This exam is worth 105 points. We will assign partial credit to partial responses, provided we can understand the response and it reflects a correct understanding of course material.

Exam rules:

- Your responses to this exam are due by Tuesday 2003-11-25 at 1:45pm sharp.
- On turning in solutions:
  - Solutions may be hand-written, but the burden of legibility rests on you.
  - If possible, please answer the questions on the exam sheets, and within the space provided. If you do use separate sheets of paper, be sure to make clear where your work on a problem begins and ends.
  - Staple your solutions together. *Put your name on every page.*
- Do not assume that the length of an answer should be proportional to the number of points assigned. Even for weighty problems, the correct solution can take just a few words, especially if you use the right technical phrases! No problem on this exam asks for, or even really wants, a lengthy essay as a solution.
- If you believe a question is underspecified, make a reasonable assumption and document it.
- The exam is open book with respect to the texts listed on the web page. You may not refer to any other sources related to the course between the time you start and finish the exam.
- All Scheme code responses must use the dialects employed in this course.
- You may not evaluate any programs related to this exam on a computer.
- You must neither give assistance to any other course participant, nor receive assistance from anyone other than the instructor, on material under examination.

Finally, thanks to Shriram K. for lending material and Greg C. for proofreading this exam.
Problem 1  [10 points]

The program

```latex
{with \{ x 4 \}
  {with \{ f \{fun \{ y \}
     (+ x y)\}}
  {with \{ x 5 \}
    \{f 10\}\}}
```

should evaluate to 14 by static scoping. Evaluating \( x \) in the environment at the point of invoking \( f \), however, yields a value of 15 for the program. Ben Bitdiddle, a sharp if eccentric student, points out that we can still use the dynamic environment, so long as we take the *oldest* value of \( x \) in the environment rather than the *newest*—and for this example, he’s right!

Is Ben right in general? If so, justify. If not, provide a counterexample program and explain why Ben’s evaluation strategy would produce the wrong answer. (A bonus point for explaining why Ben’s way of thinking about environments is conceptually wrong, irrespective of whether or not it produces the right answer.)
Problem 2  [10 points]

Is C++ eager or lazy? Write a C++ program to determine the answer to this question. The same program, run under the two different regimes, should produce different results. You may use any C++ features you want, but keep your program relatively short; we will penalize you for programs we consider excessively long or obfuscatory. (Tip: It’s possible to solve this problem with a program no more than a few dozen lines long.)

You must turn in both the source code to your program (in printed or written form) as well as an answer to the question of whether C++ is eager or lazy, and an explanation of how your program determines this. That is, you should provide a brief and unambiguous answer (e.g., “C++ is lazy”) followed by a description of what result would obtain under each regime, along with a brief explanation of why that regime would generate that result.
Problem 3  [20 points]

We have discussed how the definition of substitution results in an inefficient operator: in the worst case, it can take time at least quadratic in the size of the program (where we can define the program size as the number of nodes in the abstract syntax tree). We talked about delaying substitution using a cache. However, implementing the cache using a stack doesn’t seem very much more efficient.

Answer the following two questions.

1. Provide a schema for a program that illustrates the non-linearity of the stack-based cache implementation. Explain briefly why its execution time is non-linear in its size.

2. Describe a data structure for a substitution cache that an environment-based interpreter can use to improve its complexity, and show how the interpreter should use it (if the interpreter must change to accommodate your data structure, describe these changes by providing a pseudocode version of the new interpreter). State the new complexity of the interpreter, and (informally but rigorously) prove it. You don’t need to restrict yourself to the subset of Scheme we are using in this course you may employ all your knowledge of C++ or Java, say. However, the responsibility for providing a clear enough description lies on you; remember that simple code is often the clearest form of description.
Problem 4  [10 points]

No lazy language in history has also had state operations (such as mutating the values in boxes, or assigning values to variables). Why not? The best answer to this question would include two things: a short program (which we assume will evaluate in a lazy regime) that uses state, and a brief explanation of what problem the execution of this program illustrates. If you present a sufficiently illustrative example (which needn’t be very long!), your explanation can be quite short.
Problem 5  [20 points]

In class, we added recursive functions to our language by introducing rec expressions of the form \{rec \{f e\} body\}; for example:

\{rec \{fact \{fun \{n\} \n  \{bif \{iszero n\} \n    1 \n    \{* n \{fact \{+ n -1\}\}\}\}\} \n\{fact 10\}\}

We then extended the environment-based interpreter to support rec expressions; upon encountering a recursive function, the interpreter created a cyclic environment.

