

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

FACULTAD DE CIENCIAS E INGENIERÍA



1INF06: ESTRUCTURA DE DATOS Y PROGRAMACIÓN METÓDICA

Docente:

BELLO RUIZ, ALEJANDRO TORIBIO

H – 0582

Integrantes:

| | |
|------------------------------------|----------|
| Campana Yauri, Dan-el Mauricio | 20125121 |
| Palacios Coronado, Antony | 20141735 |
| Perez Cabrera, Estephany Elizabeth | 20142604 |

Tema:

INFORME 4: Desarrollo de programas en Linux Mint 18.3 Cinnamon usando los lenguajes de programación Python 3 y Haskell.

Abril del 2018

ÍNDICE:

| | |
|---|-----------|
| Introducción..... | 3 |
| Implementación de la TAD Pilas en Haskell..... | 3 |
| Integración de la implementación en Haskell con la interfaz en Python..... | 14 |
| Parte trabajada por los integrantes..... | 17 |
| Comentarios..... | 18 |
| Bibliografías..... | 19 |
| Anexos..... | 20 |

Introducción

En el siguiente informe se presentará la implementación del tipo de dato abstracto Pila en base a las especificaciones sugeridas en el material del curso EDMA. El lenguaje utilizado para realizar dicha implementación fue Haskell. Este lenguaje es de tipo funcional ya que trabaja bajo la modalidad de que todo el programa se constituye únicamente de funciones matemáticas.

El tipo de dato pilas es una estructura de datos lineal cuya característica principal es que el acceso a los elementos se realiza en el orden inverso al de su almacenamiento. Por esa razón, esta estructura es llamada LIFO (Last In, First Out – último en entrar, primero en salir).

Los tipos de datos LIFO son muy usados en los diseños de algoritmos, ya que resultan muy útiles, por ejemplo, en algunas aplicaciones como una calculadora, se usan para almacenar y evaluar los datos de las operaciones introducidas, así también se emplean en los editores de texto para determinar la posición del cursor, y en las páginas web para manejar las páginas visitadas anteriormente, etc.

Implementación de la TAD Pilas en Haskell

especificación PILAS[ELEM]

usa BOOLEANOS

tipos pila

operaciones

| | | |
|-----------------|-----------------|-------------------------|
| pila-vacia | : | → pila {constructora} |
| apilar | : elemento pila | → pila {constructora} |
| desapilar | : pila | → _p pila |
| cima | : pila | → _p elemento |
| es-pila- vacia? | : pila | → bool |

variables

e : elemento
p : pila

* Para empezar, definimos el tipo de dato Pila en base a sus constructoras, en este caso serían Vacía (que representa una pila vacía) y P x (Stack x) (que representa la acción de apilar un elemento en una pila).

data Stack x = P x (Stack x) | Vacía

* Para probar el correcto funcionamiento de algunas de las operaciones implementadas que devuelven una pila se creó la función imprimir. Primero se define el caso base, en el cual se recibe una pila vacía y se devuelve una lista vacía.

Para el caso general, se hace uso de la recursividad, devolviendo la cima en una lista y concatenándolo con el resultado de imprimir el resto de la pila hasta que eventualmente se llegue al caso base y se concatene con una lista vacía.

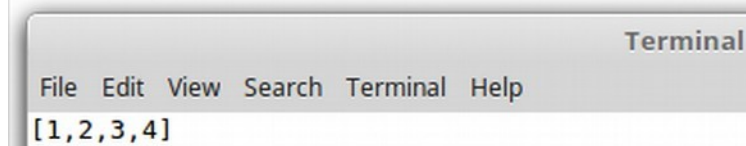
```
imprimir :: Stack x -> [x]
imprimir Vacía = []
imprimir (P x xs) = [x] ++ imprimir(xs)
```

```
main = do
```

```
    print(imprimir (P 1 (P 2 (P 3 (P 4 Vacía)))))
```

```
[1,2,3,4]
```

```
main = do
    print(imprimir (P 1 (P 2 (P 3 (P 4 Vacía)))))
```



* Para la operación desapilar, primero planteamos el caso de error en el cual se recibe una pila vacía. Para el caso general, se representa la pila recibida en forma de su constructora apilar, es decir con un elemento y una pila, y “se remueve la cima” al devolver solo la pila.

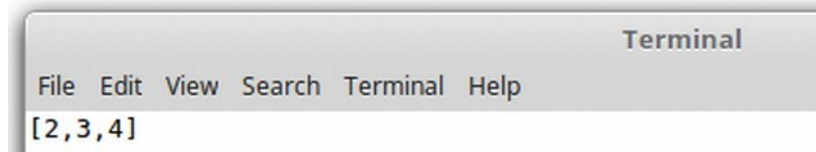
```
desapilar :: Stack x -> Stack x
desapilar Vacía = error "ERROR"
desapilar (P _ p) = p
```

```
main = do
```

```
    print(imprimir(desapilar(P 1 (P 2 (P 3 (P 4 Vacía)))))
```

```
[2,3,4]
```

```
main = do
    print(imprimir(desapilar(P 1 (P 2 (P 3 (P 4 Vacía)))))
```



* Para la operación cima, primero se plantea el caso de error si es que se recibe una pila vacía. Para el caso general, se representa la pila recibida en forma de su constructora, es decir apilando un elemento al resto de la pila, y se devuelve ese elemento.

```
cima :: Stack x -> x
cima Vacía = error "ERROR"
cima (P x _) = x
```

```
main = do
```

```
    print(cima (P 1 (P 2 (P 3 (P 4 Vacía)))))
```

```

1
main = do
  print(cima (P 1 (P 2 (P 3 (P 4 Vacia)))))

```



* Para la operación `esta_vacia`, primero se coloca el caso obvio, es decir cuando se recibe una pila vacía y se devuelve `True`. Para cualquier otro caso, es decir sin importar lo que se reciba, se devuelve `False`.

```

esta_vacia :: Stack x -> Bool
esta_vacia Vacia = True
esta_vacia _ = False

```

```

main = do
  print(esta_vacia (P 1 (P 2 (P 3 (P 4 Vacia)))))

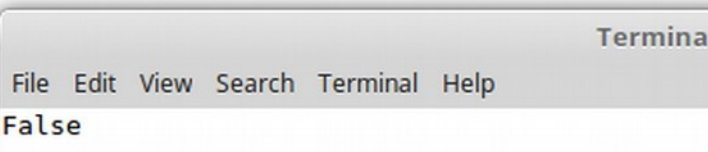
```

False

```

main = do
  print(esta_vacia (P 1 (P 2 (P 3 (P 4 Vacia)))))

```



Extendiendo Pilas:

especificación PILAS+(ELEM)
usa PILAS(ELEM), NATURALES
operaciones

```

profundidad : pila → nat
fondo : pila → p elemento
inversa : pila → pila
duplicar : pila → pila
concatenar : pila pila → pila
entremezclar : pila pila → pila

