CS 565 August 24, 1997
Jens Palsberg Readings: EOPL, Chap. 1–2, 3.4–3.6

Lecture Note 1:

Functional Programming in Scheme

Contents

1 The Scheme Language

1.1 Basics

Interaction loop.

```
3

(+ 1 3)

(a b c)

'(a b c)

(define x 3)

x

(+ x 1)

(define l (a b c))

(define l '(a b c))

(define u (+ x 1))
```

Creating procedures with lambda:

```
(lambda (x) body)

(lambda (x) (+ x 1))

((lambda (x) (+ x 1)) 4)

(define mysucc (lambda (x) (+ x 1)))

(mysucc 4)

(define myplus (lambda (x y) (+ x y)))

(+ 3 4)
((lambda (x y) (+ x y)) 3 4)
(myplus 3 4)
```

Procedures can take other procedures as arguments:

```
((lambda (f x) (f x 3))
 myplus 5)
```

Procedures can return other procedures; this is called Currying:

```
(define twice
  (lambda (f)
    (lambda (x)
      (f (f x)))))

(define add2 (twice (lambda (x) (+ x 1))))

(define f-of-x+1
  (lambda (f)
    (lambda (x)
      (f (+ x 1)))))

(f-of-x+1 add2)  -- takes x, sends x+1 to add2

((f-of-x+1 add2) 3) = (add2 4) = 6

(f-of-x+1 (f-of-x+1 add2))  -- a procedure!

((f-of-x+1 (f-of-x+1 add2)) 4)

= 8
```

— Much more on this later.

1.2 List Processing

Kinds of data:

Symbols: sequence of letters and digits starting with a letter. The sequence can also include other symbols, such as `-`, `$`, `=`, `*`, `/`, `?`, `_`. See the Scheme manual for details.

Numbers: integers (for now)

Strings: `"this is a string"` – we'll leave this until later.

Basic values = Symbols ∪ Numbers ∪ Strings ∪ Lists

Lists:

(1) the empty list is a list `()`

(2) a sequence `(`s_0 ... s_{n-1}`)` where each s_i is a value (either a symbol, number, string, or list

(3) nothing is a list unless it can be shown to be a list by rules (1) and (2).

This is an inductive definition, which will play an important part in our reasoning. We will often solve problems (e.g. write procedures on lists) by following this inductive definition. Will see this later.

Examples of lists:

```
(turkey)
(atom)
((turkey) (atom))
xyz -- a symbol, not a list
(xyz)
(((turkey) (atom)) (xyz))
(x y z (u v blab))
```

```
((byrd "byrd" "b" "y" "r" "d")
 (magic "magic" "m" "a" "g" "i" "c"))
()
```

Putting list literals in programs:

```
'(a b c)
'()
```

Basic operations on lists:

`car`: if `l` is (s_0 ... s_{n-1}), then `(car l)` is s_0. The `car` of the empty list is undefined.

```
e.g. (define l '(a b c))    (car l) => a
```

`cdr`: if `l` is ($s_0 s_1$ ... s_{n-1}), then `(cdr l)` is (s_1 ... s_{n-1}). The `cdr` of the empty list is undefined.

```
(cdr l) => (b c)
```

Examples of car and cdr on previous lists.

Combining car and cdr:

```
(car (cdr l))
(cdr (cdr l))
etc.
```

```
(define first car)
(define second (lambda (l) (car (cdr l))))
(define second (compose car cdr))
etc.
```

Building lists:

`cons`: if `s` is the value s, and `l` is the list ($s_0 \ldots\ s_{n-1}$), then `(cons s l)` is the list ($s\ \ s_0 \ldots\ s_{n-1}$)

Cons builds a list whose car is s and whose cdr is l.

```
(car (cons s l)) = s
(cdr (cons s l)) = l
```

```
cons : value * list -> list
car  : list -> value
cdr  : list -> list
```

1.3 Booleans

Literals:

`#t, #f`

Predicates:

```
(number? s)
(symbol? s)
(string? s)
(null? s)
(pair? s)
(eq? s1 s2)      --works on symbols
(= n1 n2)        --works on numbers
(zero? n)
(> n1 n2)
etc
```

Conditional:

```
(if bool e1 e2)
```

1.4 Defining Procedures Recursively

Imagine we want to define a procedure to find powers, eg.

