Scheme: Basics

CS 331 Programming Languages
Lecture Slides
Friday, March 20, 2026

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 I
  - Scheme: macros II

Review

Review
Introduction to Scheme [1/3]

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 (hello-world)
  (begin
    (display "Hello, world!")
    (newline)
    )
  )
```

Introduction to Scheme [2/3]

Scheme's type system is very similar to that of Lua (dynamic, implicit, structural, fixed set of types). However, Scheme has something like 36 types—as compared to Lua's 8.

Two heavily used types are **pair** and **null**, which are used to construct lists.

Values of all other types are **atoms**. Here are a few of these:

- Booleans: `#t`, `#f`.
- Strings: `"This is a string."`
- Characters: `#\a`
- **Symbols**—first-class identifiers: `abc  !@a=  a-long-symbol?`
- Number types, including arbitrarily large integers, floating-point numbers, exact rational numbers, complex numbers.
- **Procedure** types: first-class functions.

Introduction to Scheme [3/3]

Scheme has no special syntax for flow of control.

Here is some Lua code and more or less equivalent Scheme code.

Lua

```
if x == 3 then
    io.write("three")
else
    io.write("other")
end
```

Scheme

```
(if (= x 3)
    (display "three")
    (display "other")
    )
```

Scheme uses the same kind of syntax for flow of control (`if`), operators (`=`), and regular function calls (`display`). In Lua—and most other PLs—different syntax is used for these three.

Scheme: Basics

Scheme: Basics
From the Reading: Expressions, Defining

The normal evaluation rule for a list: attempt to evaluate each item, then attempt to call the result from the first item, as a procedure, with the results from the others as its arguments. Example: `(list 2 4 10 (+ 33 66))`

Important!

Informally, we say that `list` is a procedure.
Actually, it is a symbol that evaluates to a procedure.

Bind a symbol to a value with `define`: `(define abc (+ 21 21))`

We can also create a procedure with `define`. The first argument is a picture of a procedure call: `(define (square x) (* x x))`

Note that `define` breaks the normal evaluation rule, since its arguments are not evaluated prior to calling `define`. But `define` is not a procedure; it is a *macro* (discussed on another day).

For code from this topic, see `basic.scm`.

Scheme: Basics
Simple Output

Output anything with `display`. Print a newline with `newline`.

```
> (display '(3 "abc" #t 4.))
(3 abc #t 4.0)
```

A single quote suppresses evaluation.

`begin` is given zero or more expressions. It evaluates them all, returning the last value. We can use `begin` to combine I/O calls.

```
(define (print-stuff)
  (begin
    (display "Hello, ")
    (display "world!")
    (newline)
    )
  )
```

Scheme: Basics
Conditionals

`if` takes a condition (truthy/falsy) and two expressions. Depending on the condition, *one* expression is evaluated and returned.

```
(if (> n 0) (+ n 2) (- n 4))
```

Every Scheme value is truthy—treated as true—except for `#f` (false).

`cond` is like if … elseif—or Haskell guards. It takes a list of 2-item lists, each with a condition and an expression. The expression with the *first* truthy condition is evaluated and returned. Optionally, the last condition is `else` (like Haskell's `otherwise`).

```
(cond
  [(> n 0)   (+ n 2)]
  [(< n -5)  (* n 3)]
  [else      (- n 4)]
  )
```

We may replace parentheses with brackets. These must match properly.

When typing the closing delimiter, DrRacket automatically matches an existing opening delimiter.

Scheme: Basics
Lists [1/2]

Two heavily used procedures are `car` and `cdr`. Each takes a pair. `car` returns the first item of the pair. `cdr` returns the second.

For a nonempty list, `car` returns the first item, while `cdr` returns a list of the remaining items.

"`car`" and "`cdr`" come from the processor architecture that the first Lisp implementation ran on. They stood for "Contents of the Address part of the Register" and "Contents of the Decrement part of the Register".

```
> (car '(5 4 2 7))
5
> (cdr '(5 4 2 7))
(4 2 7)
```

Again, a single quote suppresses evaluation. We do not want to treat `5` as a procedure, passing `4`, `2`, `7` as arguments. We want the list.

`cons` constructs a pair. We can use it to construct a list from an item and a list, like ":" in Haskell (which we called "cons").

```
> (cons 5 '(4 2 7))
(5 4 2 7)
```

Scheme: Basics
Lists [2/2]

It is common to use combinations of `car` & `cdr`. For example, `(car (cdr x))` returns the second item of list `x`.

```
> (car (cdr '(5 4 2 7)))       ; Second item
4
> (car (cdr (cdr '(5 4 2 7))   ; Third item
2
```

All such combinations, up to 5 `car`/`cdr` applications, are implemented as predefined procedures in Scheme.

```
> (cadr '(5 4 2 7))    ; Second item
4
> (caddr '(5 4 2 7))   ; Third item
2
```

Scheme: Basics
Predicates

Recall: a **predicate** is a function returning a Boolean. Names of Scheme predicates often end in a question mark.

This convention is common in PLs that support it.

```
> (even? 4)
#t
> (even? 3)
#f
```

`list?` is not actually a type-checking predicate, since *list* is not a type. It is not even a collection of types (as *number* is, for example).

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

- `number?` Returns `#t` if given a number, otherwise `#f`.
- `null?` Returns `#t` if its argument is null (an empty list).
- `pair?` Returns `#t` if its argument is a pair. So if the argument is a list, then it checks if the list is nonempty. If neither `null?` nor `pair?` returns `#t` for a value, then the value is an atom.
- `list?` Returns `#t` if argument is a list. **Linear-time.**

Scheme: Basics
Processing Lists

As in Haskell, we can process lists recursively in Scheme. Often, an empty list is a natural base case. In the recursive case, handle the head (`car`) and recurse on the tail (`cdr`).

TO DO

- Write some list-processing in Scheme: list length, checking for a list, *map*, lookup by index, etc.

Done. See `basic.scm`.

Useful

- Pass a procedure to a procedure by giving it as a normal argument.

```
(foo bar)  ; Pass bar as an argument to foo
```

- `error`—takes a message (string) and crashes with that message.
- `lambda`—returns an unnamed procedure.

```
… (lambda (x) (* x x)) …  ; Unnamed square procedure
```