Today

- Haskell Basics (cont)

Assignments

- Exam 2 out
- HW7 due Tues (4/7)
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 types for us!
- We can see these types using the :type command

    Prelude> :type False
    False :: Bool

    Prelude> :type True
    True :: Bool

    Prelude> :type 'a'
    'a' :: Char

    Prelude> :type "ab"
    "ab" :: [Char]       -- List of Char

    Prelude> :type 3 == 4
    3 == 4 :: Bool       -- Lazy evaluation

    Prelude> :type 3
    3 :: Num t => t      -- For now: any number type

    Prelude> :type 3.14
    3.14 :: Fractional t => t  -- More later: any non-int num type

- We’ll talk more about the last two later ...
  - 3 has a type t, and t is a Num type
  - thus, 3 is compatible with any of the Num types (including Integer)
Haskell Lists

Lists in Haskell take the form

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** ("homogeneous")

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]

- 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