$$e(n, x) = x^n \ .$$

It is easy to define a bunch of procedures $e_0(x) = x^0$, $e_1(x) = x^1$, $e_2(x) = x^2$:

$$\begin{aligned}
e_0(x) &= 1 \\
e_1(x) &= x \times e_0(x) \\
e_2(x) &= x \times e_1(x) \\
e_3(x) &= x \times e_2(x)
\end{aligned}$$

and in general, we can define

$$e_n(x) = x \times e_{n-1}(x)$$

How did we do this? At each stage, we used the fact that we had already solved the problem for smaller n. This is the principle of mathematical induction.

How can we parameterize this construction to get a single procedure e?

If `n` is 0, we know the answer: `(e 0 x) = 1`.

If n is greater than 0, assume we can solve the problem for $n-1$. Then the answer is `(e n x) = (* x (e (- n 1) x))`.

```
(define e
  (lambda (n x)
    (if (zero? n)
        1
        (* x
           (e (- n 1) x)))))
```

Why does this work? Let's prove it works for any n, by induction on n:

(1) It surely works for n=0.

(2) Now assume (for the moment) that it works when `n` $= k$. Then it works when `n` $= k + 1$. Why? Because then `(e n x) = (* x (e` k `x))`, and we know e works when its first argument is k. So it gives the right answer when its first argument is $k+1$.

What's the moral? If we can reduce the problem to a smaller subproblem, then we can call the procedure itself ("recursively") to solve the smaller subproblem. Then, as we call the procedure, we ask it to work on smaller and smaller subproblems, so eventually we will ask it about something that it can solve directly (eg n=0, the basis step), and then it will terminate successfully.

Principle of structural induction: If you always recur on smaller problems, then your procedure is sure to work.

```
(e 3 2)
= (* 2 (e 2 2))
= (* 2 (* 2 (e 1 2)))
= (* 2 (* 2 (* 2 (e 0 2))))
= (* 2 (* 2 (* 2 1)))
= 8
```

```
(define fact
  (lambda (n)
    (if (zero? n) 1 (* n (fact (- n 1))))))
```

```
   (fact 4)
= (* 4 (fact 3))
= (* 4 (* 3 (fact 2)))
= (* 4 (* 3 (* 2 (fact 1))))
= (* 4 (* 3 (* 2 (* 1 (fact 0)))))
= (* 4 (* 3 (* 2 (* 1 1))))
= (* 4 (* 3 (* 2 1)))
= (* 4 (* 3 2))
= (* 4 6)
= 24
```

This is the natural recursive definition of factorial. Each call of fact is made with a promise that the value returned will be multiplied by the value of n at the time of the call. Thus fact is invoked in larger and larger control contexts as the calculation proceeds.

Compare this behavior with that of the following procedures.

```
(define fact-iter
  (lambda (n)
    (fact-iter-acc n 1)))

(define fact-iter-acc
  (lambda (n a)
    (if (zero? n) a (fact-iter-acc (- n 1) (* n a)))))

   (fact-iter 4)
= (fact-iter-acc 4 1)
= (fact-iter-acc 3 4)
= (fact-iter-acc 2 12)
= (fact-iter-acc 1 24)
= (fact-iter-acc 0 24)
= 24
```

Here, fact-iter-acc is always invoked in the same context (in this case, no context at all). When fact-iter-acc calls itself, it does so at the "tail end" of a call to fact-iter-acc. That is, no promise is made to do anything with the returned value other than return it as the result of the call to fact-iter-acc. Thus each step in the derivation above has the form `(fact-iter-acc` n a`)`. This will be vital later on.

Let's try this on a list processing problem.

Procedure: `nth-elt`

Input: a list l and an integer k.

Output: Returns the k-th element of list l, (counting from 0). If l does not contain sufficiently many elements, an error is signaled.

