Design and Implementation of Programming Languages

Prof. Mehran S. Fallah

Assignment #3

Lisp Programming Language / Garbage Collection / Higher-Order Functions / Concurrency Concerns in Lisp

Done with **love** by

Ali Gholami

with Stu. #

9531504

Department of Computer Engineering and Information Technology

Amirkabir University of Technology

November 2017

The following are problems from the book, **Concepts in Programming Languages, John Mitchell**.

3.1 **Cons Cells Representation**

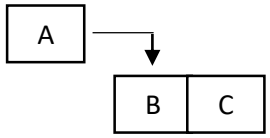
(a) Draw the list structure created by evaluating (cons 'A (cons 'B 'C)).

(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.

Answer. (a)



Answer. (b)

$$(cons\ (cons\ 'A\ (cons\ 'B\ 'C)\ )\ (cons\ 'B\ 'C)\ )$$

Answer. (c)

$$(define\ BC\ (cons\ 'B\ 'C))$$

$$(cons\ (cons\ 'A\ BC)\ BC)$$

3.2 **Conditional Expressions in Lisp**

The semantics of the Lisp conditional expression

$$(cond\ (P_1 E_1) \ldots (P_N E_N))$$

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 $(cond\ (P_1 E_1) \ldots (P_N E_N))$ should be the value of 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?

Answer. (a)

That's not true. Assuming the value of $(cond\ (P_1 E_1) \ldots (P_N E_N))$ will be e_k, if the p_k is true. In this case, all of the conditions before p_k are intended to be either undefined

or false. There is an urge on the $evaluation$ of the previous conditions. So there might be some unlimited loops for the $abstract\ machine$ which is defining the above language.

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

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

$ii.$ If some p_k are 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.

Note 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.

Answer. (b)

The conditions can be evaluated in parallel if the evaluation of e_k is not going to take too long(which will be confused with a simple unlimited loop.)

(c) Under the original interpretation, the function

$$(defun\ odd\ (x)(cond\ ((eq\ x\ 0)\ nil)\ ((eq\ x\ 1)\ t)\ \big((>\ x\ 0)\ (odd\ (-x\ 2))\big)\ (t\ \big(odd\ (+x\ 2)\big))))$$

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 alternative interpretation described in part (b).

Answer. (c)

$$(define\ odd(x)\Big(cond\ ((eq\ x\ 0)\ nil)((eq\ x\ 1)\ true)\big((>\ x\ 1)(odd\ (-\ x\ 2))\big)\big((<\ x\ 0)(odd\ (+x\ 2))\big)\Big))$$

(d) The normal implementation of Boolean OR is designed not to evaluate a subexpression unless it is necessary. This is called the short-circuiting OR, and it may be defined as follows:

$$SCOR(e_1, e_2) = \begin{array}{c} true \ \ if \ e_1 = true \\ true \ \ if \ e_1 = false \ and \ e_2 = true \\ false \ \ if \ e_1 = e_2 = false \\ undefined \ \ otherwise \end{array}$$

It allows e_2 to be undefined if e_1 is true and also allows e_1 to be undefined if e_2 is true. 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 interpretation would make implementations of short-circuiting OR difficult? Which interpretation would make implementation of parallel OR difficult? Why?

Answer. (d)

The implementation of the section a will make the definition of Por harder a bit but, defining $Scor$ with the implementation in the section b would take years to be done.

3.4 **Lisp and Higher-Order Functions**

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

$$(define\ compose\ \Big(lambda\ (f\ g)\ \Big(lambda\ (x)(f\ (g\ x))\Big)\Big))$$

$$(define\ mapcar\ \Big(lambda\ (f\ xs)\Big(cond\ \Big((eq?\ xs(\ \ )(\ \ )\Big)\Big(\#t\ \Big(cons(f\ (car\ xs))(mapcar\ f(cdr\ xs))\Big)\Big)\Big)\Big))$$

$$(define\ maplist\ \Big(mabda\ (f\ xs)\Big(cond\ \Big((eq?\ xs\ (\ \ )(\ \ )\Big)\Big(\#t\ \Big(cons\ (f\ xs)(maplist\ f(cdr\ xs))\Big)\Big)\Big)\Big))$$

The difference between maplist and mapcar is that maplist applies f to every sublist, whereas mapcar applies f to every element. Write a version of compose that lets us define mapcar from maplist. More specifically, write a definition of compose2 so that

$$\big((compose2\ maplist\ car)f\ xs\big) = (mapcar\ f\ xs)$$

(a) Fill in the missing code in the following definition. The correct answer is short and fits here easily.

Answer. (a)

Compose2 is a function that tales two functions as its arguments and returns another two-argument function as a result.

$$(define\ compose2\ \Big(lambda\ (g\ h)\Big(lambda\ (f\ xs)\ \big(g\ \big(lambda\ (xs)\ \big(f\ (h\ (xs))\big)\big)xs\ \big)\Big)\Big))$$

(b) When $(compose2\ maplist\ car)$ is evaluated, the result is a function defined by $(lambda\ (f\ xs)(g\ \ldots))$ above, with
$i.$ Which function replacing g?
$ii.$ and which function replacing h?

