Announcements

- HW due Monday
- Review Monday, Test Wednesday
- Friday: Can people bring laptops?
History of Haskell:

Meeting in 1987 to discuss state of functional programming.

At the time, there were many roughly equivalent functional languages.

Response to a talk: (by John Backus in '78) "Can Programming be Liberated from the von Neumann style?"

Named in honor of logician Haskell B. Curry.
Haskell:

Basic structure:

- Pure functional (so no variable assignment!)
- Lazy evaluation
- Statically typed (w/ strong typing and checking at compile time)
- Uses type inference (like Python)
- Very concise
Nice Features for us:
- on Turing
- website provides limited functionality
- easy to download & install
A first program: Quick sort

What is it?

divide & conquer sorting

$O(n^2)$ ($O(n \log n)$ expected)
void qsort(int a[], int lo, int hi) {
    int h, l, p, t;
    if (lo < hi) {
        l = lo; h = hi; p = a[hi];
        do {
            while ((l < h) && (a[l] <= p))
                l = l+1;
            while ((h > l) && (a[h] >= p))
                h = h-1;
            if (l < h) {
                t = a[l]; a[l] = a[h]; a[h] = t;
            }
        } while (l < h);
        a[hi] = a[l]; a[l] = p;
        qsort(a, lo, l-1);
        qsort(a, l+1, hi);
    }
}

// To sort array a[] of size n: qsort(a,0,n-1)
Haskell quicksort

```haskell
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (p:xs) = (quicksort lesser) ++ [p] ++ (quicksort greater)

where
  lesser = filter (< p) xs
  greater = filter (>= p) xs
```
Back to basics

- Type ghci to start
- Can do basic numerical ops.  
  Caution: 5 * -3 → error
- Booleans: ||, &&, not
- Type checking: 5 + "llama"
Functions

- Prefix notation, no parenthesis
  - succ 5
  - min 9 10

- Functions have highest precedence:
  - succ 9 + \(\max 5, 14, 1\)
  - \(\max 9, 10, 5, +1 = 16\)
  - \((\text{succ } 9) \ast 10\) \(= 100\)
No parenthesis!

$$\text{bar (3, "haha") in C}$$

$$\Rightarrow \text{bar 3 "haha"}$$

So $$\text{foo (bar 3)}$$

$$\Rightarrow \text{foo (bar (3)) in C}$$
Making functions

Open your favorite text editor.

doubleMe x = x + x

→ Save as firstex.hs

→ Type :l firstex at prompt.

→ Can use this function.

doubleMe 9 → 18

doubleMe 8.3 → 16.6
Another example

doubleUs x y = x*2 + y*2

Same as

doubleUs x y = doubleMe x + doubleMe y
IF statements

Must have an else. Why?

No matter what, need return value.

Ex: double Small Number $x = \begin{cases} \text{if } x > 100 \amp \text{then } x \\ \text{else } x \times 2 \end{cases}$

Ex.2: double Small Number’ $x = \begin{cases} \text{if } x > 100 \\ \text{then } x \\ \text{else } x \times 2 \end{cases} + 1$
Can define constant functions

\[
erin = \text{“It’s me, Erin!”}
\]

No input parameters

(In essence, this function works like a const variable.)

Note: \( a = 13 \) is same as:

\[
\text{let } a = 13
\]

in interactive mode
Lists

- homogeneous

- look like Python: [2, 4, 6, 8]

- a bit like C: "hello" is same
  ["h", "e", "l", "l", "o"]

- concatenate:
  [1, 2] ++ [3, 4, 5]

  "hello" ++ "world"
**Efficiency + lists**

- Appending to end of big list is slow:

  "really really big word" ++ '.'

  Why? **Must traverse the first list**

**Contrast**: Putting on front with : is fast:

`A`: "programming language"

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

↑ single element
Lists

Stored as \( \text{list} = \text{value} : \text{list} \)

So \( [1, 2, 3] \) is really \( 1 : 2 : 3 : [] \)

Can get an element:
\( [3.2, 1.1, 6.9, 42.3] \) // 2

Lists can contain lists:
- \( [[]] \)
  - \( \rightarrow \) \( [1, 2, 3], [5, 5], [4, 2, 1] \) // \( 1 : 1 \)
Head & Tail

Two big operators for lists

\[ \text{head} \ [5, 4, 3, 2, 1] \rightarrow 5 \]
\[ \text{tail} \ [5, 4, 3, 2, 1] \rightarrow [4, 3, 2, 1] \]

Also:

\[ \text{last} \ [5, 4, 3, 2, 1] \rightarrow 1 \]
\[ \text{init} \ [5, 4, 3, 2, 1] = [5, 4, 3, 2] \]

(All give errors on empty lists)
Other functions

- length
- sum
- null - Tor T
- product
- reverse
- elem - in in Python
- take : take 3 \([5,4,3,2,1]\)
  \[\rightarrow [5,4,3]\]
- drop
- maximum & minimum
Ranges

\[
\begin{align*}
[1..20] \\
[\text{\textquoteleft}a\text{\textquoteleft}.. \text{\textquoteright}z\text{\textquoteright}] \\
[\text{\textquoteleft}j\text{\textquoteright}.. \text{\textquoteright}l\text{\textquoteright}]
\end{align*}
\]

\[
\begin{align*}
\text{Can do:} \\
[2, 4.. 20] \\
[3, 6.. 20] \\
[20, 19.. 1]
\end{align*}
\]

\[
\begin{align*}
\text{Can\textquoteright\textquoteright do:} \\
[1, 2, 4, 8, 16.. 100] \\
[20.. 1] \\
[0.1, 0.3.. 1] \quad \text{why?}
\end{align*}
\]

\[
\begin{align*}
\text{how?} \\
(\text{Remember succ?})
\end{align*}
\]
Neat tricks

Get 1st 24 multiples of 13:
\[ \{13, 26, \ldots, 24+13\} \]

Better: take 24 \[ \{13, 26, \ldots\} \]

\text{infinite lists}
Infinite lists

\[ 1, 2, 3, \ldots \]

cycle list - cycles input list infinitely

Ex: take 10 (cycle [1, 2, 3, \ldots])

take 12 (cycle "LOL")

repeat val

Ex: take 10 (repeat 5)
List Comprehension

Based on set theory:

\[ \{ 2x \mid x \in \mathbb{N}, \ x \leq 10 \} \]

In Haskell:

\[ [x \times 2 \mid x \leftarrow [1..10]] \]

Can even refine (or filter):

\[ [x \times 2 \mid x \leftarrow [1..10], \ x \times 2 \geq 12] \]

\[ [x \mid x \leftarrow [50..100], \ \text{mod} \ x \ 7 = 3] \]