Universidad Nacional Experimental Simón Bolívar
Lenguajes de Programación I
Prof. Ricardo Monascal

Caracas, 15 de febrero 2021

Alumno y Carné:
Victoria Torres 12-11468

Examen 1

Constantes: X=4; Y=6; Z=8

1)

(a) De una breve descripción del lenguaje escogido

Visual Basic .NET es un lenguaje de programación orientado a objetos, que se puede considerar una evolución de Visual Basic, implementada sobre el Framework .NET (el modelo de programación diseñado para simplificar la programación de aplicaciones en un entorno sumamente distribuido).

Este lenguaje es totalmente diferente a sus antecesores, permite crear aplicaciones de escritorio, web, móviles y brinda un completo número de características para hacer que el desarrollo de aplicaciones sea realmente rápido.

i. Diga qué tipo de alcances y asociaciones posee, argumentando las ventajas y desventajas de la decisión tomada por los diseñadores del lenguaje, en el contexto de sus usuarios objetivos.

VB.NET es básicamente un lenguaje compilado.

Para mantener eficacia en el desarrollo de las aplicaciones, la mayoría de programadores de VB.NET utilizan el IDE Microsof Visual Studio y éste genera código CIL (Common Intermediate Language) que es un lenguaje ensamblador orientado a objetos basado en pilas.

De esta manera dentro de este lenguaje las asociaciones se hacen de manera temprana (estática) antes de la ejecución del programa como tal. Sin embargo, esto no es tan definitivo como se podría creer ya que el código CIL no es 100% compilado. No puedes ejecutarlo, así como está (intermedio) en un sistema operativo, por lo cual éste es, a su vez, compilado a código de máquina por el CLR (Common Language Runtime).

Luego, en tiempo de ejecución, el compilador del CLR convierte el código CIL en código nativo para el sistema operativo. Pero a diferencia del intérprete, se hace la compilación completa en lugar de ejecutar una instrucción a la vez, por lo que no existe la misma sobrecarga, no hay una máquina virtual ejecutando el programa.

Además, el CLR realiza compilación JIT (Just-in-Time) a medida que el programa invoca métodos generando entonces el código máquina real que se ejecuta en la plataforma del cliente. Así, el

código ejecutable obtenido se almacena en la memoria caché del ordenador, siendo recompilado de nuevo sólo en el caso de producirse algún cambio en el código fuente.

Cabe destacar que la máquina donde se ejecuta el código debe tener forzosamente el .NET Framework instalado porque si no, no podrá ser ejecutado.

Pero entonces, ¿podemos decir que VB.NET es un lenguaje compilado?

Pues una de las grandes ventajas que ofrece el IDE Visual Studio es, ser a la vez un compilador y un intérprete, tiene ambos componentes de manera que, en modo de depuración, es un intérprete y al ejecutar paso a paso se puede encontrar exactamente dónde falla el código, permitiendo también inspeccionar los valores de las variables y leer las excepciones.

Para que este comportamiento sea posible, VB.NET usa los archivos ".pdb", que son pequeñas bases de datos que guardan los valores que se están usando al ejecutar el programa (variables, objetos, etc).

Por lo cual el lenguaje Visual Basic .NET, si bien teóricamente es compilado, tiene la posibilidad de disfrutar, en cierta manera, lo mejor de los dos mundos.

Y esto puede verse en cosas que permite hacer el lenguaje. Por ejemplo, en las guías del lenguaje en la página de Microsoft (https://docs.microsoft.com/es-es/dotnet/visual-basic/), podemos encontrar que, si se tiene activo el modo "Option Strict", entonces las variables deben ser declaradas al inicio con sus tipos y en tiempo de compilación las asociaciones se establecen de manera estática.

Sin embargo, en este lenguaje existen variables "Objeto" (Object), que se pueden utilizar cuando se desconoce la clase específica de cierta variable hasta el momento de la ejecución del código, y con ellas se crea una referencia general a cualquier tipo de objeto hasta que la clase específica se asigne en tiempo de ejecución (para poder hacer esto se activa la opción "Option Strict Off"). Esto significa que, para estos casos, esa asociación se realiza en tiempo de ejecución, es decir, de manera tardía o dinámica.

Por supuesto, cuando esto ocurre, se necesitará tiempo de ejecución adicional y hasta se podrían producir errores en tiempo de ejecución si el código intenta obtener acceso a miembros de una clase diferente, pero el hecho es que en este lenguaje también se hacen presente este tipo de asociaciones y con esto confirmamos que se pueden tener casos de ambos tipos.

