UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

**CS 294-8** **Prof. R. Fateman**
**Fall, 2006**

## Assignment 4: Simplify, Memos, Pipes/Streams, Optimization

**Due:** Oct. 16, 2006

I expect that you will get stuck on some of these questions and I hope you will come ask me for help, rather than just spin your wheels. The sooner you start, the LESS time you'll take overall, and the MORE you'll learn.

**1.** Run the `auxfns.lisp + nmacsyma.lisp + macsymar.lisp` code from the class home page (norvig directory) and find some examples where the simplifier, or even the parser, fails. What does it mean to "fail"? A mathematically incorrect answer (not equivalent to the input) is a failure. An expression that we know is really zero, but the simplifier fails to figure it out, is another. This kind of failure is more excusable because in fact zero-equivalence is, strictly speaking, not computable.

**2.** Add the fact that $\cos(n\pi) = (-1)^n$ for integer $n$. How much further can you push the pattern matcher to simplify the cosine of "special arguments"? (Describe what you would have to do: no need to program this).

**3.** Some people have advocated doing arithmetic and simplification on sets. For example, $\sqrt{9}$ is arguably the set $\{3, 3\}$. Think through what difficulties might occur if you tried to introduce this to the simplifier. Consider what role the empty set might play in such a case.

**4.** Look back at the definition of the `permutations` function given by Norvig on page 150 (it is also in file `/norvig/gps.lisp`). Notice that it returns the full list of $n!$ permutations of $n$ items.

Suppose you are trying to find the first (or any) "appropriate" permutation — one that passes some particular test. It would most likely be a waste of time and space to generate the list of $n!$ permutations before testing any of them. The pipe[1] mechanism of section 9.3 provides a way around this problem. Using this technique, write and test a permutations-pipe program, and show how it can be used.

[1] In CS61A we, along with Abelson/Sussman referred to this as a "stream" but since Common Lisp uses the word stream for input/output, Norvig uses the word pipe. This is not a great choice because the UNIX operating system uses the word pipe, and it is not the same, quite.

**5.** Compare the speed of your program, generating a "full set" of the permutations, to the full-generation version. Presumably because of overhead, the pipe version is slower if you compare the generation of *all* the permutations. With any luck, the pipe version is faster if you need only $k$ (for $k < n!$) permutations though. Find some $k$ and $n$ for which this is true.

Since we are playing with comparing speed you will have to compile your code and use appropriate compiler declarations as suggested in section 10.1 and in whatever documentation you have for your Lisp (e.g. the Allegro CL on-line documentation) to get the greatest speed. For suggestions on speeding up programs, see in the on-line manual the topics `about-declarations` and `about-compiler-declarations`.

**6.** While it is a nice abstraction, a direct implementation of this pipe stuff is really not going to be useful if you need lots of permutations in a hurry. For this problem, write a program that when called, burps out a permutation and (in effect) suspends itself; then when next called burps out the "next" permutation, etc. Ideally this program would use no explicit conses at all. It may modify a list or vector in place, or use other hacks. Is this faster than the pipe program? (I wrote one in 20 lines, but it uses conses..)

See the next page for some details and sample timing results.

(If you are stumped at how to approach this, talk to me, or other people in the class or look up the really obscure goto-ful non-recursive programs in various places. If you are familiar with Algol 60, look in the Collected Algorithms of the ACM!)

**7.** Compare and contrast the UNIX operating system notion of a pipe to the notion of section 9.3. You can read the UNIX on-line manual for a concise definition of what its pipe means. (The emacs `M-x man` command will display a page for you). Please don't do this from memory. ("What, you want me to do research AND write a report?" — Calvin, of *Calvin and Hobbes*)

**8.** Memoization works only for "pure" functions that do not refer to any part of the machine "state" other than the arguments to the function. What if you wanted to use memo functions more generally. What techniques might you use for getting around this problem?

**9.** Memoization as given by Norvig works for functions of a single argument. Make a version that works for 2 arguments, or more generally for any fixed number of arguments. There are several approaches. See if you can come up with one that does not require any CONSes in case the answer is already in the hash table.

**10.** A form of "simplification of predicate calculus expressions" similar to what is required in Norvig's exercise 8.7 turns out to be of major interest circuit design.

Use a set of variables plus the operators `and`, `or` and `not` as well as the Boolean constants `true` and `false`.

Show that it is possible to do a really crummy job of this with a few rules, and a more careful one with a few more rules.

A carefully designed program that solves the simplification optimally (according to some criteria of complexity of circuits) has earned people Ph.Ds and/or significant income.

Speculate on what kinds of approaches might work better than rules.

A really good job researching the problem, discovering and implementing appropriate data structures, heuristics, etc. could be a term project or more.

The permutations contest

We'll put a challenger from the past (in object-code only form) on our class web page for comparison; an Allegro fasl file. We can compile it on other systems if you have a favorite.

To be a competitor you must define a function (permuter '(a b c d)) which returns a FUNCTION that when called repeatedly with no arguments returns a permutation of permuter's input. You can set it up so that the function always returns a vector, or what might be nicer, a version that returns a vector if it was given a vector as (permuter (vector 'a 'b 'c) ) or a list if it was given a list.

To try out your function, consider

```
(defun ttest (y n)(let((r (permuter y))) ;;silent one for timing
    (dotimes (i n) (funcall r))))

(defun test (y n)(let((r (permuter y))) ;;noisy one for debugging
   (dotimes (i n) (print (funcall r)))))
```

So you can test your function (or mine) by typing

(test '(a b c) 7) and you can time it by (time (ttest '(a b c) 7)).