Homework 1

Due 8 October

Handout 1
CS242: Autumn 2003
1 October

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. .... 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) $\mathtt{f(x) = if\ x * 5 > 40\ then\ x * x\ else\ x + 1/0}$

(b) $\mathtt{f(x) = if\ x > 1\ then\ 1\ else\ f(x + 1)}$

(c) $\mathtt{f(x) = if\ x \leq 1\ then\ log(x)\ else\ f(x + 2)}$

2. .... Deciding Simple Properties of Programs

Suppose you are given a function Halt_{42} that, given the source code of a function f and a number n, will determine whether or not the call $f(n)$ returns the number 42. To be more precise, assume that you are writing a C or Java program that reads in another program as a string. Your program is allowed to call Halt_{42} with a pair of string inputs. Assume that the call to Halt_{42} returns true if the arguments are a program P and an input n such that $P(n)$ returns 42, and returns false if the arguments are a program P and an input n such that $P(n)$ does not return 42. You should not make any assumptions about the behavior of Halt_{42} on arguments that do not consist of a syntactically correct program and some input to that program.

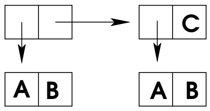
Can you solve the halting problem using Halt_{42}? More specifically, can you write a program that reads a program text P as input, reads an integer n as input, and then decides whether $P(n)$ halts? You may assume that any program P you are given begins with a read statement that reads a single integer 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 Halt_{42}, 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 Halt_{42}, then explain briefly why you think not.

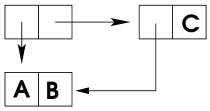
3. .... Cons Cell Representations

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

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



(c) Write a Pure Lisp expression that will result in this representation, with sharing of the `(A . B)` 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. .... Self-application and recursion

As noted in the text, McCarthy's 1960 paper comments that the lambda notation alone is inadequate for defining recursive functions. This is not correct, but the method for expressing recursion using only lambda is tricky enough that we can't fault McCarthy for not figuring it out. This problem asks you to explain a clever way of writing factorial without using anything that looks like a recursive declaration, and then asks you to figure out how to write the Fibonacci function in the same way.

(a) This expression computes 6! by applying a clever factorial function to the number 6:

```
((lambda (n)
   ((lambda (fact)
      (fact fact n))
    (lambda (ft k)
      (if (= k 0)
          1
          (* k (ft ft (- k 1)))))))
 6)
```

This uses the Scheme conditional expression `(if test-exp then-exp else-exp)`. If the value of the `test-exp` is true then the `then-exp` is returned, else the `else-exp`. Explain why this factorial function works. Your explanation should include some English sentences and some symbolic calculation in the style of β-reduction. If you are confused and want to use Google, try "Y-combinator" or "fixed-point operator".

(b) Devise an analogous expression for computing the 10th Fibonacci number.

6. .... 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.

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.