

Conditionals and Tuples

Reading: Thompson Ch.3

COMP1100 — Introduction to Programming and Algorithms

Ben Lippmeier

Australian National University

Semester 2, 2006

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?

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
```

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
```

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 useful expression `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
```

Tuples — Combining data components

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*.

Extracting the components

The functions `fst` and `snd` defined in the `Prelude` allow us to extract the components of a 2-element tuple.

For example:

```
addPair :: (Int, Int) -> Int
addPair t = fst t + snd t
```

```
addPair (2, 3)
⇒ fst (2, 3) + snd (2, 3)
⇒ 2 + 3
⇒ 5
```

Tuple patterns

Haskell also provides a mechanism called *pattern matching* which allows us to write functions on tuples (and other data types) in a more natural form.

We could rewrite the previous function as:

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

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

as-patterns

Sometimes we want to refer to the *whole* tuple as well as bind its components with a pattern. We can do this with an *as* pattern.

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

In this example we used an *as*-pattern to bind the whole tuple to the name *t*, as well as giving the components their own names *x* and *y*.

Another example, representing cartesian coordinates:

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

```
moveRight :: Point -> Int -> Point
```

```
moveRight (x, y) dx = (x + dx, y)
```

```
moveUp :: Point -> Int -> Point
```

```
moveUp (x, y) dy = (x, y + dy)
```

```
movePoint :: Point -> Int -> Int -> Point
```

```
movePoint point dx dy
```

```
    = moveUp (moveRight point dx) dy
```

Nested Tuples

Tuples and other data types may be nested.

The following expression is a tuple which contains more tuples.

```
(( 'a' , 23 ) , ( "Perch" , ( True , False ) ) )
```

It has type:

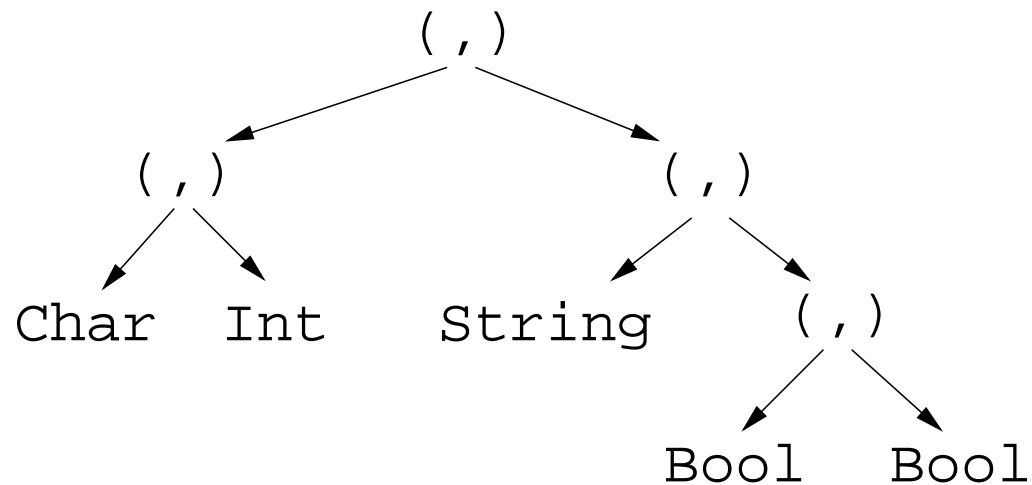
```
(( Char , Int ) , ( String , ( Bool , Bool ) ) )
```

Pattern matching also works on nested types.

