CS 3L Summer 2011
Homework 5: Berkeley Bowling

Specs:
* 8 points
* Individual assignment: you may discuss the problem with others, but you may *not* share your code, work on the problems together at the same computer, etc. See the Course Info / Policies document.
* Due by 11:59:59 PM on Tuesday, July 19. Can be handed in late (for much less credit) until 11:59:59 PM on Thursday, July 21.
* Put all your code in a file called `hw5.scm`. Type `submit hw5` from the appropriate directory.

Some of you have probably been to Berkeley Bowl, the Berkeley grocery store that was once a bowling alley (really!) Now it's a place to compete with 10,000 other Berkeleyans to be the first to grab the most succulent heirloom tomatoes. In this tough economic climate, though, even Berkeley Bowl can probably use a little extra income, so let's suppose that they've started turning the store back into a bowling alley at night. Customers roll organic cantaloupes down the aisles, trying to knock down squashes that are roughly pin-shaped. For the purposes of this assignment, this "Berkeley Bowling" is **identical to regular bowling; it has exactly the same rules**. All the stuff about Berkeley Bowl and produce is just for flavor.

I. Bowling Rules and Scoring

For a game with such a simple premise (throw a ball at some pins and try to knock them down), bowling has a strangely complex scoring system.

* Each player gets ten "frames" (rounds) of bowling. In each "frame", you get up to two throws (possibly three, in the case of the tenth frame), and the goal is to knock down all ten pins. The results of your throws are recorded.
** If you knock down all ten pins on your first throw of a frame, it's called a strike, and you record a letter X for that frame. This immediately ends the frame (unless it's the tenth frame; see below).
** If you knock down 0 to 9 pins on your first throw of a frame and then knock down all the rest on your second throw, it's called a spare. You record these two throws as: the number of pins knocked down on the first throw, followed by a forward slash /. For example, if your first throw knocks over eight pins and your second throw takes out the remaining two, your throws for that

frame would be recorded as: 8/

** If your two throws in a frame fail to knock over all of the pins, the throws are recorded as the two numbers of pins you knocked down in the two frames, e.g., 70 would mean:

> In this frame, I knocked down 7 pins on the first throw, and then I knocked down 0 pins on the second throw. (That's seven-zero, not seventy!)

* The tenth and final frame is handled specially:
** If you get a strike on your first throw, you get a new set of ten pins and two "bonus throws". These act much like a normal frame, except that if you get another strike on your second bonus throw, then you get still another set of pins... but you still only have one bonus throw remaining to use on them. Even if that final bonus throw is a strike, it doesn't earn you any more bonus throws. Otherwise, you can get a spare or nothing on the two bonus throws, just as you would in a normal frame.
** If you get a spare in the tenth frame, then you get a new set of pins and a single bonus throw. You can earn a strike on this throw, but it doesn't earn you any more bonus throws.
** If you don't get a strike or a spare in the tenth frame, you get no bonus throws and the frame ends as normal.
** So, your throws for the tenth frame could look like any of the following, for example: 72, 7/4, X02, X8/, XX4, XXX. To be clear, these are all separate possible tenth frames, not the results of a single game.

* Scoring works as follows:
** Your base score for a frame is the total number of pins knocked down in that frame.
*** For frames 1-9, if the frame includes a spare, then you further add the number of pins from the throw immediately following the spare. (If the spare occurs on the final bonus throw of the tenth frame, then there is no immediately following throw to add.)
*** For frames 1-9, the frame includes a strike, then you further add the total number of pins from the two throws immediately following the strike. (However, strikes occurring on any bonus throws in the tenth frame do not get the numbers of pins from future throws, if they exist, added as a bonus. That is, those pins do count, but they don't get counted again because of the strike(s).)
*** For frame 10, it's probably easiest to think of the score as just the total number of pins knocked down on all regular and bonus throws. The scores for the six examples above would be: 72 = 9, 7/4 = 14, X02 = 12, X8/ = 20, XX4 = 24, XXX = 30.

If this seems confusing, it might help to read the same information in other words. Here is one description; here is another, if you don't mind awful sidebar ads. If you find a clearer description, let us know so that we can share it with the class. Also, if you find any apparent

inconsistencies or omissions in our description, let us know ASAP! We don't want anyone to be confused about the rules.

