Conditionals [*H2010* 3.6]

- For expressions, we can write a conditional using `if ... then ... else`

  exp → `if` *exp* `then` *exp* `else` *exp*

- The else-part is compulsory, and cannot be left out (why not?)
- The (boolean-valued) expression after `if` is evaluated:
  If true, the value is of the expression after `then`
  If false, the value is of the expression after `else`

Prefix vs. Infix

- Functions with identifier names are prefix:
  `myfun x y = 2*x + y`
  - However, 2-argument identifiers can be used infix-style:
    `` 1 `myfun` 2 ``
    (surround with *back*-quotes)
- Functions with symbol names are infix: `x <+> y = 2*x - y`
  - However, can be used prefix-style: `(<+>) 5 7`
    (surround with parentheses)
- Note: there is a difference between "back-quotes" (`) and "single quotation marks" (') noting that they can render differently in different situations.

Operators [*H2010* 3]

- Expressions can built up as expected in many programming languages
  `3     x     x+y     (x<=y)     a + c * d - (e * (a / b))`
- Some operators are left-associative like `+ - * /` :
  `a + b + c` parses as `(a + b) + c`
- Some operators are right-associative like `: . ^ && ||`:
  `a:b:c:[]` parses as `a:(b:(c:[]))`
- Other operators are non-associative like `== /= < <= >= >` :
  `a <= b <= c` is illegal,
  but `(a <= b) && (b <= c)` is ok.
- The minus sign is tricky: `e - f` parses as "`e` subtract `f`",
  `(- f)` parses as " minus `f`",
  but `e (- f)` parses as
  "function `e` applied to argument minus `f`"

Function Application/Types

- Function application is denoted by juxtaposition, and is left associative
- `f x y z` parses as `((f x) y) z`
- If we want `f` applied to both `x`, and the result of the application of `g` to `y`, we must write `f x (g y)`
- In types, the function arrow is right associative
  `Int -> Char -> Bool` parses as `Int -> (Char -> Bool)`
- The type of a function whose first argument is itself a function,
  has to be written as `(a -> b) -> c`
- Note the following types are identical:
  `(a -> b) -> (c -> d)`
  `(a -> b) -> c -> d`

Identifiers as Operators [*H2010* 3.2]

- We can take a variable identifier that denotes a function taking two arguments and turn it into a infix operator by surrounding it with backquotes.
- `mod` is a prefix function that computes the value of its first argument modulo its second
```
> mod 37 5
2
```
- However, adding backquotes allows it to be used in a infix setting
```
> 37 `mod` 5
2
```
- Don't confuse the back-quote used here (`) with the single quote (') used for characters.

Sections [*H2010* 3.5]

- A "section" is an operator, with possibly one argument surrounded by parentheses,
  which can be treated as a prefix function name.
- `(+)` is a prefix function adding its arguments
  (e.g. `(+) 2 3  = 5`)
- `(/)` is a prefix function dividing its arguments
  (e.g. `(/) 2.0 4.0  = 0.5`)
- `(/4.0)` is a prefix function dividing its single argument by 4
  (e.g. `(/4.0) 10.0 = 2.5`)
- `(10.0/)` is a prefix function dividing 10 by its single argument
  (e.g. `(10/) 4.0 = 2.5`)
- `(- e)` is not a section, use `subtract e` instead.
  (e.g. `(subtract 1) 4 =  3`)

Decomposing Problems

- In a very real sense, programming *is* problem decomposition
- We break a big problem down into small problems
- Solve all the small problems
- Connect the solutions to the small problems together into a solution to the big problem

Decomposing Problems

In a lot of languages, you can get away with a certain bad habit

1. Start writing a solution to the big problem
2. Keep programming - when two parts need to share data, make a piece of shared data
3. Keep programming - eventually end up with a solution with lots of sections that depend on the value of a variable shared with other parts

What's wrong with this?

- No way to track how the different parts talk to each other
- No defined interfaces between parts

So when someone tries to modify the code, they need to keep *the entire structure of the application* in their head.
This is frequently referred to as "spaghetti code"

Decomposing Problems

- In Haskell, this is impossible because:
- No mutation - shared variables can't ever change
- But this has consequences
- In Haskell, it's possible to *really* program yourself into a corner and be unable to fix the code
- The keep-going-til-it-works approach is a recipe for pain and frustration

Doing it "right"

1. What do I have?
2. What do I want?
3. How do I get there?
4. Implement the first piece.
5. Go to 1.

Doing it "right" - Haskell

1. What do I have? - this is the initial type `a`
2. What do I want? - this is the final type `b`
3. How do I get there? - this is a function `a -> b`
4. Identify the steps to take to solve the problem - use your own judgement to decide whether these should be separate functions
5. Implement the sub-functions using this same process

If you have to type the same piece of code more than once, it should be a separate function. If the problem is more complex, you may need to have several initial objects at step 1.
Why is this any better?

Doing it "right" - Haskell

- At each step, there is a defined interface that the compiler will enforce - the *type* of the function
- If a function changes, then the program will not compile until you have fixed *every* place where you call it

Decomposition example - `splitAt`

- Wanted, `splitAt :: Int -> [a] -> ([a],[a])` where if `(xs1,xs2) = splitAt n xs` then `xs1` is the first `n` elements of `xs` and `xs2` is `xs` with the first `n` elements removed.
- Idea: compute `xs1` and `xs2` separately
  1. `take :: Int -> [a] -> [a]` where if `xs1 = take n xs` then `xs1` is the first `n` elements of `xs`
  2. `drop :: Int -> [a] -> [a]` where if `xs2 = drop n xs` then `xs2` is `xs` with the first `n` elements removed.
- Assemble the result

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

Is this the best solution? (Discuss)

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

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 >= length xs` then `xs1 = xs`.
  If `n <= 0` then `xs1 = []`.
-
```
take n _ | n <= 0  =  []
take _ []          =  []
take n (x:xs)      =  x : take (n-1) xs
```
- How long does `take n xs` take to run?
- 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 >= length xs` then `xs2 = []`.
  If `n <= 0` then `xs2 = xs`.
- ```
  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.
  ```