Fundamentals of Computer Science I (CS151.01 2006F)

Homework 6: Recursive Procedures

Assigned: Friday, 15 September 2006
Due: Tuesday, 19 September 2006
No extensions!

Summary: In this assignment, you will practice writing recursive procedures for a variety of tasks.

Purposes: To give you experience writing recursive procedures. To provide you with some feedback on the expected "style" of writing procedures.

Expected Time: One hour.

Collaboration: You may work in a group of any size between one and four, inclusive. You may consult others outside your group, provided you cite those others. You need only submit one assignment per group.

Submitting: Email me your work, using a subject of *CSC151 Homework 6*.

Warning: So that this exercise is a learning assignment for everyone, I may spend class time publicly critiquing your work.

Assignment

The focus of this assignment is a series of small problems, rather than a larger problem. Write procedures to solve each of the problems below. Since this is an exercise in writing procedures, and not in documenting procedures, you are not required to write documentation (although you may earn extra credit for writing good documentation).

Problem 1: A Better Sum

Write a procedure, `(sum values)`, that, given a list of values as a parameter, computes the same of all numeric values in the list. Your `sum` procedure should ignore all non-numeric values.

For example,

```
> (sum (list 1 2 3))
6
> (sum (list 3 'a 'b 5))
8
> (sum (list 'a 'b))
0
```

Problem 2: Reversing Lists

Suppose the `reverse` procedure were not included in Scheme. Could you write it yourself? Certainly! It should be possible to implement `reverse` recursively.

One strategy is to use the standard recursive formulation of

```
(define my-reverse
  (lambda (lst)
    (if (null? lst)
        base-case
        (combine (car lst) (my-reverse (cdr lst))))))
```

Finish this implementation.

When using this strategy, you'll need to think about the question "Suppose I've reversed the cdr of a list. What do I do with the car to get the reversal of the complete list?"

Problem 3: Reversing Lists, Revisited

Of course, you've seen more than one strategy for writing recursive procedures. Another possibility is to use a helper that includes a "what I've done so far" parameter. For example,

```
(define my-other-reverse
  (lambda (lst)
    (my-other-reverse-helper null lst)))

(define my-other-reverse-helper
  (lambda (reversed-so-far remaining-elements)
    (if (null? remaining-elements)
        base-case
        (my-other-reverse-helper (modify reversed-so-far)
                                 (cdr remaining-elements)))))
```

Finish this implementation.

Problem 4: Splitting Lists

Define and test a Scheme procedure, (unriffle *lst*), that takes a list as argument and returns a list of two lists, one comprising the elements in even-numbered positions in the given list, the other comprising the elements in odd-numbered-positions.

```
> (unriffle (list 'a 'b 'c 'd 'e 'f 'g 'h 'i))
((a c e g i) (b d f h))
> (unriffle (list))
(() ())
> (unriffle (list 'a))
((a) ())
> (unriffle (list 'b))
((b) ())
> (unriffle (list 'a 'b))
((a) (b))
```

Hint: One strategy is to define a separate helper procedure that takes the car of the given list and the result of the recursive call as its arguments and rearranges the pieces as necessary to obtain the final result.

Problem 5: Finding Skips

Write a procedure, (`find-first-skip` *lst*) that takes a list of symbols as a parameter and returns the index of the first instance of `skip` in *lst*, if `skip` appears in *lst*. If `skip` does not appear in *lst*, `find-first-skip` should return #f.

```
> (find-first-skip (list 'hop 'skip 'and 'jump))
1
> (find-first-skip (list 'skip 'hop 'jump 'skip 'and 'skip 'again))
0
> (find-first-skip (list 'hop 'to 'work 'jump 'to 'school 'but 'never 'skip 'class))
8
> (find-first-skip (list 'hop 'and 'jump))
#f
```

Problem 6: Finding Arbitrary Values

Write a procedure, (`index-of` *sym lst*) that takes a symbol and a list of symbols as a parameter and returns the index of the first instance of *sym* in *lst*, if the symbol appears in *lst*. If the symbol does not appear in *lst*, `index-of` should return #f.

```
> (index-of 'skip (list 'hop 'skip 'and 'jump))
1
> (index-of 'skippy (list 'hop 'skip 'and 'jump))
#f
> (index-of 'hop (list 'hop 'skip 'and 'jump))
0
> (index-of 'jump (list 'hop 'skip 'and 'jump))
3
> (index-of 'jump (list))
#f
```

Important Evaluation Criteria

Students who provide correct procedures for each question will earn a check.

Students who provide oddly formatted or inelegant solutions to the problems may be publicly critiqued for their odd formatting and inelegance, and will also receive a grade penalty.

Students who provide particularly elegant formatting or strategies will earn a higher grade.