Version of October 1, 1998, 5:28pm

MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall 1997

Problem Set 9

Issued: Tuesday, 25 November

Due: Friday, 5 December, in recitation.

Tutorial preparation for Week of 1 December: Exercises 1A, 4B, 6A below

Reading: Attached code files `tr-flatten.scm`, `fotr-evl.scm`, `fotrinit.scm`, and `derived.scm`. Review *SICP* Sections 4.1–4.1.6, 4.2.

A Tail-Recursive First-Order Scheme Evaluator

1 Tail-recursive calls

The difference between iterative procedures and more general recursive procedures is that the number of "things to do" after a recursive procedure call returns a value can grow unboundedly with the size of the arguments. With iterative procedure calls, the number of things to do after the call stays bounded.

A procedure call is *tail-recursive* when there is *nothing* to do after the call. For example, if a call occurs as the last item in a `begin` expression, it is tail-recursive. Or if a call occurs as a branch of an `if`, it is tail-recursive. A call in a `let` expression is tail-recursive if it is tail-recursive in the body of the `let`. On the other hand, a procedure call inside another procedure call, is *not* tail-recursive. For example, the call to `factorial` in

```
(* n (factorial (- n 1)))
```

occurs within the call to the primitive procedure `*`; it is not tail-recursive because there will still be a multiplication to do after the `factorial` call returns. An expression is tail-recursive if all the procedure calls, aside from calls to primitive procedures, are tail-recursive. A procedure definition is tail-recursive if the sequence of expressions and definitions in its body are all tail-recursive.

Tail-recursive procedures are easier to simulate by hand than general recursive procedures because at any point in an evalution, the only things which need be recorded are the location in the procedure definition of the subexpression currently being evaluated, and the current environment. If the procedure is also *first-order*, meaning it does not use procedures as arguments of other procedures or as returned values, then it isn't even necessary to keep track of the environment: it is enough to record only the values of the parameters and global variables in the procedure definition. For this reason, first-order tail-recursive procedures are also easy to translate into lower-level machine language instructions.

In this problem set we will show you a first-order tail-recursive intepreter for Scheme. The step-by-step behavior of this interpreter is much more explicit than the meta-circular interpreter of Section 4.1 or of PS8. It serves to explain how to implement *all* of Scheme in terms of an elementary, small fragment of Scheme.

And in addition to reducing Scheme to a simpler language, the tail-recursive interpreter will allow us to explain a variety of new programming languages features, for example, MIT Scheme's `error` procedure.

2 Using a stack to turn recursion into tail-recursion

There are general techniques for rewriting recursive procedures as tail-recursive ones. The basic idea is to rewrite the procedures to keep a shared, explicit, `stack` of the computational "things to do" after a procedure call. We'll illustrate this idea with the procedure which flattens a tree, that is, it returns a list of the leaves of the tree in depth-first left-to-right order.

We've already seen `flatten` written as a tree recursion:

```
(define (flatten tree)       ;tree-recursive
  (cond ((null? tree) '())
        ((pair? tree)
         (append (flatten (car tree))
                 (flatten (cdr tree))))
        (else (list tree)))))   ;TREE is a leaf
```

To turn this into a tail-recursive procedure, we look at the non-tail-recursive calls above and think of the computational task which remains to do after each call. We'll assume for simplicity that `append` is tail-recursive (we know how to define it so it would be). This leaves two remaining calls:

```
(flatten (car tree))
```

and

```
(flatten (cdr tree))
```

Neither of these is tail-recursive, since whatever order they are performed in, an `append` will have be done after they return.

We could rewrite `flatten` tail-recursively to do either of these calls first. Let's pick the call to flatten the `car` as the one to do first. Just after flattening the `car` of the tree, what remains to be done? Looking at the definition of `flatten` above, what's next is to flatten the cdr of the tree, then append the previously computed flattened car with the flattened cdr, and return the result.

As an aid to our memory, we'll use the string

```
"(append [] (flatten (cdr tree)))"
```

as a tag to identify this kind of task. Here the "hole", "[]", indicates how this remaining task uses the previously computed flattened car. (The choice of identifying tag is arbitrary. For example, since it will turn out in a moment that there are only two kinds of tasks needed for a tail-recursive `flatten`, we could — and would in a machine implementaion — have used a single bit as a tag instead of a long string.)

