INDUCTIVE SETS OF DATA

INDUCTIVE SPECIFICATION

a method of specifying a set of values.

(dfn) ex a natural number n is in S if & only if
1. n = 0, or
2. n-3 ∈ S

} top-down approach

0 ∈ S
(3-3) = 0 ∈ S ⇒ 3 ∈ S
(6-3) = 3 ∈ S ⇒ 6 ∈ S
...
1 ∉ S (1-3) = -2 ∉ S
4 ∉ S (4-3) = 1 ∉ S

∴ we conclude that S is the set of natural numbers that are multiples of 3.

contract
in-S? : N → Bool
usage: (in-S? n) = #t if n is in S, #f otherwise.

```
(define in-S?
  (lambda (n)
    (if (zero? n) #t
      (if (>= (- n 3) 0)
        (in-S? (- n 3))
        #f))))
```

alternative:

```
(define in-S?
  (lambda (n)
    (if (< n 0) #f
        (if (zero? n) #t
            (in-S? (- n 3))))))
```

(dfn) ex define the set S to be the smallest set contained in N and satisfying the following 2 properties:

1. $0 \in S$, and
2. if $n \in S$, then $n+3 \in S$

} bottom-up definition

rules of inference version

$$\frac{}{0 \in S} \qquad \frac{n \in S}{(n+3) \in S}$$

hypothesis / antecedent

if-then

conclusion / consequent

axiom (no hypothesis)

rule of inference (rule)

implicit "and" if > 1 mentioned.

(dfn) ex (list of integers, top-down)

a scheme list is a list of integers if and only if either:

1. it is the empty list
2. it is a pair whose car [0] is an integer and whose cdr [1...end] is a list of integers.

Int: set of all integers, List-of-Int: set of lists of integers

(dfn) ex (list of integers, bottom-up)

the set List-of-Int is the smallest set of scheme lists satisfying the following 2 properties:

1. () ∈ List-of-Int, and
2. if n ∈ Int and l ∈ List-of-Int, then (n . l) ∈ List-of-Int.
   ↳ "cons" (construct list)

(dfn) ex

$$\frac{}{() \in \text{List-of-Int}} \qquad \frac{n \in \text{Int} \quad l \in \text{List-of-Int}}{(n.l) \in \text{List-of-Int}}$$

ex 1. () — list of integers

2. (14 . ()) — list of integers

$$\frac{14 \in \text{Int} \quad () \in \text{List-of-Int}}{(14.\ ()) \in \text{List-of-Int}}$$

3. (3 . (14 . ())) — list of integers.

4. (-7 . (3 . (14 . ()))) — list of integers.

$$\frac{3 \in \text{Int} \quad (14\ .\ ()) \in \text{List-of-Int}}{(3\ .\ (14\ .\ ())) \in \text{List-of-Int}}$$

$$\frac{-7 \in \text{Int} \quad (3\ .\ (14\ .\ ())) \in \text{List-of-Int}}{(-7\ .\ (3\ .\ (14\ .\ ()))) \in \text{List-of-Int}}$$

5. nothing is a list of integers unless it is built in this fashion

converting from dot notation to list notation

(), (14), (3 14), (-7 3 14) are all members of List-of-Int.

$$\frac{\dfrac{\dfrac{14 \in N \quad () \in \text{List-of-Int}}{3 \in N \quad (14 . ()) \in \text{List-of-Int}}}{-7 \in N \quad (3 . (14 . ())) \in \text{List-of-Int}}}{(-7 . (3 . (14 . ()))) \in \text{List-of-Int}}$$

derivation / deduction tree.

E1.1 write inductive definitions of the following sets. write each definition in all 3 styles (top-down, bottom-up, rules-of-inference) using your rules, show the derivation of some sample elements of each set.

