

## CSci 360, Fall 2004, Project 2

This project, worth 85 points, is due at noon, Thursday, December 9. To submit your solution, hand a paper copy of your report into my office and send the test programs as attachments to an e-mail to [cburch@cburch.com](mailto:cburch@cburch.com).

You've decided to write a magazine article giving benchmark results analyzing whether functional languages can be efficient. After consulting with the editor, you've decided to choose at least one language from each of the following categories:

- a compiled functional language (e.g., Haskell)
- a compiled imperative statement (e.g., C, Ada, Java, or C++)
- an interpreted imperative language (e.g., BASIC, Python, or Perl)<sup>1</sup>

Technically, you won't be comparing *languages* as much as the *compilers and interpreters* used to execute them. You might assume, though, that the writers of the systems would naturally milk as much efficiency as they can given the language, so you would think that any differences are based more on the language's limitations than the compiler/interpreter. (That assumption is somewhat wrong, because popular compilers naturally have a lot more people combing the compiler for potential new, perhaps difficult optimizations.)

The Hugs interpreter we've used in assignments is of course not a compiler. But installed on `grendel` is a Haskell compiler that you can use, `ghc`. The following illustrates its usage.

```
grendel% cat myprogram.hs
module Main where
import System

outputString name = "hello " ++ name ++ "!!"

main = do args <- getArgs
        putStrLn (outputString (head args))
grendel% ghc -O filename.hs
Compilation is NOT required           ignore this statement
grendel% ./a.out you
hello you!!
```

Note that the structure of the program, beginning with the `module` line, followed by a sequence of function definitions, including `main`, the function that will be executed by the program, whose type should be `IO ()`. This particular example also illustrates the `getArgs` function from `ghc`'s `System` module: The `getargs` function returns an `IO [String]` encapsulating all command-line arguments.

To perform the timings, you can use the Unix command `time`.

```
grendel% time ./a.out you
hello you!!

real    0m0.003s
user    0m0.002s
sys     0m0.002s
```

This includes three numbers. The "user" time is the most meaningful, as it represents the amount of time the operating system spent in executing the code of the program. The other numbers aren't important here, but in case you're curious the "real" time is the actual amount of time elapsed while the program was running

---

<sup>1</sup>Yes, I'm aware that you may not already be familiar with a suitable interpreted language. None of the ones that I have listed, though, are all that difficult to learn to the extent necessary for this project. Personally, I'd recommend Python, which you can read about at [www.python.org/doc/](http://www.python.org/doc/).

I don't believe BASIC is currently on the Unix systems, but if you want it, I'll be happy to look into installing it for you.

(including time the operating system spent executing other things), and the “sys” time is the amount of time the operating system spent performing work on behalf of the process. You wouldn’t expect “real” ever to be less than the sum of “user” and “sys”: That it is in the above transcript is just a rounding oddity.

There are several guidelines to consider in constructing your benchmarking procedure.

- Perform benchmarks on a variety of very different problems; one comparison alone isn’t enough. I’d recommend at least numeric computation problem and one string-processing problem. I’m intentionally not suggesting particular problems for you to use to benchmark: I expect you to come up with your own problems (and I expect that they will not match with any classmates’ except by coincidence).
- None of the problems should involve heavy I/O, as the performance of I/O operations is more a function of the hardware and the operating system than of the languages.
- It’s important that your problems be ones that take lots of time — like more than a full second — to complete. Timings of quick programs (like the example above) are not very meaningful, since the system can’t measure short times accurately, and since the bulk of the time will be spent in starting the program rather than in running it.
- When you compile programs for benchmarking, you should tell the compiler to work hard to optimize the output code. Most compilers include a “-O” option for this (as in the *ghc* invocation above).
- Don’t ever trust a single timing. To compute your results, try running the program several times (maybe five), throw out the least and the greatest times, and report based on the average of the rest. Do not report individual times — just the final results.
- Obviously, you should do all your timing on the same machine.
- It’s quite important that you write each program with a careful eye toward implementing it as efficiently as possible. In Haskell, I’ve found that there is a *huge* speedup if you declare your functions’ types, using `Double` or `Int` to deal with numbers.<sup>2</sup> Otherwise, Haskell will compile the functions to deal with general numeric types, forcing a lot of type checking to occur at run-time. You may want to rewrite your program with several minor tweaks to see whether the tweaks help or not.
- It may be that for the same problem, the efficient algorithm for the functional program has to be markedly different from the efficient algorithm for the imperative programs. This is not desirable, but it may be unavoidable for your problems. Since this means that you’re not really comparing apples to apples, you should describe such differences in your article.
- Describe your experimental procedure thoroughly in your article. You should not include the specific code, but you should describe the problems and algorithms implemented, and you should describe your experimental procedure, including the language systems tested, the hardware used, and the process followed to arrive at the numbers you report. (What inputs did you use? How many trials?)

I expect that the article would be two or three single-spaced pages in length, including tables/charts of results, but longer is fine if you can keep it interesting.

Your writing should be good magazine-quality writing: In particular, you can be somewhat informal, if that’s what it takes to keep the article interesting. But by no means should that prevent you from providing information essential to understanding your procedure and results. A convenient technique, if it’s not overused, is to relegate methodological details to footnotes. The article should conclude with an overall assessment of the language systems’ relative speeds.

I will grade the project based on the thoroughness of your results, the efficiency of your programs, the rigor of your testing procedure, the readability of your article, and the soundness of your conclusions.

---

<sup>2</sup>In class, we’ve used the `Integer` class, which is for multiprecision integers. The `Int` class uses 32-bit integers.