The tag "(append [] (flatten (cdr tree)))" indicates the kind of task to be performed, but not the exact datum to perform it on, namely, the `cdr` of the tree. So each particular task of this kind is represented by two items: the tag and the `cdr` of the tree.

Now to process this remaining task, we have to do the second call to flatten the `cdr`. Once that's done, the remaining task is to append the flattened car to the just-computed value of the flattened cdr, and return the result. As a reminder of this, we'll use the tag

```
"(append flattened-car [])"
```

and represent such tasks as lists of two items:

```
("(append flattened-car [])" <flatcar>)
```

where `<flatcar>` is the flattened `car` of the tree obtained from the first call.

Now we're ready to write our tail-recursive `flatten`. The idea is to write a tail-recursive procedure `tr-flatten` which acts like recursive `flatten`, except that instead of *returning* the flattened tree, its "contract" is to call a special `return` procedure, which is also tail-recursive, with the value of the flattened tree:

```
(define (tr-flatten tree)
  (cond ((null? tree) (RETURN '()))
        ((pair? tree) ...)
        (else (RETURN (list tree)))))   ;TREE is a leaf
```

The "contract" of the **return** procedure is to use the value it initially gets from **tr-flatten** to carry out the next task. The next tasks are stored on a stack which is shared by the **tr-flatten** and **return** procedures. The **return** procedure pop's the stack to get the next task. If the stack is empty, which means there is no next task, then **return** really returns the value it gets.

So **tr-flatten** can fulfill its contract by pushing the next task

```
("(append [] (flatten (cdr tree)))" CDR-OF-TREE)))
```

onto the stack and calling itself on the car of the tree. Assuming **tr-flatten** fulfills its contract on the **car** of the tree, the call to itself will result in a call

```
(return <flattened-car>)
```

According to its contract, this call to **return** will complete the task. So we can complete the definition of **tr-flatten**:

```
(define (tr-flatten tree)
  (cond ((null? tree) (return '()))
        ((pair? tree)
         (let ((next-task
                (list
                 "(append [] (flatten (cdr tree)))"
                 (cdr tree))))
           (begin
             (push next-task)
             (tr-flatten (car tree)))))
        (else
         (return (list tree)))))
```

Here we're regarding the stack operation **push** as a primitive procedure. (An implementation the stack operations appears at the end of the file **fotrinit.scm**.)

Now we have to define a tail-recursive **return** procedure which carries out the two kinds of tasks. But that's not hard since **return** can use **tr-flatten**:

```
(define (return val)
  (if (stack-empty?)
      val
      (let* ((next-task (pop))
             (tag (car next-task)))
        (cond
         ((string=? "(append [] (flatten (cdr tree)))" tag)
          (let ((cdr-of-tree (cadr next-task))
                (further-task
                 (list
                  "(append flattened-car [])"
                  val)))                ;VAL is the flattened car
            (begin
              (push further-task)
              (tr-flatten cdr-of-tree))))
         ((string=? "(append flattened-car [])" tag)
          (let ((flatcar (cadr next-task)))
            (return (append flatcar val)))) ;VAL is the flattened cdr
         (else (error "RETURN: unknown task" next-task))))))
```

Again, we're regarding the stack operations as primitive.

So finally we have tail-recursive `flatten`:

```
(define (flatten tree)          ;tail-recursive
  (begin
    (clear-stack!)
    (tr-flatten tree)))
```

The code for tail-recursive `flatten` and the stack implementation is in the file `tr-flatten.scm`. To help you see how this works, we list the successive calls of `tr-flatten` and `return`, and the stack contents after each `push` in the evaluation of

```
(flatten '((a b) c))
```

```
tr-flatten: ((a b) c)
stack:
(("(append [] (flatten (cdr tree)))" (c)))

tr-flatten: (a b)
stack:
(("(append [] (flatten (cdr tree)))" (b))
 ("(append [] (flatten (cdr tree)))" (c)))

tr-flatten: a
return: (a)
stack:
(("(append flattened-car [])" (a))
 ("(append [] (flatten (cdr tree)))" (c)))

tr-flatten: (b)
stack:
(("(append [] (flatten (cdr tree)))" ())
 ("(append flattened-car [])" (a))
 ("(append [] (flatten (cdr tree)))" (c)))

tr-flatten: b
return: (b)
stack:
(("(append flattened-car [])" (b))
 ("(append flattened-car [])" (a))
 ("(append [] (flatten (cdr tree)))" (c)))

tr-flatten: ()
return: ()
return: (b)
return: (a b)
stack:
(("(append flattened-car [])" (a b)))

tr-flatten: (c)
stack:
(("(append [] (flatten (cdr tree)))" ())
 ("(append flattened-car [])" (a b)))

tr-flatten: c
return: (c)
stack:
(("(append flattened-car [])" (c))
 ("(append flattened-car [])" (a b)))

tr-flatten: ()
return: ()
(return: (c))
(return: (a b c))
;Value: (a b c)
```

Tutorial Exercise 1A: What properties of the tree being flattened determine the number of pushes and maximum stack depth?

