CSc 372, Fall 2001
Final Examination Solutions

Problem 1: (7 points)

Write an Emacs Lisp function `change` that changes every letter in the current buffer to "`x`", and every decimal digit to "`#`".

```
(defun change ()
    (interactive)
    (goto-char 1)
    (let (ch)
        (while (not (eobp))
            (setq ch (char-after (point)))
            (delete-char 1)
            (cond
                ((letterp ch) (insert ?x))
                ((digitp ch) (insert ?#))
                (t (insert ch)))
            )
        )
)
```

Problem 2: (15 points—do only two of problems 2, 3, and 4)

Write a function `ruler`, bound to `ESC-R`, that inserts a "ruler" before the current line.

```
(defun ruler ()
    (interactive)
    (let ((w (1- (window-width))) (current (point)) start end)
        (forward-line 0)
        (setq start (point))
        (let ((n (/ w 10)) (i 1))
            (while (<= i n)
                (insert "         " (int-to-string i))
                (setq i (1+ i)))
            (insert "\n")
            (setq i 1)
            (while (<= i w)
                (insert (int-to-string (% i 10)))
                (setq i (1+ i)))
            (insert "\n")
            )
        (read-char)
        (delete-region start (point))
        (goto-char current)
        )
    )

(global-set-key "\eR" 'ruler)
```

Problem 3: (15 points—do only two of problems 2, 3, and 4)

Write an Emacs Lisp function `longest` that positions the cursor at the beginning of the longest line in the current buffer. If there are ties for the longest line, choose the first one.

```
(defun longest()
    (interactive)
    (goto-char 1)
    (let ((max -1) (last (point)) (maxpos 1) len)
        (while (not (eobp))
            (forward-line 1)
            (setq len (- (point) last))
            (cond
                ((> len max) (setq max len)
                             (setq maxpos last)))
            (setq last (point))
            )
        (goto-char maxpos)
        )
    )
```

Problem 4: (15 points—do only two of problems 2, 3, and 4)

Write an Emacs Lisp function `marquee` that displays a horizontally scrolling line of text.

```
(defun marquee (text)
    (interactive "sText? ")
    (get-empty-buffer "*m*")
    (switch-to-buffer "*m*")
    (insert-char ?\n (/ (window-height) 2))
    (setq pos (point))
    (insert text)
    (let (ch)
        (while t
            (goto-char pos)
            (setq ch (char-after pos))
            (delete-char 1)
            (end-of-line)
            (insert ch)
            (sit-for 0 70)
            )
        )
    )
```

Problem 5: (9 points)

Background: In Icon, built-in functions supply default values for omitted arguments when there's a reasonable default. Emacs Lisp is very inconsistent in this regard. For example, the count for `forward-line` is optional but the count for `delete-char` is required. The instructor believes that consistent use of reasonable defaults would be an improvement for Emacs Lisp.

Problem: Identify two more elements from ML, Icon, Prolog, and/or Java that would be improvements for Emacs Lisp. Briefly describe how Emacs Lisp would benefit from their inclusion. The elements may be from the same language or from different languages.

One thing that comes to mind for me is to add object-orientation so that instead of a big pile of functions we could have classes like `Buffer`, `Window`, etc. Buffer-specific operations might be methods of the `Buffer` class, for example. It is the case however, that a simpler module-based approach, somewhat like ML's structures, would provide sufficient grouping to make the task of finding functions more manageable.

For a second item, I think that Icon's idea of keywords—things like `&subject` and `&pos` might be useful. For example, imagine `&point`. To move forward one might say `&point +:= 1`. `&point := 1` would go to the beginning of the buffer and `&point := 0` would go to the end of the buffer. Instead of several functions to adjust the point, one would simply change the value of `&point`.

Problem 6: (6 points)

Write an ML function `rm_prefix(P, L)` that **ASSUMES** that P is a prefix of the list L and returns a copy of L with P removed.

```
fun rm_prefix([],L) = L
 |  rm_prefix(_::ps, _::xs) = rm_prefix(ps, xs);
```

Problem 7: (2 points)

Write an ML function `listeq(L1, L2)` that returns true if the lists `L1` and `L2` are identical and returns false otherwise. Style does matter on this problem.

```
fun listeq(L1, L2) = L1 = L2;
```

Problem 8: (8 points)

Write an Icon procedure `extsort(L)` that accepts a list of file names and sorts them by their extension.

```
procedure extsort(L)
    return swap(sort(swap(L)))
end

procedure swap(L)
    L2 := []
     every w := split(!L, ".") do
         put(L2, w[2] || "." || w[1])
    return L2
```

```
        end
```

Problem 9: (8 points)

Write a Prolog predicate `find(+N, [-A,-B,-C])` that produces all combinations of integers between 1 and `N` inclusive such that `N = A * B - C`.

```
find(N,[A,B,C]) :- iota(N,L), getone(A,L,R), getone(B,R,R2),
    getone(C, R2, _), N is A * B - C.
```

Problem 10: (10 points)

Pick one interesting element from any of the languages we have studied and, in the style of an e-mail note, describe that element to a colleague familiar only with Java programming. Be sure to talk about the syntactic form, the operation, and describe a practical usage of the element.