An alternative design for adding recursive functions is to introduce make-rec expressions of the form \{make-rec \{f e\}\}; the factorial example would be:

\{with \{fact \{make-rec \{fun \{n\} \n  \{bif \{iszero n\} \n    1 \n    \{* n \{f \{+ n -1\}\}\}\}\}\} \n\{fact 10\}\}

A make-rec expression creates a recursive function and returns it immediately. In other words:

\{make-rec \{f e\}\} is equivalent to \{rec \{f e\} \{f\}\}

Your task: extend the original substitution interpreter (fun-subst.ss, which includes numbers and functions) to support make-rec expressions. Make sure to update the datatype, parser, and interpreter; however, you may elide those sections of code that remain the same.
Problem 6  [20 points]

We studied mutation through boxes, which is related to but different from the notion of mutation found in most programming languages, such as C and Java. In those languages, one mutates a variable directly, rather than a box bound to an identifier: for instance, a programmer can write

\[ x = 4; \]

To model this operation, we define the language that has functions, arithmetic, sequencing, and one new operator, \( := \). Its grammar is as follows:

\[
<\text{expr}> ::= <\text{id}>
| <\text{number}>
| (+ <\text{expr} <\text{expr}>)
| \{\text{fun} <\text{id}> <\text{expr}>\}
| \{<\text{expr} <\text{expr}>\}
| \{\text{seqn} <\text{expr} <\text{expr}>\}
| \{:= <\text{id} <\text{expr}>\}
\]

The semantics of \( := \) is to first evaluate the sub-expression, then assign its value to the named variable, and finally return the sub-expression’s value as that of the entire expression. We will assume that the parser converts instances of \texttt{with} into immediate function definition and application. Thus, for instance, the program

\[
\{\text{with} \{x 3\}
| (+ \{:= x 4\}
| x)\}
\]

evaluates to 8 (the assignment changes \( x \) to 4 and returns that value; addition operates left-to-right; looking up \( x \) yields its new value, 4, for a sum of 8), while

\[
\{\text{with} \{x 2\}
| (+ \{\text{with} \{x 3\}
| \{:= x 4\})
| x)\}
\]

evaluates to 6, since the mutation affects only the inner \( x \).

Let’s implement this model. As usual, we want to avoid writing an interpreter that uses too many Scheme features, so we’ll use a purely functional interpreter. Given below is the skeleton of an interpreter written in store-passing style. You will need to complete the definition of \texttt{interp}, and provide implementations for any auxiliary procedures you introduce. On the next two pages, we have given you an interpreter based on that from the lecture on implementing boxes; you need not provide implementations of procedures already in the notes (such as \texttt{next-location})—make sure you don’t change any of them! You shouldn’t need to modify any code we’ve given, but if you want to, you are welcome to do so.

Hint: You will want the environment to map names to locations, and the store to map locations to values. In the absence of \texttt{newbox}, what should allocate a new store location?

(continued on next page)
(define-datatype expr expr?
[\textit{varE}\ (\textit{id}\ symbol?)\n[\textit{numE}\ (n\ number?)\n[\textit{plusE}\ (le\ expr?)\ (re\ expr?)\n[\textit{funE}\ (\textit{param}\ symbol?)\ (\textit{body}\ expr?)\n[\textit{appE}\ (\textit{fun-expr}\ expr?)\ (\textit{arg-expr}\ expr?)\n[\textit{seqnE}\ (expr1\ expr?)\ (expr2\ expr?)\n[\textit{asgnE}\ (\textit{var}\ symbol?)\ (e\ expr?)\n]
)
)

(define-datatype \textit{Env} \textit{Env}?
[\textit{mtSub}\n[\textit{aSub}\ (\textit{id}\ symbol?)\n (\textit{value}\ number?)\n (rest-env\ \textit{Env}?)\n]
)

