Today

- Higher-order functions (cont)
- Prolog intro

Assignments

- R-10, HW-10 due
- HW-11 due Thurs
Accumulating values (from the left) with \texttt{foldl}

\[
\texttt{foldl} :: (\text{a} \to \text{b} \to \text{a}) \to \text{a} \to \text{[b]} \to \text{a}
\]

\begin{itemize}
  \item \((\text{a} \to \text{b} \to \text{a})\) is the \textbf{step function}
  \item first \text{a} is an \textbf{accumulator}
  \item \([\text{b}]\) are the input values
  \item last \text{a} is the \textbf{accumulated value}
  \item new\_\text{accumulator} = \text{step\_function} \text{accumulator} \text{b\_value}
\end{itemize}

\begin{verbatim}
Prelude> foldl (+) 0 [1, 2, 3]
6
\end{verbatim}

\item This just sums up the list of values

\textbf{How it works ...}

\begin{verbatim}
foldl _ acc [] = acc
foldl step acc (x:xs) = foldl step (step acc x) xs
\end{verbatim}

\begin{verbatim}
foldl (-) 9 [5, 3, 1]
==> foldl (-) (9 - 5) [3, 1]
==> foldl (-) ((9 - 5) - 3) [1]
==> foldl (-) (((9 - 5) - 3) - 1) []
==> (((9 - 5) - 3) - 1)
==> 0
\end{verbatim}
Accumulating values from the right with foldr

foldr :: (b -> a -> a) -> a -> [b] -> a

Prelude> foldr (+) 0 [1, 2, 3] -- same as foldl in this case
6

• Similar to foldl but works right-to-left
• new_accumulator = step_function b_value accumulator
• ... where the b_value is from the list

How it works ...

foldr _ acc [] = acc
foldr step acc (x:xs) = step x (foldr step acc xs)

foldr (-) 9 [5, 3, 1] -- try 3 instead of 9
==> 5 - (foldr (-) 9 [3, 1])
==> 5 - (3 - (foldr (-) 9 [1]))
==> 5 - (3 - (1 - (foldr (-) 9 [])))
==> 5 - (3 - (1 - 9))
==> -6
Many recursive functions follow the fold pattern

\[
\text{filter} :: (a \to \text{Bool}) \to [a] \to [a]
\]
\[
\text{filter} p \ [\] = \[
\text{filter} p \ (x:xs)
| \ p \ x = x : \text{filter} p \ xs
| \ \text{otherwise} = \text{filter} p \ xs
\]

Q: How can \text{filter} be defined using \text{foldr}?

\[
\text{filter'} p \ xs = \text{foldr} \ \text{step} \ [\] \ xs
\]
\[
\quad \text{where} \ \text{step} \ x \ \text{acc}
| \ p \ x = x : \text{acc}
| \ \text{otherwise} = \text{acc}
\]

- For example ... 

\[
\text{filter'} \ \text{odd} \ [1,2,3]
\]
\[
\Rightarrow \ \text{foldr} \ \text{step} \ [\] \ [1,2,3]
\]
\[
\Rightarrow \ \text{step} \ 1 \ (\text{foldr} \ \text{step} \ [\] \ [2,3])
\]
\[
\Rightarrow 1 : (\text{foldr} \ \text{step} \ [\] \ [2,3])
\]
\[
\Rightarrow 1 : (\text{step} \ 2 \ (\text{foldr} \ \text{step} \ [\] \ [3]))
\]
\[
\Rightarrow 1 : (\text{foldr} \ \text{step} \ [\] \ [3])
\]
\[
\Rightarrow 1 : (\text{step} \ 3 \ \text{foldr} \ \text{step} \ [\] \ [])
\]
\[
\Rightarrow 1 : (3 : (\text{foldr} \ \text{step} \ [\] \ []))
\]
\[
\Rightarrow 1 : (3 : [])
\]

Q: How can \text{filter} be defined using \text{foldl}?

\[
\text{filter'} p \ xs = \text{foldl} \ \text{step} \ [\] \ xs
\]
\[
\quad \text{where} \ \text{step} \ \text{acc} \ x
| \ p \ x = \text{acc} ++ \ [x]
| \ \text{otherwise} = \text{acc}
\]
We can also define map using foldr

map' :: (a -> b) -> [a] -> [b]
map' f xs = foldr step [] xs
    where step x ys = f x : ys

