Definiciones teóricas y anotaciones para el parcial

Practica 6

- **Parámetro:** es una forma de compartir datos entre diferentes unidades. Es la más flexible y permite la transferencia de diferentes datos en cada llamada. Proporciona ventajas en legibilidad y modificabilidad. Nos permiten compartir los datos en forma abstracta ya que indican con precisión qué es exactamente lo que se comparte
- **Parámetro real:** : es un valor u otra entidad utilizada para pasar a un procedimiento o función. Están en la parte de la invocación
- **Parámetro formal:** es una variable utilizada para recibir valores de entrada en una rutina, subrutina etc. Se ponen en la parte de la declaración. Es una variable local a su entorno.
- **Ligadura posicional:** los parámetros formales y reales se ligan según la posición en la llamada y en la declaración.
- **Ligadura por palabra clave o nombre:** los parámetros formales y reales se ligan por el nombre. Se debe conocer los nombres de los parámetros formales.

Modo IN (mecanismo default):

- El parámetro formal recibe el dato desde el parámetro real. (la conexión es al inicio y se corta la vinculación)
- Posee dos tipos:
  - Por valor: Se transfiere el dato real y se copia. El parámetro formal actúa como una variable local de la unidad llamada, y crea otra variable.
    - Ventajas: El parámetro real se protege, ya que se hace una copia, por lo que el "verdadero" no se modifica. No hay efectos colaterales.
    - Desventaja: Consume tiempo para hacer la copia y ocupa más almacenamiento ya que el dato se duplica.
  - Por valor contante: es lo mismo que el de valor, pero tiene la particularidad de que no se puede modificar su valor. Ejemplo: parámetros In en ADA.
    - Ventajas: Protege los datos de la unidad llamadora, el parámetro real no se modifica.
    - Desventajas: Requiere realizas más trabajo para implementar los controles.

Modo OUT:

- El valor del parámetro formal se copia al parámetro real al terminar de ejecutarse la unidad que fue llamada (rutina)
- Posee dos tipos:
  - Por resultado: El parámetro formal es una variable local. El parámetro formal es una variable sin valor inicial porque no recibe nada. Debemos inicializar de alguna forma si el lenguaje no lo hace por defecto. Una vez que se vuelve asigna el valor a devolver (sobreescribo la variable que pase como parámetro al terminarse la rutina)
    - Ventajas: protege los datos de la unidad llamadora, el parámetro real no se modifica en la ejecución de la unidad llamada

        - Desventajas: Consume tiempo y espacio porque hace copia al final. Debemos inicializar la variable en la unidad llamada de alguna forma (si el lenguaje no lo hace por defecto)
    - Por Resultado de funciones: Es el resultado que me devuelven las funciones. Remplaza la invocación en la expresión que contiene el llamado, es decir, debo hacer una asignación para que ese reemplazo tenga sentido.

Modo IN/OUT:

- El parámetro formal recibe el dato del parámetro real y el parámetro formal le envía el dato al parámetro real al finalizar la rutina. (la conexión es al inicio y al final)
- La conexión no es todo el tiempo: evalua, conecta ligadura, ejecuta, conecta ligadura.
- Posee tres tipos:
    - Por Valor/Resultado: El parametro formal es una variable local que recibe una copia (a la entrada) del contenido del parametro real y el parametro real (a la salida) recibe una copa de lo que tiene el parametro formal. Basciamente Lo que hace la rutina lo copio en la variable. Cada referencia al parametro formal es una referencia local.
        - Tienen las ventajas y desventaja de cada uno.
    - Por Referencia: Comparte la direccion del parametro real al parametro formal. Como si fuese una especie de puntero el parametro formal. El parametro formal va a ser una variable local a la unidad llamadora que contiene la direccion en el ambiente no local. Cualquier cambio que se realice en el parametro formal dentro del cuerpo del subprograma quedara registrado en el parametro real.
        - Ventajas: eficiente en tiempo y espacio ya que no se realizan copias de datos.
        - Desventajas: El acceso a datos es lento por la indireccion. Es posible que se modifque el dato sin querer. Es muy dificil la verificacion de programa, esto se debe a que se pueden generar alias que afecten la legibilidad.
    - Por nombre: El parámetro formal es sustituido textualmente por una expresión del parámetro real más un puntero al entorno del parámetro real. (se maneja una estructura aparte que resuelve esto). Se establece la ligadura entre parámetro formal y parámetro real en el momento de la invocación, pero la "ligadura de valor" se difiere hasta el momento en que se lo utiliza (la dirección se resuelve en ejecución). Distinto a por referencia. Es decir, no apunta a una dirección fija, puede ir cambiando (pero el nombre tiene que si ser el mismo)
        - Ventajas: Es un metodo mas flexible pero mas lento, ya que debe evaluarse cada vez que se usa.
        - Desventajas: Es dificil de implementar y genera soluciones confusas para el lector y el escritor.

Tipo de pasaje de parámetros	Lenguaje
• Por defecto con copia IN • Por resultado OUT • IN OUT ○ Para los tipos primitivos indica por valor-resultado	ADA

○ Para los tipos no primitivos, datos compuestos (arreglo , registro) se hace por referencia.	
• Por valor (Si se necesita por referencia se usa punteros) • Permite pasaje por valor constante, agregando const	C
• Por valor, pero si se pasa un objeto "mutable", no se hace una copia sino que se trabaja sobre él.	Ruby
• El único mecanismo contemplado es el paso por copia de valor. Pero como las variables de tipo no primitivos son todas referencias a variables anónimas en la heap, el paso por valor de una de estas variables son en realidad un paso por referencia de las variables	JAVA
• Se puede pasar de dos formas: ○ Inmutables: actuara como por valor ○ Mutables: No se hace una copia sino que se trabaja sobre él.	Python

a- Ada es más seguro que Pascal, respecto al pasaje de parámetros en las funciones. Explique por qué.

Ada es más seguro que Pascal en cuanto al pasaje de parámetros en las funciones debido a su sistema de tipos más estricto y su sintaxis más clara y explícita para especificar los modos de paso de parámetros.

b- Explique cómo maneja Ada los tipos de parámetros in-out de acuerdo al tipo de dato

Ada maneja los parámetros in-out de acuerdo al tipo de dato que se está utilizando, utilizando una técnica de paso por referencia para tipos de datos simples, y una técnica de copia y devolución para tipos de datos más complejos. Esto permite un manejo seguro y eficiente de los parámetros in-out en los programas Ada.

Pasaje de función como parámetro y el ambiente local

- SHALLOW (parecido a buscar una variable por cadena dinámica): El ambiente de referencia, es el del subprograma que tiene el parámetro formal subprograma.