(define-datatype \textit{value} \textit{value}?
[\textit{numV}\ (n\ number?)\n[\textit{closureV}\ (\textit{param}\ symbol?)\n (\textit{body}\ expr?)\n (\textit{env}\ \textit{Env}?)\n]
)

(define-datatype \textit{Store} \textit{Store}?
[\textit{mtSto}\n[\textit{aSto}\ (location\ number?)\n (\textit{value}\ value?)\n (\textit{store}\ \textit{Store}?)\n]
)

(define-datatype Value Store Value Store?\n[v-s\ (value\ value?)\ (store\ \textit{Store}?)\n]

;; env-lookup : symbol \textit{Env} \to location
(\textbf{define}\ \textit{(env-lookup} name\ \textit{an-env})\n(\textbf{cases}\ \textit{Env}\ an-env\n [\textit{mtSub} () \textit{(error} \textit{env-lookup} \textit{(string-append \textit{unbound}: }\textit{symbol} \textit{string name)}\n)\n [\textit{aSub} (bound-name\ bound-value\ rest-env)\n (\textbf{cond}\n [(\textit{symbol=\ bound-name name}) bound-value]\n [\textit{else} \textit{(env-lookup name}\ rest-env)])\n))\n
;; store-lookup : location \textit{Store} \to value
(\textbf{define}\ \textit{(store-lookup} location\ \textit{a-store})\n(\textbf{cases}\ \textit{Store}\ a-store\n [\textit{mtSto} () \textit{(error} \textit{store-lookup} \textit{(string-append \textit{invalid store location: } number \textit{string location)})\n)\n [\textit{aSto} (filled-location\ filled-value\ rest-store)\n (\textbf{cond}\n [(\textit{= location}\ filled-location) filled-value]\n [\textit{else} \textit{(store-lookup location}\ rest-store)])\n))\n
(continued on next page)
:: interp : expr Env Store → Value*Store

(define (interp exp env store)
  (cases expr exp
    [numE (n) (v-s (numV n) store)]
    [plusE (le re)
      (cases Value*Store (interp le env store)
        [v-s (lhs-val lhs-sto)
          (cases Value*Store (interp re env lhs-sto)
            [v-s (rhs-val rhs-sto)
              (v-s (numV+ lhs-val rhs-val) rhs-sto)])])]
    [varE (id) . . ]
    [funE (param body)
      (v-s (closureV param body env)
        store)]
    [appE (fun-expr arg-expr)
      (cases Value*Store (interp fun-expr env store)
        [v-s (fun-val fun-sto)
          (cases Value*Store (interp arg-expr env fun-sto)
            [v-s (arg-val arg-sto)
              (cases value fun-val
                [closureV (cl-param cl-body cl-env)
                  [else (error "interp can only apply functions")]])])]
    ]
    [seqnE (e1 e2)
      (cases Value*Store (interp e1 env store)
        [v-s (e1-val e1-sto)
          (interp e2 env e1-sto)])]
    [asgnE (id expr)
      . . ]))
Problem 7  [15 points]

CPS the following Scheme function. You don’t need to CPS primitives such as empty?, first, rest, cons, and <. You may also assume that the function argument to the function is in CPS.

\[
\text{filter} : (X \rightarrow \text{boolean}) \rightarrow (\text{list of } X) \rightarrow (\text{list of } X)
\]

\[
\text{(define \ \text{filter} \ f \ l)}
\]

\[
\text{(cond}
\]

\[
[\text{empty? } l \ \text{empty}]
\]

\[
[\text{else (cond}
\]

\[
[\text{f (first } l) \ \text{(cons (first } l)
\]

\[
(\text{filter } f \ (\text{rest } l))])
\]

\[
[\text{else (filter } f \ (\text{rest } l))])]
\]

Now change the following function application expressions to use their corresponding CPSed versions.

\[
(\text{filter (lambda (x) (}} < x \ 3))
\]

\[
(\text{cons 1 (cons 4 empty)) ;; this evaluates to (list 1)}
\]