

## CS 405/505 EXAM #2 SAMPLE QUESTIONS (SOLUTION)

1. The mathematical model of computation that denotational semantics is based upon is called lambda calculus.
2. A(n) loop invariant is a condition which is true upon entry to a loop every time the loop is entered, and is still true at the termination of the loop.
3. True or False. Denotational semantics is a method for defining the semantics of a programming language by giving an interpreter for that language.

False. Denotational semantics defines a mapping from syntax domains to functions over semantic domains. These functions may be executed to effect an interpreter.

4. In pure  $\lambda$ -calculus, integers, Boolean values, lists, etc., must all be represented by  $\lambda$ -expressions. For example,  $0 = \lambda x.\lambda y.y$ ,  $1 = \lambda x.\lambda y.x y$ ,  $2 = \lambda x.\lambda y.x (x y)$ , etc., and  $succ = \lambda z.\lambda x.\lambda y.x ((z x) y)$ .  $succ$  is a function which adds 1 to its argument. Evaluate  $succ\ 1$ . Did you get 2? Why or why not?

$$succ\ 1 = (\lambda z.\lambda x.\lambda y.x ((z\ x)\ y)) (\lambda x.\lambda y.x\ y) \Rightarrow \lambda x.\lambda y.x (((\lambda x.\lambda y.x\ y)\ x)\ y)$$

Technically, the reduction stops here as there are no actual parameters corresponding to  $x$ . However, by inspecting the inside expression, there is an internal redex which may be reduced further as follows (including further reductions of internal redexes).

$$\begin{aligned} \lambda x.\lambda y.x (((\lambda x.\lambda y.x\ y)\ x)\ y) &\Rightarrow \lambda x.\lambda y.x ((\lambda y.x\ y)\ y) \\ &\Rightarrow \lambda x.\lambda y.x ((\lambda y.x\ y)\ y) \Rightarrow \lambda x.\lambda y.x (x\ y) = 2 \end{aligned}$$

5. Consider the following Lisp function:

```
(defun update (store V n)
  (prog (currentenv newbinding newstore)
    (setq currentenv (cdaddr store))
    (setq newbinding (cons (list 'eq 'V (list 'quote V)) (list n)))
    (setq newstore (list 'lambda (list 'V) (cons 'cond (cons newbinding currentenv))))
    (return newstore)
  )
)
```

Note that the `(prog (list of local variables) (exp-1) (exp-2) ... (exp-n) (return exp))` construction declares a list of local variables used in the expressions with `exp` being the value returned. Let  $S$  represent the Lisp expression `(lambda (V) (cond ((eq V (quote x)) 4) (t (quote bottom))))`. What is the internal list representation that Lisp uses for  $S$ ? What is `(cdaddr S)`?

```
((EQ V 'X) 4) (T 'BOTTOM)
```

Trace the execution of (update S 'x 5), showing the values of currentenv, newbinding, and newstore.

```
>(setq currentenv (cdaddr store))
(((EQ V 'X) 4) (T 'BOTTOM))
>(setq newbinding (cons (list 'eq 'V (list 'quote 'x)) (list 5)))
((EQ V 'X) 5)
>(setq newstore (list 'lambda (list 'V) (cons 'cond (cons newbinding currentenv))))
(LAMBDA (V) (COND ((EQ V 'X) 5) ((EQ V 'X) 4) (T 'BOTTOM)))
```

6. Consider the Prolog program which solves the Towers of Hanoi problem. This problem is to move a stack of blocks from one location to another without ever changing the order in which blocks are placed on top of each other. A third stack may be used but only one block may be moved at a time.

```
hanoi(N) :- move(N, left, center, right).
```

```
move(0, X, Y, Z).
```

```
move(N, X, Y, Z) :-
```

```
    N > 0, M is N - 1, move(M, X, Z, Y), print([X,Y]), move(M, Z, Y, X).
```

