Contents

Conditionals

- For expressions, we can write a conditional if `if...then...else`
  - `exp -> if exp then exp else exp`
- The else-part is compulsory and cannot be left out
- The boolean expressed after `if` is evaluated:
  - If `true`, the value is of the expression after `then`
  - If `false`, the value is of the expression after `else`

Local Declerations

- A let-expression has the form
  - $let\{d_1;...d_n\}in\, e$
  - d_i are declarations, e is an expression
- Scope of each d_i is e and the righthand side of all the $d_j s$ (mutual recursion)
- Example: $ax^2 + bx + c = 0$ means $x = \frac{-b \pm (\sqrt{b^2 - 4ac})}{2a}$

```
solve a b c
  = let twoa = 2 * a
        discr = b*b - 2 * twoa * c
```

```
    droot = sqrt discr
  in ((droot-b)/twoa, negate ((droot+b)/twoa))
```

- A where expression has the form
  - $where\{d_1; ...; d_n\}$
  - d_i are declarations
- Scope of each d_i is the expression that precedes `where` and righthand side of all the d_is (mutual recursion)

```
solve a b c
  = ((droot-b)/twoa, negate ((droot+b)/twoa))
  where
    twoa = 2 * a
    discr = b*b - 2 * twoa * c
    droot = sqrt discr
```

Let vs Where

- What is the difference between `let` and `where`?
- The `let...in...` is a full expression and can occur anywhere an expression is expected
- There `where` keyword occurs at certain places in declarations of
  - case expressions
  - modules
  - classes
  - instances
  - function and pattern righthand sides
- Both allow mutual recursion among the declarations

Case Expression

- A case expression has the form
  - $caseeof\{p_1 \rightarrow e_1; ...; p_n \rightarrow e_n\}$
  - p_i are patterns, e_i are expressions

```
odd x =
  case (x `mod` 2) of
    True -> False
    False -> True
```

```
vowel x =
  case x of
    'a' -> True
    'e' -> True
    'i' -> True
    'o' -> True
    'u' -> True
    _   -> False

exmpty x =
  case x of
    [] -> True
    _  -> False
```

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 ``
- Functions with symbol names are infix
  - `x <+> y = 2*x - y`
  - However can be used prefix-style `(<+>) 5 7`

Using Other Functions

- Function `even` returns true if its integer argument is event
  - `` even n = n `mod` 2 == 0 ``
  - We use the modulo function `mod` fromt he Prelude
- Function `recip` calculates the reciprocal of its arguments
  - `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 funcfions `take` and `drop` from the Prelude

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 bigger problem
- 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 where someone tries to modify the code, they need to keep *the entire structure of the application in their head* (spaghetti code)
- In Haskell, this is impossible

    – No mutation - shared variables can't ever change
    – It's possible to *really* program yourself into a corner and be unable to fix the code
    – The keep-going-till-it-works approach is a recipe for pain and frustration

Doing it 'right'

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. Implement the first piece
5. Go to 1

- 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