Scheme: Macros

CS 331 Programming Languages
Lecture Slides
Friday, March 28, 2025

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

Unit Overview
The Scheme Programming Language

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

Introduction to Scheme

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))
  )
```

Review
Scheme: Evaluation

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.

`+` is a symbol that evaluates to a procedure.

```
> (+ (- 4 1) (* 2 3))
9
```

Using the same evaluation method, these evaluate to 3 and 6, respectively.

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

`define` is a macro.

```
> (define abc 42)
```

`abc` is passed to `define` without being evaluated. So `define` sees the symbol `abc`, not its value.

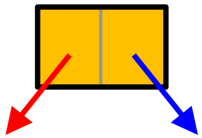
Our procedure `add` takes arbitrarily many arguments:

See `data.scm`.

```
> (add 1 2 3 4 5)
15
```

The above list `(add 1 2 3 4)` is the same as `(add . (1 2 3 4))`.
So a procedure call is a pair. The `car` is the procedure; the `cdr` is a list of the arguments.

Procedure Call



PROC *ARGS*

`define` will also take this form of a "picture" of a procedure call.

```
(define (add . args)
  …
)
```

Scheme: Macros

Scheme: Macros
Background — Symbols, Procedures, Code

```
> (define (cube x) (* x x x))
```

`cube` is a **symbol**. `cube` now evaluates to a **procedure**. A procedure is a black box; we cannot examine the internals.

We can also write an expression that evaluates to a procedure without using `define` or giving it a name: a **lambda**.

```
> (define cube (lambda (x) (* x x x)))   ; Same as above
> ((lambda (x) (* x x x)) 4)             ; No name
64
```

`(* x x x)` is **code** for an expression; it is not a black box. The power of Lisp-family PLs lies in the ability to manipulate code and then evaluate it.

For code from this topic, see `macro.scm`*.*

Background — Reflection

Lisp-family PLs like Scheme have excellent support for **reflection**: the ability of a program to deal with its own code.

For example, we can build and evaluate expressions at runtime.

```
> (define a (list '+ 1 'x))
> (define b (list '+ 2 'x))
> (define c (list '* a b))
> c
(* (+ 1 x) (+ 2 x))
> (define x 10)
> (eval c)
132
```

But the above is not how reflection is usually done in Scheme.

Scheme: Macros
History [1/4]

Early Lisp had code transformations called *fexprs*. A **fexpr** is much like a procedure, but takes its arguments unevaluated. So a fexpr sees its arguments' ASTs, rather than their values.

Like a procedure, a fexpr is a *value*. We can bind a variable to a fexpr, put fexprs in containers, pass them as arguments, etc.

But code-transformations-as-values are problematic. They allow arbitrary changes in the code to happen *anywhere*.

```
(ff a b c d e)  ; Is ff bound to a procedure or a fexpr?
                ; If a fexpr, what does it do?
                ; Before execution, we cannot tell!
```

As a result, writing an optimizing Lisp compiler was difficult—if not impossible—since there was little we could actually know about code before executing it.

Scheme: Macros
History [2/4]

In the mid-1960s, **macros** were introduced to Lisp. These are specified code transformations that are known to the PL implementation, but are not values.

```
> (define pp +)
> (pp 1 2 3)
6
> (define mm lambda)
```

ERROR

"`+`": procedure, which is a first-class value

"`lambda`": macro, which is *not* a value

Macros do not have the code-transformation-as-value problem. By the mid-1980s they had largely replaced fexprs in *practical* Lisp.

Pattern-based macros became common. Code that matches a pattern is replaced by other code, then evaluated as usual.

Scheme: Macros
History [3/4]

The mid-1980s saw discussions about a problem with macros: unwanted interactions between symbols.

Consider: below, there are two distinct variables named `x`. But this is not a problem, since procedure parameters are local.

```
(define (cube x) (* x x x))
(define x 2)
(display (cube (+ x 1)))
```

`x` inside procedure cube

`x` outside procedure cube

Suppose we define a macro analogous to `cube`. It takes an AST called `x`, which it transforms into a list: `*` then `x` three times.

If we use the macro as above, then the `x` inside the macro and the `x` outside could be treated as the same variable. Our macro might do different things in different environments.

Scheme: Macros
History [4/4]

Work-arounds for the problem are available. Many Lisp-family PLs, including Scheme, have a standard procedure `gensym`, which returns a symbol guaranteed not to have been used elsewhere.