Por otra parte, en cuanto al alcance:

Visual Basic .NET, normalmente define el alcance de sus variables y expresiones en tiempo de ejecución (alcance estático), entonces por ejemplo para las variables el alcance se determinará en perspectiva según el lugar en el que se declara.

Sin embargo, vimos anteriormente que también es posible en el lenguaje tener expresiones tipo "Object" por lo cual el procesamiento de la expresión se puede diferir hasta el tiempo de ejecución. De manera que, en estos casos, el alcance se resuelve en tiempo de ejecución real de la expresión. Por lo tanto, puede tenerse alcance dinámico en estas situaciones.

ii. Diga qué tipo de módulos ofrece (de tenerlos) y las diferentes formas de importar y exportar nombres.

Visual Basic proporciona varios módulos que permiten simplificar las tareas comunes del código, como manipular cadenas, realizar cálculos matemáticos, obtener información del sistema, realizar operaciones de archivos y directorios, realizar conversiones de números a otras bases, entre otros.

Los módulos también tienen niveles de acceso que se pueden ajustar desde los más conocidos como "Públic", "Protected" y "Private", hasta otros menos convencionales como "Friend".

Para importar y exportar nombres, se puede utilizar la instrucción "Imports", por ejemplo, para importar un módulo que controla la visibilidad de los espacios de nombres de un ensamblado particular. Su sintaxis es:

"Imports [Aliasname =] Namespace"

Donde "Aliasname" hace referencia a un nombre corto que puede usarse en el código para hacer referencia a un espacio de nombres importado y "Namespace" es, tal cual, un espacio de nombres disponible a través de una referencia de proyecto, a través de una definición dentro del proyecto o a través de un "Imports" anterior.

También se podría utilizar de la siguiente manera:

"Imports [Aliasname =] namespace.element"

Donde "element" sería el nombre de un elemento de programación declarado en el espacio de nombres al que refiere "namespace".

iii. Enumere y explique las estructuras de control de flujo que ofrece.

Visual Basic .Net divide en 4 categorías sus estructuras de control de flujos:

1. Estructuras de Decisión:
   En estas podemos conseguir:
   a. Estructuras de Selección: Para decidir el fujo de instrucciones entre 2 o más alternativas. Encontrándose:
      - Simple: "If-then-else"
      - Múltiple: "Select… Case"

b. Excepciones y Especulaciones: Con “Try… Catch… Finally”, donde se permite ejecutar un conjunto de instrucciones en un entorno en el que se conserva el control si una de éstas produce una excepción, es decir, si algo sale mal se puede hacer algo al respecto, pero si todo sale bien entonces el flujo continúa normalmente.

2. Estructuras de bucle: Son estructuras de repetición donde literalmente se busca repetir un fragmento de código de forma:
    a. Determinada: Donde esta previamente calculada la cantidad de iteraciones. En estas encontramos bucles “for” y bucles “for each”.

    b. Indeterminada: Donde la cantidad de iteraciones depende de una condición. En estas encontramos bucles “while” y bucles “do… Loop”.

3. Estructuras de Control Adicionales: Donde VB.NET proporciona estructuras de control que ayudan a desechar un recurso o a reducir el número de veces que tiene que repetir una referencia de objeto. En estas se utilizan:
    a. “Using… End Using”, con la que se establece un bloque de instrucciones en el que se usa un recurso (como un identificador de archivo, un contenedor COM o una conexión SQL) y posteriormente al salir del bloque con “End Using” se desecha automáticamente el recurso para que pueda estar disponible para que lo use otro código (el recurso debe ser local y descartable).

    b. “With… End With”, con la que se permite especificar una sola referencia de objeto y, a continuación, ejecutar una serie de instrucciones que hacen referencia a dicho único objeto o estructura. Esto puede ayudar a que se simplifique el código y se mejore el rendimiento, ya que no se tendría que volver a establecer la referencia para cada instrucción que tiene acceso a ella.

4. Estructuras de Control Anidadas: Se permite el anidamiento entre instrucciones de control en tantos niveles como se desee, por ejemplo, colocar un “If-Then-Else” dentro de un “For… Next”. Es importante hacer énfasis en que, para anidar las estructuras, la más interna deben estar completamente contenida en la más externa puesto que no se admiten “superposiciones”.