When k is 0, we know the answer: `(car l)`. If k is greater than 0, then the answer is the $k-1$st element of `(cdr l)`. The only glitch is that we have to *guard* the `car` and `cdr` operations:

```
(define nth-elt
  (lambda (l k)
    (if (null? l)
      (error nth-elt "nth-elt:  ran off end")
      (if (zero? k)
        (car l)
        (nth-elt (cdr l) (- k 1))))))
```

This is the same as the procedure `list-ref`, except that `list-ref` does not signal errors.

```
(nth-elt '(a b c d e) 3)
= (nth-elt '(b c d e) 2)
= (nth-elt '(c d e) 1)
= (nth-elt '(d e) 0)
= d
```

Procedure: `list-of-symbols?`

Input: a list l

Output: true if l consists entirely of symbols, false if it contains a non-symbol.

```
(define list-of-symbols?
  (lambda (l)
    (if (null? l)
        #t
        (if (symbol? (car l))
            (list-of-symbols? (cdr l))
            #f))))
```

Procedure: `member?`

Input: a symbol i and a list l of atoms.

Output: true if l contains the symbol i as one of its members, false otherwise.

```
(define member
  (lambda (i l)
    (if (null? l) #f
        (if (eqv? i (car l)) #t
            (member i (cdr l))))))
```

Procedure: `remove-first`

Input: a symbol i and a list l of atoms.

Output: A list the same as l, except that the first occurrence of i has been removed. If there is no occurrence of i, the list will be unchanged.

```
(define remove-first
  (lambda (i l)
    (if (null? l) '()
        (if (eqv? i (car l))
            (cdr l)
            (cons (car l)
                  (remove-first i (cdr l)))))))
```

Procedure: `remove`

Input: a symbol i and a list l of atoms.

Output: A list the same as l, except that *all* occurrences of i have been removed. If there is no occurrence of i, the list will be unchanged.

```
(define remove
  (lambda (i l)
    (if (null? l) '()
        (if (eqv? i (car l))
            (remove i (cdr l))
            (cons (car l)
                  (remove i (cdr l)))))))
```

Procedure: `remove-all`

Input: An symbol i and a list l

Output: Like `remove`, except that all occurrences of i in l or any of its sublists (at any level) are to be removed.

```
(define remove-all
  (lambda (i l)
    (if (null? l) '()
        (if (atom? (car l))
            (if (eqv? i (car l))
                (remove-all i (cdr l))
                (cons (car l) (remove-all i (cdr l))))
            (cons (remove-all i (car l))
                  (remove-all i (cdr l)))))))
```

1.5 Syntactic Abstraction: Common special forms

cond

In `remove-all` we see some common patterns of usage. First, there is the pattern:

```
(if test1 exp1
    (if test2 exp2
        exp_n))
```

This is a common language construct– the many-way branch. We have a syntactic abbreviation for this pattern:

```
(cond
  (test1 exp1)
  (test2 exp2)
  ...
  (else exp_n))
```

So we can rewrite `remove-all` as

```
(define remove-all
  (lambda (i l)
    (cond
      ((null? l) '())
      ((atom? (car l))
       (if (eqv? i (car l))
           (remove-all i (cdr l))
           (cons (car l) (remove-all i (cdr l)))))
      (else (cons (remove-all i (car l))
                  (remove-all i (cdr l)))))))
```

let

Lots of times we need *local names* for things. For this we use the special form `let`:

```
(let ((var1 val1)
      (var2 val2)
      ...)
  exp)
```

For example, we write

```
(let ((n 25))
  (let ((x (sqrt n))
        (y 3))
    (+ x y)))

= (let ((x (sqrt 25))
        (y 3))
    (+ x y))

= (let ((x 5)
        (y 3))
    (+ x y))

= (+ 5 3)

= 8
```

```
(let ((x 3))
  (let ((y (+ x 4)))
    (* x y)))

= (let ((y (+ 3 4)))
    (* 3 y))

= (let ((y 7))
    (* 3 y))

= (* 3 7) = 21
```

compare to:

```
(let ((x 3)
      (y (+ x 4)))
  (* x y))
```

Here the occurrence of `x` on the second line refers to whatever the enclosing value of `x` happens to be– maybe set by another `let`.

So we can rewrite `remove-all` as

```
(define remove-all
  (lambda (i l)
    (cond
      ((null? l) '())
      ((atom? (car l))
       (let ((rember-cdr (remove-all i (cdr l))))
         (if (eqv? i (car l))
             rember-cdr
             (cons (car l) rember-cdr))))
      (else (cons (remove-all i (car l))
                  (remove-all i (cdr l)))))))
```

`Let`'s can be nested– see EOPL Sec 2.3; this is important.