II. Your task, part one: Scoring a series of frames

Berkeley Bowl doesn't have an automated scoring system. They have an employee named Ginger who stands at the end of the aisle, next to the squash "pins", and records the results for each frame. Ginger is great at balancing squashes on end, cleaning up cantaloupe guts, and recording the throws for each frame correctly. However, like most Americans, Ginger doesn't entirely understand the rules of bowling scoring, and would appreciate it if you would write a Scheme program to turn the recorded data for a game into a bowling score.

Write a procedure `score-game` that takes one argument, a sentence `frames` that contains exactly ten words representing the results of the ten frames of the game (each character of each word will be a number 0-9, /, or X), and returns the score for that game. **You may assume that `score-frames` will always be given a valid set of frame data corresponding to a "legal" bowling game. For example, you will never see something like 5X or X5 or /3 or 78 or 422 in one frame, or a tenth frame like 34X.**

Note: Any frame beginning with a 0 must be surrounded by double quotes, or else Scheme will consider it to be a number starting with 0, think "that's silly, let me fix that for you", and reduce it to the second digit. See part IV of this document.

Examples:

```
> (score-game '("00" "01" 12 23 34 45 13 24 35 14))
48   ; when there are no spares or strikes, the total score is
     ; just the total number of pins knocked down

> (score-game '(X X X X X X X X X XXX))
300  ; a "perfect game" -- the highest attainable score.

> (score-game '("0/" 1/ 2/ 3/ 4/ 5/ 6/ 7/ 8/ 9/X))
155

> (score-game '(10 10 10 10 10 10 10 10 10 X9/))
29

> (score-game '(X "00" X "00" X "00" X "00" X "00"))
```

```
50   ; this is about the biggest bummer of a game you can have
     ; in bowling
```

III. Your task, part two: Interpreting a series of throws

Ginger can't work every night, after all, so the employees in the Berkeley Bowl robotics department (of course they have a robotics department!) have created R.A.D.I.S.H. (Robotic Automated Detector of Impact-based Squash Horizontality). Whereas Ginger records bowling scores by frame, as most humans and most real bowling scoring systems would, R.A.D.I.S.H. doesn't know anything about frames or bowling rules and scoring. It doesn't even know what frame the player is on, or which throw within a frame the player is on. R.A.D.I.S.H. can only do one thing: determine how many squashes (pins) have been knocked over on each throw. (It still knows when zero pins are knocked down, because it still sees the ball go by.) So a game that Ginger might report as:

```
45 6/ 3/ 20 X 7/ X X 0/ X3/
```

would be reported by R.A.D.I.S.H. as:

```
4 5 6 4 3 7 2 0 10 7 3 10 10 0 10 10 3 7
```

Like organic produce, R.A.D.I.S.H. may be buggy. Sometimes it will record the wrong number of pins, slip in an extra throw, forget to record a throw at all, etc. You don't need to worry about exactly how or why R.A.D.I.S.H. makes mistakes; the Berkeley Bowl staff just want you to be able to check its output to see whether a series of throws *could* correspond to a valid bowling game. (In theory, R.A.D.I.S.H. could make a mistake that accidentally results in a new valid game that isn't the same as the game that was bowled, but that's not your problem, and you'd never be able to reconstruct the original game anyway.)

Write a procedure `check-game` that takes one argument, a sentence `throws` (which may be of any length; R.A.D.I.S.H. has been known to go bananas and claim that there were 0 or 1000 throws in a game). **You may assume that each thing in this sentence will be a single number between 0 and 10, inclusive.** If this series of throws could represent the results of a valid game of bowling, then return a sentence of ten "frames", corresponding to that series of throws, **in exactly the same format as the sentences Ginger gave you above.**

On the other hand, if the series of throws couldn't constitute a valid game, then return an error message of your choice, as a string (see part IV). **The error message(s) do not need to explain *why* each particular game is invalid**, although you may find it helpful (for your own

debugging purposes, and for reassurance) to make them descriptive. While an error message doesn't *have* to contain a pun based on fruits or vegetables -- e.g., "it's *chard* to believe that constitutes a valid game" -- why *wouldn't* you include one? (There may be a prize for the best pun, although of course you won't be graded on whether you include a pun, or on how good it is!) This is an unusual case in which you have free rein with some of the output of your program; the grading code will use the `string?` predicate procedure, and it will consider any string to be an error message, and anything other than a string to be not a valid error message. **Sentences are not strings!** Don't return a sentence as an error message.