A parte de estas 4 divisiones principales, también se pueden mencionar que VB.NET utiliza,

5. Estructuras de Abstracción Procedural: Éstas son básicamente procedimientos, funciones, subrutinas, entre otras (Call, function, Property, Sub).

iv. <u>Diga en qué orden evalúan expresiones y funciones.</u>

Cuando se producen varias operaciones en una expresión, cada parte se evalúa y se resuelve en un orden predeterminado según la precedencia de los operadores siguiendo estas reglas:

1. Los operadores aritméticos y de concatenación tienen mayor prioridad que los operadores de comparación, lógicos y bit a bit. Se evalúan, de mayor prioridad a menor, como sigue:

   Exponenciación (^)

   Identidad unaria y negación (+ ,–)

   Multiplicación y división de punto flotante (*,/)

   División de enteros (\)

   Aritmética modular (Mod)

   Suma y resta (+,–)

   Concatenación de cadenas (&)

   Desplazamiento de bits aritmético (<< , >>)

2. Todos los operadores de comparación tienen la misma prioridad y todos tienen mayor prioridad que los operadores lógicos y bit a bit, pero tienen menor prioridad que los operadores aritméticos y de concatenación (= , <> , < , <= , > , >= , Is , IsNot , Like , TypeOf ... Is).

3. Los operadores lógicos y bit a bit, tienen menor prioridad que los operadores aritméticos, de concatenación y de comparación y se evalúan en el siguiente orden:

   Negación (Not)

   Conjunción (And , AndAlso)

   Disyunción inclusiva (Or , OrElse)

   Disyunción exclusiva (Xor)

4. Los operadores con la misma prioridad se evalúan utilizando una asociatividad de izquierda a derecha, en el orden en que aparecen en la expresión.

Los operadores unarios usan la notación “prefija” y los operadores binarios la notación “infija”.

Además, es importante destacar que en Visual Basic.NET existen:

1. Tipos de Valor: Que contienen los datos dentro de su propia signación de memoria. Ej: Todos los tipos de datos numéricos, Boolean, Char, Date, etc.

2. Tipos de Referencia: Que almacenan una referencia a sus datos. Ej: String, todas las matrices (incluso si sus elementos son tipos de valor), tipos de clases como “From”, etc.

3. Elementos que no son Tipos: Son elementos que no se califican como alguno de los anteriores, por ejemplo: Módulos, Espacios de nombres, Eventos, etc.

4. Tipo “Object”: Se le puede asignar un tipo de referencia o un tipo de valor y dependiendo de cuál sea, se comportará almacenando los dotas o una referencia respectivamente.

Finalmente, otro aspecto que se puede destacar es que en este lenguaje se pueden pasar argumentos a procedimientos, tanto por valor como por referencia y se puede determinar en la declaración del procedimiento mismo utilizando la palabra clave “ByVal” o “ByRef”.

(b) Implemente los siguientes programas en el lenguaje escogido:

i. Dado un entero no–negativo n, calcular el factorial de n:

Archivo “factorial.vs” en el git, en la carpeta correspondiente a pregunta 1.

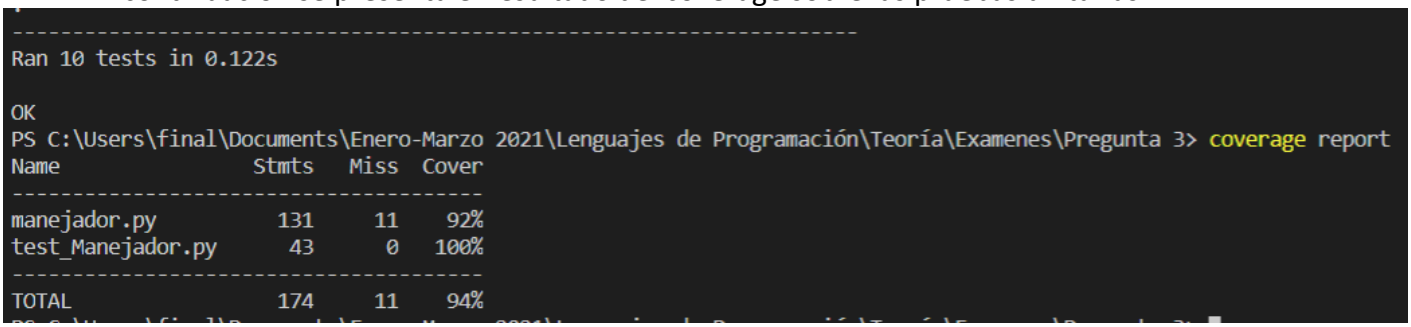
ii. Dadas dos matrices A y B (cuyas dimensiones son N x M y M x P, respectivamente), calcular su producto AxB (cuya dimensión es N x P):

Archivo “productoMatriz.vs” en carpeta de pregunta 1 en el git.

