Table of Contents

License

Author: Kasra Eskandari **Student ID:**955361005

Basic Concepts

these are the concepts that are defined and described in *lambda.pdf*

identity

In [1]:
```
I=lambda x:x
```

Arithmatics
numbers

In [2]:
```
N0=lambda s: lambda z : z
N1=lambda s: lambda z : s(z)
N2=lambda s: lambda z : s(s(z))
N3=lambda s: lambda z : s(s(s(z)))
N4=lambda s: lambda z : s(s(s(s(z))))
N5=lambda s: lambda z : s(s(s(s(s(z)))))
```

In [3]:
```
# this function will help us to convert Church encoded numbers into regular numbers
c2n=lambda c:c(lambda x:x+1)(0)
print(f"to make sure the number functions are working fine:\n\tN3={c2n(N3)}")
```

```
to make sure the number functions are working fine:
	N3=3
```

Successor function and Addition

In [4]:
```
S=lambda w:lambda y:lambda x: y(w(y)(x))
```

In [5]:
```
print(f"successor of 2 is : {c2n(S(N2))}")
print(f"2+3={c2n(N2(S)(N3))}")
```

```
successor of 2 is : 3
2+3=5
```

Multiplication

In [6]:
```
M = lambda x:lambda y:lambda z:x(y(z))
```

In [7]:
```
N6=M(N2)(N3)
N20=M(N4)(N5)
print(f"2*3={c2n(N6)}")
print(f"4*5={c2n(N20)}")
```

```
2*3=6
4*5=20
```

Logics
True and False

In [8]:
```
T=lambda x:lambda y:x
F=lambda x:lambda y:y
```

```
In [9]: # this function will help us to convert Church encoded True and False into regular values
c2l=lambda c:c(True)(False)
print(c2l(T))
print(c2l(F))
```

```
True
False
```

Logical Operations

```
In [10]: AND=lambda x:lambda y: x(y)(F)
OR =lambda x:lambda y: x(T)(y)
NOT=lambda x: x(F)(T)
```

```
In [11]: # test
print(f"T∧F={c2l(AND(T)(F))}")
print(f" ¬T={c2l(NOT(T))}")
print(c2l(AND(F)(F)))
print(c2l(AND(F)(T)))
print(c2l(AND(T)(F)))
print(c2l(AND(T)(T)))

print(c2l(OR(F)(F)))
print(c2l(OR(F)(T)))
print(c2l(OR(T)(F)))
print(c2l(OR(T)(T)))

print(c2l(NOT(T)))
print(c2l(NOT(F)))
```

```
T∧F=False
 ¬T=False
False
False
False
True
False
True
True
True
False
True
```

A Conditional Test

The following finction `Z` checks if the entry number is zero or not

```
In [12]: Z = lambda x: x(F)(NOT)(F)
```

In [13]:
```python
print(c2l(Z(N0)))
print(c2l(Z(N2)))
```

```
True
False
```

The predecessor function

To define predecessor function, pair of numbers will be needed.\ Lets define `pair(a,b)`

In [14]:
```python
PAIR = lambda a:lambda b: lambda z: z(a)(b)
p01=PAIR(N0)(N1)
print(c2n(p01(T)))
print(c2n(p01(F)))
```

```
0
1
```

In [15]:
```python
# this function will help us to convert Church encoded pairs into r
egular tuples
c2p=lambda x:(c2n(x(T)),c2n(x(F)))
print(c2p(p01))
```

```
(0, 1)
```

In [16]:
```python
# ϕ is PHI
PHI=lambda p: lambda z: z(S(p(T)))(p(T))
p00=PAIR(N0)(N0)
P=lambda n: n(PHI)(p00)(F)
```

In [17]:
```python
print(f"the predecessor of 2 is 1:\t{c2n(P(N2))}")
print(f"predecessor of zero is zero:\t{c2n(P(N0))}")
print(f"20-5={c2n(N5(P)(N20))}")
```

```
the predecessor of 2 is 1:      1
predecessor of zero is zero:    0
20-5=15
```

Equality and inequalities

greater than or equal

In [18]:
```python
G = lambda x: lambda y: Z(x(P)(y))
```

In [19]:
```python
print(c2l(G(N5)(N3)))
print(c2l(G(N3)(N5)))
```

```
True
False
```

Equal

If x ≥ y and y ≥ x, then x = y.

In [20]:
```
E = lambda x:lambda y: AND(G(x)(y))(G(y)(x))
```

In [21]:
```
print(c2l(E(N1)(N2)))
print(c2l(E(N1)(N1)))
print(c2l(E(N2)(N2)))
```

```
False
True
True
```

Question1

Rewrite these **Boolean** expressions as Lambda expressions

- α.β + α.γ + β.γ
- xor (or α β) (and not α γ)

part A

In [22]:
```
Q1a=lambda a:lambda b: lambda c: ((a(b)(F))(T)(a(c)(F)))(T)(b(c)(F
))
print(c2l(Q1a(T)(T)(T)))
```

```
True
```

In [23]:
```
# lets test it
Q1a_= lambda a,b,c: (a and b) or (a and c) or (b and c)
testCases=[list(map(lambda x:{'0':F,'1':T}[x],tuple(f"{i:03b}"))) f
or i in range(8)]
for a,b,c in testCases:
    assert Q1a_(c2l(a),c2l(b),c2l(c)) == c2l(Q1a(a)(b)(c)) , (c2l(a
),c2l(b),c2l(c))
    print(c2l(a),c2l(b),c2l(c)," -- was correct")
```

```
False False False  -- was correct
False False True  -- was correct
False True False  -- was correct
False True True  -- was correct
True False False  -- was correct
True False True  -- was correct
True True False  -- was correct
True True True  -- was correct
```

part B

In [24]:
```python
XOR = lambda a:lambda b: a(NOT(b))(b)
print(c2l(XOR(F)(F)))
print(c2l(XOR(F)(T)))
print(c2l(XOR(T)(F)))
print(c2l(XOR(T)(T)))
```

```
False
True
True
False
```