- DEEP (parecido a buscar una variable por cadena estática): El ambiente es el del subprograma dónde está declarado el subprograma usado como parámetro real. Se utiliza en los lenguajes con alcance estático y estructura de bloque.

ACLARACION: Hacer una cadena dinámica usando DEEP, es lo mismo que hacer SHALLOW, así que si piden hacer un ejercicio, básicamente solo se haría de un SHALLOW y un DEEP, sin especificar que cadena.

ANOTACIONES:

- Cuando se trata del pasaje por nombre, tal que fun1 (a[i]), la i siempre es la del ambiente donde se invoca. Literalmente escribo el parámetro formal (en la pila) como el real que estoy pasando, es decir, lo escribo como a[i]
- Como marcar parámetro en la pila:
    - Por valor: X = 1
    - Por VR X = (VR: i) = 1
    - Por ref x= (y apunto al que hace referencia)
    - Por nombre X = (↑ vec [i])
    - Por resultado X = (R i)

Practica 7

Ejercicio 1: Sistemas de tipos:

1. **¿Qué es un sistema de tipos y cuál es su principal función?**

   Conjunto de reglas usadas por un lenguaje para estructurar y organizar sus tipos. El objetivo de un sistema de tipos es escribir programas seguros.

   Funciones:
   - Provee mecanismos de expresión:
       - Expresar tipos intrínsecos o definir tipos nuevos.
       - Asociar los tipos definidos con construcciones del lenguaje.
   - Define reglas de resolución:
       - Equivalencia de tipos – ¿dos valores tienen el mismo tipo?
       - Compatibilidad de tipos – ¿puedo usar el tipo en este contexto?
       - Inferencia de tipos – ¿cuál tipo se deduce del contexto?
   - Mientras más flexible el lenguaje, más complejo el sistema

2. **Definir y contrastar las definiciones de un sistema de tipos fuerte y débil (probablemente en la bibliografía se encuentren dos definiciones posibles. Volcar ambas en la respuesta). Ejemplificar con al menos 2 lenguajes para cada uno de ellos y justificar.**

   TIPADO FUERTE – TIPADO DÉBIL Se dice que el sistema de tipos es fuerte cuando especifica restricciones sobre como las operaciones que involucran valores de diferentes tipos pueden operarse. Lo contrario establece un sistema débil de tipos.

   Tipado débil → C
   Tipado fuerte → Python

3. **Además de la clasificación anterior, también es posible caracterizar el tipado como estático o dinámico. ¿Qué significa esto? Ejemplificar con al menos 2 lenguajes para cada uno de ellos y justificar.**

   Tipado estático: ligaduras en compilación.

Para esto puede exigir:
- Se puedan utilizar tipos de datos predefinidos
- Todas las variables se declaran con un tipo asociado
- Todas las operaciones se especifican indicando los tipos de los operandos requeridos y el tipo del resultado.

Tipado dinámico: ligaduras en ejecución, provoca más comprobaciones en tiempo de ejecución

Tipado estático → C
Tipado dinámico → Python

Ejercicio 2: Tipos de datos:

1. **Dar una definición de tipo de dato.**

   Podemos definir a un tipo como un conjunto de valores y un conjunto de operaciones que se pueden utilizar para manipularlos.

2. **¿Qué es un tipo predefinido elemental? Dar ejemplos.**

   Reflejan el comportamiento del hardware de abajo y son una abstracción de él.

   Ejemplos: enteros, reales, caracteres, booleanos, strings

3. **¿Qué es un tipo definido por el usuario? Dar ejemplos.**

   Los lenguajes de programación permiten al programador especificar agrupaciones de objetos de datos elementales y de forma recursiva, agregaciones de agregados

   Ejemplos: arreglos, listas, registros, enumerados.

Ejercicio 3: Tipos compuestos:

1. **Dar una breve definición de: producto cartesiano (en la bibliografía puede aparecer también como product type), correspondencia finita, uniones (en la bibliografía puede aparecer también como sum type) y tipos recursivos.**

   - Producto Cartesiano: es el producto entre conjuntos de diferentes tipos. Produce registros (Pascal) o struct (C).
   - Correspondencia Finita: un conjunto de valores en un conjunto de dominio. Son los vectores.
     - Es una función de un conjunto finito de valores de un tipo de dominio DT en valores de un tipo de dominio RT

  - DT-> tipo de dominio (int por ej)
  - RT-> resultado del dominio (acceso a través del índice)
- Unión y unión discriminada: la unión de uno o más tipos, es la disyunción de los tipos dados. Se trata de campos mutuamente excluyente (uso uno o el otro), no pueden estar al mismo tiempo con valores.
  - La unión discriminada se agrega un descriptor (enumerativo) que me permite saber con quién estoy trabajando y acceder correctamente a lo que tengo que acceder. Básicamente manipulo el elemento según el valor del discriminante, es una mejora de la unión.
  - La unión es insegura.
  - El chequeo de tipos debe hacerse en ejecución
- Recursión: un tipo de dato recursivo T se define como una estructura que puede contener componentes de tipo T. Un ejemplo son las listas.
  - Define datos agrupados
    - Cuyo tamaño puede crecer arbitrariamente
    - Cuya estructura puede ser arbitrariamente compleja
  - Los lenguajes soportan la implementación de tipos de datos recursivos a través de los punteros

2. **Identificar a qué clase de tipo de datos pertenecen los siguientes extractos de código. En algunos casos puede corresponder más de una:**

Java

```
class Persona {
  String nombre;
  String apellido;
  int edad;
}
```

Producto cartesiano

C

```
typedef struct _nodoLista {
  void *dato;
  struct _nodoLista *siguiente
} nodoLista;

typedef struct _lista {
  int cantidad;
  nodoLista *primero
} Lista;
```

Producto cartesiano y recursión

C

```
union codigo {
  int numero;
  char id;
};
```

Unión

Ruby correspondencia

```
hash = {
  uno: 1,
  dos: 2,
  tres: 3,
  cuatro: 4
}
```

PHP

```
function doble($x) {
  return 2 * $x;
}
```

Python

```
tuple = ('physics',
'chemistry', 1997, 2000)
```

Correspondencia finita

Haskell

```
data ArbolBinarioInt =
  Nil |
  Nodo int
    (ArbolBinarioInt dato)
    (ArbolBinarioInt dato)
```

Ayuda para interpretar:
'ArbolBinarioInt' es un tipo de dato que puede ser Nil ("vacío") o un Nodo con un dato número entero (int) junto a un árbol como hijo izquierdo y otro árbol como hijo derecho

Recursión

Haskell

```
data Color =
  Rojo |
  Verde |
  Azul
```

Ayuda para interpretar:
'Color' es un tipo de dato que puede ser Rojo, Verde o Azul.