```

operaciones privadas

```

apilar-inversa : pila pila → pila

```

variables

```

e, f : elemento
p, q : pila

```

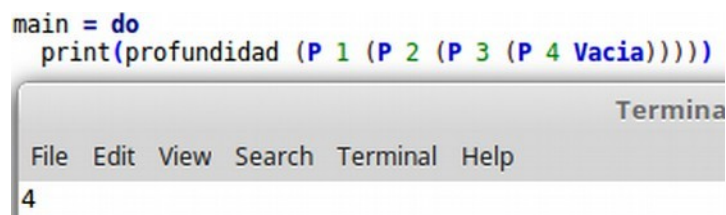
* Para la operación profundidad, primero se define el caso base, es decir cuando se recibe una pila vacía y se devuelve como resultado cero. En el caso general, se hace uso de la recursividad, quitando el primer elemento de la pila hasta llegar al caso base y sumando 1 por cada elemento retirado de la pila.

```
profundidad :: Stack x -> Int
profundidad Vacía = 0
profundidad (P x xs) = 1 + profundidad (xs)

main = do

    print(profundidad (P 1 (P 2 (P 3 (P 4 Vacía)))))
```

4



```
main = do
    print(profundidad (P 1 (P 2 (P 3 (P 4 Vacía)))))
```

Termina

File Edit View Search Terminal Help

4

* Para la operación fondo, primero se define el caso de error, el cual será cuando se reciba una pila vacía. En el caso general, se declara como caso base cuando se reciba una pila con un único elemento y se devuelve ese elemento. Para los demás casos se hace uso de la recursividad, retirando cada uno de los elementos hasta llegar al caso base.

```
fondo :: Stack x -> x
fondo Vacía = error "ERROR"
fondo (P x Vacía) = x
fondo (P x xs) = fondo(xs)

main = do

    print(fondo (P 1 (P 2 (P 3 (P 4 Vacía)))))
```

4



```
main = do
    print(fondo (P 1 (P 2 (P 3 (P 4 Vacía)))))
```

Te

File Edit View Search Terminal Help

4

*Para la operación privada apilar_inversa, primero se define el caso base, en el cual se recibe una pila vacía y una pila cualquiera, y se devuelve la segunda pila. En el caso general, se hace uso de la recursividad para ir apilando la cima de la primera pila y después llamar de nuevo a la función con lo que queda de la primera pila para de nuevo apilar su cima y así sucesivamente hasta llegar al caso base.

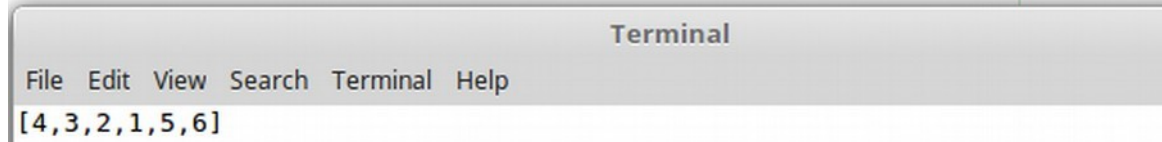
```
apilar_inversa :: Stack x -> Stack x -> Stack x
apilar_inversa (Vacía) (P y ys) = (P y ys)
apilar_inversa (P x xs) ys = apilar_inversa (xs) (P x ys)
```

```
main = do
```

```
  print(imprimir( apilar_inversa(P 1 (P 2 (P 3 (P 4 Vacia)))) (P 5 (P 6 Vacia))))
```

```
[4,3,2,1,5,6]
```

```
main = do
  print(imprimir( apilar_inversa(P 1 (P 2 (P 3 (P 4 Vacia)))) (P 5 (P 6 Vacia))))
```



*Para definir la operación inversa se hace uso de la operación `apilar_inversa`. Primero, se define el caso base, en el cual se recibe una pila vacía y se devuelve también una pila vacía. En el caso general, se utiliza `apilar_inversa`, la cual recibe una pila cualquiera y una pila vacía, para finalmente apilar la pila recibida de manera inversa sobre la pila vacía.

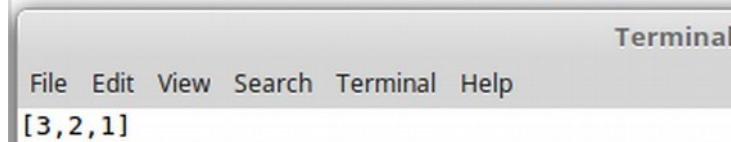
```
inversa :: Stack x -> Stack x
inversa Vacia = Vacia
inversa xs = apilar_inversa (xs) (Vacia)
```

```
main = do
```

```
  print(imprimir(inversa(P 1 (P 2( P 3 Vacia)))))
```

```
[3,2,1]
```

```
main = do
  print(imprimir(inversa(P 1 (P 2( P 3 Vacia)))))
```



*Para la operación duplicar, primero se define el caso base, en el cual se recibe una pila vacía y se devuelve una pila vacía. Para el caso general se representa la pila recibida en la forma de la constructora `apilar` un elemento sobre lo que queda de la pila; de este modo se apila dos veces ese elemento, que sería la cima, y luego se llama de manera recursiva a la misma función con lo que queda de la pila hasta llegar al caso base.

```
duplicar :: Stack x -> Stack x
duplicar Vacia = Vacia
duplicar (P x xs) = (P x (P x (duplicar(xs))))
```

```
main = do
```

```
  print(imprimir( duplicar(P 1 (P 2 (P 3 (P 4 Vacia)))))
```

```
[1,1,2,2,3,3,4,4]
```

```
main = do
  print(imprimir( duplicar(P 1 (P 2 (P 3 (P 4 Vacia))))))
```

Terminal

File Edit View Search Terminal Help

[1,1,2,2,3,3,4,4]

* Para la operación concatenar, primero se define el caso base, en el cual al recibir una pila y una pila vacía, nos devuelve la misma pila. Para el caso general se mantiene la acción de apilar la cima de la segunda pila en espera hasta que la recursividad llegue al caso base, es decir cuando se quede vacía, en ese momento se empieza a apilar los elementos que quedaron en espera de la segunda pila a la primera pila.

```
concatenar :: Stack x -> Stack x -> Stack x
concatenar xs Vacia = xs
concatenar xs (P y ys) = (P y (concatenar xs ys))
```

```
main = do
```

```
  print(imprimir( concatenar (P 5(P 6 Vacia))(P 1 (P 2 (P 3 (P 4
Vacia)))))
```

[1, 2, 3, 4, 5, 6]

```
main = do
  print(imprimir( concatenar (P 5(P 6 Vacia))(P 1 (P 2 (P 3 (P 4 Vacia)))))
```

Terminal

File Edit View Search Terminal Help

[1,2,3,4,5,6]

* Para la operación entremezclar, primero se define el caso base en el cual se reciben dos pilas vacías y se devuelve una pila vacía. Así también para el caso en el que se reciba una pila y una pila vacía se devuelve la pila. Para el caso general, primero se consulta cual de las pilas tiene mayor longitud y dependiendo de eso se elige un caso. En el caso de que la primera pila sea de mayor longitud, se debe apilar su cima sobre el resultado de entremezclar lo que queda de esa pila con la segunda pila de manera recursiva. Y en el caso de que la segunda pila sea de mayor longitud, se apila la cima de la segunda pila sobre el resultado de entremezclar la primera pila con lo que queda de la segunda de manera recursiva.

