Interpreter Project, Part 1[*]

Due November 8th at 10pm

There are two steps to submitting this assignment:

(1) Place your write-up in a Google Drive folder that is shared with me.

(2) Fill out the Interpreter Part 1 Submission Form.

1 Preliminaries

Over the next three weeks, we will write an interpreter for a substantial subset of Racket in Racket. The interpreter project will be broken up into three one-week parts.

You will write your interpreter project in file named **interpreter.rkt**. You will also be responsible for writing tests for your code, which will be kept in the **itests.rkt** file. You will submit your tests each week along with your code, and you will be graded on the thoroughness of your tests as well as the correctness of your code.

We will be building what is known as a “bottom-up" interpreter. This means that we will start at the “bottom", getting the lowest-level components working first before moving “up" to higher-level components. Extensive testing of each component is extremely important as small mistakes early on will make successful completion of this project much more difficult.

Throughout the project, we will emphasize the idea of data abstraction. This means that we will create abstraction barriers between the data manipulated by the interpreter and the actual way in which the data is represented. This will be similar to what you have done in past assignments. Be sure to use constructor and selector procedures whenever you want to create or access data.

Here is a summary of the topics we will implement each week:

Week 1	Variables and binding Evaluation	binding, frame and environment abstractions i-eval, global-env
		read-eval-print-loop
		quote,define
	extra credit	simple syntax checking
Week 2	Primitive procedures	primitive procedure abstraction, i apply
		first, rest,list
		+, -, *, /
	Special forms	if, begin, cond
	Extra credit	i-print
Week 3	User-defined procedures	lambda, let
	More special forms	map, apply
	Meta-circularity	interpret your interpreter
	Extra credit	let*

[*]Credit to Rich Wicentowski for developing the original version of this project.

In addition to the assignment write-up for each week, I will also provide a summary file containing the signatures of each of the functions you will write.

2 The binding abstraction

The first component of the interpreter begins by developing environments. Environments will be used by the interpreter when evaluating Racket expressions. Every expression in Racket is evaluated within some environment. Often it is a special environment called the *global-env* (defined in Part 2).

At the lowest level of an environment are variable bindings. A **binding** is the joining of a symbol to a value. We will use lists to represent bindings. For example, the binding (list 'x 3) indicates that the symbol x is bound to the value 3, and (list 'lst '(1 2)) indicates that the symbol lst is bound to the list (1 2).

These are the functions that you will use to create, access and change variable bindings:

(make-binding var val)	;constructor
(binding-variable binding)	;selector
(binding-value binding)	;selector

Examples:

```
> (define sample (make-binding 'today 'monday))
> sample
(list 'today 'monday)
> (binding-variable sample)
'today
> (binding-value sample)
'monday

> (define sample2 (make-binding 'a-pair (list 1 2)))
> sample2
(list 'a-pair '(1 2))
> (binding-variable sample2)
'a-pair
> (binding-value sample2)
'(1 2)

> (define sample3 (make-binding 'a-list (cons 1 (cons 2 null))))
> sample3
(list 'a-list '(1 2))
> (binding-variable sample3)
'a-list
> (binding-value sample3)
'(1 2)
```

You may also want to see the summary page for more details.

3 The frame abstract

Next we need to implement frames, which are lists of bindings. The procedure *make-frame* takes a list of variables and a list of values, both of which should be the same length, and creates a frame containing bindings of the variables to the values. If the variable and value lists are of different lengths, make-frame should print an appropriate error message.

For example:

```
> (make-frame ’(a b c) ’(6 7 8))
(list (list ’a 6) (list ’b 7) (list ’c 8))

> (make-frame ’(d e) ’((1 2) (3 4)))
(list (list ’d ’(1 2)) (list ’e ’(3 4)))

> (make-frame ’(x y z) ’(#t #f))
Error: too many variables!

> (make-frame ’(today tomorrow) ’(mon tue wed))
Error: too many values!
```

After completing and testing make-frame, write and test the remaining frame functions. (See the summary page for more details.)

(make-frame vars vals)	; constructor
(empty-frame? frame)	
(first-binding frame)	; selector
(rest-of-bindings frame)	; selector
(adjoin-binding binding frame)	
(binding-in-frame var frame)	

Be sure you retain all of your tests in the i-tests.rkt file since you will be graded on your tests as well as your code!

