1.a.1.

normal-eval[ (foo (goo 0)) ] 🡺

normal-eval[foo] 🡺 <Closure (x) (display x)(newline)(+ x 2)>

normal-eval[ (display (goo 0))(newline)(+ (goo 0) 2) ]

normal-eval[ (display (goo 0) ] 🡺

normal-eval[ display ] 🡺 <primitive-procedure display>

normal-eval[ (goo 0) ] 🡺

normal-eval[ goo ] 🡺 <Closure(x)(display 5)(newline)(+ x 1)>

🡺

normal-eval[ (display 5)(newline)(+ 0 1) ]

normal-eval[ (display 5) ] 🡺

normal-eval[ display ] 🡺 <primitive-procedure display>

normal-eval[ 5 ] 🡺 5

* (to screen) 5

normal-eval[ newline ] 🡺 <primitive-procedure newline>

* (to screen) \n

normal-eval[ (+ 0 1) ] 🡺

normal-eval[ + ] 🡺 <primitive-procedure +>

normal-eval[ 0 ] 🡺 0

normal-eval[ 1 ] 🡺 1

* 1

🡺 1

🡺 (to screen) 1

normal-eval[ newline ] 🡺 <primitive-procedure newline>

🡺 (to screen) \n

normal-eval[ (+ (goo 0) 2) ] 🡺

normal-eval[ + ] 🡺 <primitive-procedure +>

normal-eval[ (goo 0) ] 🡺

normal-eval[ goo ] 🡺 <Closure(x)(display 5)(newline)(+ x 1)>

🡺

normal-eval[ (display 5)(newline)(+ 0 1) ]

normal-eval[ (display 5) ] 🡺

normal-eval[ display ] 🡺 <primitive-procedure display>

normal-eval[ 5 ] 🡺 5

* (to screen) 5

normal-eval[ newline ] 🡺 <primitive-procedure newline>

* (to screen) \n

normal-eval[ (+ 0 1) ] 🡺

normal-eval[ + ] 🡺 <primitive-procedure +>

normal-eval[ 0 ] 🡺 0

normal-eval[ 1 ] 🡺 1

* 1

🡺 1

normal-eval[ 2 ] 🡺 2

🡺 3

1.a.2. normal-eval:

5

1

5

3

Applicative-eval:

5

1

3

1.a.3. It depends. If both normal and applicative eval algorithms running on an expression terminate, they will evaluate to the same value, as they were planned to follow the same language rules, but they differ in the way of calculating it. ***If we DO consider infinite loops***, one might construct an expression that will result in an infinite loop in applicative-eval while terminating in normal-eval. This is a result of the lazy attribute of normal-eval – it won’t evaluate procedure arguments if it doesn’t have to.

An example for such infinite-tricky-loop: (define f (lambda(x)(f x)))(define g (lambda(x) 7)) (g(f 0))

Normal-eval will never evaluate (f 0) as the body of ‘g’ totally ignores it, while applicative-eval will and will therefore be stuck in an infinite loop, calling itself infinite times.

2.a.

1.

(lambda (f n)

(lambda (x)

(\* n (f x))))

The expression leaves are: \*, n, f, x

1. {x: T1} |- x: T1 (Variable axiom)
2. {f: T2} |- f: T2 (Variable axiom)
3. By application of the Application rule to (i) and (ii) with type substitutions **S1=T1, S=T3, T1=[T1 -> T3],**

{f: [T1 -> T3], x: T1} |- (f x): T3

1. {} |- \*: Number\*Number -> Number (Primitive procedure axiom)
2. {n: T4} |- n: T4 (Variable axiom)
3. By application of the Application rule to (iii), (iv) and (v),

{n: Number, f: [T1 -> Number]} |- (\* n (f x)): Number

1. By application of the Procedure rule to (vi), with type substitutions **S1=T1, S=Number**

{n: Number, x: T1, f: [T1 -> Number]} |- (lambda(x)(\* n (f x))): [T1 -> Number]

1. By application of the Procedure rule to (vii), with type substitutions **S1=[T1 -> Number], S2=Number, S=[T1 -> Number]**

{ } |- (lambda(f n)(lambda(x)(\* n (f x)))): **[T1 -> Number]\*Number -> [T1 -> Number]**

This proves the expression is well typed.