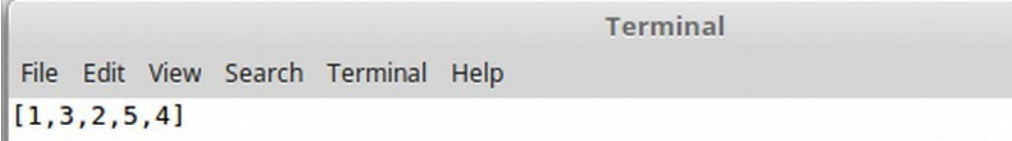
```
entremezclar :: Stack x -> Stack x -> Stack x
entremezclar Vacia Vacia = Vacia
entremezclar (xs) (Vacía) = xs
entremezclar (Vacía) (ys) = ys
entremezclar (P x xs) (P y ys)
  | (profundidad(xs) + 1) > (profundidad(ys) + 1) = (P x
(entremezclar (xs) (P y ys) ))
  | otherwise = (P y (entremezclar (P x xs) (ys) ))
```

```
main = do
```

```
  print(imprimir(entremezclar(P 1 (P 3 (P 5 Vacia))(P 2(P 4 Vacia))))
```


[1,2,3,4,5]

```
main = do
  print(imprimir(entremezclar(P 1 (P 3 (P 5 Vacia)))(P 2(P 4 Vacia))))
```



Secuencia de pilas:

especificación SECUENCIAS[ELEM]

usa PILAS[ELEM], BOOLEANOS

tipos secuencia

operaciones

crear : \leftarrow secuencia
insertar : secuencia elemento \leftarrow secuencia
eliminar : secuencia \leftarrow p secuencia
actual : secuencia \leftarrow p secuencia
avanzar : secuencia \leftarrow p secuencia
reiniciar : secuencia \leftarrow p secuencia
fin? : secuencia \leftarrow p bool
es-sec- vacía? : secuencia \leftarrow p bool

operaciones privadas

($_$, $_$) : pila pila \leftarrow secuencia { constructora }

Variables

e : elemento
s : secuencia
iz, dr : pila

* Para la operación crear una secuencia, se recibe dos pilas y se devuelve el resultado de apilar_inversa, cuyos parámetros serían las pilas recibidas. De este modo se crea una secuencia ordenada de dos pilas, cuyas cimas se colocan de manera consecutiva. Cabe resaltar que en una secuencia se denomina punto de interés a la cima de la segunda pila. (Se pintará el punto de interés de color verde)

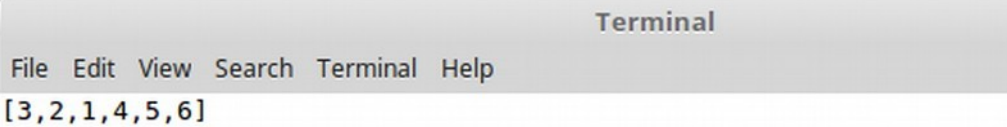
```
crear :: Stack x -> Stack x -> Stack x
crear xs ys = apilar_inversa xs ys
```

```
main = do
```

```
  print(imprimir(crear (P 1 (P 2 (P 3 Vacia)))(P 4 (P 5 (P 6 Vacia)))))
```

[3, 2, 1, 4, 5, 6]

```
main = do
  print(imprimir(crear (P 1 (P 2 (P 3 Vacia)))(P 4 (P 5 (P 6 Vacia)))))
```



Terminal

File Edit View Search Terminal Help

[3,2,1,4,5,6]

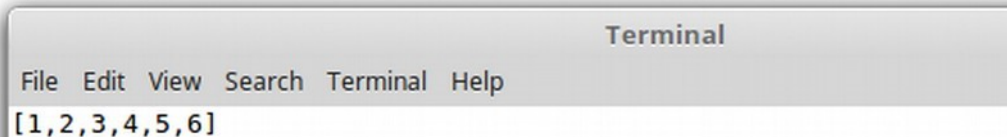
* Para la operación insertar un elemento delante del punto de interés, se hace uso de la función `apilar_inversa`, cuyos parámetros serían el resultado de apilar dicho elemento en la primera pila y la segunda pila. De este modo el elemento quedaría apilado sobre la cima de la segunda pila, es decir sobre el punto de interés.

```
insertar :: Stack x -> Stack x -> x -> Stack x
insertar xs ys e = apilar_inversa (P e xs) (ys)
```

```
main = do
  print(imprimir(insertar(P 3 (P 2 (P 1 Vacia))) (P 5 (P 6 Vacia)) 4))
```

[1, 2, 3, 4, 5, 6]

```
main = do
  print(imprimir(insertar(P 3 (P 2 (P 1 Vacia))) (P 5 (P 6 Vacia)) 4))
```



Terminal

File Edit View Search Terminal Help

[1,2,3,4,5,6]

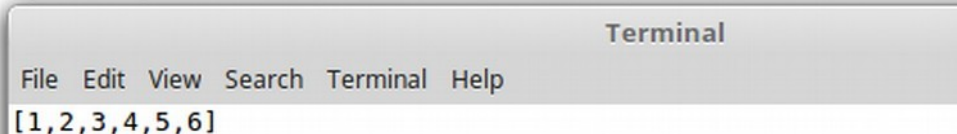
* Para la operación eliminar el elemento en el punto de interés, primero se define el caso de error cuando se recibe una pila y una pila vacía, es decir cuando no existe el punto de interés. Para el caso general, se coloca la segunda pila en la forma de su constructora apilar un elemento sobre lo que queda de la pila y se hace uso de la función `apilar_inversa`, cuyos parámetros serían la primera pila y lo que queda de la segunda pila.

```
eliminar :: Stack x -> Stack x -> Stack x
eliminar xs Vacia = error "ERROR"
eliminar xs (P y ys) = apilar_inversa xs ys
```

```
main = do
  print(imprimir(eliminar(P 3 (P 2 (P 1 Vacia))) (P 5 (P 6 Vacia)) )
```

[1, 2, 3, 6]

```
main = do
  print(imprimir(eliminar(P 3 (P 2 (P 1 Vacia))) (P 5 (P 6 Vacia)) )
```



Terminal

File Edit View Search Terminal Help

[1,2,3,4,5,6]

* Para la operación actual, la cual se refiere a mostrar el punto de interés, primero se define el caso de error cuando se reciba una pila y una pila vacía, ya que no existiría el punto de interés. Para el caso general, se coloca la segunda pila recibida en la forma de la constructora apilar un elemento sobre lo que queda de la pila y se devuelve dicho elemento, es decir la cima de la segunda pila, ya que corresponde al punto de interés.

