

## CS 3723: Homework #1 Solutions

### PART 1:

1-

A total function has exactly one result value for every input argument in its domain. However, a function declared in a program may return a result or may not if some loop or sequence of recursive calls does not terminate.

2-

Assuming the domain/range of integers:

a) Graph:  $\{(-1,-1), (1,1)\} \cup \{(x,5x) \mid \text{where } x > 1\}$

Partial function.

Defined over integers for all positive  $x$  and for  $-1$ .

Undefined for  $x = 0$  and (if range is restrict to integers)  $x < -1$ .

b) Graph:  $\{(x,3)\}$

Total function.

c) Graph:  $\{(x,1) \mid x \geq 0\}$

Partial function.

Defined for all non-negative integers. Undefined for  $x < 0$ .

3-

First: “a mathematical definition” of partial ( $\mu$ -recursive) functions.

Second: “a kind of idealized computing device called a Turing machine.”

Third: “an equivalent definition that uses lambda calculus.”

4-

A language that is Turing complete can be used to express precisely the same class of partial recursive functions as the Turing-machine,  $\lambda$ -calculus, and most programming languages.

5-

Most programming languages are Turing-complete including as Ada, C++, and Lisp.

6-

Example of a non-Turing complete language: Cascade-Style Sheets. CSS includes no looping/repetition/recursion constructs, and all HTML/CSS web pages finish rendering fairly quickly (in a finite amount of time).

7-

csh is Turing-complete because it provides control flow operations such as while, if-else with arbitrary controlling expressions.

8-

The fact that all programming languages are Turing-complete means they are equally powerful in terms of what they can be used to compute.

9-

A Turing machine can use an arbitrary amount of space on the tape and an arbitrary amount of time to perform a computation. However, in real life, we are limited in the amount of *time* and storage *space* available to solve a problem.

10-

A solution to the halting problem,  $Halt(P,I)$ , could be written by taking the program  $P$  that reads an input and modifying to create a  $P'$  that has the input hard-coded. (The specifics of how to do this depend on what assumptions are made about the language of  $P$ . If you assume (as in Mitchell, Exercise 2.3) that  $P$  begins with a read statement that reads a single integer, then  $I$  should be single integer, and you can simply replace the read statement with an assignment of integer  $I$  to the same variable that would otherwise be assigned to by the read statement.) This  $P'$  can then be passed to  $Halt_0$  to determine whether  $P'$  halts. If  $P'$  halts then  $P$  halts on  $I$ ; if  $P'$  does not halt,  $P$  does not halt on  $I$ .

## PART 2:

1-

Artificial Intelligence (specifically, logical deduction and symbolic calculation)

2-

Recursive functions, Lists, Programs as data (a program can build the list representation of a function or other forms of expression and then use the eval function to evaluate the expression.), Garbage collection.

3-

eq: "tests whether its arguments are the same sequence of memory locations"

eq1: "tests whether its arguments are the same symbol or number"

equal: "is a recursive equality test on lists or atoms that is implemented by use of eq and ="

=: "is numeric equality" (Mitchell, p. 28)

eq corresponds to eq? in Scheme

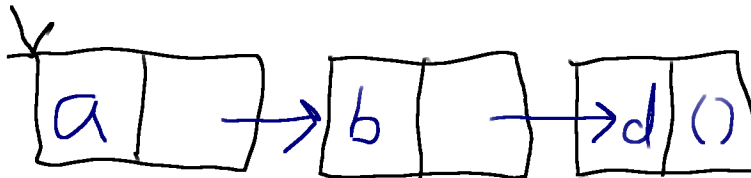
eq1 corresponds to eqv? in Scheme

equal corresponds to equal? in Scheme

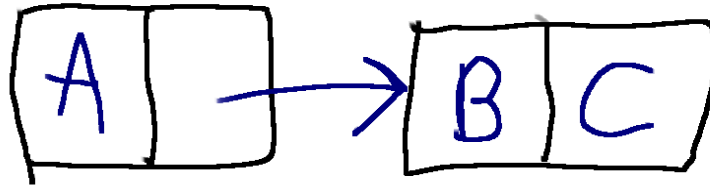
= corresponds to = in Scheme

4-

'(a b d)



5-  
a)



b)  
(cons

```
(cons 'A (cons 'B 'C))  
(cons 'B 'C))
```

c)  
((lambda (x) (cons (cons 'A x) x))  
(cons 'B 'C))

First, (cons 'B 'C) is evaluated, then it is given to the function.

6-

```
(define factorial (lambda (n)  
  (if (or (= n 0) (= n 1))  
      1  
      (* n (factorial (- n 1))))))
```

7-

```
(define count (lambda (lst sym)  
  (cond  
    ((null? lst) 0)  
    ((pair? lst) (+ (count (car lst) sym) (count (cdr lst) sym)))  
    (else (if (eqv? lst sym) 1 0))))))
```

8-

a)  
(define compose2  
 (lambda (g h)  
 (lambda (f xs)  
 (g (lambda (xs) (f (h xs))) xs))))

b)  
I. g is replaced with maplist  
II. h is replaced with car

c)  
f and h: (lambda (x) (f (h xs))) = (compose f h)

9-

- a) No. There may still be a live variable holding a pointer to some cons cell, but that variable may never be used. This cons cell would not be garbage according to “McCarthy” but would be according to “our definition”
- b) Yes. Since if a cons cell is garbage according to McCarthy, there will be no way to traverse the data structures in memory to get to the cons cell, then there can be no continued program execution that will access such a cons cell.
- c) No. “Our definition” is stronger than McCarthy's, and to write a program that finds “garbage, our definition” precisely would require solving the halting problem. The only access to some memory location might occur after some function call. Determining whether that function call returns would require solving the Halting Problem.