Today

- Lists (cont)
- Haskell if-then-else

Assignments

- HW-2, R-2 due
- HW-3, R-3 out

Announcements

- Project status update due in 2.5 weeks (Feb. 11th)
- Quiz 2 on Thursday
Haskell list “enumeration” syntax

```
Prelude> [1..4]
[1,2,3,4]
```

- Works for enumerable values (e.g., ["bar"..'foo"] won't work)
- The syntax is pretty flexible

```
Prelude> [1, 4 .. 10]
[1,4,7,10]

Prelude> [1.0, 1.25 .. 2.0]
[1.0,1.25,1.5,1,75,2.0]
Prelude> [0.1, 0.3 .. 1.0] -- look out for precision!
[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]

Prelude> [10, 9 .. 1]
[10,9,8,7,6,5,4,3,2,1]
Prelude> [10, 6 .. 1]
[10,6,2]

Prelude> [1, 5 .. 4]
[1]

Prelude> ['a' .. 'd']
"abcd"
Prelude> ['a', 'c' .. 'g']
"aceg"

Prelude> [1, 4, 6 .. 20]
<interactive>:1:9: parse error in input ‘..’
```

- Similar to range in Python
Selection (if-then-else)

Haskell provides support for if-then-else statements

\[
\text{if } c \text{ then } e_1 \text{ else } e_2
\]

- \(c\) is a Boolean expression (that is, has type \text{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

\[
\text{converge } x = \text{if } x < 0 \\
\quad \text{then } x + 1 \\
\quad \text{else } x - 1
\]

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

\[
\text{min3 } x \ y \ z = \begin{array}{l}
\text{if } x \leq y \\
\quad \begin{array}{l}
\text{if } x \leq z \\
\quad \text{then } x \\
\quad \text{else } z \\
\text{else (if } y \leq z \\
\quad \text{then } y \\
\quad \text{else } z \\
\end{array}
\end{array}
\]

- Note that the parens are \textit{not} required ... why not?
- \ldots can't have a "dangling else"

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

Another Example

Use if-then-else to write a "safe" version of \texttt{tail}

Call your function \texttt{safeTail}, which should return [] if input is []

```
safeTail xs = if null xs
  then xs
  else tail xs
```
Let and where

We can define **local bindings** within functions

```haskell
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 efficient (only evaluate expression once ...)

Different indentation styles work, e.g.:

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

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

```haskell
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 \texttt{where} blocks

\begin{verbatim}
lendAmt \ amt \ bal = 
  if newBal < reserve then 0
  else \ amt
where reserve = 100
       newBal = bal - \ amt
\end{verbatim}

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

Both \texttt{where} and \texttt{let} can be used to define nested functions:

\begin{verbatim}
-- 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
\end{verbatim}