

Conditionals and Tuples

Reading: Thompson Ch.3

COMP1100 — Introduction to Programming and Algorithms

Clem Baker-Finch

Australian National University

Semester 1, 2006

Example evaluation:

```
max 5 3 ⇒ if 5 >= 3 then 5 else 3 ⇒  
          if True then 5 else 3 ⇒ 5
```

Sometimes the choice is not so simple:

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

Nested conditionals can be hard to read, but layout can help:

```
signum x = if x < 0 then -1  
           else if x == 0 then 0  
           else 1
```

Conditional Expressions

So far, all our example scripts have unconditionally performed the same computation.

How would we implement a function: `max :: Int -> Int -> Int`

which returns the greater of its two arguments?

We want `max x y` to return `x` if `x >= y`, otherwise it should return `y`.

Haskell has *conditional expressions* to allow us to make such a choice:

```
if <condition> then <value if true> else <value if false>
```

so we can write:

```
max x y = if x >= y then x else y
```

Guards

Some languages (Haskell included) have *guarded expressions* as an alternative notation for conditionals:

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

GHC evaluates each guard *in turn, first to last* until it finds one that equals `True`. The right hand side that corresponds to that guard is chosen.

If none of the guards are true, GHC will report an error (at run time).

Haskell provides a special guard `otherwise` that is always true.

(Experiment: evaluate `otherwise` in GHCi.)

`otherwise` is used as a catch-all at the *end* of a sequence of alternatives:

```
min x y
  | x <= y      = x
  | otherwise   = y
```

Another example: How many premiership points does a team get, given the score at the end of the match?

```
points :: Int -> Int -> Int
points for against
  | for > against    = 2
  | for == against   = 1
  | otherwise        = 0
```

Tuple Patterns

Functions on tuples are usually defined using **pattern matching**.

For example, here is a function to add a *pair* of Ints:

```
addPair :: (Int, Int) -> Int
addPair (x,y) = x + y
```

The pattern `(x,y)` matches any pair **and sets** `x` **to be the first element of the pair** and `y` **to be the second element**.

Tuples — Combining Different Data

to represent real-world data, we often want to *combine* types. For example, an item in a supermarket may need a bar code, a name and a price.

Haskell lets us combine any n types into an ordered n -tuple.

```
(723476, "Peanut Butter", 375)
```

has type:

```
(Int, String, Int)
```

Notice that the *type* is written in a way that corresponds to the way we write the *expressions*.

For example, representing cartesian coordinates:

```
type Point = (Int, Int)
```

```
-- The origin of the coordinate system
origin :: Point
origin = (0,0)
```

```
-- Move a point a distance to the right
moveRight :: Point -> Int -> Point
moveRight (x,y) distance = (x+distance,y)
```

```
-- Move a point upwards
moveUp :: Point -> Int -> Point
moveUp (x,y) distance = (x,y+distance)
```

Definitions with where clauses

It is often possible to simplify an expression by extracting some part and naming it. This can help in two ways: putting a name to a sub-expression can make it easier to understand; a repeated sub-expression may only be evaluated once.

Suppose we wanted to compute the real roots of a quadratic:

$$ax^2 + bx + c = 0$$

The standard formula is

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Layout of function definitions

In any programming language, the layout of your program is important for the readability of your programs.

In Haskell, *layout rules* help to get rid of the annoying punctuation used in many other languages (semicolons, braces, etc.).

Haskell uses **indentation** to decide the ends of definitions, expressions and so on.

Once you get into good habits, it will be very natural.

When you are *learning*, you might need to be a bit careful.

(Emacs Haskell mode helps with indentation. Hit the TAB key a few times...)

The formula $b^2 - 4ac$ is the *discriminant*.

The discriminant must be ≥ 0 for the roots to be real.

```
roots :: Float -> Float -> Float -> (Float, Float)
roots a b c
  | discrimin >= 0 = ((-b + (sqrt discrimin))/(2*a),
                    (-b - (sqrt discrimin))/(2*a))
  | otherwise     = error "No real roots"
  where discrimin = b^2 - 4*a*c
```

(This isn't a particularly satisfactory design — a quadratic may have 0, 1 or 2 real root. We may revisit this example later.)

The Off-Side Rule

A definition *ends* when a (non-space) symbol appears in the same column as the first symbol of the definition.

See textbook pages 47 & 48.

Recommended layout

Something like:

```
fun p_1 p_2 .. p_n
  | guard_1 = e_1
  | guard_2 = e_2
  . . .
  | guard_k = e_k
  where
  local_1 a_1 .. a_m = r_1
  local_2           = r_2
  . . .
```