CS 3110 Fall 2014 **Due at 11:59 PM, 10/23/14**
Problem Set 4
Version 3 (last modified October 19, 2014)

Revision log

- [10/19/14] (i) Clarified that it is an error if the variables bound by a `let*` or `letrec` do not all have distinct names. (The same is not true of `let`.) The reference implementation had a bug and did not correctly detect this error. That bug has been fixed, and the reference implementation has been updated on CMS. (ii) Two examples in the example transcript from the reference implementation were out-of-order; their order has been corrected, and the transcript has been updated on CMS. (iii) The interpreter's parser `interpreter/parser.mly` had a bug with dotted lists. The bug has been fixed, and the file has been updated on CMS.
- [10/15/14] Corrected the names of `TakeIterator` and `RangeIterator` in the writeup.

Objectives

- Use imperative features, including `ref`s and `array`s.
- Gain more experience using the OCaml module system.
- Learn about iteration abstraction.
- Experience a different functional paradigm by implementing an interpreter for an untyped functional language.
- Practice reading a large requirements document and producing a piece of software that satisfies those requirements.

Recommended reading

The following materials should be helpful in completing this assignment:

- Course readings: lectures 11, 12 and 13; recitations 10 and 11
- The CS 3110 style guide
- The OCaml tutorial
- Real World OCaml, Chapter 8

What we supply

We provide an archive `ps4.zip` that you should download from CMS. We also provide a compiled reference implementation of Problem 3 on CMS.

Partners and source control

You are required to work with a partner for this problem set. Each partner is responsible for understanding all parts of the assignment. You need not use the same partner(s) as in previous problem sets.

You are required to use `git`, a version control system, to work with your partner. Your repository must be private. We expect the `git` log you submit to show evidence of work over a period of time, not just a single commit at the end.

Grading issues

Compilation errors: All code you submit must compile. If your submission does not compile, we will notify you immediately. You will have 48 hours after the due date to supply us with a patch. If you do not submit a patch, or if your patched code does not compile, you will receive an automatic zero.

Naming: We use automated grading, so it is crucial that you name your functions and order their arguments according to the problem set instructions, and that you place the functions in the correct files. Incorrectly named functions will be treated as compilation errors.

Code style: Refer to the CS 3110 style guide and lecture notes. Ugly code that is functionally correct will nonetheless be penalized. Take extra time to think and find elegant solutions.

Late submissions: Carefully review the course policy on submission and late assignments. Verify before the deadline on CMS that you have submitted the correct version.

Function Specification and Testing

You should continue to document specifications and write unit tests for functions that you submit as part of your solutions.

Imperative Features

Imperative features are now permitted. Side effects are not necessarily bad—they can be quite useful in real-world programming. But we urge you to think carefully before choosing to use imperative features!

Problem 1: Mutability (30 points)

[code] Starter code for this problem is provided in `mutability.ml`. You may not change `mutability.mli`.

Exercise 1.

Implement a function `count_up_from : int -> int -> unit -> int`. Calling `count_up_from n k` returns a function of type `unit -> int`; let's call that returned function `f`. The first time `f` is called, it should return `n`. The second time `f` is called, it should return `n+k`. More generally, the i^{th} time `f` is called, it should return `n+(i-1)*k`.

```
let c = count_up_from 3 2;;
val c : unit -> int = <fun>
c ();;
- : int = 3
c ();;
- : int = 5
c ();;
- : int = 7
```

Exercise 2.

Implement a function `tabulate : (int -> 'a) -> int -> 'a array` that takes a function `f` and an integer `n`, and returns an array of length `n` where the element at index `i` equals `f i`. For example, `tabulate (fun x -> x*x) 5` yields the array `[|0; 1; 4; 9; 16|]`. You may assume that `f` is pure.

Exercise 3.

Implement `fold_left_imp`, which has the same type and specification as `fold_left`, but without using the `rec` keyword. Do not use library functions other than those in `Pervasives`. (For karma, do it without looping—i.e., do not use `for` or `while`.)