Using `let`, we can give names to local procedures and pass them around:

```
(let ((f (lambda (y z) (* y (+ y z))))
      (g (lambda (u) (+ u 5))))
  (f (g 3) 17))

(let ((x 5))
  (let ((f (lambda (y z) (* y (+ x z))))
        (g (lambda (u) (+ u x)))
        (x 38))
    (f (g 3) 17)))

(let ((f (lambda (y) (y (y 5))))
      (g (lambda (z) (+ z 8))))
  (f g))

(let ((f (lambda (y z) (y (y z))))
      (g (lambda (z) (+ z 8))))
  (f g 5))
```

1.6 Defining Local Recursive Procedures

Lots of times we'd like to define recursive procedures locally, like `fact-iter` above.

```
(define fact-iter
  (lambda (n)
    (fact-iter-acc n 1)))

(define fact-iter-acc
  (lambda (n a)
    (if (zero? n) a (fact-iter-acc (- n 1) (* n a)))))
```

You'd like to write

```
(define fact-iter
  (lambda (n)
    (let ((fact-iter-acc
            (lambda (n a)
              (if (zero? n)
                  a
                  (fact-iter-acc (- n 1) (* n a))))))
      (fact-iter-acc n 1))))
```

But we can't do that, because the scope of `fact-iter-acc` doesn't include it's definition. Instead, we use `letrec`:

```
(letrec
 ((name1 proc1)
  (name2 proc2)
  ...)
 body)
```

Letrec creates a set of mutually recursive procedures and makes their names available in the body.

So we can write:

```
(define fact-iter
  (lambda (n)
    (letrec ((fact-iter-acc
               (lambda (n a)
                 (if (zero? n) a
                   (fact-iter-acc (- n 1) (* n a))))))
      (fact-iter-acc n 1))))
```

Here's another example:

```
(define remove
  (lambda (i l)
    (if (null? l) '()
      (if (eq? i (car l))
        (remove i (cdr l))
        (cons (car l)
              (remove i (cdr l)))))))
```

can be replaced by

```
(define remove
  (lambda (i l)
    (letrec
      ((loop
         ;; (loop l) = (remove i l)
         (lambda (l)
           (if (null? l) '()
             (if (eq? i (car l))
               (loop (cdr l))
               (cons (car l)
                     (loop (cdr l))))))))
      (loop l))))
```

Another example:

```
(define linear-search
  (lambda (pred list fn)
    (letrec
      ((loop
         ;; (loop list) = (linear-search pred list fn)
         (lambda (list)
               (cond
                 ((null? list) #f)
                 ((pred (car list)) (fn (car list)))
                 (else (loop (cdr list)))))))
      (loop list))))
```

1.7 Sequencing

Want to have some finer control over the order in which things are done in Scheme. We need this for worrying about (a) side-effects and (b) termination. In general, there is only one rule about sequencing in Scheme:

Arguments are evaluated before procedure bodies.

So in

```
((lambda (x y z) body) exp1 exp2 exp3)
```

`exp1`, `exp2`, and `exp3` are guaranteed to be evaluated before `body`, but we don't know in what order `exp1`, `exp2`, and `exp3` are going to be evaluated, but they will all be evaluated before `body`.

This is **precisely** the same as

```
(let ((x exp1) (y exp2) (z exp3))
  body)
```

In both cases, we evaluate `exp1`, `exp2`, and `exp3`, and then we evaluate `body` in an environment in which `x`, `y`, and `z` are bound to the values of `exp1`, `exp2`, and `exp3`.

Think about this. It is important.

2 Data Types and their Representations in Scheme

1. Want to define new data types. This comes in two pieces: a *specification* and an *implementation*. The specification tells us *what* data (and what operations on that data) we are trying to represent. The implementation tells us *how* we do it.

2. We want to arrange things so that you can change the implementation without changing the code that uses the data type (user = client; implementation = supplier/server).

3. Both the specification and implementation have to deal with two things: the data and the operations on the data.

4. Vital part of the implementation is the specification of how the data is represented. We will use the notation $\lceil v \rceil$ for "the representation of data v".

2.1 Numbers

Data Specification: the non-negative integers.