2) Respuesta en Documento PDF “Pregunta 2”, En la carpeta correspondiente a esta pregunta en el git.

3) Respuesta en carpeta de pregunta 3 del git.

A continuación se presenta el resultado del coverage sobre las pruebas unitarias:



```
----------------------------------------------------------------------
Ran 10 tests in 0.122s

OK
PS C:\Users\final\Documents\Enero-Marzo 2021\Lenguajes de Programación\Teoría\Examenes\Pregunta 3> coverage report
Name                Stmts   Miss  Cover
---------------------------------------
manejador.py          131     11    92%
test_Manejador.py      43      0   100%
---------------------------------------
TOTAL                 174     11    94%
```

4) Respuestas en carpeta de pregunta 4 del git.

a.i) Video subido como “4.a.i” al drive.

a.ii) En archivo de preg4 en el git

b.i) Lamentablemente no pude hacer el video a tiempo ☹

b.ii)

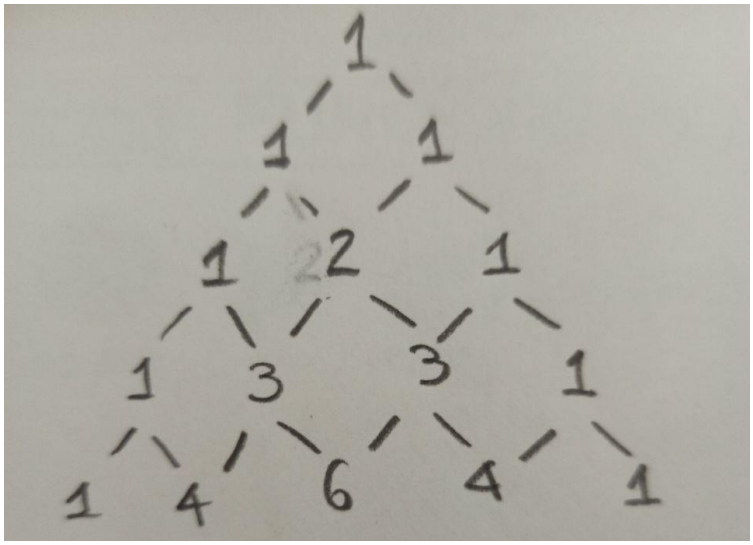
La subrutina Misterio, primero utiliza la instrucción yield para generar el valor que se le pasa como parámetro. Luego, inicializa un arreglo “acumulador” en el que va guardando todos los elementos que resultan de la llamada a la función “Zipwith”, con los parametros “[0,*p]”, “[*p,0]” (en los que el operador * extrae los valores de p, colocándolos en un array) y la función “x+y” (suma).

Una vez que zipWith va retornando valores para la “p” del ciclo, cada uno de estos valores los agrega en el acumulador “acum”, por clo cual este contendrá al final todos los elementos generados por zipWith.

Luego, cuando finaliza el ciclo y zipWith realizó todas las sumas entre los dos parámetros (“[0,*p]”, “[*p,0]”), se vuelve a llamar a misterio pero en esta ocasión se le pasa todo el acumulado, de manera que éste lo va devolviendo con yield para que al final pueda ser impreso.

Al final de todo, misterio está generándonos los coeficientes binomiales (la pirámide xd jajaja).

