Scheme: Data

CS 331 Programming Languages
Lecture Slides
Wednesday, March 25, 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

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**.

Review
From the Scheme Reading

Numeric comparison operators: `=` `<` `<=` `>` `>=`

There is no standard numeric inequality operator.

Use these comparison operators only with numbers.

Given a nonempty list, `car` returns its first item (head), and `cdr` returns a list of the remaining items (tail). `cons` constructs a list, given head and tail.

```
> (car '(1 2 3))
1
> (cdr '(1 2 3))
(2 3)
> (cons 1 '(2 3))
(1 2 3)
```

Actually, `car`, `cdr`, and `cons` are more general, working with pairs, not just lists. More on this soon.

Scheme: Evaluation — Expressions

`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.

```
> (eval '(+ 2 3))
5
```

The first evaluation gets rid of the quote. The second calls `+` with `2` and `3` to get `5`.

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
```

Scheme: Data

Scheme: Data
Data Format [1/5]

The dot notation originally used in S-expressions is also valid in Scheme.

```
> '(1 . 2)
(1 . 2)
```

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

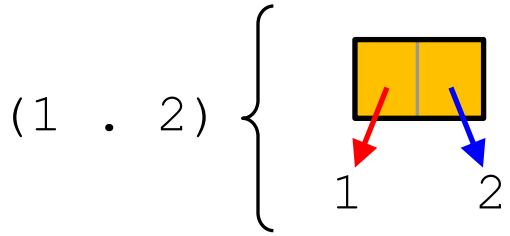
List notation is really shorthand for the equivalent dot notation, again, just as in the original S-expression syntax.

```
> '(1 . (2 . (3 . (4 . ()))))
(1 2 3 4)
```

Dot (`.`) is *not a procedure*. It is simply another way of typing S-expressions. If you want a procedure that puts things together the way dot does, use `cons`.

Scheme: Data
Data Format [2/5]

The main building block for constructing data structures in Scheme is the **pair**. You can think of this as a node with two pointers.



We get the item referenced by the left pointer using **`car`**; similarly use **`cdr`** for the right pointer.

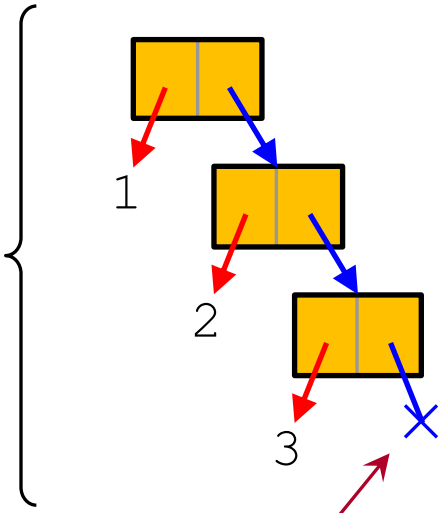
```
> (car '(1 . 2))
1
> (cdr '(1 . 2))
2
```

Scheme: Data
Data Format [3/5]

Lists are constructed from pairs and null.

```
(1 2 3) ≡ (1 . (2 . (3 . ())))
```

I am using this symbol to mean "is the same as".

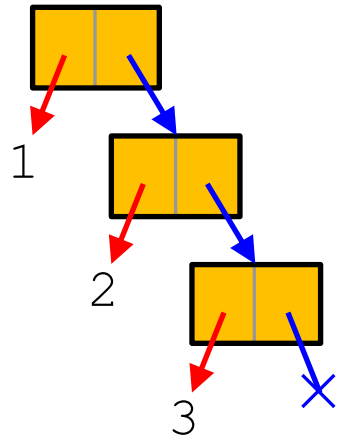


This represents *null*. Think of a null pointer, if you want. (But how it is represented internally is implementation-dependent.)

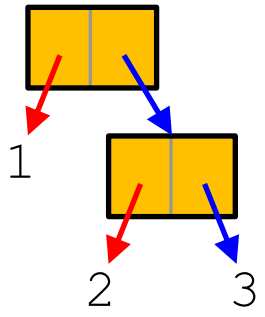
Scheme: Data
Data Format [4/5]

The full story on the dot syntax is that the dot may optionally be added between the *last two* items of something that otherwise looks like a list.

```
(1 2 3)
≡ (1 . (2 . (3 . ())))
```



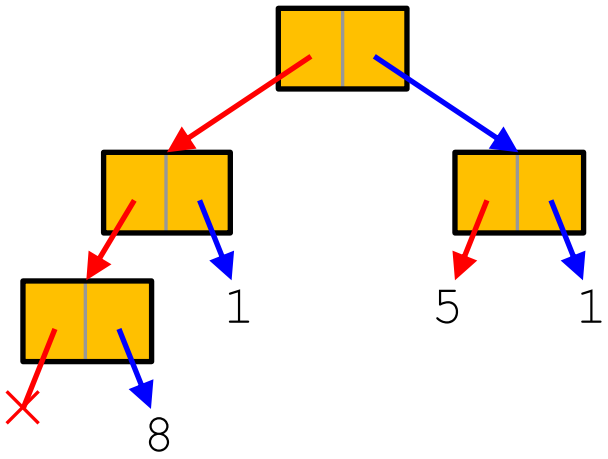

```
(1 2 . 3)
≡ (1 . (2 . 3))
```



Scheme: Data
Data Format [5/5]

We can create arbitrary binary trees—with the restriction that only leaf nodes contain data.

