Department of Computer Science and Engineering
University of California at San Diego

CSE 230
Winter 1998

Problem Set 6 Solutions

1. Explain the semantics of functions and macros in Common Lisp, as described in pp. 9–20, 82–94, 106-111, and 118–124 of *On Lisp: Advanced Techniques for Common Lisp* by Paul Graham (Prentice-Hall, 1994).

   The Lisp operators that you should explain include `#'`, `lambda`, and `funcall`, the comma `,` and backquote `` ` ``, and `destructuring-bind`. The issues that you should discuss include closures and lexical scoping, how macros have "two layers" of evaluation, variable capture and how to prevent it, and whether the difference between interpretation and compilation is visible to the programmer.

   In your answer, try to be precise but general, and concise but explicit. Use correspondences with ML and concepts discussed in class as much as possible.

   [Solution due to David Hessler.] Lisp has three namespaces, one each for variables, functions, and macros. The namespace chosen for looking up an identifier (an atom) depends on the context of the appearance of the identifier. Consider this simple example:

   ```
   (+ a b)
   ```

   When executing an expression represented by a list such as this, Lisp considers the first entry to be a function, and locates it in the function namespace. The remaining list entries are considered to be variable parameters and are found in the variable namespace. This sort of assumption can be unfortunate when a function is to be used as a parameter.

   Fortunately, Lisp has a built-in operator (it ought to—it seems to have a built-in operator for everything), `#'`, that locates the function named by its parameter and returns the actual value of the function. Now a function can be used anonymously. The hash-quote operator can be seen as having the effect of removing the name from a function, leaving just the function.

   All of this is a sharp contrast with ML, in which there is a single namespace in which identifiers representing both functions and other variables are located.

   Doing and undoing come together in pairs. The hash-quote operator has a double, the `funcall` operator. If we take the function returned by hash-quote and try to execute it, we cannot. Lisp now sees it as a common variable to which execution is inapplicable.

   The `funcall` function assures Lisp that the first parameter is indeed a function and executes this function with the remaining parameters as parameters. Thus,

   ```
   (funcall #'+ 1 2)
   ```

adds two numbers.

Lisp, like ML, has a function for defining new functions, and both have a method for creating unnamed functions. In Lisp, an unnamed function is created by another operator, `lambda`. As part of creating a function, `lambda` names variables to be bound to parameter values when the function is executed:

```
(lambda (a b c) (* (+ a b) c))
```

Any values bound to `a`, `b`, or `c` outside of this function are hidden during the function's execution. We see the same effect in ML.

So how is a function called in Lisp? It appears to be like most other languages: the parameter expressions are evaluated; the resulting values are bound to the parameter names as named in the function definition, and the function expression itself is evaluated. Finally, the parameter name bindings are removed and the value to which the function evaluated is returned.

Common Lisp uses lexical scoping. Lexical scoping requires that when a function refers to variables outside its own scope, the outside variable bindings must be kept with the function. This of course is a closure. This closure is different from ML's but like that of imperative languages. In Lisp, the variable is bound; in ML the value is bound. A Lisp function could refer to a variable that could be changed from outside the function.

Lisp has a class of constructs that output actual Lisp code. These are of course macros. As with functions, Lisp has many built-in, and allows the programmer to define new ones. A macro has parameters like a function, but where a function call causes runtime execution of the function body, a macro call is replaced at compile-time by its expansion: parameters are substituted into the macro expression, and the resulting expression is left in place of the call. The expression is evaluated as usual at run-time.

Some might say that a macro call has two phases—expansion and execution. I say that a macro call has one phase: expansion. Any execution is done on the resulting expression. By that time the macro is gone.

Macro expansion is recursive. Any macros revealed by the expansion are also expanded until no more macro calls remain. Though the result might appear to be the same, there are important differences between macros and functions. By run-time, any macro calls in a program are gone. Redefinition of a macro within a program will not change any calls to that macro that precede the redefinition. Macro calls may be recursive, but the functionality embodied by a function can not be, not without potentially unbounded expansion.

Macros come with a big "when does it get evaluated?" headache. There may be parts of a macro that are to be evaluated as the macro is defined. Other parts should be left alone during the creation of the macro and evaluated only when the macro has been expanded. The backquote character ` in front of an expression defers interpretation of the expression until after the macro has been expanded. Thus,

```
`(a b c)
```

in a macro definition is not interpreted, even if `a`, `b`, and `c` have values to which they could be evaluated. The comma operator `,` undoes an enclosing backquote, so

```
`(a ,b c)
```

results in the actual argument for `b` being inserted at the time of the macro expansion.

One way to look at these two operators is as if they manipulate a "hold" count on expressions. Backquote increments the hold count; comma decrements it. Any expression with a positive hold count is not evaluated; if the hold count is zero, it is evauated. (Negative hold counts are not allowed.)

Macros, like functions, name parameters to work on them. Free variables within macros are different from those in functions. Free variables in functions are bound to whatever is available at the time the function is created. If something unexpected gets bound to, at least it will be the same thing every thime the function gets called. Not so with macros. Macros have a sort of dynamic scoping in that any free variables are bound when the macro is expanded. The same variable may be bound to something different every time the macro is expanded. This is capture. It is conceivable that this behavior could be harnessed and used in productive ways, but when it goes wrong, one joins the choir of the damned.

For all the cleverness of Lisp, it seems odd that the `avoid-variable-capture` macro has never been written. Instead, the programmer must avoid capture by hand. A rigorous protocol could be set forth to locate and remove instances of variable capture, but the best procedure is this: (1) write short macros; (2) do not use names in the body of a macro which do not appear in the parameter list.

Lisp has a sort of pattern matching not unlike that of ML. This pattern matching comes in the form of the `destructuring-bind` macro. Lisp has just one lowest-level data structure, the dotted pair, so application of this pattern matching function can appear just about anywhere. Given a topology with named nodes, a parameter dotted pair tree is squeezed in. The parts of the parameter corresponding to the named nodes are bound to the names at those nodes.

Lisp may be both interpreted and compiled. Can a program determine whether it is compiled or interpreted? First: who cares? Surely somebody does. Let's assume that some operation is expensive, and we wouldn't want to spend the time if the function were interpreted.

Testing the output of print is dicey since any implementation could print anything it wants, and compiledness might not be part of this information.

Since the function is the typical unit of compilation, a program could (attempt to) redefine a function in a program, and see if the redefinition affects the result of the functions execution. I suggest that this would not prove anything conclusive since the implementation might link the interpreter with the executable, and take over when things get too self- referential.

Timing the execution of a function might be the best bet because fast interpretation would get the same result as a slow compiled function. If it runs fast, it doesn't matter whether its compiled or not.

2. In lazy ML, the sieve of Eratosthenes algorithm for generating the prime numbers can be implemented as follows.

```
- fun map f l = if l=[] then []
                         else (f (hd l))::(map f (tl l));
- fun remove pred l =
      if l=[] then []
      else if pred (hd l) then remove pred (tl l)
      else (hd l)::(remove pred (tl l));

- fun multiple p = (fn x => (x rem p) = 0);

- val rec series = 2 :: map (fn x => x+1) series;

- fun sieve (p::rest) = p::(sieve (remove (multiple p) rest));
```

Make minimal changes to the code above to make it work in standard ML.

[Solution due to Adam Taylor.] The following code behaves analogously to the given code. In writing it, I used the discussion of streams in *Structure and Interpretation of Computer Programs* by Abelson and Sussman (pp. 242–292) as a guide. The basic idea is to implement a datatype, called a stream, which is analogous to a regular list, except that the evaluation of the second item of a cons-pair is delayed, and is truly evaluated only when needed. Once this has been done, we simply substitute calls to our analogous stream functions in the place of the usual list access and constructor functions (i.e. `hd`, `tl`, tt ::, and the empty list `[]`).

Delaying the evaluation of the second item of a stream cons-pair is done by "thunking" it, i.e. instead of a `StreamCons` being a tuple of a base type and a stream of the base type, as might be expected, it is instead a tuple of a base type and a function from the zero-tuple `unit` to a stream of the base type. Since ML uses static scoping, this business of wrapping a "nothing function" around the thing whose evaluation we wish to delay serves to save the current environment in a closure, and this is exactly what a 'real' thunk does in a language like Algol 60. The code should clarify any confusion.

```
datatype 'a stream = StreamEmpty |
                     StreamCons of 'a * (unit -> 'a stream);

exception NoStreamHeadOfStreamEmpty;

fun streamHead StreamEmpty = raise NoStreamHeadOfStreamEmpty
  | streamHead (StreamCons (h,t)) = h;

fun streamTail StreamEmpty = StreamEmpty
  | streamTail (StreamCons (h,t)) = (t) ();

fun isStreamEmpty StreamEmpty = true
  | isStreamEmpty (StreamCons (h,t)) = false;
```

```
fun streamMap f s =
  if isStreamEmpty s then
    StreamEmpty
  else
    StreamCons (f (streamHead s),
                fn () => streamMap f (streamTail s));

fun streamRemove pred s =
  if isStreamEmpty s then
    StreamEmpty
  else if pred (streamHead s) then
    streamRemove pred (streamTail s)
  else
    StreamCons (streamHead s,
                fn () => streamRemove pred (streamTail s));

fun multiple p = (fn x => (x rem p) = 0);

fun naturalsFrom n = StreamCons (n,fn () => naturalsFrom (n+1));

val series = naturalsFrom 2;

fun sieve StreamEmpty = StreamEmpty
  | sieve (StreamCons (p,rest)) =
      StreamCons (p,fn () =>
        sieve (streamRemove (multiple p) (rest ())));

val primes = sieve series;

fun primesAfterNth n =
  if (n = 0) then primes
  else streamTail (primesAfterNth (n-1));

fun nthPrime n = streamHead (primesAfterNth (n-1));
```

The transcript of an SML/NJ session which demonstrates the usage of these constructs follows.

```
- nthPrime 1;
val it = 2 : int
- nthPrime 2;
val it = 3 : int
- nthPrime 3;
val it = 5 : int
```

```
- nthPrime 4;
val it = 7 : int
- nthPrime 100;
val it = 541 : int
```

Some auxiliary functions let us look at many primes at once.

```
fun enumerate (l,h) =
  if (l>h) then nil
  else l::enumerate ((l+1),h);

fun printIntStream (s:int stream) =
  if isStreamEmpty s then nil
  else (print (streamHead s);
        print "\n";
        printIntStream (streamTail s));

- map nthPrime (enumerate (1,100));
val it = [2,3,5,7,11,13,17,19,23,29,31,37,...] : int list

- printIntStream primes;
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
...
```

There's more, of course, but in the interests of brevity...