1. $\{3n+2 \mid n \in N\}$ ← set builder notation
2. $\{2n+3m+1 \mid n, m \in N\}$
3. $\{(n, 2n+1) \mid n \in N\}$
4. $\{(n, n^2) \mid n \in N\}$

do not mention squaring in your rules. as a hint, remember the equation $(n+1)^2 = n^2 + 2n + 1$

1. $\{3n+2 \mid n \in N\}$ ← S

$\{2, 5, 8, \ldots\}$

top-down definition:

a natural number n is in S if and only if

1. $n = 2$, or
2. $n - 3 \in S$

$2 \in S$

$5 - 3 = 2 \in S \Rightarrow 5 \in S$

$8 - 3 = 5 \in S \Rightarrow 8 \in S$

bottom-up definition:

Set S is the smallest set contained in N that satisfies the following properties:

1. $2 \in S$, and
2. if $n \in S$, then $n+3 \in S$.

rules of inference definition:

$$\frac{n \in S}{n+3 \in S} \qquad 2 \in S$$

$$\frac{\frac{(2) \in S}{2+3 = (5) \in S}}{5+3 = (8) \in S}$$

$\vdots$

2. $\{ 2n + 3m + 1 \mid n, m \in N \}$

$\{ 1, 3, 4, 5, 7, \ldots \}$

top-down definition:

a natural number n is in S if and only if:

1. $n = 1$, or
2. $n - 2 \in S$, or
3. $n - 3 \in S$

bottom-up definition:

Set S is the smallest set contained in N that satisfies the following properties:

1. $1 \in S$, and
2. if $n \in S$, then $n+2 \in S$ and $n+3 \in S$

rules of inference definition:

$$\frac{}{1 \in S} \qquad \frac{n \in S}{n+2 \in S} \qquad \frac{n \in S}{n+3 \in S}$$

examples:

$$\frac{\frac{\frac{\frac{}{1 \in S}}{3 \in S} \quad \frac{}{4 \in S}}{5 \in S} \quad \frac{}{6 \in S} \quad \frac{}{6 \in S} \quad \frac{}{7 \in S}}{7 \in S} \quad \frac{}{8 \in S} \quad \frac{}{8 \in S} \quad \frac{}{9 \in S} \quad \frac{}{9 \in S} \quad \frac{}{10 \in S}$$

3. $\{(n, 2n+1) \mid n \in N\}$

top-down approach:

a number pair (n, m) is in S if and only if

1. $n=0, m=1$, or
2. $(n-1, m-2) \in S$

bottom-up approach:

Set S is the smallest set contained in $N \times N$ that satisfies the following properties:

1. $(0, 1) \in S$, and
2. if $(n, m) \in S$, then $(n+1, m+2) \in S$

rules of inference definition:

$$\frac{}{(0, 1) \in S} \qquad \frac{(n, m) \in S}{(n+1, m+2) \in S}$$

examples:

$$\frac{(0,1) \in S}{(1,3) \in S}$$

$$\frac{(2,5) \in S}{(3,7) \in S}$$

4. $\{(n, n^2) \mid n \in N\} \longleftarrow S$

$\{(0,0), (1,1), (2,4), (3,9), \ldots\}$

top-down definition:

a number pair (n, m) is in S if and only if:

1. $n=0, m=0$; or
2. $(n-1, m-2n+1) \in S$

bottom-up definition

set S is defined as the smallest set contained in $N \times N$ that satisfies the following properties:

1. $(0,0) \in S$
2. if $(n,m) \in S$, then $(n+1, m+2n+1) \in S$

rules of inference definition:

$$(0,0) \in S \qquad \frac{(n,m) \in S}{(n+1, m+2n+1) \in S}$$

examples:

$$\frac{(0,0) \in S}{(1,1) \in S}$$

$$\frac{(2,4) \in S}{(3,9) \in S}$$

E1.2 what sets are defined by the following pairs of rules? explain why.

1. $(0,1) \in S$ $\quad \dfrac{(n,k) \in S}{(n+1, k+7) \in S}$

2. $(0,1) \in S$ $\quad \dfrac{(n,k) \in S}{(n+1, 2k) \in S}$

3. $(0,0,1) \in S$ $\quad \dfrac{(n,i,j) \in S}{(n+1, j, i+j) \in S}$

4. $(0,1,0) \in S$ $\quad \dfrac{(n,i,j) \in S}{(n+1, i+2, i+j) \in S}$

1. $(0,1) \in S$ $\quad \dfrac{(n,k) \in S}{(n+1, k+7) \in S}$

$S = \{ (n, 7n+1) \mid n \in N \}$

because with every step, n increments by 1, and k by 7.

