Today
- Haskell Basics

Assignments
- Exam 2 out
- HW7 due Tues (4/7)
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 \textit{infix} (above) notation and as functions

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

- In fact “+” is just a function!
- Note that you shouldn’t write arithmetic expressions this way though!
A First Look at Haskell Functions

In Haskell, functions are called like this:

```
Prelude> f a1 a2 a3
```

- `f` is the function name
- `a1 a2 a3` 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(a1, a2, a3)` means something different!
Functions are called from left-to-right in Haskell

- Consider the “built-in” functions:
  - min and max, each with two parameters
  - pred and succ each with one parameter

- We would call min and pred like this:

  Prelude> min 3 4
  Prelude> pred 3

- What is the bug in the following?

  Prelude> pred min 3 4

- Only works if pred takes three arguments (since pred is leftmost function)
  - function calls actually work similarly to λ-calculus function application
  - e.g., pred (min 3 4) versus (pred min) 3 4
  - but type checking prevents us from treating these equally (more later)

- To compose function calls like this, use parentheses:

  Prelude> pred (min 3 4)

- Here we apply pred to the result of calling min on 3 and 4 (composition)

- Can save parenthesis using the function application operator ($) 

  Prelude> pred $ min 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 *

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

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: $(\lambda x. (\lambda y. (+)xy))3 = (\lambda y. (+)3y)$
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 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: \texttt{add x y = return x + y}
- \texttt{return} means something different in Haskell (more later)

Functions always \underline{evaluate to a single value}

- Haskell is "expression oriented"
- An expression is a statement that \textit{always} evaluates to a value

Functions can be defined to take zero or more arguments

\begin{verbatim}
x = 5
e = 2.71828  -- or e = \texttt{exp} 1
\end{verbatim}

- 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

\begin{verbatim}
e = 2.71828
e = \texttt{exp} 1
Prelude> :l example.hs
... Multiple declarations of 'Main.e' ...
\end{verbatim}

- \texttt{e} refers to a \texttt{value}, not a location in memory (like in most languages)