CSC324 Assignment 2: Language Features (due Nov. 11)

One of the reasons Racket is commonly used in teaching programming language courses is that its powerful macro system makes it (relatively) easy to extend the syntax of the core language in rather new ways. In this unit of the course, we look at two particular language features:

- classes and object (based on closures)
- backtracking search (based on continuations)

This assignment, unlike the first one, is not based on one big project. Instead, you are given a set of smaller tasks to complete. These will involve extending the two language features we've studied in interesting ways, so it's an excellent idea to review the material from lecture (and provided in the starter code)

Part 1: Extending classes

Starter code: `class.rkt`, containing the class macro from lecture.

Sample tests: `class_test.rkt`, containing a few sample tests. Please note that these are quite incomplete!

Note: each of the three questions should be done independently of each other: you will submit a different macros for each question.

1. Now that we have attributes and methods under our belt, let's introduce some basic metaprogramming for our class system. Many programming languages have powerful introspection capabilities for determining properties of objects and classes at *runtime*. These include Java's Reflection API (`https://docs.oracle.com/javase/tutorial/reflect/`) and plenty of Ruby's `Object` methods (`http://ruby-doc.org/core-2.1.3/Object.html`). In `class.rkt`, create a macro called `class-meta` which behaves exactly the same as `class`, except every class has two additional attributes named `_attributes` and `_methods`, which return a nested list of lists, where each inner list consists of:

   - a string name of an attribute/method
   - the corresponding value (which might be a function) for the instance.

   Both lists should be sorted by the names of the attributes/methods, in alphabetical order. **Important**: `_attributes` and `_methods` should not appear in the list returned by `_attributes`. Note: your work does not need to take into account traits (see Q2) or any object-level inheritance. That is, if you instantiate a class and then create an object whose parent is the instantiation, accessing `_attributes` or `_methods` on the new object doesn't need to show the attributes/methods defined in the new object.

2. One of the hallmarks of object-oriented programming is the ability to reuse code for one object for another object. The most common form of this is probably *class-based inheritance*, in which a class can reuse all the code of another class through subclassing. Inheritance is limited in that only full-fledged classes can share code. Some languages relax this condition by supporting **traits**. A trait is a bundle of methods that can be added to a class, but unlike a class, are not standalone, and hence cannot have instance objects. In order for a trait to interoperate with a class, it must access the class interface; and so the main purpose of a trait is to augment existing functionality.

   Specifically, you will add new pattern(s) to our existing class macro (renamed to `class-trait`) to support the following syntax: Note that `<trait> ...` might contain zero traits, so you might have a subexpression that looks like (with). Such a use of the macro should do exactly what class does. If there are name collisions for method definitions, either between traits or between a trait and the class definition, use the following lookup rules:

```
(class-trait <Class> (<attr> ...) (with <trait> ...)
  [(<method> <param> ...) <body>] ...)
```

The list of attributes is followed by an extra clause with a emphliteral keyword `with` and then a list of `trait` functions which should then be automatically applied to each instance of the class. Note that `<trait> ...` might contain zero traits, so you might have a subexpression that looks like `(with)`. Such a use of the macro should do exactly what class does. If there are name collisions for method definitions, either between traits or between a trait and the class definition, use the following lookup rules:

- if two or more traits define the same method name, use the method of the trait that appears *earliest* (left-most) in the with expression
- trait method names always take precedence over class attribute and method names

3. The last variation of the class macro we will consider will be to fix one severe limitation with the current macro: the lack of a true constructor. We currently specify attributes of a class and make them explicit parameters of the constructor of the class, which is not always the behaviour we want. Our current macro cannot handle initializing an attribute to some constant default value, nor performing some auxiliary computation before initialization. Your task is to design a new macro which behaves like the class macro, but requires the user to define their own constructor. For the purpose of this question, the only functionality of the constructor should be to set values for *all* of the attributes and return the resulting object. Unlike the previous questions, **it is up to you to design the syntax for this behaviour**. We will not be autotesting your code. There are many possible approaches for both the syntax and how to implement it, and so you can be creative and decide on an approach that makes sense to you. You will submit your work for this question in a separate file, `constructor.rkt`. This file must contain three things:

   - Your macro (you can name it whatever you like)
   - Documentation describing your `implementation` of the macro. This should be one or two paragraphs of written text and code contained in a docstring above your macro.
   - An expression which uses your macro to translate the following class definition from Python to Racket. Note: your constructor should support (but not require) local name bindings because you want users to be able to define it in a similar (though perhaps not identical) style to other functions.

```
class MyClass:
    def __init__(self, a, b):
        r = f(a)
        self.x = f(a)
        self.y = [b, 100, r]
        self.z = "you are cool"

def f(r):
    return r + 5
```

It will be tempting to use mutation to accomplish this task - don't! There are many other viable approaches which you can use that are purely functional in nature.