2.

((lambda (f n)

(lambda (x) (\* n (f x)))

) f1 3)

The expression leaves are: \*, n, f, x, 3

Deriving the lambdas just like the previous section.

1. {} |- 3: Number (Number axiom)
2. {f1: T5} |- f1: T5 (Variable axiom)
3. By application of the Application rule to (viii), (ix) and (x), with type substitutions **S1=T5, S2=Number, S=[T1 -> Number], T5=[T1 -> Number]**

{f1: [T1 -> Number]} |- ((lambda(f n)(lambda(x)(\* n (f x)))) f1 3): [T1 -> Number]

Since we assume f1 to be of type [T1 -> Number] in order to derive the type correctly, this expression is NOT well typed.

3.

(define dg

(lambda (number)

(if (< number 10)

1

(+ (dg (quotient number 10)) 1))

))

(dg ((lambda(dg2) dg2) 450))

The expression leaves are: dg, dg2, 450, <, number, 10, 1, quotient, +

1. {} |- +: Number\*..\*Number -> Number (primitive procedure axiom)
2. {} |- <: Number\*Number -> Boolean (primitive procedure axiom)
3. {} |- quotient: Number\*Number -> Number (primitive procedure axiom)
4. {} |- 450: Number (number axiom)
5. {} |- 1: Number (number axiom)
6. {} |- 10: Number (number axiom)
7. {dg2: T1} |- dg2: T1 (Variable axiom)
8. {number: T2} |- number: T2 (Variable axiom)
9. By application of the Application rule to (iii), (viii) and (vi), with type substitutions **S1=T2=Number, S2=Number, S=Number**

{number: Number} |- (quotient number 10): Number

1. {dg: [T3 -> T4]} |- dg: [T3 -> T4]
2. By application of the Application rule to (x) and (ix), with type substitutions **S1=T3=Number, S=T4**

{dg: [Number -> T4], number: Number} |- (dg (quotient number 10)): T4

1. By application of the Application rule to (i), (xi) and (v) with type substitutions **S1=T4=Number, S2=Number, S=Number**

{dg: [Number -> Number], number: Number} |- (+ (dg (quotient number 10)) 1): Number

1. By application of the Application rule to (ii), (viii) and (vi), with type substitutions **S1=Number, S2=Number, S=Boolean**

{dg: [Number -> Number], number: Number} |- (< number 10): Boolean

1. By application of the If rule to (xiii), (v) and (xii), with type substitutions **S1=Boolean, S2=Number, S3=Number**

{dg: [Number -> Number], number: Number} |- (if (< number 10) 1 (+ (dg (quotient number 10)) 1)): Number

1. By application of the Procedure rule to (viii) and (xiv), with type substitutions **S1=T2=Number, S=Number**

{} |- (lambda (number) (if (< number 10) 1 (+ (dg (quotient number 10)) 1))): Number

1. By application of the Procedure rule to (vii), with type substitutions **S1=T1,S=T1**

{dg2: T1} |- (lambda(dg2) dg2): [T1 -> T1]

1. By application of the Application rule to (xvi) and (iv), with type substitutions **S1=T1=Number, S=Number**

{} |- ((lambda(dg2) dg2) 450): Number

1. By application of the Application rule to (xv) and (xvii), with type substitutions **S1=Number, S=Number**

{} |- (dg ((lambda(dg2) dg2) 450)): Number

**The expression is well typed, and the type is Number.**

4.

