Lecture 05: Useful Haskell Syntax, HO Programming Continued

- Goodbye to Bare Bones Haskell: Built-in syntax for lists & tuples
- Lambda expressions and Beta-Reduction
- Let and Case Expressions

Reading: Hutton Ch. 4 & 7

You should be starting to look through the Standard Prelude in Appendix B, particularly the list processing functions!
Useful Haskell Syntax: Built-In Types

We have used Bare Bones Haskell notation for Lists, Pairs, and Triples in order to emphasize the importance of pattern-matching in defining functions. However, enough is enough! Here is a more convenient syntax which is built into the basic Haskell syntax (and not just implemented as functions in the Prelude):

**BB Haskell**

```haskell
data Bool = False | True

(&&) :: Bool -> Bool -> Bool
False && _ = False
True && b = b
```

**Flesh and Blood Haskell**

```haskell
data Nat = Zero | Succ Nat

add :: Nat -> Nat -> Nat
add Zero x = x
add (Succ x) y = Succ (add x y)
```
Useful Haskell Syntax: Built-In Types

We have used Bare Bones Haskell notation for Lists, Pairs, and Triples in order to emphasize the importance of pattern-matching in defining functions. However, enough is enough! Here is a more convenient syntax which is built into the basic Haskell syntax (and not just implemented as functions in the Prelude):

**BB Haskell**

```haskell
data Bool = False | True
(&&) :: Bool -> Bool -> Bool
False && _ = False
True && b = b
```

**Flesh and Blood Haskell**

Built in to the Prelude exactly as we presented it:

```
Bool, True, False, &&, ||, not
```

Built in types Integer, Double, .....
Useful Haskell Syntax: Built-In Tuples

**BB Haskell**

```haskell
data Pair a b = P a b
data Triple a b c = T a b c

fst :: Pair a b -> a
fst (P x _) = x

snd :: Pair a b -> b
snd (P _ x) = x

toLeft :: (Pair a (Pair b c))
        -> (Pair (Pair a b) c)
toLeft (P x (P y z)) = (P (P x y) z)

p2T :: (Pair a (Pair b c))
     -> (Triple a b c)
p2T (P x (P y z)) = (T x y z)
```

```haskell
Main> P 3 True
P 3 True

Main> (P 4 (P True (-9)))
P 4 (P True (-9))

Main> (T 3 5 9)
T 3 5 9

Main> (T 9 False 2)
T 9 False 2

Main> fst (P 3 True)
3

Main> snd (P 3 (P True 2))
P True 2

Main> toLeft (P 4 (P True (-9)))
P (P 4 True) (-9)

Main> p2T (P 4 (P True (-9)))
T 4 True (-9)
```
# Useful Haskell Syntax: Built-In Tuples

## BB Haskell

<table>
<thead>
<tr>
<th>Command</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main&gt; P 3 True</td>
<td>P 3 True</td>
</tr>
<tr>
<td>Main&gt; (P 4 (P True (-9)))</td>
<td>P 4 (P True (-9))</td>
</tr>
<tr>
<td>Main&gt; (T 3 5 9)</td>
<td>T 3 5 9</td>
</tr>
<tr>
<td>Main&gt; (T 9 False 2)</td>
<td>T 9 False 2</td>
</tr>
</tbody>
</table>

```haskell
fst :: (a,b) -> a
fst (x,_ ) = x

snd :: (a,b) -> b
snd (_,x) = x

toLeft :: (a,(b,c)) -> ((a,b),c)
toLeft (x,(y,z)) = ((x,y),z)

p2T :: (a,(b,c)) -> (a,b,c)
p2T (x,(y,z)) = (x,y,z)
```

## Flesh and Blood Haskell

<table>
<thead>
<tr>
<th>Command</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main&gt; (3,True)</td>
<td>(3,True)</td>
</tr>
<tr>
<td>Main&gt; (4,(True,(-9)))</td>
<td>4 (True,(-9))</td>
</tr>
<tr>
<td>Main&gt; (3,5,9)</td>
<td>(3,5,9)</td>
</tr>
<tr>
<td>Main&gt; (9,False,2)</td>
<td>(9,False,2)</td>
</tr>
<tr>
<td>Main&gt; fst (3,True)</td>
<td>3</td>
</tr>
<tr>
<td>Main&gt; snd (3,(True,2))</td>
<td>(True,2)</td>
</tr>
<tr>
<td>Main&gt; toLeft (4,(True,(-9)))</td>
<td>((4,True),-9)</td>
</tr>
<tr>
<td>Main&gt; p2T (4,(True,(-9)))</td>
<td>(4,True,-9)</td>
</tr>
<tr>
<td>Main&gt; (2,3,True,5,'a',7,4,&quot;hi&quot;,5)</td>
<td>(2,3,True,5,'a',7,4,&quot;hi&quot;,5)</td>
</tr>
</tbody>
</table>

Tuples can be any length, but `fst` and `snd` only work on pairs. Provided in Prelude
Useful Haskell Syntax: Built-In Lists

