Today

- Recursion

Assignments

- R-8, HW-8 out (due Tues)

Announcements

- Quiz on Tuesday
More Function Types: Higher order functions

Parentheses are important in function types

Prelude> any even [1, 3 .. 11]
False

Prelude> :type any
any :: (a -> Bool) -> [a] -> Bool

• The first argument is a function from a to Bool

For example

Prelude> let any1 = any 1
... No instance for (Num (a -> Bool)) arising from the literal '1'

• Why doesn't any1 work? ... 1 is not a function from a to Bool

Another example

Prelude> :type even
even :: (Integral a) => a -> Bool

Prelude> let anyEven = any even

• This works! ... why? (Hint: what is the type of even?)
More examples of higher order functions

The `map` function applies a function to each element of a list

```haskell
Prelude> map even [1,2,3,4]
[False,True,False,True]
```

```haskell
Prelude> map (+1) [1,2,3,4]
[2,3,4,5]
```

The `zipWith` function applies a function to each zip pair

```haskell
Prelude> zipWith (+) [1,2,3] [10,20,30]
[11,22,33]
```

```haskell
Prelude> zipWith (<) [1,2,3] [10,20,30]
[True,True,True]
```

**Exercise 4**
Specifying Function Types

When defining functions, always good practice to specify function types

Although Haskell can infer them for you ...

• you should define types for all your functions:

```
add :: (Num a) => a -> a -> a
add x y = x + y
```

• Haskell checks declared and inferred types match (early warning)

• helps document your functions
Recursion

Basic idea:

• Solve a problem by breaking the problem into smaller (identical) subproblems
• Each subproblem makes a little bit of progress
• But taken together, end up solving the big problem

Sum the elements of a list

1. The sum of the empty list is 0 ... “Base Case”
2. The sum of a non-empty list is first elem plus sum of remaining list

One way to write this in Haskell

```haskell
mySum xs = if null xs
    then 0
    else head xs + mySum (tail xs)
```

How it works

```haskell
mySum [2,4,6]
  ==> 2 + mySum [4,6]
  ==> 2 + 4 + mySum [6]
  ==> 2 + 4 + 6 + mySum []
  ==> 2 + 4 + 6 + 0
```

Q: What is the type of mySum?

```haskell
Prelude> :type mySum
mySum :: (Num a) => [a] -> a
```
Q: How does Haskell know this?

• by looking at the function definition!

• for example:

  - `null :: [a] -> Bool` ... `xs` must be a list
  - `0 :: Num a => a` ... return type is a `Num` instance
  - `head :: [a] -> a` ... `xs` element
  - `(+) :: (Num a) => a -> a` ... `list elem types a Num instance`

Why recursion?

• Just as powerful as iteration (e.g., for/while loops)

• Often much easier to write, read, and understand with recursion
  
  – e.g., the “natural” way to define functions in mathematics

**Iteration in Haskell is performed through recursion**

• this means you’ll become a recursion expert!
Examples

Drop

Recall that drop n xs returns a list with first n elems of xs removed

Prelude> drop 2 "foobar"
"obar"

Prelude> drop 4 "foobar"
"ar"

Prelude> drop 4 [1,2]
[]

Prelude> drop 0 [1, 2]
[1, 2]

Prelude> drop 7 []
[]

Prelude> drop (-2) "foo"
"foo"

Q: What are the base cases?

- Empty list (nothing to drop)
- n ≤ 0 (nothing to drop)
Q: How can we define our own version of drop?

\[
\text{myDrop } n \ x s = \text{if } n \leq 0 \ \text{||} \ \text{null } x s \\
\hspace{1cm} \text{then } x s \\
\hspace{1cm} \text{else myDrop } (n-1) \ (\text{tail } x s)
\]

- if either base case, return list
- otherwise, call myDrop on smaller n and smaller xs

Q: What is the type of myDrop?

- Hint 1: \((\leq) :: (\text{Ord } a) \Rightarrow a \to \text{Bool}\)
- Hint 2: \((-) :: (\text{Num } a) \Rightarrow a \to a \to a\)

\[
\text{myDrop } :: (\text{Num } a, \text{Ord } a) \Rightarrow a \to [b] \to [b]
\]

**Exercise 5**
Exponentiation (Power)

Q: Using recursion, define a function \texttt{pow x y} that computes $x^y$ for $y \geq 0$

\begin{verbatim}
pow x y = if y == 0
        then 1
        else x * pow x (y-1)
\end{verbatim}

Q: What is the type of \texttt{pow}?

- Hint 1: $(\cdot) :: (\text{Num } a) \Rightarrow a \rightarrow a \rightarrow a$

\begin{verbatim}
pow :: (\text{Num } a, \text{Num } b, \text{Eq } b) \Rightarrow a \rightarrow b \rightarrow a
\end{verbatim}

Append

Q: Using recursion, define a function \texttt{append xs ys} that computes $xs \ ++ \ ys$

\begin{verbatim}
append xs ys = if null xs
               then ys
               else (head xs) : append (tail xs) ys
\end{verbatim}

The basic idea:

- appending an empty list to $ys$ is just $ys$ (base case)
- otherwise, create the new list by:
  - adding head of $xs$ to the result of a smaller append
  - which simply appends $ys$ to the tail of $xs$
How it works: w/out laziness

```
append [1,2] [3,4]
  ==> 1 : append [2] [3,4]
  ==> 1 : 2 : append [] [3,4]
  ==> 1 : 2 : [3,4]
  ==> [1,2,3,4]
```

Another example: w/ laziness

```
append [1,2] [3..]
  ==> (head [1,2]) : append (tail [1,2]) [3..])
  ==> (head [1,2]) : (head [2]) : append (tail [2]) [3..]
  ==> (head [1,2]) : (head [2]) : [3..]
```

- second list never has to (is never asked to) be evaluated
- result is called a "thunk" ("suspension", "delayed computation", "future")

Why does `append` have the type `[a] -> [a] -> [a]`?

- `null :: [a] -> Bool` ... `xs :: [a]`
- `(:) :: b -> [b] -> [b]` ... result must be of type `[b]`
- `then ys` ... `ys must of type [b]`
- `head xs :: a` ... so `a = b`

**Exercise 6**
Type Inference and the “Occurs Check” Error

Q: How can we define a `flatten` function in Haskell?

e.g., `flatten [[1,2],[3,4]]` should return `[1,2,3,4]`

```haskell
flatten xs = if null xs
    then xs -- this doesn’t work!
    else head xs ++ flatten (tail xs)
```

Q: What is the type of `flatten`?

1. `xs` must be a list since we call `null xs` so: `[]` -> `?`
2. `head xs` is a list since we use `++` on it so: `[[a]]` -> `?`
3. result must have same type as `head xs` so: `[[a]]` -> `[a]`
4. but result must be same type as `xs` (from `then xs`) so: `[a] ≡ [[a]]`
5. repeating this out gives `[[[ · · · a · · · ]]]` *infinite type!*

Q: How can we fix this?

```haskell
flatten :: [[a]] -> [a]
flatten xs = if null xs
    then [] -- this works!
    else head xs ++ flatten (tail xs)
```

Q: Why does this work?

- ... we broke the connection between the input type and the output type