The Scheme Language

History of Scheme

- 1958: Lisp language invented by John McCarthy
  (based on Church's lambda calculus, alternative to Turing machines)
- 1958: Steve Russell writes `eval` in machine code,
  creates first Lisp interpreter
- 1962: First Lisp *compiler*, written in Lisp
- 1970s, 80s, 90s: Lisp is the dominant language for AI research
- 1975: Scheme created by Steele & Sussman:
  minimal Lisp dialect focused on functional programming
- 1985: *Structure and Interpretation of Computer Programs*:
  teaching Scheme in first-year at MIT
- 1991: *How to Design Programs*:
  teaching Scheme to beginners based on *design recipes*

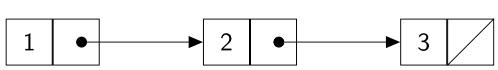
Scheme building blocks

From Lab 01:

- Syntax: `(`*`procedure arg1 arg2`*` ...)`
- Arithmetic: `+`, `*`, `remainder`, etc.
- Logic: `and`, `or`, `not`, `<`, etc.
- `define`: Create constants and functions
- `if` and `cond`
- `cons`, `car`, `cdr`

Lists in Scheme

Remember how a singly-linked list works:



Making linked lists in Scheme:

- Use `cons` for every node
- Use `'()` for the empty list

How to write the list above?

Using and building lists

- `'()` is an empty list.
- For an item `a` and list `L`, `(cons a L)` produces a list starting with `a`, followed by all the elements in `L`.
- `(car L)` produces the first thing in a non-empty list `L`.
- `(cdr L)` produces a list with the first item of `L` removed.
- Interpreter prints the list `(cons 1 (cons 2 (cons 3 '())))` as `(1 2 3)`
- Lists can be nested.

Useful list functions

- `(list a b c ...)` builds a list with the elements `a`, `b`, `c`, ...
- `cXXXr`, where `X` is `a` or `d`. Shortcut for things like `(cdr (car (car (cdr L))))` → `(cdaadr L)`
- `(pair? L)` — returns true iff `L` is a cons.
- `(null? L)` — returns true iff `L` is an empty list.
- `(append L1 L2)` — returns a list with the elements of `L1`, followed by those of `L2`.
  Can you write this function?

Recursion on lists

Here is a general pattern for writing a recursive function that processes a list:

```
(define (list-fun L)
  (if (null? L)

      ; Base case for empty list goes here
      0

      ; Recursive case goes here.
      ; Get the recursive call and do something with it!
      (+ 1 (list-fun (cdr L)))))
```

Symbols

Scheme has a new data type: **symbols**:

- They are kind of like strings
- Except they're **immutable** (can't be altered)
- Somewhat similar to `enum`'s in C.
- Usually symbols are short words (no spaces)
- The predicate `symbol?` is useful!
- Use `eqv?` for comparisons.

To make a symbol, use a single quote: `'these 'are 'all 'symbols '!`

Typical Uses

- Names from a short list (months, weekdays, grades, . . .)
- Used to *tag* data: `(cons 10.3 'feet)`

Quoting

The single quote `'` is a shorthand for the `quote` function.
So `(quote something)` is the same as `'something`.

Quoting in Scheme means "*don't evaluate this*"
— and it's really useful!

What do you think `(quote (1 2 3))` would produce?
How else could you get the same thing?

Quoting Lists

Quote is the reason why `'()` means an empty list.
You can also use it for a nonempty list: `'(a b c)`.

Quote also works *recursively*, so we can make nested lists: `'(1 (2 3) 4)` is equivalent to `(list 1 (list 2 3) 4)`

What do you think this program will produce?

```
(define x 3)
'(1 2 x)
(list 1 2 x)
```

The need for local variables

This code finds the largest number in a list:

```
(define (lmax L)
  (cond ((null? (cdr L)) (car L))
        ((>= (car L) (lmax (cdr L))) (car L))
        (else (lmax (cdr L)))))
```

What's the worst-case running time?
How could we fix it?

The `let` special form

Scheme provides `let` as a way to re-use temporary values:

```
(define (lmax L)
  (if (null? (cdr L))
      (car L)
      (let ((rest-max (lmax (cdr L))))
        (if (>= (car L) rest-max)
            (car L)
            rest-max))))
```

Note the **extra parentheses** — to allow multiple definitions:
`(let ((a 5) (b 6)) (+ a b))` ⇒ 11

Components of Programs

The basic building blocks of any programming language are atoms, values, expressions, and statements.

Of course they are related:

- Every atom is a value.
- Every value is an expression.
- Expressions specify the data in statements.
- A program is a series of statements.

Atoms and Values

An atom is an indivisible piece of data.
Sometimes these are called "literals".
Examples of atoms: numbers, chars,...

A value is any fixed piece of data..
Values include atoms, but can also include more complicated things like: arrays, lists,...

Expressions and Statements

An expression is code that *evaluates to* a value.
Examples: arithmetic, function calls,...

A statement is a stand-alone complete instruction.

- In Scheme, every expression is also a statement.
- In C++, most statements end in a semicolon.

Scheme grammar

Here is a CFG for the Scheme syntax we have seen so far:

CFG for Scheme

$exprseq \rightarrow expr \mid exprseq\ expr$
$expr \rightarrow atom \mid (\ exprseq\ )$
$atom \rightarrow identifier \mid number \mid boolean$

This is incredibly simple!

Scheme is lists!

Everything in Scheme that looks like a list. . . *is a list*!
Scheme evaluates a list by using a general rule:

- First, turn a list of expressions `(e1 e2 e3 ...)` into a list of values `(v1 v2 v3 ...)` by recursively evaluating each `e1`, `e2`, etc.
- Then, apply the procedure `v1` to the arguments `v2`, `v3`, . . .

Can you think of any exceptions to this rule?
What if `v1` is not a procedure?

Special Forms

The only exceptions to the evaluation rule are the **special forms**.

Special forms we have seen: `define`, `if`, `cond`, `and`, `or`.

What makes these “special” is that they
do not (always) evaluate (all) their arguments.

Example: evaluating `(5)` gives an error, but
`(if #f (5) 6)` just returns 6 — it never evaluates the “`(5)`” part.

Scheme evaluation and unevaluation

We can use the built-in function `eval` to evaluate a Scheme expression within Scheme!

- Try `(eval (list + 1 2))`
- Even crazier: `(eval (list 'define 'y 100))`

What is the opposite (more properly, the *inverse*) of `eval`?

This makes Scheme *homoiconic* and *self-extensible*