(+ (lambda (x) 5) x)

The expression leaves are: +, x, 5

1. {} |- +: Number\*..\*Number -> Number (primitive procedure axiom)
2. {} |- 5: Number (number axiom)
3. By application of the Procedure rule, **S1=T1**, **S=Number**

{} |- (lambda(x) 5): [T1 -> Number]

1. ERROR! Wrong type given to +. Expected Number, got [T1 -> Number].

We assume x to be Number, so obviously this expression is not well typed.

2.b.

2.c.

(define number-proc

(lambda(par)

(if (number? par)

(+ par 1)

(+ (par 1) 1))))

(number-proc 10)

Expression leaves: +, number?, par, 1, 10, number-proc

i.{} |- number?: [T1 -> Boolean] (Primitive procedure axiom)

ii. {par: T2} |- par: T2

iii. {} |- 1: Number

iv.{} |- 10: Number

v.{} |- +: Number\*..\*Number -> Number (Primitive procedure axiom)

vi. By applying the Application rule on (v), (ii) and (iii) with type substitutes **S1=T2=Number, S2=Number, S=Number,**

{par: Number} |- (+ par 1): Number

vii. By applying the Application rule on (ii) and (iii) with type substitutes **S1=Number, T2=[Number -> T3], S=T3,**

{par: [Number -> T3]} |- (par 1): T3

viii. By applying the Application rule on (v), (vii) and (iii) with type substitutes **S1=T3=Number, S2=Number, S=Number,**

{par: [Number -> Number]} |- (+ (par 1) 1): Number

ix. By applying the Application rule on (i) and (ii) with type substitutes **S1=T2, S=Boolean,**

{} |- (number? Par): Boolean

x. By applying the If rule on (x), (vi) and (viii) with type substitutes **S1=Boolean, S2=Number, S3=Number,**

{par: [Number OR [Number -> Number]]} |- (if (number? par) (+ par 1)(+ (par 1) 1)): Number

xi.By applying the Procedure rule on (x) with type substitutes **S1=[Number OR [Number -> Number]], S=Number,**

{par: [Number OR [Number -> Number]]} |- (lambda(par) (if (number? par) (+ par 1)(+ (par 1) 1))): [[Number OR [Number -> Number]] -> Number]

xii.By applying the Application rule on (xi) and (iv) with type substitutes **S1=Number, S=Number,**

{} |- (number-proc 10): Number

3.a.

**1. (lambda(f)**

**(f ((lambda(g y) (+ 1 (g y))) + 10)))**

|  |  |
| --- | --- |
| ***Expression*** | ***Variable*** |
| (lambda(f)(f ((lambda(g y)(+ 1 (g y))) + 10))) |  |
| (f ((lambda(g y)(+ 1 (g y))) + 10)) |  |
| f |  |
| ((lambda(g y)(+ 1 (g y))) + 10) |  |
| (lambda(g y)(+ 1 (g y)) |  |
| + |  |
| 10 |  |
| (+ 1 (g y)) |  |
| 1 |  |
| (g y) |  |
| g |  |
| y |  |

|  |  |
| --- | --- |
| ***Expression*** | ***Equation*** |
| 10 |  |
| 1 |  |
| + |  |
| (lambda(f)(f ((lambda(g y)(+ 1 (g y))) + 10))) |  |
| (f ((lambda(g y)(+ 1 (g y))) + 10)) |  |
| ((lambda(g y)(+ 1 (g y))) + 10) |  |
| (lambda(g y)(+ 1 (g y)) |  |
| (+ 1 (g y)) |  |
| (g y) |  |

|  |  |
| --- | --- |
| ***Equation*** | ***Substitution*** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

1. Substitution is empty, add equation to the list.

|  |  |
| --- | --- |
| ***Equation*** | ***Substitution*** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

1. Substituting and adding to list.

|  |  |
| --- | --- |
| ***Equation*** | ***Substitution*** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

1. Adding .

|  |  |
| --- | --- |
| ***Equation*** | ***Substitution*** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

1. Substituting – Equation with same type constructor on both sides:

|  |  |
| --- | --- |
| ***Equation*** | ***Substitution*** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