Here are some examples of the above mentioned functions:

```
> (define frame (make-frame ’(a b c) ’(6 7 8)))
> (first-binding frame)
(list ’a 6)
> (rest-of-bindings frame)
(list (list ’b 7) (list ’c 8))
> (empty-frame? frame)
#f
> (binding-in-frame ’a frame)
6
> (binding-in-frame ’c frame)
8
> (binding-in-frame ’x frame)
```

```
#f
```

4 The environment abstraction

Now that frames have been implemented, we can implement **environments**, which are represented as a **mutable** list of frames. An empty environment is represented as the empty list.

So far in this class, we have not used any mutation. Now, however, we will use mutable lists to represent environments. This makes it easier to make updates to the global environment (when we introduce new frames or new variable bindings within a frame).

We don't need mutation: we could instead pass around the current environment to each function. Functions that 'modify' the global environment would instead create and return a new environment. Although this is a feasible strategy, it makes for a somewhat bulkier abstraction, so we are going to use mutation **just for environments** instead (do not use mutable lists or any other form of mutation in any of the other parts of the interpreter!).

Mutable lists in Racket are created using **mcons**, the mutable version of *cons*. The empty mutable list is *(list)*. Instead of *first* and *rest*, though, mutable lists use **mcar** to get the first element and **mcdr** to get the rest of the list.

You can update the first element in a mutable list using **set-mcar!** (the '!' signals mutation in Racket).

Implement the following environment functions. Make sure to test all of them thoroughly before going on. (See the summary page for more details.)

(empty-env)	; constructor
(empty-env? env)	
(first-frame env)	; selector
(rest-of-frames env)	; selector
(set-first-frame! env new-frame)	; mutator
(adjoin-frame frame env)	
(extend-env vars vals base-env)	
(binding-in-env var env)	
(lookup-variable var env)	; outside of the 'environment' abstraction

A variable may appear in several different frames of an environment. For example:

```
(define env1 (extend-env '(a b c) '(1 2 3) (empty-env)))

(define env2 (extend-env '(a c d e) '(red blue green yellow) env1))

(define env3 (extend-env '(a f) '(#t #f) env2))
```

In env3, there are three different bindings for the variable a and two different bindings for the variable c.

Write the procedure *binding-in-env* that will take a variable name and search through the frames of an environment until it finds the first binding for that variable. In this example, the first binding for a would be (list 'a #t), the first binding for c would be (list 'c 'blue) and the first binding for b would be (list 'b 2). It should return #f if no binding is found. Be sure to use any appropriate abstraction functions, such as *empty-env?* and *binding-in-frame*.

```
> (binding-in-env 'c env3)
'blue
```

Finally, use *binding-in-env* to write the procedure *lookup-variable* to find the value of a variable in a given environment. If no binding is found, an error message should be generated. For example:

```
> (lookup-variable 'c env3)
'blue

> (lookup-variable 'g env3)
Error: g is unbound in this environment!
```

5 Temporarily completing the environment

In order to continue with our interpreter, we need to be able to complete two pieces of the environment abstraction: the function *setup-env* and the variable *global-env* which stores the variables in the global environment.

Racket predefines many bindings in its global environment: *null*, *null?*, *first*, *rest*, *list*, and *cons* to name just a few. At this point, we will start with a very small global environment with just a single binding: the symbol *null* will be bound to the empty list ().

This is a temporary solution. As we go on, we will greatly expand our global environment. For now, we will use the following as our definition for setup-env:

```
(define setup-env
     (lambda()
         (extend-env '(null) '(()) (empty-env))))
```

We will then set the global environment to the environment returned by setup-env:

```
(define global-env (setup-env))
```

6 Evaluation

The central component of our interpreter is the evaluator. This is the component that takes a Racket expression and an environment and evaluates the expression in the context of that particular environment. We will do this evaluation by first determining what type of expression we are given, and then dealing with each type on a case-by-case basis.

The most basic types of expressions are the self-evaluating expressions. Self-evaluating expressions are expressions that evaluate to be themselves regardless of the environment. In our evaluator, booleans (#t and #f), numbers (e.g. 5, -56, 3.1415) and strings (e.g. "hello", "this is a string") will be self-evaluating. Functions will not be self-evaluating in our interpreter.

We will deal with other types of expressions, such as the special forms and *define* later in Part

2. We will postpone implementing the remaining special forms (*if*, *begin*, *cond*, *lambda*, *let*, etc.) until Weeks 2 and 3. Our interpreter will begin to be able to evaluate procedure applications in Week 2, and we will complete the evaluation of procedure applications in Week 3.

As always, you can refer to the the summary page to get an overview of the functions you are working on in this part of the project.

6.1 Creating i-eval

Since the name eval has already been taken by the real Racket interpreter, we will call our evaluator *i-eval* (short for interpreter-evaluator).

