CSE 341: Homework 3

due 1/18/99

All functions you are asked to implement and turn in should be typed and tested with the scheme interpreter. With each function turn in the source code listing as well as sample output for a demonstrative set of tests. Do not forget to test *end cases*. Use the `turnin` program to turn in an electronic copy of your source code.

Part I: (Not to be turned in.)

Try these questions out before section.

1. Given definitions

```
(define a '(1 2 3))
(define b '(a b c))
(define c '(#t #f))
(define d '(4))

(define x (cons 10 a))
(define y (append a b))
(define z (list a c))

(set-car! a 99)
(set-car! b 'z)
(set-cdr! c '(9 10 11))
(set-cdr! d d)
```

   What are the values of:

   (a) `x`
   (b) `y`
   (c) `z`
   (d) `c`
   (e) `(car d)`
   (f) `(cadr d)`
   (g) `(caddr d)`
   (h) `(cadddr d)`

Part II: (Not to be turned in)

These problems will be discussed in section.

1. Write a higher-order function `overload-1` that allows you to overload a one argument function so that, depending on what type of argument it is called with, it does different things. For example, conversions to strings. Scheme provides `symbol->string` and `number->string`. What if you had a list of symbols and

numbers and you wanted to convert them all to strings. You could not just map either of these functions over the list because they would crash when called with the inappropriate type. Instead we use `overload-1` as:

```
(define sym-or-num->string
        (overload-1 number->string symbol->string symbol?)
)

(map sym-or-num->string '(give me 5))       => ("give" "me" "5")
```

Basically, `(overload-1 orig-func new-func arg-type?)` should call `new-func` if `arg-type?` is true on its argument. Otherwise it should call `orig-func`.

2. Use `overload-1` recursively to define `anything->string` that successfully converts numbers, symbols, strings, and booleans to strings. If it is given an argument that is not of any of these types it should call `error` with an error message.

Part III: (Due Monday, 10/18/99. Turn this in.)

(20 points) One nice feature of C++ is operator overloading. That is, the ability to define two different functions with the same name that take different types arguments. C++ uses the function signature (its name and parameter types) to disambiguate functions with the same name. As this is a useful feature, we might like the same sort of functionality in Scheme. For example, if we defined our own complex number type[1], we might want the arithmetic operators (e.g. `+` and `*`) to work on it as well. Unfortunately there is no builtin mechanism for doing this in Scheme. Recall that the `+` symbol is bound to the `#<primitive-procedure +>` in the global environment. The evaluator looks up this `+` to get `#<primitive-procedure +>` before the arguments to the function are even looked at. What we wish to accomplish here is similar to that of Part II; however, we want to allow for arbitrary number of arguments.

This assignment involves creating a system for combining functions that should have the same name into one function that will first check the type of the arguments it is called with and then call the function that is designated to handle function calls with arguments of that type. Consider the following example. Say we wish to have a procedure `add` that takes in two items and either adds them if they are numbers, appends them of they are lists, or string-appends them if they are strings. We might then hard-code `add` as:

```
(define (add x y)
   (cond ((and (string? x) (string? y)) (string-append x y))
         ((and (list? x) (list? y)) (append x y))
         ((and (number? x) (number? y)) (+ x y))
         (else (error "Bad arguments to add:" x y))
   )
)
```

[1]Note that Scheme has a builtin complex number type, but lets pretend that it does not.

This method works just fine until we try to add our own complex number type. After implementing `complex-number?` and `add-complex-number` we would have to edit `add` to add the function. It would be cleaner to modify `add` as follows:

```
(define old-add add)
(define (add x y)
   (if (and (complex-number? x) (complex-number? y))
       (add-complex-number x y)
       (old-add x y)
   )
)
```

This is fine and well, but we still have not made our solution any more general. Every time we want to overload a function we have to do it by hand. Also, if some versions of `add` allow more than one argument then by wrapping them up in a two argument version of `add` we have limited them to only two arguments. Further more, we should not assume that the arguments are always all the same type. We might want a function `foo` that does one thing when it is passed a string and a number and does something else when it is passed three numbers.

In your solution to this problem, you should implement the following functions:

arguments-ok? arg-list argument-constraint This uses `argument-constraint` to determine if `arg-list` is a valid list of arguments. For example for the function that wants to append two strings, `arguments-ok?` would return true if `arg-list` is a list with two strings in it. E.g. `("hi" "there")`.

overload orig-func new-func argument-constraint This creates a new function that if called with arguments that pass the `argument-constraints` will call `new-func` on the arguments. Otherwise, it calls `orig-func`.

overload-list orig-func func-constraint-list This creates a new overloaded function. See the example below for details. Here `func-constraint-list` should be a list of function – argument-constraint tuples.

We also need a way to easily express argument-constraints like "a function followed by any sequence of numbers" or "two strings and a number." To achieve this we should be able to combine argument-constraints like we would combine lists. For example, we should be able to get the constraint that is "a function followed by any sequence of numbers" by appending the constraint that is "a function" to the constraint that is "any sequence of numbers". You should provide the following functions for manipulating constraints:

make-constraint predicate
This makes a constraint from a predicate. For example `(make-constraint symbol?)` would return the constraint that would require the arguments consist of exactly one symbol.

empty-constraint

This function when called as `(empty-constraint)` returns the constraint which means "no arguments."

anything-constraint

This function when called as `(anything-constraint)` returns the constraint that means "any single argument."

append-constraints constraint-1 constraint-2 ...

This takes any number of constraints and creates a new constraint that is the result of appending these. Only the last constraint is allowed to be an infinite length constraint (like "any sequence of numbers"). The "no arguments" constraint is the identity for `append-constraints`.

combine-constraints constraint-1 constraint-2

This takes two constraints and returns the constraint which is either one or the other. For example, if `c1` is the constraint "two strings," `c2` is the constraint "one number," and `c3` is the constraint "any number of symbols", then

```
(append-constraints (combine-constraints c1 c2) c3)
```

should return the constraint "either a number followed by any number of symbols or two strings followed by any number of symbols."

repeat-constraint constraint k

Makes the constraint that is `k` repetitions of `constraint`. For example, if `c` was "one symbol" then `(repeat-constraint c 5)` would be "five symbols."

infinite-repeat-constraint constraint

For example, `(infinite-repeat-constraint constraint)` would make the argument-constraint that would correspond to "any number of symbols."

Note, you do not have to support constraints of the form "any number of symbols followed by any number of strings."

With these implemented you should be able to implement `add` as:

```
(define add
   (overload-list
      (curry error "Bad arguments:")
      (list
         (list
            +
            (infinite-repeat-constraint (make-constraint number?))
         )
         (list
            append
            (infinite-repeat-constraint (make-constraint list?))
         )
```

```
        (list
           string-append
           (infinite-repeat-constraint (make-constraint string?))
        )
      )
    )
)
```

You can always redefine `add` later to take into account your own complex number type.

```
(set! add (overload add
                    add-complex-number
                    (repeat-constraint
                        (make-constraint complex-number?)
                        2
                    )
          )
)
```

One final thing to note about this system for overloading functions is that the predicates that we use to determine whether we have matched an argument do not need to be simple type checks. They can be any boolean function of the value of the argument. For example, you could call different functions depending on weather the first argument is positive or negative.