CSE341—Midterm Exam

Instructor: Greg J. Badros — Summer 1999

23 July 1999

This mid-term exam is open book, open notes. You may use any of your class notes and your book. You may *not* use any of your neighbors' class notes or books. You may *not* use a laptop, a calculator, your neighbor, a cell phone, a pager, or any other computing or communication device. **Write your name on each page.** You will receive one point per page on which your name is clearly written, for 10 possible points. **Pick 6 of the 7 questions. Circle the numbers of the six questions you choose to answer.** Each of the 6 problems you choose to have us grade is worth 15 points, for 90 possible points. The exam altogether will be scored out of 100 points: 10 for your name, and 90 for your answers. Partial credit will be given, so please write down your thoughts even when you think you might not have a completely correct answer. **Be sure to clearly indicate final answers.** Please answer questions by writing directly on the exam. We cannot award points to answers that we cannot read, so please **write clearly**. Enough space is provided on the test pages for correct answers; however if you need more space, please use the back sides of the exam's pages. You have 60 minutes. **Good luck!**

Name:

Student ID:

1

Programmer Wanda B. Rich just implemented a real-time stock-trading system in Scheme that takes as input live data about stock prices from a satellite feed and continually updates a display of statistics about the stock values. After several days of running her system, she notices that there are occasionally brief interruptions during which her display seems to not be responding despite the new data flowing in from the satellite. Give a suggestion for what might be going wrong, and what Wanda's options are to correct the problem. (**15 pts**)

Her Scheme system is likely using a garbage collector that is using some variant of a mark-and-sweep collection algorithm. The delays are due to the stoppage of computation during the garbage collection.

There is little Wanda can do to her code to eliminate the outages. She could be more careful with wasteful allocations to increase the time between outages, perhaps. The better solutions are to find a different Scheme implementation that has a garbage collector that is better suited to a real time system. A reference-counting Scheme system would suffice if she didn't use many circular data structures (but few Scheme systems use reference counting). Alternatively, she could find a system that uses a more sophisticated incremental mark-and-sweep algorithm. She could also force garbage-collection to occur when the system is otherwise idle, and hope that the heap is large enough that another GC will not occur at an inopportune moment.

Another possibility is that she could rewrite the system in a language that requires explicit deallocation (and does not do automatic garbage collection). Many Scheme systems are simply not the best choice for real-time mission-critical software. □

2

Suppose we want to write a number-squaring procedure in Scheme that returns the square of its single argument, but also denotes the sign of that argument so the caller can tell whether it was a positive or a negative number that was squared. One possibility is to write the procedure like so:

```
(define (square-preserve-sign n)
  (if (< n 0)
      (number->string (* n n))
      (* n n)))
```

This procedure returns the square as a string if `n` is negative, or it returns the square as a number if `n` is non-negative. That is, returning a string value means that the argument was negative, while returning a numeric value means that the argument was positive. For example:

```
(square-preserve-sign 7) => 49
(square-preserve-sign -4) => "16"
```

Can one write a function that works like this (i.e., one that can return values of differing types) in C++? Why or why not? (**4 pts**)

No, C++ is statically typed and functions are required to be defined to return an explicit type. (In C++, we could work around this by making the various object types we want to be able to return have a common parent class, and then make the return type be either a reference or a pointer to an object of that parent class; other workarounds are possible, such as using `void*`, but all are ugly). □

Can one write a function that works like this in Haskell? Why or why not? (**4 pts**)

No, again, because Haskell is statically typed. Though Haskell's polymorphism lets us state the return type more flexibly than C++, returning different types when given the same type as input will be disallowed by the type checker. Polymorphism does *not* let us return multiple different values along different code paths of the same function. □

...question continues onto next page...

...question continued from prior page...

Consider rewriting the function as follows:

```
(define (square-preserve-sign-2 n)
  (list (* n n) (if (< n 0) 'negative 'positive)))
```

For example:

```
(square-preserve-sign-2 7) => (49 positive)
(square-preserve-sign-2 -4) => (16 negative)
```

Why might a Scheme programmer choose to write the function like this instead? (**7 pts**)

When she is invoking `square-preserve-sign-2`, she may often be interested in just using the squared value. It'd be awkward to have to use `string?` to test the type of the return value and convert it to a number when using `square-preserve-sign` if she only wants the squared value. Using `square-preserve-sign-2`, she could just use `car` on the returned list:

```
(car (square-preserve-sign-2 -4))
```

Similarly, if she just wanted the sign of the argument, she could just use the `cadr`.