1. Substituting and adding to the list.

|  |  |
| --- | --- |
| ***Equation*** | ***Substitution*** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

1. Adding .

|  |  |
| --- | --- |
| ***Equation*** | ***Substitution*** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

1. 🡪 .

|  |  |
| --- | --- |
| ***Equation*** | ***Substitution*** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

1. Replacing , ,,

|  |  |
| --- | --- |
| ***Equation*** | ***Substitution*** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
| ***Equation*** | ***Substitution*** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

1. Replacing

|  |  |
| --- | --- |
| ***Equation*** | ***Substitution*** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**Success! Type:**

3.a.

**2. (lambda(f)**

**(lambda(y)**

**(y (f f))))**

|  |  |
| --- | --- |
| ***Expression*** | ***Variable*** |
| (lambda(f)(lambda(y)(y (f f)))) |  |
| (lambda(y)(y (f f))) |  |
| (y (f f)) |  |
| y |  |
| (f f) |  |
| f |  |

|  |  |
| --- | --- |
| ***Expression*** | ***Equation*** |
| (lambda(f)(lambda(y)(y (f f)))) |  |
| (lambda(y)(y (f f))) |  |
| (y (f f)) |  |
| (f f) |  |
| ***Equation*** | ***Substitution*** |
|  |  |
|  |  |
|  |  |
|  |  |

1. Empty substitution. Move equation to the list.

|  |  |
| --- | --- |
| ***Equation*** | ***Substitution*** |
|  |  |
|  |  |
|  |  |

1. Substitute and add to the list.

|  |  |
| --- | --- |
| ***Equation*** | ***Substitution*** |
|  |  |
|  |  |

1. Substitute and add to the list.

|  |  |
| --- | --- |
| ***Equation*** | ***Substitution*** |
|  |  |
|  |  |
|  |  |

1. ERROR! Cannot substitute as it contains a circular reference. **FAIL.**

**Failed!** Circular substitution on .

3.a.

**3. (lambda(x)**

**(+ x (x 1)))**

|  |  |
| --- | --- |
| ***Expression*** | ***Variable*** |
| (lambda(x)(+ x (x 1))) |  |
| (+ x (x 1)) |  |
| + |  |
| x |  |
| (x 1) |  |
| 1 |  |

|  |  |
| --- | --- |
| ***Expression*** | ***Equation*** |
| (lambda(x)(+ x (x 1))) |  |
| (+ x (x 1)) |  |
| (x 1) |  |
| + |  |
| 1 |  |

|  |  |
| --- | --- |
| ***Equation*** | ***Substitution*** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

1. Empty substitution. Move to the list.

|  |  |
| --- | --- |
| ***Equation*** | ***Substitution*** |
|  |  |
|  |  |
|  |  |
|  |  |

1. Move to the list.

|  |  |
| --- | --- |
| ***Equation*** | ***Substitution*** |
|  |  |
|  |  |
|  |  |

1. Substitute and move to the list.

|  |  |
| --- | --- |
| ***Equation*** | ***Substitution*** |
|  |  |
|  |  |
|  |  |

1. Substituting equations with the same type constructor:

And substituting the constant

**ERROR!**  is impossible.

3.b.

Since **Let** is just syntactic sugar for constructing a lambda closure with as parameters, as the closure body, and applying it on the given values.

Therefore, combining the lambda rule and the application rule, we shall construct the **Let** type equation rule as follows:

**Let:** For (let (( ) construct the equation .

4.a.

Concrete syntax:

<case> -> ‘case’ <exp> <exp> ‘(‘ <case-clause>\* ’)’ <else-clause>

<case-clause> -> ‘(‘ <exp>+ ’) <exp>+’

Abstract syntax:

<case>:

Components: Key: <exp>

Predicate: <exp>

Clause: <case-clause>. Amount >= 0. Ordered.

Else-Clause: <else-clause>

<case-clause>:

Components: Datum: <exp>. Amount >= 1. Ordered.

