Data Model 02

Thumrongsak Kosiyatrakul
tkosiyat@pitt.edu

Reverse

- The reverse of the list `[1,2,3,4]` is `[4,3,2,1]`
- The list `[4,3,2,1]` is `[4,3,2] +++ [1]`
  - `+++` is our concatenation operator
  - Without our `+++` operator, simply use `++` operator
  - The list `[4,3,2]` is the reverse of the list `[2,3,4]`
- Thus, we have
  - `reverse2 [] = []`
  - `reverse2 (x:xs) = (reverse2 xs) +++ [x]`

Reverse

- An example of concatenation:

```
reverse2 :: [a] -> [a]
reverse2 lst = if null lst
               then []
               else reverse2 (tail lst) +++ [head lst]
```

- Load it into GHCi and do some tests

```
ghci> :t reverse2
reverse2 :: [a] -> [a]
ghci> reverse2 []
[]
ghci> reverse2 [1,2,3,4]
[4,3,2,1]
ghci> reverse2 "Hello"
"olleH"
```

Tuple

- A `tuple` is a type with a fix number of components
  - Components in a tuple can have different types
  - Values are written between parentheses and separated by commas

```
ghci> (1, True)
(1,True)
ghci> ([2,5], "Hello", 3, False)
([2,5],"Hello",3,False)
ghci> :t ("Hello", True, if 2 > 3 then 'G' else 'L')
("Hello", True, if 2 > 3 then 'G' else 'L') :: ([Char], Bool, Char)
```

- Note that the expression `if 2 > 3 then 'G' else 'L'` has type `Char`
- A tuple can be used for a function to return more than one value

Tuple

- A pair is a tuple with exactly two components
- A pair is just a type of tuple but it is not a type in Haskell

```
ghci> :t (True, "Haskell")
(True, "Haskell") :: (Bool, [Char])
```

- Common functions for pairs are provided:
  - `fst` which returns the first component
  - `snd` which returns the second component

```
ghci> fst (True, "Haskell")
True
ghci> snd ("Haskell", [False, True, False])
[False,True,False]
ghci> fst (True, 4, False)

<interactive>:9:5: error:
```

- `fst` and `snd` are for two-element tuples only

```
ghci> :t fst
fst :: (a, b) -> a
ghci> :t snd
snd :: (a, b) -> b
```

The `maxmin` function

- Suppose we want to write a function named `maxmin` that returns the maximum and the minimum values on a given list
- This function needs to return two values
- We can return a pair `(max, min)` where
  - `max` is the maximum value on the list
  - `min` is the minimum value on the list
- Obviously `maxmin` or an empty list is undefined (ignore for now)
- If the given list contains exactly one number, that number is the maximum as well as the minimum
- How to check that a list contains exactly one element?
- If the tail of a list is empty, the list contains exactly one element

```
ghci> null (tail [1,2,3])
False
ghci> null (tail [5])
True
```

The `maxmin` function

- This is what we have so far (incomplete)

```
maxmin lst = if null (tail lst)
             then (head lst, head lst)
             else (...,...)
```

- How to recursively find the maximum value in a given list?
  - Maximum value of `[]` is undefined
  - Maximum value of `[x]` is `x`
  - Maximum value of `x:xs` is???
    - `x` if `x` is greater than the maximum value of the list `xs`
    - the maximum value of the list `xs`, otherwise
- How to recursively find the minimum value in a given list?
  - Minimum value of `[]` is undefined
  - Minimum value of `[x]` is `x`
  - Minimum value of `x:xs` is???
    - `x` if `x` is less than the minimum value of the list `xs`
    - the minimum value of the list `xs`, otherwise

The `maxmin` function

- Now, we have

```
maxmin lst = if null (tail lst)
             then (head lst, head lst)
             else ( if head lst > fst (maxmin (tail lst))
                    then head lst
                    else fst (maxmin (tail lst))
                  ,
                    if head lst < snd (maxmin (tail lst))
                    then head lst
                    else snd (maxmin (tail lst))
                  )
```

- Do not forget to run some tests:

```
ghci> maxmin [5]
(5,5)
ghci> maxmin [3,5,4,1,2]
(5,1)
```