Also, notice that the new version of the function is statically type checkable. This will make the function easier to re-write in other languages (such as Haskell, or C++). Finally, the new function has less code duplication, which is always a good thing (but not the most important aspect of this example). □

3

In Scheme, there are several ways that we can make bindings of variable names to values. The most common *explicit* technique involves using the `let` special form. However, bindings between actual arguments and procedure formal parameters are made implicitly as part of procedure application. Rewrite this expression that uses `let`:

```
;; assume x2, x1, y2, and y1 all have values from elsewhere
;; evaluates to the square of the Euclidean distance from (x1,y1) to (x2,y2)
(let ([dx (- x2 x1)]
      [dy (- y2 y1)])
  (+ (* dx dx) (* dy dy)))
```

into an equivalent expression that performs the binding using `lambda`. Your answer should preserve the expression:

```
(+ (* dx dx) (* dy dy))
```

exactly as it appears above and should use a `lambda` and procedure application to make the bindings from `dx` to `(- x2 x1)` and from `dy` to `(- y2 y1)`. (**15 pts**)

([lambda (dx dy) (+ (* dx dx) (* dy dy))] (- x2 x1) (- y2 y1))

Remember, the implicit bindings in procedure invocation are made in binding the formal parameters to the actual arguments. Thus, to bind `dx` and `dy` to values, they need to be the formals of the anonymous procedure we use to do the binding. To pick what they are bound to, we pass the function the arguments that we want those formals to be bound to. Specifically, we pass the expressions `(- x2 x1)` and `(- y2 y1)`.

The reason for the `let` special form is to make the bindings more clear–in the above code snippet it's harder to see that `dx` is getting bound to `(- x2 x1)`, etc., because they are not as close to one another in the code. □

4

In Haskell, we saw how a function could be specialized by partial application—by applying the function to only a leading subset of its arguments. For example,

```
mult :: Integer -> Integer -> Integer
mult x y = x * y

mult_by_5 :: Integer -> Integer
mult_by_5 = mult 5
```

Scheme can also perform this partial application if you give it a little help. Specifically, this question asks you to to write a function, `p-apply`, that works just like the built-in Haskell counterpart. `p-apply` must take an arbitrary number of arguments. The first argument to `p-apply` is a procedure, `proc`, that takes n arguments. The next k arguments to `p-apply` are values to use for the first k arguments of the procedure `proc` (assume $k < n$). `p-apply` returns a new procedure that takes $(n - k)$ arguments and operates as `proc` does except the first k arguments are bound already and no longer passed as arguments. For example:

```
;; mult take two arguments
(define (mult x y) (* x y))

;; p-app returns a new procedure that has only the first argument, x,
;; of mult already bound to the value 5
(define mult-by-5 (p-apply mult 5))
mult-by-5 => #<procedure>  ;; this procedure takes only one argument

(mult-by-5 3) => 15

(define (sum4 w x y z) (+ w x y z))

((p-apply sum4 1 2 3) 4) => 10
```

...question continues onto next page...

...question continued from prior page...

Write the Scheme procedure `p-apply` using the template below. (Hint: Consider using `apply` and `append` in your solution.) (**10 pts**)

```
;; remember the "." notation:
;; args, below, is made to be a list of all the arguments
;; to p-apply except the first (that one is bound to proc)
(define (p-apply proc . args)


   (lambda rest (apply proc (append args rest)))


    )
```

Define a Scheme procedure `squareList` as a specialization of `map` created via your `p-apply` procedure. (**5 pts**)

(define squareList (p-apply map (lambda (x) (* x x))))

Notice that you do not have to define the function `squareList` using function definition syntax. It is sufficient to simply bind `squareList` to the result of the `p-apply` expression because that explicitly returns a value that is a procedure that takes one argument, a list (since the arguments to `map` are a procedure, which we already bound, and a list). Compare the above to:

(define (squareList items) (map (lambda (x) (* x x)) items))

which does not use partial application at all.

5

An association list (or "a-list" for short) in Scheme is a list of `cons` pairs where the `car` of each pair in the list is often thought of as a key for the `cdr`, its value. E.g., the below creates an a-list and names it `x`:

```
(define x '((greg . 37) (ken . 13) (sean . 19) (chun . 6)))
```

This a-list associates each name to that person's lucky number.

