CSE 341, Winter 2004, Assignment 5
Due: Wednesday, February 23 at 10:00pm

Last updated: 16 February

Warning: This assignment is much more involved than previous homeworks. It is highly recommended that you start early!

What programming languages course would be complete without an assignment that has you implement your very own interpreter? In this assignment, you will write an interpreter in Scheme for a fairly large subset of the Scheme language we'll call *MiniScheme*. For the functionality implemented, MiniScheme will have the same exact syntax and semantics of its big brother Scheme.

Your interpreter will be a function `MiniScheme`, that when executed, starts a read-eval-print loop for evaluating MiniScheme sentences. Just like in DrScheme, your REP loop will repeatedly read in MiniScheme sentences provided by the user through the input prompt, evaluate them, and print the results of evaluation. Typing `exit` from within your interpreter will quit the interpreter.

We first provide the syntax of MiniScheme sentences. The formal grammar for defining MiniScheme sentences is small and straightforward. The grammar defines 2 categories of MiniScheme sentences: MiniScheme definitions `<def>`, and MiniScheme expressions `<exp>`. MiniScheme expressions can take on a variety of forms, including MiniScheme variables `<var>` and constants `<con>` and primitive operations `<prim>`. The full grammar for MiniScheme is given below:

```
<var> = x | apple | count | ...

<con> = #t | #f | empty_list | (<con> ... <con>) | ... -1 | 0 | 1 ...

<prim> = car | cdr | cons | list | + | - | * | > | =

<def> = (define <var> <exp>)

<exp> = <var>
      | <con>
      | <prim>
      | (<exp> <exp> ...<exp>)
      | (lambda (<var> ...<var>) <exp>)
      | (if <exp> <exp> <exp>)
```

Informally, a MiniScheme sentence can be either a variable definition, or an expression with 6 possible forms: variables, the constants #t for true, #f for false, the set of integers, the empty list "empty_list", a length ≥ 1 list of constants, primitive operations, function application, lambdas, and if-expressions.

In order to write and interpret MiniScheme sentences, we will need to represent MiniScheme sentences in Scheme using an *abstract syntax*. The particular abstract syntax we'll adopt for this assignment is outlined below:

- MiniScheme variables `<var>` will be represented as Scheme symbols:

  ```
  <var> = 'x | 'apple | 'count | ....
  ```

- MiniScheme constants `<con>` will be represented using their corresponding, built-in Scheme constants:

  ```
  <con> = #t | #f | 'empty_list | (<con> ... <con>) | ... -1 | 0 | 1 ...
  ```

  Note: the provided skeleton code defines `'empty_list` in the minischeme global environment so that it is in fact a Scheme constant for the empty list `'()`.

- MiniScheme primitive operations <prim> are represented as Scheme symbols:

```
<prim> = 'car | 'cdr | 'cons | 'list | '+ | '- | '* | '> | '=
```

- MiniScheme definitions <def> are represented as a Scheme list where the first value is the symbol 'define, the second value is a MiniScheme variable, and the third is a MiniScheme expression. For example, the abstract syntax for the MiniScheme definition (define x 3) would be (list 'define 'x 3).

- MiniScheme expressions <exp> are represented in Scheme as follows: The <var> and <con> and <prim> cases have already been considered. The remaining three cases are represented in Scheme as:

```
(<exp> <exp> ...<exp>) = (list <exp> <exp> ... <exp>)
(lambda (<var> ...<var>) <exp>) = (list 'lambda (list <var> ...<var>) <exp>)
(if <exp> <exp> <exp>) = (list 'if <exp> <exp> <exp>)
```

So for example, we could write a MiniScheme function greater_than_3 in Scheme using our abstract syntax as:

```
(list 'define 'greater_than_3 (list 'lambda (list 'n) (list '> 'n 3)))
```

As you might have guessed, writing MiniScheme sentences in abstract syntax can be a real pain since we have to write them using Scheme lists and symbols. Luckily, Scheme provides us with *list-literals* that will turn out to make our life easier. Specifically, a Scheme list (list 1 2 'apple (list 1 'pear)) can instead by written using a "list-literal" as: '(1 2 apple (1 pear)). In this example, this list consists of 2 numbers, a symbol, and a list of a number and a symbol. The apostrophe blocks the evaluation of the whole list, so that it is not necessary to quote separately the symbols that occur as elements of the list, or recursively quote any inner lists (this explains why we can write the inner list as (1 pear)). Try it our for yourself! Using list-literals, we can rewrite the abstract syntax for our greater_than_3 function as '(define greater_than_3 (lambda (n) (> n 3))). List-literals are a very handy feature because they will allow us to write the list-based abstract syntax of MiniScheme sentences by simply writing their corresponding Scheme syntax and surrounding it with an apostrophe!