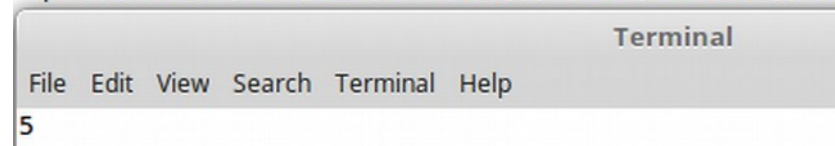
```
actual :: Stack x -> Stack x -> x
actual xs Vacía = error "ERROR"
actual xs (P y _) = y
```

```
main = do
```

```
    print(actual (P 3 (P 2 (P 1 Vacía))) (P 5 (P 6 Vacía)) )
```

5

```
main = do
    print(actual (P 3 (P 2 (P 1 Vacía))) (P 5 (P 6 Vacía)) )
```



* Para la operación avanzar, primero se define el caso de error cuando se reciba una pila y una pila vacía ya que no existiría ningún elemento de la segunda pila que pueda ser avanzado hacia la primera pila. Para el caso general, se coloca la segunda pila recibida en la forma de la constructora apilar un elemento sobre lo que queda de la pila y se usa la función apilar_inversa, cuyos parámetros serían el resultado de apilar la cima de la segunda pila sobre la primera pila y lo que queda de la segunda pila. De este modo se logra mover el punto de interés al elemento siguiente de la segunda pila.

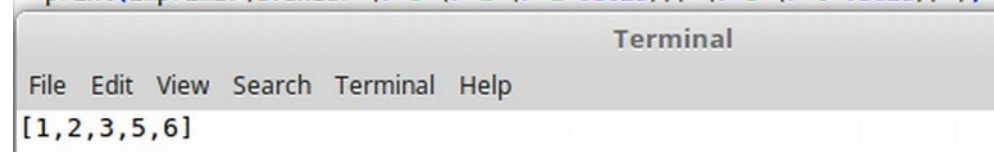
```
avanzar :: Stack x -> Stack x -> Stack x
avanzar xs Vacía = error "ERROR"
avanzar xs (P y ys) = apilar_inversa (P y xs) ys
```

```
main = do
```

```
    print(imprimir(avanzar (P 3 (P 2 (P 1 Vacía))) (P 5 (P 6 Vacía)) ))
```

[1, 2, 3, 5, 6]

```
main = do
    print(imprimir(avanzar (P 3 (P 2 (P 1 Vacía))) (P 5 (P 6 Vacía)) ))
```



* Para la operación reiniciar, primero se define el caso base cuando se reciba una pila vacía y una pila ya que no habrían elementos para pasar de la primera pila a la segunda pila, por lo que solo se retorna la segunda pila. Para el caso general, se coloca la primera pila en la forma de la constructora apilar un elemento sobre lo que

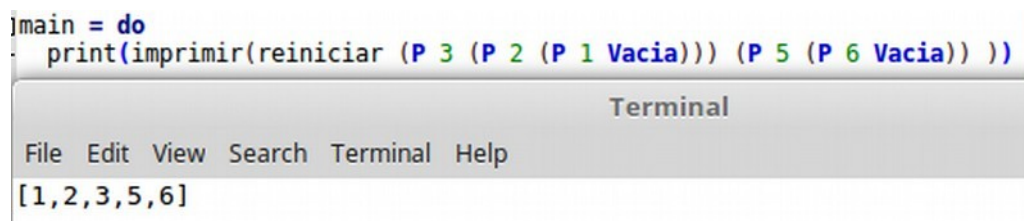
queda de la pila y se hace uso de la recursividad con la misma función, cuyos parámetros serían lo que queda de la primera pila y el resultado de apilar la cima de la primera pila a la segunda pila. Este proceso se realizará hasta que eventualmente la primera pila quede vacía, es decir cuando todos sus elementos hayan pasado a la segunda pila.

```
reiniciar :: Stack x -> Stack x -> Stack x
reiniciar Vacía xs = xs
reiniciar (P x xs) (ys) = reiniciar (xs) (P x ys)
```

```
main = do
```

```
    print(imprimir(reiniciar (P 3 (P 2 (P 1 Vacía))) (P 5 (P 6 Vacía)) ))
```

```
[1, 2, 3, 5, 6]
```



```
main = do
  print(imprimir(reiniciar (P 3 (P 2 (P 1 Vacía))) (P 5 (P 6 Vacía)) ))
```

Terminal

File Edit View Search Terminal Help

[1,2,3,5,6]

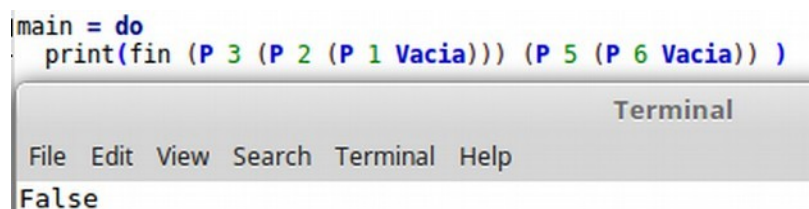
* Para la operación fin, la cual nos devuelve si la segunda pila ya ha sido transferida a la primera pila, primero se define el caso obvio, es decir cuando se reciba una pila y una pila vacía, en este caso se devuelve True ya que la segunda pila no tiene más elementos para transferir. El otro caso es cuando se recibe dos pilas cualquiera, lo cual implica que la segunda pila aún tiene uno o más elementos por lo que se devuelve False.

```
fin :: Stack x -> Stack x -> Bool
fin _ Vacía = True
fin _ _ = False
```

```
main = do
```

```
    print(fin (P 3 (P 2 (P 1 Vacía))) (P 5 (P 6 Vacía)) )
```

```
False
```



```
main = do
  print(fin (P 3 (P 2 (P 1 Vacía))) (P 5 (P 6 Vacía)) )
```

Terminal

File Edit View Search Terminal Help

False

* Finalmente, para la operación es_sec_vacia, la cual nos retorna si tanto la primera como la segunda pila son vacías, primero se define el caso evidente, es decir cuando se recibe dos pilas vacías se devuelve True, y en cualquier otro caso se devuelve False.

```
es_sec_vacia :: Stack x -> Stack x -> Bool
es_sec_vacia Vacía Vacía = True
```

```
es_sec_vacia _ _ = False
```

```
main = do
```

```
    print(es_sec_vacia (P 3 (P 2 (P 1 Vacia))) (P 5 (P 6 Vacia)) )
```

```
False
```

```
main = do
```

```
    print(es_sec_vacia (P 3 (P 2 (P 1 Vacia))) (P 5 (P 6 Vacia)) )
```

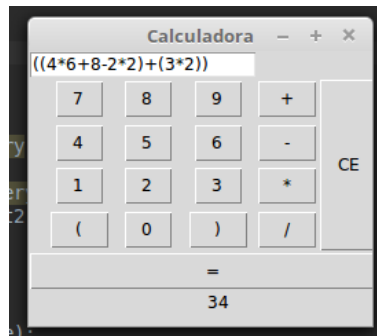
Terminal

File Edit View Search Terminal Help

False

IMPLEMENTACIÓN EN PYTHON USANDO HASKELL

Ventana de librería Tkinter



El programa desarrollado es una calculadora infija que usa pilas para separar los números y los símbolos.

