

Programa 4.2 interprete de Lenguaje de Programación Esencial (LPE)

Teoría de la computación

Royce Richmond Ramirez Morales
rramirezm2021@cic.ipn.mx

Objetivo

Implementar un interprete de las operaciones suma, multiplicación, exponente, predecesor, monus, igualdad, cociente, impar, fibonacci, factorial, división y función de Ackerman; empleando el Lenguaje de Programación Esencial (LPE). Se empleará Python para programar el interprete y sus funciones.

Solución:

Primero se definen las funciones elementales del lenguaje de programación esencial, que son el incremento y decremento (el decremento solo funciona hasta el cero, ya que en el lenguaje de programación esencial no existen los números negativos), en el bloque de código 1 se encuentran estas funciones y en la Imagen 1 la muestra de ejecución.

```
1 def incr (a):  
2     return (a+1)  
3 def decr (a):  
4     while (a!=0):  
5         return (a-1)  
6     return (a)
```

Bloque de código 1: Definición del incremento y decremento en Python.

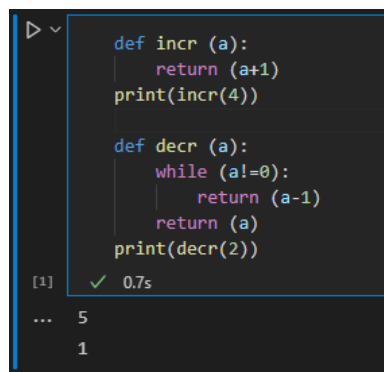


Imagen 1: Ejecución de la función incremento y decremento

Posteriormente se define la función limpiar que restablece una variable a cero, asignación que asigna a una variable el valor de otra, la función cero que retorna un cero y la función sucesor que muestra el sucesor de un numero, en el bloque de código 2 se encuentran estas funciones y en la Imagen 2 se puede ver la muestra de ejecución de estas operaciones.

```
1 def clr (a):
2     while (a!=0):
3         a=decr(a)
4     return (a)
5 def assi (x1,x2):
6     aux=1
7     aux=clr(aux)
8     x1=clr(x1)
9     while (x2!=0):
10        aux=incr(aux)
11        x2=decr(x2)
12    while (aux!=0):
13        x1=incr(x1)
14        x2=incr(x2)
15        aux=decr(aux)
16    return(x1)
17 def cero():
18     z=2
19     return (clr(z))
20 def suce(a):
21     aux=3
22     aux=incr(assi(aux,a))
23     return aux
```

Bloque de código 2: Definición de las funciones limpiar, asignación, cero y sucesor en Python.

```
def clr (a):
    while (a!=0):
        a=decr(a)
    return (a)

def assi (x1,x2):
    aux=1
    aux=clr(aux)
    x1=clr(x1)
    while (x2!=0):
        aux=incr(aux)
        x2=decr(x2)
    while (aux!=0):
        x1=incr(x1)
        x2=incr(x2)
        aux=decr(aux)
    return(x1)

def cero():
    z=2
    return (clr (z))

def suce(a):
    aux=3
    aux=incr(assi(aux,a))
    return aux

print(clr(5))
print(assi(3,7))
print(cero())
print(suce(3))

[2] ✓ 0.4s

... 0
7
0
4
```

Imagen 2: Ejecución de la función cero

En el bloque de código 3 se encuentra la definición de la operación suma, multiplicación y exponente, en la Imagen 3 y 4 se muestran las pruebas de ejecución de estas funciones.

```
1
2 def G(a) :
3     z1=2
4     return ( assi ( z1 , a ) )
5 def H(a) :
6     z1=2
7     return ( incr ( assi ( z1 , a ) ) )
```

```

8  def suma (x1 , x2) :
9      x3=1
10     z1=2
11     aux=3
12     z1=G(x1)
13     aux=assi (aux , x2)
14     x2=clr (x2)
15     while (aux !=0) :
16         x3=assi (x3 , z1)
17         z1=H(x3)
18         x2=incr (x2)
19         aux=decr (aux)
20     return (z1)
21 def mult (x1 , x2) :
22     aux=2
23     z1=2
24     z1=clr (z1)
25     aux=assi (aux , x2)
26     while (aux !=0) :
27         z1=assi (z1 , suma(z1 , x1))
28         aux=decr (aux)
29     return z1
30 def expo (x1 , x2) :
31     aux1=2
32     aux2=3
33     z1=2
34
35     z1=clr (z1)
36     aux1=assi (aux1 , x2)
37     aux2=assi (aux2 , x2)
38     aux2=decr (aux2)
39     while (aux1 !=0) :
40         z1=assi (z1 , mult(x1 , suce(cero())))
41         while (aux2 !=0) :
42             z1=assi (z1 , mult(z1 , x1))
43             aux1=decr (aux1)
44             aux2=decr (aux2)
45         return z1
46     return (suce(cero()))

