1 Haskell

1.1 Tuples

We now begin by discussing tuples.

1.1.1 Pairs

Recall that, in Lambda Calculus, we created our own pairs that could be used to store different things. In Haskell, we could use the Lambda Calculus encoding of pairs, *or* we can use the built-in pair. For example:

```
myPair :: (String, Int)     -- Pair of String and Int
myPair = ("apple", 3)
```

Here, `(,)` is the pair constructor.

1.1.2 Accessing Fields

How do we access the entries of a pair? One way is by using library functions:

```
whichFruit = fst myPair         -- "apple"
howMany    = snd myPair         -- 3
```

However, you don't need to use `fst` or `snd`. To see what is meant, consider the following definition:

```
isEmpty :: (String, Int) -> Bool
isEmpty p = (fst p == "") || (snd p == 0)
```

This syntax looks a bit verbose. However, we can use *pattern matching* to make it look better.

```
isEmpty (s, n) = s == "" || n == 0
```

Note that this is the same thing as writing:

```
isEmpty = \(s, n) -> s == "" || n == 0
```

Or, we can do:

```
isEmpty p = let (s, n) = p in s == "" || n == 0
```

Let's suppose now that you want to have both the pair and the components; then, we can either use the example above or we can do

```
isEmpty p@(s, n) = s == "" || n == 0
```

Note that, with equations and patterns, we can have multiple equations with multiple patterns. However, with `let`-patterns and lambda patterns, you can only have one. In particular, we can do this with equations:

```
isEmpty ("", _)     = True
isEmpty (_, 0)      = True
isEmpty _           = False
```

(Quiz.) Suppose you have the following function definition and implementation:

```
f :: String -> [(String, Int)] -> Int
f _ []      = 0
f x ((k, v) : ps)
    | x == k        = v
    | otherwise     = f x ps
```

What happens if the function is called with input `f "hi" [("hi", 5), ("apple", 10)]`?

(a) Syntax error.

(b) Type error.

(c) First pattern matches.

(d) Second pattern matches and binds

```
x -> "hi", k -> ("hi", 5), v -> ("apple", 10)
```

(e) Second pattern matches and binds

```
x -> "hi", k -> "hi", v -> 5, ps -> [("apple", 10)]
```

The answer is **E**. Since the list is non-empty, we know that the second equation must be executed.

- Now, it should be obvious that `"hi"` binds to `x`.
- We also note that the `((k, v) : ps)` means that we're taking the head element (a tuple) and destructuring it into `k` and `v`; additionally, `ps` is the tail list (i.e. the list without the head). Therefore, `"hi"` binds to `k` and `5` binds to `v`.
- Finally, since `((k, v) : ps)` breaks down the head element into its components and `ps` is the tail list (the list without the head element), it follows that `ps` must just be `[("apple", 10)]`.

1.1.3 Triples & More

Of course, we can implement triples like in λ-calculus. However, Haskell has support for triples as well. In fact, Haskell has native support for n-tuples.

```
-- Pair
myPair  :: (String, Int)
myPair = ("apple", 3)

-- Triple
myTriple :: (Bool, Int, [Int])
myTriple = (True, 1, [1, 2, 3])

-- 4-Tuple
my4Tuple :: (Float, Float, Float, Float)
my4Tuple = (pi, sin pi, cos pi, sqrt 2)

...
```

One thing to note is that a one-tuple doesn't exist. While we could "define" one, it'll just be the same as the type itself. That is, `(a)` is the same thing as `a` for a type `a`.

A tuple with 0 elements, however, does make sense. A zero-tuple is known as an *unit*[1]. It is defined like so:

```
myUnit :: ()
myUnit = ()
```

(Quiz.) Assume that

```
(+) :: Int -> Int -> Int.
```

[1]Because it has no value.

Which of the following terms is *ill-typed*?

(a) `(\(x, y, z) -> x + y + z) (1, 2, True)`

(b) `(\(x, y, z) -> x + y + z) (1, 2, 3, 4)`

(c) `(\(x, y, z) -> x + y + z) [1, 2, 3]`

(d) `(\x y z -> x + y + z) (1, 2, 3)`

(e) All of the above.

The answer is **E**. To see why this is the case, we note that:

- A is ill-typed because we cannot add an `Int` to a `Bool`. Note that this function takes in a triple of `Ints` because of the addition being performed on each component.
- B is ill-typed because we're trying to pass a four-tuple as an argument to a function that accepts a triple.
- C is ill-typed because we're trying to pass a list of three elements as an argument to a function that accepts a triple. A list is *not* a tuple.
- D is ill-typed because we're trying to pass a triple as an argument to a function that accepts three arguments.

1.2 List Comprehension

List comprehension is a conveninent way to construct lists from other lists. To get an idea of what we mean, suppose we want to convert a string s to its uppercase form (e.g. `abc` becomes `ABC`). Using list comprehension, we can do:

```
comp1 s = [toUpper c | c <- s]
```

Here, note that `toUpper` is a library function on characters that converts a character to uppercase. So, this is basically saying: call `toUpper` on `c` for each character `c` in the string `s`. In Python, we can write:

```
[c.upper() for c in "hello"]
```

We say that the part on the right (i.e. to the right of the pipe) is called the **generator**. We can also have multiple generators! For example, consider the following expression (which uses multiple generators where one generator depends on the other):

```
comp2 = [(i, j) | i <- [1..3],
                  j <- [1..i]]
```

Essentially, this is saying that we're creating (ordered) pairs (i, j) such that $j \leq i$ for all $i \in \{1, 2, 3\}$. Mathematically, this means

$$\texttt{comp2} = \{(i, j) \mid i \in \{1, 2, 3\}, j \in \{1, \ldots, i\}\}.$$

Thus, `comp2` will contain the list:

```
[(1, 1), (2, 1), (2, 2), (3, 1), (3, 2), (3, 3)]
```

Another example is:

```
comp3 = [(i, j) | i <- [0..5],
                  j <- [0..5],
                  i + j == 5]
```

Here, `comp3` is the list of all (ordered) pairs (i, j) where $i + j = 5$ and $i, j \in \{0, 1, 2, 3, 4, 5\}$. Mathematically, this means

$$\texttt{comp3} = \{(i, j) \mid i, j \in \{0, \ldots, 5\}, i + j = 5\}.$$

So, `comp3` would contain the list:

```
[(0, 5), (1, 4), (2, 3), (3, 2), (4, 1), (5, 0)]
```

1.2.1 Quicksort

A quicksort implementation is as follows:

```
sort :: [Int] -> [Int]
-- Base Case
sort []         = []
-- Otherwise, let 'x' be the pivot, since we conveninently have it.
-- Then, we want to sort the list with respect to the pivot.
-- In particular, we have a list 'ls' which contains all elements
-- smaller than or equal to the pivot, and a list 'rs' which contains
-- all elements bigger than the pivot.
sort (x : xs)   = sort ls ++ [x] ++ sort rs
    where
        -- We use list comprehension to create the sublists based
        -- on the pivot values (as described above).
        ls      = [l | l <- xs, l <= x]
        rs      = [r | r <- xs, x < r]
```