Scheme

- categorize Scheme into parts:
  - primitives `(if...) (lambda ()...)` - small but functional set
  - library `(not...) (and...)`
    - `and` - macro for something simpler
- required
  - integers
- optional (implementer can supply these)
  - multiple precision floating point
- extensions (supplied by impl but not standard)

Mistakes/bugs/etc.

- implementation restrictions
  - ex: RAM, virtual mem.
- unspecified behavior
  - ex: `(eq ? 0 (- 2 2))` - wrong equality check used
  - some Scheme implementations account for this so its unspecified
- programs should be robust

`(eq ? A B)`	`(eqv ? A B)`	`(equal ? A B)`	`(= A B)`
O(1)	O(N)	∞	O(N)
pointer comparison	'contents' comparison (non-recursive)	recursive map	numeric comparison

- an error is signaled
  - ex: `(open-input-file "foo") (car 39) # signals error`
- undefined behavior
  - implementation can do anything

Scheme interpreter needs two things:

- ip (instruction pointer)
  - context for instructions

- ep (equivalent pointer)
    - what to do when function returns
    - what context to use after returning
- list of everything we want from the instructed

Green Thread

```
(define gtlist `())
(define (gt-cons thunk)
    (set! gt-list (append gt-list (list thunk))))

(define (start)
    (let ((next-gt) (car gt-list)))
    (set! gt-list (cdr gt-list))
    (next-gt))

(let (yield)
    (call/cc
    (lambda (k)
        (gt-cons k)
        (start))))

(gt-cons (lambda () (let f () (display "h") (yield) (f))))
(gt-cons (lambda () (let f () (display "i") (yield) (f)))))
(gt-cons (lambda () (let f () (newline) (yield) (f))))
(start)
```

Can do continuation passing style

- slower but works in any language that supports high level functions

Storage/Memory Management

We need to store:

- contents of variables (esp. big ones)
- return addresses (ip)
- environment pointers (ep)
- instructions
- I/O Buffers

- partition memory into 3 (basic) areas

constants read-only instr	text
	data/ initialized data
	bss (zeroed data)
grows downward	heap, new, malloc
	unused
grows upward	stack

Fortran 1958

- all variables, frames, etc. allocated statically
- pros:
    - simple
    - no memory exhaustion
    - fast
- cons:
    - must trust all code
    - no recursion
    - inflexible

C (1975)

- allocate fixed size activation records (frames) on stack
- local variables can live in frame --> allows recursion
- malloc/free - manage objects on heap
- pros:
    - free objects in any order
- cons:
    - more expensive

Algol 60

- like C, but local array size determined when you declare