Homework 7: Abstracting Procedures

Due:

Testing for the problems in this homework is important, as it shows that you can not only write code like in the book, but that you understand how it works. It's best if you make up at least one or two test cases on your own. Be sure to include transcripts showing your test cases for all problems.

All the directions from homework 2 apply to this homework. In particular, remember to hand in a printout for procedures, and either a calculation or a transcript showing your testing.

We highly recommend that you put each problem in a separate file, and when making a transcript use the Scheme procedure `load` to read that file in to the Scheme interpreter.

1 Section 7.2 (except pp. 197–201), Procedures as Arguments and Values

1. (0 points; suggested practice) Read *The Little LISPer*, chapter 9, which is on reserve at the library. You can stop around the discussion of `curry-maker` (p. 151 in my copy), or whenever you start to see even more weird things than we've discussed in class.

2. (0 points; suggested practice only) Fill out a table of types as suggested in the "Chapter 7" section of the *Make Your Own Scheme Reference Card* handout.

3. (1 point; basic) Do exercise 7.1 on page 202. You can handwrite your answer.

4. (4 points; basic) Do exercise 7.2. For example,

   ```
   ((compose3 add1 add1 add1) 4) ==> 7
   ```

5. (5 points) Do exercise 7.4. You should not use the procedure `-`, but you may use `sub1` in your answer.

6. (9 points; basic) Do exercise 7.5. You should handwrite this. Clearly label each part of your answer.

   In addition, write why each answer is what it is, in your own words. If you use the computer, only use it to check your work, not to think for you.

7. (0 points; suggested practice only) Do as many of the following until you get the idea of how to use `map` to solve problems.

   Use `map` to solve exercises 2.16, 3.4, 4.3, and 4.4 of the book.

8. (10 points; basic) Do exercise 7.6.

9. (0 points; suggested practice) Do exercise 7.7. You should assume that the second argument, `ls`, is non-empty. Note that the last example returns `#t` as noted in the errata for the first printing.

10. (10 points) Define a procedure `red` of type

    ```
    all T . (T,T->T), (non-empty list of T) -> T
    ```

    that takes a procedure `proc` and a non-empty list `lst`. The behavior of `red` is characterized by the following equations.

$$(\texttt{red}\ p\ (\texttt{cons}\ x\ \texttt{'()})) = x$$
$$(\texttt{red}\ p\ (\texttt{cons}\ x\ (\texttt{cons}\ y\ l))) = (p\ x\ (\texttt{red}\ p\ (\texttt{cons}\ y\ l)))$$

For example,

```
(red - '(3)) ==> 3
(red - '(3 4 5)) = (- 3 (- 4 5)) = (- 3 -1) ==> 4
(red max '(3 -4 5)) ==> 5
(red (lambda (x y) (or x y)) '(#t #f #f #t)) ==> #t
```

11. (0 points; suggested practice only) Use **red**, from the previous problem, to solve exercise 3.1 of the book. You may assume that this exercise is modified to take a non-empty list as an argument.

12. (0 points; suggested practice) Do exercise 7.9.

13. (0 points; suggested practice) Do exercise 7.11 and check your answer on the computer.

14. (7 points; extra credit only) Do exercise 7.8.

15. (10 points; extra credit only) Define a procedure, `map-poly`, that is takes a procedure `f` and a polynomial `poly` as arguments, and returns a polynomial. The procedure `f` must have type, term $\rightarrow$ polynomial, where a term is a monomial. The result is obtained by taking the results of applying `f` to each term of `poly`, and adding the resulting polynomials. For example,

```
(map-poly (lambda (trm) (make-term (degree trm) (* 3 (leading-coef trm))))
          (poly-cons 3 12.0 (poly-cons 2 -5.1 (poly-cons 0 2.2 the-zero-poly))))
```

returns the polynomial $36.0x^3 + -15.3x^2 + 6.6$. Similarly

```
(map-poly (lambda (trm) (make-term (add1 (degree trm)) (+ 3 (leading-coef trm))))
          (poly-cons 3 12.0 (poly-cons 2 -5.1 (poly-cons 0 2.2 the-zero-poly))))
```

returns the polynomial $15.0x^4 + -2.1x^3 + 5.2x$.

Hint: you should pattern your answer after `map`. You may use `p+` in your solution.

2 Section 7.2, pp. 197 – 201, Unrestricted Lambda

16. (5 points) The Scheme procedure `min` can take any number of arguments. How is this defined? To see it, define a procedure `minimum` using `min` as if it only took exactly 2 arguments. For example,

```
(minimum 3) ==> 3
(minimum 3 -5 4 9) ==> -5
```

17. (5 points) Do exercise 7.3. As a base case, for all values x, `((compose-many)` x`)` $= x$; that is, `compose-many` with zero arguments returns the identity function.

18. (5 points; extra credit only) Alonzo Typed has noticed that for non-empty lists, the procedure `for-each` is more general than necessary. To make a procedure `for-each-non-empty`, he copies the code from `for-each`, changes the name, and then changes the test from `(null? ls)` to `(null? (cdr ls))`. But now `for-each-non-empty` doesn't work. How would you fix it, keeping the test as `(null? (cdr ls))`?

19. (10 points; extra credit) Do exercise 7.10. Note that the sentence following the faulty definition in the exercise should read as follows, with the bracketed words inserted: "This program, as written, is incorrect, because [we would like] the two invocations of *map* within the definition ...." Note that there are three parts to this exercise `map`, `ormap`, and the question at the end.