Exercise 4.

Suppose we have two abstract types, `t` and `u`. In the absence of imperative features, it is a theorem that for any function `f : t -> u` and any list `xs : t list`, it holds that

`List.map f (List.rev xs)` = `List.rev (List.map f xs)`.

Using imperative features, write a function `zardoz : t -> u` that will invalidate this theorem when invoked on a list `lst`. You may choose what the representation types of `t` and `u` are. Our test case will use your bindings of `zardoz` and `lst` to assert that the above equality does not hold.

Problem 2: Iterators (50 points)

[code] An *iterator abstraction* (henceforth, *iterator*) produces or *yields* a sequence of results, one at a time. For example, an iterator might yield each element of a list, or each node of a tree, or each of the prime numbers in increasing order. An iterator might eventually run out of results to yield, or it might keep producing results indefinitely. Iterators are primarily an imperative idiom, although pure functions like `fold` resemble iterators.

The exercises in this problem ask you to complete code in `iterator.ml`. Some code has already been provided for you; you should not change that code. The code provides many specifications for you. Your implementation should satisfy those specifications.

Note on starter code: The starter code we provide to you has an interface file named `iterator_when_done.mli`. When you are done writing your solution, you should rename that file to `iterator.mli` and confirm that your solution compiles against that interface. Out of the box, `iterator.ml` will not compile against the signature in `iterator_when_done.mli`, because `iterator_when_done.mli` requires four modules that `iterator.ml` does not yet implement.

Exercise 1.

Implement the `LIST_ITERATOR` signature with a module `ListIterator`.

Exercise 2.

Implement the `INORDER_TREE_ITERATOR` signature with a module `InorderTreeIterator`. Recall that an *in-order* tree traversal recursively visits the left subtree, then the root, then the right subtree.

Exercise 3.

Implement the functor `TakeIterator`. Its behavior is described by the functor type `TAKE_ITERATOR`.

Exercise 4.

Implement the `IteratorUtilsFn` functor. *Hint:* When applied to `ListIterator`, the behavior of `IteratorUtilsFn.fold` should be the same as `List.fold_left`.

Exercise 5.

Implement the functor `RangeIterator`. Its behavior is described by the functor type `RANGE_ITERATOR`. *Hint:* Leverage your solutions to previous exercises.

Problem 3: Scheme3110 Interpreter (140 points)

[code] At this point in your CS 3110 career, you have a good grasp of the *typed* style of functional programming. In this problem, you will gain experience with the *untyped* style of functional programming. Each style has its own benefits; the goal of this problem is to let you see first-hand the benefits of each, by implementing one in terms of the other.

Your task here is to implement an interpreter for language called *Scheme3110*. Scheme3110 is an impure, dynamically typed functional language closely related to Scheme. Scheme3110, like Scheme, is a member of the LISP family of languages, known for their use of *s-expressions* for both program structure and data structure. Scheme was the first LISP to use lexical closures, which you will be implementing in Scheme3110.

We have provided starter code for you in the `interpreter/` directory of the release. There are five modules we provide:

- `Ast`: code for representing the syntax of Scheme3110
- `Environment`: code for representing dynamic environments
- `Eval`: code for implementing evaluation of Scheme3110 programs
- `Identifier`: code for representing names
- `Repl`: code that implements a top-level for Scheme3110

The only file you may modify is `eval.ml`.

Game plan: Your goal is to implement evaluation of Scheme3110 programs. The rest of the writeup below is designed to help you with that by describing

- the Scheme3110 REPL,
- examples of Scheme3110 programs,
- the syntax and semantics of Scheme3110,
- the initial environment of a Scheme3110 program,
- translation of Scheme3110 programs from an *external representation* to expressions,
- evaluating expressions to values, and
- a plan of attack for the order in which you should implement evaluation of expressions.