Answer. (b)

Function g will be replaced with the function maplist. The maplist function contains two arguments in which one of the is a list ($list\ [xs]$) and the other one is a function ($[f\big(h(xs)\big)]$).

(c) We could also write the subexpression $\big(lambda\ (xs)(\ldots)\big)$ as $(compose\ (\ldots)(\ldots))$ for two functions. Which two function are these?

Answer. (c)

$$lambda\ (xs)(f\ \big(h\ (xs)\big)) = compose\ (f\ h) = compose(f\ car)$$

3.5 **Definition of Garbage**

This question asks you to think about garbage collection in Lisp and compare our definition of garbage in the text to the one given in McCarthy's 1960 paper on Lisp. McCarthy's definition is written for Lisp specifically, whereas our definition is stated generally for any programming language. Answer the question by comparing the definitions as they apply to Lisp only. Here are the two definitions.

$Garbage, our\ definition.$ At a given point in the execution of a program P, a memory location m is garbage if no continued execution of P from this point can access location m.

$Garbage, McCarthy's\ definition.$ "Each register that is accessible to the program is accessible because it can be reached from one or more of the base registers by a chain of car and cdr operations. When the contents of a base register are changed, it may happen that the register to which the base register formerly pointed cannot be reached by a car–cdr chain from any base register. Such a register may be considered

abandoned by the program because its contents can no longer be found by any possible program."

(a) If a memory location is garbage according to our definition, is it necessarily garbage according to McCarthy's definition? Explain why or why not.

Answer. (a)

Yes, it is. If a particular section of the memory (say m) is not accessible from any part of the program, we can easily infer that there is no reference pointing to that part of the memory.

(b) If a location is garbage according to McCarthy's definition, is it garbage by our definition? Explain why or why not.

Answer. (b)

No, it is not. Forgetting about a register (for example an address in some memory) cannot cause the address to be forgotten according to the general definition of the garbage.

(c) There are garbage collectors that collect everything that is garbage according to McCarthy's definition. Would it be possible to write a garbage collector to collect everything that is garbage according to our definition? Explain why or why not.

Answer. (c)

Yes, it is. Since the definition is based on the execution of the program, the garbage detection can be done easily like what we'll see in the next question but, it is important to note that predicting the garbage in a program statically might be hard or even undecidable.

3.6 **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, cons. Impure Lisp alsp 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 is returned to the free-storage list.

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

$$(car\ \left(cdr\ \left(cons\ (cons\ a\ b)(cons\ c\ d)\right)\right))$$

How many of the three cons cells generated by the evaluation of this expression can be returned to the free-storage list?

Answer. (a)

The second cons cell which is $(cons\ a\ b)$ 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. Explain why the reference counting algorithm does not work properly on this structure.

Answer. (b)

There might be possible occurrence of reference-to-self with an infinite depth, which is out of control using predefined garbage collection algorithm.

3.8 **Concurrency in Lisp**

The concept of $future$ was popularized by R. Halstead's work on the language Multi-lisp for concurrent Lisp programming. Operationally, a future consists of a location in memory and a process that is intended to place a value in this location at some time "In the future."

(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?

Answer. (a)

Considering no future, we'll have the following complexity for the fib function:

$$O(fib(N)) = O\big(fib(N-1)\big) + O\big(fib(n-2)\big) = O(n^2)$$

But there will be better results with the future:

$$O\big(fib(N)\big) = MAX\left\{ O\big(fib(N-1)\big), O\big(fib(N-2)\big) \right\} = O(n)$$

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

Which differ only because an occurrence of a subexpression e is replaced with (future e), would be equivalent. However, there are some circumstances in which 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.

Answer. (b)

The behavior of the following expressions is obviously differing from each other.

$$(cons\ x\ (cons\ (rplaca\ (cddr\ x)y)x))$$

$$(cons\ x\ \big(cons\ \big(future\ (rplaca\ (cdr\ x)y)\big)\ x\big))$$

(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.

Answer. (c)

Comparing the following expressions, in the first expression, we will obviously expect the value x from the condition (considering b is a true value) but, in the second expression, the parallel evaluation of the condition expression may result in the unwanted undefined value.

$$(cond\ (false\ x)(true\ y)(false\ unwantedvalue(undefined))$$
$$(cond\ (false\ x)(true\ y)(false\ future(unwantedvalue(undefined))$$

(d) Suppose you are part of a language design team that has adopted futures as an approach to concurrency. Name two problems or important interactions between error handling and concurrency that you think need to be considered.

Answer. (d)

i. When a concurrent execution of a program, wants to access the same shared area of the memory, at the same time, there will be a crucial need for the error handling to play an unfaulty role.

ii. Multiple programmer maybe included to handle the exceptions, if the system is not properly automated.