Scheme: Macros II
Introduction to Semantics

CS 331 Programming Languages

Lecture Slides

Monday, March 30, 2026

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

© 2017–2026 Glenn G. Chappell

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 I
- Scheme: macros II

Review

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.

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

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

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.

Review
Scheme: Macros I — Introduction

Scheme supports reflection through **macros**: specified code transformations that are known to the PL implementation.

Macros can involve unwanted interactions between symbols. **Hygienic macros**, which are standard in Scheme, strictly limit interactions between identifiers inside a macro and those outside, much as a procedure does.

Scheme's hygienic pattern-based macro facilities:

- `define-syntax-rule` ← Covered in *Scheme: Macros I*
- `define-syntax` + `syntax-rules`
- `define-syntax` + `syntax-case`

For code from this topic, see `macro1.scm`*.*

Scheme: Macros I — Single-Pattern Macros [1/2]

In a **pattern-based macro**, code that matches a given pattern is transformed in a manner that we specify, then evaluated.

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

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

For example:

```
(define-syntax-rule (my-quote x)
  'x
  )
```

Above, `(my-quote 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 `my-quote` matches.

Scheme: Macros I — Single-Pattern Macros [2/2]

`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
  )
```

DONE

- Write a pattern-based macro that implements a for-loop.

See `macro1.scm`.

Scheme: Macros II

Scheme: Macros II
Multiple-Pattern Macros [1/4]

All the macros we have written have involved a single pattern. We can also write pattern-based macros with multiple patterns.

Recall the general form of `define-syntax-rule`.

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

This is actually shorthand for the following.

```
(define-syntax MACRO_NAME
  (syntax-rules ()
    [(PATTERN) TRANSFORMED_CODE]
    )
  )
```

The name of the macro being defined.

This mysterious list is required. We explain it shortly. For now, make it an empty list.

PATTERN begins with *MACRO_NAME*.

(Actually, while the first item of *PATTERN* must be a symbol, it is ignored. But I suggest that you make it *MACRO_NAME*.)

For code from this topic, see `macro2.scm`*.*

Scheme: Macros II
Multiple-Pattern Macros [2/4]

```
(define-syntax MACRO_NAME
  (syntax-rules ()
    [(PATTERN) TRANSFORMED_CODE]
    )
  )
```

Our macro `qlist` and a rewrite, `qlistx`, in this expanded form:

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

(define-syntax qlistx
  (syntax-rules ()
    [(qlistx . args) 'args]
    )
  )
```

Scheme: Macros II
Multiple-Pattern Macros [3/4]

`syntax-rules` allows for multiple patterns. The first matching pattern is used.

```
(define-syntax MACRO_NAME
  (syntax-rules ()
    [(PATTERN1) TRANSFORMED_CODE1]
    …
    [(PATTERNn) TRANSFORMED_CODEn]
    )
  )
```

TO DO

- Write a macro with multiple patterns.

Done. See `macro2.scm`*.*

Scheme: Macros II
Multiple-Pattern Macros [4/4]

With multiple patterns, we can write recursive macros, using one pattern for the base case and another for the recursive case.

This can be useful for macros taking a variable number of arguments. The recursive case can deal with one argument and then recurse to handle the rest.

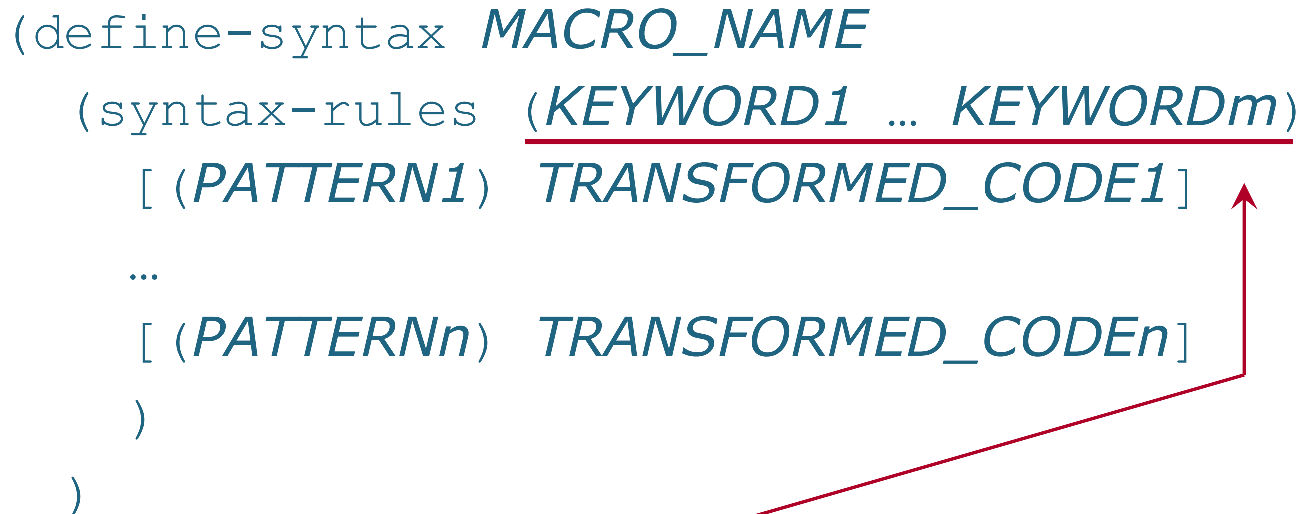
```
(define-syntax MACRO_NAME
  (syntax-rules ()
    [(MACRO_NAME) (void)]
    [(MACRO_NAME arg . rest) … (MACRO_NAME rest) …]
    )
  )
```

TO DO

- Write a recursive macro.

