Thought for the day

"A language that doesn't affect the way you think about programming, is not worth knowing."

— Alan J. Perlis, Epigrams on Programming, 1982

Overview

Last class

- Examples of simple Haskell programs.
- Infix and prefix functions
- How Haskell works

Today

- Basic types and classes

Syntax

- comments are either
  `--` comment to end of line or `{-` comment `-}`
- variables either:
  - prefix: made up of letters, digits, ' or _ and start with a lower-case letter
  - infix: made up of sequences of other characters
- indentation is significant
- parentheses are used for precedence and tuples
- Function application binds most strongly
  `factorial 3*5` means
  `(factorial 3)*5`
- Binary prefix functions can be made infix using back-quotes, e.g. `` `div` ``
  Infix operators can be made prefix using parentheses, e.g. (*)

Definition of a function

- Function Definition:

  `name x1 x2 ... xk = e`

  `x1 x2 ... xk` are formal parameters
  e is an expression
- Multiple equations can define a function; the first one to succeed is used.

Evaluation of Haskell program

- Haskell evaluates expressions.
- Haskell knows how to implement some expressions (such as `3+4*7`)
- Given the defintion of name:

  ```
  name x1 x2 ... xk = e
  ```

  The expression

  ```
  name v1 v2 ... vk
  ```

  when all `k` arguments are provided, evaluates to value of `e` but with each `xi` replaced with `vi`

  ```
  foo x y = 1000*x+y
  foo 9 3
  bar = foo 7
  bar 3
  ```

Type Declarations

For the defintion of name:

`name x1 x2 ... xk = e`

- Type declaration:

  `name :: t1 -> t2 -> ... -> tk -> t`

  ti is type of *xi*, and *t* is the type of *e*.
- Each function takes only one argument:
  `name v1` is a function of type `t2 -> ... -> tk -> t`
  `name v1 v2 ... vk` is a value of type `t`
  It's value is the value of `e` with each `xi` replaced by `vi`

Today

- Haskell Types:
  - Bool (&&, ||, not)
  - Num (+, −, ∗, abs)
    - Integral (div, mod)
      - Int
      - Integer
    - Fractional (/)
      - Floating (log, sin, exp, ...)
        - Double
  - Eq (==, /=)
    - Ord (>, >=, <=, <)
  - Char
  - String

Type: Bool

- `Bool` is a type with two values `True` and `False`.
- operations:

`&&`	and
`||`	or
`not`	not

- How can we define exclusive-or (`xor`)?
- How can we define if-then-else?
- What would happen if we tried to do this in Java?
  (Answer: because Java evaluates a method's arguments before calling the method, a method implementation of if-then-else would not halt for recursive methods.)

Integral types

- Intergral types represent integers.
- They implement `+ * ^ - div mod abs negate`
- Two implementations:
  - Int - fixed-precision integers
  - Integer - arbitrary precision integers
- Integral is a class.
  Int and Integer are types in class Integral.
  Only types have implementations.
  (Haskell classes are like Java interfaces)
- `div :: Integral a => a -> a -> a`
  `div` takes two arguments of the same type, and returns a value of that type.
  That type must be in the `Integral` class.

Fractional types

- Fractional types represent real numbers.
- They implement `+ * ^ - / abs negate`
- Floating types also implement `log sin exp` ...
- Multiple implementations:
  - Double - double precision floating-point numbers (64 bit)
  - Float - single precision floating-point numbers (32 bit) — don't use
  - Rational - exact rational numbers
- There are no types that are both Integral and Fractional.
- Num types implement `+ * ^ - abs negate`
  Num is a class (elements are types).
  Integral and Fractional are subclasses of Num

Eq and Ord classes

- Eq types implement `==` `/=`
- Ord types implement `>` `>=` `<=` `<` `max` `min`
- Int, Integer, Double implement Eq and Ord
- Can you think of a Num type that isn't an Ord type?
  How about Complex?
- What is the type of 3?
- What is the type of `div 100 3`?
  What is the type of 3.7?
  What is the type of `(div 100 3) + 3.7`?
- `fromIntegral` converts an integer to a Num.

Guards

- Guards are used for if-then-else structure in definition of functions.
- Example

```
mymax x y
    | x>y = x
    | otherwise = y
```

It evaluates the guards; the first one succeeding, the corresponding expression is returned

Guards

- General case:

```
name x1 x2 ... xk
    | g1 = e2
    | g2 = e2
     ...
    | gn = en
```

- evaluate g_1, g_2 in turn until the first one g_i evaluates to true, then return value of e_i.
- An Exception is raised if none of the guards are True
- Typical to have last condition to be `otherwise` which is a variable with value `True`.
- How can we implement max3?
- Haskell also has "if ... then ... else ..." structure