We now define informally how to evaluate MiniScheme sentences. MiniScheme sentences evaluate to MiniScheme values. A MiniScheme *value* is either a constant <con>, a function closure (a lambda and its environment), or a Scheme primitive operation (more on this shortly...). MiniScheme sentences are evaluated to values in an *environment*. An environment is a data structure that maps MiniScheme variables to their corresponding MiniScheme values. That way, we can define and store variables and access them later. For your interpreter, you will represent an environment as a Scheme list of pairs of MiniScheme variables and MiniScheme values. Given an environment, variable lookup is done by searching the list from the beginning and returning the value of the first variable that matches.

The interpreter starts out with an initial, global environment called global-env. This environment is already defined for you in the provided skeleton code. This global environment is initialized to map the MiniScheme primitive operations <prim> to their corresponding, built-in Scheme primitive functions. Thus, the global environment contains MiniScheme variables for each of the primitive operations: car, cdr, cons, list, +, -, *, >, =. This way, you can do variable lookup on these primitive functions so that your MiniScheme sentences can perform arithmetic, boolean operations, and list construction. This of course means that MiniScheme variables may evaluate to Scheme primitives, which is why we included this as a possible MiniScheme value. Scheme primitives will add an extra case to how we evaluate function application, since we'll need to test if the function being applied is a MiniScheme closure or a primitive Scheme operation. If the function is a Scheme primitive, then we will simply call Scheme's apply function to apply the primitive operation. Don't worry! This case is already taken care of for you in the skeleton code.

So given an environment env, MiniScheme sentences are evaluated to *values* according to the following rules:

- Variables `<var>` evaluate to their value in the current environment `env`. Evaluation should raise an error if the variable is not in the environment.
- Constants `<con>` evaluate to themselves (that is, their Scheme counterparts). Note though that in MiniScheme, the only way to build a list is through the use of primitive operations. Therefore, MiniScheme doesn't allow you to write a non-empty list explicitly without denoting the list as a function application of a primitive list construction operation. As such, your interpreter will *never* have to evaluate a non-empty list to itself.
- Primitive operations `<prim>` evaluate to their value in the current environment `env`. By definition of our global environment `global-env`, this value will be a *Scheme* primitive. This means that for our interpreter, you can treat `<prim>` and variable evaluation as the same case.
- Definitions `<def>` mutate the environment `env` as follows: if there is an existing binding in the environment for this variable, that binding is mutated to the new value. Otherwise, the environment is mutated to add a new `(var,val)` pair to the front of the environment list. The result of evaluating a definition should be the MiniScheme constant `ok` (represented as a Scheme symbol `'ok`), indicating that the definition executed without errors.
- Expressions `<exp>` evaluate to values depending on their form:
  - We've already discussed how to evaluate variables, constants, and primitive operations.
  - Function application `(<exp_1> <exp_2> ... <exp_n>)` is slightly tricky. Evaluate all of the n sub-expressions to values, in left-to-right order from `<exp_1>` to `<exp_n>`. If the first value is a Scheme primitive operation (+,-,*,car,....), then simply call Scheme's `apply` to apply the primitive to the 2nd through nth values of the application. (This case is already implemented for you in the skeleton code). Else, the first value better be a MiniScheme closure, in which case evaluate the closures functions body in the closures environment extended to map the closure's functions arguments to the 2nd through nth values of the function application.
  - Lambdas are lexically scoped: A lambda `(lambda (<var_1> ...<var_n>) <exp>)` evaluates to a MiniScheme closure holding the function and the current environment. Use the provided `make-closure` routine in the skeleton code to make your closures.
  - A conditional `(if <exp_1> <exp_2> <exp_3>)` evaluates its first subexpression to a value. If this value is the constant #t, then the conditional evaluates to the value of evaluating the 2nd subexpression. Else the conditional evaluates to the value of evaluating the third subexpression.

Your interpreter is run by executing the function `minischeme`. This will start a REP loop for evaluating MiniScheme sentences provided as input by the user from a prompt, indicated by the symbol `$`. An example interaction with the interpreter is shown below:

```
Welcome to DrScheme, version 206p1.
Language: Textual (MzScheme, includes R5RS).
> (minischeme)
Welcome to MiniScheme!
$ (define x 3)
ok
$ x
3
$ (define foo (lambda (var) (+ x var)))
ok
$ (foo 5)
8
$ (cons 1 empty)
```

```
(1)
$ (define x (cons 1 empty_list))
ok
$ x
(1)
$ (define x (car x))
ok
$ x
1
$ (define y (list 1 2 3))
ok
$ y
(1 2 3)
$ (define z (cdr y))
ok
$ z
(2 3)
$ exit
```

As you can see, MiniScheme sentences are inputted from the prompt *without* writing them in abstract syntax! This is just like how you can enter Scheme sentences into the DrScheme interpreter using Scheme syntax. This works because the Scheme `read` function in our REP loop acts as a simple MiniScheme sentence parser, automatically appending an apostrophe `'` to the front of the input sequence of characters. Now you can see why list-literals and our choice of a list-based abstract syntax are so useful for our interpreter!

