Interpreter Project, Part 2[*]

Due November 15th 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 2 Submission Form.

1 Preliminaries

1.1 Verifying Part 1

Please make sure your Part 1 code passes the tests I have provided. You should also type some or all of the code in by hand to verify that the interactions are working properly. (Note that I have given you a short-cut for (read-eval-print-loop), (repl), so that you don't have to type as much.)

```
> (repl)
INTERPRETER> (define x 5)
x
INTERPRETER> x
5
INTERPRETER> (define z 'hello)
z
INTERPRETER> z
hello
INTERPRETER> (define y z)
y
INTERPRETER> y
hello
INTERPRETER> exit
INTERPRETER done.
```

If your code passes all of the tests, you can move on to Part 2.

1.2 Introduction to Part 2

This week, you will add the ability to evaluate expressions that use Racket's primitive procedures. **We will not write our own implementations of these primitive procedures**; when a user of our interpreter asks to evaluate an expression with a primitive procedure, we will call the Racket primitive. This will save us from having to write our own versions of procedures like +, <, and cons.

2 Primitive procedures

In order to properly handle expressions which involve calls to primitive procedures, it is important to recall the rule of evaluation for Racket procedure applications. Assume that x has been defined as

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

8 and y has been defined as 9. When Racket encounters an expression such as *(= x y)*, Racket first determines the type of the expression to be a procedure application. Racket knows the expression is a procedure application because:

1. the expression is a list, and
2. the first element of the list is not a special form, such as *quote* or *define* (which we implemented in Part 1) or *begin* or *if* (which we will implement in later in Part 2) or any other special form you implement in Part 3.

To evaluate *(= x y)*, Racket first evaluates = to get the primitive implementation for numerical equality tests. (To see that Racket really does this, type = on a line by itself in Racket: it evaluates to *#<procedure:=>*.) Next, Racket evaluates the variable x to get 8 and it evaluates the variable y to get 9. (You handled evaluation of variables in Part 1.) Now that all of the elements of the list *(= x y)* have been evaluated, we apply the procedure *#<procedure:=>* to the arguments 8 and 9 (to get #f).

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

2.1 (Partially) implementing *i-apply*

In order to implement primitive functions, we will begin by working on handling procedure applications. (See the summary page for details.) We will start by writing the definitions of the tester and selector functions for applications:

```
(application? exp)
(operator exp)
(operands exp)
```

The function *application?* should return #t if the expression is a procedure application. Use the definition above, but do not worry if the expression is a special form for now, we will deal with this case later. Note that the *application?* is trivial to write. The selector functions operator and operand should be clear, as well. For example:

```
(application? '(cons 1 y)) ; => #t
(application? '(newline))  ; => #t
(application? 'cons)       ; => #f
(operator '(cons 1 y))     ; => 'cons
(operands '(cons 1 y))     ; => '(1 y)
(operands '(newline))      ; => '()
```

Note that you want *(application? '())* to return #f.

Once we know that an expression is a procedure application, we have to evaluate each of the procedure arguments. To do this, we will write a helper function called *eval-operands* that evaluates each element in the list of the operands, and returns a new list containing the evaluated operands. (This list of evaluated operands will become the args parameter to the function *i-apply*.)

eval-operands should perform as follows:

```
(define env1 (extend-env '(a b c) '(1 0 8) (empty-env)))
```

```
(eval-operands '(a 9 b c a) env1) ; => '(1 9 0 8 1)
```

Before continuing with our implementation of *i-apply*, we need to look at how primitive functions will be implemented in our interpreter.

2.2 Primitives

Now that we can evaluate the operands in a procedure application, we need to determine the appropriate procedure to apply by evaluating the first element in the list (e.g. +, as shown in the introduction). For primitive functions, this means that we will need have each of the primitive procedure names bound to their implementations in the global environment.