Para la operación se seguirá el siguiente algoritmo:

1. Read an input character
2. Actions that will be performed at the end of each input
 - Opening brackets (2.1) Push it into stack and then Go to step (1)
 - Digit (2.2) Push into stack, Go to step (1)
 - Operator (2.3) Do the comparative priority check
 - (2.3.1) if the character stack's top contains an operator with equal or higher priority, then pop it into op
 - Pop a number from integer stack into op2
 - Pop another number from integer stack into op1
 - Calculate op1 op op2 and push the result into the integer stack
 - Closing brackets (2.4) Pop from the character stack
 - (2.4.1) if it is an opening bracket, then discard it and Go to step (1)
 - (2.4.2) To op, assign the popped element
 - Pop a number from integer stack and assign it op2
 - Pop another number from integer stack and assign it to op1
 - Calculate op1 op op2 and push the result into the integer stack
 - Convert into character and push into stack
 - Go to the step (2.4)
 - New line character (2.5) Print the result after popping from the STOP

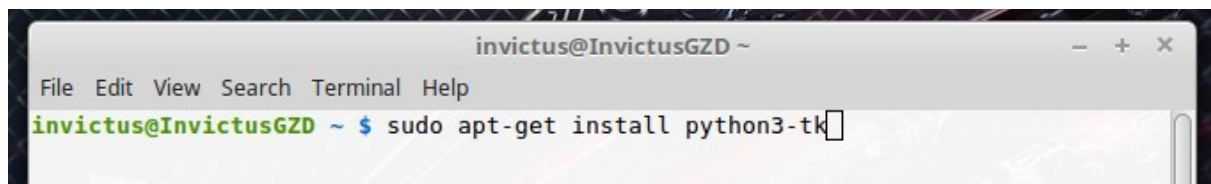
Véase código de python en el Anexo

Conexión Python3 y Haskell:

Para la creación de la interfaz de Python usaremos la librería Tkinter. Además, se usará funciones implementadas en Haskell para la obtención y devolución de datos. Para la conexión de ambos lenguajes usaremos un archivo de que almacene texto. Este puede ser de cualquier extensión, por ejemplo .txt.

Para descargar la librería tkinter usaremos el siguiente comando en la terminal:

sudo apt-get install python3-tk



Implementación de funciones en Haskell:

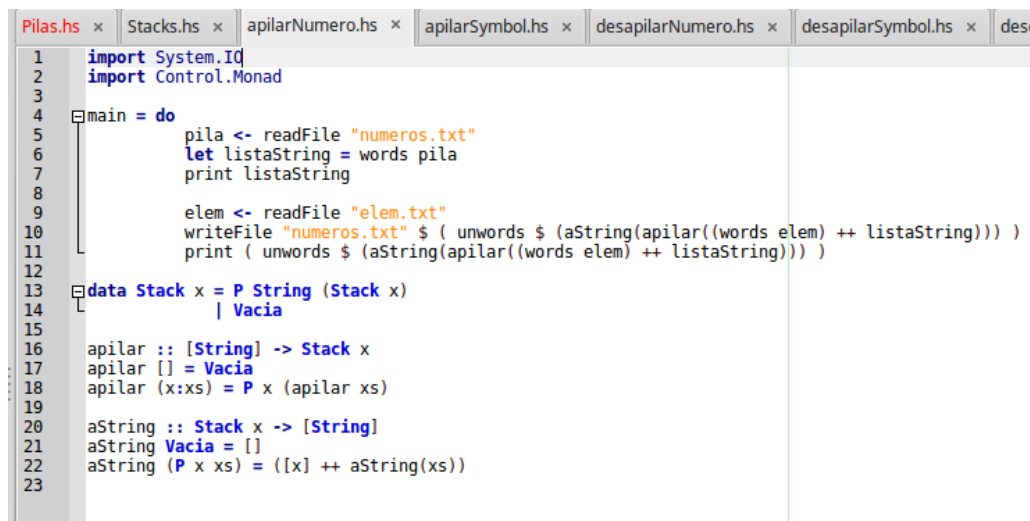
Primero, crearemos las funciones en Haskell las cuales usaremos en la interfaz más adelante.

Para ello crearemos un archivo “.hs”, el cual tendrá como funciones:

readFile: Para leer el archivo que almacena los datos

words: Obtener las palabras del archivo de texto y colocarlos como una lista de strings.

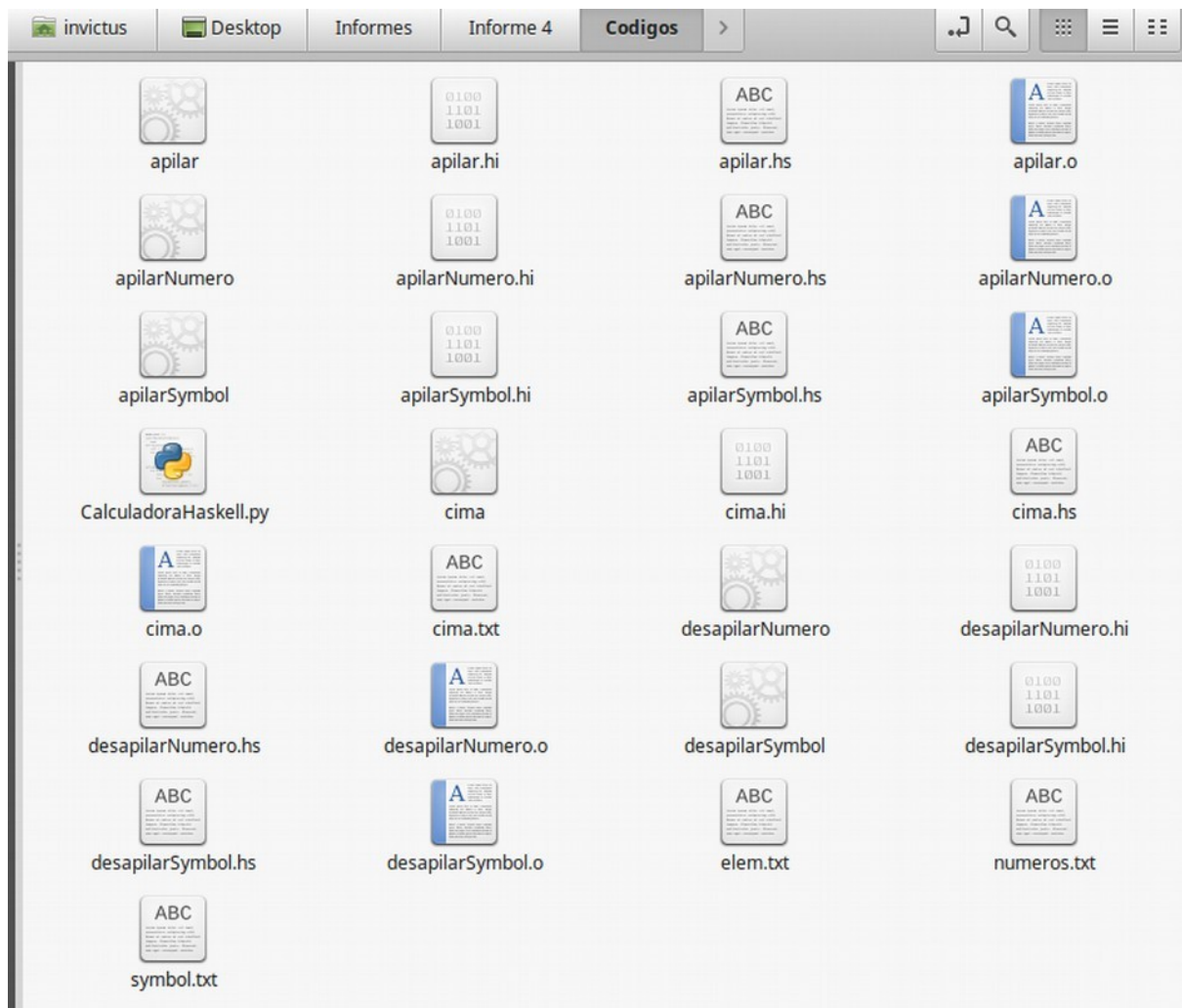
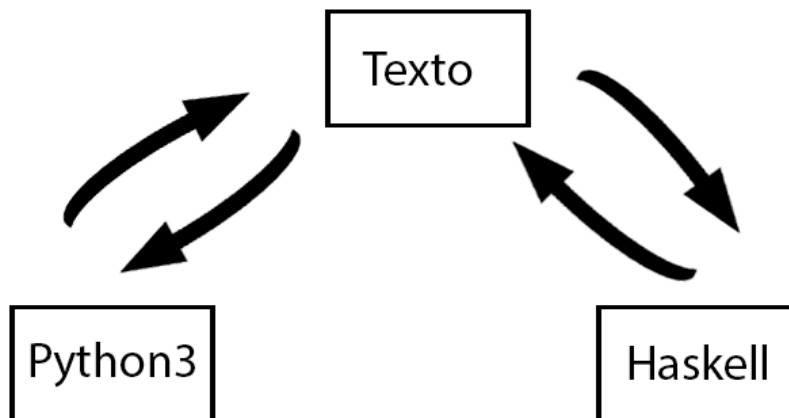
writeFile: Colocaremos los datos en un archivo de texto.



