Today

• Getting started with Haskell

Assignments

• HW-1, R-1, PROJ-1 Proposal due
• HW-2, R-2 out

Announcements

• Quiz 1 on Thur
**Very Brief History of Haskell**

"Prehistory"

- Alonzo Church invented "lambda calculus" in the 1930s
- John McCarthy created Lisp based on it in 1958
- Robin Milner developed ML in the 1970s

**Haskell**

- By the late 1980s there were many functional languages
- A group of researchers got together (in Portland, OR!) to develop a standard/common FP language
- This produced Haskell 1.0, and later Haskell 98
What is Unique About Haskell?

Some of the major features of Haskell

1. A purely functional language
   - In general, functions do not have **side effects** (do not modify state)
     - some nice features: memoization, recursion
   - Values (variables) are **immutable**
   - Functions (and operations) always produce **entirely new values**
   - This is very different than most other PLs

2. Static typing
   - All type checking done at compile time (statically)
   - Employs **type inference** (unobtrusive—w/out type annotations)

3. “Strong” typing
   - Guarantees a program cannot contain certain type errors
   - Haskell places limits on type conversion (implicit/explicit)

4. Functions are “first-class” objects
   - i.e., used just like any other kind of value in the language
   - e.g., functions can be defined that take functions as parameters (and call them in the function body)
   - Can **create** new functions during program execution
   - Can **store** functions in data structures
   - Can **pass** functions as arguments to other functions
   - Can **return** functions as values of other functions
5. Lazy evaluation
   - Defer computation until the result is needed
   - One benefit: possible performance gain (no needless computations)
     - e.g., using quicksort, can ask for first (first two, etc.) values, without sorting entire list
   - Another benefit: “infinite” data structures
     - and in particular, the ability to compute with them
     - somewhat similar to iterators
   - Another benefit: programmer-defined control structures
     - e.g., short circuit evaluation of if-then-else
     - this means you don’t need special constructs for control flow

6. Expression-oriented
   - All statements return values (e.g., even if statements)
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
- \( a1 \ a2 \ a3 \) 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

- 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:

  ```haskell
  Prelude> add 3 4
  ```

- What is the bug in the following?

  ```haskell
  Prelude> inc add 3 4
  ```

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

- To compose function calls, use parentheses:

  ```haskell
  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 ($)

  ```haskell
  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 $\times$

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

b). The “functional” version of both $+$ and $\times$

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

c). The function application operator $(\mathit{this \ one \ is \ a \ bit \ tricky})$

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

$\text{“}(+) \, 3\text{” is really a function that takes a number to add to 3}$

$\text{e.g., in $\lambda$-calculus: } (\lambda x. (\lambda y. (x \ y))) 3 = (\lambda y. (x \ y)) 3 y$
**A First Look at Defining Functions**

**Functions can be defined directly in ghci using** `let`

Prelude> `let f p_1 p_2 ... p_n = e`

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

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