We could bind the symbol *'first* directly to the procedure which implements it, and we could bind the symbol *'cons* directly to the procedure that implements it, and so forth. But, we would like a way of determining whether the symbol is bound to a primitive procedure or a user-defined procedure (created with lambda which we will implement in Part 5). To do this, we will "tag" each primitive procedure with the symbol *primitive* followed by the procedure's name and its implementation. We will do this using the procedure *make-primitive*:

```
(define make-primitive
  (lambda (name proc)
    (list 'primitive name proc)))

;; for example
(make-primitive 'car car) ; => '(primitive car #<procedure:car>)
```

We could actually write our own implementations of each of Racket's primitive procedures (like we did in our data structures and types lectures). For example, we could say:

```
(define addition
  (lambda (x y)
     (cond ((< x 0) (addition (add1 x) (sub1 y)))
           ((= x 0) y)
           ((= y 0) x)
           (else (addition (sub1 x) (add1 y))))))

(make-primitive '+ addition)
```

Doing this for each of Racket's primitives would be tedious. In fact, you'd probably end up using Racket's primitives to implement yours, and your versions would be slower than Racket's versions. It's important to note that using Racket's primitives to implement your primitives is not "cheating". If you were writing a Racket interpreter in C or Java, you'd use Java's implementation of + when writing the Racket implementation of + ... you wouldn't write it from scratch ... so why do you feel the need to do so here?

Since we will just use Racket's implementation for each of the primitives that we would like in our interpreter, we could simply say:

```
(make-primitive '= =)       ; => '(primitive = #<procedure:=>)
(make-primitive 'cons cons) ; => '(primitive cons #<procedure:cons>)
```

Write the tester and selector functions for primitives (see the summary page for details):

```
primitive-procedure?    ; is it the result of calling make-primitive?
primitive-name           ; returns the name of the primitive
primitive-implementation  ; returns the implementation of the primitive
```

Once these have been completed, you should test the new version of *setup-env* which I have provided for you.

This new version of *setup-env* will add the definitions of many primitives to the global environment. In addition, the list *primitive-procedures* provides an easy way for you to add other primitive functions in the future, should you wish to do so. (You would need to add more functions in order to make your interpreter meta-circular). Be sure you understand how primitives work in our interpreter.

2.3 Completing primitive procedure application

To complete the implementation for primitive procedure application, you will need the definitions for *i-apply* and *apply-primitive-procedure*, which I have provided below:

```
(define i-apply
   (lambda (proc vals) ;; note: these "vals" have already been evaluated
     (cond
      ((primitive-procedure? proc) (apply-primitive-procedure proc vals))
      (else (println "Error: unknown procedure type!")))))

(define apply-primitive-procedure
   (lambda (proc vals)
     (apply (primitive-implementation proc) vals)))
```

You can then add the following line to our conditional test in *i-eval*:

```
((application? exp) (eval-application exp env))
```

This leaves you to write *eval-application*, which will evaluate a procedure application in an environment.

Since the *application?* test is a very general test— it matches both procedure applications and special forms— be sure you place the test after you have exhausted all of the tests for special forms.

2.4 Testing

Your interpreter should now handle the following tests. You must write your own tests in the i-tests.rkt file as well.

```
> (repl)
INTERPRETER> (define x 8)
x
INTERPRETER> (define y (+ x 1))
y
```

```
INTERPRETER> (define lst (cons x (cons '(x y z) (cons y null))))
lst
INTERPRETER> lst
(8 (x y z) 9)
INTERPRETER> (define z (+ (first lst) (first (rest (rest lst)))))
z
INTERPRETER> z
17
INTERPRETER> (= (+ x y) z)
#t
INTERPRETER> (define w *)
w
INTERPRETER> (w x y)
72
```

You should also verify some of the code by hand.

```
> (repl)
INTERPRETER> (define x (cons 1 (cons 2 null)))
x
INTERPRETER> x
(1 2)
INTERPRETER> (null? x)
#f
INTERPRETER> (null? (first (rest x)))
#f
INTERPRETER> (= (first x) (- (first (rest x)) 1))
#t
INTERPRETER> (define y (cons (first x) x))
y
INTERPRETER> y
'(1 1 2)
INTERPRETER> exit
INTERPRETER done.
```

3 Special forms

Recall that Racket's rule of evaluation states that all the arguments of a procedure must be evaluated before procedure application. We have seen in class that this rule cannot work for a set of “special forms" that includes *define*, which you have already implemented in your interpreter.