Lab Exercise 1B: The file `tr-flatten` which loads with this problem set contains the definitions of the flattening procedures above. The stack operations are implemented in `fotrinit.scm` which is also loaded.

Modify the definitions of `tr-flatten` and `return` in the file `tr-flatten.scm`, and the definition of `push` in `fotrinit.scm`, so that an application of `flatten` will produce a trace of calls and stack contents like the one above. (Your trace needn't be as tidy as the one above, which was hand edited.) Hand in some informative example traces obtained using your modified definitions.

Exercise 2. Here is the usual recursive definition of `append`:

```
(define (append x y)
  (if (null? x)
      y
      (cons (car x) (append (cdr x) y))))
```

Using the stack approach to make this tail-recursive, we might come up with

```
(define (append x y)
  (clear-stack!)
  (tr-append x y))

(define (tr-append x y)
  (if (null? x)
      (return y)
      (begin
        (push (list "(cons (car x) [])" (car x)))
        (tr-append (cdr x) y))))
```

Pre-Lab 2A: Write the `return` procedure which goes with `tr-append`. You should not need to have `return` handle any other kind of task than the one in `tr-append`.

Pre-Lab 2B: There are some obvious simplifications possible in the above definitions of `tr-append` and `return`. For example, since there is only one kind of task needed, the string tag `"(cons (car x) [])"` is unnecessary. Also, by making the stack an explicit extra argument of both `tr-append` and `return`, the side-effects on the stack can be replaced by simple list operations. With these hints, write a simple functional definition of a tail-recursive `append`.

Lab Exercise 2C: Show some traces of the procedure calls and stack contents of the tail-recursive procedures of **2a**. Compare them to the traces of the functional versions you wrote for **2b**.

3 Using the Evaluator

The code for this Problem Set also includes these files:

- `fotr-evl.scm` is the code for the tail-recursive interpreter. It handles self-evaluating and quoted data, variables, lambda expressions, `if`, sequences, `define` and `set!` directly, and `and`, `or`, `cond`, and `let` as derived expressions. It uses the essentially the same environment/frame and compound procedure representations as the basic metacircular evaluator described in Section 4.1.1 of *SICP*, except that it does not use abstract syntax.
- `fotrinit.scm` contains the procedures that define and initialize the evaluator's representation of environments. It also contains the procedures defining the evaluator's read-eval-print loop.
- `derived.scm` contains the procedures which allow `and`, `or`, `cond`, and `let` to be treated by the evaluator as derived expressions.

You use the evaluator in the same way as the one for Problem Set 8.

- To start up, do

  ```
  M-x load-problem-set: 9
  ```

  in Edwin. This loads the files `fotr-evl.scm`, `fotrinit.scm`, and `derived.scm`. Then evaluate `(start-fotr-eval)` in Scheme; this starts the read-eval-print loop for the tail-recursive evaluator with a freshly initialized global environment, which then prompts for an expression to evaluate.
- To evaluate an expression, you may type the expression into the `*scheme*` buffer followed by `ctrl-x ctrl-e`. (Don't use `M-z` to evaluate the expression in the `*scheme*` buffer—the presence of the evaluator prompts confuses the `M-z` mechanism. But `M-z` works fine for sending expressions to the `*scheme*` buffer from other Scheme-mode buffers.)

  For example,

  ```
  fotr-eval>> (+ 3 4)
  ;;fotr-value:
  7
  ```

  shows an interaction with the `fotr-eval` evaluator.
- It will be helpful to keep separate files of procedure definitions you want to run the evaluator on. When your Edwin Scheme buffer is running the `fotr-eval` read-eval-print loop, you can then visit this file and type `M-o` to have the definitions evaluated by `fotr-eval`.

