CS 565 August 23, 1996
Jens Palsberg Readings: EOPL, Sec 3.4–3.6

Lecture Note 2:

Data Types and their Representations

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".

Example: Arithmetic

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:

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

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 (lambd (n) (- n 1)))
```

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

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

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

Finite Functions

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

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
```

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

$\langle \text{ff-tree} \rangle$ ::= (`empty-ff`)
 ::= (`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
          "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

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

```
(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 "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.

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.

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:

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

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) | (incr . a) | (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
            "unknown instruction ~s" instruction))))))
```

Testing the machine

```
(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:

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

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))))
          ...
          )))))
```