Specification of Operations:

$$
\begin{aligned}
\texttt{zero} &= \lceil 0 \rceil \\
(\,\texttt{is-zero?}\ \lceil n \rceil\,) &= \begin{cases} \texttt{\#t} & n = 0 \\ \texttt{\#f} & n \neq 0 \end{cases} \\
(\,\texttt{succ}\ \lceil n \rceil\,) &= \lceil n+1 \rceil \\
(\,\texttt{pred}\ \lceil n+1 \rceil\,) &= \lceil n \rceil
\end{aligned}
$$

Now we can write procedures to do the other arithmetic operations, and these will work no matter what representation of the natural numbers we use.

```
(define plus
  (lambda (x y)
    (if (is-zero? x) y
      (succ (plus (pred x) y)))))
```

will satisfy $(\,\texttt{plus}\ \lceil x \rceil\ \lceil y \rceil\,) = \lceil x+y \rceil$, no matter what implementation of the integers we use.

Implementations:

1. Unary representation:

$$\lceil 0 \rceil = \texttt{( )}$$
$$\lceil n+1 \rceil = \texttt{(cons \#t } \lceil n \rceil \texttt{)}$$

So the integer n is represented by a list of n `#t`'s.

It's easy to see that we can satisfy the specification by writing:

```
(define zero '())

(define is-zero? null?)

(define succ
  (lambda (n) (cons #t n)))

(define pred cdr)
```

2. Scheme number representation:

$\lceil n \rceil$ = the Scheme integer `n`

```
(define zero 0)
(define is-zero? zero?)
(define succ (lambda (n) (+ n 1)))
(define prec (lambda (n) (- n 1)))
```

3. Bignum representation (Base N, least-significant bigit first):

$$\lceil n \rceil = \begin{cases} \texttt{()} & n = 0 \\ \texttt{(cons } r \ \lceil q \rceil \texttt{)} & n = qN + r,\ 0 \leq r < N \end{cases}$$

So if $N = 16$, $\lceil 33 \rceil$ = `(1 2)`. Exercise: write the operations in this implementation.

2.2 Finite Functions

Data: A function whose domain is a finite set of Scheme symbols, and whose range is unspecified: $\{(s_1, v_1), \ldots, (s_n, v_n)\}$.

Specification of operations

Interface:

```
;;; interface finite-function =
;;;   type ff
;;;   empty-ff : ff
;;;   apply-ff : ff * key -> value
;;;   extend-ff : key * val * ff -> ff
;;;
```

$$
\begin{array}{ll}
\texttt{empty-ff} & = \lceil \emptyset \rceil \\
(\ \texttt{apply-ff}\ \lceil f \rceil\ s\,) & = f(s) \\
(\ \texttt{extend-ff}\ s\ v\ \lceil f \rceil\,) & = \lceil g \rceil, \\
 & \quad \text{where } g(s') = \begin{cases} v & s' = s \\ f(s') & \text{otherwise} \end{cases}
\end{array}
$$

This tells us what procedures are available for dealing with finite functions: it defines the *interface*. The interface gives the type of each procedure and a description of the intended behavior of each procedure.

Procedural representation

```
;;; Procedural Representation
;;; uses ff = key -> value
```

$\texttt{f} = \lceil\{(s_1, v_1), \ldots, (s_n, v_n)\}\rceil$ iff $(\texttt{f}\ \ s_i) = v_i$.
Implement the operations by:

```
(define apply-ff
  (lambda (ff z) (ff z)))

(define empty-ff
  (lambda (z)
    (error 'env-lookup (format "couldn't find ~s" z))))

(define extend-ff
  (lambda (key val ff)
    (lambda (z)
      (if (eq? z key)
        val
        (apply-ff ff z)))))
```

```
;;; Examples

(define ff-1 (extend-ff 'a 1 empty-ff))
(define ff-2 (extend-ff 'b 2 ff-1))
(define ff-3 (extend-ff 'c 3 ff-2))
(define ff-4 (extend-ff 'd 4 ff-3))
(define ff-5 (extend-ff 'e 5 ff-4))

> ff-5
<Procedure>
> (apply-ff ff-5 'd)
4
> (apply-ff empty-ff 'c)
error in env-lookup: couldn't find c.
> (apply-ff ff-3 'd)
error in env-lookup: couldn't find d.
```