2. $(0,1) \in S$ $\quad \dfrac{(n,k) \in S}{(n+1, 2k) \in S}$

$S = \{ (n, 2^n) \mid n \in N \}$

because with every step, n increments by 1, and k is multiplied by 2 (here gets reflected as a 2^n).

3. $(0,0,1) \in S$ $\quad \dfrac{(n,i,j) \in S}{(n+1, j, i+j) \in S}$

$S = \{ (n, fib(n), fib(n+1)) \mid n \in N \}$

where $fib(n)$ = nth fibonacci number

because with every step, n increments by 1, and the other 2 numbers follow the fibonacci series.

4. $(0, 1, 0) \in S \qquad \dfrac{(n, i, j) \in S}{(n+1, i+2, i+j) \in S}$

$\{(0, 1, 0), (1, 3, 1), (2, 5, 4), (3, 7, 9) \ldots\}$

$S = \{(n, 2n+1, n^2) \mid n \in N\}$

where because i increments by 2 every step and starts with 1 and j increments itself by that odd number (i) every step ... and everyery n^2 is separated by $2n+1$, hence the term for j is n^2.

E1.3 find a set T of natural numbers such that $0 \in T$, and whenever $n \in T$, then $n+3 \in T$, but $T \neq S$, where S is the set, defined in definition 1.1.2.

Key here to remember is that T is not the smallest set, and hence we can add some elements of our choice to it.

we know $0 \in T \Rightarrow 3, 6, 9, \ldots \in T$

lets add 1, 2 as well $\Rightarrow 4, 7, 10, \ldots \in T$

$\Rightarrow 5, 8, 11, \ldots \in T$

$\therefore T = N$ and $T \neq S$

DEFINING SETS USING GRAMMARS

dfn grammars are typically used to define/specify sets of strings, but we can use them to define sets of values as well:

grammar (compilers): List-of-Int ::= () ① ← production
List-of-Int ::= (Int . List-of-Int) ②

(nt sym: List-of-Int; terminal sym: ())

old $() \in$ List-of-Int

$$\frac{n \in \text{Int} \qquad l \in \text{List-of-Int}}{(n.l) \in \text{List-of-Int}}$$

① empty list is in List-of-Int
② if n is in Int, and l in List-of-Int, then (n.l) is in List-of-Int

dfn non-terminal symbols: (syntactic categories) (compilers)
names of sets being defined
Can use backus-naur form BNF ⟨expression⟩.

dfn terminal symbols:
characters in external representation → . C)

dfn productions:
the rules. LHS = non-terminal symbol.
RHS = non-terminal symbols ② + terminal symbols ①.
::= → is / can be

often some syntactic categories mentioned in a production are left undefined. when their meaning is sufficiently clear from context (Int).

notational shortcuts

List-of-Int ::= ()
::= (Int . List-of-Int).

or

List-of-Int ::= () | (Int . List-of-Int)

Kleene star

List-of-Int ::= ({Int}*)

{...}* → any number of instances (0+)
{...}+ → one or more instances (1+)

Kleene plus

{Int}*(,) ← separated list notation

8

14 , 12

7 , 3 , 14 , 16

{Int}*(;)

8

14 ; 12

7 ; 3 ; 14 ; 16

notational shortcuts are not essential. it is always possible to recorite grammar without them.

dfn List-of-Int
⇒ (Int . List-of-Int)
⇒ (14 . List-of-Int)
⇒ (14 . ())

} syntactic derivation

List-of-Int
⇒ (Int . List-of-Int)
⇒ (Int . ())
⇒ (14 . ())

E 1.4 Write a derivation from List-of-Int to (-7 . (3 . (14 . ())))

List-of-Int
⇒ (Int . List-of-Int)
⇒ (-7 . List-of-Int)
⇒ (-7 . (Int . List-of-Int))
⇒ (-7 . (3 . List-of-Int))
⇒ (-7 . (3 . (Int . List-of-Int)))
⇒ (-7 . (3 . (14 . List-of-Int)))
⇒ (-7 . (3 . (14 . ())))

dfn many symbol manipulation procedures are defined to operate on symbols and similarly restricted lists. → s-lists

S-list ::= ({S-exp}*)
S-exp ::= Symbol | S-list

ex (a b c)

(an (((s-list)) (with () lots) ((of) nesting)))

dfn binary tree with numeric leaves and interior nodes labeled with symbols.