Action: <exp>. Amount >= 1. Ordered.

<else-clause>:

Components: Action: <exp>. Amount >= 1. Ordered.

4.b.

**Pseudo code:**

**Explanation:**

Similar to the cond applicative-eval algorithm, but extended – we go over all the datums, looking

for one that the application of *pred* on it and *key* will result in #t. Once we find such datum, we stop the search and evaluate the matching expressions, returning the value of the last one.

If no such datum was found, we evaluate the else clause expressions and return the value of the last one.

5.a.

(let ((gcd (lambda(a b)

(if (= b 0)

a

(gcd b (modulo a b) )

)

))

)

(gcd (+ (+ 3 3) (+ 3 3)) 8))

**Converting to its real form (+ making it actually readable):**

(

(lambda(gcd1) (gcd1 (+ (+ 3 3) (+ 3 3)) 8)))

(lambda(a1 b1)

(if (= b1 0)

a1

(gcd2 b1 (modulo a1 b1) )

)

)

)

* gcd1 is limited to the scope of the lambda we are executing, so in that scope it is bound.
* a1, b1 are limited to the scope of the lambda we are passing as an argument.
* All appearances of a1 and b1 in the argument lambda are bound to it.
* In the second lambda, gcd2 is a free variable – it is not declared anywhere in this code!

5.b.

**Obviously,** this code will fail when trying to apply gcd2 as it is **not bound** in this expression; So, unless we are pre-defined gcd2, this code will try to apply a non-existent procedure.

What the intended behavior was trying to achieve is recursion; Allowing a local procedure to call itself. Without the global environment this fails as a closure ‘does not know of its own existence’ (which is what a local procedure really is – just a closure passed around).

**5.c. & 5.d.**

(let ((gcd (lambda(a b gcdnew)

(if (= b 0)

a

(gcdnew b (modulo a b) gcdnew)))))

(gcd (+ (+ 3 3) (+ 3 3)) 8 gcd))

**Converting to its real form (+ making it actually readable):**

(

(lambda(gcd) (gcd (+ (+ 3 3) (+ 3 3)) 8 gcd))

(lambda(a b gcdnew)

(if (= b 0)

a

(gcdnew b (modulo a b) gcdnew)

)

)

)

**(5.c) applicative-eval:**

a-e[((lambda(gcd) (gcd (+ (+ 3 3) (+ 3 3)) 8 gcd))

(lambda(a b gcdnew)

(if (= b 0)

a

(gcdnew b (modulo a b) gcdnew))))] 🡺

a-e[(lambda(gcd) (gcd (+ (+ 3 3) (+ 3 3)) 8 gcd))] 🡺 <Closure(gcd)(gcd (+ (+ 3 3) (+ 3 3)) 8 gcd)>

a-e[(lambda(a b gcdnew) (if (= b 0) a (gcdnew b (modulo a b) gcdnew)))] 🡺 <Closure(a,b,gcdnew)(if (= b 0) a (gcdnew b (modulo a b) gcdnew))>

a-e[(<Closure(a,b,gcdnew)(if (= b 0) a (gcdnew b (modulo a b) gcdnew))>

(+ (+ 3 3) (+ 3 3))

8

<Closure(a,b,gcdnew)(if (= b 0) a (gcdnew b (modulo a b) gcdnew))>

)] 🡺

a-e[(if (= 8 0)

(+ (+ 3 3) (+ 3 3))

(

<Closure(a2,b2,gcdnew2)(if (= b2 0) a2 (gcdnew b2 (modulo a2 b2) gcdnew2))>

8

(modulo (+ (+ 3 3) (+ 3 3)) 8)

<Closure(a2,b2,gcdnew2)(if (= b2 0) a2 (gcdnew2 b2 (modulo2 a2 b2) gcdnew2))>

)

