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 la 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.

```python
def incr (a):
    return (a+1)
def decr (a):
    while (a!=0):
        return (a-1)
    return (a)
```

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

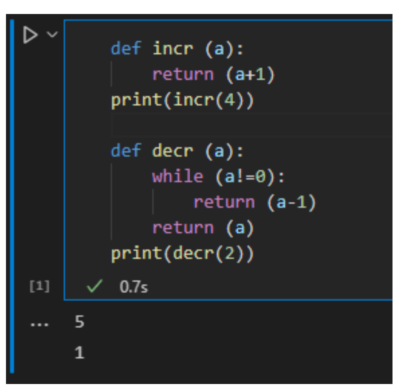


Imagen 1: *Ejecución de la función incremento y decremento*

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 pude ver la muestra de ejecución de estas operaciones.

```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
```

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

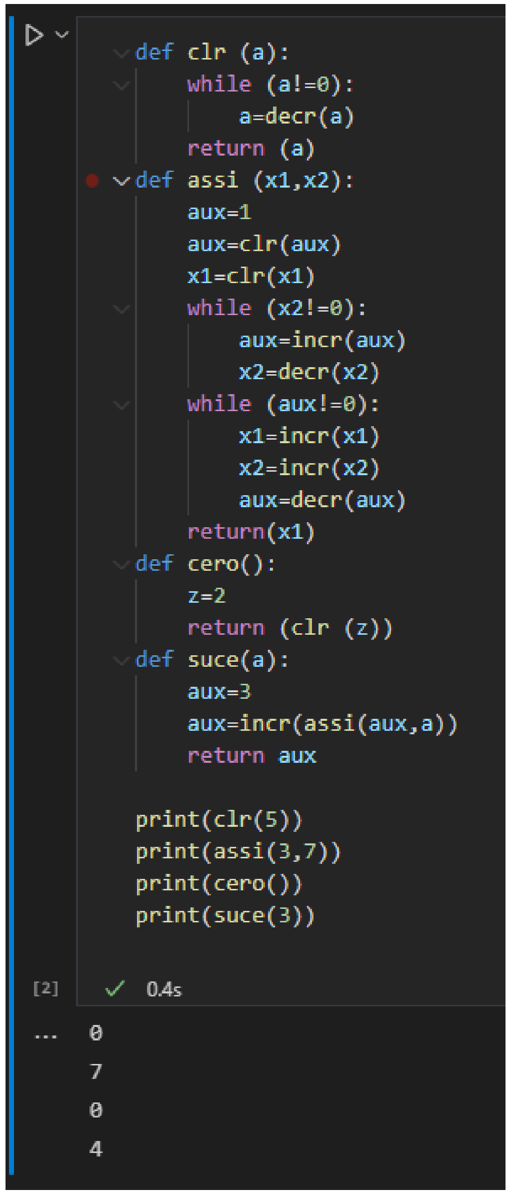


Imagen 2: *Ejecución de la función cero*

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.

```

def G(a):
    z1=2
    return (assi(z1,a))
def H(a):
    z1=2
    return (incr(assi(z1,a)))
```

```python
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)
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()))
```

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

```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*

```python
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.

```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((suce(cero())),z1)
    return z1
```