Segundo, se tendrá archivos de texto donde se almacenarán los datos, llamados, por ejemplo, numeros.txt (se almacena los números en una lista, la cual será la pila) y elem.txt (se guarda el elemento que se usará para apilar).

En cada archivo de Haskell se colocará la función a realizar y usar los comandos para editar archivos de texto.

Explicación en imagen del funcionamiento:



Para descargar códigos:

<https://drive.google.com/drive/folders/1uWmOnDws5Y9PWPiq4UAU4GehELmJnSc?usp=sharing>

Partes trabajadas por los integrantes

En el presente trabajo, todos los integrantes trabajaron por igual, tanto en la elaboración del informe como en la implementación del código en Haskell y en Python. Con respecto a la redacción del informe, se trabajó en la plataforma de Google Drive, es decir se trabajó en simultáneo. En cuanto a la interfaz, esta se fue implementando gradualmente en las reuniones grupales.

Código en Haskell:

| | |
|------------------------------------|----------|
| Campana Yauri, Dan-el Mauricio | 20125121 |
| Palacios Coronado, Antony | 20141735 |
| Perez Cabrera, Estephany Elizabeth | 20142604 |

Informe:

| | |
|------------------------------------|----------|
| Campana Yauri, Dan-el Mauricio | 20125121 |
| Palacios Coronado, Antony | 20141735 |
| Perez Cabrera, Estephany Elizabeth | 20142604 |

Métodos Python:

| | |
|------------------------------------|----------|
| Campana Yauri, Dan-el Mauricio | 20125121 |
| Palacios Coronado, Antony | 20141735 |
| Perez Cabrera, Estephany Elizabeth | 20142604 |

Interfaz en Python:

| | |
|------------------------------------|----------|
| Campana Yauri, Dan-el Mauricio | 20125121 |
| Palacios Coronado, Antony | 20141735 |
| Perez Cabrera, Estephany Elizabeth | 20142604 |

Comentarios

Al ser este un lenguaje de programación nuevo para algunos miembros del grupo, se tuvo que buscar y aprender conceptos básicos, así como la sintaxis de haskell.

Se tuvo que usar un tipo de dato definido por el usuario, el cual no debía ser hecho de la forma clásica, sino de una forma más inferida. Esto trajo algunos contratiempos porque se tuvo que investigar otras formas de hacerlo.

Usando lo definido en el EDMA, algunas funciones no compilaban. Esto debido a que se estaban pasando valores de otro tipo a la función.

Siendo haskell un lenguaje de programación muy estricto, los errores de compilación abundaban pero con practica se pudieron reducir.

Al momento de crear los archivos de Haskell para el enlace con Python no se estaba seguro si usar los tipos de datos o no, ya que para enviar los datos a los archivos de texto necesitaban ser tipo de dato String y los tipos de datos definidos por el usuario no eran de este tipo. Lo ideal sería haber implementado el TAD Pilas en forma de cadena String, pero por motivos de uso TAD se tuvo que crear el tipo de dato Stack. Es decir, convertir de [String] a Stack y nuevamente de Stack a [String] para poder guardarlo en el archivo de texto.

Al momento de crear la interface se tuvo problemas ya que los valores para operar debían de pasarse de Int a Str y de Str a Int. Esto debido a que siempre se leía en formato String del archivo de texto para el enlace.

