Decoding The Expression

In [39]:

```
def get_number(f)-> int:
    INC = lambda x: x+1
    return f(INC)(0)
```

1)

In [40]:

```
FALSE = lambda a: lambda b: b
TRUE  = lambda a: lambda b: a

AND   = lambda a: lambda b: a(b)(FALSE)
OR    = lambda a: lambda b: a(TRUE)(b)
NOT   = lambda a: a(FALSE)(TRUE)

XOR   = lambda a: lambda b: a(b(FALSE)(TRUE))(b(TRUE)(FALSE))
```

In [41]:

```
Alpha = TRUE
Beta  = TRUE
Gama  = FALSE

# section I)
Ans = OR(OR(AND(Alpha)(Beta))(AND(Alpha)(Gama)))(AND(Beta)(Gama))
print("1) I)")
if Ans == TRUE:
    print("TRUE")
else:
    print("FALSE")

# section II)
Ans = XOR(OR(Alpha)(Beta))(AND(NOT(Alpha))(Gama))
print("1) II)")
if Ans == TRUE:
    print("TRUE")
else:
    print("FALSE")
```

```
1) I)
TRUE
1) II)
TRUE
```

2)

In [42]:

```
I   = lambda a: a
is_zero    = lambda a: a(lambda _: FALSE)(TRUE)
inc = lambda a: lambda b: lambda c: b(a(b)(c))
dec = lambda a: lambda b: lambda c: a(lambda x: lambda y: y(x(b)))(lambda _: c)(I)
sub = lambda a: lambda b: b(dec)(a)

less_than    = lambda a: lambda b: is_zero(sub(inc(a))(b))
greater_than = lambda a: lambda b: is_zero(sub(inc(b))(a))
```

In [43]:

```
zero  = FALSE
one   = lambda a: lambda b: a(b)
two   = lambda a: lambda b: a(a(b))
three = lambda a: lambda b: a(a(a(b)))
four  = lambda a: lambda b: a(a(a(a(b))))
five  = lambda a: lambda b: a(a(a(a(a(b)))))
six   = lambda a: lambda b: a(a(a(a(a(a(b))))))
seven = lambda a: lambda b: a(a(a(a(a(a(a(b)))))))
eight = lambda a: lambda b: a(a(a(a(a(a(a(a(b))))))))
nine  = lambda a: lambda b: a(a(a(a(a(a(a(a(a(b)))))))))
ten   = lambda a: lambda b: a(a(a(a(a(a(a(a(a(a(b))))))))))
```

In [44]:

```
# Example I)
Ans = greater_than(two)(three)
if Ans == TRUE:
    print("TRUE")
else:
    print("FALSE")

# Example II)
Ans = less_than(six)(seven)
if Ans == TRUE:
    print("TRUE")
else:
    print("FALSE")
```

```
FALSE
TRUE
```

3)

In [45]:

```
# pair
cons = lambda a: lambda b: lambda c: c(a)(b)
car  = lambda a: a(TRUE)
cdr  = lambda a: a(FALSE)

XNOR = lambda a: lambda b: NOT(XOR(a)(b))

add = lambda a: lambda b: a(inc)(b)
diff = lambda a: lambda b: add(sub(a)(b))(sub(b)(a))
less_than_or_equal = lambda a: lambda b: is_zero(sub(a)(b))

sadd   = lambda a: lambda b: (
    XNOR(car(a))(car(b))
    (cons(car(a))(add(cdr(a))(cdr(b))))  # same sign
    (
        cons    # opposite sign
        (XOR(car(a))(less_than_or_equal(cdr(a))(cdr(b))))  # calculate sign
        (diff(cdr(a))(cdr(b)))   # calculate value
    )
)
ssub = lambda a: lambda b: sadd(a)(cons(NOT(car(b)))(cdr(b)))
```