Unión

Ejercicio 4: Mutabilidad/Inmutabilidad:

1. **Definir mutabilidad e inmutabilidad respecto a un dato. Dar ejemplos en al menos 2 lenguajes. TIP: indagar sobre los tipos de datos que ofrece Python y sobre la operación #freeze en los objetos de Ruby.**

   Mutabilidad e inmutabilidad son términos que se refieren a la capacidad o no de un dato de ser modificado después de su creación. Un dato mutable es aquel que se puede cambiar después de haber sido creado, mientras que un dato inmutable es aquel que no se puede cambiar después de haber sido creado.

   En Python, algunos ejemplos de datos inmutables son los enteros, las cadenas y las tuplas. Por ejemplo, si creamos una cadena en Python, no podemos cambiar su contenido, solo podemos crear una nueva cadena que contenga los cambios que deseamos. Por otro lado, los datos mutables en Python incluyen listas, conjuntos y diccionarios, ya que se pueden modificar después de haber sido creados.

```python
# Ejemplo de datos inmutables en Python
a = 5         # Entero
b = "Hola"    # Cadena
c = (1, 2, 3)    # Tupla

# No se puede modificar una cadena
# La siguiente línea producirá un error de tipo TypeError
b[0] = "h"
```

   En Ruby, la mayoría de los objetos son mutables por defecto, pero se puede hacer que un objeto sea inmutable utilizando el método freeze. Después de que un objeto es congelado, no se puede modificar.

```ruby
# Ejemplo de datos mutables e inmutables en Ruby
str = "Hola"    # String mutable
arr = [1, 2, 3]    # Array mutable

# Congelar un objeto lo vuelve inmutable
```

```
 6 str.freeze
 7 arr.freeze
 8
 9 # No se puede modificar una cadena congelada
10 # La siguiente línea producirá un error de tipo RuntimeError
11 str[0] = "h"
```

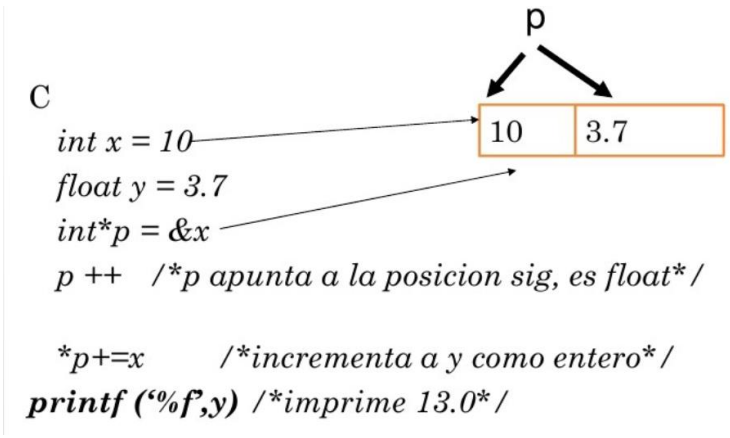
Ejercicio 5: Manejo de punteros:

1. **¿Permite C tomar el l-valor de las variables? Ejemplificar.**

   Sí, C permite tomar el l-valor (dirección de memoria) de las variables utilizando el operador "&".

2. **¿Qué problemas existen en el manejo de punteros? Ejemplificar.**

   1. Violación de tipos
      - la imagen explica mejor



   2. Referencias sueltas – referencias dangling
      - Una referencia suelta o dangling es un puntero que contiene una dirección de una variable dinámica que fue desalocada, si luego se usa el puntero producirá error

   3. Punteros no inicializados
      - Peligro de acceso descontrolado a posiciones de memoria
      - Soluciones:
        - Nil en pascal
        - Void en C/C++
        - Null en ADA y Python
   4. Punteros y uniones discriminadas

```
union ojo{
int int_var
int* int_ref}
```

- Me permite acceder a algo que no debo
- En el caso de C, este es el mismo efecto que causa la aritmética de punteros. Para resolver este problema asociado con los punteros Java elimina la noción de puntero explicito completamente.

5. Alias
   -

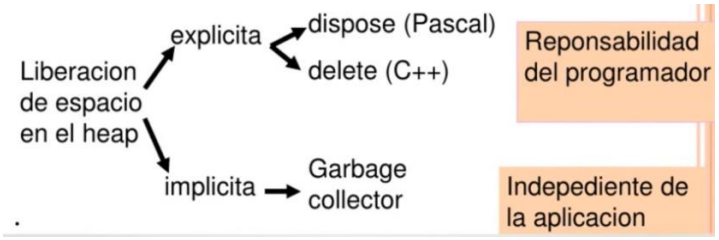
```
int* p1
int* p2
int x
p1 = &x
p2 = &x
```

p1 y p2 son punteros
p1 y x son alias
p2 y x también lo son

   - Si uno cambia su valor se ve reflejado en el otro. Sucede lo mismo si libre y esto genera el punto 2.

6. Liberación de memoria: objetos perdidos
   - Las variables puntero se alocan como cualquier otra variable en la pila de registros de activación
     - Los objetos apuntados que se alocan a través de la primitiva new son alocados en la heap
     - La memoria disponible (heap) podría agotarse a menos que de alguna forma se devuelva el almacenamiento alocado liberado

Liberación de memoria



- Requiere o no la intervención del usuario
- Reconocimiento de que porción de memoria es basura
- Explicita
  - El reconocimiento de la basura recae en el programador, quien notifica al sistema cuando un objeto ya no se usa
  - No garantiza que no haya otro puntero que apunte a esa dirección

  Ejemplos de esto serian el dispose en pascal, el cual es una forma explícita de liberación de memoria

- Implícita
  - El sistema, durante la ejecución tomará la decisión de descubrir la basura por medio de un algoritmo de recolección de basura (garbage collector)

- Importante para los lenguajes que utilicen frecuentemente variables dinámicas (LISP, Python)
- Se ejecuta durante el procesamiento de las aplicaciones
- Debe ser muy eficiente.

Ejercicio 6: TAD :

1. **¿Qué características debe cumplir una unidad para que sea un TAD?**

   Tipo abstracto de dato (TAD) es el que satisface:
   - Encapsulamiento: la representación del tipo y las operaciones permitidas para los objetos del tipo se describen en una única unidad sintáctica. Refleja las abstracciones descubiertas en el diseño
   - Ocultamiento de la información: la representación de los objetos y la implementación del tipo permanecen ocultos. Refleja los niveles de abstracción. Modificabilidad

Practica 8

¿Qué es una excepción?