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

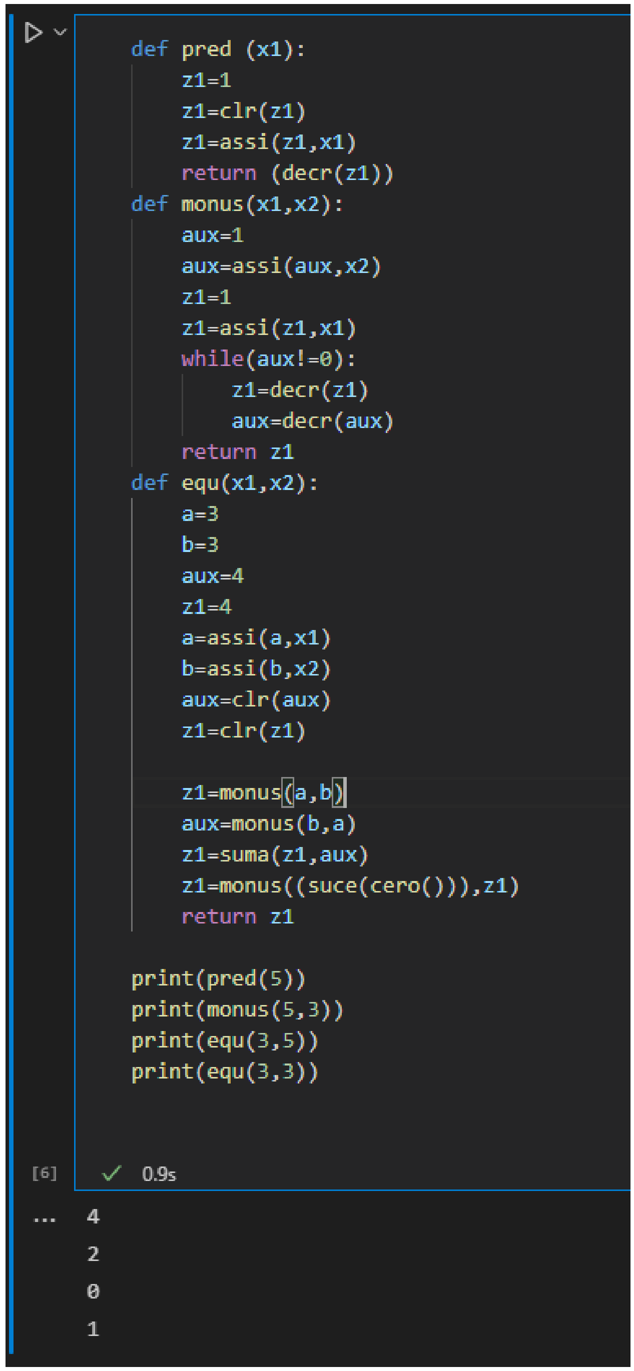


Imagen 5: *Ejecución de la función predecesor, monus, igualdad a cero e igualdad a uno*

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.

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

```
    z1=clr(z1)
    aux1=5
    aux1=assi(aux1,x1)

    aux2=5
    aux2=assi(aux2,x2)

    while(aux1!=0):
        while(aux2!=0):
            z1=suma(coci(pred(aux1),aux2),equ(aux1,suma(aux2,mult(coci(pred(aux1),aux2),aux2))))
            aux2=clr(aux2)
        aux1=clr(aux1)
    return z1
def fib(x1):
    z1=3
    z1=clr(z1)

    aux1=5
    aux1=assi(aux1,x1)

    aux1=decr(aux1)
    while(aux1!=0):
        aux1=decr(aux1)
        while(aux1!=0):
            aux1=decr(aux1)
            while(aux1!=0):

                aux1=assi(aux1,x1)
                z1=suma(fib(decr(aux1)),fib(decr(decr(aux1))))
                return z1

            return suce(z1)
        return suce(z1)
    return z1
def impa(x1):
    z1=3
    z1=clr(z1)

    aux1=5
    aux1=assi(aux1,x1)

    while(aux1!=0):
        return monus(x1,mult(2,coci(x1,2)))
```

```
    return z1
```

