Rule notation:
Example 1: Natural Numbers (Nat)
---------------- (rule 1)
0 \in Nat
n \in Nat
---------------- (rule 2)
s(n) \in Nat
s(s(0)) \in Nat?
---------------- (rule 1)
0 \in Nat
---------------- (rule 2)
s(0) \in Nat
----------------- (rule 2)
s(s(0) \in Nat

Example 2: Binary trees with natural numbers at the leaves (BTree)
n \in Nat
--------------------------- (Rule 1)
leaf(n) \in BTree
l \in BTree, r \in BTree
-------------------------------- (rule 2)
node(l, r) \in BTree
node(leaf(s(0)), leaf(s(s(0))) \in BTree?
s(0) \in Nat   s(s(0)) \in Nat
---------------------(rule 1)   --------------------- (rule 1)
leaf(s(0)) \in BTree   leaf(s(s(0))) \in BTree
-- (rule 2)
node(leaf(s(0)), leaf(s(s(0))) \in BTree

Grammar Notation:
Example 1 (revisited using "grammar notation")
<Nat> ::= 0 | s(<Nat>)
<Nat> is called a non-terminal. Symbols 0, s, "(". ")", are called terminals.
<Nat> ::= 0 is called a production.
<Nat> ::= s(<Nat>) is another production
<Nat> ::= 0 | s(<Nat>) is shorthand for:
<Nat> ::= 0
<Nat> ::= s(<Nat>)
s(s(0)) \in <Nat>? yes, here is a derivation of that fact.
<Nat> --> s(<Nat>) --> s(s(<Nat>)) --> s(s(0))

Example 2 (revisited using “grammar notation”)
<BTree> ::= leaf(<Nat>) | node(<BTree>,<BTree>)
<BTree> and <Nat> are non terminals. leaf, "(", ")" ",", node are terminals
node(leaf(s(0)), leaf(s(s(0)))) \in <BTree>
<BTree> --> node(<BTree>, <BTree>) --> node(leaf(<Nat>), <BTree>)
--> node(leaf(s(0)), <BTree>)
--> node(leaf(s(0)), leaf(<Nat>))
--> node(leaf(s(0)), leaf(s(s(0))))

Define Data type:

```
(require eopl/eopl)
(define-datatype btree btree?
          (leaf (n number?))
          (node (l btree?)
                    (r btree?)))

;; btree
(define t1
     (node
          (node (leaf 1) (leaf 2))
          (node (leaf 3) (leaf 4))))

;; btree -> num
(define (sumBT t)
     (cases btree t
          (leaf (n) n)
          (node (l r) (+ (sumBT l) (sumBT r)))))

;; btree -> btree
(define (incBT t)
     (cases btree t
          (leaf (n) (leaf (+ n 1)))
          (node (l r) (node (incBT l) (incBT r)))))

;; btree -> [num]
(define (poBT t)
      (cases btree t
          (leaf (n) (list n))
          (node (l r) (append (poBT l) (poBT r)))))

;; {num -> num, btree} -> btree
(define (mapBT f t)
      (cases btree t
          (leaf (n) (leaf (f n)))
          (node (l r) (node (mapBT f l) (mapBT f
r)))))

;; btree -> btree
(define (mirrorBT t)
      (cases btree t
          (leaf (n) (leaf n))
          (node (l r) (node (mirrorBT r) (mirrorBT
l)))))

;; {num -> b, {b, b} -> b, btree} -> b
(define (foldBT f g t)
          (cases btree t
                    (leaf (n) (...))
                    (node (l r) (...))))

(require eopl/eopl)
(define-datatype env env?
          (empty-env)
          (extend-env (s symbol?)
                    (n number?)
                    (old-env env?)))
;; env
(define e1
     (extend-env 'x 3 (extend-env 'y (- 5)
          (extend-env 'z 2 (empty-env)))))

;; {sym, num, env} -> env
(define (extend-environment s n e)
          (extend-env s n e))

;; {env, sym} -> num
(define (apply-env e s)
     (cases env e
          (empty-env () #f)
          (extend-env (t n old-e) (if (eqv? s t)
                         n
                         (apply-env old-e s)))))

;; alternative (more general)
;; representation of environments

(define-datatype genv genv?
          (empty-genv)
          (extend-genv (s (list-of symbol?))
                    (n (list-of number?))
                    (old-env genv?)))

;; genv
(define e2
          (extend-genv '(x y) '(2 4)
          (extend-genv '(u v w) '(1 2 3) (empty-
genv))))

(define (applyg e s)
          (error 'applyg "undefined"))

(define (extend-genvironment e xs ns)
          (error 'extend-genviromnet "undefined"))
```

Fold:

```
(define (foldr2 f e xs)
	(match xs
		['()  e]
		[(cons h t)  (f h (foldr2 f e t))]]))

(foldr2 f e '(x1 x2 x3))
(f x1 (f x2 (f x3 e)))
(+ x1 (+ x2 (+ x3 0)))
(* x1 (* x2 (* x3 1)))

(define (foldl2 f e xs)
	(match xs
		['() e]
		[(cons h t) (foldl2 f (f e h) t)]]))

(foldl2 f e '(x1 x2 x3))
(f (f (f e x1) x2) x3)
(+ (+ (+ 0 x1) x2) x3)
(* (* (* 1 x1) x2) x3)

;; { {a, b, b} -> b, b, tree a} -> b
(define (foldT f e t)
	(match t
		[(list 'empty) e]
		[(list 'node d l r) (f d (foldT f e l) (foldT f e r))]]))
```

f is function
e is initial
xs or t is list/tree its being applied to

```
(define (f xs)
	(let ((g (lambda (x r) (if (even? x) (+ r 1) r))))
		(foldr g 0 xs)))
```

this function counts the number of evens in a list

Lambda Calculus:

<exp> ::= <identifier>
| λ<identifier>.<exp>
| (<exp> <exp>)

Examples:

y

(λx. x)

(yz)

((λx. x) (λy. y))

((λx. (x x)) (λx. (x x)))

(λx. x+x)((λy. 2∗y)4)

→ (λx. x+x)(2∗4)

→ (λx. x+x)8

→ 8+8

→ 16

Declaration vs. Reference:

λx .x

x is a declaration/formal parameter

x is a reference

Free & Bound Variables:

Bound: x is bound in an expression E if it refers to a formal parameter introduced in E

Free: x is free in E if it is not declared in E

At run-time, all variables must be either

1. lexically bound: bound by a formal parameter, or

2. globally bound: bound by a top-level definition or supplied by the system