Es una condición inesperada o inusual que surge durante la ejecución del programa y no puede ser manejada en el contexto local. Condición anómala e inesperada que necesita ser controlada.

Para que un lenguaje trate excepciones debe proveer mínimamente:

- Un modo de definirlas
- Una forma de reconocerlas
- Una forma de lanzarlas y capturarlas
- Una forma de manejarlas especificando el código y respuestas
- Un criterio de continuación

No todos los lenguajes lo proveen, por ejemplo C estándar no provee manejo de excepciones

¿Qué ocurre cuando un lenguaje no provee manejo de excepciones? ¿Se podría simular? Explique cómo lo haría

Cuando un lenguaje de programación no proporciona un mecanismo de manejo de excepciones, puede ser más difícil y menos estructurado manejar errores y situaciones excepcionales en el código. Sin un manejo adecuado de excepciones, los errores pueden propagarse sin control, lo que puede llevar a un comportamiento inesperado y a una mayor dificultad para depurar el código.

Aunque no se disponga de un mecanismo de manejo de excepciones incorporado en el lenguaje, es posible simularlo utilizando técnicas alternativas. Una forma común de simular el manejo de excepciones es mediante el uso de estructuras de control condicionales y funciones de retorno de errores.

¿Qué modelos diferentes existen en este aspecto?

- **Reasunción**: cuando se produce una excepción, se maneja y al terminar de ser manejada se devuelve el control a la sentencia **siguiente** de donde se levantó la excepción.
    - **PL/1.**
- **Terminación**: cuando se produce una excepción, el bloque donde se levantó la excepción es **terminado** y se ejecuta el manejador asociado a la excepción.
    - **ADA**
    - **CLU**
    - **C++**
    - **Java**
    - **Python**
    - **PHP**

Excepciones en los distintos lenguajes en Resumen de la clase 10.

ACLARACION:

- PL/1 Maneja una pila de excepciones, cuando sale ON CONDITION (Manejador) begin … end, lo que pasa es que se apila en la pila. Si se desaloca la función en donde se declaro el manejador, el manejador sale de la pila. Si se encuentra SIGNAL CONDITION (Manejador), se va a agarrar el ultimo manejador de la pila que coincida con el nombre.

En Java para indicar que una excepción se propaga se utiliza la declaración “throws Exception1, Exception2” en la firma del método que arroja la excepcion. Esto indica que el método puede lanzar esas excepciones y que cualquier llamada a ese método debe manejar o propagar esas excepciones.

Indique diferencias y similitudes entre Phyton y Java con respecto al manejo de excepciones.

El manejo de excepciones en Python y Java comparte algunos conceptos fundamentales, pero también hay diferencias significativas. A continuación, se presentan las similitudes y diferencias clave entre Python y Java en cuanto al manejo de excepciones:

Similitudes:

1. Estructura try: Ambos lenguajes utilizan una estructura try para capturar y manejar excepciones. Se puede colocar código potencialmente problemático dentro de un bloque try, y las excepciones se capturan y manejan en bloques catch o except correspondientes.
2. Tipos de excepciones: Tanto Python como Java tienen una jerarquía de clases de excepciones predefinidas. Ambos lenguajes proporcionan una serie de excepciones estándar, como NullPointerException, IOException, entre otras, que se utilizan para capturar errores comunes.
3. Bloque finally: Ambos lenguajes permiten el uso de un bloque finally opcional, que se ejecuta siempre, ya sea que se produzca o no una excepción. Se utiliza para realizar limpieza de recursos u otras acciones necesarias, independientemente de si se lanzó o no una excepción.

Diferencias:

1. Declaración de excepciones: En Java, se requiere que los métodos declaren las excepciones específicas que pueden lanzar utilizando la palabra clave throws en su firma. En Python, no se requiere una declaración explícita de excepciones en la firma de un método.
2. Tipos de excepciones: En Java, las excepciones se dividen en dos categorías: excepciones comprobadas (checked exceptions) y excepciones no comprobadas (unchecked exceptions). Las excepciones comprobadas deben ser declaradas o manejadas explícitamente en el código. En Python, todas las excepciones son consideradas excepciones no comprobadas y no requieren una declaración explícita o manejo.
3. Bloque except: En Python, se utiliza la palabra clave except para capturar excepciones específicas y manejarlas en un bloque de código correspondiente. En Java, se utiliza la palabra clave catch para capturar y manejar excepciones.

¿Qué modelo de excepciones implementa Ruby?. ¿Qué instrucciones específicas provee el lenguaje para manejo de excepciones y cómo se comportan cada una de ellas?

Proporciona una serie de instrucciones específicas para el manejo de excepciones que permiten capturar, lanzar y manejar errores en el código. A continuación, se describen las instrucciones principales y su comportamiento en Ruby:

1. begin-rescue-end: Esta es la estructura básica utilizada para capturar y manejar excepciones en Ruby. El código problemático se coloca dentro del bloque begin, y las excepciones se capturan y manejan en el bloque rescue. Por ejemplo:

   ```ruby
   begin
     # Código problemático
   rescate ExceptionType
     # Manejo de la excepción
   end
   ```

   ExceptionType puede ser una clase de excepción específica o el tipo

   StandardError para capturar cualquier excepción estándar.

2. raise: Esta instrucción se utiliza para lanzar una excepción explícitamente en un punto determinado del código. Puede lanzar una excepción predefinida o una excepción personalizada. Por ejemplo:

   ```ruby
   raise ExceptionType, "Mensaje de error"
   ```

   ExceptionType es la clase de excepción que se lanza.

   "Mensaje de error" es un mensaje opcional que proporciona información adicional sobre la excepción.

3. rescue: Esta instrucción se utiliza dentro de un bloque begin-rescue para capturar excepciones específicas. Puede capturar excepciones individuales o agrupar múltiples excepciones. Por ejemplo:

   ```ruby
   begin
   ```

```
  # Código problemático
rescue ExceptionType1
  # Manejo de la excepción de tipo ExceptionType1
rescue ExceptionType2, ExceptionType3
  # Manejo de las excepciones de tipo ExceptionType2 y
ExceptionType3
end
```

Se pueden proporcionar múltiples cláusulas rescue para capturar diferentes tipos de excepciones.