3 Section 7.3, Currying

20. (4 points; basic) Curry the procedure `expt` to get a procedure `expt-c`, and use that to define a procedure `power-of-2` that raises 2 to the exponent of its argument. For example,

    ```
    ((expt-c 2) 3) ==> 8
    ((expt-c 3) 2) ==> 9
    (power-of-2 5) ==> 32
    ```

21. (0 points; suggested practice only) Do exercise 7.13. Check your answer on the computer.

22. (0 points; suggested practice) Curry the procedure `poly-value` from Chapter 5 to get a procedure `poly-value-c` and use it to define the procedure `binary->decimal-c` that works like Program 5.17 on page 157 of the text.

23. (3 points; basic) Do exercise 7.14.

24. (5 points) Do exercise 7.15.

25. (0 points; suggested practice only) Do exercise 7.16. Check your answer on the computer.

26. (5 points; extra credit only) Do exercise 7.17.

27. (7 points) Do exercise 7.18.

28. (0 points; suggested practice only) Curry the procedures `map` and `for-each` in programs 7.1 and 7.2. Use your curried `map-c` procedure to solve exercises 2.16, 3.4, 4.3, and 4.4 of the book.

29. (8 points) Do exercise 7.19. You can handwrite your answers. Use the computer only to check your work, not to think for you.

30. (5 points; extra credit only) Do exercise 7.21.

31. (extra credit; points below) John Backus was the principal scientist on the team that invented FORTRAN. In 1978 the Association for Computing Machinery gave him its highest award, the Turing Award. In his influential Turing Award lecture, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs" (*Communications of the ACM*, Volume 21, Number 8, August 1978, pages 613–641), he gives reasons for abandoning languages like FORTRAN and for the kind of programming we have been doing in this chapter. (You might enjoy reading it; it's in the library.) In this problem you will explore the alternative language, called FP, which he proposed.

    One of the features of FP, in Scheme terms, is that every function takes one argument, which in the case of procedures like `+`, means that the argument is a list of 2 numbers.

    (a) (5 points; extra credit only) Write a procedure `fp-ize` that takes a Scheme function of 2 arguments, such as `+`, and returns the FP style function. For example,

    ```
    ((fp-ize +) '(3 4)) ==> 7
    ((fp-ize -) '(3 4)) ==> -1
    ```

    Define `fp-ize` so that the function it returns gives an error if it is applied to a list that is not exactly 2 elements long. Use `fp-ize` to define procedures `fp-plus`, `fp-times`, and `fp-minus` to be the FP versions of the Scheme arithmetic procedures.

    (b) (5 points; extra credit only) What is the FP version of the `compose` function?

    (c) (5 points; extra credit only) In FP, many of the functions are curried. For example, the reduction function, `red` is curried. But the function argument is an FP-style function. Thus `((red` *f*`) (list` *x* *y* ...`))` = `(`*f* `(list` *x* `((red` *f*`) (list` *y* ...`)))`. How would you write the FP version of the `red` function?

(d) (5 points; extra credit only) In FP Backus defined a function `constr` that takes a list of FP functions and turns it into a FP function that satisfies the following. `((constr (list` f_1 f_2 `...)) ` x`) = (list (`f_1 x`) (`f_2 x`) ...)`. How would you write the FP version of the `constr` function?

(e) (10 points; extra credit only) How would you write a matrix product procedure using FP conventions? For example,

```
(mat-prod '(((1 2 3) (4 5 6))
             ((7 8) (9 10) (11 12) (13 14))))
 ==> ((25 28) (73 82))
```

4 Section 7.4, Procedural Abstraction of Flat Recursion

32. (3 points; basic) Do exercise 7.22.
33. (5 points) Do exercise 7.23.
34. (5 points) Do exercise 7.24.
35. (0 points; suggested practice) Do exercise 7.25.

5 Section 7.5, Procedural Abstraction of Deep Recursion

Note that in this section, instead of the procedure `deep-recur` described in the book, you should use the procedure `deep-recur-abs`. The code for this is in `/home/cs227/lib/deep-recur-abs.ss`.

36. (20 points) Do exercises 7.26 and 7.27. You are to write a total of 4 procedure definitions for this problem as described in the exercises. Your definitions for 7.26 should look like the code we have been writing in class for deep recursion, instead of the code in the book. You should think of the procedure `remove-all-c` as a procedure that takes a symbol and a tree of symbols as its arguments. Use `deep-recur-abs`.
37. (10 points) Do exercise 7.28. Use `deep-recur-abs`.
38. (0 points; suggested practice only) Do exercise 7.29. Use `deep-recur-abs`.
39. (5 points) Do exercise 7.30. Use `deep-recur-abs`.
40. (5 points) Define `flat-recur` using `deep-recur-abs`.

6 Chapter 8, Sets and Relations

We are not going to cover chapter 8 in class this semester. But there are many interesting exercises in that chapter. Besides their intrinsic interest, these exercises will increase your mathematical sophistication, and thus may help you in math and computer classes (such as Com S 330). (You may want to do them when you take such a class.)

41. (0 points; suggested practice only) Read chapter 8. This will help you understand lambda.
42. (many points; extra credit only) Do one or more of the following exercises, for the extra credit listed. Your solutions should be clearly written with good style, clearly labeled, and clearly tested (to be easier on your TA). Note that you should, as you do these concentrate on the functional style described in the chapter, not just getting the right answer. 8.1 (4 points), 8.2 (4 points), 8.3 (5 points), 8.4 (8 points), 8.5 (10 points), 8.6 (5 points), 8.8 (10 points), 8.9 (4 points), 8.10 (5 points), 8.12 (10 points), 8.13 (1 point), 8.14 (2 points), 8.15 (5 points), 8.17 (5 points), 8.18 (5 points), 8.19 (5 points), 8.20 (10 points), 8.21 (5 points), 8.22 (5 points).