```
> (gensym)
g13491
```

A true solution involves **hygienic macros**, which limit interaction between identifiers in a macro and those outside, in much the same way that a procedure does.

Hygienic macros have been optional in Scheme since R4RS (1991) and required since R5RS (1998). But other than Scheme and its spin-offs, there has been little to no adoption of hygienic macros within the Lisp family of PLs. Hygiene limits what a macro can do, and some find this unacceptable.

Scheme: Macros
Overview

Scheme has three standard ways to define hygienic pattern-based macros. From least powerful to most powerful:

- `define-syntax-rule`
- `define-syntax` + `syntax-rules`
- `define-syntax` + `syntax-case`

We will look at the first two in detail and briefly discuss the third.

Note. The standard terminology for Scheme macros uses some words differently from the way we do.

- Macros are said to be **syntax** constructions, although their syntax—in our sense—is no different from anything else in Scheme.
- As we will see, the term *keyword* is also used differently.

Single-Pattern Macros [1/7]

Suppose we wish to define our own version of quote, called `myquote`.

Here is an idea:

```
> (define myquote quote)
```

ERROR

But the above does not work. As a macro, `quote` is not actually something that has a value. Rather, it is a *syntax* construction that our Scheme implementation knows about.

However, there *is* a way to define our `myquote` macro. *Read on.*

Scheme: Macros
Single-Pattern Macros [2/7]

In a **pattern-based macro**, we give a pattern that code can match. Any matching code is transformed in a manner that we specify. The resulting code is evaluated as usual. (If we want to avoid this last step, we can quote the transformed code).

Define a single-pattern macro using `define-syntax-rule`.

`(define-syntax-rule (`*PATTERN*`)` *TRANSFORMED_CODE*`)`

Code that matches *PATTERN* is replaced by *TRANSFORMED_CODE*, and then evaluate.

Scheme: Macros
Single-Pattern Macros [3/7]

`(define-syntax-rule (`*PATTERN*`)` *TRANSFORMED_CODE*`)`

Here is `myquote`.

```
(define-syntax-rule (myquote x)
  'x
  )
```

Above, `(myquote x)` is a **pattern**. The rule is that the first word matches only itself, while other words match anything. So any 2-item list beginning with `myquote` matches. For the purposes of the transformation, `x` means the second item of this list.

```
> (myquote (+ a b))
(+ a b)
```

Scheme: Macros
Single-Pattern Macros [4/7]

`define-syntax-rule` accepts the dot syntax, just like `define`.

Here is a quoting macro that goes inside the list it quotes. `qlist` is like `list`, except that it does not evaluate its arguments.

```
(define-syntax-rule (qlist . args)
  'args
  )
```

Try it.

```
> (qlist (+ 1 3) (* 3 8))   ; Arguments NOT evaluated
((+ 1 3) (* 3 8))
> (list (+ 1 3) (* 3 8))    ; Arguments evaluated
(4 24)
```

Scheme: Macros
Single-Pattern Macros [5/7]

What about actual code transformations?

TO DO

- Write a macro `swap` that takes a 2-item list, reverses the order of the items, and evaluates the result.

```
> (swap ("abc\n" display))
abc
```

- Write a macro `toprod` that takes a nonempty list, changes the first item to `*`, and evaluates the result (as in `reflect.scm`).

```
> (toprod (+ 5 4))
20
```

Done. See `macro.scm`.

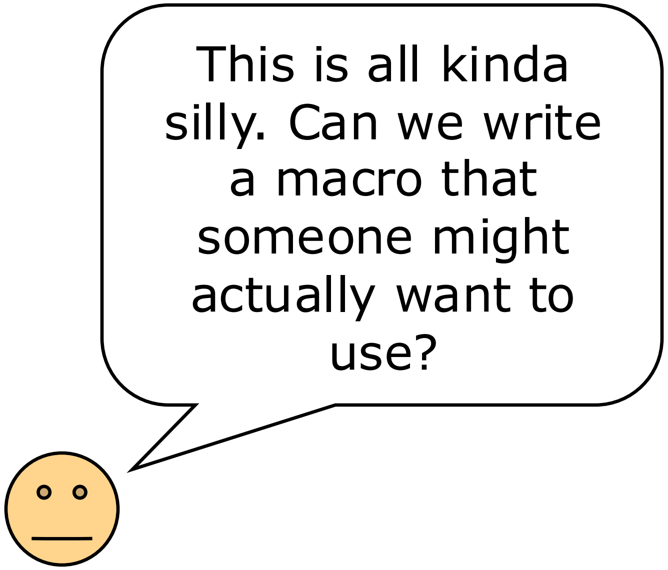
Scheme: Macros
Single-Pattern Macros [6/7]

TO DO

- Write a macro `deftwo` that binds two symbols, each to its own value, as shown below.

Done. See `macro.scm`.

```
> (deftwo a 1 b (+ 2 3))
> a
1
> b
5
```



Scheme: Macros
Single-Pattern Macros [7/7]

Let’s write a pattern-based macro that implements a for-loop, with specified index variable, start and end values, and loop body.

TO DO

- Step 1. Write a for-loop macro `for-loop1` that is used as follows, with `proc` being called with values 3, 4, 5, 6, 7.

```
(define (proc i) (begin (display i) (newline)))
(for-loop1 (3 7) proc)
```

- Step 2. Write a for-loop macro `for-loop2` that is used as follows.

```
(for-loop2 (i 3 7)
    (display i)
    (newline)
    )
```

Done. See `macro.scm`.

Scheme: Macros
TO BE CONTINUED …

Scheme: Macros will be continued next time.