4. ensure: Esta instrucción se utiliza dentro de un bloque begin-rescue para ejecutar código que se debe ejecutar siempre, independientemente de si se produce o no una excepción. Por ejemplo:

```
begin
  # Código problemático
rescue ExceptionType
  # Manejo de la excepción
ensure
  # Código que se ejecuta siempre
end
```

El bloque ensure se ejecutará incluso si no hay una coincidencia de excepción en el bloque rescue correspondiente.

5. Excepciones predefinidas: Ruby proporciona una serie de excepciones predefinidas que se pueden capturar y manejar según sea necesario. Algunas de las excepciones comunes incluyen StandardError (base de las excepciones estándar), TypeError, ArgumentError, ZeroDivisionError, entre otras. Estas excepciones se pueden capturar y manejar individualmente utilizando la cláusula rescue.

Además de estas instrucciones y conceptos principales, Ruby también ofrece funcionalidades adicionales como la definición de excepciones personalizadas mediante la creación de clases, el acceso a la traza de la pila de excepciones (backtrace), y la posibilidad de propagar excepciones a través de múltiples niveles de llamadas de métodos.

En el caso de Ruby, encaja más con el enfoque de "Reasunción" cuando se produce una excepción. En Ruby, cuando se lanza una excepción y se captura en un bloque rescue, una vez que se maneja la excepción, el control vuelve a la sentencia siguiente de donde se levantó la excepción.

Indique el mecanismo de excepciones de JavaScript.

JavaScript utiliza un mecanismo de excepciones similar a otros lenguajes de programación. Proporciona una estructura de control try-catch-finally para capturar y manejar excepciones. A continuación se describe el mecanismo de excepciones de JavaScript:

1. try-catch-finally: Esta estructura se utiliza para capturar y manejar excepciones en JavaScript. El código problemático se coloca dentro del bloque try, y las excepciones se capturan y manejan en el bloque catch. El bloque finally es opcional y se utiliza para ejecutar código que siempre debe ejecutarse, independientemente de si se produce una excepción o no. Por ejemplo:

```
try {
  // Código problemático
} catch (error) {
  // Manejo de la excepción
} finally {
  // Código que se ejecuta siempre
}
```

El bloque catch captura la excepción y la asigna a la variable error (que puede tener cualquier nombre que elijas).
El bloque finally se ejecutará incluso si no hay una coincidencia de excepción en el bloque catch correspondiente.

2. throw: Esta instrucción se utiliza para lanzar una excepción explícitamente en un punto determinado del código. Puede lanzar una excepción predefinida o una excepción personalizada. Por ejemplo:

   ```
   throw new Error('Mensaje de error');
   ```

   Error es una de las excepciones predefinidas en JavaScript, pero también puedes crear tus propias excepciones personalizadas.

3. Excepciones predefinidas: JavaScript proporciona un conjunto de excepciones predefinidas, como Error, TypeError, ReferenceError, SyntaxError, entre otras. Estas excepciones se pueden capturar y manejar utilizando la estructura try-catch.

4. Error object: En JavaScript, todas las excepciones son objetos que heredan de la clase Error. Estos objetos contienen información sobre el error, como el mensaje de error, la traza de la pila y otros detalles relevantes.

Es importante tener en cuenta que JavaScript también proporciona otros mecanismos adicionales para manejar excepciones, como el uso de bloques catch anidados para capturar excepciones específicas y el acceso a la traza de la pila a través de la propiedad stack del objeto Error. Además, las excepciones en JavaScript pueden propagarse a través de las llamadas de función hasta que se capturen o lleguen al ámbito global, donde pueden generar un error no capturado que detiene la ejecución del programa.

Tanto en Java como en Python, las excepciones primero se propagan estáticamente en try-catch y try-exception respectivamente anidados. Si no se encuentra el manejador, se propagan dinámicamente.

Practica 9

Estructura a nivel de sentencia

- Se dividen en tres grupos.

Sentencia

Simple:

- Es el flujo de control más simple.
- Ejecución de una sentencia a continuación de otra.
- El delimitador más general y más usado es el ";"
    - Sentencia 1;
    - …
    - Sentencia n;
- Hay lenguajes que NO tienen delimitador
    - Establecen que por cada línea haya sólo 1 instrucción. Se los llaman orientados a línea.
    - Ej: Fortran, Basic, Ruby, Phyton

Compuesta:

- Se pueden agrupar varias sentencias en una con el uso delimitadores:
    - Begin y End. En Ada, Pascal
    - { } en C, C++, Java, etc.

Asignación

- Es una sentencia que produce cambios en los datos de la memoria de forma que al L-Valor de un objeto de dato se le asigna el R-Valor de una expresión.
- Sintaxis de diferentes lenguajes:

A := B	Ej: Pascal, Ada, etc.
A = B	Ej: Fortran, C, Prolog, Python, Ruby, etc.
MOVE B TO A	COBOL
A← B	APL
(SETQ A B)	LISP

Si usamos un lenguaje con = tanto como para asignar como para realizar expresiones. ¿Cómo las distinguimos?

Distinción entre sentencia de asignación y expresión:

- En lenguajes convencionales existen diferencia entre sentencia de asignación y expresión.
    - Las sentencias de asignación devuelven el valor de la expresión y modifican la posición de memoria.
- La mayoría de los lenguajes de programación requieren que sobre el lado izquierdo de la asignación aparezca un l-valor

- Pero en C se define la sentencia de asignación como una expresión que tiene efectos colaterales
    - Las sentencias de asignación devuelven valores.
    - C evalúa de derecha a izquierda.
    -
    - C a su vez también toma como verdadero todo lo que sea distinto de 0.
        - Por si usás = y no ==.

Circuitos cortos (short-circuit):

En un lenguaje que utiliza el circuito corto, la evaluación de una expresión lógica se detiene tan pronto como se determina el resultado final sin necesidad de evaluar el resto de la expresión.

- Operador "y" (&&): Si la primera parte de la expresión es falsa, el resultado final será falso sin importar el valor de la segunda parte. Por lo tanto, en un circuito corto, si la primera parte es falsa, la segunda parte no se evalúa.
- Operador "o" (||): Si la primera parte de la expresión es verdadera, el resultado final será verdadero sin importar el valor de la segunda parte. Por lo tanto, en un circuito corto, si la primera parte es verdadera, la segunda parte no se evalúa.

Circuitos largos (non-short-circuit):

En un lenguaje que utiliza el circuito largo, la evaluación de una expresión lógica continúa evaluando todas las partes de la expresión, incluso si el resultado final ya se ha determinado.