Bintree ::= Int | (Symbol Bintree Bintree)

ex 1

2

(foo 1 2)

(bar 1 (foo 1 2))

(baz
  (bar 1 (foo 1 2))
  (biz 4 5))

dfn lambda calculus is a language that consists of variable references, procedures that take a single argument, and procedure calls.

bound variable

LcExp ::= Identifier
  ::= (lambda (Identifier) LcExp)
  ::= (LcExp LcExp)

Identifier → any symbol other than lambda.

ex (lambda (x) (+ x 5))

dfn ↖ bound variable ↑

it binds / captures any occurences of variable in the body.

ex ((lambda (x) (+ x 5)) (- x 7))

↑ bound variable ↑ not bound

dfn these grammars are said to be "contextfree" because a rule defining a given syntactic category may be applied in any context that makes reference to that syntactic category.

sometimes this may not be restrictive enough:

Binary-search-tree ::= () | (Int Binary-search-tree Binary-search-tree)

✓ describes structure

x (ignores) all keys in left subtree <= value of current node,

all keys in right subtree > value of current node

→ context-sensitive constraints / invariants.

INDUCTION

having described sets inductively, we can use the inductive definitions in 2 ways →

- to prove theorems about members of sets.
- to write programs that manipulate them.

thm let t be a binary tree, then t contains an odd number of nodes. ← induction hypothesis IH

1. for 0 nodes, IH(0) = ● true.
2. for integer K, say IH(K) = true.
   (any tree with ≤ K nodes has odd no. of nodes).
3. for nodes = K+1

   ~~the new node could be Int~~

   t could be Int ⇒ 1 node ⇒ odd.

   t could be (sym tree$_1$ tree$_2$)

nodes(t_1, t_2) < nodes(t)

since nodes(t) ≤ K+1

⇒ nodes(t_1), nodes(t_2) ≤ K

⇒ odd odd

∴ total no. of nodes = odd + odd + 1 = odd.

∴ IH(K+1) holds ✓.

the key to the proof is that the substructures of a tree t are always smaller than t itself. this pattern of proof is called Structural induction.

retry:

thm let t be a binary tree, then t contains an odd no. of nodes.

let size(t) = no. of nodes in tree t.

induction hypothesis IH(K) = tree of size ≤ K
⇒ t has odd no. of nodes.

IH(K) = size(t) ≤ K
⇒ size(t) = odd.

1. for 0 nodes tree, IH(0) holds trivially.

2. for K node tree,

1. IH(0) holds trivially, there are no trees with 0 nodes.

2. IH(K) K∈ integer, any tree with ≤ K nodes has odd no. of nodes (assumption).

3. RTP IH(K+1) holds

tree t could be of the form

(a) Int ← integer. one node ⇒ odd.

(b) (Sym t_1 t_2)
↑ ↖ ↑
symbol btree

now size(t_1) + size(t_2) + 1 = size(t)
⇒ size(t_1) ≤ K & size(t_2) ≤ K
⇒ size(t_1) = odd & size(t_2) = odd.

∴ size(t) = odd + odd + 1 = odd.

this completes the proof that IH(K+1) holds, and therefore completes the induction.

retry:

thm let t be a binary tree, then t contains an odd no. of nodes.

let size(t) = no. of nodes in tree t

IH(k): size(t) ≤ k ← integer

⇒ size(t) = odd

induction hypothesis

} lets assume it holds for some k.

base case:

no trees with 0 nodes, IH(0) holds trivially.

inductive assumption:

assume IH(k) holds for some k ← integer.

size(t) ≤ k

⇒ size(t) = odd

RTP IH(k+1) holds:

from the definition of binary tree t, it can be of the forms

(a) t = Int ⇒ size(t) = 1 = odd (IH(k+1) holds)

(b) t = (sym t_1 t_2)

sym: symbol; t_1, t_2: trees

∴ size(t) = 1 + size(t_1) + size(t_2)

size(t) ≤ k+1 ⇒ size(t_1) ≤ k & size(t_2) ≤ k

⇒ size(t_1) = odd & size(t_2) = odd (from IH(k))

∴ size(t) = 1 + odd + odd

= odd

∴ this completes the proof that IH(k+1) holds, and therefore completes the induction.