The rest of this writeup is exceedingly long compared to previous problem sets. You will need to read it carefully if you are to be successful. The good news is that our reference solution is only about 300 lines of code, so the amount of code you need to write is small in proportion to the size of the writeup. But it can be tricky code to write, so you'll need to start early. **More than ever before, it is crucial that you think carefully before writing code.**

Errors: In several places below, we use the phrase "it is an error." By this we mean that your implementation should raise an exception. We do not specify what that exception should be. You might even choose to raise different exceptions for different errors.

3.1 The REPL

We provide a compiled reference implementation of the interpreter on CMS. This is the Scheme3110 equivalent of `utop`, i.e., a REPL (read-eval-print loop). When you're not sure about the evaluation of some Scheme3110 program, the reference REPL can help answer your question. Don't decompile this file: we have obfuscated it, so it's highly unlikely to give you any useful information; and *doing so is a breach of academic integrity.*

To build your own REPL after modifying `eval.ml`, run `cs3110 compile repl.ml`, and to run your own REPL, run `cs3110 run repl.ml`.

3.2 Examples of Scheme3110 programs

We provide a text file `interpreter/example-scheme3110-session.txt`, which contains sample input and output from our reference REPL. **The next thing you should do is read that file.** Come back here when you have finished reading it.

To continue gaining an intuition for the syntax and semantics of Scheme3110, run `interpreter/scheme3110-reference`. Try entering the programs from the sample session. Play around with programs of your own.

3.3 The syntax of Scheme3110

Scheme3110 syntax can be divided into two parts: expressions, and external representations. The *expression syntax* describes expressions that can be evaluated to values. The *external representation syntax* describes data.

Here's an example expression: `(+ a b)`. We read that as calling a procedure named `+` with two arguments. Here's an example external representation: `(+ a b)`. We read that as a list containing three identifiers, the first of which is written `+`. But they look the same—don't they? The essence of Scheme3110 syntax (and Scheme/LISP syntax in general) is that **code is data**: what we normally think of as code (e.g., `(+ a b c)`) can be thought of as data, too.

In the BNF below, boldface denotes syntactic classes of expressions, and $s+$ denotes one or more occurrences of syntactic form s. Whitespace and comments are insignificant other than to act as separators.

3.3.1 External representation syntax

We now describe how data are represented. Note that there is no discussion of evaluation here—data are not yet ready to be evaluated.

datum ::= **atom** | **nil** | **cons-cell**

A *datum* is the fundamental unit of representation in Scheme3110. Even code can be represented as data.

atom ::= **boolean** | **integer** | **identifier**

An *atom* is an indivisible syntactic unit that can be either a Boolean, an integer, or an identifier.

boolean ::= `#t` | `#f`

A Boolean atom may be either true, written `#t`, or false, written `#f`.

integer ::= *same as OCaml integers*

An integer atom may be any integer representable using OCaml's `int` type.

identifier ::= *mostly the same as OCaml identifiers*

Roughly, any sequence of letters, digits, and punctuation that begins with a character is an identifier. Some punctuation by itself is also an identifier. Here are some examples of identifiers: `lambda`, `q`, `list->vector`, `soup`, `+`, `V17a`, `<=?`, `a34kTMNs`, and `the-word-recursion-has-many-meanings`. Identifiers are not case-sensitive. So `HELLO`, `hello`, and `hElLo` are all the same identifier.

If you're interested in the exact syntax for identifiers, see `lexer.mll`.

nil ::= `()`

Nil is also known as the *empty list*. Note: don't confuse this with OCaml's unit value, which is is written with the same syntax.

cons-cell ::= `(` **datum** `.` **datum** `)`

A *cons-cell* is a pair whose first element is called the *car* (pronounced CAR, as in an automobile), and whose second element is called the *cdr* (pronounced COULD-er). The names are historical: they originally stood for "Contents of the Address part of Register number" and "Contents of the Decrement part of Register number."

