Introduction (1 of 2)

- Scheme and Common Lisp are the two main dialects of the original Lisp, which was designed by John McCarthy (MIT), in 1958.
- Only Fortran is older: by one year. Fortran has changed, drastically. Essentially, Lisp has not changed.
- The "pure" subset of Lisp is functional (i.e., no side effects), and is based on the λ-calculus of Alonzo Church (1930).
- Imperative features (i.e., assignments and loops) have been added, but purists ignore them. In Scheme, their names end with a bang (e.g., `set-car!`).
- Jorge Santayana said: "Those who cannot remember the past are condemned to repeat it." Lisp programmers tease: "Those who don't know Lisp are doomed to reinvent it, poorly."

Introduction (2 of 2)

- Scheme has an extremely simple syntax, used for both programs and data.
- Scheme is strongly and dynamically typed.
- Scheme is statically scoped.
- Scheme is higher-order. Functions are first-class objects, which can be constructed and evaluated during execution.
- Scheme is embeddable. From Wikipedia: “Guile is used in programs such as GnuCash, LilyPond, GNU Guix, GNU Debugger, GNU TeXmacs and Google’s Schism.” For example:

  `pub/etc/GNUmakefile`

Program structure (1 of 4)

- As with many PLs, a Scheme program is a set of function definitions, followed by a sequence of function calls.
- Function definition:
  - Java:

    $t_f\ f(t_1\ p_1,\ t_2\ p_2,\ \cdots,\ t_n\ p_n)\ \{\ \mathit{body}\ \}$
  - Scheme:

    `(define (`$f\ p_1\ p_2\ \cdots\ p_n$`)` body`)`
- Function call:
  - Java:

    $f(p_1,\ p_2,\ \cdots,\ p_n)$
  - Scheme:

    $(f\ p_1\ p_2\ \cdots\ p_n)$

Program structure (2 of 4)

- Here's a simple example:

  `pub/sum/scheme/sum.scm`
- Apparently, symbols (e.g., function names) can contain funny characters.
- Quotation: What's with that creepy apostrophe? It's just an abbreviation: `'x` means `(quote x)` and `'(···)` means `(quote (···))`.
- The built-in function `quote` returns its (one) parameter, unevaluated.

Program structure (3 of 4)

- But wait! A function definition is apparently just a call to a function that defines a function (e.g., `define`).
- So, nested function definitions are natural.

  `pub/sum/scheme/sumtail.scm`
- Does `define` cause a side effect? Yes.
- Can `define` redefine a symbol? Yes.
- Is `define` an imperative feature? Yes, if misused.

Program structure (4 of 4)

- Enough about functions, already! What about variables?
- If you think about it, a function definition and a variable definition both simply bind a value to a symbol (i.e., its name). The difference is the value's type.
- How shall we denote a callable value?
- How about `(lambda` $\cdots$), since λ isn't on your keyboard:

  `pub/etc/lambda.scm`

Syntax and Semantics

- We've seen some examples. Now, let's consider a five-rule grammar for Scheme:

  `pub/etc/scheme-grammar`
- A program is a sequence of symbolic expressions, called *S-expressions*.
- A literal, of course, evaluates to itself.
- A symbol evaluates to its defined value.
- A parenthesized sequence of S-expressions evaluates to the result of calling the value of the first (i.e., a function) and passing it the values of the rest (i.e., as parameters). That is the *only* meaning of parentheses!
- Some parameters to some functions are not evaluated prior to the call (e.g., `quote`, `lambda`, conditionals, and logicals).

Translation

- A program is input, analyzed, and executed by the *read-eval-write* loop, which calls three built-in functions of the same names.
- `read` reads a complete S-expression from `stdin`, and returns it. If it is parenthesized, `read` constructs a list to contain it.
- `eval` evaluates and returns the result of `read`, as described earlier.
- `write` writes the result of `eval` to `stdout`.
- For example:

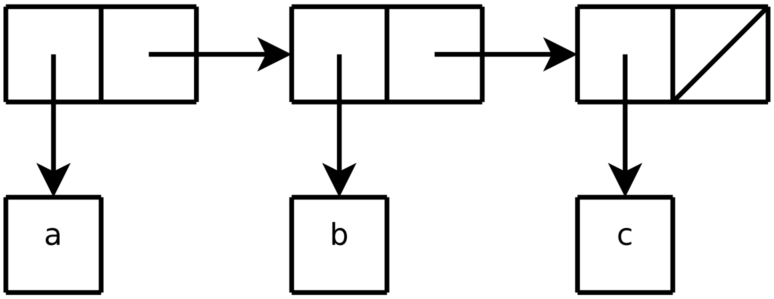
  `pub/etc/repl.scm`

List representation (1 of 2)

- Suppose `read` reads this S-expression:

  `(a b c)`

  It will construct, and return a reference to, a list that looks like this:



- The left and right fields of a cell are called the `car` and `cdr`, respectively, due to the IBM 701 registers of those names.
- One possible implementation:

  `pub/etc/Pair.java`

List representation (2 of 2)

- Notice that we did not need to explicitly end the list with an empty-list, “null”, or “nil” value. That’s because `(a b c)` is an abbreviation for:
  ```
  (a b c . ’())
  ```
  This dotted-pair syntax is rarely needed, but you might see it if you build your lists improperly.
- If this list was passed to `eval`, it would evaluate the symbol `a`, discover that its value is not a function, and fail.
- We could pass any of these S-expressions to `read` and `eval` to produce our list:
  ```
  ’(a b c)
  (quote (a b c))
  (list ’a ’b ’c)
  (cons ’a (cons ’b (cons ’c ’())))
  ```