PROOF BY STRUCTURAL INDUCTION

to prove that a proposition IH(s) is true for all structures s, prove the following.

1. IH is true on simple structures (those without substructures).
2. if IH is true on the substructures of s, then it is true on s itself.

E is prove that if $e \in$ LcExp, then there are the same no. of left and right parentheses in e.

LcExp ::= Identifier
::= (lambda (Identifier) LcExp)
::= (LcExp LcExp)

~~left brackets(e) > no. left brackets in LcExp e.~~

let diff(e) = no. of left parentheses in e −
no. of right parentheses in e ← LcExp

IH(k): expansions(e) $\leq$ k

...

LcExp ::= Identifier
::= (lambda (Identifier) LcExp)
::= (LcExp LcExp).

let left(e) = no. of left parentheses in LcExp e.
right(e) = no. of right parentheses in LcExp e.

IH(K): left(e) + right(e) ≤ K
⇒ left(e) = right(e).

base case:
e = Identifier
⇒ left(e) = 0
right(e) = 0
∴ left(e) = right(e)
(IH(0) holds trivially).

inductive assumption:
IH(K): left(e) + right(e) ≤ K for some integer K.
⇒ left(e) = right(e)

RTP (inductive step) IH(K+1) holds:
from the definition of LcExp e, it can be of the forms:

(a) e = Identifier
⇒ left(e) = right(e)

(b) e = (lambda (Identifier) e_1)
left(e) = 2 + left(e_1)
right(e) = 2 + right(e_1)
as left(e) ≤ K+1

retry:

let left (e) = no. of left parentheses in LcExp e

right (e) = no. of right parentheses in LcExp e

IH (k): $\max\{ left(e), right(e) \} \leq k$

$\Rightarrow$ left (e) = right (e)

base case:

e = Identifier

$\Rightarrow$ left (e) = 0

right (e) = 0

$\therefore$ left (e) = right (e)

(IH (0) holds trivially).

inductive assumption:

IH (k) $\max\{ left(e), right(e) \} \leq k$ for some integer k

$\Rightarrow$ left (e) = right (e)

RTP (inductive step) IH ($k+1$) holds:

from the definition of LcExp e, it can be of the forms:

(a) e = Identifier

$\Rightarrow$ left (e) = right (e)

(b) e = (lambda (Identifier) e_1)

left (e) = 2 + left (e_1)

right (e) = 2 + right (e_1)

as left (e) $\leq k+1$ $\Rightarrow$ left (e_1) $\leq k-1$

as right(e) $\leq k+1 \Rightarrow$ right(e_1) $\leq k-1$

$\therefore \max\{\text{left}(e_1), \text{right}(e_1)\} \leq k-1$

$\Rightarrow$ left(e_1) = right(e_1) = b

$\therefore$ left(e) = b+2, right(e) = b+2

$\Rightarrow$ left(e) = right(e)

$\therefore$ IH(k+1) holds

(c) $e = (e_1\ e_2)$

$\max\{\text{left}(e), \text{right}(e)\} \leq k+1$

left(e) = left(e_1) + 1 $\Rightarrow$ left(e) $\leq k$

right(e) = right(e_1) + 1 $\Rightarrow$ right(e) $\leq k$

$\therefore$ left(e_1) $\leq k$ & right(e_1) $\leq k$

$\Rightarrow$ left(e_1) = right(e_1)

(e) $e = (e_1\ e_2)$

left(e) = 1 + left(e_1) + left(e_2)

right(e) = 1 + right(e_1) + right(e_2)

$\Rightarrow$ left(e_1) $\leq k$ & left(e_2) $\leq k$

$\Rightarrow$ right(e_1) $\leq k$ & right(e_2) $\leq k$

left(e_1) = right(e_1) = c left(e_2) = right(e_2) = d.

$\therefore$ left(e) = 1 + c + d

right(e) = 1 + c + d

$\Rightarrow$ left(e) = right(e)

IH(k+1) holds.

