Instituto Tecnológico de Costa Rica **Ingeniería en Computación** **Sede Regional San Carlos**	Primer Tarea Programada Lenguajes de Programación
Prof. Oscar Víquez Acuña.	Micro intérprete de bytecode de Python

Descripción:

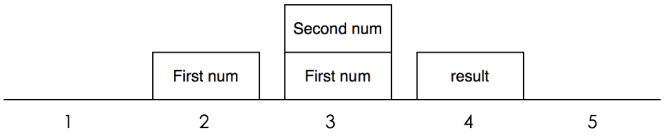
El proceso de compilación y ejecución de programas del lenguaje Python se divide en varias etapas. Inicialmente la compilación del lenguaje procesa detalles sintácticos y algunos semánticos para cierto código fuente de entrada y posteriormente genera el bytecode equivalente que será procesado por el intérprete de Python (CPython) para su ejecución. Dicho intérprete es una aplicación compleja que realiza la corrida del código de bajo nivel y en el caso de encontrar posibles errores los reporta al usuario y detiene la eventual corrida. Además es un intérprete tipo REPL en el que se puede ejecutar instrucciones por lote o en secuencia.

En este contexto, el estudiante realizará la implementación de un Micro intérprete de Python para la ejecución de un subconjunto de instrucciones originales del lenguaje con algunas variantes que limitan dicho funcionamiento a una simple fracción de lo que realmente ocurre. Cabe destacar que para el mejor entendimiento del funcionamiento de estas instrucciones, se recomienda generar programas pequeños en Python, desensamblar el código bytecode generado y observar su comportamiento.

Inicialmente se deben contemplar, para efectos del funcionamiento del micro intérprete, una serie de estructuras básicas necesarias para la ejecución de las instrucciones.

1. Todo programa en Python estará compuesto de variables las cuales para efectos de su representación en el intérprete, deberán ser administradas desde alguna estructura de datos que permita el manejo del nombre de la variable, el tipo de la misma y el valor contenido por esta. Dado que Python no define tipos de datos y que la implementación de este micro intérprete se realizará en GO, se debe contemplar alguna estrategia para "mapear" dichos tipos de datos en el almacén. Inicialmente la estructura debe soportar variables de tipo entero, carácter, string, flotante y listas y para cada una de ellas guardar lo necesario según el criterio del programador.

2. El procesamiento de las instrucciones en el micro intérprete necesita de una o varias pilas de ejecución que soporten los valores según los tipos de datos descritos en el punto anterior. Quiere decir que este intérprete es una máquina de pila y por lo tanto todas las instrucciones harán uso de la(s) misma(s) para realizar sus procesamientos. El siguiente dibujo representa el uso de una pila para la ejecución de una serie de instrucciones para realizar una suma, desde la carga de los operandos, hasta la operación y el almacenamiento en la pila del resultado de la misma.



La siguiente es la lista de instrucciones que se deberán programar:

INSTRUCCIÓN	PARÁMETROS	DESCRIPCIÓN	DEBE HABER EN LA PILA	QUÉ DEJA EN LA PILA	Observaciones
LOAD_CONST	*const*	Coloca el valor de la constante en el tope de la pila	[]	[const]	Cualquier constante de cualquier tipo de dato permitido
LOAD_FAST	*varname*	Coloca el valor del contenido de la variable en la pila			
STORE_FAST	*varname*	Escribe el contenido del tope de la pila en la variable	[const]	[]	
LOAD_GLOBAL	*name*	Carga en el tope de la pila o el valor de la variable o la referencia a la función	[]	[const | ref]	Nótese que el parámetro no dice varname, sino name porque globales serán tanto las variables como las funciones.
CALL_FUNCTION	*numparams*	Realiza un salto a la dirección de código de la función	[…params…, funcref]		Nótese que ..params… son varias constantes que se pueden encontrar en la pila a partir del tope seguidas por la referencia a la función
COMPARE_OP	*op*	Realiza una comparación booleana según el op que reciba	[oper2, oper1]	[result]	El resultado es algo que represente o un TRUE o un FALSE
BINARY_SUBSTRACT		Realiza una resta de operandos	[oper2, [oper1]	[result]	
BINARY_ADD		Realiza una suma de operandos	[oper2, oper1]	[result]	
BINARY_MULTIPLY		Realiza una multiplicación de operandos	[oper2, oper1]	[result]	
BINARY_DIVIDE		Realiza una división entera de operandos	[oper2, oper1]	[result]	
BINARY_AND		Realiza un AND lógico	[oper2, oper1]	[result]	
BINARY_OR		Realiza un OR	[oper2,	[result]	