3.3.2 Syntactic sugar in the external representation

There are two convenient forms of syntactic sugar with which we can extend the external representation syntax:

dotted-list ::= `(` **datum+** `.` **datum** `)`

A dotted-list holds two or more data, with the final datum being separated from the initial data by a single period. The dotted-list `(` $\texttt{datum}_1$ $\texttt{datum}_2$ ... `.` $\texttt{datum}_m$ `)` is syntactic sugar for the cons-cell `(` $\texttt{datum}_1$ `.` `(` $\texttt{datum}_2$ `.` `(` ... `.` $\texttt{datum}_m$ `)` `)` `)`.

list ::= `(` **datum+** `)`

A list holds one or more data, and is implicitly terminated by an empty list. The list `(` $\texttt{datum}_1$ $\texttt{datum}_2$ `...` $\texttt{datum}_n$ `)` is syntactic sugar for the cons-cell `(` $\texttt{datum}_1$ `.` `(` $\texttt{datum}_2$ `.` `(` `...` `.` `(` $\texttt{datum}_n$ `.` `()` `)` `)` `)`.

For example,

the datum...	can be desugared to...
`(define x 2110)`	`(define . (x . (2110 . ())))`
`(quote (+ 1 2))`	`(quote . ((+ . (1 . (2 . ()))) . ()))`

3.3.3 Syntax and semantics

We now give the syntax and semantics of Scheme3110 programs. Programs evaluate to values, where a *value* is either a datum or a closure.

program ::= **toplevel+**

A Scheme3110 program is a series of top-level phrases, evaluated in left-to-right order. The value of the final phrase is returned.

toplevel ::= **definition** | **expr**

A *top-level phrase* is either a definition or an expression. These are the syntactic forms most commonly entered into the top-level—i.e., the REPL—whence the name.

definition ::= `(` `define` **variable expr** `)`

A definition binds the variable to the result of evaluating the body expression, as follows. First, if the variable is not already bound to any value, then it is bound to the empty list, creating a possibly new environment. Second, the body expression is evaluated in that environment. Finally, the identifier is mutated to be the result of this evaluation. The entire definition evaluates to the empty list.

The reason for the initial binding to the empty list is to enable definition of recursive functions: the function will have its own name available as a binding in the environment part of its closure.

variable ::= *Any* **identifier** *that is not a* **keyword**

A variable is an identifier that can be bound to a value in an environment. To evaluate a variable, look up its value in the current environment. Keywords may not be used as variable names. The provided function `Identifier.is_valid_variable` can be used to determine whether an identifier is a valid variable.

keyword ::= `define` | `quote` | `if` | `lambda` | `set!` | `let` | `let*` | `letrec`

A keyword is an identifier reserved by the Scheme3110 language for its own use. It may not be used as a variable name.

expr ::= **self-evaluating** | **variable** | **quote** | **if** | **lambda** | **assignment** | **let** | **let-star** | **letrec** | **procedure-call**

Expressions are the primary building-block of Scheme3110 programs, just as they are for OCaml programs. Scheme3110 expressions are not typed, however.

self-evaluating ::= **boolean** | **integer**

Self-evaluating expressions evaluate to themselves.

quote ::= `(` `quote` **datum** `)` | `'` **datum**

A quote expression evaluates to its unquoted datum. The form `'` **datum** (single-quote and an expression) is syntactic sugar for `(` `quote` **datum** `)`.

Quoting is an important operation in Scheme3110: it allows an arbitrary datum to appear in the program. This syntax enables a program to operate on arbitrary data—including other programs! See the example REPL transcript for an example of using quoting.

if ::= `(` `if` $\textbf{expr}_1$ $\textbf{expr}_2$ $\textbf{expr}_3$ `)`

If $\textbf{expr}_1$ evaluates to `#f`, then the if expression evaluates to the result of evaluating $\textbf{expr}_3$. Otherwise, the if expression evaluates to result of evaluating $\textbf{expr}_2$. Note that any value that is not explicitly false is treated as true.

lambda ::= `(` `lambda` `(` **variable**+ `)` **expr**+ `)`

A lambda expression creates a *procedure*, which is Scheme-speak for a function. The expression evaluates to a closure containing three things: its list of argument variables, its list of body expressions, and the lexical environment which it was evaluated in. None of the body expressions is evaluated yet. It is an error if the variables do not all have distinct names.

