Defining Functions

Yan Huang
yh33@indiana.edu

The Very Basic Syntax

```
double :: Int → Int

double x = x + x
```

Conditional Expressions

As in most programming languages, functions can be defined using <u>conditional expressions</u>.

```
abs :: Int -> Int
abs x = if x >= 0 then x
        else -x
```

abs takes an integer n and returns n if it is non-negative and -n otherwise.

```
abs 5 => 5
abs -2 => 2
abs 0 => 0
```

Conditional Expressions

As in most programming languages, functions can be defined using <u>conditional expressions</u>.

```
abs   :: Int → Int
abs n = if n ≥ 0 then n else -n
```

abs takes an integer n and returns n if it is non-negative and -n otherwise.

Conditional expressions can be nested:

Define the function signum, which returns -1 when given a negative integer; returns 1 when given a positive integer; and 0 if given 0.

```
signum :: Int -> Int
signum n = if n > 0 then 1
           else if n < 0 then -1
                else 0
```

Conditional expressions can be nested:

```
signum  :: Int → Int
signum n = if n < 0 then -1 else
           if n == 0 then 0 else 1
```

- In Haskell, conditional expressions *must always have an else branch*, which avoids any possible ambiguity problems with nested conditionals.

Guarded Equations

As an alternative to conditionals, functions can also be defined using *guarded equations*.

```
abs :: Int -> Int
abs n | n >= 0    = n
      | n < 0     = -n

      | otherwise = -n
        True
```

Guarded Equations

As an alternative to conditionals, functions can also be defined using *guarded equations*.

```
abs n | n ≥ 0     = n
      | otherwise = -n
```

As previously, but using guarded equations.
The catch all condition otherwise is defined in the "Prelude" by otherwise = True.

Guarded equations can be used to make definitions involving multiple conditions easier to read. E.g.,
Try define signum using guarded equations.

```
signum n | n > 0     = 1
         | n == 0    = 0
         | otherwise = -1
```

Guarded equations can be used to make definitions involving multiple conditions easier to read:

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

Pattern Matching

Many functions have a particularly clear definition using *pattern matching* on their arguments.

```
not       :: Bool → Bool
not False = True
not True  = False
```

not maps False to True, and True to False.

Functions can often be defined in many different ways using pattern matching. For example

```
(&&)          :: Bool → Bool → Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

can be defined more compactly by

```
True && True = True
_    && _    = False
```

The underscore symbol _ is a *wildcard* pattern that matches any argument value.

However, the following definition is more efficient, because it avoids evaluating the second argument if the first argument is False:

```
True  && b = b
False && _ = False
```

- Patterns are matched in order. For example, the following definition always returns False:

```
_    && _    = False
True && True = True
```

- You may not repeat variables in the same pattern. For example, the following definition gives an error:

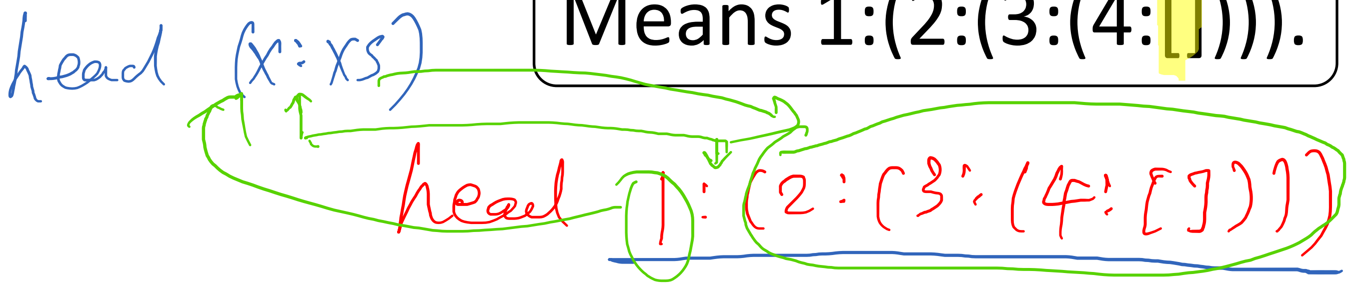
```
b && b = b
_ && _ = False
```

List Patterns

Internally, every non-empty list is constructed by repeated use of an operator (:) called "cons" that adds an element to the start of a list.

[1,2,3,4]

Means 1:(2:(3:(4:[]))).



Functions on lists can be defined using x:xs pattern.

