Programming Languages

Spring 1992

1 hour, 60 points

1. (*20 points*) This question is about statement and program equivalence. For the purposes of this question, let us use the word *program* to mean a program that reads no input and either runs forever or halts without producing any output. The reason for this is just to simplify the observable behavior of programs. Under this assumption, we say that two programs are *equivalent* if both halt or both run forever.

   Two statements, S_1 and S_2, are $\mathcal{L}$-*equivalent* if, for every program P in the programming language $\mathcal{L}$, the program obtained by replacing one or more occurrences of S_1 by S_2 is equivalent to P. In other words, two statements are equivalent if we can replace one by the other, in any larger program, without changing the observable behavior of the program.

   (a) Suppose one language, $\mathcal{L}_1$, is a subset of another, $\mathcal{L}_2$. This means that every expression, statement, etc. of $\mathcal{L}_1$ is an expression, statement, etc. of $\mathcal{L}_2$. In addition, let us assume there is one compiler that we use to compile programs of both languages.

   Explain how two statements, S_1 and S_2, can be $\mathcal{L}_1$-equivalent but not $\mathcal{L}_2$-equivalent (see part (b) for part of an example). Is it possible for S_1 and S_2 to be $\mathcal{L}_2$-equivalent but not $\mathcal{L}_1$-equivalent? Why or why not?

   (b) Let $\mathcal{L}_1$ be Pascal and $\mathcal{L}_2$ be Pascal extended with `cobegin` ... `coend`. If $S_1, \ldots, S_k$ are statements, then

   `cobegin` S_1 `and ... and` S_k`coend`

   is executed by executing $S_1, \ldots, S_k$ in parallel. These statements may contain global variables (variables declared outside the `cobegin` ... `coend`) and call global procedures. For simplicity, let us assume that none may jump out of the `cobegin` ... `coend` block. The `cobegin` ... `coend` block terminates when all of $S_1, \ldots, S_k$ have terminated.

   Give an example of two statements that are $\mathcal{L}_1$-equivalent but not $\mathcal{L}_2$-equivalent. (Hint: remember that if S_1 and S_2 are Pascal statements, then $S_1; S_2$ is a Pascal statement.)

2. (*20 points*) Syntactically, the main difference between Prolog clauses and ML functions defined by pattern matching is that in Prolog a variable may occur twice in the head of a clause while no variable may occur more than once in an ML pattern. To refresh your memory, ML and Prolog programs to find a key in a binary search tree (from Sethi's book) are given in Table 1.

   Compare unification in Prolog with pattern-matching in ML. Which is easier to implement? Why? How would the ML implementation change if we allow a variable to occur more than once in a pattern? If we extend ML by generalizing patterns in this way, do we get a "logic programming" language? For example, is there an easy translation of Prolog programs into ML ? Discuss.

3. (*20 points*) An idea that is currently in vogue is to compile higher-level programming languages into C, instead of generating assembly code. This question asks you to identify some advantages and disadvantages of compiling to C.

```
ML:

datatype searchtree = empty | node of int * searchtree * searchtree;

fun member(x, empty) = false
  | member(x, node(y, t1, t2)) = if x=y then true else
           if x < y then member(x,t1) else member(x,t2);

Prolog:

member(K, node(K, _, _)).
member(K, node(N, S, _)) :-  K < N, member(K,S).
member(K, node(N, _, T)) :-  K > N, member(K,T).
```

Table 1: Example ML and Prolog programs

(a) If you are designing a new language, what are the main advantages of compiling it to C? What are some general problems that would arise in compilation and debugging?

(b) In a laboratory where Lisp or Scheme, an object-oriented language such as Smalltalk, and a Pascal-like language are all commonly used, what are the main advantages of compiling all three languages to C? Can this help you call a Lisp function from a Pascal program?

(c) What specific features of a function-based language such as Scheme, Lisp or ML make compilation to C difficult? Assuming that function declarations in the source language are compiled into C functions, what storage management problems occur? Discuss the trade-offs between allocating space for local variables on the C run-time stack and allocating activation records explicitly in your compiler.