Special forms are “special" because your interpreter must handle each of these forms differently. For example, in a define expression, such as *(define x 5)*, the first argument, x, is never evaluated. An *if* expression, such as *(if (< x y) x y)*, always evaluates its first argument, but will only evaluate one of its remaining two. Furthermore, since *define*, *if*, *cond*, etc. are not primitives, they have no binding in the environment. Therefore, with special forms, the operator is never evaluated.

On the other hand, all the primitive procedure applications (and, as we will see in Part 3, also lambda-procedure applications) are handled the same way by the your interpreter: the operator and all of the operands are evaluated, then the evaluated operator is applied to the evaluated operands. For example, in the expression *(= x y)*, the operator =, and the operands x and y are all evaluated.

3.1 Special form: *begin*

Extend your interpreter so that it can handle the special form *begin*. Though we have not used *begin* in class, it is a relatively straightforward. The syntax of a *begin* expression is as follows:

```
(begin exp1 exp2 ... expn)
```

The *begin* expression evaluates each of the expressions (exp$_1$ through exp$_n$) in order and returns the value of the last expression. (See the Racket documentation for more information on *begin* if you do not understand how it works.) If there are no expressions except the keyword *begin*, you should return *void*. (For now, your interpreter will print out "#<void>".) For example:

```
INTERPRETER> (begin (define x 5) (+ x 3))
8
INTERPRETER> (begin (display "x + 3 = ")
                    (display (+ x 3))
                    (newline)
                    'done)
x + 3 = 8
done
INTERPRETER> (begin) ; this returns void
#<void>
```

To get the interpreter to return *void*, use the Racket primitive function *void* which you call by simply writing *(void)*. You may also wish to add *void* to the list of primitive functions your interpreter supports.

To implement *begin* in your interpreter, you should continue using the same style of abstraction we have been using. Please reference the summary sheet so that the functions you write have the same names as those described.

1. define a tester procedure, in this case, *begin?*
2. write any necessary selectors, in this case, *begin-expressions*
3. and write a controller, in this case, *eval-begin*

3.2 Special form: *if*

Next, we will allow our interpreter to handle the special form *if*. The syntax of an *if* expression is:

```
(if test-expression then-expression else-expression)
```

To evaluate an *if* expression, first the test-expression is evaluated. If the result is true, the then-expression is evaluated and returned as the result of the if expression. Otherwise, the else-expression is evaluated and returned. **Important note**: In Racket, **anything** other than #f is considered true. For example:

```
INTERPRETER> (if (cons 1 2) 'true 'false)
true
INTERPRETER> (if 'not-false #t #f)
#t
INTERPRETER> (if (< 5 4) 'less 'not-less)
not-less
```

Implement all of the necessary procedures to handle *if* in your interpreter.

Your interpreter should now pass the following tests:

```
INTERPRETER> (define lst (cons 1 (cons 2 null)))
lst
INTERPRETER> (if (< (first lst) (first (rest lst)))
             (first lst)
             (first (rest lst)))
1
INTERPRETER> (* 5 (if (= (+ 3 3) (- 8 2)) 10 20))
50
```

In Racket, it is not possible to write an *if* expression that has a then-expression but has no else-expression. However, in our version of Racket, we will allow this to happen. Modify your implementation so that it can handle *if* expressions of this form. You should return #<void> (the result of calling *(void)*) if the test-expression evaluates as #f and there is no else-expression. For example:

```
INTERPRETER> (define x 5)
x
INTERPRETER> (if (= x 6) (+ x 1))  ;this returns void
#<void>
INTERPRETER> (if (= x 5) (+ x 1))
6
```

Important note: Of course you are allowed to use Racket's *if* to help define *if* in your interpreter! (The same is true for *cond* below.)