assignment ::= `(` `set!` **variable** **expr** `)`

A set-bang expression mutates the variable to the result of evaluating the body expression. It is an error if the identifier is not already bound in the environment. (Note how that is unlike `define`.) The expression evaluates to the empty list.

let-star ::= `(` `let*` `(` **let-binding**+ `)` **expr**+ `)`

let-binding ::= `(` **variable** **expr** `)`

A let-star expression evaluates its let-bindings in order from left-to-right. To evaluate a let-binding, the binding's expression is evaluated in the current environment, then the environment is extended to bind the variable to the result. Thus the second binding is done in an environment where the first is visible, and so forth. Later bindings shadow earlier bindings, in the event that variables do not all have distinct names. (So let-star is similar to an OCaml expression `let v1=e1 in let v2=e2 in...in e`.) Finally, the body expressions of the let-star are evaluated in left-to-right order, starting with the environment containing all the let-bindings.

let ::= `(` `let` `(` **let-binding**+ `)` **expr**+ `)`

A let expression is similar to a let-star expression, but the bindings take effect simultaneously. Each let-binding's expression is evaluated in the original environment of the let expression; the let-bindings should be evaluated from left-to-right. It is an error if the variables do not all have distinct names. (So let is similar to an OCaml expression `let v1=e1 and v2=e2 and...in e`.) Then, a new environment is produced that contains all of the bindings. Finally, the body expressions are evaluated in left-to-right order, starting with this new environment.

letrec ::= `(` `letrec` `(` **let-binding**+ `)` **expr**+ `)`

A letrec expression is similar to a let expression, but every binding is evaluated in an environment where all the other bindings are also visible, thus enabling definition of mutually recursive procedures. It is an error if the variables do not all have distinct names. (So letrec is similar to an OCaml expression `let rec v1=e1 and v2=e2 and...in e`.) It must be possible to evaluate each binding expression without reading or writing the value of a variable in any of the other binding expressions. If not, the behavior of your interpreter is unspecified. (This restriction is necessary to ensure that the mutually recursive definitions can be correctly constructed without exposing an environment in which some of them are fully defined and others are not.)

procedure-call ::= `(` **expr**$_1$ **expr**+ `)`

As long as `expr`$_1$ is not any of the above forms (that is, it is not `quote`, `if`, `lambda`, `set!`, `let`, `let*` or `letrec`), any list of two or more elements is a procedure call. The first expression is evaluated, and if it evaluates to a closure, the remaining expressions are evaluated in order and used as arguments to the procedure. If the first expression does not evaluate to a closure, it is an error. Each argument variable is bound to the corresponding evaluated argument expression. It is an error if the procedure is called with a different number of arguments than the closure expects. The closure environment is then extended with each of these argument bindings, and each of the body expressions is evaluated in order, beginning with the extended environment. The procedure call evaluates to result of evaluating the final body expression of the procedure.

Some procedures, such as addition, are built-in to the interpreter. These builtins are implemented with OCaml code. For example, the Scheme3110 `+` procedure will be implemented with the OCaml `(+)` function.

3.4 The initial environment

The initial environment of a Scheme3110 program contains the following bindings:

- The variable `course`, set to the integer `3110`.
- The procedures `car` and `cdr`, which take in a single cons-cell datum and return the first element and second element of of the cons-cell, respectively. If the procedures are called on anything other than a single cons-cell datum, it is an error.

- The procedure `cons`, which takes in two data and returns a cons-cell whose car is the first argument and cdr is the second argument. If the procedure is not called with exactly two arguments, it is an error.

- The procedures `+` and `*`, which take in a variable number of integer data, and return the OCaml sum and product, respectively, of all their arguments. For example, `(+ 0)`, `(+ 0 0)`, and `(+ 0 0 0)` all evaluate to `0`. If the procedures are called with any non-integer arguments, or are called with zero arguments, it is an error.