For example ...

    map' odd [1,2,3]
    ==> foldr step [] [1,2,3]
    ==> step 1 (foldr step [] [2,3])
    ==> odd 1 : (foldr step [] [2,3])
    ==> odd 1 : (step 2 (foldr step [] [3]))
    ==> odd 1 : (odd 2 : (foldr step [] [3]))
    ==> odd 1 : (odd 2 : (step 3 (foldr step [] [])))
    ==> odd 1 : (odd 2 : (odd 3 : (foldr step [] [])))
    ==> odd 1 : (odd 2 : (odd 3 : []))

Why care about these higher-order functions?

- In general, should use them whenever possible ...
- ... Can make functions easier to understand (shorter)
- ... Well behaved (fewer bugs)
- ... Optimization
List Comprehensions

List comprehensions mimic set definitions (“set builder” notation):

\[ A \times B = \{(a, b) \mid a \in A \land b \in B\} \]

- The cartesian product operation

Using list comprehensions:

\[ \text{cprod } xs \; ys = [(x,y) \mid x <- xs, y <- ys] \]

- here xs and ys have to be in scope

Another example:

\[ [x*2 \mid x <- [1..10]] \]

With a predicate (to filter)

\[ [x*y \mid x <- [1..10], y <- [1..3], \text{even } x] \]

In general, a lot like lambda functions

- but for defining lists “on the fly”
- syntactic sugar for defining lists (list set builders)
- can use anywhere you’d expect a list
Wrapping up

What we didn’t cover

- A lot! ... Haskell has many features

- The IO type (recall “purity”)
  - Various I/O operations
  - All the normal things you’d expect to write real apps

- Monads
  - A pattern (data type) to chain together a list of operations
  - Functional machinery to sequence commands
  - In homework, the do expression is an example
    \[
    \text{do } \text{input} \leftarrow \text{readFile} \text{ inputFile} \\
    \text{putStrLn} \text{ (function input)}
    \]
  - Appears imperative, but is still functional

- The Monad typeclass has a sequencing function >>=
  \[
  \text{Prelude> } :\text{type (>>=)} \\
  (\gg\gg) : \text{ (Monad m)} \Rightarrow m \ a \rightarrow (a \rightarrow m \ b) \rightarrow m \ b
  \]
  - extracts (unwraps) value on left from the Monad
  - passes it to a function that returns a wrapped result value
• And a return function

```haskell
Prelude> :type return
return :: (Monad m) => a -> m a
```

– Takes a value and wraps it into a Monad member

• Here is a (silly) example ... Note Maybe is a Monad

```haskell
wrapIt x = Just x

go = wrapIt "hi" >>=
  \v1 -> wrapIt (head v1) >>=
  \v2 -> return v2

Prelude> go
Just 'h'
```

– return for Maybe is defined to create a Just value
– i.e., return x = Just x

• The do keyword provides a shorthand ...

```haskell
go' = do
  v1 <- wrapIt "hi"
  v2 <- wrapIt (head v1)
  return v2                         # return does the wrapping
```

• Here is a better example using the IO monad

```haskell
do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn ("Nice to meet you " ++ name)
```
Prolog

Prolog stands for **PROgramming in LOGic**

- Prolog is a “logic-based” programming language
- Syntax is close to **first-order logic** (FOL)

The first Prolog system was developed circa 1972

- Colmerauer, Roussel, Kowalski
- Univ. of Aix-Marsielle and Univ. of Edinburgh
- Born out of work in **Automated Theorem Proving**
- As well as Natural Language Understanding
- Because of this, considered an “AI” language
Some features of Prolog

- Extremely simple syntax
- Declarative (what not how)
  - Haskell leans declarative as well
- A very different way to think about programming
  - Use “rules” or “constraints”
  - Hard to pick up at first ...
- Efficient at certain problems
  - Prolog runtime is fairly small ...
  - Prolog itself is generally fast (proof trees)
- No data types!

Prolog programming today

Not widely used today to build applications ...


