Today

- Quiz 1
- Haskell lists, tuples, and conditionals

Assignments

- HW-2, R-2 out
Haskell Functions (cont)

Functions in Haskell are single expressions

- e.g., we don’t need to write: `add x y = return x + y`
- this actually means something different in Haskell (more later)

Functions always evaluate to a single value

- Haskell is “expression oriented”
- An expression is a statement that always evaluates to a value

Functions can be defined to take zero or more arguments

\[
x = 5 \\
e = 2.71828 \quad \text{-- or } e = \text{exp} \ 1
\]

- These are actually zero-argument functions!
- Here we bind a constant expression (value) to a name (immutable “variable”)

Once bound, the value of the variable cannot change

```haskell
e = 2.71828 \\
e = \text{exp} \ 1 \\
Prelude> :l example.hs
... Multiple declarations of Main.e ...
```

- e refers to a value, not a location in memory (like in most languages)
Haskell Boolean Values

- Boolean values are True or False (instead of 1 and 0)

  Prelude> True && False
  False

  Prelude> False || True
  True

  Prelude> True && 1

  <interactive>:1:8:
  No instance for (Num Bool)
  arising from the literal ’1’ at <interactive>:1:8
  Possible fix: add an instance declaration for (Num Bool)
  In the second argument of ’(&&’), namely ’1’
  In the expression: True && 1
  In the definition of ’it’: it = True && 1

- Dissecting the error message:

  No instance for (Num Bool)

  - ghci tries to treat the numeric value 1 as a Bool (which fails)
  - Here it is saying Bool is not a member of the Num (numeric) types
  - We’ll talk more about Haskell typing later

    Possible fix: add an instance declaration for (Num Bool)

    - ghci is suggesting a way to fix the problem

      In the second argument ...

    - The remaining part is telling us where the error occurs
Boolean comparisons are similar to C derivatives (C++, Java, etc.)

Prelude> 1 == 1
True

Prelude> 2 < 3
True

Prelude> 4 >= 3.99
True

Prelude> 2 /= 3  -- instead of !=
True

Prelude> not True  -- instead of !
False
First Look at Haskell Typing (More later)

So far we have not needed to declare any types

- Haskell automatically infers the types of values for us!
- Saves us from having to write them (but still statically typed)
- We can see these types using the :type command

```
Prelude> let x = 5
Prelude> :type x
x :: Integer -- arbitrarily large Int

Prelude> let x = 3.14
Prelude> :type x
x :: Double -- Double by default

Prelude> let x = True
Prelude> :type x
x :: Bool

Prelude> :type False
False :: Bool

Prelude> :type 5
5 :: (Num t) => t
```

- We’ll talk more about the last one later (and why different than the first)
  - States that 5 has a type t that is a member of the Num class of types
  - The value is compatible with any of the Num types
Haskell Lists

Lists in Haskell take the form

```haskell
Prelude> [1,2,3,4]
[1,2,3,4]
```

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

```haskell
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]
```

```haskell
Prelude> :type [1, 5.0]
[1, 5.0] :: (Fractional t) => [t]
```

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

Exercise: Nested Lists
Characters and Strings

- Strings in Haskell are just lists of Characters

```
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 :: [Integer]`

  :: is read “has the type” (or just “has type”)

  [Integer] is read “list of Integer”

  So “myList has type list of Integer”
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: List Ops
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)

- access elements using `fst` and `snd` functions

Prelude> let p = (42, 67)
Prelude> fst p
42
Prelude> snd p
67
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

\[
\text{converge } x = \text{ if } x < 0 \text{ then } x + 1 \text{ else } x - 1
\]

- can also put on one line
- or separate lines as above (requiring indentation)
- can also nest ...
Nested Example

\[
\text{min3 } x \ y \ z = \begin{cases} 
    x & \text{if } x \leq y \\
    y & \text{if } y \leq z \\
    z & \text{otherwise}
\end{cases}
\]

- Note that the parens are *not* required ... why not?
- ... can't have a “dangling else”

```
if (x < y)
    if (y < z)
        return x;
    else // which if does this belong to?
        return z;
```

Another Example

Use if-then-else to write a “safe” version of tail

Call your function `safeTail`, which should return `[]` if input is `[]`

```
safeTail xs = if null xs
    then xs
    else tail xs
```