Trace the above program using the query hanoi(3). Show all queries which are generated by the execution and all moves which are printed. You may wish to draw a tree to show this clearly.

```
Script started on Tue 22 Mar 2005 08:57:15 AM CST
redstone% sbp
SB-Prolog Version 3.1
| ?- consult('hanoi.pro').
yes
| ?- trace(hanoi/1).
yes
| ?- trace(move/4).
yes
| ?- hanoi(3).
    (1) Call: hanoi(3) ? y
no
| ?- hanoi(3).
    (2) Call: hanoi(3) ?
    (3) Call: move(3,left,center,right) ?
    (4) Call: move(2,left,right,center) ?
    (5) Call: move(1,left,center,right) ?
    (6) Call: move(0,left,right,center) ?
    (6) Exit: move(0,left,right,center)
[left,center] (7) Call: move(0,right,center,left) ?
    (7) Exit: move(0,right,center,left)
    (5) Exit: move(1,left,center,right)
```

```
[left,right] (8) Call: move(1,center,right,left) ?
(9) Call: move(0,center,left,right) ?
(9) Exit: move(0,center,left,right)
[center,right] (10) Call: move(0,left,right,center) ?
(10) Exit: move(0,left,right,center)
(8) Exit: move(1,center,right,left)
(4) Exit: move(2,left,right,center)
[left,center] (11) Call: move(2,right,center,left) ?
(12) Call: move(1,right,left,center) ?
(13) Call: move(0,right,center,left) ?
(13) Exit: move(0,right,center,left)
[right,left] (14) Call: move(0,center,left,right) ?
(14) Exit: move(0,center,left,right)
(12) Exit: move(1,right,left,center)
[right,center] (15) Call: move(1,left,center,right) ?
(16) Call: move(0,left,right,center) ?
(16) Exit: move(0,left,right,center)
[left,center] (17) Call: move(0,right,center,left) ?
(17) Exit: move(0,right,center,left)
(15) Exit: move(1,left,center,right)
(11) Exit: move(2,right,center,left)
(3) Exit: move(3,left,center,right)
(2) Exit: hanoi(3)
```

yes

| ?- ^D

Halt. Program terminated normally

redstone% exit

redstone%

script done on Tue 22 Mar 2005 08:59:28 AM CST

7. Consider the denotational semantics below:

$$\begin{aligned}
S[\mathbf{S}_1; \mathbf{S}_2] \text{ store} &= S[\mathbf{S}_2] (S[\mathbf{S}_1] \text{ store}) \\
S[\mathbf{V} := \mathbf{E}] \text{ store} &= \text{store}[E[\mathbf{E}] \text{ store}/\mathbf{V}] \\
S[\mathbf{while} \ \mathbf{C} \ \mathbf{loop} \ \mathbf{S} \ \mathbf{end} \ \mathbf{loop}] \text{ store} &= \\
&\quad \text{if } C[\mathbf{C}] \text{ store} \text{ then } S[\mathbf{while} \ \mathbf{C} \ \mathbf{loop} \ \mathbf{S} \ \mathbf{end} \ \mathbf{loop}] (S[\mathbf{S}] \text{ store}) \text{ else } \text{store} \\
C[\mathbf{E}_1 > \mathbf{E}_2] \text{ store} &= \text{if } E[\mathbf{E}_1] \text{ store} > E[\mathbf{E}_2] \text{ store} \text{ then } \text{true} \text{ else } \text{false} \\
E[\mathbf{E}_1 + \mathbf{E}_2] \text{ store} &= E[\mathbf{E}_1] \text{ store} + E[\mathbf{E}_2] \text{ store} \\
E[\mathbf{E}_1 - \mathbf{E}_2] \text{ store} &= E[\mathbf{E}_1] \text{ store} - E[\mathbf{E}_2] \text{ store} \\
E[\mathbf{I}] \text{ store} &= N[\mathbf{I}] \\
E[\mathbf{V}] \text{ store} &= \text{if } \text{store}[\mathbf{V}] = \perp \text{ then } \top \text{ else } \text{store}[\mathbf{V}]
\end{aligned}$$

Assume that  $N$  returns the integer value of its argument. Given the initial store  $(\lambda \mathbf{V}. \perp) [1/u] [5/y] [5/z]$ , compute the store which results from evaluating  $S$  on the program below, showing all steps used in the evaluation.

```

while (u > 0) loop
  u := u - 1;
  z := z + y
end loop

```

8. Consider the axioms of assignment, composition, and loop below:

$$\text{Assignment} \quad \{P \ [E/V]\} \ \mathbf{V} := \mathbf{E} \ \{P\}$$

Composition

$$\frac{\{P\}S_1\{Q\}, \{Q\}S_2\{R\}}{\{P\}S_1; S_2\{R\}}$$

Loop

$$\frac{\{P \ \& \ B\}S\{P\}}{\{P\} \mathbf{while} \ B \ \mathbf{loop} \ S \ \mathbf{end} \ \mathbf{loop} \ \{P \ \& \ \neg B\}}$$

Prove the correctness of the following program segment.

```

{n = j * (j + 1) / 2 & i >= 0}
while j != i loop j := j + 1; n := n + j; end loop
{n = j * (j + 1) / 2 & i >= 0 & ¬(j = i)}

```

Show all steps used in the proof.