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

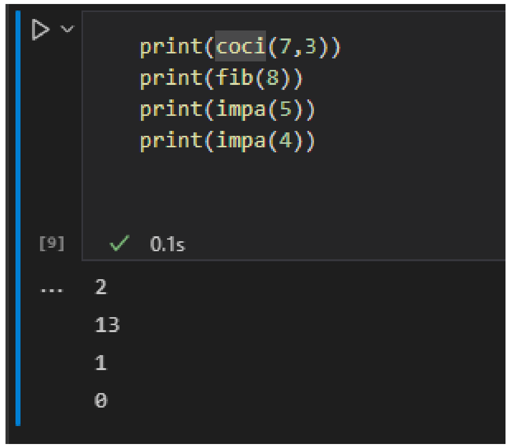


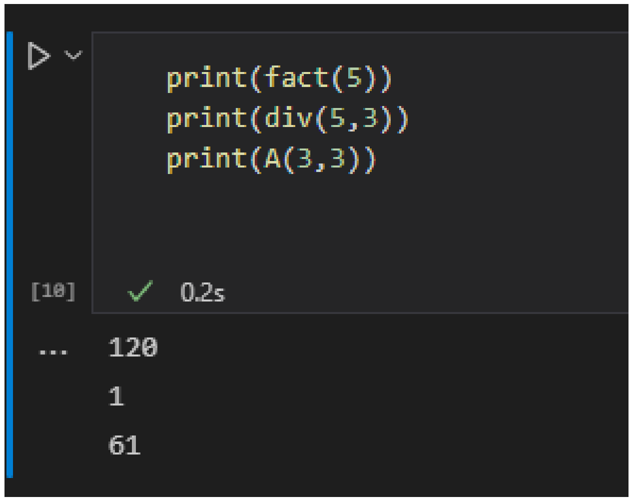
Imagen 6: *Ejecución de la función cociente, fibbonaci e impar*

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.

```
def fact(x1):
    z1=5
    z1=clr(z1)

    aux1=5
    aux1=assi(aux1,x1)

    aux2=4
    aux2=clr(aux2)
    aux2=incr(aux2)

    z1=incr(z1)
    while(aux1!=0):
        z1=mult(z1,aux2)
        aux2=incr(aux2)
        aux1=decr(aux1)
    return z1

def GF (x1,x2,t):
    return monus((suma (x1,1)),(suma((mult(t,x2)),x2)))
def div(x,y):
    z1=3
    z1=clr(z1)

```

```
    aux1=5
    aux1=assi(aux1,x)

    aux2=5
    aux2=assi(aux2,y)

    ite=6
    ite=clr(ite)

    z1=GF(aux1,aux2,ite)
    while(z1!=0):
        ite=incr(ite)
        z1=GF(aux1,aux2,ite)
    z1=assi(z1,ite)
    return z1
div(17,3)



def A(m,n):
    z1=3
    z1=clr(z1)

    aux1=5
    aux1=assi(aux1,m)

    aux2=5
    aux2=assi(aux2,n)

    while(aux1!=0):
        while(aux2!=0):
            z1=A((decr(aux1)),(A (aux1,(decr(aux2)))))
            return z1
        z1=A((decr(aux1)),(suce(cero())))
        return z1
    z1=incr(aux2)
    return z1
```

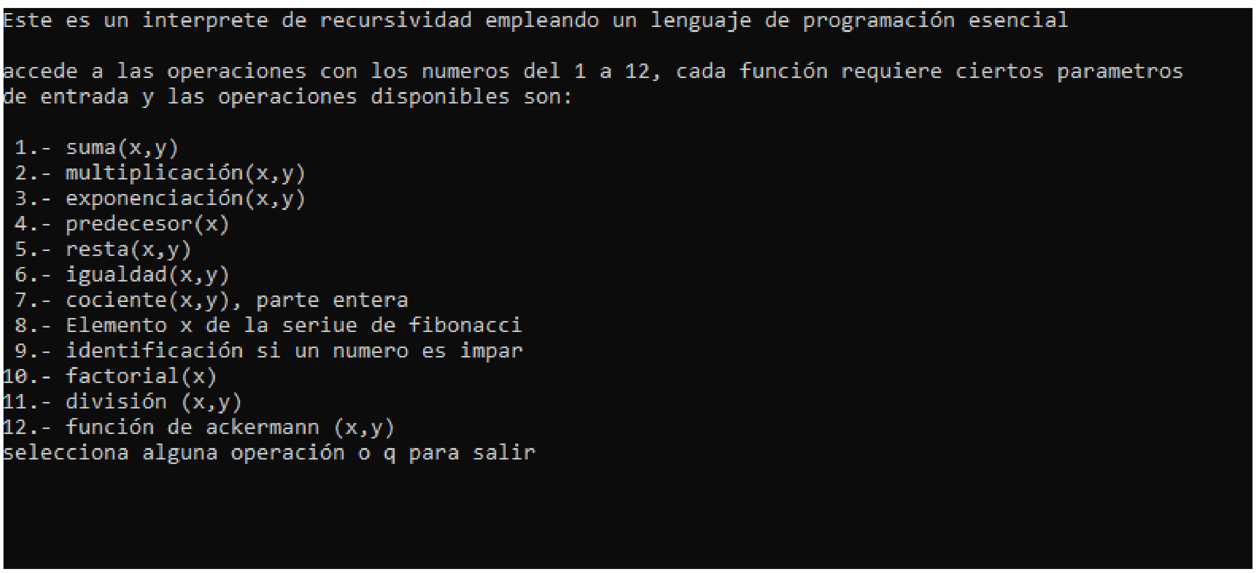
Bloque de código 6: *Definición de las funciones factorial, división y función de Ackermann en Python.*



```
print(fact(5))
print(div(5,3))
print(A(3,3))

[10]  ✓ 0.2s

...  120
     1
     61
```

