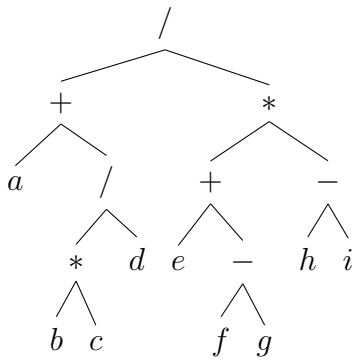
Suggested Solutions to Midterm Problems: Part I

1. (10 points)

   (a) Draw an abstract syntax tree for the following expression in the postfix notation:

   ```
   a b c * d / + e f g - + h i - * /
   ```

   Solution. (Jui-Shun Lai)

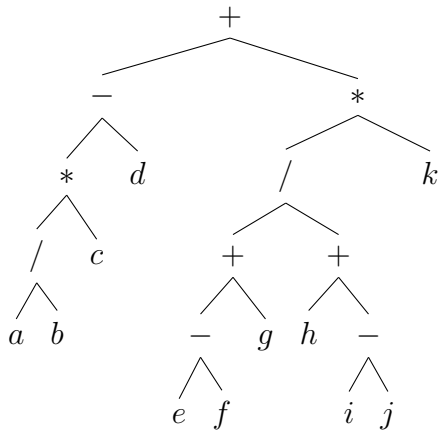


□

   (b) Draw an abstract syntax tree for the following expression in the prefix notation:

   ```
   + - * / a b c d * / + - e f g + h - i j k
   ```

   Solution. (Jui-Shun Lai)



□

2. (10 points) What are the characteristics of functional programming? Identify any two of them and explain.

   Solution.

- Programming without assignments.
  The value of an expression depends only on the values of its subexpressions, if any. The order in which the subexpressions are evaluated (as long as the evaulations terminate) should not affect the final value of the expression.
- Implicit storage management.
  Storage is allocated as necessary by built-in operations on data. Storage that becomes inaccessible is automatically deallocated.
- Functions as first-class values.
  Functions have the same status as any other values. A function can be the value of an expression, it can be passed as an argument, and it can be put in a data structure.

□

3. (10 points) What is a type? How can types be built from other more basic types? How can new basic types be created?

   Solution. A type consists of a set of elements (called values) together with a set of functions (called operations).

   New types can be created by applying type constructors, such as function (`->`), product (`*`), and list, to basic types or other types so defined. For example, `int -> bool` is the type consisting of all functions from `int` to `bool`

   The simplest way for creating a new basic type is enumeration, i.e., explicit listing of all elements in the type. Recursive/inductive defintions may be used to define a new basic type with an infinite number of elements. □

4. (10 points) Fill in the blanks in the following equivalences:

   (a) `reduce _____ x _____` ≡ `append x y`

   Solution. (Jui-Shun Lai)

   `reduce (fun a b -> a::b) x y` ≡ `append x y` □

   (b) `reduce _____ x _____` ≡ `map f x`

   Solution. (Jui-Shun Lai)

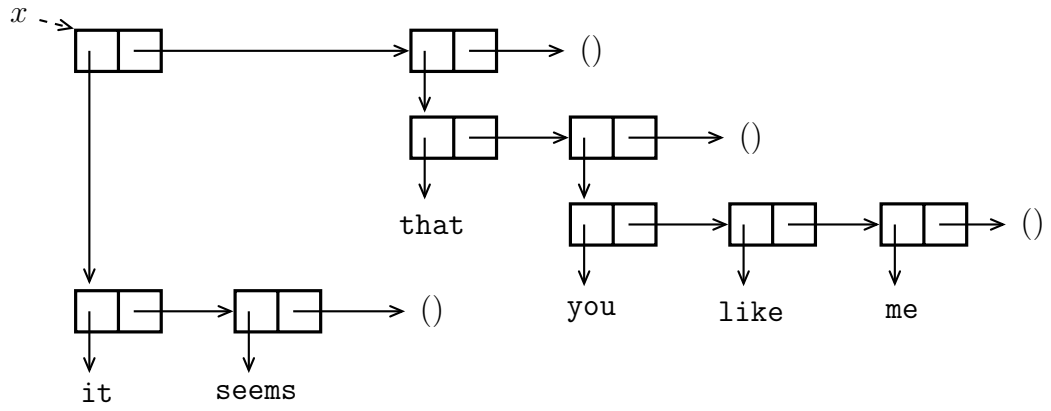
   `reduce (fun a b -> (f a)::b) x []` ≡ `map f x` □

5. (10 points) Suppose we have typed the following Common Lisp expression into an interpreter and got the response.

```
(setq x '((it seems) (that (you like me))))
```

   (a) Draw a figure showing the structure of allocated cells for `x`.

   Solution. (Jui-Shun Lai)



□

(b) What is the value of `(car (cdr x))`?

Solution. (Jui-Shun Lai)

`(that (you like me))` □

6. (5 conditional extra points; credited only if your score of this part would not exceed 50 points)

Though we did not formally define the notion of a tail-recursive function, we have seen and practiced writing in homework assignments several tail-recursive functions like the following.

```
let rec len x res =
  match x with
    [] -> res
  | _::y -> len y (1 + res)
```

So, what is a tail-recursive function? Please try to give it a more precise definition. (Hint: think about the innermost evaluation of a function step by step.)

Solution. A function is tail-recursive if every recursive invocation of the function appears as the last step in the evaluation of (the current invocation of) the function. □

Appendix

- User-defined `append`, `map`, and `reduce` in OCaml:

```
let rec append x y =
  match x with
    [] -> y
  | a::xs -> a :: (append xs y)

let rec map f x =
```

```
    match x with
      [] -> []
    | a::y -> (f a) :: (map f y)

let rec reduce f x v =
  match x with
    [] -> v
  | a::y -> f a (reduce f y v)
```