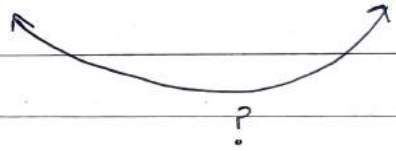
∴ this completes the proof that IH(k+1) holds, and therefore completes the induction.

DERIVING RECURSIVE PROGRAMS

we can analyze an element of an inductively defined set to see how it is built from smaller elements in the set.

THE SMALLER SUBPROBLEM PRINCIPLE

if we can reduce a problem to a smaller subproblem, we can call the procedure that solves the problem to solve the subproblem.
!! (is it correct?)



LIST-LENGTH

```
> (length '(a b c))
3
> (length '((x) ()))
2
```

Contract: set of possible arguments & possible return values.

list-length: List → Int
usage: (list-length l) = the length of l.

List ::= () | (scheme value . List)

list-length: List → Int
usage: (list-length l) = the length of l.
(define list-length
  (lambda (lst)
    (if (null? lst)
      0
      (+ 1 (list-length (cdr lst))))))

(list-length '(a (b c) d))
= (+ 1 (list-length '((b c) d)))
= (+ 1 (+ 1 (list-length '(d))))
= (+ 1 (+ 1 (+ 1 (list-length '()))))
= (+ 1 (+ 1 (+ 1 0)))
= 3

NTH-ELEMENT

> (list-ref '(a b c) 1)
b

same notation as in mathematics
f: A × B → C

nth-element: List × (Int) Scheme value → Scheme Value
usage: (nth-element lst n) = nth element of the list (0-index)
(define nth-element
  (lambda (lst n)
    (if (null? lst)

```
(report-list-too-short n)
(if (zero? n)
    (car lst)
    (nth-element (cdr lst) (- n 1))))))
```

```
(define report-list-too-short
  (lambda (n)
    (oops:error 'nth-element
                "List too short by ~s elements.~%" (+ n 1))))
```

abort

\n

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

compute/ derivation

E1.6 if we reversed the order of the tests, what would go wrong?

it would work well (as before) on non-empty list, but crash @ (car lst) or (cdr lst) depending upon the value of n. ex→ (nth-element '() 1)

also in the cases where n does not lie within the bounds of the list, it would crash, as mentioned above.
ex → (nth-element '(a b) 2)
(nth-element '(1 2 3 4) -2)

E1.7 the error message from nth-element is unifn formative. rewrite nth-element so that it produces a more informative error message, such as "(a b c) does not have 8 elements".

```
(define nth-element.internal
  (lambda (lst n)
    (if (null? lst)
        #f
        (if (zero? n)
            (car lst)
            (nth-element.internal (cdr lst) (- n 1))))))

(define nth-element
  (lambda (lst n)
    (let ([ret (nth-element.internal lst n)])
      (if (= ret #f)
          (eopl:error 'nth-element
            "~s does not have ~s elements.~%"
              lst n)
          ret))))
```

REMOVE-FIRST

```
> (remove-first 'a '(a b c))
'(b c)
```

```
> (remove-first 'b '(e f g))
'(e f g)
> (remove-first 'a4 '(c1 a4 c1 a4))
'(c1 c1 a4)
> (remove-first 'x '())
'()
```

List-of-symbol ::= () | (Symbol . List-of-Symbol)

remove-first: Sym × Listof(Sym) → Listof(Sym)
usage: (remove-first s los) returns a list with the same elements arranged in the same order as los, except that the first occurrence of the symbol s is removed.

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

E1.8 In the definition of remove-first, if the last line was replaced by (remove-first s (cdr los)), what function would the resulting procedure compute? give the contract, including the usage statement, for the revised problem.

remove-list-after: Sym × Listof(Sym) → Listof(Sym)
(...) returns list after the first occurrence of symbol s.