```

Bloque de código 3: Definición de las funciones suma, multiplicación y exponente en Python.

```
def G(a):
    z1=2
    return (assi(z1,a))
def H(a):
    z1=2
    return (incr(assi(z1,a)))
def suma (x1,x2):
    x3=1
    z1=2
    aux=3
    z1=G(x1)
    aux=assi(aux,x2)
    x2=clr(x2)
    while (aux!=0):
        x3=assi(x3,z1)
        z1=H(x3)
        x2=incr(x2)
        aux=decr(aux)
    return(z1)
print(suma(3,7))
```

[3] ✓ 0.8s

... 10

Imagen 3: Ejecución de la función suma

```
def mult(x1,x2):
    aux=2
    z1=2
    z1=clr(z1)
    aux=assi(aux,x2)
    while (aux!=0):
        z1=assi(z1,suma(z1,x1))
        aux=decr(aux)
    return z1
def expo (x1,x2):
    aux1=2
    aux2=3
    z1=2

    z1=clr(z1)
    aux1=assi(aux1,x2)
    aux2=assi(aux2,x2)
    aux2=decr(aux2)
    while (aux1!=0):
        z1=assi(z1,mult(x1,suce(cero())))
        while (aux2!=0):
            z1=assi(z1,mult(z1,x1))
            aux1=decr(aux1)
            aux2=decr(aux2)
        return z1
    return (suce(cero()))

print(mult(5,3))
print(expo(3,5))
```

[5] ✓ 0.2s

... 15

243

Imagen 4: Ejecución de la función multiplicación y exponente

En el bloque de código 4 se encuentra la definición de la operación predecesor, monus (resta con tope a cero) e igualdad, en la Imagen 5 se muestran las prueba de ejecución de estas funciones.

```
1  def pred (x1):
2      z1=1
3      z1=clr (z1)
4      z1=assi (z1 , x1)
5      return (decr (z1))
6  def monus(x1 , x2):
7      aux=1
8      aux=assi (aux , x2)
9      z1=1
10     z1=assi (z1 , x1)
11     while (aux!=0):
12         z1=decr (z1)
13         aux=decr (aux)
14     return z1
15 def equ(x1 , x2):
16     a=3
17     b=3
18     aux=4
19     z1=4
20     a=assi (a , x1)
21     b=assi (b , x2)
22     aux=clr (aux)
23     z1=clr (z1)
24
25     z1=monus(a , b)
26     aux=monus(b , a)
27     z1=suma (z1 , aux)
28     z1=monus(( suce (cero ())) , z1)
29     return z1
```

Bloque de código 4: Definición de las funciones predecesor, monus e igualdad en Python.

```
def pred (x1):
    z1=1
    z1=clr(z1)
    z1=assi(z1,x1)
    return (decr(z1))

def monus(x1,x2):
    aux=1
    aux=assi(aux,x2)
    z1=1
    z1=assi(z1,x1)
    while(aux!=0):
        z1=decr(z1)
        aux=decr(aux)
    return z1

def equ(x1,x2):
    a=3
    b=3
    aux=4
    z1=4
    a=assi(a,x1)
    b=assi(b,x2)
    aux=clr(aux)
    z1=clr(z1)

    z1=monus[a,b]
    aux=monus(b,a)
    z1=suma(z1,aux)
    z1=monus((sucesero()),z1)
    return z1

print(pred(5))
print(monus(5,3))
print(equ(3,5))
print(equ(3,3))
```

[6] ✓ 0.9s

... 4
2
0
1

Imagen 5: Ejecución de la función predecesor, monus, igualdad a cero e igualdad a uno

En el bloque de código 5 se encuentra la definición de la operación cociente, fibonacci e impar, en la Imagen 6 se muestran las pruebas de ejecución de estas funciones.

```
1 def coci (x1 , x2) :
2     z1=3
```

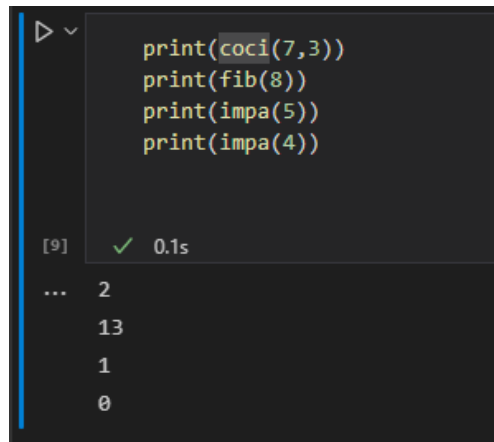
```

3      z1=clr ( z1 )
4      aux1=5
5      aux1=assi ( aux1 , x1 )
6
7      aux2=5
8      aux2=assi ( aux2 , x2 )
9
10     while ( aux1 !=0 ) :
11         while ( aux2 !=0 ) :
12             z1=suma( coci ( pred ( aux1 ) , aux2 ) , equ ( aux1 , suma ( aux2 , mult ( coci ( pred (
13                 ↪ aux1 ) , aux2 ) , aux2 ) ) ) )
14             aux2=clr ( aux2 )
15             aux1=clr ( aux1 )
16     return  z1
17 def  fib ( x1 ) :
18     z1=3
19     z1=clr ( z1 )
20
21     aux1=5
22     aux1=assi ( aux1 , x1 )
23
24     aux1=decr ( aux1 )
25     while ( aux1 !=0 ) :
26         aux1=decr ( aux1 )
27         while ( aux1 !=0 ) :
28             aux1=decr ( aux1 )
29             while ( aux1 !=0 ) :
30
31                 aux1=assi ( aux1 , x1 )
32                 z1=suma( fib ( decr ( aux1 ) ) , fib ( decr ( decr ( aux1 ) ) ) )
33                 return  z1
34
35     return  suce ( z1 )
36     return  suce ( z1 )
37     return  z1
38 def  impa ( x1 ) :
39     z1=3
40     z1=clr ( z1 )
41
42     aux1=5
43     aux1=assi ( aux1 , x1 )
44
45     while ( aux1 !=0 ) :
46         return  monus ( x1 , mult ( 2 , coci ( x1 , 2 ) ) )

```


46 **return** z1

Bloque de código 5: Definición de las funciones cociente, fibonacci e impar en python.



```
print(coci(7,3))
print(fib(8))
print(imp(5))
print(imp(4))
```

[9] ✓ 0.1s

... 2
13
1
0

Imagen 6: Ejecución de la función cociente, fibonacci e impar

En el bloque de código 6 se encuentra la definición de la operación factorial, división (empleando minimización) y función de Ackermann, en la Imagen 7 se muestra la prueba de ejecución de estas funciones.