When describing the idea of this course to someone I often use `member` in Prolog as an example of an interesting way to do something in another language:

In Prolog I might define a *predicate* called `member` that tests for membership in a list. Here is the definition of member:

```
member(X, [X|_]).
member(X, [_|T]) :- member(X, T)
```

That says that `X` is a member of any list of which it is the first element or any list that it is a member of the tail. An interesting thing about Prolog is that because predicates describe relationships, we can use that same predicate to generate the elements in a list:

```
?- member(X, [1, 2, 3]).
X = 1? ;
X = 2? ;
X = 3?
```

Problem 11: (10 points)

Present an argument either for or against the prospect of a single language becoming dominant for the majority of software development. Here are some points to possibly consider: What major elements would a dominant language need to have? How might such a language come into existence? What are forces that would work in opposition to a language becoming dominant?

Before Java's quick rise to popularity I was more skeptical about the prospect of a single language being dominant but Java, with all its problems, has nonetheless transformed much of the programming landscape. I have to think that a language of comparable complexity that addresses Java's weaknesses might have a chance of becoming dominant across many, if not most application areas.

It seems that a language with a potential of dominance would have to come from academia or the open source community. Vendors have a bad track record of supporting ideas from other vendors, no matter how good they are. Ideas that emerge from neutral parties often have a greater likelihood of widespread adoption.

In terms of language characteristics, it seems that object-orientation, a conventional syntactic structure, automatic memory management, and the potential of efficient execution are all musts.

With regard to opposing forces, it seems to be the case that successful software systems grow and grow. A dominant language would probably grow and grow until at some point it became clear that a language with a subset of the capabilities would serve just as well in many cases and perhaps excel in other areas. A crisp new language would perhaps attract those who like to be different and perhaps the cycle would repeat itself.

Problem 12: (10 points)

For five points each, answer TWO of the following questions. If more than two questions are answered, only the first answers on the paper will be graded.

(A) In languages like Java, C++, and ML, words like "if", "fun", and "class" are called *reserved words*—they can't be used for variable or routine names. The PL/I language has no reserved words. One can write statements like `if = while + then(else)`. What are some advantages and/or disadvantages to having no reserved words? Disregard issues related to compiling languages with no reserved words.

A big advantage is that you don't have work around reserved words when choosing variable names. I've many times wanted to use the variable "`to`" in an Icon program but that conflicts with `to-by`.

In terms of negatives, there is the potential for writing code like the above, but that's rarely a problem in practice. Also, it's difficult to write ad-hoc source code analysis tools to deal with programs written in such languages.

(B) Icon has several polymorphic operators and functions. For example, the unary `*` operator returns the number of elements in an object. The `delete` function removes elements from sets and tables. What are some tradeoffs that a language designer must take into account when considering inclusion of a polymorphic operation in a language?

There can be a tendency to have a polymorphic operator cover too much ground. For example, should Icon's `!` operator handle numbers as well? Should the `delete` function operate on lists as well? There can also be a problem when it comes to naming an operation but using an operator instead of a function is one way to avoid the question of what-to-call-it.

(C) Imagine a language that has heterogeneous lists like Prolog, Icon, and Lisp, but that also has a tuple type that is very similar to ML. Present an argument either for or against the claim that if a language has heterogeneous lists, tuples provide no additional benefit.

Python is a language that has heterogenous lists as well as tuples. Python's tuples are immutable—they can't be changed—and in *Learning Python* that immutability is cited as the best reason for the co-existence of the two mechanisms. I have used Icon's lists as tuples in various situations and there is something that feels a little bit uncomfortable about it, but I can't put my finger on it—maybe it is mutability.

(D) What do Java's exception handling mechanism and Icon's failure mechanism have in common?

The common element is the situation of an expression for which there's no reasonable result. In Icon an out of bounds array index fails; in Java an exception is thrown. Exceptions are far more general but much more cumbersome to use. It would be nice to have a mechanism with the generality of exceptions and Icon's conciseness of expression.

(E) The instructor said that "every programmer should have a language like Icon in their toolbox". What are the characteristics of Icon, and similar languages, that make it worthwhile to know such a language in addition to mainstream languages like Java and C++? (Or, argue that knowing a single language like Java or C++, and knowing it well, is all that's needed.)

The common elements that I see are (1) simple I/O, (2) good string handling, (3) data types like lists and tables, and (4) a library that's sufficiently well-designed that you don't need to look up the arguments for every other function.

(F) A language implemented in C, such as Emacs Lisp, typically has some library functions coded in C and the balance coded in the language itself. What factors influence whether a function is implemented in the language itself?

One way to think about built-in functions is which can be implemented in the language and which cannot. For example, there's no way to exactly implement Icon's `push/pop/pull/`... functions with the other mechanisms of the language—you can't write those functions in Icon.

Another consideration is efficiency. It may possible to implement a function in Lisp, but if the function is long-running and heavily used, it might be better written in C.

A third issue is complexity. A function may be sufficiently complicated that it is impractical to write in the implementation language.

EXTRA CREDIT SECTION (one point each)

(a) What's the difference between a hack and programming technique?

When you do it, it's a programming technique. When somebody else does it, it's a hack.

(b) Order these languages from oldest to youngest: Icon, Java, Lisp, ML, Prolog.

Lisp, Prolog, ML, Icon, Java

(c) Write an expression that is valid in three of the languages we studied but with a notably different interpretation in each.

Good question!

(d) As of midnight on December 12, how many messages have been sent to the CSc 372 mailing list? (Your answer must be within 10%.)

280 messages

(e) The instructor has immediate plans to rewrite the Icon assignments in another language to see how that language compares to Icon. What's the language?

Python

(f) What is the type of `rm_prefix` (problem 6)?

```
'a list * 'b list -> 'b list
```