Today

- Haskell Basics (cont)

Assignments

- HW 7 and Revised Proj Prop out (due Tues)
Haskell Lists

Lists in Haskell take the form

\begin{verbatim}
Prelude> [1,2,3,4] [1,2,3,4]
\end{verbatim}

- Lists can be of any length (including empty `[]`)
- But all values in a list must be of the same type ("homogeneous")

\begin{verbatim}
Prelude> [1, 2, False]
<interactive>:1:4:
No instance for (Num Bool)
    arising from the literal `1' at<interactive>:1:4 ...

Prelude> [1, 5.0]
[1.0, 5.0]

Prelude> :type [1, 5.0]
[1, 5.0] :: (Fractional t) => [t]
\end{verbatim}

- Last one works since the value 1 can be of any number type
Characters and Strings

• Strings in Haskell are just lists of Characters

```haskell
Prelude> :type 'H'
'H' :: Char

Prelude> :type "Hi"
"Hi" :: [Char]

Prelude> :type ['H', 'i']
['H', 'i'] :: [Char]

Prelude> "Hi" == ['H', 'i']
True
```
Lists and Types

When defining lists in Haskell, we do not have to give

- The type of the list
- The size of the list

Lists though have both ... thanks to Haskell type inference

- So instead of writing something like this (C++)
  
  int myList[4] = {1, 2, 3, 4};

- We simply write
  
  Prelude> let myList = [1, 2, 3, 4]

- And we still get static typing (this is a good thing!)
  
  Prelude> :type myList
  myList :: Num t => [t] -- [t] s.t. t is a num type
List Operations

List concatenation (++)

- Returns an entirely new list
- Values in the second list are appended to the values of the first list

Prelude> [3, 1, 3] ++ [3, 7]
[3,1,3,3,7]

Prelude> [True, False] ++ []
[True, False]

List construction (:)

- List construction is also often called “cons”
- Creates entirely new list with 1 prepended to values of given list
- Entire list can be written as a sequence of cons operations

Prelude> 1 : [2, 3]
[1,2,3]

Prelude> 1 : 2 : 3 : []
[1,2,3]

- Is cons (:) right or left associative?
  - Right associative!
  - 1 : (2 : (3 : []))
  - 1 : 2 is a type error since second operand is not a list!
List head (aka “car”) gives first value of a list

    Prelude> head [4, 1, 5, 3]
    4

    Prelude> head []
    *** Exception: Prelude.head: empty list

List tail (aka “cdr”) gives list minus head value

    Prelude> tail [4, 1, 5, 3]
    [1, 5, 3]

    Prelude> tail [1]
    []

    Prelude> tail []
    *** Exception: Prelude.tail: empty list

List null checks for the empty list

    Prelude> null [4, 1, 5, 3]
    False

    Prelude> null []
    True
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 ...

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

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

The special “unit” tuple

```
Prelude> ()
()

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

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

There are no one-valued tuples in Haskell

```
Prelude> :type ("hello")
("hello") :: [Char]
```
Lists and tuples can be nested to any depth

\[(555, ("Alice", "Smith"), ["manager", "developer"])]

\[[[1, 2], [5, 4, 3], [2, 2]]\]

\[(['a', 'b'), ('c', 'd'), ('e', 'a')]]\]

Homework uses “pairs” (2-tuples)

- Can access elements using \texttt{fst} and \texttt{snd} functions

\[
Prelude> \text{let } p = (42, 67)
Prelude> \text{fst } p
42
Prelude> \text{snd } p
67
\]

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

Better approach: “Pattern Matching”

\[
pairAdd1 p = ((\text{fst } p) + 1, (\text{snd } p) + 1)
\]

... vs ...

\[
pairAdd1 (x,y) = (x+1, y+1)
\]

- we are “matching” on subcomponents of the structure (first two elements)
- more on pattern matching later
Selection (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 \( \text{Bool} \))
- \( e_1 \) 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

\[
\text{min3 } x \ y \ z = \text{if } x \leq y \\
\quad \text{then (if } x \leq z \\
\quad \quad \text{then } x \\
\quad \quad \text{else } z) \\
\quad \text{else (if } y \leq z \\
\quad \quad \text{then } y \\
\quad \quad \text{else } z)
\]

- 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;
```