Homework 1

Due 8 October 2002

Reading

1. Read Chapters 1–3 in the course reader. Chapter 1 is on the Cambridge University Press site at http://books.cambridge.org/0521780985.htm or reachable from http://www.stanford.edu/~jcm.
2. *(Optional)* J. McCarthy, Recursive functions of symbolic expressions and their computation by machine, *Comm. ACM* 3,4 (1960) 184–195. You can find a link to this on the CS242 web site. The most relevant sections are 1, 2 and 4; you can also skim the other sections if you like.

Problems

1. Partial and Total Functions

For each of the following function definitions, give the graph of the function. Say whether this is a partial function or a total function on the integers. If the function is partial, say where the function is defined and undefined.

For example, the graph of $\mathtt{f(x) = if\ x > 0\ then\ x + 2\ else\ x/0}$ is the set of ordered pairs $\{\langle x, x+2\rangle \mid x > 0\}$. This is a partial function. It is defined on all integers greater than 0 and undefined on integers less than or equal to 0.

Functions:

(a) $\mathtt{f(x) = if\ x + 2 > 3\ then\ x * 5\ else\ x/0}$

(b) $\mathtt{f(x) = if\ x < 0\ then\ 1\ else\ f(x - 2)}$

(c) $\mathtt{f(x) = if\ x = 0\ then\ 1\ else\ f(x - 2)}$

2. Existential Halting Problem

Suppose you are given a function $\mathsf{Halt}_\exists$ that, given a program P as input, will determine whether or not there exists an input n for which $P(n)$ halts. To make this concrete, assume that you are writing a C or Pascal program that reads in another program as a string. Your program is allowed to call $\mathsf{Halt}_\exists$ with a string input. Assume that the call to $\mathsf{Halt}_\exists$ returns true if the argument is a program that will halt on some unspecified input, and returns false if the argument is a program that runs forever on all inputs. If $\mathsf{Halt}_\exists$ returns true, all you know is that there exists an input for which P halts; it does not return the specific *value* of that input. You should not make any assumptions about the behavior of $\mathsf{Halt}_\exists$ on an argument that is not a syntactically correct program.

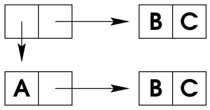
Can you solve the halting problem using $\mathsf{Halt}_\exists$? More specifically, can you write a program that reads a program text P as input, reads an integer n as input, and then decides whether $P(n)$ halts? You may assume that any program P you are given begins with a read statement that reads a single integer from standard input. This problem does not ask you to write the program to solve the halting problem. It just asks whether it is possible to do so.

If you believe that the halting problem can be solved if you are given $\mathsf{Halt}_\exists$, then explain your answer by describing how a program solving the halting problem would work. If you believe that the halting problem cannot be solved using $\mathsf{Halt}_\exists$, then explain briefly why you think not.

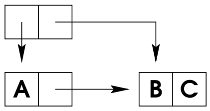
3. .... Cons Cell Representations

(a) Draw the list structure created by evaluating `(cons 'A (cons 'B 'C))`.

(b) Write a Pure Lisp expression that will result in this representation, with no sharing of the `(B . C)` cell. Explain why your expression produces this structure.



(c) Write a Pure Lisp expression that will result in this representation, with sharing of the `(B . C)` cell. Explain why your expression produces this structure.



While writing your expressions, use only these Lisp constructs: `lambda` abstraction, function application, the atoms `'A 'B 'C`, and the basic list functions (`cons, car, cdr, atom, eq`). Assume a simple-minded Lisp implementation that does not try to do any clever detection of common subexpressions or advanced memory allocation optimizations.

4. .... Conditional Expressions in Lisp

The semantics of the Lisp conditional expression

$$(\texttt{cond}\ (p_1\ e_1)...(p_n\ e_n))$$

is explained in the text. This expression does not have a value if $p_1, \ldots, p_k$ are false and p_{k+1} does not have a value, regardless of the values of $p_{k+2}, \ldots, p_n$.

Imagine you are an MIT student in 1958 and you and McCarthy are considering alternative interpretations for conditionals in Lisp.

(a) Suppose McCarthy suggests that the value of $(\texttt{cond}\ (p_1\ e_1)...(p_n\ e_n))$ should be the value is e_k if p_k is true and if, for every $i < k$, the value of expression p_i is either false or undefined. Is it possible to implement this interpretation. Why or why not? (*Hint:* Remember the halting problem.)

(b) Another design for conditional might allow any of several values if more than one of the guards $(p_1, \ldots, p_n)$ is true. More specifically (and be sure to read carefully), suppose someone suggest the following meaning for conditional:

   i. The conditional's value is undefined if none of the p_k are true.

   ii. If some p_k is true, then the implementation *must* return the value of e_j for *some* j with p_j true. However, it need not be the first such e_j.