Done. See `macro2.scm`.

Scheme: Macros II
Keywords in Macros



The mysterious list after `syntax-rules` is the list of **keywords**. A *keyword* in a pattern matches only that exact word.

"`else`" as it is used inside a `cond` is a keyword—which *might* be implemented as above.

Here, *keyword* means something different from the way we have used the term.

TO DO

- Write a macro that uses a keyword.

Done. See `macro2.scm`*.*

Scheme: Macros II
Fancier Macros

The 2007 Scheme standard (R6RS) added a more powerful macro facility: `define-syntax` + `syntax-case` (which we will not be covering in any detail). In contrast to the pattern-template method, `syntax-case` allows execution of arbitrary Scheme code to perform transformations of code that matches a pattern.

Possibly this new facility was seen by some as too complex for a minimalist programming language. The 2013 Scheme standard (R7RS) made `syntax-case` an optional feature.

How to best add code transformations to a programming language remains an area of active investigation.

Unit Overview
Semantics & Interpretation

Our sixth unit: Semantics & Interpretation.

Topics

- Introduction to semantics
- Specifying semantics
- How interpreters work
- Writing an interpreter

This unit has another topic, kinda:
Thoughts on Assignment 6.

After this we will cover The Prolog Programming Language.

Introduction to Semantics

Definitions [1/2]

From the first day of class.

Syntax is the correct *structure* of code.

- The string "`a + b`" is a syntactically correct Swift expression.
- The string "`a b +`" is not a syntactically correct Swift expression (but it is a syntactically correct Forth expression).

Semantics is the *meaning* of code.

- In Swift, the semantics of "`a + b`" is roughly as follows: function "`+`" is called, with `a` and `b` passed as its arguments. The return value of this function becomes the value of the expression.

> Usage Note
>
> "Semantics" is an uncountable noun (like "butter"). So "Semantics is …", usually not "Semantics are …".
>
> "Semantic" is the adjective form. We might talk about *semantic correctness*.

Some things the semantics of a PL might address:

Dynamic Semantics	Runtime behavior of code	"`a + b`" computes the sum of the values of `a`, `b` and returns the result.
Static Semantics (in some PLs)	Static typing	"`var nn: Int`" declares a variable named `nn` of type `Int`.
	Correctness of cases	"`switch nn { case 1: print() }`" is an illegal switch-statement, because its cases are not exhaustive.
	Data structure organization	"`struct Xyz { var pp: Int }`" declares a struct named `Xyz` with one property, `pp`, of type `Int`.

None of the above concern issues of syntax; these are covered by the semantics of the PL. However, while the top item is about runtime, the other three—*in Swift at least*—concern issues handled before runtime.

Thus, we distinguish **dynamic semantics** from **static semantics**.

Introduction to Semantics
When It Is Used [1/3]

The title of this unit is “Semantics & Interpretation”.

Q. Why have I put these two topics together in a single unit?

A. Lexing and parsing can be done without any knowledge of the semantics of the PL being lexed/parsed.* It is when we get to interpretation—and also compilation—that semantics is used, and knowledge of semantics becomes necessary.

*Previous assignments had you write a lexer and a parser for a PL. But I have told you nothing about the semantics of this PL—yet.

Introduction to Semantics
When It Is Used [2/3]

“Semantics” is a term that you have probably not used very much.

However, you have certainly used programming-language semantics, *a lot*—though perhaps without referring to it by that name. There is very little that can be done in software development without some knowledge of semantics.

Introduction to Semantics
When It Is Used [3/3]

When is semantics used?

- A **programmer** needs to know the semantics of a PL in order to write code that performs the desired tasks correctly.
- The design of a **compiler** needs to be based on the semantics of both the source PL and the target PL, so that correct object code can be generated.
- Similarly, the design of an **interpreter** needs to be based on the semantics of the source PL, so that correct actions can be performed.
- Semantics is used in **optimization**: altering code so as to improve its performance, while keeping its semantics the same—or nearly the same.
- Semantics is used in **verification**: checking that code performs the actions it is supposed to.

Introduction to Semantics
What is Next?

Over the next few days we will cover:

- A brief look at how semantics can be specified, both formally and informally.
- How semantics is specified for some real-world PLs.
- The various kinds of interpreters and how they work.
- Writing a simple interpreter—and making use of semantics in the process.