		lógico	oper1]		
BINARY_MODULO		Realiza el cálculo del cociente de la división de dos operandos	[oper2, oper1]	[result]	
STORE_SUBSCR		Realiza la operación: array[index] = value	[index, array, value]	[]	
BINARY_SUBSCR		Carga en el tope de la pila el elemento de un arreglo en el índice indicado	[index,array]	[array[index]]	
JUMP_ABSOLUTE	*target*	Salta a la línea de código indicada por "target"	[]	[]	"target" es un índice indicado en el archivo que debe coincidir con el # de instrucción una vez almacenado el código en memoria
JUMP_IF_TRUE	*target*	Si el tope de la pila es True, salta a "target"	[valueTF]	[]	"target" es un índice indicado en el archivo que debe coincidir con el # de instrucción una vez almacenado el código en memoria
JUMP_IF_FALSE	*target*	Si el tope de la pila es False, salta a "target"	[valueTF]	[]	"target" es un índice indicado en el archivo que debe coincidir con el # de instrucción una vez almacenado el código en memoria
BUILD_LIST	*elements*	Construye una lista con "elements" cantidad de elementos	[elem1.. elemN]	[lista]	Considerar lista como un slice con uso o no de punteros
END		Termina el programa	[]	[]	Última instrucción del código.

El proyecto debe permitir leer archivos de entrada con instrucciones en bytecode entendibles por el Micro intérprete de manera que al finalizar la corrida pueda mostrarse la ejecución del mismo.

El archivo tendrá un formato específico definido por el programador que podría contener:

1. Consecutivo incremental para cada una de las instrucciones
2. Cada instrucción tendrá o no solo un parámetro
3. Posibilidad de separar con TABS o más de un espacio en blanco

PARA EFECTOS DE PODER MOSTRAR UNA SALIDA DE LOS PROGRAMAS, EL CALL FUNCTION DEBE PERMITIR SOLAMENTE LLAMAR LA FUNCIÓN print AUNQUE NO HAYA SIDO DECLARADA O NO SE PERMITA DECLARAR MÉTODOS EN ESTE INTÉRPRETE.

Puntos claves para la funcionalidad del programa y que serán tomados en cuenta en la evaluación:

- Manejo de memoria para las estructuras
- Manejo de archivos de entrada de los programas. Debe adjuntar las pruebas suficientes para que se puedan probar todas las instrucciones del intérprete
- Funcionamiento del intérprete
- Salida de datos con print.
- Modularidad en la programación del trabajo de manera que exista suficiente separación de conceptos que permita la escalabilidad del programa
- Documentación efectiva al comunicar el mensaje

Documentación

Se solicita como documentación la grabación de un video explicativo de su proyecto que comprima los siguientes apartados en un periodo de tiempo **no mayor a 5 minutos**:

- Explicación general del problema
- Explicación de las estructuras internas utilizadas y la estrategia de ejecución en general
- La explicación de la funcionalidad lograda en término de la ejecución del programa para un ejemplo diferente al aportado por el profesor en este enunciado

Notas Finales para la Tarea:

- Tarea Individual
- Fecha de entrega aproximada para ambas tareas: viernes 31 de marzo del 2023 antes de las 10:00 pm.
- Se recomienda que se empiece a trabajar desde hoy.
- Cualquier tipo de fraude será severamente castigado.
- La entrega del trabajo se hará a través del Tec-Digital.
- El trabajo debe ser implementado en el lenguaje GO.

Referencias para el trabajo:

- https://docs.python.org/3.3/library/dis.html
- http://cs263-technology-tutorial.readthedocs.io/en/latest/
- http://akaptur.com/blog/2013/08/14/python-bytecode-fun-with-dis/

Ejemplo de bytecode de Python:

El siguiente es un ejemplo en Python de un código desensamblado. Notar que el comportamiento de las instrucciones que viene en el código debe ser como se indica en el proyecto y no como lo hace el intérprete de Python.