Part 2: Choices and Backtracking

Note: these questions will assume that you are comfortable with the backtracking API from lecture. The file `choice.rkt` should be imported in your submission for this part, but you do not need to submit it - it will be provided when we run tests.
Starter Code: `choice_uses.rkt`
Sample Tests: `choice_tests.rkt`

4. Define function which takes a list and returns a subset of the elements of that list, represented as a list. Calling (next) repeatedly should generate all subsets of the list, but the order in which they appear doesn't matter. No subset should appear more than once; sets are unordered, and so the lists `'(1 2 3)` and `'(3 1 2)` represent the same set, and should not both appear on repeated invocations of next. The order in which the subsets are returned doesn't matter.

5. Sudoku is a popular puzzle game in which you are given a partially filled grid of numbers, and must fill in the rest of the grid according to certain constraints. In this question, you'll express the constraints in a function passed to the query predicate ?−, and hence write a brute force Sudoku solver. Note: the purpose of this question is not to come up with an efficient algorithm for solving Sudoku puzzles, but to see how we can approach a problem like this by describing the properties of a solution, and letting "the computer" (in our case, the backtracking mechanisms we developed in lecture) do the work for us. Now, the details: though normal Sudoku is played on a 9-by-9 game board, we will use a 4-by-4 board instead, so that the brute force approach will terminate in a non-lethal amount of time. A *valid Sudoku 4-board* will have the following properties:

   - Each row contains the numbers 1 through 4 exactly once
   - Each column contains the numbers 1 through 4 exactly once
   - Each 4-cell quarter (upper left, upper right, lower left, lower right) contains the numbers 1 through 4 exactly once.

   We will represent a Sudoku board using a list of lists, where each inner list represents a row in the board. For example, here is a representation of a valid Sudoku 4-board in Racket:

   ```
   '((1 2 3 4)
     (3 4 1 2)
     (4 1 2 3)
     (2 3 4 1))
   ```

   A partial board will have the same structure, except with empty strings representing blank squares. Your task is to implement the function `sudoku-4`, which takes a partial board, and returns a complete board which is consistent with the partial board, and is valid according to the above rules. You should use backtracking, and so calling `next` after calling `sudoku-4` should yield different valid solutions to the initial partial board.

6. Suppose we want to take our choice expressions and perform some computation over all the possible choices; for example, compute the maximum of all possible outcomes of a choice expression. Right now, we only have one way of doing so: accumulating all of the values into a list, and then perform whatever computation we'd like on that list. Your task is to develop a more general tool called `fold-<`, which behaves like `foldl`, except its last argument is a choice expression instead of a list:

   ```
   > (fold-< max 0 (sin (* (-< 1 2 3 4) (+ (-< 100 200) (-< 1 2)))))
   0.9948267913584064
   ```

   Please note that the combine function should behave the same as in Racket's `foldl` function: it is a binary function whose first argument is the "next choice" and whose second is the accumulator parameter.

7. In a separate file called `peek.rkt`, copy over the existing backtracking library and *modify its implementation* to support a new function called `peek`, which returns a `quote` (symbol) representation of the choices stored on the top (most recent) choice point on the stack of choices. (Side note: I have no idea if it's possible to present a string representation of the continuation using pattern-based macros. Let me know if you find out!) Note that you can modify the implementation of `choices` as well as other existing functions/macros, but not the public interface; in other words, your new library should be completely backwards compatible with all your other backtracking code. Here are some examples of the expected behaviour:

```
> (-< 1 2 3)
1
> (peek)
'(-< 2 3)
> (next)
2
> (peek)
'(-< 3)
> (next) ; Note: this call was originally missing!
3
> (peek)
"false."
> (+ (-< 1 2) (-< 10 20))
11
> (peek)
'(-< 20)
```

We will not be autotesting your submission for this question. Clearly document all of the changes you make so that your TAs can note them.