In [25]:
```python
Q1b=lambda a:lambda b:lambda c: XOR(OR(a)(b))(AND(NOT(a))(c))
```

In [26]:
```python
Q1b_=lambda a,b,c: (a or b) ^ (not a and c)
for a,b,c in testCases:
    assert Q1b_(c2l(a),c2l(b),c2l(c)) == c2l(Q1b(a)(b)(c)) , (c2l(a
),c2l(b),c2l(c))
    print(c2l(a),c2l(b),c2l(c)," -- was correct")
```

```
False False False  -- was correct
False False True  -- was correct
False True False  -- was correct
False True True  -- was correct
True False False  -- was correct
True False True  -- was correct
True True False  -- was correct
True True True  -- was correct
```

Question2

Define > and < for two [Church encoded] numerical arguments

part A

according to `a>b ≡ not(b≥a)` the definition for `>` is:

In [27]:
```python
Bigger = lambda x: lambda y: NOT(G(y)(x))
```

In [28]:
```python
print(f"2>2 is {c2l(Bigger(N2)(N2))}")
print(f"5>1 is {c2l(Bigger(N5)(N1))}")
print(f"1>5 is {c2l(Bigger(N1)(N5))}")
```

```
2>2 is False
5>1 is True
1>5 is False
```

part B

In the same way, Initialy the `≤` (less than or equal) should be defined:

In [29]:
```python
L = lambda x:lambda y: Z(y(P)(x))
```

according to `a<b ≡ not(b⩽a)` ,the definition for `<` is:

In [30]:
```python
Smaller = lambda x:lambda y: NOT(L(y)(x))
```

In [31]:
```python
print(f"2<2 is {c2l(Smaller(N2)(N2))}")
print(f"5<1 is {c2l(Smaller(N5)(N1))}")
print(f"1<5 is {c2l(Smaller(N1)(N5))}")
```

```
2<2 is False
5<1 is False
1<5 is True
```

Question3

Define positive and negative **integers** using pairs of **natural** numbers

- Define addition and subtraction

Definition of negetive and posetive integers

In order to define signed numbers, lets consider each number as pair of numbers, which one them is zero always.

- A natural number is converted to a signed number by `CONV` function
- Negation is performed by swapping the values using `NEG` function

In [32]:
```python
CONV=lambda x:PAIR(x)(N0)
NEG =lambda x:PAIR(x(F))(x(T))
```

In [33]:
```python
# for instance
plus3 =CONV(N3)
minus3=NEG(plus3)
print(c2p(plus3))
print(c2p(minus3))
```

```
(3, 0)
(0, 3)
```

all signed numbers are natural if and only if one of the pair is zero.The `OneZero` function achieves this condition. to understand how this function works here is the psuedocode:

```
def OneZero(x:pair):
  if (not Z(xT)) and (not Z(xF)):
    return OneZero(pair(P(xT))(P(xF)))
  else:
    return x
```

it is as same as:

```
def OneZero(x:pair):
  if not (Z(xT) or Z(xF)): # ¬∨(xT)(xF)
    return OneZero(pair(P(xT))(P(xF)))
  else:
    return x
```

`IF` condition in λ-calculus can be defined as:

> it means `if x then a else b`

In [34]:
```
IF =lambda x:lambda a:lambda b: (x(a)(b))()
OneZero=lambda x: IF( OR (Z(x(T))) (Z(x(F))) ) (lambda :x)  (lambda
:OneZero(PAIR(P(x(T)))(P(x(F)))))
```

In [35]:
```
p53=PAIR(N5)(N3)
print(c2p(OneZero(p53)))
p35=PAIR(N3)(N5)
print(c2p(OneZero(p35)))
```

```
(2, 0)
(0, 2)
```

Addition and subtraction

In [36]:
```
ADD = lambda x:lambda y:OneZero(PAIR(x(T)(S)(y(T)))(x(F)(S)(y(F))))
SUB = lambda x:lambda y:OneZero(PAIR(x(T)(P)(y(T)))(x(F)(P)(y(F))))
```

In [37]:
```
p3=CONV(N3)
m5=NEG(CONV(N5))
print(c2p(ADD(m5)(p3)))
```

```
(0, 2)
```

Question4

As we know:

```
n/m = if n⩾m then 1+(n-m)/m else 0
```

using λ-calculus notations divide function is:

In [38]:
```python
DIV=lambda n:lambda m: IF(G(n)(m))(lambda:S(DIV(m(P)(n))(m)))(lambda: N0)
```

In [39]:
```python
N9=N4(S)(N5)
N10=S(N9)
print(c2n(DIV(N9)(N3)))
print(c2n(DIV(N10)(N2)))
```

```
3
5
```

Question5

recursive form of the **factorial function** is:

```
def fact(n):
  if n==0:
    return 1
  else:
    return n*fact(n-1)
```

using the λ-calculus notations the factorial function is:

In [40]:
```python
FACT = lambda n:IF(Z(n))(lambda: N1)(lambda:M(n)(FACT(P(n))))
```

In [41]:
```python
print(c2n(FACT(N4)))
```

```
24
```

In [ ]: