Programming Languages — Scheme and High-level Concerns

syntax Rules for producing valid sequences of tokens in a language. *How* to write valid sentences.
semantics Rules for interpreting valid sequences of tokens in a language. *What* valid sentences **mean**.

static anything that is *known* (or *knowable*) **before** load time; *e.g.* size of Java `int`, type of C++ variable, address of global function `main` in C.
dynamic anything that is not static; *e.g.* value of `argv[1]` for C program, memory address of a *local* variable in a stack-based language, the type of a Python variable.

Question: define *recursion* in terms of a programming language *function* or subprogram.
Question: is the *return address* of a function *static* or *dynamic*? Where would it be stored to support **recursive** functions?

Abstract Data Type (ADT)

An abstract data type has:

- interface — the collection of world-facing operations
- implementation — the collection of internal fields and functions in memory
- encapsulation — limiting external interaction to the interface; complete hiding of the implementation from client code

Question: How does this relate to object-oriented programming in Java? How is the implementation hidden?
Question: How is a Java `float` an ADT?
Question: What is the *interface* to an array? (No, the type of elements **in** the array is immaterial to the answer.)

`scheme`

Scheme is a Lisp dialect. Programs are written in *prefix* notation with the operator coming before the operands in a list. All lists are enclosed in parentheses. A list, an *s-expression* is both the fundamental data type in Scheme *and* the fundamental building block of code in the language. This simplifies writing Scheme to manipulate Scheme.
Environment is the lookup table for definitions. Think of the calling stack for Java. Environments are linked and each `define` or `lambda` or `let` introduces a new one. Remember that an environment is linked to a parent environment.
Question: What is the parent environment when you call a function? How does that relate to the idea of a *closure*?
Higher-level functions are functions that take other functions as *parameters* or return them as *results*.

```
(define make-adder (n)
  (lambda (x) (+ x n))) => make-adder
(map (make-adder 1) '(10 20 30 40)) => (11 21 31 41)
```

Language Processing Programs

Our computers are

- digital — values selected from a discrete set

- binary — the set is $\{0, 1\}$
- general-purpose — arbitrary *types* can be encoded, including **instructions** that direct the processor

Language processors lie along a spectrum:

compiler Reads a description of an algorithm or process in a (typically high-level) language and translates it to a (typically lower-level) language while preserving the semantics.
In the Russian book analogy: the Russian monk who *reads the book once* and produces an *English version* and then goes home. (What happens if I want to know what was on p. 10 a **second** time?)
interpreter Reads *and performs* a description of an algorithm in a language. *Perform* means to execute steps matching the semantics of the algorithm as written. Notice that the performance happens as the program is read.
In the Russian book analogy: the Russian monk who *reads a page* and translates it to *English*, jumping to whatever following page they need to and doing it again. The monk can never go home. (What happens if I want to know what was on p. 10 a **second** time?)

Question: What do you know about *hybrid* language processors in the middle of the spectrum? How are tasks divided in time and between compiling and interpreting?
A **macro** is a text-transformation function. Think of it like a form-letter template. When processed (perhaps with parameters), the macro processor generates new text from the template. In programming terms, that generated text is compiled/interpreted as the program to execute. The macro processor in this model comes before the lexical and syntactic processing.
Question: C++ was originally written as a *preprocessor* that took as input C++ and produced as output C which could then be fed to a standard C compiler to produce an executable. Explain this in terms of the language processing spectrum.

Helpful Information

Remember that you will get this in the exam but you need to understand it.

```
;; define is used to define new names.
(define x 10)
(define double (lambda (x) (* x 2)))

;; quote quotes literal data (symbols or lists)
;; the tick-mark ' is syntactic sugar
(quote hello)          => hello
(quote (1 2 3))        => (1 2 3)
'(1 2 foo bar)         => (1 2 foo bar)

;; lambda is used to generate new functions
(lambda (x) (+ x 10))  ; an anonymous function
(define plus10 (lambda (x) (+ x 10)))

;; if is a two-branch conditional
(if (equal? '(+ 5 8) 13)
  'fibonacci
  'non-fibonacci)              => fibonacci

;; cond is a general conditional
(cond
  ((eq? 'foo 'bar) 'hello)
  ((= 10 20) 'goodbye)
  (#t 'sorry))                 => sorry

;; let, let*, letrec are for locals
(let
  ((x 10)
   (y 20))
  (+ x y))
                => 30
;; --- not your usual length ---
(letrec ; -- let recursive
  ((length (lambda (lst)
            (if (null? lst)
                0
                (+ 2 (length (cdr lst)))))))
  (length '(1 2 3 4))
)                              => 8

;; begin is the sequencing construct
(begin
  (* 1300 (- 567 391))
  (sqrt 127000)
  (+ 2 2)
)                              => 4

;; arithmetic:  +, -, *, /, quotient, modulo
;; relational: <, <=, >, >=, =
(quotient 87 9)           => 9
(= 1 2)                   => #f   ; = for numbers
```

```
;; Equality and identity:  eq? and equal?
(eq? 'hello 'goodbye)      => #f   ; identity test
(eq? 'hello 'hello)        => #t
(eq? '(1 2) '(1 2))        => #f
(define foo '(1 2))
(define foo bar)
(eq? foo bar)              => #t
(equal? foo bar)           => #t   ; if they look the same
(equal? foo '(1 2))        => #t

;; Lists:  cons, car, and cdr
;; Making new lists, via quoting, cons, or list
(define foo '(1 2 3))
(define bar (cons 1 (cons 2 (cons 3 ()))))
(define baz (list 1 2 3))

;; Process lists with car, cdr, and null?
(null? '(1 2))             => #f
(null? ())                 => #t
(car '(1 2))               => 1
(cdr '(1 2))               => (2)

;; takes two single parameter functions, f and g
;; returns the f composed g function.
(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x)))))

;; applies f to every element of the-list
(define map
  (lambda (f the-list)
    (if (null? the-list)
      the-list
      (cons (f (car the-list))
            (map f (cdr the-list))))))

(map even? '(1 2 3 4))        => (#f #t #f #t)

;; association lists
(define e '((a 1) (b 2) (c 3) (7 g)))
(assq 'a e)                            => (a 1)
(assq 'b e)                            => (b 2)
(assq 'd e)                            => #f
(assoc 7 e)                            => (7 g)
(assq (list 'a) '(((a)) ((b)) ((c))))  => #f
(assoc (list 'a) '(((a)) ((b)) ((c)))) => ((a))
```

Cribbed from `https://courses.cs.washington.edu/courses/cse341/02wi/scheme/cheat-sheet.html`