- Operador "y" (&&): En un circuito largo, ambas partes de la expresión se evalúan independientemente del valor de la primera parte. Esto asegura que se realicen todas las comprobaciones necesarias.
- Operador "o" (||): En un circuito largo, ambas partes de la expresión se evalúan independientemente del valor de la primera parte. Esto asegura que se realicen todas las comprobaciones necesarias.

¿Qué regla define Delphi, Ada y C para la asociación del else con el if correspondiente? ¿Cómo lo maneja Python?

En lenguajes como Delphi, ADA y C, la regla de asociación del else se basa en la coincidencia de las llaves o palabras clave correspondientes al if más cercano sin un else.

La asociación del else se basa en la estructura de anidación de los bloques de código y la coincidencia de las llaves o palabras clave correspondientes.

El else cierra el último if abierto que se aplica cuando no se utilizan llaves o palabras clave adicionales para delimitar explícitamente los bloques de código. Si se utilizan llaves o palabras clave adicionales, la asociación del else se determina por la coincidencia de esas delimitaciones adicionales.

En cuanto a Python, utiliza una indentación especial para delimitar bloques de código en lugar de llaves o palabras clave como begin y end. En Python, la asociación del else con el if correspondiente se basa en la indentación. El else se asocia con el if anterior que tenga la misma indentación.

Las estructuras de control están explicadas en Resumen de la clase 9.

Qué diferencia existe entre el generador YIELD de Python y el return de una función. De un ejemplo donde sería útil utilizarlo

La diferencia principal entre el generador yield de Python y el return de una función es que yield permite generar una secuencia de valores en lugar de retornar un solo valor como lo hace return.

Cuando se utiliza yield en una función, esta se convierte en un generador. El generador no devuelve un resultado inmediatamente como lo haría una función con return, sino que genera un valor cada vez que se le solicita, manteniendo su estado y la capacidad de continuar desde donde se detuvo.

Un ejemplo útil de uso de yield sería una función que genere una secuencia infinita de números pares. En lugar de calcular todos los números pares hasta un límite y almacenarlos en una lista, podemos utilizar un generador con yield para obtener los números pares uno a uno, evitando así almacenar todos los números en memoria.

Describa brevemente la instrucción map en Javascript y sus alternativas.

La función map en JavaScript es una función de orden superior que se utiliza para transformar los elementos de un arreglo original y generar un nuevo arreglo con los resultados de aplicar una función a cada elemento del arreglo original.

Algunas alternativas a map en JavaScript incluyen:

- forEach: La función forEach también recorre cada elemento del arreglo original, pero no crea un nuevo arreglo con los resultados de la transformación. En su lugar, se utiliza para realizar operaciones o acciones en cada elemento del arreglo sin modificar el arreglo original ni generar un nuevo arreglo.
- filter: La función filter se utiliza para crear un nuevo arreglo que contenga solo los elementos que cumplan una determinada condición. En lugar de transformar cada elemento como lo hace map, filter decide si un elemento se incluye en el nuevo arreglo basado en una condición booleana.
- reduce: La función reduce se utiliza para combinar todos los elementos de un arreglo en un solo valor. A diferencia de map, que genera un nuevo arreglo con la misma longitud que el original, reduce puede generar un resultado de cualquier tipo, como un número, una cadena o un objeto.

Determine si el lenguaje que utiliza frecuentemente implementa instrucciones para el manejo de espacio de nombres. Mencione brevemente qué significa este concepto y enuncie la forma en que su lenguaje lo implementa. Enuncie las características más importantes de este concepto en lenguajes como PHP o Python.

Un espacio de nombres es un contenedor lógico que se utiliza para agrupar y organizar elementos, como variables, funciones, clases u otros identificadores, dentro de un programa. Su propósito principal es evitar colisiones de nombres y proporcionar un contexto separado para los elementos con el mismo nombre.

Un espacio de nombres permite que múltiples elementos con el mismo nombre coexistan en un programa sin conflicto, ya que cada elemento se encuentra dentro de un espacio de nombres único.

Al utilizar espacios de nombres, se puede acceder a los elementos dentro de ellos mediante una notación de calificación, utilizando el nombre del espacio de nombres seguido de un separador (.) y luego el nombre del elemento específico. Esto ayuda a evitar ambigüedades y asegura que se esté haciendo referencia al elemento correcto.

JavaScript no implementa instrucciones nativas para el manejo de espacios de nombres de la misma manera que otros lenguajes como PHP o Python. Sin embargo, se pueden simular espacios de nombres en JavaScript utilizando objetos y módulos.

En JavaScript, los objetos actúan como contenedores y se pueden utilizar para agrupar funciones, variables y otros elementos relacionados. Esto permite crear una estructura similar a un espacio de nombres para organizar el código.

En lenguajes como PHP y Python, los espacios de nombres tienen características más avanzadas, como la capacidad de importar y exportar elementos entre espacios de nombres, y la posibilidad de tener múltiples niveles de anidamiento. Esto facilita la organización y el control más sofisticado de los elementos dentro de un programa.

En PHP, por ejemplo, se puede definir y utilizar espacios de nombres utilizando la palabra clave namespace y se pueden importar elementos utilizando use. Además, los espacios de nombres pueden estar organizados en una estructura de directorios que refleja la estructura de los archivos fuente.

En Python, los módulos actúan como espacios de nombres, y se pueden importar utilizando la declaración import. Python también permite la creación de paquetes, que son directorios que contienen módulos relacionados y proporcionan una forma de organizar y agrupar elementos.

A su vez, en un programa de Python hay tres tipos de espacios de nombres:

- Integrado: contiene los nombres de todos los objetos integrados de Python.
- Global: contiene cualquier nombre definido en el nivel del programa principal.
- Local: espacio de nombres local para la función y permanece hasta que la función finaliza.

Practica 10

Ejercicio 1: Un programa en un lenguaje procedural es una secuencia de instrucciones o comandos que se van ejecutando y producen cambios en las celdas de memoria, a través de las sentencias de asignación. ¿Qué es un programa escrito en un lenguaje funcional? y ¿Qué rol cumple la computadora?

Un programa escrito en un lenguaje funcional se basa en la evaluación de funciones y expresiones sin modificar directamente el estado de las celdas de memoria. La computadora desempeña el papel de evaluar y simplificar las expresiones, permitiendo resolver problemas de manera eficiente y concisa.

Ejercicio 2: ¿Cómo se define el lugar donde se definen las funciones en un lenguaje funcional?