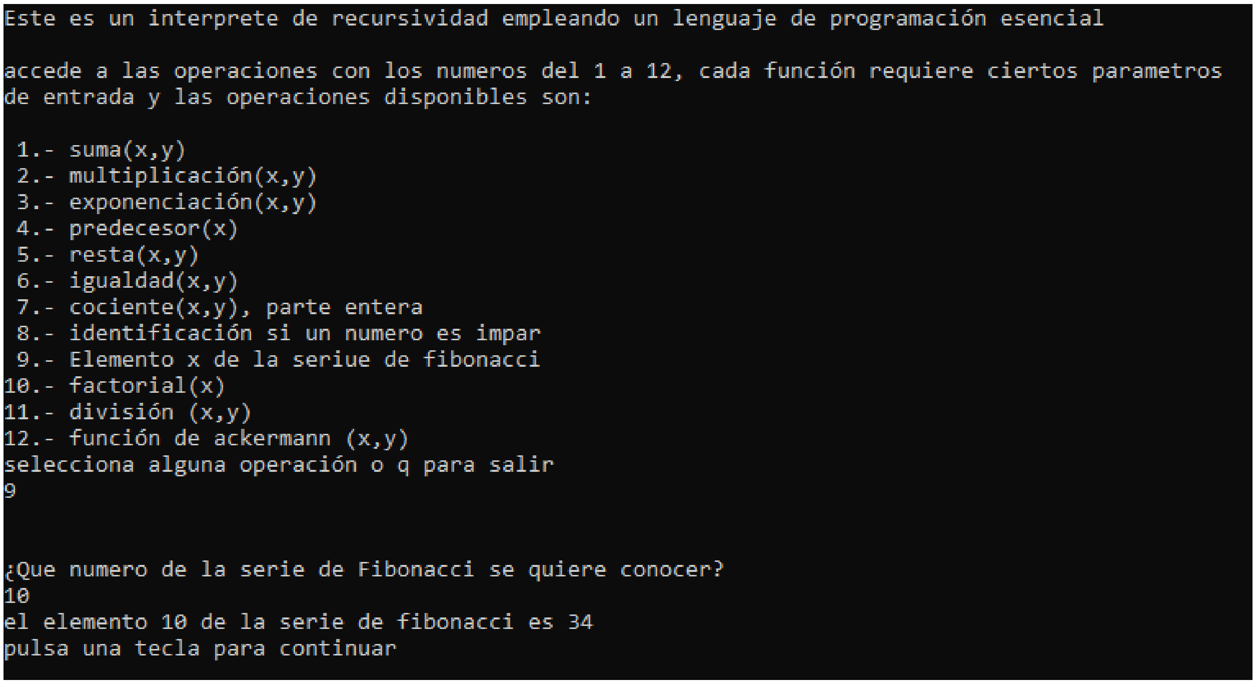
Imagen 7: *Ejecución de la función factorial, división y Ackermann*

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 seriue 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 seriue 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.