Your code must not crash when it should return an error message; any crash at all will result in a substantial deduction.

Note that when `check-game` outputs a game, its format should be compatible with what `score-game` expects, and so `(score-game (check-game s))` should correctly return the score for a game consisting of a valid series of throws.

Examples:

```
> (check-game '(0 0 0 1 1 2 2 3 3 4 4 5 1 3 2 4 3 5 1 4))
("00" "01" 12 23 34 45 13 24 35 14)

> (check-game '(0 0 0 1 1 2 2 3 3 4 4 5 1 3 2 4 3 5 1 4 2))
"error"
```

I accidentally left in some games that were in the frame-by-frame format instead of the throw-by-throw format. Sorry for the confusion. check-game will **only** be given sentences of individual throws, **not** of frames.

```
> (check-game '(1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0))
"error"

> (check-game '(1 2 3))
"error"

> (check-game '(10 5 10 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1))
"error"
```

IV. Strings

A **string** is another Scheme data type. Think of it as very much like a long word that can include spaces and single quotes and other things you can't put in a word. To make a string, just surround the desired text with double quotes:

```
"it's full o' spaces and quotes!"
```

We haven't talked about strings thus far because we haven't needed them, and we probably won't have too many reasons to use them again. You don't need to worry about sneaky exam questions involving them. We're only introducing them now so that you don't freak out when you see frames like `"05"` instead of `05`. **(`check-game` may output results that stringify frames that begin with a number, i.e., you will see "6/" instead of 6/. These are equivalent, and will also be treated as equivalent for grading purposes.)** You can take the `first`, `butfirst`, `last`, and `butlast` of strings just as if they were words, and the empty string `""` is exactly the same thing as the empty word.

V. Advice and Hints

* There is no skeleton code for this assignment. Part of the assignment is to come up with your own strategy and break down the problem, so we won't impose a particular strategy on you by giving you a framework. (But we're always happy to discuss strategies with you!)

* You can write the two procedures in either order. Understanding one will help you understand the other better.

* It may seem difficult to determine whether a series of throws could represent a valid bowling game, but if you just start at the beginning and think about each throw in turn, there should be no ambiguity about which frame you're on and which throw within a frame you're on.

* Ian found it easiest to write `score-frames` as a recursive (expanding) process with no helpers and `check-game` as an iterative (non-expanding) process with a helper. But it doesn't matter what sort of process each procedure generates, as long as they do the right thing!

* Think very carefully about frame 10! You should write out all the things that can possibly happen in frame 10 before you write any code that handles it. For that matter, for either procedure, you might want to draw out a flowchart of what should happen before you try to implement it in Scheme. **These tasks are complex enough that very few people would be able to just sit down and bang out correct code -- plan first, then code!**

* For `check-game`, if you use the `word` constructor to assemble a frame, you won't have to worry about specially handling frames starting with a 0; `word` will do the right thing.

* It can be helpful to use `print` statements in debugging if you want to know what value a variable held at the time of an error/crash. Put in a `(print <nameofthatvariable>)` shortly before the place where you know the crash will occur. (The syntax for this must be legal -- the `(print)` should be in a series of expressions in the body of a `define`, `let`, or `begin`.)

* `trace`ing your own procedures may also be useful in debugging, since the trace log will show what arguments the procedures were called with each time.

* Instead of returning `error` for each possible error, you should probably make your messages informative, so that you at least know which particular error cases are being reached.

* **Write test cases!** Earlier, Ian said that you could share test cases, and it's fine if you've done that for HW2 and HW3 so far, but we've realized that we don't want people sending around entire lists of test cases. It would be unfair to people who aren't as "well-connected" in the class, and it violates the spirit of the "no looking at / sharing around other people's code" rule. And we don't want to just give you all a big list of test cases, because writing correct, airtight code is part of the assignment, and we don't want to encourage you to write something sloppy, run a bunch of official tests, and then fix things as needed. So the official rule from now on is: **you may discuss test cases, but you may not share entire lists of them, either verbally or in writing.**

* If you're stuck because your check-game is returning sentences that are full of error messages, and you've written your check-game in such a way that you can't get it to just return an error message that isn't embedded in a sentence, there is a hack that you can use instead of rewriting everything. Just make your current check-game a helper, and then have check-game call your helper, get the answer you would have gotten before, and then check to see whether the sentence contains error messages, and return an error message if so. It's ugly, but if it works...