We will start with self-evaluating expressions. Racket has primitive functions that can test whether or not an expression is one of the self-evaluating types: *boolean?*, *number?*, and *string?*. Since these are the only types of self-evaluating expressions in our language, we can write an evaluator to handle these fairly easily.

Read through the implementation I give below and test it out on a few simple expressions to make sure you understand how it works.

```
(define i-eval
  (lambda (exp env)
    (cond
     ((boolean? exp) exp)
     ((number? exp) exp)
     ((string? exp) exp)
     (else (println "Error: unknown expression type")))))
```

6.2 Read-eval-print-loop

Now, we can create the *(read-eval-print-loop)* which will **read** input from the user, **evaluate** the input in the global-env, **print** the result, then **loop**. For example:

```
> (read-eval-print-loop)
INTERPRETER> 5
5
INTERPRETER> -3.14
-3.14
INTERPRETER> #f
#f
INTERPRETER> "a short string"
"a short string"
INTERPRETER> exit
INTERPRETER done.
```

An initial version of the *read-eval-print-loop* function is shown below. You need to add the ability to allow the user to exit the interpreter by typing the command exit.

```
(define read-eval-print-loop
  (lambda ()
    (display "INTERPRETER> ")
    (let ((user-input (read)))
      (display (i-eval user-input global-env))
      (newline)
      (read-eval-print-loop))))
```

You will get tired of typing (read-eval-print-loop), so you can define a shortcut (repl).

6.3 Implementing quote and quoted expressions

Your interpreter should now handle the self-evaluating boolean, number and string expressions. Now you will extend the range of expressions your interpreter can handle by allowing for quoted expressions. Recall that in Racket, you can create a quoted expression either by using the *quote* special form, or by using its short-hand notation, the single quote:

```
> (quote this-is-quoted)
this-is-quoted
> 'this-is-also-quoted
this-is-also-quoted
> (quote (4 5 6))
(4 5 6)
> '(3.14 #t "hello")
(3.14 #t "hello")
```

Remember that a single-quote placed immediately before an expression is equivalent to using the full quote notation. The read function automatically expands an apostrophe to the quote notation, so our interpreter doesn't need to worry about expressions that start with single-quotes: handling expressions which use the full quote notation is sufficient.

You should write the procedure *quoted?* to test to see if an expression is quoted. Notice from the examples above that when a user enters a quoted expression, they are simply entering a list whose first element is the symbol quote. To test out, your function, try the following line, which will read one line from you and then return whether or not the line you entered was a quoted expression.

```
(quoted? (read))   ;See important note below
```

This will allow you test if the expressions you type are quoted. You should try a number of different inputs to make sure it works. The following should return #t: 'x, (quote x), '(quote x). The following should return #f: 7, "hello", #t

(Note: You can not simply say (quoted? 'x). In evaluating the expression (quoted? 'x), Racket evaluates 'x to be the symbol x and so the procedure quoted? is called with the symbol x – which is no longer quoted; therefore, (quoted? 'x) ; ==> #f. To get the desired effect, you'd have to put your test case in a second quote: (quoted? ''x). When evaluating (quoted? ''x), Racket evaluates ''x, or (quote (quote x)), to be the list (quote x), which will return #t when passed to quoted?.)

You should also write the function *text-of-quotation* which returns the text of a quoted expression. For example, the text of the quotation (quote x) is x. You can test this as follows:

```
(text-of-quotation (read)) ; try typing (quote x) or '(1 2 3)
```

Extend *i-eval* to evaluate quoted expressions. You should use the functions *quoted?* and *text-of-quotation* which you just wrote.

(See the summary page for more details.)

6.4 Implementing define and variables

Your interpreter can now evaluate self-evaluating expressions as well as quoted expressions. You will now add the ability to define (and, in the next section, mutate) the environment by extending your interpreter to handle variable definitions. Notice that your interpreter's *define* special form returns the name of the variable you are defining:

```
> (read-eval-print-loop)
INTERPRETER> (define pi 3.1415)
pi
INTERPRETER> pi
3.1415
INTERPRETER> (define a (quote apple))
a
INTERPRETER> a
apple
INTERPRETER> (define j (quote jacks))
j
INTERPRETER> j
jacks
```

Fortunately, you have already laid the necessary groundwork for implementing *define*. To begin the implementation, we will need to add a case to our *cond* statement in *i-eval* to identify the expression type. You will write the tester function *definition?* for identifying define expressions.