- You can interrupt `fotr-eval` by typing `ctrl-c ctrl-c`. To restart it with the recent definitions still intact, evaluate `(fotr-restart)`.
- The `fotr-eval` evaluator does not include much of an error system. If you hit an error you will often bounce back into ordinary Scheme. You can restart the current evaluator, with its global environment still intact, by evaluating `(fotr-restart)`.
- Since environments are generally complex, circular list structures, we have set Scheme's printer so that it will not go into an infinite loop when asked to print a circular list. This was done by

  ```
  (set! *unparser-list-depth-limit* 7)
  (set! *unparser-list-breadth-limit* 10)
  ```

  at the end of the file `fotrinit.scm`. You may want to alter the values of these limits to vary how much list structure will be printed as output.

Lab exercise 3: Extend the `fotr-eval` evaluator to handle `let*` as a derived expression. You should do this by defining a tail-recursive procedure `let*->`*¡some-other-form¿*`&eval` similar to those in `derived.scm` and adding a `let*` clause to the `fotr-eval` procedure.

Lab Exercise 4A: Several useful procedures that are built into Scheme, such as `map`, have been omitted from `fotr-eval`. Add a `map` procedure to the evaluator so it is available at the initial `fotr-eval>>` prompt. (There are several ways to do this. Any one is ok, as long as the interpreter remains first-order and tail-recursive.)

Tutorial Exercise 4B: Be prepared to discuss some of the other ways to pre-install `map` into the evaluator.

You may notice that this evaluator responds to errors in different ways: for some errors, like attempting to evaluate an undefined variable, it displays an error message and returns to its own read-eval-print loop. For others, it bounces back into Scheme, sometimes with its own error message and sometimes not.

Lab Exercise 5

5A: Find examples of each of these kinds of error behavior by the evaluator.

5B: Modify the evaluator so that it returns to its own read-eval-print loop with a sensible error message instead of bouncing back to Scheme, for at least one of your examples in **5A**.

Tutorial Exercise 6A: We have used Scheme's `error` procedure throughout the course without much comment. (If you're not sure about how it behaves, look it up in the MIT Scheme Manual (not R^4RS).) Explain how it would be possible, but cumbersome, to add an `error` procedure to the part of Scheme which `meval` of *SICP* 4.1 or PS8 could interpret. This procedure should return to the `meval` read-eval-print loop, *not* that of the underlying Scheme, so you can't simply bind the meval variable `error` to Scheme's underlying `error` procedure.

Lab Exercise 6B: On the other hand, it's easy to add `error` to the fotr-evaluator: all the procedure need do is to display and/or return a message, set the stack to the read-eval-print task, `("repl")`, and call `return`.

A `*builtin-error*` procedure can be added to `fotr-eval` in a way similar to `*builtin-apply*`. Namely, add a special clause for `*builtin-error*` to the `apply-dispatch` procedure in `fotr-evl.scm`. Also, the variable `error` will have to be bound to `*builtin-error*` in the `fotr-eval` global environment.

Add an `error` procedure to `fotr-eval`. Try it on some examples which demonstrate that it behaves properly. For example,

```
fotr-eval>>
(begin (newline) (display 1) (error 'jumped-out) (display 2))
1
;;fotr-value:
jumped-out

fotr-value>>
```

displays `1`, but *not* `2`, and returns to the `fotr-eval` loop, not underlying Scheme.

Lab Exercise 7: (Optional) Define a predicate `tail-recursive` which takes a Scheme expression as an argument and determines whether it is tail-recursive. Run your procedure on the definitions and expressions in `fotr-evl.scm` and `fotrinit.scm` to determine which ones are really tail-recursive. (Not all are, because it was convenient to treat some non-tail-recursive Scheme procedures as primitives.)

Turn in this sheet along with your answers to the questions in the problem set:

How much time did you spend on this homework assignment?

Indicate all of the following facilities you in doing this problem set:

- The 6.001 lab in building 36.
- MIT Scheme on Project Athena.
- MIT Scheme on a Windows PC.
- EdScheme on a PC.
- Some other Scheme system (which one?).

We encourage you to work with others on problem sets as long as you acknowledge it (see the 6.001 General Information handout).

If you cooperated with other students, LA's, or others, or found portions of your answers for this problem set in references other than the course text (such as some of the course archives), please indicate your consultants' names and your references in the space below. Also, explicitly label all text and code you are submitting which is the same as that being submitted by one of your collaborators.

Otherwise, write "I worked alone using only the course reference materials." and sign your statement.