Eg.9 define remove, which is like remove-first, except that it removes all occurences of a given symbol from a list of symbols, not just the first.

```
(define remove
  (lambda (s los)
    (if (null? los)
        '()
        (if (eqv? (car los) s)
            (remove s (cdr los))
            (cons (car los) (remove s (cdr los)))))))
```

OCCURS-FREE?

```
> (occurs-free? 'x 'x)
#t
> (occurs-free? 'x 'y)
#f
> (occurs-free? 'x '(lambda (x) (x y)))
#f
> (occurs-free? 'x '(lambda (y) (x y)))
#t
> (occurs-free? 'x '((lambda (x) x) (x y)))
#t
> (occurs-free? 'x '(lambda (y) (lambda (z) (x (y z)))))
#t
```

LcExp ::= Identifier
::= (lambda (Identifier) LcExp)
::= (LcExp LcExp)

Ex.10 we typically use "or" to mean "inclusive or". what other meanings can "or" have?

hint: not both

exclusive-or (do you want tea or coffee?)

occurs-free?: Sym x LcExp → Bool
usage: returns #t if the symbol var occurs free in exp, otherwise returns #f.

```
(define occurs-free?
 (lambda (var exp)
   (cond
     ((symbol? exp) (eqv? exp var))
     ((eqv? (car exp) 'lambda)
           (and
             (not (eqv? (car (cadr exp)) var))
             (occurs-free? var (caddr exp))))
     (else
           (or
             (occurs-free? var (car exp))
             (occurs-free? var (cadr exp)))))))
```

SUBST

```
> (subst 'a 'b '((b c) (b () d)))
'((a c) (a () d))
```

s-list ::= ({s-exp}*)

s-exp ::= symbol | s-list

s-list ::= ()
::= (s-exp . s-list)

s-exp ::= symbol | s-list

```
(define subst-sexp
  (lambda (new old sexp)
    (if (symbol? sexp)
        (if (eqv? old sexp)
            new
            sexp)
        (subst new old sexp))))
```

```
(define subst
  (lambda (new old slist)
    (if (null? slist)
        '()
        (cons (subst-sexp new old (car slist))
              (subst new old (cdr slist))))))
```

E1.11 in the last line of subst-sexp, the recursion is on sexp and not a smaller substructure. why is the recursion guarenteed to halt?

because it calls subst with operates on 2 smaller substructures (of the s-list).

E1.12 eliminate the one call to subst-sexp in subst by replacing it by its definition and simplifying the resulting procedure. the result will be a version of subst that does not need subst-sexp. the techaique is called inlining and used by optimizing compilers.

```
(define subst
  (lambda (new old slist)
    (if (null? slist)
        '()
        (cons
          (if (symbol? slist car s
```

```
(define subst
  (lambda (new old slist)
    (if (null? slist)
        '()
        (cons
          (let ([sexp (car slist)])
            (if (symbol? sexp)
                (if (eqv? sexp old)
```

```
                new
                sexp)
            (subst new old sexp)))
        (subst new old (cdr slist))))))
```

E1.13 in our example, we began by eliminating the Kleene star in the grammar for S-list. write subst following the original grammar by using map.

```
(define subst-sexp
  (lambda (new old sexp)
    (if (symbol? sexp)
        (if (eqv? sexp old)
            new
            sexp)
        (subst new old sexp))))

(define subst
  (lambda (new old slist)
    (map (lambda (sexp)
           (subst-sexp new old sexp)) slist)))
```

FOLLOW THE GRAMMAR

when defining a procedure that operates on inductively defined data, the structure of the program should be patterned after the structure of the data.

AUXILIARY PROCEDURES AND CONTEXT ARGUMENTS

some procedures need to be generalized, with additional arguments, and auxiliary procedure, in order to solve them (number-elements).

number-elements-from: ListofSchemeVal x Int → List
ListofList(Int, SchemeVal))

usage: (number-elements-from '(v0 v1 v2 ...) n)
= ((n v0) (n+1 v1) (n+2 v2) ...)

```
(define number-elements-from
  (lambda (lst n)
    (if (null? lst)
        '()
        (cons (list n (car lst))
              (number-elements-from (+ n 1) (cdr lst))))))
```

context argument / inherited attribute

number-elements: List → ListofList(Int, SchemeVal))

```
(define number-elements
  (lambda (lst)
    (number-elements-from lst 0)))
```

NO MYSTERIOUS AUXILIARIES!

when defining an auxiliary procedure, always specify what it does on all arguments, not just the initial values.

functional language #haskell
haskell (irc) → internet relay chat
ML
OCaml

WHAT IS HASKELL

- purely functional programming language
- lazy
- static type

imperative programming
↳ is it basically defining an FSM