Principles of Programming Languages - Final (Solutions)

K. Subramani
LDCSEE,
West Virginia University,
Morgantown, WV
{ksmani@csee.wvu.edu}

1 Instructions

(a) The final is to be turned in by $10:30$ am.

(b) Each question is worth 4 points.

(c) Attempt as many questions as you can; you will be given partial credit.

2 Problems

1. **Syntax:**
   Consider the context-free grammar $G = \langle V, T, P, S \rangle$, where $V = \{S\}$, $T = \{0, 1\}$, and the productions P are defined by:

   $$S \rightarrow 0\,S\,1 \mid 1\,S\,0 \mid \epsilon$$

   Argue that every string generated by this grammar is *balanced*, i.e., if w is derived from S, then $n_0(w) = n_1(w)$, where $n_0(w)$ and $n_1(w)$ stand for the number of 0s and 1s respectively in w.

   Solution:

   We use induction on the number of steps used in the leftmost derivation of w from S.

   BASIS: Let w be derived from S in exactly one step. From the production rules, it is clear that w must be ϵ and hence w is indeed balanced.

   INDUCTIVE STEP: Assume that the theorem is true for all strings w, whose leftmost derivations from S, take at most n steps.

   Now consider the case in which the leftmost derivation of w from S takes $n+1$ steps, where $n \geq 1$. The first step of the derivation must be one of $S \Rightarrow 0S1$ or $S \Rightarrow 1S0$.

   Assume that the first step of the derivation is $S \Rightarrow 0S1$. It follows that $w = 0x1$, where x is a string in Σ^*. Since $S \Rightarrow^* w$, we must have $S \Rightarrow^* x$; however, the leftmost derivation of x from S can take at most n steps. By the inductive hypothesis, it follows that x is balanced. Consequently, $w = 0x1$ is also balanced.

   An identical argument can be used for the case, in which the first step of the derivation is $S \Rightarrow 1S0$. $\Box$

2. **Semantics:**

   (a) Explain with an example the difference between the scope of a binding and its visibility. What is a scope-hole?

   (b) Explain the difference between storage semantics and pointer semantics when it comes to variable assignment.

   (c) Which of the following expressions is an l-value in **C**:

(i) $\&(*x + 1)$.
(ii) $*(\&x + 1)$.

Solution:

(a) The scope of a binding refers to the region of the program over which it is maintained, whereas its visibility refers to the region of the program where it applied. For instance, in the following **C** fragment,

```
int x;

int p (void)
{
  int x;

  x = 5;
}
```

the scope of the globally declared x extends over p, whereas its visibility does not. The global declaration of x is said to have a scope-hole in $p()$.

(b) In storage semantics, the assignment $x = y$, causes the *value* of y to be copied over into the value of x, whereas in pointer semantics, the location of x is bound to the location of y.

(c) (i) Not an l-value, since addresses are assigned by the system and cannot be reassigned.
(ii) The expression is an l-value, although unsafe; it is referring to the memory location just above (or just below) the location alloted to x.

□

3. **Procedures and Environments:**

(a) Provide a brief description of the working of the *mark-and-sweep* algorithm with respect to storage reclamation, identifying its drawbacks.

(b) Describe one improvement to the *mark-and-sweep* framework that has been implemented in current memory management systems.

Solution:

(a) The *mark-and-sweep* algorithm consists of two passes; in the first pass, the environment pointer is followed to recursively identify all blocks of memory that are accessible; these blocks are marked accordingly. In the second pass, which is a linear sweep of all memory, the unmarked blocks are returned to the free memory list. The disadvantages are as follows: extra storage for marking and significant delays in processing.

(b) One improvement to the above scheme is the *stop and copy* technique; here the available memory is divided into two halves and storage is allocated from only one half at a time. In the marking phase, all the identified blocks are moved immediately to the second half and the first half is considered deallocated (i.e., free). Under this scheme, only one pass is required and there is no need for a storage bit.

□

4. **Functional Programming:**

(a) In class, we discussed a SCHEME function to reverse a list; that function reverses only the top level of the input list. For instance, consider the list $((1\ 2)\ 3\ (4\ 5)$; the function discussed in class would produce the list $((4\ 5)\ 3\ (1\ 2))$ on reversal. Write a SCHEME function called *deep-reverse* that recursively reverses all the sublists of an input list; for instance, the deep-reversal of the above list should return $((5\ 4)\ 3\ (2\ 1))$. You may assume the existence of the *append()* function in SCHEME which takes two lists L and M as input and returns a list which contains all the elements of L followed by all the elements of M, in precisely the same order as they occur in L and M.

(b) Consider the following expression in SCHEME:

```
(let ((x 2) (y 3) E)
```

where E is an arbitrary expression. Rewrite the above expression as a lambda application without using **let**.

Solution:

(a) Here is one approach:

```
(define (deep-reverse L)
   (cond ((null? L) L)
         ((list? L) (append (deep-reverse (cdr L))
                            (list (deep-reverse (car L)))))
         (else L)))
```

Try writing the same function in **C** and you will appreciate the power of SCHEME!

(b) `((lambda x y) E) 2 3)`

□

5. **Logic Programming:**

(a) Let P, Q and R be propositions. Prove the validity of the following argument without using truth-tables.

$$[P \wedge (Q \vee R)] \rightarrow [(P \wedge Q) \vee (P \wedge R)]$$

You may assume the following tautology called the Deduction Method:

$$[(A \wedge B) \rightarrow (C \rightarrow D)] \leftrightarrow (A \wedge B \wedge C) \rightarrow D)$$

(b) Write a Prolog fragment that reverses a list, using pattern-directed invocation. You may assume that the list is composed of atoms only.

Solution:

(a) Observe that the conclusion of the argument, i.e., $[(P \wedge Q) \vee (P \wedge R)]$ can be rewritten as: $[(P \wedge Q)' \rightarrow (P \wedge R)]$. Hence, the original argument can be reformulated as:

$$[P \wedge (Q \vee R)] \rightarrow [(P \wedge Q)' \rightarrow (P \wedge R)]$$

Using the Deduction Method, we can reformulate the above argument as:

$$[P \wedge (Q \vee R) \wedge (P \wedge Q)'] \rightarrow (P \wedge R)$$

Consider the following proof sequence:

(i) P hypothesis.
(ii) $Q \vee R$ hypothesis.
(iii) $(P \wedge Q)'$ hypothesis.
(iv) $P' \vee Q'$ (iii), De Morgan's Law.
(v) $Q' \rightarrow R$ (ii), implication.
(vi) $P \rightarrow Q'$ (iv), implication.
(vii) Q' (i), (vi), Modus Ponens.
(viii) R (v), (vii), Modus Ponens.
(ix) $(P \wedge R)$ (i), (viii), Conjunction.

(b) Here is one approach:

```
append([],Y,Y).
append([A|B],Y,[A|W]) :- append(B,Y,W).

reverse([],[]).
reverse([H|T],L):-reverse(T,L1), append(L1,[H],L).
```

□