```
g1B :: (( Char , Int ) , ( String , ( Bool , Bool ) ) ) -> Bool
g1B (( c1 , i ) , ( str , ( b1 , b2 ) ) ) = b1
```

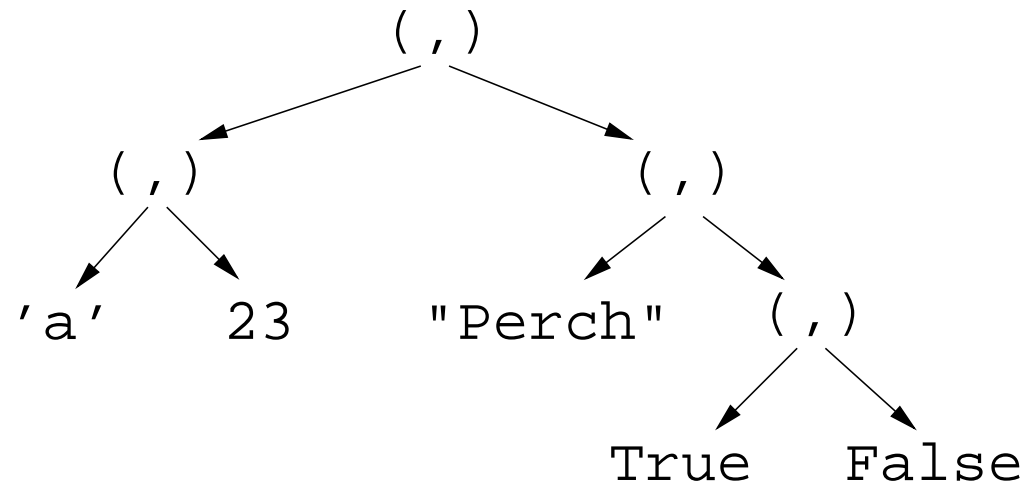
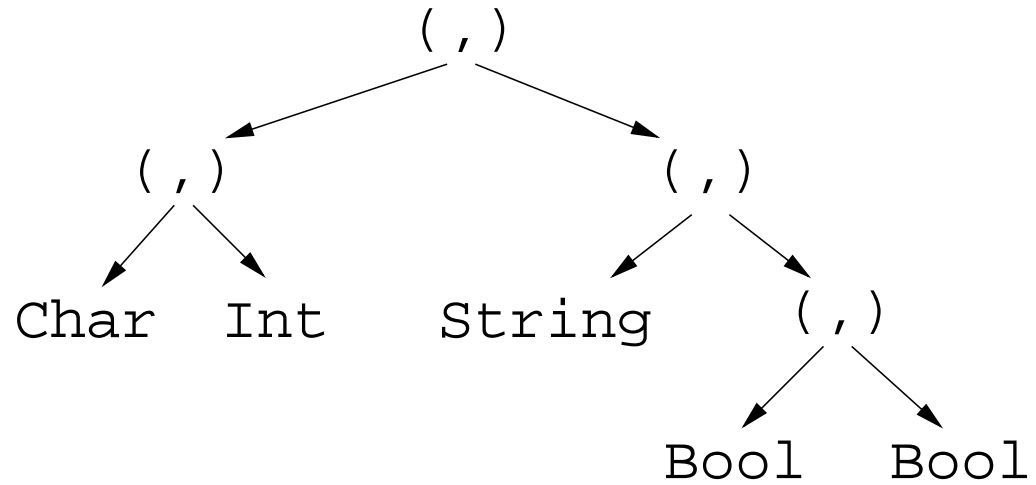
Sometimes it is helpful to draw a nested type as a tree so we can see its structure.

`((Char, Int), (String, (Bool, Bool)))`



For structures with more than a few components, we would usually use Abstract Data Types (ADTs) instead of tuples. We will cover ADTs in week 4.

Notice how the structure of the type reflects the structure of the data.



Definitions with `where` clauses

It is often possible to simplify an expression by extracting some part and naming it.

- giving a name to a sub-expression can make it easier to understand.
- if a sub-expression has a descriptive name then it is less likely to need a comment explaining what it does.
- a repeated sub-expression only needs to be evaluated once.

An example:

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

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

The standard formula is

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

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
  | discriminant >= 0
  = ((-b + sqrt discriminant) / (2*a),
      (-b - sqrt discriminant) / (2*a))
  | otherwise
  = error "No real roots"
  where discriminant = 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.)

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.

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.

If in doubt, follow the layout from one of the examples in the textbook or lecture notes.

Things to watch for.

- all the | symbols in an expression with guards *must line up*.
- in a guard, the = must be in the same column as the |, or in any column to the right of it.
- if a where expression has multiple bindings, then the first characters in each of the bound names must line up.