- The procedure `equal?`, which returns whether its first argument is structurally equal (in the OCaml sense of `(=)`) to its second argument. If the procedure is not called with exactly two arguments, it is an error. The behavior of `equal?` on procedures is unspecified.

- The procedure `eval`, which takes in a datum value, parses that datum into an expression, and evaluates that expression in the current environment, returning the resulting value.

3.5 Translating from external representation to expressions

The input your Scheme3110 interpreter receives will be in the external representation form given above—that is, it will be a **datum**. The first task your interpreter will need to perform is to *parse* that **datum** into a **program**. We provide a complete representation of Scheme3110 syntax for you in `ast.ml` and `identifier.ml`, which you may not modify. (AST is an acronym for *abstract syntax tree*.)

To implement parsing, complete these functions in `eval.ml`:

```
val read_expression: datum -> expression
val read_toplevel: datum -> toplevel
```

Each function should take in a single datum, syntax check the datum using the expression syntax above, and return an expression (for `read_expression`) or a top-level phrase (for `read_toplevel`). If the datum is not valid Scheme3110 syntax, it is an error. To avoid code duplication, you will want to have `read_toplevel` delegate to `read_expression`, since all valid expressions are valid top-level phrases.

Take care to design your parsing of cons-cells. In implementing that, you might find OCaml `when`-clauses to be useful:

```
match e0 with
  | p when e1 -> e2
  | ...
```

A `when`-clause in a `match` expression qualifies its pattern with a conditional check. The pattern is only matched when that conditional check is satisfied. We have provided you the start of such a `match` statement in the starter code in `eval.ml`.

3.6 Evaluating expressions to values

To implement evaluation, complete the following functions in `eval.ml`:

```
val eval: expression -> environment -> value
val eval_toplevel: toplevel -> environment
                   -> value * environment
```

The first function implements evaluation of expressions. That evaluation happens in the context of a *dynamic environment* in which bindings may be mutated. We provide a complete implementation of environments for you in `environment.ml`, which you may not modify.

The second function implements evaluation of top-level phrases, which may be either expressions or definitions. Top-level evaluation therefore takes in an `toplevel` (which is either a definition or an expression) and an environment to evaluate it in, and returns a value and an environment with any new bindings that should be visible to future evaluations. Your implementation of this function should delegate to `eval` for expressions, because they do not introduce any new bindings into the dynamic environment that should persist in the evaluation of other expressions (though they might mutate the environment).

You also need to complete

```
val initial_environment : unit -> environment
```

which supplies the initial dynamic environment to the interpreter. (The reason why `initial_environment` is a function that takes in `unit` has to do with how we implemented the REPL. It's not really important.)

Implementation strategy: We recommend that you implement evaluation in the following stages. Not only will it allow you to iteratively develop and test the interpreter, but it is also the order that our test suite will test your interpreter. So you will maximize the partial credit you receive by following this strategy. Note that, even though we have identified several stages, you will turn in only a single implementation of your interpreter.

- **Stage 1:** Variables, Booleans, integers, and the `course` variable in the initial environment.
- **Stage 2:** Quote.
- **Stage 3:** If expressions.
- **Stage 4:** Builtins, procedure calls to built-ins, and the rest of the initial environment.
- **Stage 5:** Lambdas and procedure call.
- **Stage 6:** Define and assignment.
- **Stage 7:** Let, let-star, and letrec.

Hint on Stage 7: It is actually easiest to implement let and let-star expressions by rewriting the expression to an equivalent expression involving `lambda`, then applying `eval` to that expression. For example, `(let ((x e1)) e2)` is equivalent to `((lambda (x) e2) e1)`. To implement letrec expressions, the equivalent expression will also involve `set!`.

Problem 4 (0 points)

[written,ungraded] In your file of written solutions, please include any comments you have about the problem set or about your solutions. This would be a good place to list any known problems with your submission that you weren't able to fix, or to give us general feedback about how to improve the problem set.

Include a statement of what work in this problem set was done by which partner. The ideal case is that each of you contributed to every problem. But (especially since this problem is ungraded) please be honest about how you divided the work.