Scheme: Evaluation

CS 331 Programming Languages Lecture Slides Monday, March 24, 2025

Glenn G. Chappell Department of Computer Science University of Alaska Fairbanks ggchappell@alaska.edu

© 2017–2025 Glenn G. Chappell

Topics

- ✓ PL feature: identifiers & values
- PL feature: reflection
- PL category: Lisp-family PLs
- Introduction to Scheme
- ✓ Scheme: basics
 - Scheme: evaluation
 - Scheme: data
 - Scheme: macros

Review

Scheme is a Lisp-family PL with a minimalist design philosophy.

Scheme code consists of parenthesized lists, which may contain atoms or other lists. List items are separated by space; blanks and newlines between list items are treated the same.

```
(define (print-sum-2-7)
(display (+ 2 7))
)
```

Normal evaluation rule for a list: attempt to evaluate each list item, then attempt to call the result from the first item, as a **procedure**, with the results from the others as its arguments. For example, display and + (above) evaluate to procedures. Things that break this rule, like define (above), are **macros**.

> (pair? '(1 2 3))

Recall: a **predicate** is a function returning a Boolean. Traditionally the name of a Scheme predicate ends in a question mark.

Type-checking predicates take one argument—of any type.

- number? Returns true (#t) if given a number, otherwise false (#f).
- null? Returns true if its argument is null (an empty list).
- pair? Returns true if its argument is a pair. Thus, if the argument is a list, then it returns true if the list is nonempty. If neither null? nor pair? returns true for a value, then the value is an atom.
- list? Checks if argument is a list. Linear-time, so use sparingly.

list? is not actually a type-checking predicate, since list is not a type or collection of types.

A single quote suppresses evaluation.

#f

#t

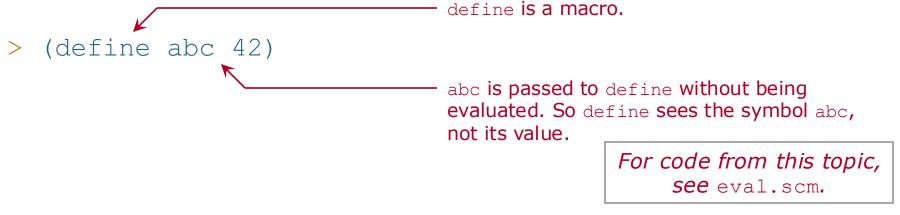
> (pair? +)

Scheme: Evaluation

Scheme: Evaluation Expressions [1/3]

Normal evaluation rule for a list (again): attempt to evaluate each list item, then attempt to call the result from the first item, as a procedure, with the results from the others as its arguments.

When the first item of a list is a macro, the arguments are passed to it unevaluated.



quote is a macro that takes one argument, It returns the argument without evaluating it.

```
> (quote (1 2 3))
(1 2 3)
```

The leading-single-quote syntax is actually shorthand for quote.

```
> '(1 2 3) ; Same as (quote (1 2 3))
(1 2 3)
```

Informally speaking, evaluation removes the quote.

'(1 2 3) (1 2 3) evaluation

eval is a procedure that takes one argument and evaluates it.
Being a procedure, eval does not suppress normal argument evaluation. So evaluation actually happens twice: the argument is evaluated, and then it evaluates the result.



A variation is the procedure apply. This takes a procedure and a list of arguments. It calls the procedure with the given arguments and returns the result.

```
> (apply + '(2 3))
```

5

When evaluation of an expression leads to a call to a Scheme procedure, the call is made in an environment that includes variables in the environment the procedure was defined in.

In short, a Scheme procedure is a closure. The things we have done with closures in other PLs work just fine in Scheme.

TO DO

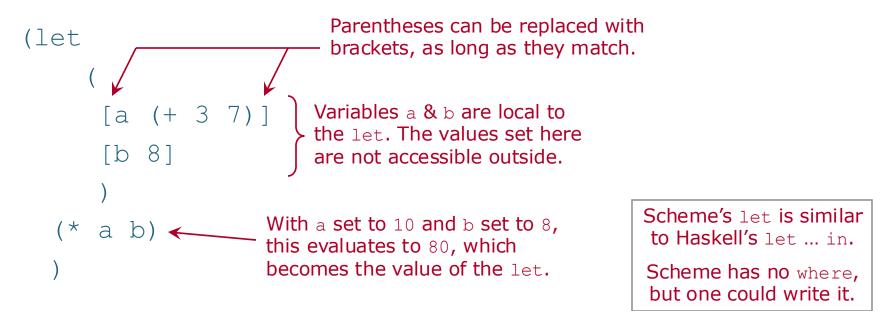
Write some code that uses a closure.

