Computer Science 312

Functional Programming and Haskell

The “Software Crisis”

- Improvements in processing speed and memory capacity (Moore’s Law) allow applications to grow in size and complexity

- As software grows in size and complexity, it becomes more challenging to debug, verify, and maintain

- A crisis point was reached in the mid-1970s

A Primary Culprit: Side Effects

- Much of the complexity of software depends on the fact that programs change the state of the computer with time
- The number of interactions between states can grow exponentially with the number of states
- Many of these interactions, also called side effects, can lead to errors

Early Remedies: Restrict Interactions

- Modularize software, to support information hiding, security, and independent, plug-and-play testing and development (Modula, Ada)

- Factor code into objects and classes, with data encapsulation, polymorphism, and inheritance (Smalltalk, C++, later Java, Python)

- State survives, still challenging to detect errors due to side effects

Example Changes of State

- A method can change the states of its parameters
- A method can change the state of the current object’s instance variables, either directly or by calling other methods that do so
- A method can modify class variables, module variables, or any others that are in scope
- A method can perform I/O (files, interactive, network)

Another Remedy: Drop the States!

- Think of computation as a set of transformations of data values into other data values, rather than as a sequence of changes of state

- Each transformation guarantees the same results for the same data values

- One constructs a set of transformations just like one constructs a proof – verification built in!

Functional Programming

- A program consists of a set of cooperating functions
- Each function is ***pure***, guaranteeing the same results for the same data values
- There is just single assignment, for parameters and for temporary variables
- ***No*** side effects!

Historical Precedents

- Haskell Curry – combinatorial logic (1930)
- Alonzo Church – recursive lambda calculus (1940)
- John McCarthy – the first LISP system (1959)
- John Backus – ACM Turing Award lecture (1977)
- First purely functional languages – ML, Miranda, Haskell (1970s and 80s)

Haskell

- Created by a committee in 1987, to collect the best features of existing functional languages
- First definition released to the public in 1990
- Static, compile-time type checking
- Iteration by means of recursion
- Functions as first-class data, higher-order functions

Haskell

- Pattern matching simplifies control
- Lazy evaluation of function arguments
- Compiles to native code
- Strictness is augmented with “monads” for stateful computations, such as I/O and generating random numbers

Obtaining Haskell

- Download the Haskell Platform and documentation from https://www.haskell.org/

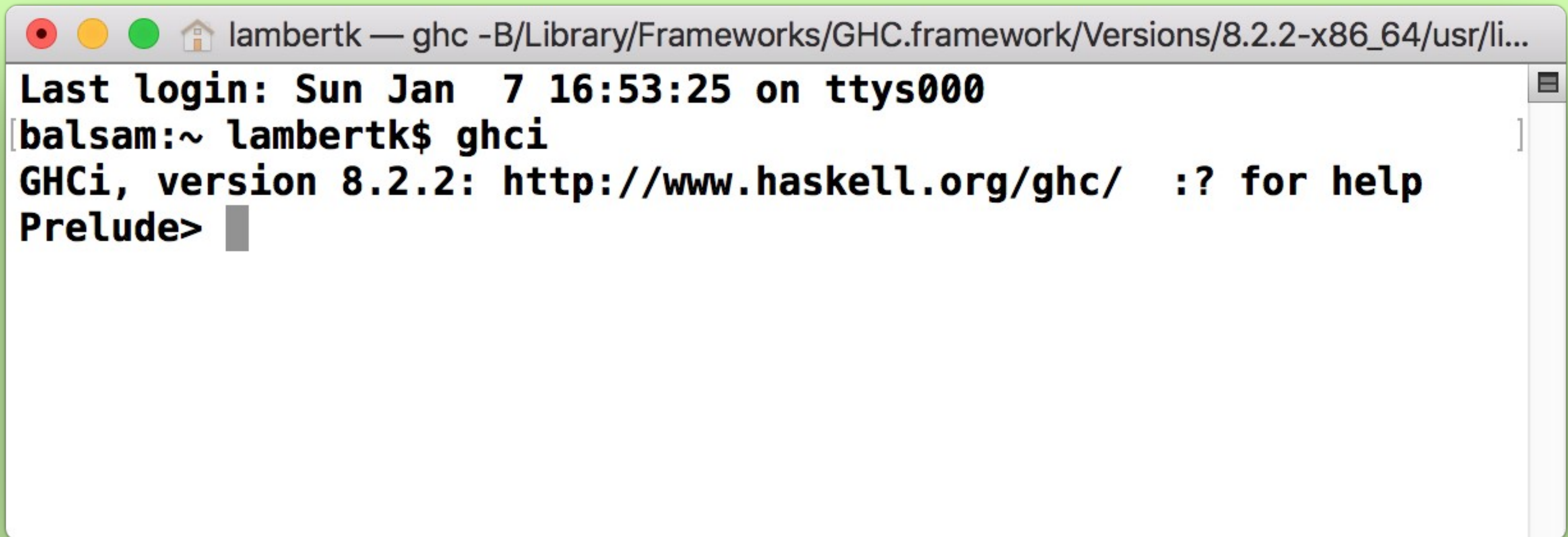
- Use your favorite editor to create programs, or a plugin for an IDE such as Eclipse (Atom and Editra are good)

- Tools include the GHC (Glasgow Haskell Compiler) and the GHCI (Glasgow Haskell Compiler Interactive), as well as many others for creating documentation, libraries, etc.

Haskell Pragmatics

- Expressions can be evaluated interactively in an interpreter (the GHCI)
- Programs consist of modules of function definitions, which must be saved in files, compiled, and imported for use
- Programs can be compiled as standalone apps for release

Firing up the ghci Shell



The Haskell **Prelude** is the default standard module in the GHCI

As does Python's IDLE, the GHCI provides a REPL (read-eval-print loop)

Numbers, Arithmetic, Comparisons

```
Prelude> 53                             -- Syntax: <expression>
53

Prelude> 3.14
3.14

Prelude> 3 + 4 * 2
11

Prelude> 3 == 4
False

Prelude> 3 <= 4
True

Prelude> 3 /= 4
True
```

Numbers and Function Calls

```
Prelude> sqrt 2
1.4142135623730951

Prelude> mod 3 2          --Prefix notation, no parentheses
1

Prelude> abs -2
eError:
• Non type-variable argument in the constraint: Num (a -> a)

Prelude> :type abs
abs :: Num a => a -> a

Prelude> abs (-2)
2

Prelude> 3 / 2
1.5

Prelude> 3 `div` 2
1
```

Special Interpreter Commands

Command	What It Does
`:browse[!] [[*]<mod>]`	Display the names defined by module <mod> (!: more details; *: all top-level names)
`:cd <dir>`	Change directory to <dir>
`:edit <file name>`	Edit <file name>
`:help, :?`	Display all commands
`:load <module>`	Load <module> and its dependents
`:quit`	Quit the GHCi
`:type <expression>`	Display the type of <expression>
`:!<command>`	Run a shell command

For next time

Simple data types

Variables and assignment

Simple function definitions