Homework 1

Due 6 October

Handout 1
CS242: Autumn 2003
29 September

Reading

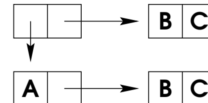
1. Read Chapters 1–3 in the textbook.
2. *(Optional)* J. McCarthy, Recursive functions of symbolic expressions and their computation by machine, *Comm. ACM* 3,4 (1960) 184–195. You can find a link to this on the CS242 web site. The most relevant sections are 1, 2 and 4; you can also skim the other sections if you like.

Problems

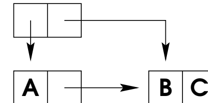
1. .... Cons Cell Representations

(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.

2. .... 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 (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?

3. .... Detecting Errors

Evaluation of a Lisp expression can either terminate normally (and return a value), terminate abnormally with an error, or run forever. Some examples of expressions that terminate with an error are `(/ 3 0)`, division by 0; `(car 'a)`, taking the car of an atom; and `(+ 3 ``a'')`, adding a string to a number. The Lisp system detects these errors, terminates evaluation, and prints a message to the screen. Your boss wants to handle errors in Lisp programs without terminating the computation, but doesn't know how, so your boss asks you to ...

(a) ... implement a Lisp construct `(error? E)` that detects whether an expression E will cause an error. More specifically, your boss wants evaluation of `(error? E)` to halt with value *true* if evaluation of E terminates in error and halt with value *false* otherwise. Explain why it is not possible to implement the error? construct as part of the Lisp environment.

(b) ... implement a Lisp construct `(guarded E)` that either executes E and returns its value, or if E would halt with an error, returns 0 without performing any side effects. This could be used to try to evaluate E and if an error would occur, just use 0 instead. For example,

```
(+ (guarded E) E')    ; just E' if E halts with an error; E+E' otherwise
```

will have the value of E' if evaluation of E would halt in error, and the value of $E + E'$ otherwise. How might you implement the guarded construct? What difficulties might you encounter? Notice that unlike (error? E), evaluation of (guarded E) does not need to halt if evaluation of E does not halt.

4. .... 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.)

5. .... Reference Counting

This question is about a possible implementation of garbage collection for Lisp. Both impure and Pure Lisp have lambda abstraction, function application, and elementary functions `atom`, `eq`, `car`, `cdr`, and `cons`. Impure Lisp also has `rplaca`, `rplacd` and other functions that have side-effects on memory cells.

Reference counting is a simple garbage collection scheme that associates a reference count with each datum in memory. When memory is allocated, the associated reference count is set to 0. When a pointer is set to point to a location, the count for that location is incremented. If a pointer to a location is reset or destroyed, the count for the location is decremented. Consequently, the reference count always tells how many pointers there are to a given datum. When a count reaches 0, the datum is considered garbage and returned to the free-storage list. For example, after evaluation of `(cdr (cons (cons 'A 'B) (cons 'C 'D)))`, the cell created for `(cons 'A 'B)` is garbage, but the cell for `(cons 'C 'D)` is not.

(a) Describe how reference counting could be used for garbage collection in evaluating the expression:

```
(car (cdr (cons (cons a b) (cons c d))))
```

where `a, b, c, d` are previously defined names for cells. Assume that the reference counts for `a, b, c, d` are initially set to some numbers greater than 0, so that these do not become garbage. Assume that the result of the entire expression is not garbage. How many of the three `cons` cells generated by the evaluation of this expression can be returned to the free-storage list?

(b) The "impure" Lisp function `rplaca` takes as arguments a cons cell c and a value v and modifies c's address field to point to v. Note that this operation does *not* produce a new cons cell; it modifies the one it receives as an argument. The function `rplacd` performs the same function with respect the decrement portion of its argument cons cell.

Lisp programs using `rplaca` or `rplacd` may create memory structures that cannot be garbage collected properly by reference counting. Describe a configuration of `cons` cells that can be created using operations of Pure Lisp and `rplaca` and `rplacd`. Explain why the reference counting algorithm deos not work properly on this structure.

(c) Think of another context (Hint: e.g. file system) in which reference counting is used to collect unused resources. Describe how the problem you discovered in the previous question is avoided or solved in this context.

6. .... 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.