In [46]:

```
NUM1 = cons(TRUE)(ten)
NUM2 = cons(TRUE)(five)

signed_ans = ssub(NUM1)(NUM2)

sign = car(signed_ans)
value = cdr(signed_ans)

number = get_number(value)
if sign == TRUE:
    print(number)
elif sign == FALSE:
    print(-1*number)
else:
    print("Error!")
```

5

In [47]:

```
NUM1 = cons(TRUE)(four)
NUM2 = cons(TRUE)(five)

signed_ans = sadd(NUM1)(NUM2)

sign = car(signed_ans)
value = cdr(signed_ans)

number = get_number(value)
if sign == TRUE:
    print(number)
elif sign == FALSE:
    print(-1*number)
else:
    print("Error!")
```

```
9
```

4)

In [48]:

```
# True logic
T  = lambda a: lambda b: a
# False logic
F = lambda a: lambda b: b

# Identity function
I = lambda a: a

# this boolean AND operator takes two boolean variables and if the first one were TRUE ret
urns the second one else if returns FALSE
AND = lambda a: lambda b: a(b)(F)

# this boolean OR operator takes two boolean variables and if the first one were TRUE retu
rns the first one(or TRUE) else if returns the second one
OR  = lambda a: lambda b: a(T)(b)

# this boolean NOT operator takes one boolean variable and returns TRUE if the variable we
re FALSE and vice versa
NOT = lambda a: a(F)(T)

# this lambda expression is recursion idea that takes a function f and returns the same as
to entry
# I mean fixed point of a function is an input that is unchanged by that function
# some example to clarity :
#
#     Y g = g Y g = g(g(Yg)) = g(  ...  g(Yg)   ...)
#

Y = lambda f: ( (lambda x: f(lambda y: x(x)(y)) ) ( lambda x: f(lambda y: x(x)(y)) ) )
# is zero operator takes a variable( or name ) and returns the entry
# that the entry had been zero the second one selected else returns a constant
# function that takes every thing and returns False(lambda _: F)
is_zero    = lambda a: a(lambda _: F)(T)

# notes: Less than:
#    LT := λab. NOT (LEQ b a)
#    Less than or equal to:
#    LEQ := λmn. ISZERO (SUB m n)
less_than = lambda a: lambda b: is_zero(sub(inc(a))(b))
# The successor operator (given a natural number n, calculate n+1):
#
#    SUCC := λnfx. f (n f x)
#
# the b and c are the aoxial variables that acts the zero rule in brackets

inc = lambda a: lambda b: lambda c: b(a(b)(c))
add = lambda a: lambda b: a(inc)(b)

# The predecessor operator (for all n > 0, calculate n-1; for zero, return zero):
#
#    PRED       :=      λnfx. n (λgh. h (g f)) (λu. x) (λu. u)

dec = lambda a: lambda b: lambda c: a(lambda x: lambda y: y(x(b)))(lambda _: c)(I)
sub = lambda a: lambda b: b(dec)(a)
```

```
mul = lambda a: lambda b: lambda c: a(b(c))
div = Y(
        lambda f: lambda a: lambda b: less_than(a)(b)
        (lambda _: zero)
        (lambda _: inc(f(sub(a)(b))(b)))
        (zero)
        )

def get_number(f)-> int:
    INCR = lambda x: x+1
    return f(INCR)(0)
```

In [49]:

```
get_number(div(ten)(five))
```

Out[49]:

```
2
```

5)

In [50]:

```
mul = lambda a: lambda b: lambda c: a(b(c))

fac = Y(
    lambda f: lambda n: is_zero(n)
    (lambda _: one)
    (lambda _: mul(n)(f(dec(n))))
    (zero)
)
```

In [51]:

```
get_number(fac(six))
```

Out[51]:

```
720
```

6)

