CS 538

Midterm Exam

Wednesday, April 3, 2002

7:15 PM — 9:15 PM

1221 Computer Sciences

Instructions

Answer question #1 and any three others. (If you answer more, only the first four will count.) Point values are as indicated. Please try to make your answers neat and coherent. Remember, if we can't read it, it's wrong. Partial credit will be given, so try to put something down for each question (a blank answer always gets 0 points!).

1. (1 point)
   The MultiLisp programming languages is based upon:
   (a) APL
   (b) Autocoder
   (c) Jovial
   (d) Scheme (and Lisp indirectly)
   (e) Snobol

2. (a) (16 points)
   It is often the case that subprograms are called with some arguments that are constants rather than variables. In what ways can the execution of a subprogram be improved if it is known that a particular parameter will always be a given constant parameter? Are the improvements you propose **safe**? That is, will your "improved" subprogram always compute the same result as the original subprogram?

   (b) (17 points)
   In MultiLisp we can use `pcall` to specify that the function and parameters in a call are to be evaluated in parallel. Is it safe to convert every function call into a `pcall`? If so, explain why. If not, how can a programmer decide which calls are appropriate for conversion into `pcall`s?

3. (a) (5 points)
   Scheme's `map` function, called as `(map function list)` applies its `function` argument to each element of its `list` argument producing a list of results. The i-th element of the output list is the result of applying the function to the i-th element of the input list. However, the exact order in which the function is applied to list elements (left-to-right, right-to-left, or some other) is unspecified.

   Write a simple Scheme function that uses `map` that will expose the order in which `map` applies its function to list elements. Indicate how the output produced by the function you provide will answer our ordering question.

(b) (14 points)

`map` normally applies one function to a list of parameter values. Assume we want a variant of map, called `mapF`, that applies a list of functions to a single parameter value. For example, given

```
  (define (add1 v) (+ v 1))
  (define (add2 v) (+ v 2))
  (define (add3 v) (+ v 3))
```

the call `(mapF (list add1 add2 add3) 10)` produces `(11 12 13)`.
Give Scheme code that defines `mapF`.

(c) (14 points)

Now let us consider a version of `map` called `mapFL` that takes a list of functions and a list of parameter values. That is, `(mapFL FunL L)` takes the first function in `FunL` and applies it to each value in `L`, forming a list of results. This list is then appended to a list of results formed by applying the second function in `FunL` to the values in `L`. Then the third function in `FunL` is applied to `L`, etc., until all functions in `FunL` have been applied to `L`. For example,
`(mapFL (list add1 add2 add3) '(10 11 12))` produces `(11 12 13 12 13 14 13 14 15)`.
Give Scheme code that defines `mapFL`.

4. (a) (18 points)

Let `L` be a list of values. Write a Scheme function `(split L)` that splits the values of `L` into two sublists as follows:

(i) If `L` is null, the pair `(() . ())` is returned.
(ii) If `L` is (v$_1$) the pair ((v$_1$) . ()) is returned.
(iii) In general, `L` is divided into a pair of lists containing values at odd numbered positions and values at even numbered positions. Thus (v$_1$ v$_2$ v$_3$ v$_4$ ...) is split into ((v$_1$ v$_3$ ...) . (v$_2$ v$_4$ ...))

For example, `(split '(a b c d e)` should return `((a c e) . (b d))`.

(b) (15 points)

Transform your Scheme version of `split` into a MultiLisp program by adding `future`s where appropriate. Explain why you added each `future`. Do not add any `future` that does not contribute significant potential concurrency.

5. (a) (8 points)

What is a current continuation? What does the current continuation bound to `k` in the following call do?

```
(+ (* 2 5) (call/cc (lambda (k) (k 7))) )
```

(b) (25 points)

The following Scheme function prints an infinite sequence of integers starting at 1:

```
(define (GenInts)
     (let loop ((I 1))
          (display I)
          (loop (+ I 1))
     )
)
```

Harry Hacker decides to change `GenInts` into a coroutine that can be resumed when needed to produce the next integer in sequence. `GenInts` will return a pair: the desired integer and a continuation that can resume the co-routine:

```
(define (GenInts)
   (let loop ((I 1))
        (call/cc (lambda (k) (cons I k)))
        (loop (+ I 1))
   )
)
```
Unfortunately, Harry must have been dozing in class the day continuations were discussed because his code doesn’t work! What is wrong?
How must `GenInts` be changed to correctly operate as a coroutine?