Para desensamblar código en Python, este debe estar circunscrito a un método, pero para efectos del bytecode que se escriba como ejemplos de entrada en el proyecto, no deberemos escribir nada relacionado con declaración de métodos u etiquetas para dicho fin.

```
def prueba():
    x = 0
    lista = [0,1,2,3,4,5,6,7,8,9]
    while (x < 10):
        if(x%2==0):
            print(lista[x])
        x = x + 1

prueba()
```

```
>>> dis.dis(prueba)
 5           0 LOAD_CONST               1 (0)
             2 STORE_FAST               0 (x)

 6           4 BUILD_LIST               0
             6 LOAD_CONST               2 ((0, 1, 2, 3, 4, 5, 6, 7, 8, 9))
             8 LIST_EXTEND              1
            10 STORE_FAST               1 (lista)

 7          12 LOAD_FAST                0 (x)
            14 LOAD_CONST               3 (10)
            16 COMPARE_OP               0 (<)
            18 POP_JUMP_IF_FALSE       32 (to 64)

 8     >>   20 LOAD_FAST                0 (x)
            22 LOAD_CONST               4 (2)
            24 BINARY_MODULO
            26 LOAD_CONST               1 (0)
            28 COMPARE_OP               2 (==)
            30 POP_JUMP_IF_FALSE       22 (to 44)

 9          32 LOAD_GLOBAL              0 (print)
            34 LOAD_FAST                1 (lista)
            36 LOAD_FAST                0 (x)
            38 BINARY_SUBSCR
            40 CALL_FUNCTION            1
            42 POP_TOP

10     >>   44 LOAD_FAST                0 (x)
            46 LOAD_CONST               5 (1)
            48 BINARY_ADD
            50 STORE_FAST               0 (x)

 7          52 LOAD_FAST                0 (x)
            54 LOAD_CONST               3 (10)
```

```
          56 COMPARE_OP               0 (<)
          58 POP_JUMP_IF_TRUE        10 (to 20)
          60 LOAD_CONST               0 (None)
          62 RETURN_VALUE
     >>   64 LOAD_CONST               0 (None)
          66 RETURN_VALUE
```

Posible adaptación de bytecode para el proyecto (archivo de entrada como prueba):

```
0  LOAD_CONST          0
1  STORE_FAST          x
2  LOAD_CONST          0
3  LOAD_CONST          1
4  LOAD_CONST          2
5  LOAD_CONST          3
6  LOAD_CONST          4
7  LOAD_CONST          5
8  LOAD_CONST          6
9  LOAD_CONST          7
10 LOAD_CONST          8
11 LOAD_CONST          9
12 BUILD_LIST          10
13 STORE_FAST          lista
14 LOAD_FAST           x
15 LOAD_CONST          10
16 COMPARE_OP          <
17 JUMP_IF_FALSE       37
18 LOAD_FAST           x
19 LOAD_CONST          2
20 BINARY_MODULO
21 LOAD_CONST          0
22 COMPARE_OP          ==
23 JUMP_IF_FALSE       29
24 LOAD_GLOBAL         print
25 LOAD_FAST           lista
26 LOAD_FAST           x
27 BINARY_SUBSCR
28 CALL_FUNCTION       1
29 LOAD_FAST           x
30 LOAD_CONST          1
31 BINARY_ADD
32 STORE_FAST          x
33 LOAD_FAST           x
34 LOAD_CONST          10
35 COMPARE_OP          <
36 JUMP_IF_TRUE        18
37 END
```

Commented [OMVA1]: Atención a la forma de imprimir que puede no ser la misma que escojan ustedes… otra opción según vimos en clase podría ser:

```
24 LOAD_FAST lista
25 LOAD_FAST x
26 BINARY_SUBSCR
27 CALL_PRINT 1
...
```

Posible bytecode equivalente al anterior:

```
0  LOAD_CONST          0
1  STORE_FAST          x
2  LOAD_CONST          0
```

```
3  LOAD_CONST        1
4  LOAD_CONST        2
5  LOAD_CONST        3
6  LOAD_CONST        4
7  LOAD_CONST        5
8  LOAD_CONST        6
9  LOAD_CONST        7
10 LOAD_CONST        8
11 LOAD_CONST        9
12 BUILD_LIST        10
13 STORE_FAST        lista
14 LOAD_FAST         x
15 LOAD_CONST        10
16 COMPARE_OP        <
17 JUMP_IF_FALSE     34
18 LOAD_FAST         x
19 LOAD_CONST        2
20 BINARY_MODULO
21 LOAD_CONST        0
22 COMPARE_OP        ==
23 JUMP_IF_FALSE     29
24 LOAD_GLOBAL       print
25 LOAD_FAST         lista
26 LOAD_FAST         x
27 BINARY_SUBSCR
28 CALL_FUNCTION     1
29 LOAD_FAST         x
30 LOAD_CONST        1
31 BINARY_ADD
32 STORE_FAST        x
33 JUMP_ABSOLUTE     14
34 END
```

Commented [OMVA2]: Atención a la forma de imprimir que puede no ser la misma que escojan ustedes… otra opción según vimos en clase podría ser:

```
24 LOAD_FAST lista
25 LOAD_FAST x
26 BINARY_SUBSCR
27 CALL_PRINT 1
...
```