Today

- Haskell (cont)

Assignments

- HW 7 out
Using ghci as a Calculator

From the command line: (using Mac OS X or Linux)

```
$ ghci
GHCi, version ...
Loading package ...
...
Prelude>
```

Simple arithmetic

```
Prelude> 2 + 2
4

Prelude> 31337 * 100
3133700

Prelude> 7 / 2
3.5
```

Can call operators using *infix* (above) notation and as functions

```
Prelude> (+) 2 2
4
```

- In fact "+" is just a function
- To use ops as regular function calls, enclose op in parens (+)
A First Look at Haskell Functions

In Haskell, functions are called like this:

Prelude> f a1 a2 a3

• \( f \) is the function name
• \( a_1 \ a_2 \ a_3 \) are arguments
• Note no commas and no parentheses

You can add parentheses, but like this:

Prelude> (f a1 a2 a3)

• That is, you “wrap” the entire function call in parenthesis
• The expression \( f(a_1, \ a_2, \ a_3) \) means something different!
Functions are called from left-to-right in Haskell

- Let’s say we defined two functions:
  - an `add` function with two parameters
  - an `inc` function with one parameter
  - note that `add` and `inc` are **not** defined in Haskell

- We would call `add` like this:
  ```prelude
  Prelude> add 3 4
  ```

- What is the bug in the following?
  ```prelude
  Prelude> inc add 3 4
  ```

- Only works if `inc` takes **three** arguments (since `inc` is leftmost function)

- To compose function calls, use parentheses:
  ```prelude
  Prelude> inc (add x y)
  ```

- Here we apply `inc` to the result of calling `add` on 3 and 4 (composition)

- Can save parenthesis using function application operator ($)
  ```prelude
  Prelude> inc $ add x y
  ```
Exercise

Consider the expression \(3 + (4 \times 5)\). Write this expression in Haskell using:

a). The “functional” (prefix) version of + and infix version of *

\((+) \ 3 \ (4 \times 5)\)

b). The “functional” version of both + and *

\((+) \ 3 \ ((*) \ 4 \ 5)\)

c). The function application operator $ (this one is a bit tricky)

\((+) \ 3 \ $ \ ((*) \ 4 \ 5)\)

- “\((+) \ 3\)" is really a function that takes a number to add to 3
- e.g., in λ-calculus: \((λx.(λy.(+xy)))3 = (λy.(+3y))\)
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 \textit{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

```
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

```
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!
- 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
  x :: (Num a) => a -- For now: any number type

  Prelude> :type 3.14
  x :: (Fractional a) => a -- More later: any non-int type

- We’ll talk more about the last two later
  - E.g., 3 has a type `a` that is a member of the `Num` class of types
  - The value 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

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

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

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

Prelude> "Hi" == ['H', 'i']
True
```
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 ...

\[
\text{Prelude}\> (1964, "Fist Full of Dollars") \\
(1964, "Fist Full of Dollars")
\]

\[
\text{Prelude}\> :\text{type} (\text{True}, "yes") \\
(\text{True}, "yes") :: (\text{Bool}, [\text{Char}])
\]

The special “unit” tuple

\[
\text{Prelude}\> ()
\]

\[
\text{Prelude}\> :\text{type} () \\
() :: ()
\]

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

There are no one-valued tuples in Haskell

\[
\text{Prelude}\> :\text{type} ("hello") \\
("hello") :: [\text{Char}]
\]
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

\begin{verbatim}
Prelude> let p = (42, 67) Prelude> fst p 42 Prelude> snd p 67
\end{verbatim}

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

Better approach: “Pattern Matching”

\begin{verbatim}
pairAdd1 p = ((fst p) + 1,(snd p) + 1)
\end{verbatim}

\begin{verbatim}
... vs ...
\end{verbatim}

\begin{verbatim}
pairAdd1 (x,y) = (x+1,y+1)
\end{verbatim}

- 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

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

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

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

- 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;
```