Reference the summary sheet for the names of the functions you should implement.

3.3 Special form: *cond*

The last thing we will do this week is to implement *cond*. Recall that the syntax of a *cond* is:

```
(cond
   (test1 exp1 exp2 ... expn)
   (test2 exp1 exp2 ... expn)
   ...
   (testn exp1 exp2 ... expn))
```

To evaluate a *cond* expression, your interpreter should work as follows:

1. Evaluate the expression test1.

2. If test1 evaluates to be anything that is not #f, then any remaining expressions exp_1 .. exp_n are evaluated from left to right, and the value of the last expression is returned as the result of the entire cond.
   (a) Notice that *cond* is a lot like *begin*. In fact, you may want to reuse your *eval-begin* procedure to help you when writing your solution to *eval-cond* ... just be careful you pass to *eval-begin* an expression that matches the format expected by *eval-begin*.
   (b) Also, note that if a test condition evaluates to be true but there are no other expressions following the test expression, the return value is the evaluated test expression (which may not be #t).
3. If test1 evaluates to be #f, then the remaining test expressions are evaluated in order until one is found to be true, or there are no more test expressions.
4. If no true expression is found, then the result is void.
5. The symbol *else* appearing in place of a test expression should be considered to be #t. If an *else* appears in a clause which is not the last clause, produce an appropriate error message. (Try out *cond* in DrRacket to see what error message it provides and mimic its behavior.)

Here are some examples:

```
INTERPRETER> (cond
              ((= 3 5) "this is false so we skip this one")
              ((= 3 3) "this is true so we return this message")
              ((= 5 5) "this is also true but since an earlier test
                          was true this never gets returned"))
"this is true so we return this message"
INTERPRETER> (cond
              (1  "since 1 is not #f, this evaluates to be true")
              (else "and once again, we never get here"))
"since 1 is not #f, this evaluates to be true"
INTERPRETER> (cond
              ((< (+ 1 1) 0) 'nope)
              ((< 1 0) 'still-no)
              ((= 4 5) 'no-matches))  ;this returns void
#<void>
INTERPRETER> (cond
              ((< (+ 1 1) 0) 'nope)
              ((< 1 0) 'still-no)
              (else 'matched-else))
matched-else
INTERPRETER> (cond
              ((+ 1 1)))
2
INTERPRETER> (cond)  ; returns void
#<void>
```

Implement all of the procedures necessary to handle *cond* in your interpreter. (See the summary sheet.) Be careful with *first-cond-exp* and *rest-of-cond-exps* and when using *eval-begin* in your implementation of *eval-cond*: the keywords “cond" and “begin" at the start of these expressions can cause you some difficulty, so be sure you check those functions if you seem to be having unexplained difficulties!

4 Extra Credit: Implement *and* and *or*

Like other special forms, *and* and *or* cannot be implemented in our interpreter by just calling Racket’s built-in *and* and *or* functions.

Add the ability to interpret *and* and *or* expressions into our interpreter, following the same style of abstraction that we have been using.

Implement *and?* and *or?*, testors for *and* and *or* expressions, and *eval-and* and *eval-or*, which handle evaluating them. You should also add a clause to the *i-eval* function that calls the appropriate evaluation function if an expression is an *and* or *or* expression.

Note that (1) both functions can handle an arbitrary number of predicates, (2) anything that isn’t #f is considered true, and (3) that both functions always return the result of the last tested item.

```
(define x 5)
(define y 20)

(or (= x 3) (< y 10) (+ x y) (< y x)) ; => 25  --- not a typo!
(and (= x 5) (< y 30) (+ x y) (< x y)) ; => #t
(and) ; => #t
```

Remember that both are **short-circuiting**, which means that if they stop once the result is known. For example, *(and (= 3 4) (display "hello"))* will not display "hello".