2.3 Representation Strategies for Data Types

- *Environment*: Data type associating symbols with values

  { *a*: 4 *s*: 'foo *f*: λx.x }

- Interface:

  (empty-env) = [Ø]

  (apply-env [*f*] *s***)** = *f*(*s*)

  (extend-env '(s_1 ... s_k**) '(** v_1 ... v_k**)** [*f*]**)** = [*g*]

where *g*(*s*') = v_i if *s*' = s_i for some *i*, $1 \le i \le k$

= *f* (*s*') otherwise

Example:

```
> (define dxy-env
    (extend-env '(d x) '(6 7)
      (extend-env '(y) '(8)
        (empty-env))))

> (apply-env dxy-env 'x)
7
```

- A *constructor* builds a datatype: **`empty-env`**, **`extend-env`**
- An *observer* extracts info from a datatype: **`apply-env`**
- We will consider three representations for environments:
  - Procedural
  - Abstract Syntax Tree
  - “Alternative” (list-based)

- In Scheme, procedures (functions) are first-class objects: can be passed to / returned from other procedures, and stored in data structures

- E.g., we can represent numbers as procedures:

[0] = λs.λz.z

[1] = λs.λz.(s z)

[2] = λs.λz. (s (s z))

[succ] = λn.λs.λz. (s (n s z))

Now compute (succ 0) ....

```
(define empty-env
    (lambda ()
        (lambda (sym)
            (eopl:error 'apply-env
                        “No binding for ~s” sym))))
```

- An environment is a function that takes a symbol and returns its assoicated value.
- Empty environment defines no symbols.
- Therefore empty environment returns an error if you try to apply it to a symbol

Environment as procedure:

```
(define apply-env
        (lambda (env sym)
            (env sym)))
```

So **`apply-env`** is really just “syntactic sugar”

Environment as procedure:

```
(define extend-env
        (lambda (syms vals env)
            (lambda (sym)
                ; get position of sym in syms
                (let ((pos (list-find-pos sym syms)))
                    (if (number? pos) ; we found it!
                        ; return value at that pos
                        (list-ref vals pos)
                        ; not found; use current env
                        (apply-env env sym))))))
```

```
> (define xy-env
  (extend-env '(x y) '(3 4) (empty-env)))
> xy-env
#<procedure:13:7>
> (apply-env xy-env 'y)
4
> (apply-env xy-env 'z)
apply-env: No binding for z
```

Environment as Abstract Syntax Tree:

<env-rep> ::= **(empty-env)**

empty-env-record

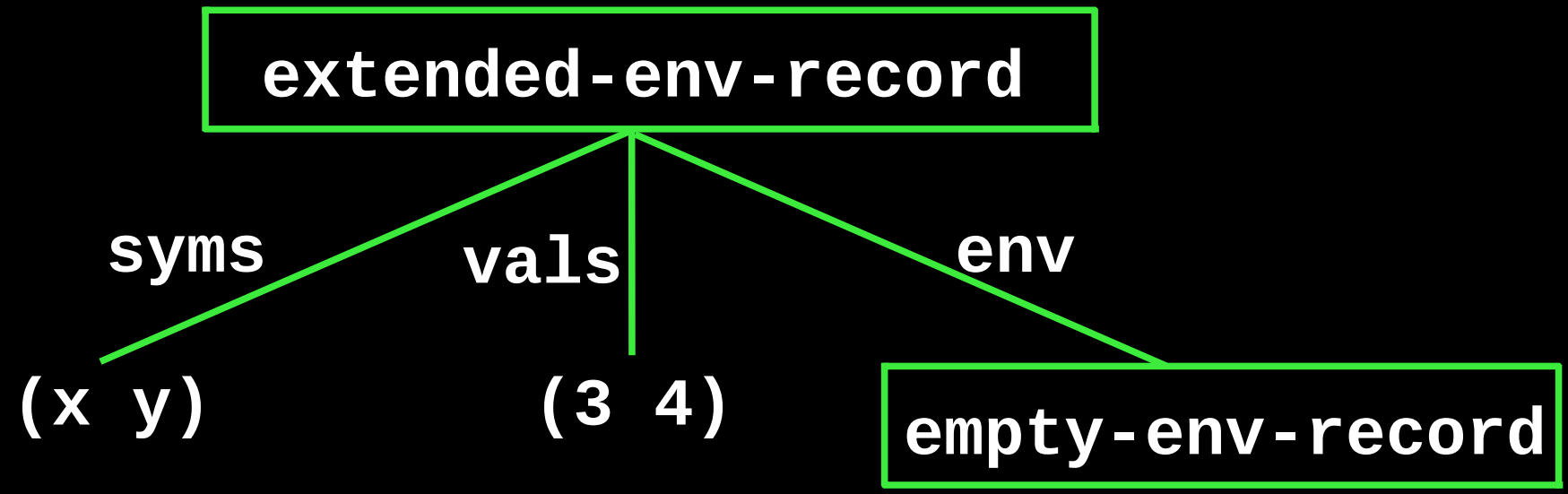
::= **(extend-env** **(**{<symbol>}***)**

({<value>}***)**

<env-rep>**)**

extended-env-record (syms vals env

Environment as AST:



Environment as AST:

```
(define-datatype environment environment?
    (empty-env-record)
    (extended-env-record
        (syms (list-of symbol?))
        (vals (list-of scheme-value?))
        (env        environment?)))


(define scheme-value? (lambda (v) #t)

> ((list-of number?) '(1 2 3))
#t
```

Environment as AST:

```
(define empty-env
    (lambda ()
        (empty-env-record)))

(define extend-env
    (lambda (syms vals env)
        (extended-env-record syms vals env)))
```

Environment as AST:

```
(define apply-env
   (lambda (env sym)
      (cases environment env
         (empty-env-record ()
            (eopl:error ...))
         (extended-env-record (syms vals env)
            (let ((pos (list-find-pos sym syms)))
               (if (number? pos)
                  (list-ref vals pos)
                  (apply-env env sym)))))))
```

Alternative (List) Representation

Actually the most intuitive:

<env-rep> ::= **()**

::= **((**{<symbol>}*)** **(**{<value>}*)** <env-rep>**)**

```
(define empty-env
   (lambda ()
      '()))
(define extend-env
   (lambda (syms vals env)
      (cons (list syms vals) env)))
```

Alternative Representation

```
(define apply-env
     (lambda (env sym)
          (if (null? env)
               (eopl:error ...)
               (let     (syms (car (car env)))
                        (vals (cadr (car env)))
                        (env (cdr env)))
                             (let ((pos (rib-find-pos sym syms)))
                                  (if (number? pos)
                                       (list-ref vals pos)
                                       (apply-env env sym)))))))

(define rib-find-pos list-find-pos)
```

Alternative Representation: Ribcage