```
(((((  .  8)  .  1)  .  (5  .  1))
```



Scheme: Data Comparisons [1/4]

Scheme has type-specific comparison procedures.

Comparisons for numbers, as we have seen: `=` `<` `<=` `>` `>=`

```
> (= 3 3.0)
#t
> (> 3 3.1)
#f
```

Comparisons for non-numeric types are named as *type*+*op*+?

```
> (string=? "abc" "abc")
#t
> (char<? #\b #\a)
#f
```

Scheme: Data Comparisons [2/4]

There are several kinds of equality in Scheme.

`eq?` tests for "same memory location". I suggest *not* using it.

`eqv?` tests for "same primitive value".

```
> (eqv? 3 3)
#t
> (eqv? 3 3.0)
#f
> (define a '(1 2))
> (eqv? a '(1 2))
#f
> (eqv? "abc" "abc")
```

IMPLEMENTATION-DEPENDENT

Lists and strings are not primitive values.

It is common to use `eqv?` *indirectly*. *See the next slide.*

Scheme: Data Comparisons [3/4]

Of greater interest is `equal?`, which does the following:

- If the types are different, then return `#f`.
- For primitive values (everything we have covered except strings and pairs) of the same type, call `eqv?`.
- For pairs, recursively call `equal?` on the `car`s & `cdr`s.
- For other non-primitive values (e.g., strings) of the same type, call an appropriate type-specific equality comparison, if one exists.

So for lists, `equal?` checks *structural equality*.

```
> (define a '(1 (2 3) 4))
> (equal? a '(1 (2 3) 4))
#t
> (equal? "abc" "abc")
#t
```

Scheme: Data Comparisons [4/4]

`equal?` mostly does what we usually want, with one caveat. Since it always returns `#f` when the types are different, it can give undesirable results with numbers.

```
> (equal? 3 3.0)
#f
```

I offer the following suggestions.

- Use `=` for numeric equality.
- Using `equal?` is fine for most other kinds of equality.
- If you want your code to indicate what type is being compared, or to flag type errors for other types, then use a type-specific equality function: `string=?`, `char=?`, etc.
- Use `eq?` or `eqv?` directly only if you are sure of what you are doing—and probably never.

Scheme: Data
More General Procedures [1/4]

So far, all the procedures we have written have taken a fixed number of parameters. But Scheme allows for procedures like "+", which can take an arbitrary number of parameters.

Let's duplicate "+", in the form of a procedure called `add`.

```
> (add 5 3)
8
> (add 1 2 3 4)
10
> (add)
0
```

We will use "+", but only as a 2-parameter procedure.

Scheme: Data
More General Procedures [2/4]

Consider a call to `add`:

```
> (add 1 2 3 4)
10
```

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. This is illustrated below.

Scheme: Data
More General Procedures [3/4]

A procedure call is a pair: `(`*PROC* `.` *ARGS*`)`. And `define` will also take this form of a "picture" of a procedure call.

```
(define (add . args)
  …
)
```

TO DO

- Write procedure `add`.

Done. See `data.scm`.

A tricky issue is how to make a recursive call on `(cdr args)`. We look at this on the next slide.

Scheme: Data
More General Procedures [4/4]

In writing procedure `add`, we need to make a recursive call on `(cdr args)`. How do we do this?

NOT like this (dot is not a procedure!):

```
(add . (cdr args))   ; WRONG!
```

`(add . (cdr args))` is just another way to write `(add cdr args)`, which is not what we want.

The following will actually work, but it is a bit unwieldy:

```
(eval (cons add (cdr args)))
```

Situations like this are why `apply` exists:

```
(apply add (cdr args))
```

Scheme: Data
Manipulating Trees

We can write code that deals with a structure, not as a list, but as a tree, traversing the tree and dealing with atoms in some way.

TO DO

- Write a procedure `atom-sum` that is given a tree `t` and returns the sum of all the numbers in `t`.
- Write a procedure `atom-map` that is given a procedure `f` and a tree `t` and returns `t` with each atom replaced by `f` of that atom.
- Write a procedure `my-flatten` that is given a tree `t` and returns a list of the atoms in `t` in inorder-traversal order.
- In order to write `my-flatten` easily, we need to be able to concatenate two lists. Write this first, as procedure `concat`.

Done. See `data.scm`.

Recall. Every Scheme value is **null** or a **pair** or an **atom**. So any value for which `null?` and `pair?` both return `#f` is an atom.