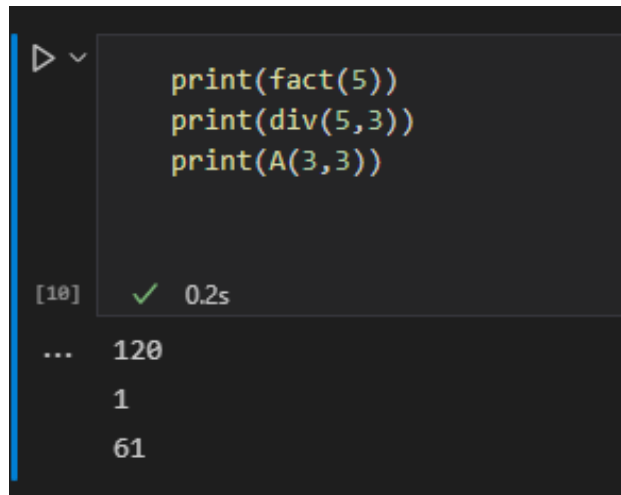
```
1  def fact (x1) :
2      z1=5
3      z1=clr ( z1 )
4
5      aux1=5
6      aux1=assi ( aux1 , x1 )
7
8      aux2=4
9      aux2=clr ( aux2 )
10     aux2=incr ( aux2 )
11
12     z1=incr ( z1 )
13     while ( aux1 != 0 ) :
14         z1=mult ( z1 , aux2 )
15         aux2=incr ( aux2 )
16         aux1=decr ( aux1 )
17     return z1
18
19 def GF (x1 , x2 , t) :
20     return monus ( ( suma ( x1 , 1 ) ) , ( suma ( ( mult ( t , x2 ) ) , x2 ) ) )
21 def div (x , y) :
22     z1=3
23     z1=clr ( z1 )
24
```

```

25     aux1=5
26     aux1=assi (aux1 , x)
27
28     aux2=5
29     aux2=assi (aux2 , y)
30
31     ite=6
32     ite=clr (ite)
33
34     z1=GF(aux1 , aux2 , ite)
35     while (z1!=0):
36         ite=incr (ite)
37         z1=GF(aux1 , aux2 , ite)
38     z1=assi (z1 , ite)
39     return z1
40 div (17 , 3)
41
42
43
44 def A(m,n):
45     z1=3
46     z1=clr (z1)
47
48     aux1=5
49     aux1=assi (aux1 , m)
50
51     aux2=5
52     aux2=assi (aux2 , n)
53
54     while (aux1!=0):
55         while (aux2!=0):
56             z1=A((decr (aux1)) ,(A (aux1 ,(decr (aux2)))))
57             return z1
58         z1=A((decr (aux1)) ,(suce (cero ())))
59         return z1
60     z1=incr (aux2)
61     return z1

```

Bloque de código 6: Definición de las funciones factorial, división y función de Ackermann en Python.



A screenshot of a code editor with a dark background. The code being executed is: `print(fact(5))`, `print(div(5,3))`, and `print(A(3,3))`. Below the code, a status bar shows `[10]`, a green checkmark, and `0.2s`. The output of the execution is displayed below the status bar: `...`, `120`, `1`, and `61`.

Imagen 7: Ejecución de la función factorial, división y Ackermann

En la Imagen 8 y 9 se puede ver el funcionamiento del interprete,

```
Este es un interprete de recursividad empleando un lenguaje de programación esencial  
accede a las operaciones con los numeros del 1 a 12, cada función requiere ciertos parametros  
de entrada y las operaciones disponibles son:  
  
1.- suma(x,y)  
2.- multiplicación(x,y)  
3.- exponenciación(x,y)  
4.- predecesor(x)  
5.- resta(x,y)  
6.- igualdad(x,y)  
7.- cociente(x,y), parte entera  
8.- Elemento x de la serie de fibonacci  
9.- identificación si un numero es impar  
10.- factorial(x)  
11.- división (x,y)  
12.- función de ackermann (x,y)  
selecciona alguna operación o q para salir
```

Imagen 8: Menú principal del interprete de funciones recursivas

```
Este es un interprete de recursividad empleando un lenguaje de programación esencial  
  
accede a las operaciones con los numeros del 1 a 12, cada función requiere ciertos parametros  
de entrada y las operaciones disponibles son:  
  
1.- suma(x,y)  
2.- multiplicación(x,y)  
3.- exponenciación(x,y)  
4.- predecesor(x)  
5.- resta(x,y)  
6.- igualdad(x,y)  
7.- cociente(x,y), parte entera  
8.- identificación si un numero es impar  
9.- Elemento x de la serie de fibonacci  
10.- factorial(x)  
11.- división (x,y)  
12.- función de ackermann (x,y)  
selecciona alguna operación o q para salir  
9  
  
¿Que numero de la serie de Fibonacci se quiere conocer?  
10  
el elemento 10 de la serie de fibonacci es 34  
pulsa una tecla para continuar
```

Imagen 9: Selección de la serie de Fibonacci en el interprete de funciones recursivas

Conclusiones

A pesar de que se puede calcular cualquier recursión, se deben emplear valores pequeños, ya que la cantidad de ciclos while y sumas incrementa de forma rápida, esto puede ser visto en la operación exponencial, factorial y Ackermann, en el caso del factorial el numero 10! no se pudo calcular debido a las limitaciones del hardware (la memoria RAM se satura y el procesador llega a un 100% de utilización), un comportamiento similar se presenta en la división en donde la división entre cero no esta definida y llenaría la memoria RAM al realizar la minimalización.