Lecture 36:

- Haskell functions (cont)

Announcements:

- HW-7 out
- Project part 3 due

Lists and tuples can be nested to any depth

```
(555, ("Alice", "Smith"), ["manager", "developer"])

[[1, 2], [5, 4, 3], [2, 2]]

[('a', 'b'), ('c', 'd'), ('e', 'a')]
```

"Pairs" (2-tuples)

- Can access elements using `fst` and `snd` functions

```
Prelude> let p = (42, 67)
Prelude> fst p
42
Prelude> snd p
67
```

- Note this only works with pairs (2-tuples)!

Better approach: "Pattern Matching" (first look)

```
pairAdd1 p = ((fst p) + 1,(snd p) + 1)

... vs ...

pairAdd1 (x,y) = (x+1,y+1)

tripleAdd1 (x,y,z) = (x+1,y+1,z+1)
```

- we are "matching" on subcomponents of the structure
- *note*: this is **not** the same as a function parameter list!

Conditional Expressions via `if-then-else`

Haskell provides support for if-then-else statements

`if` c `then` e_1 `else` e_2

- c is a Boolean expression (that is, has type `Bool`)
- e_1 and e_2 are expressions
- if c evaluates to true, then if-then-else evaluates to e_1
- otherwise, the if-then-else evaluates to e_2

Q: What type of statement is Haskell's if-then-else?

- An expression! Either the result of e_1 or e_2 is returned

Q: What must be true of e_1's and e_2's types?

- They have to be the same!
- Lets say they have type T, what is the type of the if-then-else?

Simple Example

```
converge x = if x < 0
               then x + 1
               else x - 1
```

- can also put on one line
- or separate lines as above (requiring indentation)
- can also nest ...

Nested Example

```
min3 x y z = if x <= y
                then (if x <= z
                         then x
                         else z)
                else (if y <= z
                         then y
                         else z)
```

- Note that the parens are **not** required ... why not?
- ... can't have a "dangling else"

```
if(x < y)
   if(y < z)
      return x;
   else            // which if does this belong to?
      return z;
```

Recursive Example using `if-then-else`

```
fac n =
  if n <= 1 then 1
  else n * fac (n-1)
```

- yet another if-then-else formatting style

Let and where

We can define local bindings within functions

```
lendAmt amt bal =
   let reserve = 100
       newBal  = bal - amt
   in  if newBal < reserve
          then 0
          else amt
```

- similar to `ghci let`, but `ghci let` does not have an `in` clause
- `let <bindings> in <expression>` is itself an **expression**
- Can sometimes be more effecient (only evaluate expression once ...)

A `let` expression can be used in any subexpression:

```
Prelude> 3 + (let x=4 in x)
7

Prelude> 2 + (let x=3 in (let y=4 in x+y))
9
```

Note that parens are not needed above

Example where `let` is more efficient

```
checkVal x ys =
   if x == maximum ys
      then "x is max"
      else if x > maximum ys
              then "x is too high"
              else "x is not too high"
```

Q: What is "inefficient" here?

- we're calling `maximum` twice
- `maximum` needs to check all elements in `ys`

Q: How can we use `let` to make this more efficient?

```
checkVal x ys =
   let m = maximum ys
   in if x == m
         then "x is max"
         else if x > m
                 then "x is too high"
                 else "x is not too high"
```

We'll talk more about using `let` and recursion later ...

An alternative approach using `where` blocks

```
lendAmt amt bal =
 if newBal < reserve
    then 0
    else amt
 where reserve = 100
       newBal = bal - amt
```

- Sometimes easier to read
- Has a different semantics when used with patterns (more later)
- e.g., can't be nested like a `let` expression

Both `where` and `let` can be used to define nested functions:

```
-- avg of squared difference to the mean
variance2 mean x1 x2 =
   let squareDiff x = (x - mean)^2
   in (squareDiff x1 + squareDiff x2) / 2

-- avg of squared difference to the mean
variance2' mean x1 x2 =
   (squareDiff x1 + squareDiff x2) / 2
   where squareDiff x = (x - mean)^2
```

Function types

Functions have types (either given or inferred)

```
Prelude> not True
False

Prelude> :type not
not :: Bool -> Bool
```

- The -> is read as "**to**" or "**returns**"

  "`not` has the type `Bool` to `Bool`"

  "`not` takes a `Bool` and returns a `Bool`"

Another example

```
Prelude> succ 6
7

Prelude> :type succ
succ :: Enum a => a -> a
```

- Here we have a class constraint

  "for all `Enum` types `a`, `succ` has the type `a` to `a`"