En un lenguaje funcional, las funciones se definen generalmente en el ámbito global o en el ámbito local de otras funciones. El lugar donde se definen las funciones puede variar dependiendo del lenguaje y del estilo de programación funcional utilizado. A continuación, se describen algunos escenarios comunes:

1. Definición en módulos o archivos: Muchos lenguajes funcionales permiten organizar las funciones en módulos o archivos separados. Cada módulo puede contener un conjunto de funciones relacionadas que se importan en otros módulos cuando sea necesario. Esto ayuda a mantener un código modular y facilita la reutilización de funciones en diferentes partes del programa.

2. Definición en el ámbito global: En muchos lenguajes funcionales, las funciones pueden ser definidas directamente en el ámbito global del programa. Esto significa que las funciones son accesibles desde cualquier parte del programa. Pueden ser invocadas y utilizadas en cualquier momento. Estas funciones globales pueden tener nombres únicos y pueden ser reutilizadas en múltiples partes del programa.

3. Definición local dentro de una función: En muchos lenguajes funcionales, es posible definir funciones dentro del ámbito local de otra función. Estas funciones se conocen como funciones locales o funciones internas. Estas funciones solo son accesibles dentro de la función que las contiene y se utilizan para realizar cálculos específicos o proporcionar una lógica auxiliar para la función principal. Las funciones locales pueden capturar y acceder a las variables definidas en el ámbito de la función que las contiene.

4. Definición mediante expresiones lambda: En algunos lenguajes funcionales, como Lisp o Haskell, las funciones pueden ser definidas mediante expresiones lambda. Una expresión lambda es una función anónima que no requiere un nombre explícito. Estas funciones se definen en línea y se utilizan directamente en el contexto donde son necesarias. Las expresiones lambda son especialmente útiles cuando se requiere una función simple y no es necesario nombrarla o reutilizarla en otros lugares del programa.

Ejercicio 3: ¿Cuál es el concepto de variables en los lenguajes funcionales?

La noción de variable es la de “variable matemática”, no la de celda de memoria. Las variables son inmutables y no pueden ser modificadas una vez asignado un valor. Esto promueve la seguridad, la concurrencia y facilita el razonamiento y la comprensión del código.

Ejercicio 4: ¿Qué es una expresión en un lenguaje funcional? ¿Su valor de qué depende?

Una expresión es una combinación de valores, variables y funciones que se evalúa para producir un resultado. El valor de una expresión depende únicamente de los valores de las

subexpresiones que la componen. Una expresión siempre produce el mismo resultado cuando se evalúa con los mismos valores de entrada, sin importar cuándo o dónde se evalúe.

Ejercicio 5: ¿Cuál es la forma de evaluación que utilizan los lenguajes funcionales?

Orden aplicativo: Aunque no lo necesite siempre evalúa los argumentos. Evalúa las expresiones de inmediato, es decir, tan pronto como las expresiones se encuentran en el flujo de control del programa. Calcula los valores de las expresiones antes de utilizarlos en el programa. Esto significa que los argumentos de las funciones se evalúan antes de que se llame a la función y los resultados se pasan a la función.

Orden normal: No calcula más de lo necesario. En lugar de evaluar una expresión inmediatamente, retrasa la evaluación hasta que el resultado sea necesario en otra parte del programa. Esto permite evitar la evaluación innecesaria de expresiones y mejora la eficiencia en ciertos casos. Una expresión compartida NO es evaluada más de una vez.

Ejercicio 6: ¿Un lenguaje funcional es fuertemente tipado? ¿Qué tipos existen? ¿Por qué?

Sí, en general, los lenguajes funcionales son considerados fuertemente tipados. Esto significa que los lenguajes funcionales imponen restricciones más estrictas en las operaciones y conversiones entre diferentes tipos de datos, y requieren una correspondencia precisa de tipos para realizar operaciones.

En los lenguajes funcionales, los tipos de datos son importantes y se verifican en tiempo de compilación para garantizar la coherencia y la seguridad del programa. Los tipos ayudan a prevenir errores comunes, como la aplicación de operaciones incorrectas o la asignación de valores incompatibles.

Los tipos comunes que se encuentran en los lenguajes funcionales incluyen:

Básicos: Son los primitivos, ejemplo: NUM (INT y FLOAT) (Números) BOOL(Valores de verdad) CHAR(Caracteres)

Derivados: Se construyen de otros tipos, ejemplo:

(num,char) → Tipo de pares de valores

(num → char) → Tipo de una función

Ejercicio 7: ¿Cómo definiría un programa escrito en POO?

Un programa escrito con un lenguaje POO es un conjunto de objectos que interactúan mandándose mensajes.

Ejercicio 8: Diga cuáles son los elementos más importantes y hable sobre ellos en la programación orientada a objetos.

Los elementos son: objetos, mensajes, métodos y clases.

- Los **objetos** son datos abstractos, que tienen estado interno y comportamiento.
- Los **mensajes** son peticiones de un objeto a otro para que este se comporte de una determinada manera.
- Los **métodos** son programas asociados a un objeto y cuya ejecución solo puede desencadenarse a través de un mensaje recibido por este o por sus descendientes
- Las **clases** son tipos definidos por el usuario que determina las estructuras de datos y las operaciones asociadas con ese tipo. Cada objeto pertenece a una clase y este tiene su funcionalidad.

Ejercicio 9: La posibilidad de ocultamiento y encapsulamiento para los objetos es el primer nivel de abstracción de la POO, ¿cuál es el segundo?

El segundo nivel de abstracción en la programación orientada a objetos (POO) después del ocultamiento y encapsulamiento es la herencia. Permite crear nuevas clases basadas en clases existentes, heredando sus atributos y métodos, y estableciendo relaciones jerárquicas entre las clases. La herencia promueve la reutilización de código, la modularidad y la extensibilidad en el diseño de programas orientados a objetos.

Ejercicio 10: ¿Qué tipos de herencias hay? Cuál usa Smalltalk y C++

Existen diferentes tipos de herencia en la programación orientada a objetos. Los dos tipos más comunes son la herencia simple (single inheritance) y la herencia múltiple (multiple inheritance).

1. Herencia simple: Una clase puede heredar de una única clase base. Esto significa que una clase derivada puede extender y especializar los atributos y métodos de una única clase padre. La herencia simple promueve una relación jerárquica simple entre las clases y se utiliza ampliamente en muchos lenguajes orientados a objetos.
2. Herencia múltiple: Una clase puede heredar de múltiples clases bases. Esto permite que una clase derivada herede atributos y métodos de varias clases padres diferentes. La herencia múltiple ofrece una mayor flexibilidad y capacidad de reutilización de código, pero también puede ser más compleja de manejar y puede dar lugar a problemas de ambigüedad si existen métodos con el mismo nombre en diferentes clases padres.