- Many languages are “prolog” like
  - SQL (queries)
  - Datalog (Prolog like)
  - Answer Set Programming (constraints)
  - Erlang (concurrent/parallel computing ... mixes a bit of Prolog syntax with functional programming)
SWI Prolog

We’ll be using SWI Prolog ...

- [http://www.swi-prolog.org](http://www.swi-prolog.org)
- freely available, open-source implementation
- supports most operating systems (mac, linux, windows)
- available on ada (run swipl at the command prompt)
- lots of documentation online

Like with Haskell

- you are encouraged to download, install, and run on your own machine
- if you need help, come see me

Starting SWI prolog on ada

```
$ swipl
Welcome to SWI-Prolog ...

For help, use ?- help(Topic) ...

?-
```

- The ?- symbol is the query prompt
Prolog syntax

Prolog programs are built from “terms”

- A term is either:
  - a constant
  - a variable
  - or a structure

Constants are either “numbers” or “atoms”

- numbers are integers or floating point values
- atoms represent everything else

Atoms

- Atoms usually begin with lowercase letters
  - can contain letters, digits, and underscores (_)
- Atoms in single quotes denote strings
  - any character can occur between single quotes
  - different from a list of char (denoted by double quotes)
- Some atoms contain only symbols (e.g., +)
- The true and false atoms are special
Examples of atoms

x
washington
a123BC_d

• note these are different than in most other languages!!!
• these are *not* variables!

'Washington'
'this is a long atom with spaces'
'\one back slash'
'this statement isn’t true'

• the last one prints “isn’t”
• 'isn’t’ does the same thing
Variables

Variables begin with either …

- uppercase characters
- an underscore (don’t care / wildcard)

Variables can contain letters, digits, underscores …

\[
\begin{align*}
X \\
A \\
_1234 \\
My\_Var \\
Xs
\end{align*}
\]
Structures

Structures normally take the form:

\[ p(t_1, t_2, \ldots, t_n) \]

- \( p \) is an atom and called a **functor**
- each arg \( t_i \) is a **term**
- the whole thing is a **relation**
- the number of terms \( n \) is the **arity**

Examples

- \( \text{mother}(\text{necessity}, \text{invention}) \)
- \( \text{employed_by}(X, \text{gonzaga\_university}) \)
- \( \text{employee}(\text{‘John’}, \text{‘Smith’}, 55567, 28, x5555) \)
- ‘a weird atom!?’(x, y, Var)
- \( \text{node}(\text{‘a’}, \text{node}(\text{‘b’}, \text{nil}, \text{nil}), \text{nil}) \)

- an atom is a structure with no arguments (arity = 0)
Facts and Rules

A knowledge base (KB) consists of facts and rules

- all statements end in periods ...

/* facts */
mother(necessity,invention).
mother(june,wally).
father(experience,wisdom).
father(ward,wally).
human(june).
human(wally).
human(ward).

/* rules */
parent(X,Y) :- mother(X,Y). % parent if mother
parent(X,Y) :- father(X,Y). % parent if father
human_parent(X,Y) :- parent(X,Y), human(X). % "and"

- :- is an infix functor read “if”
- e.g., "X is the parent of Y if X is the mother of Y"
- and "X is a human parent of Y if X is the parent of Y and X is human"

Load these into swi by “consulting” the file:
- e.g.: ?- consult('the_file.pl').
- or: ?- ['the_file.pl'].
- or: ?- [the_file].
- To exit swi: ?- halt.
?- consult('family.pl').
% family compiled 0.00 sec, 1,816 bytes

?- [family] % alternative load
% family compiled 0.00 sec, 1,816 bytes

?- parent(X,Y). % this is called a "query"
X = necessity,
Y = invention ;
X = june,
Y = wally ;
... etc ... .

• the ; tells prolog to search for more results
• you can hit enter to stop searching
Prolog is agnostic about inputs and outputs

?- parent(X,invention). % X is an "output"
X = necessity.

?- parent(necessity,Y). % Y is an "output"
Y = invention.

?- parent(X,Y). % X & Y are "outputs"
X = invention,
Y = wisdom.

?- parent(necessity,invention). % both inputs 
true.

- Some rules require certain “binding patterns” ... more later