Done. See eval.scm.

We can create and set local variables, while still programming in a functional style, using let. This takes two arguments:

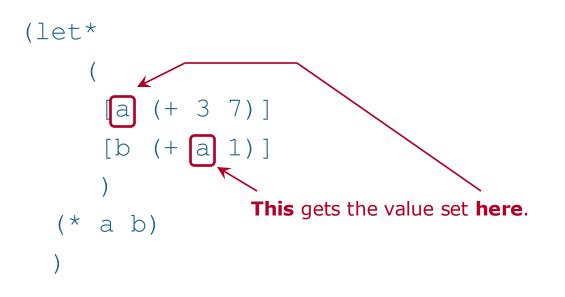
- A list of 2-item lists, each with a symbol and its desired value.
- An expression.

The second argument (the expression) is evaluated with each symbol locally set to its desired value. The result is returned.



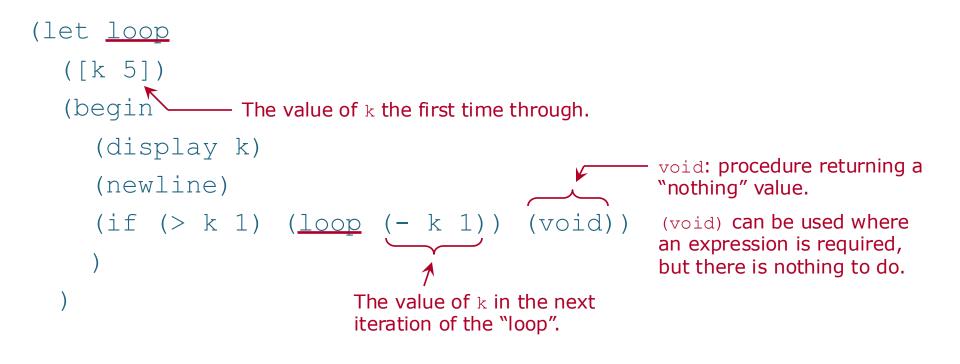
let defines all its local variables at the same time. So we cannot use one of these local variables in the definition of another.

If this is a problem, then use let*, which works just like let, but defines its local variables in order, one after the other.



let takes an optional extra first argument: a symbol. This becomes the name of a procedure that calls the let. The local variables are not set to the values given; rather, they are parameters to the procedure.

It is common for the procedure name to be loop.



TO DO

- Write *filter* in Scheme, with each invocation of the procedure calling each of car and cdr at most once.
- Write a Scheme procedure that inputs a number and prints its square.
- Write a Scheme procedure that uses the let-loop construction.

Done. See eval.scm.

Useful:

- read-line—procedure with no arguments; inputs a line of text from the console and returns it, without the newline
- string->number—procedure that takes a string; returns its numeric form, or #f if there is none

Normally, evaluation in Scheme is eager.

However, we can do lazy evaluation in Scheme, using another of Scheme's types: *promise*. A **promise** is a wrapper around an expression that leaves the expression unevaluated—until the promise is *forced*.

When we **force** a promise, the expression is evaluated, and the resulting value is stored in the promise and returned.

Create a promise using the macro delay.

```
> (define pp (delay (* 20 5)))
```

The type-checking predicate for promises is promise?.

```
> (promise? pp)
#t
```

Force a promise using the procedure force.

```
> (force pp)
100
```

Force a promise as many times as you like; evaluation only happens the first time. The same value is returned each time.

- > (define p (delay (begin (display "Eval!\n") (* 7 6))))
- > (force p)
- Eval!
- 42
- > (force p)
- 42
- > (force p)
- 42

Forcing something that is not a promise will just return it—after evaluating, because force is a procedure.

```
> (force (+ 2 3))
```

5

Using promises, we can create the kind of lazy infinite lists we saw in Haskell (but less conveniently): construct a list as usual, from pairs and null, except that a pair's car and cdr, instead of being an item and a list, are *promises* wrapping an item and a list.

TO DO

- Write code to create a lazy infinite list.
- Write code to return part of a lazy list—like Haskell's take. Make this work with both lazy and normal lists.

Done. See eval.scm.