- Smalltalk: Utiliza la herencia simple. Una clase puede tener una única clase padre de la cual hereda.
- C++: Admite tanto herencia simple como herencia múltiple. Permite que una clase herede de una única clase base o de varias clases bases diferentes.

Es importante tener en cuenta que algunos lenguajes de programación no admiten la herencia múltiple debido a la complejidad que puede generar. En su lugar, pueden proporcionar alternativas, como interfaces o mixins, para lograr la reutilización de código de manera más controlada.

Ejercicio 11: En el paradigma lógico ¿Qué representa una variable? ¿y las constantes?

En el paradigma lógico, una variable se refiere a elementos indeterminados que pueden sustituirse por cualquier otro. Los nombres de las variables comienzan con mayúsculas y pueden incluir números

"humano(X)", la X puede ser sustituída por constantes como: juan, pepe, etc.

Ejercicio 12: ¿Cómo se escribe un programa en un lenguaje lógico?

En un lenguaje lógico, los programas son una serie de aserciones lógicas.

El conocimiento se representa a través de reglas y hechos, los objetos se representan por términos, los cuales tienen constantes (determinado) y variables (indeterminado).

Hecho

son relaciones entre objetos y siempre son verdaderas:

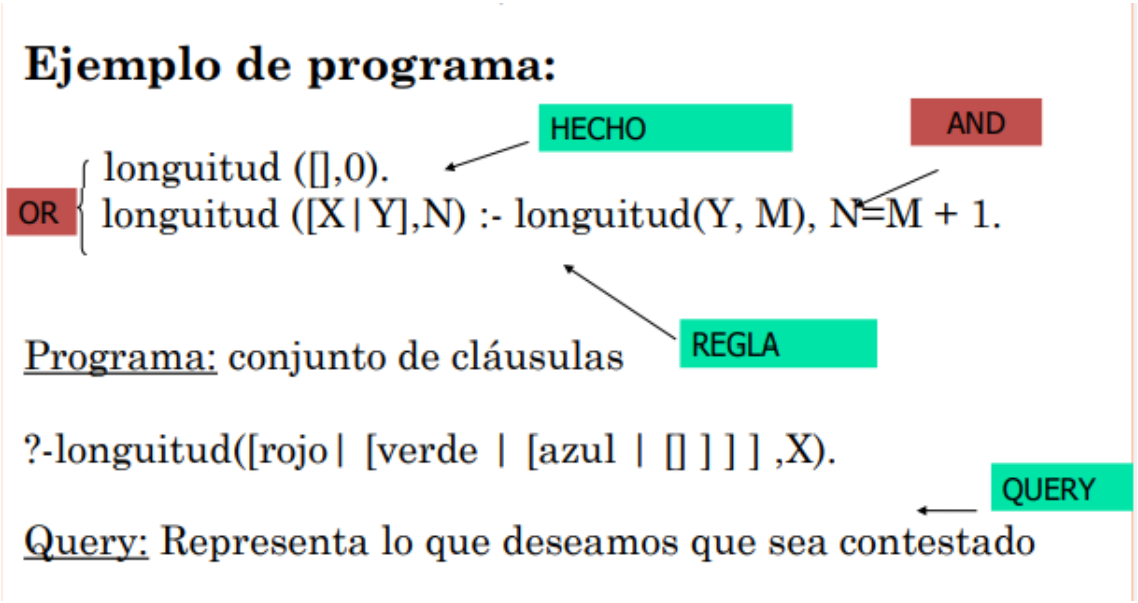
- tiene(coche,ruedas): representa el hecho que un coche tiene ruedas
- longitud([],0): representa el hecho que una lista vacía tiene longitud cero
- moneda(peso): representa el hecho que peso es una moneda.

Regla: Cláusulas de Horn

La cláusula tiene la forma conclusión :- condición

Donde :- es If, conclusión es un predicado y condición es una conjunción de predicados separados por comas, representando un AND lógico.

Seria como decir if condición else conclusión.



En estos programas se van haciendo consultas, que es un Query. El programa va deduciendo a partir de los hechos y reglas y con eso responde la consulta.

Ejercicio 14: Describa las características más importantes de los Lenguajes Basados en Scripts. Mencione diferentes lenguajes que utilizan este concepto. ¿En general, qué tipificación utilizan?

Lenguajes basados en script

En un principio estos lenguajes eran un conjunto de comandos escritos en un archivo para ser interpretado. Este archivo se denomina script.

Los primeros LBS era un conjunto de comandos interpretados como llamadas al sistema, como manejo o filtrado de archivos.

Luego se agregaron variables, flujo de control, etc y fueron escalando hasta en convertirse en lenguajes de programación completos como los conocidos actualmente.

Estos lenguajes tienden a mejorar flexibilidad, desarrollo rápido y chequeo dinámico. Su sistema de tipos se construye sobre conceptos de mas alto nivel como tablas patrones listas y archivos.

Ejemplos serian Python, Javascript, PHP y Perl

La definición seria : “Los lenguajes script no asumen la existencia de componentes útiles en otros lenguajes. Su intención no es escribir aplicaciones desde el comienzo sino por combinación de componentes.”

Sus características son:

- Combinar programas: Se diseñaron para pegar programas para construir un sistema mas grande, se utilizan como lenguajes de extensión ya que el usuario puede extender la funcionalidad de las herramientas script
- Desarrollo y evolución rápida: Algunos script se escriben y ejecutan una sola vez, en otros casos se utilizan mas frecuentemente por lo que deben ser fácilmente adaptados a nuevos requerimientos. Esto implica que deben ser fáciles de escribir y con sintaxis concisa.
- Asociado a editores livianos: Pueden ser escritos en procesadores de texto simples e incluso ejecutados en consola por su interprete
- Interpretados: La velocidad de ejecución de los script no es de importancia critica. Los gastos generales de interpretación y de comprobación dinámica se puede tolerar.
- Suelen tener un alto nivel de procesamiento de strings, un alto nivel para soporte de interfaces de usuario y tipado dinámico.
- Tipado dinámico: Puede necesitarse intercambiar datos de distinto tipo entre distintos subsistemas y estos pueden ser incompatibles, por esto el sistema de tipos simple podría ser demasiado inflexible y por otro lado uno muy completo haría que no se pueda desarrollar rápido. Que usen tipado dinámico los hace menos complicados pero son mas inseguros y menos eficientes.
- Los LBS son ricos (ñam) en conjuntos, diccionarios , tuplas, listas. Es común poder indizar arreglos a través de cadenas de caracteres , lo que implica tablas de hash y usar garbage collectors.