The primary purpose of the programming project is to provide you an opportunity to demonstrate your ability to write Haskell code that implements a nontrivial application.

Logistics

- You may work alone or with one other person. Each of the suggested projects has a number of stars next to it, indicating how challenging I think it is. In some cases, the challenge is not necessarily the Haskell itself but rather in understanding the problem being implemented.
  
  - If you are working alone, you may choose any project at all.
  - If you are working with someone, you must choose a project with at least two stars.

- You will need to submit both a paper copy and an electronic copy of all Haskell files necessary for running/testing your project.

Your code should be well-commented, including (as warranted) high-level discussion of the Haskell code (i.e., anything not patently obvious) and any additional information necessary to understand how things fit together. It is your responsibility to make sure that I can understand how your code works.

“Well-commented” does not mean that you have to comment every line: it usually suffices (1) to include comments at the beginning of the function/equation, and (2) to use mnemonic names that will be helpful to the reader (i.e., don’t use variables like \(a\), \(b\), \(x\), \(y\) throughout.)

- I will be asking people to explain aspects of their code to me, probably in 30-minute slots (more details later). These time slots may happen before the final version of the code is submitted – that is okay, as long as people can explain what’s working, what’s left to be done, and so on.

- The project is officially due on the last day of class (April 28). However, I will accept projects through Monday, May 7.

- Very important: Abide by the course honor policy, and do not submit code that you didn’t write. Also be sure to include (as part of comments) citations to reference material, as warranted.

Suggested Projects

I am very open to other suggestions for possible projects. If there is something else you’d like to implement, feel free to discuss it with me. However, do not work on an unapproved project, as you may be unhappy with the results.

1. (***) Turing machines provide a simple notion of computation that is used as the basis for most computational-complexity analysis. A Turing machine comprises an unbounded tape, a read/write head, a set of symbols that may be written on the tape, a set of states, and a state-transition function. The details are spelled out more fully in Chapter 31 of [Dew93], or in any textbook on formal languages and automata theory.

   Implement a Turing machine package that supports defining/executing Turing machines.
2. (**) Finite-state machines are somewhat simpler (there’s no tape that need to be written) than Turing machines, but they operate on the same basic principles. Regular expressions can be recognized by finite-state machines.
Implement a finite-state machine package that supports defining/executing finite-state machines. These machines are described briefly in Chapter 7 of [Dew93], or in any textbook on formal languages and automata theory.

3. (*) Implement two dynamic dictionary ADT packages: one using red-black trees and one using self-adjusting binary-search trees (also known as splay trees), both of which are described in Chapter 7 of [LD91].
Specifically, you should implement the following abstract operations: insertions, deletions, and lookups.

4. (⋆ ⋅ ⋆) Implement a vector and matrix package that works over all number types (ie, matrices and vectors of floats, matrices and vectors of ints, and so on). You should include the following operations at a minimum: vector and matrix dot product, vector and matrix addition, matrix multiplication, scalar multiplication on vectors and matrices, matrix transpose, identity matrices, and matrix dimension.

5. (⋆ ⋅ ⋆ ⋆) Implement an evaluator for a simple version of Scheme (using either lazy or eager evaluation). You’ll need to include a parser for the language.
At a minimum, your language should handle nil/(), cons, car, cdr, eq, if, quote, symbols, and lists of such things.

6. (⋆⋆) Do Abelson and Sussman’s Digital Circuit simulator in Haskell. (See sections 3.3.4 and 3.5.3—particularly page 343—in [AS96].)

7. (⋆⋆⋆) Implement the type-checking algorithm described in [Car87], for the language described there.

8. (⋆ ⋆ ⋆) Complete the implementation of the proof calculator described in Chapter 12 of Bird [Bir98]. You’ll probably find it easier to write your own parser using the Hugs ParseLib.hs library, but you can model it on the code he provides.

References