Abstract syntax tree representation

$$\langle \text{ff-tree} \rangle ::= (\ \texttt{empty-ff}\ )$$
$$::= (\ \texttt{extended-ff}\ \langle \text{symbol} \rangle\ \langle \text{value} \rangle\ \langle \text{ff-tree} \rangle\ )$$

$$\lceil \{\} \rceil = ()$$
$$\lceil \{(s_1, v_1), \ldots, (s_n, v_n)\} \rceil = (\texttt{extended-ff}\ s_1\ v_1\ \lceil \{(s_2, v_2), \ldots, (s_n, v_n)\} \rceil)$$

```
(define-record empty-ff ())
(define-record extended-ff (key val ff))

(define empty-ff (make-empty-ff))

(define extend-ff
   (lambda (key val ff)
     (make-extended-ff key val ff)))

(define apply-ff
  (lambda (tree z)
    (record-case tree
      (empty-ff ()
        (error 'env-lookup (format "couldn't find ~s" z)))
      (extended-ff (key val ff)
        (if (eq? z key) val (apply-ff ff z))))))
```

```
(define ff-1 (extend-ff 'a 1 empty-ff))
(define ff-2 (extend-ff 'b 2 ff-1))
(define ff-3 (extend-ff 'c 3 ff-2))
(define ff-4 (extend-ff 'd 4 ff-3))
(define ff-5 (extend-ff 'e 5 ff-4))
```

```
> ff-1
(extended-ff a 1 (empty-ff))
> ff-2
(extended-ff b 2
  (extended-ff a 1 (empty-ff)))
> ff-3
(extended-ff c 3
  (extended-ff b 2
    (extended-ff a 1 (empty-ff))))
> ff-4
(extended-ff d 4
  (extended-ff c 3
    (extended-ff b 2
      (extended-ff a 1 (empty-ff)))))
> ff-5
(extended-ff e 5
  (extended-ff d 4
    (extended-ff c 3
      (extended-ff b 2
        (extended-ff a 1 (empty-ff))))))
```

Association-list representation

$\lceil\{(s_1, v_1), \ldots, (s_n, v_n)\}\rceil =$ `((`s_1 `.` v_1`)` `...` `(`s_n `.` v_n`))`

```
(define empty-ff '())

(define extend-ff
  (lambda (key val ff)
    (cons (cons key val) ff)))

(define apply-ff
  (lambda (alist z)
    (if (null? alist)
      (error 'env-lookup (format "couldn't find ~s" z))
      (let ((key (caar alist))
            (val (cdar alist))
            (ff (cdr alist)))
        (if (eq? z key) val (apply-ff ff z))))))

(define ff-1 (extend-ff 'a 1 empty-ff))
(define ff-2 (extend-ff 'b 2 ff-1))
(define ff-3 (extend-ff 'c 3 ff-2))
(define ff-4 (extend-ff 'd 4 ff-3))
(define ff-5 (extend-ff 'e 5 ff-4))

> ff-5
((e . 5) (d . 4) (c . 3) (b . 2) (a . 1))
```

Also be sure to study: Section 3.6. Especially note `extend-ff*` and the ribcage implementation, which we will not do here.

2.3 Modeling a Simple Stack Machine

Let's use these ideas to model a stack machine. The machine will have two components: an *action* and a *stack*.

The stack contains the data in the machine. We'll represent the stack as a list of Scheme values, with the top of the stack at the front of the list.

The action represents the instruction stream being executed by the machine. We'll use a number of different representations of the action. For the moment, we'll think of it as a string of instructions:

$$
\begin{aligned}
\langle\text{action}\rangle ::= \; & \texttt{halt} \\
\mid \; & \texttt{incr}\ ;\ \langle\text{action}\rangle \\
\mid \; & \texttt{add}\ ;\ \langle\text{action}\rangle \\
\mid \; & \texttt{push}\ \langle\text{literal}\rangle\ ;\ \langle\text{action}\rangle \\
\mid \; & \texttt{read}\ ;\ \langle\text{action}\rangle \\
\mid \; & \texttt{zero?}\ ;\ (\ \langle\text{action}\rangle\ )\ \langle\text{action}\rangle
\end{aligned}
$$

Our major operation on actions and stacks will be `apply-action`. *apply-action* should take an action and a stack and return the value produced by the machine at the completion of the action. We will adopt the convention that the machine produces a value by leaving it on the top of the stack when it halts.

