Program 6: Scheme

How to set up

While your programs ultimately need to run on edoras, you may find it convenient to download DrRacket as a programming environment.

https://racket-lang.org/ and select Download. Instructions to get started are on the Racket site. "Racket" is the re-named "Scheme". Set the mode to "Scheme".

Using the edoras server, enter these commands if you haven't already made the grading directory.

```
edoras% mkdir -p ~/handin/p6
edoras% cp /tmp/pkraft/p6.scm   ~/handin/p6/p6.scm
```

This file p6.scm file (also available on Blackboard) already defines some lists with which you can experiment; all you need to do is add the definitions of four functions:

- evens
- oddrev
- middle
- bookends

Each of these functions should take exactly one parameter: the parameter must be a **list**. If your function was given the wrong number of parameters, we will let the interpreter complain about it. However, if the function is given the wrong **type** of parameter (i.e., not a list), the function should return a friendly error message (including a newline). Beginning with "USAGE: " followed by an indication of the correct invocation. For example,

```
USAGE: (evens {list})
```

Function definitions

EVENS:

`(evens lst)`

should return a new list, formed from the even-numbered elements taken from `lst`. That is,

`(evens '(a b c d e f g))`

should return:

`(b d f)`

`(evens (list 's 't 'u 'v 'w 'x 'y 'z))`

should return:

`(t v x z)`

`(evens '(f (a b) c (d e d) (b a) f))`

should return:

`((a b) (d e d) f)`

Both `(evens '())` and `(evens '(a))`

should return the empty list, etc.

Program 6: Scheme

ODDREV:

`(oddrev lst)`

should return a new list, formed from the odd-numbered elements taken from `lst`, but in the reverse of their original order. That is,

```
(oddrev '(a b c d e f g))
```
should return:
```
(g e c a)
```

```
(oddrev (list 's 't 'u 'v 'w 'x 'y 'z))
```
should return:
```
(y w u s)
```

```
(oddrev '((h i) (j k) l (m n)))
```
should return:
```
(l (h i))
```

```
(oddrev '())
```
should return the empty list, etc.

MIDDLE:

`(middle lst)`

should return a one-element list consisting of just the middle item of `lst`, [or return the empty list if there were an even number elements in `lst]`. That is,

```
(middle '(a b c d e f g))
```
should return:
```
(d)
```

```
(middle '(s t u v w x y z))
```
should return the empty list.

```
(middle '((a b) c (d e d) c (b a))
```
should return :
```
((d e d))
```

Note that `((d e d))` is a list containing the thing that was in the middle, which is itself a list.

BOOKENDS:

`(bookends lst)`

should return `#t` if the list `lst` if the first and last elements in the list are the same, and return `#f` otherwise. That is,

```
(bookends '(s t u v w x y z))
```
should return:
```
#f
```

Program 6: Scheme

```
(bookends (LIST 'j 'k 'l 'm 'n 'o 'j))
```
should return:
```
#t
```

Both `(bookends '())` and `(bookends '(a))`
should return `#t, etc.`

You can receive full credit for your `bookends` function as long as it works for all list that are made up entirely of atoms.

If you choose to take it further, you can try to get it to behave correctly for complex lists such as:
```
(bookends '((h i) j (k l k) j (h i)))
```
which should return `#t`. The following two lists should produce `#f`:
```
(bookends '((a b) c (d e d) c (b a))
(bookends '((y z) y))
```

In any case, you do not need to worry about what `bookends` should do with things that are not single characters like this string example:
```
(bookends '(abc))
```

Programming in Scheme

Forgetting a closing parenthesis, your functions will not work; you will get a 'premature EOF' error. It's critical that you use an editor that will match up parentheses. In vi, for example, if you position your cursor over a parenthesis and then type the '%' character, the vi cursor will jump to the matching parenthesis. If the cursor doesn't move, then you know you have a mismatch.

Grading considerations:

You must provide a functional programming solution in order to get any credit. Even though the variant of Scheme on edoras allows constants to be redefined, you must NOT store values. Therefore, all four of your functions must be defined recursively. See Sebesta Chapter 15 for examples. I've posted some examples, too.

It is okay to define "helper" functions if you find that convenient. Note: if you need a function, say, *reverse*, you must write it yourself and not use a built-in library function.

Your program will be checked against some predefined lists—some of these are defined in p6.scm. New lists will also be used, so think up some of your own to test your functions.

Weird caveat in the grading program: It won't attempt to do any grading unless you have *at least* defined the "`evens`" function into your p6.scm file. So, put a syntactically correct definition of the `evens` function in your p6.scm file early on (it doesn't have to work correctly, it merely has to exist and be "legal" Scheme code), or else you won't really have much of a clue as to what sort of tests the grading program will run.

Program 6: Scheme

In your program, be sure to put your name, have no additional input prompts, explain your algorithm, point out any failures that don't meet the requirements, give credit if you borrow code from other sources, etc.

What to hand in:

Put p6.scm in your **~/handin/p6/** directory. The autograder will retrieve your program, test it, and calculate a grade. Then your code will be inspected by a human grader: you can get no credit for an imperative solution and/or use variables, even if you get correct answers. You must use functional programming paradigms of recursion and functions.