REPORTE DE PRÁCTICA 1

Elementos fundamentales de los lenguajes de programación

1. Nombres y Entorno

- ### 1.1 Nombres y Objetos

  Forma fundamental de abstracción. Proveen un mecanismo para referirse a cualquier cosa y su ***alcance*** en el cual se mapea a una entidad en particular, deberá estar restringido a un contexto.

  [**Alcance:** Determina a que entidad se refiere un nombredentro del código fuente.]

- ### 1.2 Entornos y Bloques

  Los bloques son una unidad fundamental de organización de programas común a la mayoría de los lenguajes. Son una **sección de texto de programa con enlaces de nombres que son locales para el bloque**. Corresponde a un marco de entorno.

  - TIPOS:
    a. Corresponde al cuerpo de una función.
    b. En línea, no en el cuerpo, pero anidado en otro bloque.

- ### 1.3 Reglas de alcance

  1.3.1 Bloques de línea anidados: Cada bloque corresponde a un marco, lo que da como resultado un entorno con marcos anidados. ***Un nombre introducido por un bloque es visible dentro de un bloque anidado dentro de él***, a menos que el bloque anidado redifina el nombre.

1.3.2 Alcance en funciones: Las funciones introducen un elemento de elección que ***no está presente en los bloques en línea***. Un bloque en línea está textualmente anidado dentro de un bloque externo y su ejecución tiene lugar durante la ejecución del bloque externo.

1.3.3 Alcance estático: En este ámbito, el entorno en cualquier punto de un programa se puede deducir de la estructura sintáctica del código, sin considerar cómo evoluciona el cálculo en tiempo de ejecución.

El entorno no local de una función consta de aquellos enlaces no globales que son visibles en el texto del programa en el que aparece la definición de la función.

1.3.4 Alcance dinámico: El entorno depende de cómo evoluciona en tiempo de ejecución. El entorno no local de una función consta de aquellos enlaces que son visibles en el momento en que se llama a la función.

2. Administración de la memoria

En los lenguajes que permiten al usuario administrar manualmente la memoria de los objetos, muchos errores de programación resultan de una administración incorrecta de la memoria. Existen varias estrategias que reducen la posibilidad de errores relacionados con la gestión de la memoria.

- ### 2.1 Clases de duración de almacenamiento

- ### 2.2 Almacenamiento estático

  Se puede acceder a las variables declaradas en el ámbito global en cualquier punto de un programa, por lo que ***sus objetos correspondientes deben tener una vida útil que abarque todo el programa***. Se dice que estos objetos tienen una duración de almacenamiento estático.

- ### 2.3 Almacenamiento automático

  Los objetos asociados con variables locales a menudo tienen una duración de almacenamiento automática, lo que significa que se crean al inicio del alcance de la variable y se destruyen al salir finalmente del alcance.

- ### 2.4 Almacenamiento de hilo local o subproceso

  Los lenguajes que incluyen multi-hilos a menudo permiten que las variables se declaren con una duración de almacenamiento local del subproceso. ***La vida útil de sus respectivos objetos coincide con la duración de la ejecución de un subproceso***, de modo que un objeto local del subproceso se crea al inicio de un subproceso y se destruye al final.

- ### 2.5 Almacenamiento dinámico

Los objetos cuya vida útil ***no está vinculada a la ejecución de un fragmento de código específico*** tienen una duración de almacenamiento dinámico. Estos objetos suelen ser creados explícitamente por el programador mediante una llamada a una rutina de asignación de memoria como ***malloc()*** o mediante un mecanismo de creación de objetos como ***new***.

- ### 2.6 Semántica de valor y referencia

  Los lenguajes ***difieren en cuanto a si el almacenamiento de una variable es el mismo que el del objeto al que se refiere***, o si una variable tiene una referencia indirecta a un objeto. La primera estrategia suele denominarse semántica de valor y la segunda semántica de referencia.

- ### 2.7 Recolección de basura

  Esto implica el uso de mecanismos de tiempo de ejecución para detectar que los objetos ya no están en uso y recuperar su memoria asociada.

- ### 2.8 Reference counting

  Patrón de gestión de memoria en el que cada objeto tiene un conteo del número de referencias al objeto. Este conteo aumenta cuando se crea una nueva referencia al objeto y disminuye cuando una referencia se destruye omodifica para hacer referencia a un objeto diferente.

- ### 2.9 Tracing collectors

  La recolección de basura por rastreo, que ***rastrea periódicamente el conjunto de objetos en uso*** y recopila objetos a los que no se puede acceder desde el código del programa.

- ### 2.10 Finalizadores (finalizers)

  Son análogos a los destructores en un lenguaje como C++. Se llama a un finalizador cuando se recopila un objeto, lo que le ***permite liberar recursos internos de la misma manera que los destructores***.