)] 🡺

a-e[if] 🡺 <primitive-procedure if>

a-e[(= 8 0)] 🡺 (primitive procedure =) returns #f

a-e[(

<Closure(a2,b2,gcdnew2)(if (= b2 0) a2 (gcdnew b2 (modulo a2 b2) gcdnew2))>

8

(modulo (+ (+ 3 3) (+ 3 3)) 8)

<Closure(a2,b2,gcdnew2)(if (= b2 0) a2 (gcdnew2 b2 (modulo2 a2 b2) gcdnew2))>

) 🡺

a-e[8] 🡺 8

a-e[(modulo (+ (+ 3 3) (+ 3 3)) 8)] 🡺 (primitive procedures modulo and +) returns 4

a-e[(if (= 4 0)

8

(

<Closure(a3,b3,gcdnew3)(if (= b3 0) a3 (gcdnew3 b3 (modulo3 a3 b3) gcdnew3))>

4

(modulo 8 4)

<Closure(a3,b3,gcdnew3)(if (= b3 0) a3 (gcdnew3 b3 (modulo3 a3 b3) gcdnew3))>

)

)] 🡺

a-e[if] 🡺 <primitive-procedure if>

a-e[(= 4 0)] 🡺 (primitive procedure =) returns #f

a-e[(

<Closure(a3,b3,gcdnew3)(if (= b3 0) a3 (gcdnew3 b3 (modulo3 a3 b3) gcdnew3))>

4

(modulo 8 4)

<Closure(a3,b3,gcdnew3)(if (= b3 0) a3 (gcdnew3 b3 (modulo3 a3 b3) gcdnew3))>

)] 🡺

a-e[4] 🡺 4

a-e[(modulo 8 4)] 🡺 (primitive procedure modulo) **returns 0**

a-e[(if (= 0 0)

4

(

<Closure(a4,b4,gcdnew4)(if (= b4 0) a4 (gcdnew4 b4 (modulo4 a4 b4) gcdnew4))>

0

(modulo3 4 0)

<Closure(a4,b4,gcdnew4)(if (= b4 0) a4 (gcdnew4 b4 (modulo4 a4 b4) gcdnew4))>

)

]) 🡺

a-e[if] 🡺 <primitive-procedure if>

a-e[(= 0 0)] 🡺 (primitive procedure =) **returns #t**

a-e[4] 🡺 4

🡺 **RETURN: 4**

**(5.d) normal-eval:**

n-e[((lambda(gcd) (gcd (+ (+ 3 3) (+ 3 3)) 8 gcd))

(lambda(a b gcdnew)

(if (= b 0)

a

(gcdnew b (modulo a b) gcdnew))))] 🡺

n-e[(lambda(gcd) (gcd (+ (+ 3 3) (+ 3 3)) 8 gcd))] 🡺 <Closure(gcd)(gcd (+ (+ 3 3) (+ 3 3)) 8 gcd)>

n-e[( (lambda(a b gcdnew)

(if (= b 0)

a

(gcdnew b (modulo a b) gcdnew))

)

(+ (+ 3 3) (+ 3 3))

8

(lambda(a b gcdnew)

(if (= b 0)

a

(gcdnew b (modulo a b) gcdnew))

)

)] 🡺

n-e[(lambda(a b gcdnew)

(if (= b 0)

a

(gcdnew b (modulo a b) gcdnew))

)] 🡺 <Closure(a,b,gcdnew)(if (= b 0) a (gcdnew b (modulo a b) gcdnew))>

n-e[(if (= 8 0)

(+ (+ 3 3) (+ 3 3))

(

(lambda(a2 b2 gcdnew2)

(if (= b2 0)

a2

(gcdnew2 b2 (modulo a2 b2) gcdnew2))

)

8

(modulo (+ (+ 3 3) (+ 3 3)) 8)

(lambda(a2 b2 gcdnew2)

(if (= b2 0)

a2

(gcdnew2 b2 (modulo a2 b2) gcdnew2))

)

)

