   :: [a] -> Bool
length :: [a] -> Int
(!!)   :: [a] -> Int -> a
repeat :: a -> [a]
take   :: Int -> [a] -> [a]
drop   :: Int -> [a] -> [a]
elem   :: Eq a => a -> [a] -> Bool
```

Prelude extracts (V)

- Function Functions

```
id        :: a -> a
const     :: a -> b -> a
(.)       :: (b -> c) -> (a -> b) -> a -> c
flip      :: (a -> b -> c) -> b -> a -> c
seq       :: a -> b -> b
($), ($!) :: (a -> b) -> a -> b
```

We will re-visit these later — note that type-polymorphism here means that the possible implementations of some of these are extremely constrained!

1.1.9 Writing Functions (I)

Writing Functions (I) — using other functions

(Examples from Chp 4, Programming in Haskell, 2nd Ed., Graham Hutton 2016)

- Function `even` returns true if its integer argument is even

  ```
  even n = n ‘mod‘ 2 == 0
  ```

  We use the modulo function `mod` from the Prelude
- Function `recip` calculates the reciprocal of its argument

  ```
  recip n = 1/n
  ```

  We use the division function `/` from the Prelude
- Function call `splitAt n xs` returns two lists, the first with the first `n` elements of `xs`, the second with the rest of the elements

  ```
  splitAt n xs = (take n xs, drop n xs)
  ```

  We use the list functions `take` and `drop` from the Prelude

1.2 Examples and HOFs

Higher Order Functions

What is the difference between these two functions?

```
add x y = x + y
add2 (x, y) = x + y
```

We can see it in the types; `add` takes one argument at a time, returning a function that looks for the next argument.

This concept is known as “Currying” after the logician Haskell B. Curry.

```
add :: Integer -> (Integer -> Integer)
add2 :: (Integer, Integer) -> Integer
```

The function type arrow associates to the right, so `a -> a -> a` is the same as `a -> (a -> a)`.

In Haskell functions are *first class citizens*. In other words, they occupy the same status in the language as values: you can pass them as arguments, make them part of data structures, compute them as the result of functions...

```
add3 :: (Integer -> (Integer -> Integer))
add3 = add
```

```
> add3 1 2
3

(add3) 1 2
  ==> add 1 2
  ==> 1 + 2
```

Notice that there are no parameters in the definition of `add3`.

A function with multiple arguments can be viewed as a function of one argument, which computes a new function.

```
add 3 4
  ==> (add 3) 4
  ==> ((+) 3) 4
```

The first place you might encounter this is the notion of *partial application*:

```
increment :: Integer -> Integer
increment = add 1
```

If the type of `add` is `Integer -> Integer -> Integer`, and the type of `add 1 2` is `Integer`, then the type of `add 1` is? It is `Integer -> Integer`

Some more examples of partial application:

An infix operator can be partially applied by taking a *section*:

```
increment = (1 +) -- or (+ 1)

addnewline = (++"\n")

double :: Integer -> Integer
double = (*2)


> [ double x | x <- [1..10] ]
[2,4,6,8,10,12,14,16,18,20]
```

Functions can be taken as parameters as well.

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)

addtwo = twice increment
```

Here we see functions being defined as functions of other functions!

Function Composition (I)

In fact, we can define function composition using this technique:

```
compose :: (b -> c) -> (a -> b) -> a -> c
compose f g x = f (g x)

twice2 f = f ‘compose‘ f
```

We can use an infix operator definition for compose, even though it takes three arguments, rather than two.

```
(f ! g) x = f (g x)
twice3 f = f!f
```

We just bracket the infix application and apply that to the last (`x`) argument.

Function Composition (II) [H2010 Sec 9]

Function composition is in fact part of the Haskell Prelude:

```
(.) :: (b -> c) -> (a -> b) -> a -> c

(f . g) x = f (g x)
```

We can define functions without naming their inputs, using composition (and other HOFs)

```
second :: [a] -> a
second = head . tail
```

```
> second [1,2,3]
2
```

1.2.1 Writing Functions (IIa)

Writing Functions (II) — using recursion

- We shall show how to write the functions `take` and `drop` using recursion.
- We shall consider what this means for the execution efficiency of `splitAt`.
- We then do a direct recursive implementation of `splitAt` and compare.

Implementing `take`

- `take :: Int -> [a] -> [a]` Let `xs1 = take n xs` below. Then `xs1` is the first `n` elements of `xs`. If `n <= 0` then `xs1 = []`. If `n >= length xs` then `xs1 = xs`.

-
  ```
  take n _ | n <= 0  =  []
  take _ []          =  []
  take n (x:xs)      =  x : take (n-1) xs
  ```
- How long does `take n xs` take to run? (we count function calls as a proxy for execution time)
- It takes time proportional to `n` or `length xs`, whichever is shorter.

Implementing drop

- `drop :: Int -> [a] -> [a]` Let `xs2 = drop n xs` below. Then `xs2` is `xs` with the first `n` elements removed. If `n <= 0` then `xs2 = xs`. If `n >= length xs` then `xs2 = []`.

-
  ```
  drop n xs | n <= 0  =  xs
  drop _ []           =  []
  drop n (x:xs)       =  drop (n-1) xs
  ```

- How long does `drop n xs` take to run?

- It takes time proportional to `n` or `length xs`, whichever is shorter.

1.2.2 Writing Functions (IIb)

Implementing splitAt recursively

- `splitAt :: Int -> [a] -> ([a],[a])` Let `(xs1,xs2) = splitAt n xs` below. Then `xs1` is the first `n` elements of `xs`. Then `xs2` is `xs` with the first `n` elements removed. If `n >= length xs` then `(xs1,xs2) = (xs,[])`. If `n <= 0` then `(xs1,xs2) = ([],xs)`.

-
  ```
  splitAt n xs | n <= 0  =  ([],xs)
  splitAt _ []           =  ([],[])
  splitAt n (x:xs)
    = let (xs1,xs2) = splitAt (n-1) xs
      in  (x:xs1,xs2)
  ```

- How long does `splitAt n xs` take to run?

- It takes time proportional to `n` or `length xs`, whichever is shorter, which is twice as fast as the version using `take` and `drop` explicitly!

Switcheroo!

- Can we implement `take` and `drop` in terms of `splitAt`?

- Hint: the Prelude provides the following:

  ```
  fst :: (a,b) -> a
  snd :: (a,b) -> b
  ```

- Solution:

```
take n xs = fst (splitAt n xs)
drop n xs = snd (splitAt n xs)
```

- How does the runtime of these definitions compare to the direct recursive ones?