The University of British Columbia
CPSC 124 – Midterm

Friday, May 23, 2003
Time: 60 min

Instructor: Wolfgang Heidrich

Student's name:____________________________

Student number:____________________________

Signature:____________________________

This examination consists of 8 pages, including this cover page. **Check to ensure that this exam is complete**.

This is a closed book examination. The weight of each question is given in parentheses. The total number of marks is 60. Start with the questions you think are easiest, and then go back and do the harder ones. Show all your work. Good Luck!

1. **Each candidate should be prepared to produce his/her student ID on request.**
2. No candidate shall be permitted to enter the room after the expiration of one half hour, or to leave during the first half hour of the examination.
3. No candidate shall be permitted to ask questions of the invigilators, except in the cases of supposed errors or ambiguities in the examination questions.
4. One single sided page of handwritten notes is allowed.
5. Candidates suspected of any of the following, or similar, dishonest practices shall be immediately dismissed from the examination and shall be liable to disciplinary action.
    - Having at the place of writing communication devices, any books, papers or memoranda, calculators, audio or visual cassette players, or other memory aid devices other than the one handwritten page..
    - Speaking or communicating with other candidates.
    - Purposely exposing written papers to the view of other candidates. The plea of accident or forgetfulness shall not be received.

Q1 (10)	Q2 (12)	Q3 (11)	Q5 (11)	Q5 (16)	Total

1 Facts of Scheme (10 pts)

Answer the following questions as true or false **and provide brief explanations why**.

1. `lambda` is a special form [✓ **true**] **false**

   A: neither the parameter list nor the body of the lambda form are evaluated.

2. `char?` is a special form **true** [✓ **false**]

   A: char? is a built-in function, but it is evaluated just like any other regular form.

3. When a procedure is passed as a parameter, its body is evaluated first **true** [✓ **false**]

   A: the lambda form specifying the procedure passed as an argument is evaluated, but its value is a procedure - the body can only be evaluated after the procedure itself is called with actual arguments.

4. Scheme never passes unevaluated forms as actual parameters in procedure calls [✓ **true**] **false**

   A: According to The Rules, all arguments of a function are evaluated before the body of the function is.

5. If a procedure has a formal parameter named the same as a variable bound in the environment, then the value of the variable in the environment will be used in the evaluation of the body. I.e.

   ```
   (define h 10)
   (define f (lambda (h) (* 2 h)))
   (f 5)
   ```

   will return **20**. **true** [✓ **false**]

   The parameter h of the function hides the global variable h in the body of f. The true value returned is 10.

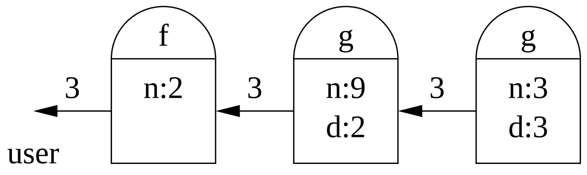
2 Recursion (12 pts)

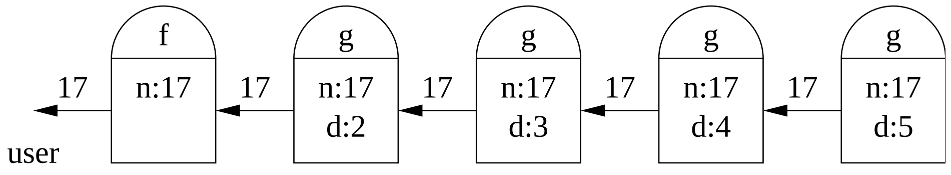
Consider the following code, where the predicate `devides?` returns whether the first argument is a divisor of the second one (i.e. the second argument is a multiple of the first).

```
(define f
  (lambda (n) (g n 2)))

(define g
  (lambda (n d)
    (if (> (* d d) n)
        n
        (if (divides? d n)
            d
            (g n (add1 d))))))

(f 9)
(f 17)
```

a) What is the result of the last two forms (you may want to draw the droids to get partial marks in case of a wrong answer)?

Solution:





b) What does `f` compute?
A: The smallest prime factor/divisor of n.

3 Procedures as Data Types - Derivatives (11 pts)

The derivative of a mathematical function $f(x)$ is the limit

$$f'(x) := \lim_{\epsilon \longrightarrow 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}.$$

as the denominator ϵ approaches 0. Therefore, for any function $f(x)$ we can *approximate* the true derivative at some point x by choosing a small value of ϵ and evaluating the formula $(f(x+\epsilon) - f(x))/\epsilon$.

Write a function `derivative` which takes two arguments (the function $f(x)$ and the value of ϵ) and returns the function that represents the approximation of the derivative of $f(x)$. For example

```
> (define f1 (lambda (x) x))
#proc...
> (define f1prime (derivative f1 0.1))
#proc...
> (f1prime 5)
1

> (define f2 (lambda (x) (* x x)))
#proc...
> (define f2prime (derivative f2 0.1))
#proc
> (f2prime 1)
2.1
```

Use the next page for writing down your solution.

Solution:

```
(define derivative
   (lambda (f epsilon)
      (lambda (x)
         (/ (- (f (+ x epsilon))
               (f x))
            epsilon))))
```

(Space for Problem 3)

4 String Manipulation (11 pts)

Sometimes you need to process strings of digits, such as `"104652"`, and compute the integer value that these strings represent. For this problem, you can assume that the digits $0 - 9$ have successive numerical codes (i.e. the code for 1 is one larger than the code for 0 and one smaller than the code for 2). This assumption is true for codes such as ASCII.

a) Write a function `digit->integer` which converts a character representing a digit into the integer value of that digit. For example, `(digit->integer #\5)` should return the integer 5 as a result.