The *definition?* tester function should end up looking a lot like the *quoted?* tester you just wrote. In fact, you will need to write many more tester functions which will have the same format. So, we will add a helper function, *(tagged-list? exp tag)*, to capture this abstraction. The *tagged-list?* function will return #t whenever:

1. exp is a list, and
2. the list has a first element, and
3. the first element of exp is tag.

For example:

```
(tagged-list? '(define x 8) 'define) ;=> #t
(tagged-list? "some string" 'define) ;=> #f  (not a list)
(tagged-list? '(quote x 6) 'define)   ;=> #f
(tagged-list? '(define x) 'define)   ;=> #t  ; See note below
```

(Notice in the third case that *tagged-list?* returns true even though the expression is not a well-formed Racket definition. The procedure *tagged-list?* is only checking to see if the expression is a list and that it begins with the tag you specify. You can address this issue in the extra credit section.)

Using *tagged-list?*, write *definition?* (and go back and rewrite *quoted?*) in terms of *tagged-list?*.

```
(definition? '(define y 3)) ;=> #t
(definition? 5)             ;=> #f
```

We also need to build the selector functions for definitions, definition-variable and definition-value.

```
> (definition-variable '(define x 6))
'x
> (definition-value '(define x 6))
6
```

After recognizing that an expression is a definition, we need to evaluate the expression, and then update the environment to reflect the new definition. The special form *define* can be only be used to create a new binding in the most recently added frame of an environment: **define cannot mutate a binding in the most recently added frame, but it can define a variable even if the variable is defined in another frame**. If the variable binding already exists in the most recently added frame, you should report an error (e.g. duplicate definition for identifier).

However, before you place a binding for a variable into the environment, you will need to evaluate the third element of the define expression. For example, when you say (define a (quote (1 2 3))), a is not bound to (quote (1 2 3)); rather, (quote (1 2 3)) is first evaluated, and its result, (1 2 3), is stored as the binding for a.

Write the function *(eval-definition exp env)* which extracts the definition-value of the define statement from exp and evaluates it using your evaluator, *i-eval*, and then calls *define-variable!* with the definition-variable, the result of evaluating the definition-value, and the environment, env.

You will also need to write the above-mentioned function *(define-variable! var val env)* which checks to see if a binding already exists for var in the first frame of env. If so, report an error. Otherwise, if no binding exists for var in the first frame, we add the new binding to the first frame of the environment. (Notice that the parameter is val, not exp. This is because val has already been evaluated as part of *eval-definition*.)

To complete the implementation for *define*, you will need to make two changes to *i-eval*. First, i-eval must recognize and properly handle define expressions (by adding a case to your *cond* statement). Second, *i-eval* must recognize and deal with variables (also by adding a case to your *cond* statement). Below is the tester *variable?* needed to recognize variables.

```
(define (variable? exp)
     (symbol? exp))
```

When you evaluate (using *i-eval*) an expression that is a variable, it should evaluate as its value in the environment. Earlier, you wrote a procedure that finds the value of a variable in an environment, so you should use that to extend *i-eval* so that it properly evaluates variables.

Be sure to thoroughly test your code. Below are some tests you can use, but you should also add your own to your itests.rkt file:

```
>  (read-eval-print-loop)
INTERPRETER> (define x 3)
```

```
x
INTERPRETER> (define x 23)
error: duplicate definition for identifier
> (read-eval-print-loop)    ; See final section for explanation
INTERPRETER> x
3
INTERPRETER> (define q x)
q
INTERPRETER> q
3
INTERPRETER> (define m 'q)
m
INTERPRETER> m
q
```

Notice that in the expression *(define q x)*, if you hadn't called *i-eval* on the value component of the *define*, q would have been bound to the symbol x rather than the value 3.

(See the summary page for more details.)

7 Extra Credit: implementing a simple syntax checker

To implement a simple syntax checker, you will write two new functions.

(tagged-list-length-n? exp tag n) will return true only if exp is a tagged-list with tag tag and exp is a list of exactly n elements.

(tagged-list-min-length-n? exp tag n) will return true only if exp is a tagged-list with tag tag and exp is a list of at least n elements.

For both functions, if exp is a tagged list with tag tag but does not meet the proper length requirements, you should print an appropriate error message. For some ideas about what your error message should look like, try things like (quote 3 5) or (define x 3 5) in Racket.

By replacing your calls to *tagged-list?* with calls to *tagged-list-length-n?* or *tagged-list-min-length-n?* (depending on which is necessary for a given expression type), your interpreter will be able to catch many simple syntax errors. For example, since *define* always requires exactly 3 parameters, you should replace *(tagged-list? expr 'define)* with *(tagged-list-length-n? expr 'define 3)*.