Various Scheme procedures operate on a-lists. For example, `assoc-ref` returns the associated value when given an association-list and a key:

```
(assoc-ref x 'ken) => 13
(assoc-ref x 'sally) => #f
```

Can we use a-lists in Haskell? That is, can we define a Haskell function that operates on the same kind of lists? If not, explain why not and suggest an alternative representation for associating key and value pairs that is more appropriate for use in Haskell. (**5 pts**)

No. Haskell requires lists to be homogeneous and proper; a-lists in Scheme are (in the general case, and in the specific example above) heterogenous improper lists.

For Haskell, it makes more sense to use 2-tuples (i.e., pairs) in a list using strings instead of symbols for the names. Tuples can be heterogeneous, and we don't lose anything by requiring each tuple to contain exactly two elements (since the point of the tuples is to store a relationship between exactly two items). (Another tenable, but less-ideal, possibility is to use parallel lists; the problem with parallel lists is that there is no single object that contains the mapping.)

The above would then be represented as:

```
[("greg",37), ("ken",13), ("sean",19), ("chun",6)]
```

Write `assoc_ref` in Haskell, and give its type below. (**10 pts**)

```
assoc_ref ::
```

```
assoc_ref :: Eq k => [(k,v)] -> k -> v
assoc_ref [] _ = error "Item not found"
assoc_ref ((k,v):pairs) m = if k == m then v else assoc_ref pairs m
```

Note that you cannot use `k` twice in the pattern in Haskell, but instead must use the `if` expression in the definition. Prolog pattern matching is a bit more expressive than Haskell's in this respect.

6

Write Prolog rules for a predicate `last_element` as specified in the below comment: (**10 pts**)

```
/* last_element(X,Y) succeeds iff Y is the last element of the
   list X.

   E.g.,
   last_element([7,2,4],9)  fails
   last_element([7,2,4],4)  succeeds
 */
```

```
last_element([X],X).
last_element([_|Xs],Y) :- last_element(Xs,Y).
```

Give at least two specific ways that the `last_element` relation (and a Prolog predicate in general) differs from a Haskell or Scheme function that takes a list as an argument and returns that list's last element. (**5 pts**)

Prolog relations take their "return value" as an argument. Also, Prolog relations can often work "backwards" to compute, e.g., the list from the last element. Also, Prolog relations can generate multiple solutions, unlike a function that computes a single unique value. □

7

Consider the Prolog relation sum_to defined (with a subtle bug) below:

```
/* sum_to(N,Res) succeeds iff the sum of the first
   N counting numbers (i.e., 1,2,3,...,N) is Res. */

sum_to(N,1) :- N =< 1.
sum_to(N,Res) :-  N1 is N-1,  sum_to(N1,Res1),  Res is Res1 + N.
```

The query:

```
sum_to(3,R).
```

succeeds and gives R=6. If we then reject that solution, Prolog generates another solution, R=7. Explain why it is responding with an erroneous solution (**4 pts**).

After rejecting the correct solution, backtracking occurs and alternative ways to prove are tried. Prolog proves that, e.g., the relation `sum_to(0,1)` holds. Those sub-goal solutions then result in alternate and erroneous final solutions.

Correct the above definition of sum_to to fix the error (you should just edit the rules appearing above). (**7 pts**)

Inserting a single cut in any of lots of places works:

```
sum_to(N,1) :- N =< 1, !.
sum_to(N,Res) :-  N1 is N-1,  sum_to(N1,Res1),  Res is Res1 + N.

sum_to(N,1) :- N =< 1.
sum_to(N,Res) :-  N1 is N-1,  sum_to(N1,Res1), !,  Res is Res1 + N.

sum_to(N,1) :- N =< 1.
sum_to(N,Res) :-  N1 is N-1,  sum_to(N1,Res1),  Res is Res1 + N, !.
```

Using either of the corrected and uncorrected version of sum_to, above, the query:

```
sum_to(N,6).
```

gives the warning, "Arguments are not sufficiently instantiated." What should the answer be from a mathematical perspective and why is Prolog unable to find that solution? (**4 pts**)

The correct answer mathematically is $N = 3$. In Prolog, arithmetic does not work backwards (unlike in CLP(R), where it does work backwards to some extent). To try to satisfy the query, Prolog tries first tries satisfying the sub-goals of the second sum_to rule and gets `N1 is N-1` where `N` is uninstantiated. In Prolog, evaluation of arithmetic expressions (as forced by the "is" operator), can only be performed on expressions with all variables instantiated. □