Specification of operations

We want to write a *specification* for `apply-action`. The specification should say what (`apply-action` c s) does for each possible value of c.

`halt` is easy. It should halt and return the top element of the stack:

```
(apply-action halt s) = (car s)
```

Note this is NOT Scheme code; it's an equation (a specification!)

The `incr` instruction is supposed to increment the top element of the stack. We can write this as a specification:

```
(apply-action incr;a (v w ...)) =
   (apply-action a (v+1 w ...))
```

Similarly, `add` is supposed to add the top 2 elements of the stack:

```
(apply-action add;a (v w x ...)) =
   (apply-action a ((v+w) x ...))
```

`push` pushes a value on the stack:

```
(apply-action push v;a (w ...)) =
   (apply-action a (v w ...))
```

These are specifications– equations written in an informal language. To write Scheme code to implement `apply-action`, we need to specify a representation of the type of actions. Let's do that the simplest way:

$$\begin{aligned}
\lceil \texttt{halt} \rceil &= \texttt{(halt)} \\
\lceil \texttt{incr} \,;\, a \rceil &= \texttt{(incr } \;\; \texttt{.} \;\; \lceil a \rceil \texttt{)} \\
\lceil \texttt{push } v \;\; ; \;\; a \rceil &= \texttt{(push } v \;\; \texttt{.} \;\; \lceil a \rceil \texttt{)}
\end{aligned}$$

so an action is represented as a list of instructions. This is like bytecode: a typical action is

```
(push 3 push 4 add halt)
```

Now we can write `apply-action` for this representation:

```
;;; Grammar for action representation
;;; a ::= (halt) |alt (incr . a) |alt (push v . a)

(define apply-action
  (lambda (action stack)
    (let ((instruction (car action))
          (next-action (cdr action)))
      (case instruction
        ((halt)
         (car stack))
        ((incr)
         (apply-action next-action
           (cons (+ (car stack) 1)
                 (cdr stack))))
        ((push)
         (let ((v (cadr action))
               (next-action (cddr action)))
           (apply-action next-action
             (cons v stack))))
        ((add)
         (apply-action next-action
           (cons (+ (car stack) (cadr stack))
                 (cddr stack))))
        (else
          (error 'apply-action
            (format "unknown instruction ~s"
                    instruction)))))))
```

```
(define start
  (lambda (action)
    (apply-action action '())))
```

```
> (start '(push 3 push 4 add halt))
7
```

Adding new instructions to the machine

The `read` instruction is supposed to read a value from the standard input and put it on the stack. We can do this by adding another entry to the `case` expression:

```
((read)
 (let ((v (prompt-read "machine>")))
   (apply-action next-action
     (cons v stack))))
...
```

```
;; auxiliary procedure:
(define prompt-read
  (lambda (prompt)
    (printf "~d " prompt)
    (read)))

> (start '(read read add halt))
machine> 3
machine> 4
7
```

The `zero?` instruction is supposed to see if the top of the stack is zero.

```
(apply-action zero?(a1)a2 (0 v ...)) = (apply-action a1 (0 v ...))
(apply-action zero?(a1)a2 (x v ...)) = (apply-action a2 (x v ...))
                                              [x != 0]
```

In our representation, this comes out as

```
((zero?)
 (let ((true-action (cadr action))
       (false-action (cddr action)))
   (if (zero? (car stack))
     (apply-action true-action stack)
     (apply-action false-action stack))))
```

```
> (start '(read zero? (halt) read add halt))
machine> 0
0
> (start '(read zero? (halt) read add halt))
machine> 2
machine> 3
5
```

Changing representations

It is easy to change the representation of actions; just need to change `apply-action` to decode the representation and do the right thing.

For example, we could change the representation so that `halt` was represented as the empty list:

$$\lceil \texttt{halt} \rceil = \texttt{( )}$$

Then we would change `apply-action` to say:

```
(define apply-action
  (lambda (action stack)
    (if (null? action)
      (car stack)
      (let ((instruction (car action))
            (next-action (cdr action)))
        (case instruction
          ((incr)
           (apply-action next-action
             (cons (+ (car stack) 1)
                   (cdr stack))))
          ...
          )))))
```