3. Estructura de control

- ### 3.1 Cortocircuito

  Para solucionar estos problemas, los operadores booleanos en muchos lenguajes evalúan su operando izquierdo antes que el derecho y también provocan un cortocircuito. Esto significa que el lado derecho no se calcula si el valor general de la expresión se puede determinar únicamente a partir del lado izquierdo.

  if (x != 0 && foo(x)) {

  . . .

  }

- ### 3.2 Secuencias explícitas

  Algunos lenguajes proporcionan un mecanismo explícito para encadenar expresiones en una secuencia ordenada. Generalmente, el resultado de la secuencia de expresión en su conjunto es el resultado de la última expresión de la secuencia.

  int x = (3, 4);

  cout << x;

- ### 3.3 Asignación compuesta

  En la evaluación de operadores de asignación compuesta, el número de veces que se evalúa el lado izquierdo puede afectar el resultado en presencia de efectos secundarios.

  def foo(values):

  values.append(0)

  return values

  mylist = []

  foo(mylist)[0] += 1

- ### 3.4 Secuencias de declaración

  Las secuencias de declaraciones a menudo se agrupan en forma de bloques, que pueden aparecer en contextos donde se espera una sola declaración.

  S_1; S_2; ... ; S_N

- ### 3.5 Transferencia de control no estructurada

  Muchos lenguajes proporcionan un mecanismo simple para transferir el control en forma de goto. Esto generalmente se usa junto con una etiqueta que especifica qué declaración se ejecutará a continuación. Por ejemplo, el siguiente código C imprime números enteros en secuencia comenzando en 0:

  int x = 0;

  LOOP: printf("%d\n", x);

  x++;

  goto LOOP;

- ### 3.6 Control estructurado

Los lenguajes modernos proporcionan estructuras de control de nivel superior que goto, lo que permite estructurar el código de una manera más legible y fácil de mantener. Los constructos más básicos son aquellos usados para expresar cómputos del tipo condicional y repetición, dos características necesarias para que un lenguaje sea Turing completo, lo que significa que el lenguaje es equivalente en poder computacional a las máquinas de Turing.

if <*test*> *then* <*statement1*> else <*statement2*>

- ### 3.7 Excepciones

  Las excepciones proporcionan un mecanismo para implementar el manejo de errores de manera estructurada. Permiten separar la detección de errores de la tarea de la recuperación de un error, ya que a menudo ocurre que la ubicación del programa donde ocurre un error no tiene suficiente contexto para recuperarse del mismo. En cambio, una excepción permite detener el flujo normal de ejecución y pasar el control a un controlador que puede recuperarse del error.

  throw Exception();

3. Subprogramas

Un método ya citado para solucionar problemas complejos es dividirlo en subproblemas (problemas más sencillos), y a continuación dividir estos subproblemas en otros más simples, hasta que los problemas más pequeños sean fáciles de resolver. Esta técnica se suele denominar "divide y vencerás".

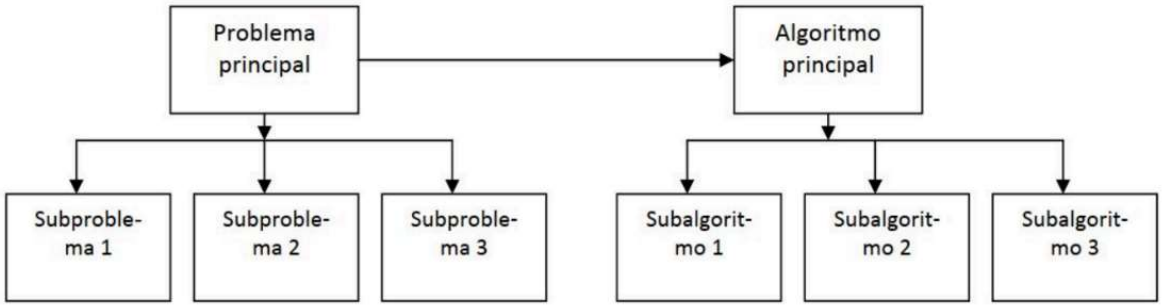
Normalmente ***las partes en que se divide un programa deben desarrollarse independientemente entre si***. Estas partes independientes ***se denominan subprogramas***.

Un subprograma puede realizar las mismas acciones que un programa:

- Leer datos
- Realizar cálculos
- Devolver resultados

Un subprograma ***es usado en un programa principal para un propósito específico***. El subprograma recibe datos desde el programa principal y devuelve resultados a este. Se dice entonces que el programa principal llama o invoca al subprograma.

El subprograma ***ejecuta una tarea, y a continuación devuelve el control al programa principal***, justo al lugar desde donde fue hecha la llamada. Un subprograma a su vez puede llamar a sus propios subprogramas. Existen dos tipos de subprogramas.


Problema principal
Algoritmo principal
Subproble-ma 1
Subproble-ma 2
Subproble-ma 3
Subalgorit-mo 1
Subalgorit-mo 2
Subalgorit-mo 3