**BB Haskell**

```
data List a = Nil  
          | Cons a (List a)

head :: List a -> a
head (Cons x _) = x

tail :: List a -> List a
tail (Cons _ xs) = xs

length :: List a -> Integer
length Nil = 0
length (Cons _ xs) = 1 + (length xs)
```

**Flesh and Bones Haskell**

```
Built in as part of syntax!

head :: [] a -> a
head (x:_:[]) = x

tail :: [a] -> [a]
tail (_::xs) = xs

length :: [a] -> Integer
length [] = 0
length (_::xs) = 1 + (length xs)
```

```
Main> (Cons 3 (Cons 9 Nil))
Cons 3 (Cons 9 Nil)

Main> head (Cons 3 (Cons 9 Nil))
3

Main> tail (Cons 3 (Cons 9 Nil))
Cons 9 Nil

Main> length (Cons 3 (Cons 9 Nil))
2
```

```
Main> []
[]

Main> head [3,9]
3

Main> tail [3,9]
[9]

Main> length [3,9]
2
```

Provided in Prelude
Useful Haskell Syntax: Built-In Lists

Start to become familiar with the list-processing functions in the Prelude, there are many useful functions already defined! See Hutton pp.285 – 287.

Main> [0,1,2] ++ [3,4]
[0,1,2,3,4]
Main> last [0,1,2,3,4]
4
Main> init [0,1,2,3,4]
[0,1,2,3]
Main> take 3 [0,1,2,3,4]
[0,1,2]
Main> drop 3 [0,1,2,3,4]
[3,4]
Main> takeWhile (<3) [0,1,2,3,4]
[0,1,2]
Main> dropWhile (<3) [0,1,2,3,4]
[3,4]
Main> splitAt 3 [0,1,2,3,4]
([0,1,2], [3,4])
Main> replicate 5 1
[1,1,1,1,1]
Main> [0,1,2] ++ [3,4]
[0,1,2,3,4]
Main> reverse [0,1,2,3,4]
[4,3,2,1,0]
Main> map (^2) [0,1,2,3,4]
[0,1,4,9,16]
Main> filter even [0,1,2,3,4]
[0,2,4]
Main> concat [[0],[1,2],[3,4]]
[0,1,2,3,4]

Many more advanced functions can be found in Data.List.
Useful Haskell Syntax: Characters and Strings

Characters (Hutton p.282)

Main> 'a'
'a'

Main> ['h','i','!']
"hi!"

Main Data.Char> isLower 'a'
True

Main Data.Char> isUpper 'a'
False

Main Data.Char> isAlpha 'a'
True

Main Data.Char> isDigit 'a'
False

Main Data.Char> ord 'a'
97

Main Data.Char> chr 97
'a'

Main Data.Char> digitToInt '9'
9

Main Data.Char> intToDigit 4
'4'

Main Data.Char> toUpper 'a'
'A'

Main Data.Char> toLower 'A'
'a'

Main Data.Char> nextChar 'a'
'b'

nextChar :: Char -> Char
nextChar c = chr ((ord c) + 1)

import Data.Char

Main Data.Char> import Data.Char
Useful Haskell Syntax: Characters and Strings

Strings are simply lists of Characters (Hutton p.282)

Main> ['h','i','!']
"hi!"

Main> "hi " ++ "there" ++ "!"
"hi there!"

Main> "hi there" !! 3
't'

Main> take 5 "hi there!"
"hi th"

Main> words "hi there!"
["hi","there!"]

Main> import Data.Char

Main Data.Char> map toUpper "hi there!"
"HI THERE!"

Any list function can be used on Strings. Check out Data.List!

This nifty function is provided in the Prelude
Case Expressions

A very useful kind of conditional expression is the case expression:

```
case expression of pattern -> result
     pattern -> result
     pattern -> result
     ...
```

In other languages, the case statement is an alternative to a long nested if-then-else, but in Haskell (of course!) it is more powerful, as it does pattern matching:

```
describe :: [a] -> String
describe []  = "empty"
describe [x] = "singleton"
describe _   = "big!"
```

*Main> describe [4]
"singleton"
Case Expressions

This solves the problem that lambda expressions can pattern match, but not do multiple patterns:

```haskell
describe :: [a] -> String
describe = \xs -> case xs of
    []   -> "empty"
    [x]  -> "singleton"
    _    -> "big!"
```
Let Expressions in Haskell

In Haskell we create local variables using let:

\[(\text{let } x = \langle\text{expr1}\rangle \text{ in } \langle\text{expr2}\rangle)\]

cylinder \( r \) \( h \) =
\[
\text{let sideArea} = 2 \times \text{pi} \times r \times h
\]
\[
\text{topArea} = \text{pi} \times r ^ {2}
\]
\[
\text{in sideArea} + 2 \times \text{topArea}
\]

Scope of local variables

let sq \( x \) = \( x \times x \) in (sq 5, sq 3, sq 2)

=> (25, 9, 4)

let \( x \) = 5
in let \( y \) = \( 2 \times x \)
in let \( z \) = \( x + y \)
in (\( w \to x \times y + z \)) 10

=> 65