   Notice that in $(\texttt{cond}\ (a\ b)\ (c\ d)\ (e\ f))$, for example, if `a` runs forever, `c` evaluates to true, and `e` halts in error, the value of this expression should be the value of `d`, if it has one. Briefly describe a way to implement conditional so that properties [i] and [ii] are true. You only need to write two or three sentences to explain the main idea.

(c) Under the original interpretation, the function

```
(defun odd (x) (cond ((eq x 0) nil)
                     ((eq x 1) t)
                     ((> x 0) (odd (- x 2)))
                     (t (odd (+ x 2)))))
```

   would give us `t` for odd numbers and `nil` for even numbers. Modify this expression so that it would always give us `t` for odd numbers and `nil` for even numbers under the alternate interpretation described in part (b).

(d) The normal implementation of boolean `OR` is designed not to evaluate a sub-expression unless it is necesary. This is called the "short-circuiting `OR`", and it may be defined as follows:

$$Scor(e_1, e_2) = \begin{cases} \texttt{true} & \text{if } e_1 = \texttt{true} \\ \texttt{true} & \text{if } e_1 = \texttt{false} \text{ and } e_2 = \texttt{true} \\ \texttt{false} & \text{if } e_1 = e_2 = \texttt{false} \\ \textit{undefined} & \text{otherwise} \end{cases}$$

It allows for e_2 to be undefined if e_1 is true.

The "parallel `OR`" is a related construct which gives an answer whenever possible (possibly doing some unnecessary sub-expression evaluation). It is defined similarly:

$$Por(e_1, e_2) = \begin{cases} \texttt{true} & \text{if } e_1 = \texttt{true} \\ \texttt{true} & \text{if } e_2 = \texttt{true} \\ \texttt{false} & \text{if } e_1 = e_2 = \texttt{false} \\ \textit{undefined} & \text{otherwise} \end{cases}$$

It allows for e_2 to be undefined if e_1 is true, and also allows e_1 to be undefined if e_2 is true. You may assume that e_1 and e_2 do not have side-effects.

Of the original interpretation, the interpretation in part (a), and the interpretation in part (b), which ones would allow us to implement *Scor* most easily? What about *Por*? Which interpretation would make implementations of "short-circuiting `OR`" difficult? Which interpretations would make implementation of "parallel `OR`" difficult? Why?

5. .... Lisp and higher-order functions

Lisp functions `compose`, `mapcar`, and `maplist` are defined as follows, writing `#t` for *true* and `()` for the empty list. Text beginning with ;; and continuing to the end of a line is a comment.

```
(define  compose
     (lambda (f  g)   (lambda (x)  (f  (g   x)))))

(define mapcar
  (lambda (f xs)
    (cond
      ((eq? xs ()) ()) ;; If the list is empty, return the empty list
      (#t             ;; Otherwise, apply f to the first element ...
          (cons (f (car xs))
                        ;; and map f on the rest of the list
            (mapcar f (cdr xs))
)))))

(define maplist
  (lambda (f xs)
    (cond
      ((eq? xs ()) ()) ;; If the list is empty, return the empty list
      (#t             ;; Otherwise, apply f to the list ...
          (cons (f xs)
                        ;; and map f on the rest of the list
              (maplist f (cdr xs))
)))))
```

The difference between `maplist` and `mapcar` is that `maplist` applies `f` to every sublist, while `mapcar` applies `f` to every element. (The two function expressions differ only in the 6th line.) For example, if `inc` is a function that adds one to any number, then

```
(mapcar inc  '(1 2 3 4)) = (2 3 4 5)
```

while

```
(maplist (lambda (xs) (mapcar inc xs))  '(1 2 3 4))
= ((2 3 4 5) (3 4 5) (4 5) (5))
```

However, you can almost get mapcar from maplist by composing with the car function. In particular, notice that

```
(mapcar f  '(1 2 3 4))
= ((f (car (1 2 3 4))) (f (car (2 3 4))) (f (car (3 4))) (f (car (4))))
```

Write a version of compose that lets us define mapcar from maplist. More specifically, write a definition of compose2 so that

```
((compose2 maplist car) f xs)  =  (mapcar f xs)
```

for any function f and list xs.

(a) Fill in the missing code in the following definition. The correct answer is short and fits here easily. You may also want to answer parts (b) and (c) first.

```
(define  compose2

    (lambda (g  h)

        (lambda  (f  xs)

             (g   (lambda  (xs)  ( ________________ ))  xs )

)))
```

(b) When (compose2 maplist car) is evaluated, the result is a function defined by (lambda (f xs) (g ...)) above, with

which function replacing g?

and which function replacing h?

(c) We could also write the subexpression (lambda (xs) (...)) as (compose (...) (...)) for two functions. Which two functions are these? (Write them in the right order.)

6. .... Regions and Memory Management

There are a wide variety of algorithms to choose from when implementing garbage collection for a specific language. In this problem, we examine one algorithm for finding garbage in Pure Lisp (Lisp without side effects) based on the concept of *regions*. Generally speaking, a region is a section of the program text. To make things simple, we will consider each function as a separate region. Region-based collection reclaims garbage each time program execution leaves a region. Since we are treating functions as regions in this problem, our version of region-based collection will try to find garbage each time a program returns from a function call.

