💯

Final 100/100

▼ Primer Parcial

▼ Cite y explique tres características de los lenguajes que pueden afectar a la simplicidad general (+).

Muchos constructos: Un lenguaje con muchas construcciones básicas es más difícil de aprender. Los programadores a menudo aprenden solo una parte del lenguaje, lo que puede causar problemas de legibilidad si otros conocen otro subconjunto del lenguaje.

Multiplicidad de características: esto se refiere a tener más de una forma de realizar una operación en particular, por ejemplo, en Java, incrementar una variable entera se puede realizar de las siguientes formas:
count = count + 1; count += 1; count++; ++count.

Sobrecarga de operadores: con la sobrecarga de operadores, un símbolo tiene múltiples significados, aunque a menudo es útil, puede ser perjudicial para la legibilidad si se le permite a los usuarios crear su propia sobrecarga y no lo hacen de manera sensata.

▼ Como se produce la ejecución de un código máquina en una computadora basada en Von Neumann.

La ejecución de un programa en este tipo de arquitectura se produce en un proceso llamado el ciclo de búsqueda-ejecución. Los programas residen en la memoria pero se ejecutan en la CPU. Cada instrucción que se va a ejecutar debe trasladarse de la memoria al CPU. La dirección de la siguiente instrucción a ejecutarse se encuentra en un registro llamado program counter. Este ciclo se puede describir mediante el siguiente algoritmo:

```
Inicializar el program counter
Repetir indefinidamente
```

```
    Buscar la instrucción apuntada por el program counter
    Incrementar el program counter
    Decodificar la instrucción
    Ejecutar la instrucción
Fin del ciclo
```

▼ Proporcione una tabla indicando los criterios de evaluación de lenguajes y las características deseables de los lenguajes que afectan según Sebesta. Explique detalladamente cada uno de los criterios. Desarrolle brevemente una de las características que se evalúan. (+++++)

	Legibilidad	Escribibilidad	Confiabilidad
Simplicidad General	x	x	x
Ortogonalidad	x	x	x
Tipos de Datos	x	x	x
Diseño de Sintaxis	x	x	x
Soporte Para la abstracción		x	x
Expresividad		x	x
Chequeo de Tipos			x
Manejo de Excepciones			x
Aliasing Restringido			x

La legibilidad es la facilidad con la que se pueden leer y entender los programas.

> La simplicidad de un lenguaje afecta su legibilidad. Un lenguaje con muchas construcciones básicas puede ser más difícil de aprender. Los programadores a menudo solo aprenden un subconjunto del lenguaje, por lo tanto, los problemas empiezan cuando otros conocen un subconjunto distinto.
> Una segunda característica problemática es la multiplicidad de características, es decir, tener más de una forma de realizar una operación en particular.
> Un tercer problema potencial es la sobrecarga de operadores, donde un

símbolo tiene múltiples significados, aunque esto a menudo es útil, puede reducir la legibilidad si se le permite a los usuarios crear su propia sobrecarga y no lo hacen de manera sensata.

La ortogonalidad significa que unas pocas construcciones primitivas pueden combinarse de varias formas para crear estructuras de control y datos. Por ejemplo, si un lenguaje tiene cuatro tipos de datos y dos operadores de tipo, si los dos operadores de tipo pueden aplicarse entre sí y a los cuatro tipos de datos, se pueden definir un gran número de estructuras de datos. La farta de ortogonalidad produce excepciones en las reglas del lenguaje.
La ortogonalidad está relacionada con la simplicidad:; mientras más ortogonalidad, menos excepciones en las reglas del lenguaje, y mayor regularidad, facilitando el aprendizaje y la comprensión. La ortogonalidad extrema lleva a una complejidad innecesaria. Además, la gran cantidad de primitivas en los lenguajes causa una explosión de combinaciones, lo que aumenta la complejidad.
La simplicidad se logra combinando pocos constructos primitivos y un uso limitado del concepto de ortogonalidad.

La presencia de facilidades adecuadas para definir tipos de datos y estructuras de datos es otra ayuda significativa para la legibilidad. Por ejemplo, el utilizar un tipo numérico para un indicador porque no hay un tipo Booleano puede llevar a declaraciones no claras.

La sintaxis tiene un efecto significativo en la legibilidad de los programas. Dos decisiones importantes en el diseño sintáctico son las palabras especiales y la forma y significado.

La escribibilidad mide qué tan fácil es usar el lenguaje para crear problemas en un dominio especifico.

La simplicidad y ortogonalidad juegan un papel importante en la escribibilidad. Un lenguaje con demasiados constructos puede resultar en que los programadores no estén familiarizados con todos ellos, lo que puede llevar al mal uso de ciertas características o al desuso de otras opciones más eficientes o elegantes. Es preferible contar con un conjunto reducido de constructos primitivos y reglas coherentes para combinarlos.