Solution:

```
(define digit->integer
   (lambda (digit)
      (- (char->integer digit) (char->integer #\0))))
```

b) Using `digit->integer`, now write a function `string->integer` that takes a string of digits (you can assume that all characters in the string are actually digits!), and returns the corresponding integer value. For example `(string->integer "88734")` would return the value 88734, and `(string->integer "00123")` would return the value 123.

Solution:

```
(define string->integer
  (lambda (str)
     (si-helper str 0)))

; take first digit in string, convert it to integer
; multiply current accumulator by 10 and add digit
; then solve recursive problem with first character
; cut off string
(define si-helper
  (lambda (str acc)
     (if (string=? str "")
        acc
        (si-helper (substring str 1 (string-length str))
                   (+ (* acc 10) (digit->integer (string-ref str 0)))))))
```

5 The Fibonacci Sequence (16 pts)

The Fibonacci sequence is given as

$$F(n) := \begin{cases} 0 & ; n = 0 \\ 1 & ; n = 1 \\ F(n-2) + F(n-1) & ; n \geq 2 \end{cases}$$

This sequence pops up all over nature, for example in the spiral patterns of the seeds of sunflowers, or in the shape of snail shells.

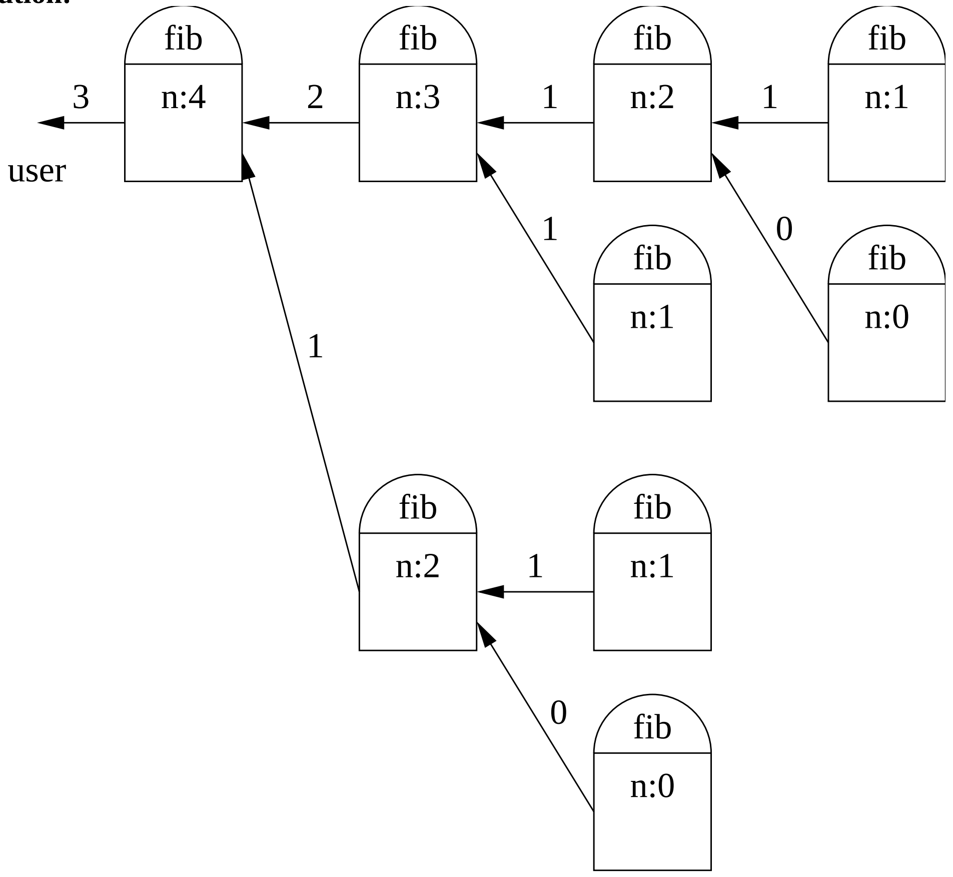
a) Directly translate the recursion formula from above into a Scheme function called `fib`.

Solution:

```
(define fib
   (lambda (n)
      ; two nested ifs work as well
      (if (or (= n 0) (= n 1))
          n
          (+ (fib (- n 2)) (fib (- n 1))))))
```

b) Write down the droids for the the function call `(fib 4).`

Solution:



c) What is the time complexity of this recursion algorithm? Hint: look at the number of droids require to compute `(fib 2)` vs. the number required for `(fib 3)`, `(fib 4)`, etc. In other words, if you increase `n` by one, how many more operations do you need?

Solution:

If we increase n by 1, we roughly double the number of operations required. Therefore, the computation of $F(n)$ with the above algorithm requires

$$\underbrace{2 \cdot 2 \cdots 2}_{n},$$

or $O(2^n)$ operations.

d) As you could see from the solution in part b), the problem with the straightforward implementation from part a) is that the same values are computed over and over again. Can you come up with a more efficient implementation of `fib` that avoid computing the same value more than once? Hint: you will need a helper function that takes additional parameters…

Solution:

```
; fib itself deals with n=0 and n=1 and
; calls a helper for all other cases
(define fib
   (lambda (n)
      ; two nested ifs work as well
      (if (or (= n 0) (= n 1))
          n
          ; n is now turned into a counter
          (fib-helper (- n 1) 0 1))))

; fib-helper deals only with cases starting at 2
; it maintains the current and the previous value
; of the sequence. On recursion, the current value
; becomes the previous one and the new current
; is computed according to the recursion formula.
(define fib-helper
   (lambda (count prev current)
      (if (= count 0)
          current
          (fib-helper (- count 1)
                      current
                      (+ prev current)))))
```