Today

- Quiz 5
- Haskell lists, tuples, and conditionals

Assignments

- R-6, HW-6 due
A First Look at Defining Functions

Functions can be defined directly in ghci using `let`

Prelude> let \( f p_1 p_2 \ldots p_n = e \)

- \( f \) is the function name
- \( p_1 \ldots p_n \) are formal parameters (no commas)
- \( e \) is an expression (i.e., evaluates to a value)
- introduces a **binding**: expression \( e \) is bound to the name \( f \)

A simple example:

Prelude> let add x y = x + y

Better and more convenient to use source files

-- ex1.hs
-- add function
add x y = x + y

- Source files can be loaded into ghci

Prelude> :load ex1
[1 of 1] Compiling Main ( example.hs, interpreted )
Ok, modules loaded: Main.
*Main> add 3 4
7

- By convention `.hs` used as file extension
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

```haskell
x = 5
e = 2.71828  -- or e = 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 = 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 \texttt{: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 \texttt{Num} class of types
  - The value is compatible with any of the \texttt{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
```

```haskell
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!