In [52]:

```
# the first number is enumerator and the second one is denumerator
rational_number = lambda a: lambda b: cons(a)(b)

rational_add = lambda a: lambda b: cons(add(mul(car(a))(cdr(b)))(mul(cdr(a))(car(b))))(mul
(cdr(a))(cdr(b)))
rational_mul = lambda a: lambda b: cons(mul(car(a))(car(b)))(mul(cdr(a))(cdr(b)))
```

In [53]:

```
A = rational_number(three)(two)
B = rational_number(five)(two)

Ans = rational_add(A)(B)
enum  = car(Ans)
denum = cdr(Ans)
print(get_number(enum)/get_number(denum))
```

```
4.0
```

7)

In [54]:

```
complex_num = lambda a: lambda b: cons(a)(b)

# sign multiplication
smul = lambda a: lambda b: cons(XOR(car(a))(car(b)))(mul(cdr(a))(cdr(b)))

# complex addition
complex_add = lambda a: lambda b: cons(sadd(car(a))(car(b)))(sadd(cdr(a))(cdr(b)))
# complex division
complex_div = lambda a: lambda b: cons(
    rational_number(sadd(smul(car(a))(car(b)))(smul(cdr(a))(cdr(b))))(sadd(smul(car(b))(ca
r(b))) (smul(cdr(b))(cdr(b)))))(
    rational_number(ssub(smul(cdr(a))(car(b)))(smul(car(a))(cdr(b))))(sadd(smul(car(b))(ca
r(b))) (smul(cdr(b))(cdr(b))))
    )
```

In [55]:

```
num1 = cons(TRUE)(one)
num2 = cons(TRUE)(one)
num3 = cons(TRUE)(two)
num4 = cons(TRUE)(two)

NUM1 = complex_num(num1)(num2)
NUM2 = complex_num(num3)(num4)

Ans = complex_add(NUM1)(NUM2)
num_x = car(Ans)
num_y = cdr(Ans)

ans = complex(get_number(cdr(num_x)),(get_number(cdr(num_y))))
print(ans)
```

```
(3+3j)
```

In [56]:

```
num1 = cons(TRUE)(one)
num2 = cons(TRUE)(one)
num3 = cons(TRUE)(three)
num4 = cons(TRUE)(one)

NUM1 = complex_num(num1)(num2)
NUM2 = complex_num(num3)(num4)

Ans = complex_div(NUM1)(NUM2)
num_x = car(Ans)
num_y = cdr(Ans)

r_num_x_enum = car(num_x)
r_num_x_denum = cdr(num_x)

r_num_y_enum = car(num_y)
r_num_y_denum = cdr(num_y)

# A + Bj
A = get_number(cdr(r_num_x_enum))/get_number(cdr(r_num_x_denum))
if car(r_num_x_enum) == TRUE:
    A = +1*A
elif car(r_num_x_enum) == FALSE:
    A = -1*A

B = get_number(cdr(r_num_y_enum))/get_number(cdr(r_num_y_denum))
if car(r_num_y_enum) == TRUE:
    B = +1*B
elif car(r_num_y_enum) == FALSE:
    B = -1*B

ans = complex(A,B)
print(ans)
```

```
(-0.4-0.2j)
```

8)

In [57]:

```
# in the proceeding cells this question be answered
```

9)

In [84]:

```
"""A = lambda x: lambda y: y
B = lambda x: lambda y: lambda z: (x)(z)((y)(z))
ID = lambda x: x

SUCC = lambda a: B(a)(a)(B(ID)(A)(ID))
get_number(SUCC(zero))"""
```

Out[84]:

```
'A = lambda x: lambda y: y\nB = lambda x: lambda y: lambda z: (x)(z)((y)(z))
\nID = lambda x: x\n\nSUCC = lambda a: B(a)(a)(B(ID)(A)(ID))\nget_number(SUCC
(zero))'
```

10)

In [59]:

```
quine = lambda z: ((lambda x: lambda z: (x)(x)) (lambda x: lambda z: (x)(x)))
```

In [ ]:

```

```