)] 🡺

n-e[if] 🡺 <primitive-procedure if>

n-e[(= 8 0)] 🡺 (primitive procedure =) returns #f

n-e[(

(lambda(a2 b2 gcdnew2)

(if (= b2 0)

a2

(gcdnew2 b2 (modulo a2 b2) gcdnew2))

)

8

(modulo (+ (+ 3 3) (+ 3 3)) 8)

(lambda(a2 b2 gcdnew2)

(if (= b2 0)

a2

(gcdnew2 b2 (modulo a2 b2) gcdnew2))

)

)] 🡺

n-e[(lambda(a2 b2 gcdnew2)

(if (= b2 0)

a2

(gcdnew2 b2 (modulo a2 b2) gcdnew2))

)] 🡺 <Closure(a2, b2, gcdnew2)(if (= b2 0) a2 (gcdnew2 b2 (modulo a2 b2 gcdnew2))>

n-e[(if (= (modulo (+ (+ 3 3) (+ 3 3)) 8) 0)

8

(

(lambda(a3 b3 gcdnew3)

(if (= b3 0)

a3

(gcdnew3 b3 (modulo a3 b3) gcdnew3))

)

(modulo (+ (+ 3 3) (+ 3 3)) 8)

(modulo 8 (modulo (+ (+ 3 3) (+ 3 3)) 8)

(lambda(a3 b3 gcdnew3)

(if (= b3 0)

a3

(gcdnew3 b3 (modulo a3 b3) gcdnew3))

)

)

)] 🡺 <primitive-procedure if>

n-e[(= (modulo (+ (+ 3 3) (+ 3 3)) 8) 0)] 🡺 (primitive procedures = and modulo, saving some steps) returns #f

n-e[(

(lambda(a3 b3 gcdnew3)

(if (= b3 0)

a3

(gcdnew3 b3 (modulo a3 b3) gcdnew3)

)

)

(modulo (+ (+ 3 3) (+ 3 3)) 8)

(modulo 8 (modulo (+ (+ 3 3) (+ 3 3)) 8)

(lambda(a3 b3 gcdnew3)

(if (= b3 0)

a3

(gcdnew3 b3 (modulo a3 b3) gcdnew3))

)

)

)] 🡺

n-e[(lambda(a3 b3 gcdnew3)

(if (= b3 0)

a3

(gcdnew3 b3 (modulo a3 b3) gcdnew3)

)

)] 🡺 <Closure(a3,b3,gcdnew3)(if (= b3 0) a3 (gcdnew3 b3 (modulo a3 b3) gcdnew3)>

n-e[(if (=(modulo 8 (modulo (+ (+ 3 3) (+ 3 3)) 8) 0)

(modulo (+ (+ 3 3) (+ 3 3)) 8)

(

(lambda(a3 b3 gcdnew3)

(if (= b3 0)

a3

(gcdnew3 b3 (modulo a3 b3) gcdnew3))

)

)

(modulo 8 (modulo (+ (+ 3 3) (+ 3 3)) 8)

(modulo (modulo (+ (+ 3 3) (+ 3 3)) 8) (modulo 8 (modulo (+ (+ 3 3) (+ 3 3)) 8))

(lambda(a3 b3 gcdnew3)

(if (= b3 0)

a3

(gcdnew3 b3 (modulo a3 b3) gcdnew3))

)

)

)] 🡺

n-e[(=(modulo 8 (modulo (+ (+ 3 3) (+ 3 3)) 8) 0)] 🡺 (= and modulo primitive procedures) **returns #t**

n-e[(modulo (+ (+ 3 3) (+ 3 3)) 8)] 🡺 (+ and modulo primitive procedures), **returns 4**

**… 🡺 4**

**RETURN: 4**

5.e.

(let ((choose (lambda(n k choose)

(cond

((= k 0) 1)

((= k n) 1)

(else (+

(choose (- n 1) k choose)

(choose (- n 1) (- k 1) choose)

)

)

)

)))

(choose 6 4 choose)

)