Y lo sospecho porque empieza a generar [1], [1,1], [1,2,1] y eso hace recordar esta conocida imagen:

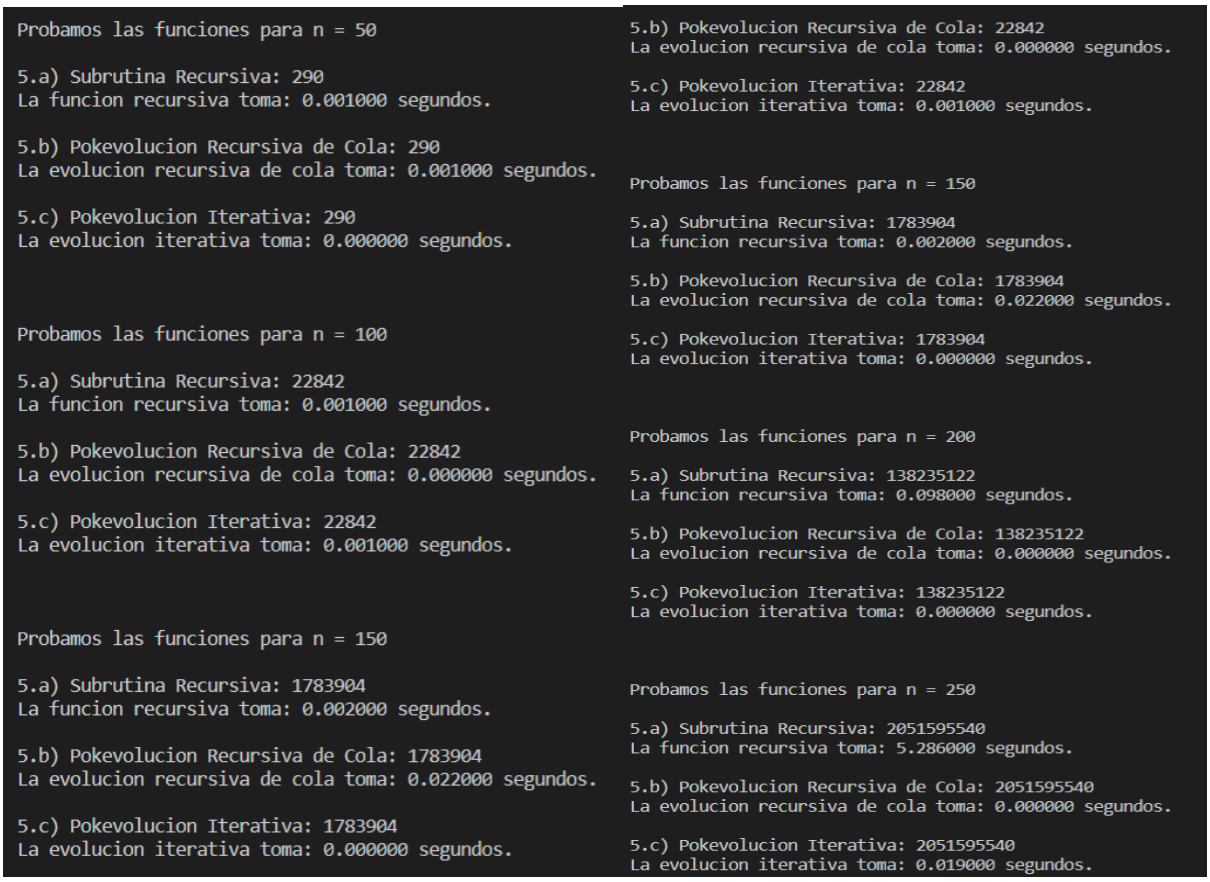


b.iii) En el siguiente enlace al git:

5) La a), b) y c), se encuentran en la carpeta correspondiente a la pregunta 5 en el git.

Para comparar las 3 implementaciones con distintos n, se utilizó la función clock() de la librería <time.h>, con la que se midió, para cada subrutina, el tiempo aproximado de CPU que trascurrió desde que se llamo hasta que se terminó de ejecutar.

A continuación, se pueden observar los resultados:



Puede observarse que en general la subrutina recursiva a medida que el n, se volvía más grande, tardaba mas en ejecutarse. De hecho, en el último caso para n = 250, fue la que más tardó en ejecutarse. En cambio, la subrutina recursiva de cola y la iterativa en general fueron más rápidas, a pesar de que con algunos “n” sus tiempos varían en milésimas de segundo, en general ambas se comportaban similar. Pero cabe destacar que hubo una oportunidad, en el caso n=150 en el cual la subrutina iterativa fue más rápida que la recursiva de cola, sin embargo no es tan critico en estas observaciones puesto que justo después para n = 200 ambas vuelven nuevamente a ser igual de rápidas.

6)

La implementación de esta pregunta puede encontrarse en la carpeta para la pregunta 6 en el git.

Adicionalmente, se presentan los resultados obtenidos al realizar el “coverage” sobre las pruebas unitarias: