Today

- Quiz 3
- Recursion

Assignments

- HW-4, R-4 out

Announcements

- Project status report next Thurs (11th)
The error function

- Useful for “error” cases
- Aborts execution (exception) without returning a value

Example:

```haskell
secondElem xs = if length xs >= 2 then head (head xs)
                   else error "list too short"
```

```haskell
> secondElem [1,2]
2
```

```haskell
> secondElem [1]
*** Exception: list too short
```

Q: What is the type of the error function?

```haskell
> :t error
error :: [Char] -> a
```

- Given a string returns a value of any type a

Q: Why does error return any type?

- Always returns a value of the “correct” type
- Thus, can be called from anywhere, without causing a type error
- (Again, never returns though ... throws an exception)
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 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?)

Exercise ...
Good Practice to Specify Function Types

Although Haskell can infer them for you ...

- you should define types for all your functions:

  \[
  \text{add} :: \text{(Num } a) \Rightarrow a \rightarrow a \rightarrow a \\
  \text{add } x \ y = x + y
  \]

- 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)
```

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 \texttt{drop \ n \ xs} returns a list with first \( n \) elems of \( xs \) removed

\begin{verbatim}
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"
\end{verbatim}

Q: What are the base cases?

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

\[
\text{myDrop } n \ x s = \begin{cases} 
  x s & \text{if either base case, return list} \\
  \text{myDrop} \ (n-1) \ (\text{tail} \ x s) & \text{otherwise, call myDrop on smaller } n \text{ and smaller } x s
\end{cases}
\]

Q: What is the type of myDrop?

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

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

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

\[
\text{pow} \ x \ y = \begin{cases} 
1 & \text{if } y == 0 \\
 x * \text{pow} \ x \ (y-1) & \text{otherwise}
\end{cases}
\]

Q: What is the type of \( \text{pow} \)?

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

\[
\text{pow} :: (\text{Num} \ a, \text{Num} \ b, \text{Eq} \ b) \Rightarrow a \rightarrow b \rightarrow a
\]

Append

Q: Using recursion, define a function \( \text{append} \ x s \ y s \) that computes \( x s \ ++ \ y s \)

\[
\text{append} \ x s \ y s = \begin{cases} 
 y s & \text{if null } x s \\
 (\text{head } x s) : \text{append} \ (\text{tail } x s) \ y s & \text{if } \text{otherwise}
\end{cases}
\]

The basic idea:

- appending an empty list to \( y s \) is just \( y s \) (base case)
- otherwise, create the new list by:
  - adding head of \( x s \) to the result of a smaller append
  - which simply appends \( y s \) to the tail of \( x s \)