The `maxmin` function

- Again, same function:

```
maxmin lst = if null (tail lst)
             then (head lst, head lst)
             else ( if head lst > fst (maxmin (tail lst))
                    then head lst
                    else fst (maxmin (tail lst))
                  ,
                    if head lst < snd (maxmin (tail lst))
                    then head lst
                    else snd (maxmin (tail lst))
                  )
```

- The above expression is not easy to read
- It also contains
  - the expression `head lst` six times, and
  - the expression `maxmin (tail list)` four times
- Easy to contain errors

Local Bindings

- A **local binding** gives a name to an expression to be used in another expression
- The `let` binding introduces bindings **before** the main expression and must end with the `in` keyword
- Example:

```
maxmin2 lst = let hd = head lst
                  temp_maxmin = maxmin2 (tail lst)
                  temp_max = fst temp_maxmin
                  temp_min = snd temp_maxmin
              in if null (tail lst)
                 then (hd, hd)
                 else (if hd > temp_max then hd else temp_max,
                       if hd < temp_min then hd else temp_min)
```

- Variables `hd`, `temp_maxmin`, `temp_max`, and `temp_min` are only available to the expression after the keyword `in`

Local Bindings

- The `where` introduces bindings **after** the main expression
- Example:

```
maxmin3 lst = if null (tail lst)
              then (hd, hd)
              else (if hd > temp_max then hd else temp_max,
                    if hd < temp_min then hd else temp_min)
              where hd = head lst
                    temp_maxmin = maxmin3 (tail lst)
                    temp_max = fst temp_maxmin
                    temp_min = snd temp_maxmin
```

- It depends on your state of mind whether to use the `let` or the `where` binding
- Some prefer to know all sub-expressions before the main expression
- Some prefer to know the main expression before all sub-expressions
- Focus on readability to reduce errors

Local Bindings

- The `let` and `where` can be used together:
- Example:

```
maxmin4 lst = let hd = head lst
              in if null (tail lst)
                 then (hd, hd)
                 else (if hd > temp_max then hd else temp_max,
                       if hd < temp_min then hd else temp_min)
                 where temp_maxmin = maxmin4 (tail lst)
                       temp_max = fst temp_maxmin
                       temp_min = snd temp_maxmin
```

- Note that bindings in `let` are not available in the scope of `where`
- The following will cause an error because of `temp_maxmin`:

```
maxmin4 lst = let hd = head lst
                  temp_maxmin = maxmin4 (tail lst)
              in if null (tail lst)
                 then (hd, hd)
                 else (if hd > temp_max then hd else temp_max,
                       if hd < temp_min then hd else temp_min)
                 where temp_max = fst temp_maxmin
                       temp_min = snd temp_maxmin
```

- Mixing `let` and `where` may make an expression harder to read

Indentation!!!

- Python is an example of an indentation-sensitive language
- Haskell is a **layout-based** language
- It relies on indentation to reduce the verbosity of the code
- All **grouped expressions** must be exactly aligned
  - Recall the `let` and `where` bindings
  - They are followed by a group of expressions
- Consider the following expression:

```
foo str = let p = null str
              t = tail str
          in if p then "empty" else t
```

- `p = ...` and `t = ...` are in the same group

Indentation!!!

- The following examples result in errors:

```
foo str = let p = null str
              t = tail str
          in if p then "empty" else t
```

```
foo str = let p = null str
             t = tail str
          in if p then "empty" else t
```

- The `where` binding uses the same rule
- The following examples result in errors:

```
bar str = if p then "empty" else t
          where p = null str
                t = tail str
```

```
bar str = if p then "empty" else t
          where p = null str
              t = tail str
```

Indentation!!!

- Some Haskellers also tend to align other symbols (e.g., =)

```
maxmin4 lst = let hd = head lst
              in if null (tail lst)
                 then (hd, hd)
                 else (if hd > temp_max then hd else temp_max,
                       if hd < temp_min then hd else temp_min)
                 where temp_maxmin = maxmin4 (tail lst)
                       temp_max    = fst temp_maxmin
                       temp_min    = snd temp_maxmin
```

- **The goal is to make your code easier to read**
- **Note**: The order in a binding does not matter:

```
maxmin4 lst = let hd = head lst
              in if null (tail lst)
                 then (hd, hd)
                 else (if hd > temp_max then hd else temp_max,
                       if hd < temp_min then hd else temp_min)
                 where temp_max    = fst temp_maxmin
                       temp_min    = snd temp_maxmin
                       temp_maxmin = maxmin4 (tail lst)
```