Com Sci 221 — Programming Languages
Final Exam, Autumn 2000

Monday, 4 December, 1:30 PM - 3:30 PM
Ryerson Annex 276
4 problems on pages 2–7

- You may use the textbook, my class materials, any other published material, and notes of your own.
- Each numbered problem is weighted equally for grading purposes.
- You may use whatever sort of paper you find convenient, but please make your answers legible, and write your name on every sheet.

Problem 1: Datatypes

Let **kbd_char** be a datatype containing the normal visible characters (letters, numbers, punctuation) and space character on a typewriter keyboard (the character type in most programming languages contains additional stuff that I don't want to include). Suppose that you want to represent typed text that is broken up into lines and pages.

Logically, such text has the type $\mathbf{textl} = ((\mathbf{kbd\_char}^*)^*)^*$, that is lists of pages, which are lists of lines, which are lists of keyboard characters. In *ML* notation, this is defined by

$$\begin{array}{llll} \mathbf{datatype} & \mathit{textl} & = & \mathit{kbd\_char\ list\ list\ list} \\ & & ; & \end{array}$$

A common implementation of typed text uses two special marks to indicate the ends of each line (*eol*) and the end of each page (*eop*). The type of lists of characters with interspersed marks is $\mathbf{textr} = (\mathbf{kbd\_char} + \{\mathit{eol}, \mathit{eop}\})^*$. In *ML* notation, this is

$$\begin{array}{llll} \mathbf{datatype} & \mathit{textrelt} & = & \mathit{letter}\ \mathbf{of}\ \mathit{kbd\_char} \\ & & \mid & \mathit{eop} \\ & & \mid & \mathit{eol} \\ & & ; & \\ \mathbf{datatype} & \mathit{textr} & = & \mathit{textrelt\ list} \\ & & ; & \end{array}$$

continued on next page

1. Encode the following members of $((\mathbf{kbd\_char}^*)^*)^*$ in $(\mathbf{kbd\_char} + \{eol, eop\})^*$. Be careful: *eol* and *eop* mark the *end* of each line and each page. You will get something different and wrong if you mark the breaks between lines and pages, or the beginnings.

   (a) $[\ ]$ (empty text)

   (b) $[\,[\ ]\,]$ (text containing empty page)

   (c) $[\,[\,[\ ]\,]\,]$ (text containing one page containing empty line)

   (d) $[\,[\ ],\ [\,[\ ]\,]\,]$ (text containing two pages, one empty, one containing empty line)

   (e) $[\,[\,[a,b,c],\ [d,e,f]\,],\ [\,[g,h,i],\ [j,k,l]\,]\,]$ (two pages each containing two lines each containing three characters)

   (f) $[\,[\,[a,b,c],\ [d,e,f]\,],\ [\,[g,h,i],\ [j,k,l],\ [\ ]\,],\ [\ ]\,]$

2. Give a brief but precise characterization of the members of $(\mathbf{kbd\_char} + \{eol, eop\})^*$ that are *not* used to represent members of $((\mathbf{kbd\_char}^*)^*)^*$.

Problem 2: Loop invariant

The solution to this problem is similar to the clever exponentiation program that we developed in class, but slightly trickier. Be *very* careful with the arithmetic reasoning. It is all simple elementary high-school algebra, but if you make a guess instead of doing the calculation, you are *very* likely to be wrong. All values in this problem are integers.

Consider a rather peculiar function f mapping two nonnegative integer arguments to a nonnegative integer result, defined recursively by

$$f(m,n) = \begin{cases} 1 & \text{if } m,n = 0 \\ f(m-1, m*2)+1 & \text{if } m > 0 \text{ and } n = 0 \\ f(m, n-1)*2+1 & \text{if } n > 0 \end{cases}$$

Even though the recursive definition above is *not* a tail recursion, there is a simple iterative way to compute values of f using a single **while** loop with initialization of some variables before the **while** loop. Let m_0 and n_0 be given nonnegative integer values. Use the invariant

$$f(m,n)*i+j = f(m_0,n_0) \text{ and } m \geq 0 \text{ and } n \geq 0$$

to develop a simple iterative program (a small number of assignments to initialize m, n, i, j, followed by a single **while** loop containing a single **if-then-else** command) computing $f(m_0, n_0)$. You are not required to prove that your program satisfies the invariant. What very simple combination of i and j gives the value of $f(m_0, n_0)$ when your loop terminates?

Construct your answer by filling in the blanks on the next page. Some of the assignments are redundant ones of the form $x := x$. I ordered the assignments carefully so that you don't need any temporary variables.

continued on next page

```
n := ______
m := ______
j := ______
i := ______
while m > 0 or n > 0 do
    if n = 0 then
        n := ______
        m := ______
        j := ______
        i := ______
    else
        n := ______
        m := ______
        j := ______
        i := ______
    fi
od
output i ___ j
```

Problem 3: Higher-order programming

Recall the recursive definition of the base-2 exponential function:

```
(define exp2
        (lambda (i)
            (if (< i 1)
                        1
                        (* 2 (exp2 (- i 1)))
            )
        )
)
```

I defined `exp2` as a `lambda` expression, instead of using the more convenient form of *Scheme* definition—`(define (exp2 i) ... )`—because it makes your work below slightly easier.

In the spirit of defining control structures as higher-order functions, here is the mother of all recursions for one-argument functions:

```
(define (recurse rbody)
        (letrec ((rfunc
                  (lambda (i)
                      ((rbody rfunc) i)
                  )
                ))
                rfunc
        )
)
```

When `rbody` is a higher-order function representing the *body* of a recursive definition, `(recurse rbody)` is the function of one argument produced by the recursive definition.

Define a higher-order function `exp2rbody`, representing the body of the recursive definition of `exp2`, so that the alternate definition of base-2 exponentiation:

```
(define nexp2 (recurse exp2rbody))
```

makes `nexp2` work just like `exp2`. The *body* of the recursive definition of `exp2` is the part starting with `(lambda (i) ...` , but you have to think about the form `(rbody rfunc)` in the definition of `recurse` in order to package it just right into `exp2rbody`. (Because `recurse` deals with pure functions rather than program fragments, the *state* parameter that enters into many of the exercises is not involved here.)

Problem 4: Conditional vs. pattern-matching in *ML*

Here is an attempt to define conditional in *ML*, using pattern-matching instead of a built-in conditional function.

$$\begin{array}{llllll} \textbf{fun} & cond & \textbf{true} & x & y & = & x \\ \mid & cond & \textbf{false} & x & y & = & y \\ ; \end{array}$$

1. What type does *ML* infer for *cond*?

2. What is the difference in results between the two following forms?

   $$cond\ c\ (x\ z)\ (y\ z)$$

   and

   $$cond\ c\ x\ y\ z$$

   What sorts of values for x, y, z will make them the same, vs. different?

3. Explain *very* briefly (one sentence) why the definition of *cond* above is not satisfactory.