The following questions use the provided skeleton code in `hw5provided.scm`. This code has a lot of the basic structure implemented for you already. *You should throughly familiarize yourself with this code before attempting to answer any of the questions.* The actual amount of code that you have to write is small once you fully understand how the code and the interpreter function.

1. Define a function `lookup` that takes as arguments a MiniScheme variable `var` and an environment `env` and returns the MiniScheme value that the variable `var` maps to in this environment. If `var` does not exist in the environment, then `lookup` should raise an Scheme error. Hint: raise an error using Scheme's `error` function, specifically: `(error 'lookup "Undefined Variable!")`.

2. Define a function `minischeme-apply` that takes as arguments a MiniScheme value and a Scheme list of MiniScheme values. Specifically, the first argument will be either a MiniScheme closure or a Scheme primitive operation. In either case, `minischeme-apply` should return the value that results from applying the closure's function or the Scheme primitive to the list of values. Hint: this function will need to call `minischeme-eval` from below. The case for primitive operation application is already implemented for you!

3. Define a function `minischeme-eval` that takes as arguments a MiniScheme sentence and an environment, and returns the value that results from evaluating this sentence in this environment. Hint: the function application case should call `minischeme-apply`.

4. The final problem will have you add let-expressions to the MiniScheme language and your interpreter. That way, you can define local variables in your MiniScheme sentences. A let-expression will be defined as just another form of MiniScheme expression `<exp>`:

   ```
   (let ((<var> <exp>) ... (<var> <exp>)) <exp>)
   ```

   We will represent a let-expression in Scheme using the following abstract syntax:

```
(let ((<var> <exp>) ... (<var> <exp>)) <exp>) =
(list 'let (list (list <var> <exp>) ... (list <var> <exp>)) <exp>)
```

To interpret let-expressions, add a new case to minischeme-eval for let-expressions. However, this case *must* interpret let-expressions by converting the let-expression into a function application. Specifically, a let-expression (let ((<var1> <exp1>) ... (<varn> <expn>)) <exp_body>) is equivalent to the function appliction ((lambda (<var1> ... <varn>) <exp_body>) <exp1> ... <expn>). That is, let-expressions are just syntactic sugar!

Hints: start by defining functions let?, let-args, let-vals, and let-body for accessing the various parts of a let-expression's abstract syntax. Write a function desugar_let that takes a let expression and returns the abstract syntax of an equivalent MiniScheme function application. This function will need to make use of Scheme's *quasiquote* feature to build up a list-based abstract syntax for your let's function app. Quasiquote is just like a quote ', but it allows you to write expressions that are mostly literal, leaving holes to be filled in with values computed at runtime. For example, suppose that we define x as (define x 3). Then, (quasiquote (x y z)) will still return (x y z) just like quote, but (quasiquote ((unquote x) y z)) will evaluate to (3 y z). unquote inside a quasiquote says "replace this literal with its value." Scheme also provides a variant of unquote for use when you want to merge an unquoted list into a literal list, rather than nesting it. It is called unquote-splicing and has the effect of stripping off a set of nested parenthesis from the unquoted value. For example, if we define x as (define x '(1 2 3)) and then evaluate (quasiquote ((unquote x) 4 5 6)), we would get ((1 2 3) 4 5 6), while (quasiquote ((unquote-splicing x) 4 5 6)) would yield (1 2 3 4 5 6).

Turn-in Instructions

- Put all your solutions in one file, hw5.scm. This file should contain everything in the skeleton code so that we can run your interpreter directly from your hw5.scm file.
- Line 1 of your .scm file should include a Scheme comment with your name and the phrase homework 5.
- Use the turn-in form linked from the course website to turn your hw5.scm file in.

Sample Output Test Cases Below are some more example outputs from a working minischeme interpreter to guide you in your development. Of course, feel free to try your own test cases since MiniScheme has most of the power of Scheme:

```
Welcome to MiniScheme!
$ (define x 3)
ok
$ (define add1tox (lambda () (+ 1 x)))
ok
$ add1tox
#0=(closure () (+ 1 x) ((add1tox #0#) (x 3) (car #<primitive:car>) (cdr #<primitive:cdr>)
(cons #<primitive:cons>) (list #<primitive:list>) (empty ()) (+ #<primitive:+>) (- #<primitive:->)
(* #<primitive:*>) (/ #<primitive:/>) (> #<primitive:>>) (= #<primitive:=>)))
$ (add1tox)
4
$ (define x 5)
ok
$ (add1tox)
6
```

```
$ (define y 5)
ok
$ (define x y)
ok
$ x
5
$ (define y 10)
ok
$ x
5
$ (define foo (lambda () (let ((y (list 1 2 3)) (x 30)) (lambda () (+ x (car y))))))
ok
$ (define bar (foo))
ok
$ (bar)
31
$ (define x 60)
ok
$ (bar)
31
$ ((foo))
31
$ (let ((x 50)) x)
50
$ x
60
$ exit
```