(a) Here is a simple idea for region-based garbage collection:

> When a function exits, free all the memory that was allocated during execution of the function.

However, this is not correct since some memory locations that are freed may still be accessible to the program. Explain the flaw by describing a program that could possibly access a previously freed piece of memory. You don't need to write more than four or five sentences; just explain the aspects of an example program that are relevant to the question.

(b) Fix the method in part (a) to work correctly. It is not necessary for your method to find all garbage, but the locations that are freed should really be garbage. Your answer should be in the form:

When a function exits, free all memory allocated by the function except ...

Justify your answer. (Hint: Your statement should not be more than a sentence or two. Your justification should be a short paragraph.)

(c) Now assume that you have a correctly functioning region-based garbage collector. Does your region-based collector have any advantages or disadvantages over a simple mark-and-sweep collector?

(d) Could a region-based collector like the one described in this problem work for Impure Lisp? If you think the problem is more complicated for Impure Lisp, briefly explain why. You may consider the problem for C instead of Impure Lisp if you like, but do not give an answer that depends on specific properties of C like pointer arithmetic. The point of this question is to explore the relationship between side effects and a simple form of region-based collection.

7. .... Concurrency in Lisp

The concept of *future* was popularized by R. Halstead's work on the language Multilisp for concurrent Lisp programming. Operationally, a future consists of a location in memory (part of a cons cell) and a process that is intended to place a value in this location at some time "in the future." More specifically, the evaluation of `(future e)` proceeds as follows:

i. The location ℓ that will contain the value of `(future e)` is identified (if the value is going to go into an existing cons cell) or created if needed.
ii. A process is created to evaluate `e`.
iii. When the process evaluating `e` completes, the value of `e` is placed in the location ℓ.
iv. The process that invoked `(future e)` continues in parallel with the new process. If the originating process tries to read the contents of location ℓ while it is still empty, then the process blocks until the location has been filled with the value of `e`.

Other than this construct, all other operations in this problem are defined as in Pure Lisp. For example, if expression `e` evaluates to the list `(1 2 3)`, then the expression

```
(cons 'a (future e))
```

produces a list whose first element if the atom `'a` and whose tail becomes `(1 2 3)` when the process evaluating `e` terminates. The value of the `future` construct is that the program can operate on the `car` of this list while the value of the `cdr` is being computed in parallel. However, if the program tries to examine the `cdr` of the list before the value has been placed in the empty location, then the computation will block (wait) until the data is available.

(a) Assuming an unbounded number of processors, how much time would you expect the evaluation of the following `fib` function to take, on positive integer argument `n`?

```
(defun fib (n)
       (cond ((eq n 0) 1)
             ((eq n 1) 1)
             (T (plus (future (fib (minus n 1)))
                      (future (fib (minus n 2)))))))
```

We are only interested in time up to a multiplicative constant; you may use "big Oh" notation if you wish. If two instructions are done at the same time by two processors, count that as one unit of time.

(b) At first glance, we might expect that two expressions

```
( ...e ...)
( ...(future e) ...)
```

which differ only because an occurrence of a subexpression `e` is replaced by `(future e)`, would be equivalent. However, there are some circumstances when the result of evaluating one might differ from the other. More specifically, side effects may cause problems. To demonstrate this, write an expression of the form `(...e ...)` so that when the `e` is changed to `(future e)`, the expression's value or behavior might be different because of side effects, and explain why. Do not be concerned with the efficiency of either computation or the degree of parallelism.

(c) Side effects are not the only cause for different evaluation results. Write a Pure Lisp expression of the form `(...e′ ...)` so that when the `e′` is changed to `(future e′)`, the expression's value or behavior might be different, and explain why.

(d) Suppose you are part of a language design team that has adopted futures as an approach to concurrency. The head of your team suggests an error handling feature called a "try block." The syntactic form of a try block is

```
(try e
     (error-1 handler-1)
     (error-2 handler-2)
     ...
     (error-n handler-n))
```

This construct would have the following characteristics:

i. Errors are programmer defined, and occur when an expression of the form `(raise error-i)` is evaluated inside `e`, the main expression of the try block.
ii. If no errors occur, then `(try e (error-1 handler-1) ...)` is equivalent to `e`.
iii. If the error named `error-i` occurs during the evaluation of `e`, the rest of the computation of `e` is aborted, the expression `handler-i` is evaluated, and this becomes the value of the try block.

The other members of the team think this is a great idea and, claiming that it is a completely straightforward construct, ask you to go ahead and implement it. You think the construct might raise some tricky issues. Name two problems or important interactions between error handling and concurrency that you think need to be considered. Give short code examples or sketches to illustrate your point(s). (*Note:* You are not asked to solve any problems associated with futures and try blocks, just identify the issues.) Assume for this part that we are using Pure Lisp (no side effects).