```
head        :: [a] → a
head (x:_)  = x

tail        :: [a] → [a]
tail (_:xs) = xs
```

head and tail map any non-empty list to its first and remaining elements.

coursera

START YOUR FUTURE TODAY

Build these 13 skills to boost your salary

Find Your Specialization

Employers are looking for specific business and technology skills. According to sources like PayScale and Monster.com*, these 13 are among the most likely to make you stand out.

Most valuable **COMPUTER SCIENCE** skills

SKILL #1

Scala[1]

22% average pay boost

Apply functional programming paradigms to write elegant Scala code.

Functional Programming in Scala

EPFL

Join Now

Scala

- Full support for functional programming with a very strong static type system, heavily influenced by Haskell
  - Currying
  - Type inference
  - Immutability
  - Lazy evaluation
  - Pattern matching
  - Algebraic data types
- Compiles to JVM bytecode

Lambda Expressions

Functions can be constructed without naming the functions by using lambda expressions.

$$\lambda x \rightarrow x + x$$

The nameless function that takes a number x and returns the result x + x.

- The symbol λ is the Greek letter lambda, and is typed at the keyboard as a backslash “\”.

λ

- In mathematics, nameless functions are usually denoted using the $\mapsto$ symbol, as in $x \mapsto x + x$.

- In Haskell, the use of the λ symbol for nameless functions comes from the lambda calculus, the theory of functions on which Haskell is based.

\x -> x+x

Why Are Lambda's Useful?

Lambda expressions can be used to give a formal meaning to functions defined using currying.

For example:

```
add x y = x + y
```

means

$$add = \lambda\ x \rightarrow (\lambda\ y \rightarrow x + y)$$

Lambda expressions are also useful when defining functions that return functions as results.

For example:

```
const    :: a → b → a
const x _ = x
```

is more naturally defined by

```
const  :: a → (b → a)
const x = \_ → x
```

const 5 0 -> 5

let f = const 5

f 3 = 5

f 4 = 5

f 'c' = 5

Lambda expressions can be used to avoid naming functions that are only <u>referenced once</u>.

For example:

```
odds n = map f [0..n-1]
        where
           f x = x*2 + 1
```

can be simplified to

```
odds n = map (\x → x*2 + 1) [0..n-1]
```

Sections

An operator written <u>between</u> its two arguments can be converted into a curried function written <u>before</u> its two arguments by using parentheses.

For example:

```
> 1+2
3

> (+) 1 2
3
```

This convention also allows one of the arguments of the operator to be included in the parentheses.

For example:

```
> (1+) 2
3

> (+2) 1
3
```

In general, if ⊕ is an operator then functions of the form (⊕), (x⊕) and (⊕y) are called <u>sections</u>.

Why Are Sections Useful?

Useful functions can sometimes be constructed in a simple way using sections. For example:

- (1+) - successor function
- (1/) - reciprocation function
- (*2) - doubling function
- (/2) - halving function

Exercises

(1) Consider a function safetail that behaves in the same way as tail, except that safetail maps the empty list to the empty list, whereas tail gives an error in this case. Define safetail using:

(a) a conditional expression;
(b) guarded equations;
(c) pattern matching.

Hint: the library function null :: [a] $\rightarrow$ Bool can be used to test if a list is empty.

(2) Give three possible definitions for the logical or operator (||) using pattern matching.

(3) Redefine the following version of (&&) using conditionals rather than patterns:

```
True && True = True
_    && _    = False
```

(4) Do the same for the following version:

```
True  && b = b
False && _ = False
```