Today

- More Haskell Lists
- More on Defining Haskell Functions

Assignments

- HW7 due
- HW8 out later today
Exercise: Nested Lists

1. Given an example of a valid nested list in Haskell with at least two elements.

2. Give an example of how to create each list below using list concatenation of two non-empty lists.
   a). "Hello World!"
   c). [[10, 20], [30, 40], [50, 60]]

3. Show how to construct each list below using only the list cons operator.
   a). [1, 2, 3]
   b). "Hi"

4. Write the following versions of `listify2`, which should take two values and create a two element list out of the values (in reverse order), e.g., `listify2 4 5` should return [5, 4].
   a). Write your function using just a list expression.
   b). Write your function using the concatenate operator.
   c). Write your function using the cons operator.
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, 2, 4 .. 20]
<interactive>:1:9: parse error in input `..'

• Similar to range in Python
Haskell Tuples

A “tuple” is a fixed size collection of values

- Each tuple value can have a different type
  - Whereas each value of a list is of the same type
- Tuple values and types are denoted using parentheses ...

```haskell
Prelude> (1964, "Fist Full of Dollars")
(1964, "Fist Full of Dollars")

Prelude> :type (True, "yes")
(True, "yes") :: (Bool, [Char])
```

The special “unit” tuple

```haskell
Prelude> ()
()

Prelude> :type ()
() :: ()
```

- Both a value and a type
- Similar to void in C++/Java

There are no one-valued tuples in Haskell

```haskell
Prelude> :type ("hello")
("hello") :: [Char]
```
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 \texttt{fst} and \texttt{snd} functions

\begin{verbatim}
Prelude> let p = (42, 67)
Prelude> fst p
42
Prelude> snd p
67
\end{verbatim}

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

Better approach: “Pattern Matching” (first look)

\begin{verbatim}
pairAdd1 p = ((fst p) + 1,(snd p) + 1)
... vs ...

pairAdd1 (x,y) = (x+1,y+1)
\end{verbatim}

- we are “matching” on subcomponents of the structure (first two elements)
Conditional Expressions via 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 \textbf{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

\[
\text{min} 3 \ x \ y \ z = \text{if } x \leq y \\
\quad \text{then (if } x \leq z \\
\quad \quad \text{then } x \\
\quad \quad \text{else } z) \\
\quad \text{else (if } y \leq z \\
\quad \quad \text{then } y \\
\quad \quad \text{else } z)
\]

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

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

We can define local bindings within functions

\[
\text{lendAmt} \text{ amt bal} = \\
\quad \text{let reserve} = 100 \\
\quad \quad \text{newBal} = \text{bal} - \text{amt} \\
\quad \text{in} \quad \text{if newBal < reserve} \\
\quad \quad \text{then 0} \\
\quad \quad \text{else amt}
\]

- 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 ...)

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

```haskell
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?

```haskell
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

```lisp
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:

```lisp
-- 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
```