Bibliografía

Marti Ortega, Verdejo

Estructura de datos y métodos algorítmicos (EDMA). Consulta 1 de Abril de 2018

Rance D. Necaise

Data Structures and Algorithms Using Python (DSAUP). Consulta 3 de Abril de 2018

Jet Brains

Download PyCharm. Consulta 3 de Abril de 2018

<https://www.jetbrains.com/pycharm/download/#section=linux>

WIKIBOOKS

Data Structures/Stacks and Queues. Consulta 3 de Abril de 2018

https://en.wikibooks.org/wiki/Data_Structures/Stacks_and_Queues

Aprende Haskell

¡Aprende Haskell por el bien de todos! Consulta 17 de Abril de 2018

<http://aprendehaskell.es/content/ClasesDeTipos.html>

Anexos

Código en Haskell:

```
----- TIPO DE DATO PILA-----

data Stack x = P x (Stack x)
              | Vacia

desapilar :: Stack x -> Stack x
desapilar Vacia = error "ERROR"
desapilar (P _ p) = p

cima :: Stack x -> x
cima Vacia = error "ERROR"
cima (P x _) = x

esta_vacia :: Stack x -> Bool
esta_vacia Vacia = True
esta_vacia _ = False

profundidad :: Stack x -> Int
profundidad Vacia = 0
profundidad (P x xs) = 1 + profundidad (xs)

fondo :: Stack x -> x
fondo Vacia = error "ERROR"
fondo (P x Vacia) = x
fondo (P x xs) = fondo(xs)

apilar_inversa :: Stack x -> Stack x -> Stack x
apilar_inversa (Vacia) (P y ys) = (P y ys)
apilar_inversa (P x xs) ys = apilar_inversa (xs) (P x ys)

inversa :: Stack x -> Stack x
inversa Vacia = Vacia
inversa xs = apilar_inversa (xs) (Vacia)

duplicar :: Stack x -> Stack x
duplicar Vacia = Vacia
duplicar (P x xs) = (P x (P x (duplicar(xs))))

concatenar :: Stack x -> Stack x -> Stack x
concatenar xs Vacia = xs
concatenar xs (P y ys) = (P y (concatenar xs ys))

entremezclar :: Stack x -> Stack x -> Stack x
entremezclar Vacia Vacia = Vacia
entremezclar (P x xs) Vacia = (P x xs)
entremezclar Vacia (P y ys) = (P y ys)
entremezclar (P x xs) (P y ys)
    | profundidad(xs) + 1 > profundidad(ys) = (P x (entremezclar (xs) (P y
ys) ))
    | profundidad(xs) <= profundidad(ys) + 1 = (P y (entremezclar (P x
xs) (ys) ))

-----TIPO DE DATO SECUENCIA -----
```

```

crear :: Stack x -> Stack x -> Stack x
crear xs ys = apilar_inversa xs ys

insertar :: Stack x -> Stack x -> x -> Stack x
insertar xs ys e = apilar_inversa (P e xs) (ys)

eliminar :: Stack x -> Stack x -> Stack x
eliminar xs Vacia = error "ERROR"
eliminar xs (P y ys) = apilar_inversa xs ys

actual :: Stack x -> Stack x -> x
actual xs Vacia = error "ERROR"
actual xs (P y _) = y

avanzar :: Stack x -> Stack x -> Stack x
avanzar xs Vacia = error "ERROR"
avanzar xs (P y ys) = apilar_inversa (P y xs) ys

reiniciar :: Stack x -> Stack x -> Stack x
reiniciar Vacia xs = xs
reiniciar (P x xs) (ys) = reiniciar (xs) (P x ys)

fin :: Stack x -> Stack x -> Bool
fin _ Vacia = True
fin _ _ = False

es_sec_vacia :: Stack x -> Stack x -> Bool
es_sec_vacia Vacia Vacia = True
es_sec_vacia _ _ = False

----- IMPRIMIR CUALQUIER STACK -----

imprimir :: Stack x -> [x]
imprimir Vacia = []
imprimir (P x xs) = [x] ++ imprimir(xs)

```

Interfaz en Python:

Código de Haskell para enlace:

```
import System.IO
import Control.Monad

main = do
    pila <- readFile "numeros.txt"
    let listaString = words pila
    print listaString

    elem <- readFile "elem.txt"
    writeFile "numeros.txt" $ ( unwords $ (aString(apilar((words elem) ++
listaString))) )
    print ( unwords $ (aString(apilar((words elem) ++ listaString))) )

data Stack x = P String (Stack x)
              | Vacia

apilar :: [String] -> Stack x
apilar [] = Vacia
apilar (x:xs) = P x (apilar xs)

aString :: Stack x -> [String]
aString Vacia = []
aString (P x xs) = ([x] ++ aString(xs))
```

```
import System.IO
import Control.Monad

main = do
    pila <- readFile "symbol.txt"
    let listaString = words pila
    print listaString

    elem <- readFile "elem.txt"
    writeFile "symbol.txt" $ ( unwords $ (aString(apilar((words elem) ++
listaString))) )
    print ( unwords $ (aString(apilar((words elem) ++ listaString))) )

data Stack x = P String (Stack x)
              | Vacia

apilar :: [String] -> Stack x
apilar [] = Vacia
apilar (x:xs) = P x (apilar xs)

aString :: Stack x -> [String]
aString Vacia = []
aString (P x xs) = ([x] ++ aString(xs))
```

```
import System.IO
import Control.Monad

main = do
    pila <- readFile "numeros.txt"
    let listaString = words pila
```

```

        print listaString

        writeFile "numeros.txt" $ ( unwords $
(aString(desapilar(listaString))) )
        print ( unwords $ (aString(desapilar(listaString))) )

data Stack x = P String (Stack x)
              | Vacia

apilar :: [String] -> Stack x
apilar [] = Vacia
apilar (x:xs) = P x (apilar xs)

desapilar :: [String] -> Stack x
desapilar [] = Vacia
desapilar (x:xs) = apilar(xs)

aString :: Stack x -> [String]
aString Vacia = []
aString (P x xs) = ([x] ++ aString(xs))

```

```

import System.IO
import Control.Monad

main = do
    pila <- readFile "symbol.txt"
    let listaString = words pila
    print listaString

    writeFile "symbol.txt" $ ( unwords $
(aString(desapilar(listaString))) )
    print ( unwords $ (aString(desapilar(listaString))) )

data Stack x = P String (Stack x)
              | Vacia

apilar :: [String] -> Stack x
apilar [] = Vacia
apilar (x:xs) = P x (apilar xs)

desapilar :: [String] -> Stack x
desapilar [] = Vacia
desapilar (x:xs) = apilar(xs)

aString :: Stack x -> [String]
aString Vacia = []
aString (P x xs) = ([x] ++ aString(xs))

```

```

import System.IO
import Control.Monad

main = do
    pila <- readFile "symbol.txt"
    let listaString = words pila
    print listaString

    writeFile "cima.txt" $ ( unwords $ (cima(apilar listaString)) )
    print ( unwords $ (cima(apilar listaString)) )

```

```

data Stack x = P String (Stack x)
              | Vacia

apilar :: [String] -> Stack x
apilar [] = Vacia
apilar (x:xs) = P x (apilar xs)

cima :: Stack x -> [String]
cima Vacia = error "ERROR"
cima (P x _) = [x]

```

Código de Python para Interface:

```

import tkinter
from tkinter import *
import os
#####-----CALCULADORA-----#####
class CalculadoraHS():
    def __init__(self):
        result = list()
        self.symbol = self.numero = result
        self.resultado = None
    def estaVacía(self):
        if self.numero==[] and self.symbol==[]:
            return True
        else:
            return False
    def apilar(self, item, nombre):
        # Guardar elemento en el texto elem.txt
        with open('elem.txt', "w") as elementoTxt:
            elementoTxt.write(str(item))
            elementoTxt.close()
        if nombre == "N":
            # El elemento ya está guardado en el texto
            # Procedemos a apilar el elemento en el texto con haskell
            os.system("./apilarNumero")
            # Colocar elementos de un txt a una lista
            with open('numeros.txt') as inputfile:
                for line in inputfile:
                    numero = list((line.strip().split(' ')))
            self.numero = numero
            return self.numero
        if nombre == "S":
            # El elemento ya está guardado en el texto
            # Procedemos a apilar el elemento en el texto con haskell
            os.system("./apilarSymbol")
            # Colocar elementos de un txt a una lista
            with open('symbol.txt') as inputfile:
                for line in inputfile:
                    symbol = list((line.strip().split(' ')))
            self.symbol = symbol
            return self.symbol
    def desapilar(self, nombre):
        #En este caso usaremos el apilar para devolvernos
        el valor que hemos quitado
        assert not self.estaVacía(), "NO HAY ELEMENTOS A OPERAR"
        if nombre == "N":
            item = self.numero[0]
            os.system("./desapilarNumero")
            if (os.stat("numeros.txt").st_size == 0):
                numero = []
            else:
                with open('numeros.txt') as inputfile:
                    for line in inputfile:
                        numero = list((line.strip().split(' ')))
            self.numero = numero
            return item
        if nombre == "S":
            item = self.symbol[0]

