CS 538 Midterm Exam

Friday, April 8, 1994

11:00 AM – 1:00 PM CDT

1240 CSST Bldg

Instructions

Answer *any four* questions. (If you answer all five, only the first four will count.) Each question is worth 25 points. 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!).

Question #1

(a) (12 points)
Because types are not specified at compile-time in Scheme, it is straightforward to *overload* an existing function so that it operates on additional types. Modify Scheme's + function so that it accepts functions as well as numbers as arguments. If both arguments to + are functions you may assume that the functions take two integers as arguments and return an integer result. If F and G are functions, (+ F G) should return a function that adds the results produced by F and G. Thus ((+ ∗ –) 10 5) should produce 55 = 10∗5 + 10–5. (Recall that procedure? tests whether a Scheme object is a procedure or function.)

(b) (13 points)
Most programming languages, including Scheme, pass parameters *positionally*. That is, the first value after the function is the first parameter, the next value is the second parameter, etc. For long parameter lists this approach is tedious and error-prone. It is easy to forget the exact order in which parameters must go. Moreover, default values for non-critical parameters are difficult to provide.

An alternative to positional parameters is *keyword parameters.* Now each parameter value is labeled with the name of the parameter it represents. The order in which parameters are passed is unimportant, and unnamed parameters may assume default values.

For example, assume F is a Scheme function that has five parameters, a to e. In keyword form (F 'a 10 'b 20) and (F 'b 20 'a 10) would both have the same effect, setting a to 10 and b to 20, with the remaining parameters assuming default values.

Assume we have function F, with parameters a to e, already coded in standard positional form. Explain how to modify F so that it now takes its parameters in keyword form. (Recall that (lambda L (body)) defines a function that takes a variable number of parameters, all bound to the list L.)

Question #2

(a) (10 points)
Let L be a list of the form (min v1 v2 ...) or (max v1 v2 ...), where v1, v2, etc., are all integers. Write a Scheme function (eval L) that evaluates L into a single integer representing the minimum (or maximum) of the integers that appear in the list. For example (eval ’(min 10 5 33 100)) = 5.

(b) (5 points)
Generalize your solution to part (a) so that min and max lists may nest. That is, now a value vi may be an integer or a min or max list. eval first reduces nested lists to integers, then reduces containing lists. Thus (eval ’(max 10 (min 10 5 33 100))) = 10. (Recall that integer? tests whether an object is an integer and list? tests whether an object is a list.)

(c) (10 points)
Generalize your solution to part (b) so that min and max lists may contain symbols (identifiers) as well as integers. As in algebra, symbols represent unknowns. Hence eval simply copies them to the answer list unchanged. A min or max list containing only symbols is returned unchanged; a min or max list containing only integers is reduced to an integer; a min or max list containing both integers and symbols is simplified so that all the integers in the list are reduced to a single minimum or maximum value; symbols are copied unchanged. Thus (eval ’(min 10 5 33 100)) = 5 (as before), (eval ’(min x y z)) = (min x y z), and (eval ’(min x y 10 z 20 30)) = (min x y z 10). (Recall that symbol? tests whether an object is a symbol.)

Question #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, of course, 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) (10 points)
Assume the test you devised for part (a) indicates that map *doesn't* apply its function in a strictly left-to-right order. Create a version of map that is guaranteed to apply its function to list arguments in a strictly left-to-right order. (Recall that Scheme *does not* guarantee that arguments to a function call are evaluated in any particular order.)

(c) (10 points)
MultiLisp is essentially Scheme with parallel constructs like pcall and future added. It is clear that calls to map can sometimes be slow to execute. Thus map is a good candidate for concurrent evaluation. Create a MultiLisp version of map that exploits MultiLisp's pcall and/or future features.

Question #4

(a) (5 points)
Explain what call/cc (also known as call–with–current–continuation) does.

(b) (10 points)
A common use of call/cc is to establish a *return point* so that a recursive function can return a value directly to its caller, bypassing pending intermediate calls.

Illustrate this use of call/cc by writing a function (min L) which calls the recursive function (min1 L return). min1 computes recursively the minimum value of the elements in list L. If it finds that an element of L is not a number, it immediately returns the string "Not defined" to min using return, which is a continuation. (Recall that number? tests whether an object is a number.)

(c) (10 points)
Another common use of call/cc is to suspend a computation and return a continuation to the caller. This continuation, when called, then resumes the suspended computation.

Assume we have an application that accesses a medical database, med–info–db. Some information in this database may be *locked*. The predicate locked? tests if a piece of data is locked. If it is, the data must be read using the function (unlock data password) where password is known only to the owner of data.

The following code fragment might appear in an application that accesses medical data:

```
(if  (locked? (lookup med-info-db info-wanted))
        (unlock (lookup med-info-db info-wanted) password)
        (lookup med-info-db info-wanted)
)
```

This code has the problem that password must always be provided, even if the info–wanted is not locked (e.g., your name or age).

Modify this code so that if a password is needed, a continuation is returned. This continuation, when given the necessary password, will resume execution by unlocking the data and finishing the computation. You may assume that the program has available a return continuation (as in part (b)), to make it easy to return values to the original caller.

Question #5

(a) (12 points)
The main advantage of the future construct in MultiLisp is that it allows a function call to execute in parallel with the function's caller. Thus in

```
        (let  (  (v  (future F arg))  )
;                                  Body of let
        )
```

the evaluation of (F arg) is concurrent with the execution of the body of the let until the value of v is needed.

Is it always safe and correct to allow the value of a let-variable to be computed in parallel with the body of the let as long as the let value isn't used? If so, explain why. If not, what restrictions must be placed on the expression defining the let variable and the body of the let construct so that they don't interfere?

(b) (13 points)
In C and Pascal the caller of a subroutine is suspended until the subroutine completes its execution. Assume we add a new statement to C or Pascal (your choice). This statement, the *parallel* statement indicates that when a subroutine call is made, the caller can resume execution immediately after the parameters are evaluated and passed. The caller need not wait for the called subroutine to complete. When the caller reaches its end, and is ready to return, it waits for any subroutine calls it started using parallel statements that haven't yet completed. Thus in

```
parallel sort(a,b);
while (i > 0) {
   ...
}
```

the sort subroutine executes concurrently with the while loop that follows it.

Is it always safe and correct to change subroutine calls into parallel subroutine calls? If so, explain why. If not, what restrictions must be placed on the subroutine called and its caller to guarantee that the two don't interfere.