```



```

os.system("./desapilarSymbol")
if(os.stat("symbol.txt").st_size == 0):
    symbol = []
else:
    with open('symbol.txt') as inputfile:
        for line in inputfile:
            symbol = list((line.strip().split(' ')))
    self.symbol = symbol
    return item
def cima(self):
    assert not self.estaVacia(), "NO HAY ELEMENTOS A OPERAR"
    os.system("./cima")
    with open('cima.txt') as inputfile:
        for line in inputfile:
            cima = line
    return cima
def calcular(self, cadena):
    for i in range(0, len(cadena)):
        c = self.convertir(cadena[i])
        if(c[1]==0):
            self.apilar(cadena[i], "S")
            continue
        elif(c[1]==2):
            #Si es un dígito
            n = cadena[i-1]
            if(48<= ord(n) <= 57):
                n = int(self.desapilar("N"))* 10 + int(c[0])
                self.apilar(str(n), "N")
            else:
                self.apilar(c[0], "N")
            continue
        elif(c[1]==3):
            #Si es un operador
            if (self.cima() == "("):
                self.apilar(c[0], "S")
            elif (self.prioridad(self.cima(), c[0]) == True):
                op2 = self.desapilar("N")
                op = self.desapilar("S")
                op1 = self.desapilar("N")
                self.resultado = self.operar(op1, op, op2)
                with open('elem.txt', "w") as elementoTxt:
                    elementoTxt.write(str(self.resultado))
                    elementoTxt.close()
                self.apilar(str(self.resultado), "N")
                with open('elem.txt', "w") as elementoTxt:
                    elementoTxt.write(c[0])
                    elementoTxt.close()
                self.apilar(c[0], "S")
            else:
                with open('elem.txt', "w") as elementoTxt:
                    elementoTxt.write(c[0])
                    elementoTxt.close()
                self.apilar(c[0], "S")
        elif(c[1]==1):
            #Cierra paréntesis
            while(1):
                op = self.desapilar("S")
                if(op == "("):
                    break
                else:
                    op2 = self.desapilar("N")
                    op1 = self.desapilar("N")
                    self.resultado = self.operar(op1, op, op2)
                    self.apilar(str(self.resultado), "N")
            return self.resultado
def prioridad(self, p1, p2):
    #Si la prioridad es mayor o igual
    if((p1=="*" or p1=="/") and (p2=="-" or p2=="+")) or (p1==p2):
        return True
    elif((p1=="*" and p2=="/") or (p1=="/" and p2=="*")):
        return True
    return False
def operar(self, op1, op, op2):
    if (op=="*"):
        return int(op1)*int(op2)
    if (op == "/"):
        return int(op1) // int(op2)

```

```

        if (op == "+"):
            return int(op1) + int(op2)
        if (op == "-"):
            return int(op1) - int(op2)
    def convertir(self, item):
        if(item == "("):
            return item, 0                #Abre paréntesis
        elif(item == ")"):
            return item, 1                #Cierra paréntesis
        elif(48<= ord(item) <= 57):
            return ord(item)-48, 2        #Si es un dígito
        else:
            return item, 3                #Si es un operador
#####-----INTERFACE-----#####
def set_text(num):
    if(num=="CE"):
        e.delete(0, END)
    elif(num=="EQUAL"):
        datos = e.get()
        C = CalculadoraHS()
        C.symbol = []
        C.numero = []
        resultado = C.calcular("(" + datos + ")")
        mLabel.config(text = resultado)
    else:
        e.insert(END, num)
    return
window = Tk()
window.geometry("250x200")
window.title("Calculadora")
e = Entry(window)
e.grid(row = 1, columnspan=4)
datos = StringVar()
b_0 = Button(window, text = "0", anchor="w", width=1, command = lambda:set_text("0"))
b_0.grid(row=7, column=2)
b_1 = Button(window, text = "1", anchor="w", width=1, command = lambda:set_text("1"))
b_1.grid(row=6, column=1)
b_2 = Button(window, text = "2", anchor="w", width=1, command = lambda:set_text("2"))
b_2.grid(row=6, column=2)
b_3 = Button(window, text = "3", anchor="w", width=1, command = lambda:set_text("3"))
b_3.grid(row=6, column=3)
b_4 = Button(window, text = "4", anchor="w", width=1, command = lambda:set_text("4"))
b_4.grid(row=5, column=1)
b_5 = Button(window, text = "5", anchor="w", width=1, command = lambda:set_text("5"))
b_5.grid(row=5, column=2)
b_6 = Button(window, text = "6", anchor="w", width=1, command = lambda:set_text("6"))
b_6.grid(row=5, column=3)
b_7 = Button(window, text = "7", anchor="w", width=1, command = lambda:set_text("7"))
b_7.grid(row=4, column=1)
b_8 = Button(window, text = "8", anchor="w", width=1, command = lambda:set_text("8"))
b_8.grid(row=4, column=2)
b_9 = Button(window, text = "9", anchor="w", width=1, command = lambda:set_text("9"))
b_9.grid(row=4, column=3)
b_clear = Button(window, text = "CE", anchor=CENTER, width=2,height=8, command =
lambda:set_text("CE"))
b_clear.grid(row=4, column=5,rowspan=4)
b_mas = Button(window, text = "+", anchor=CENTER, width=2, command = lambda:set_text("+"))
b_mas.grid(row=4, column=4)
b_menos = Button(window, text = "-", anchor=CENTER, width=2, command = lambda:set_text("-"))
b_menos.grid(row=5, column=4)
b_multi = Button(window, text = "*", anchor=CENTER, width=2, command = lambda:set_text("*"))
b_multi.grid(row=6, column=4)
b_divi = Button(window, text = "/", anchor=CENTER, width=2, command = lambda:set_text("/"))
b_divi.grid(row=7, column=4)
b_oBra = Button(window, text = "(", anchor=CENTER, width=2, command = lambda:set_text("("))
b_oBra.grid(row=7, column=1)
b_cBra = Button(window, text = ")", anchor=CENTER, width=2, command = lambda:set_text("))"))
b_cBra.grid(row=7, column=3)
b_igual = Button(window, text = "=", anchor=CENTER, width=30, command =
lambda:set_text("EQUAL"))
b_igual.grid(row=8, columnspan=15)
mLabel = Label(window, anchor = CENTER)
mLabel.grid(row = 9, column = 3)
if __name__ == "__main__":
    window.mainloop()

```