El soporte para la abstracción es la capacidad de definir y utilizar estructuras complicadas u operaciones de tal manera que los detalles puedan ser ignorados. Las dos formas de abstracción más importantes son la abstracción de procesos y la abstracción de datos.

La expresividad se refiere a que un lenguaje tiene formas relativamente convenientes, en lugar de engorrosas de especificar cálculos.

Se dice que un programa es confiable si funciona de acuerdo a sus especificaciones en todas las condiciones.

La verificación de tipos es crucial para la confiabilidad del lenguaje, ya que consiste en comprobar errores de tipo en un programa, ya sea por el compilador o durante la ejecución. Es preferible realizarlo en tiempo de compilación ya que es más económica y permite detectar errores antes.

La capacidad de un programa de interceptar errores en tiempo de ejecución, tomar medidas correctivas y luego continuar es una ayuda evidente para la confiabilidad. Esta funcionalidad se llama manejo de excepciones.

El aliasing es tener dos o más nombres distintos de un programa que puedan usarse para acceder a una misma celda de memoria. Esta es una característica peligrosa, por lo que algunos lenguajes restringen en gran medida su uso para aumentar la confiabilidad.

▼ Cite y explique 2 de los ejemplos de decisiones de diseño que afectan al diseño de Sintaxis.

Palabras especiales: La legibilidad de un programa está fuertemente influenciada por la forma de las palabras especiales del lenguaje. Especialmente importante es el método de formar declaraciones compuestas o grupos de declaraciones. Por ejemplo, utilizar llaves para agrupar declaraciones puede dificultar la lectura, ya que todos los bloques terminan de la misma forma. En Ada, la claridad es mejorar utilizando palabras de cierre especificas (end loop, end for). Usar más palabras reservadas puede aumentar la legibilidad. Esto es un ejemplo de conflicto entre simplicidad y legibilidad.

Forma y significado: Es importante que la forma o apariencia de las declaraciones sugiera su propósito para mejorar la legibilidad. La semántica o significado debe seguir directamente a la sintaxis o forma.

▼ Cuales fueron los objetivos de diseño ALGOL? Por quienes fue diseñado? (+)

ALGOL fue diseñado en 1958 por un esfuerzo conjunto de la ACM (Association for Computing Machinery) de Estados Unidos y la sociedad GAMM de Europa para desarrollar un lenguaje de programación universal. Los tres objetivos principales de la reunión de diseño del lenguaje fueron:

- La sintaxis debería ser lo más cercana posible a la notación matemática estándar, y los programas escritos en él deberían ser legibles con poca explicación adicional.
- Debería ser posible usar el lenguaje para la descripción de algoritmos en publicaciones impresas.
- Los programas en este lenguaje deben ser traducibles mecánicamente a lenguaje máquina.

▼ Evaluación de smalltalk.

Smalltalk ha hecho mucho por promover dos aspectos importantes de la informática: interfaces gráficas de usuario y programación orientada a objetos. Los sistemas de ventanas actuales, surgieron a partir de smalltalk. Hoy en día, las metodologías de diseño de software y los lenguajes más importantes son orientados a objetos. Aunque algunos conceptos de la programación orientada a objetos comenzaron con SIMULA 67, fue en Smalltalk donde alcanzaron su madurez, consolidando su impacto duradero en el mundo de la informática.

▼ Defina formalmente Expresión Regular (++++).

Una **expresión regular** es una de las siguientes:

1. Una expresión regular básica **constituida** por un solo carácter a, donde *a* proviene de un alfabeto $\sum$ de caracteres legales; el metacarácter ϵ; o el metacarácter ϕ. En el primer caso, $L(a) = \{a\}$; en el segundo, $L(\epsilon) = \{\epsilon\}$; en el tercero, $L(\phi) = \{\}$.

2. Una expresión de la forma $r|s$, donde r y s son expresiones regulares. En este caso, $L(r|s) = L(r) \cup L(s)$.
3. Una expresión de la forma rs, donde r y s son expresiones regulares. En este caso. $L(rs) = L(r)L(s)$.
4. Una expresión de la forma r^*, donde r es una expresión regular. En este caso, $L(r^*) = L(r)^*$.
5. Una expresión de la forma (r), donde r es una expresión regular. En este caso, L((r))=L(r). De este modo, los paréntesis no cambian el lenguaje, sólo se utilizan para ajustar la precedencia de las operaciones.

▼ Explique cómo influye el avance general de la informática entre las razones para estudiar los conceptos de lenguajes de programación (+).

Aunque a menudo se pueden conocer los motivos de la popularidad de un lenguaje, muchos creen que los lenguajes más populares no siempre son los mejores. A veces, un lenguaje se usó ampliamente porque quienes elegían no estaban familiarizados con los conceptos de programación.

Por ejemplo, muchos creen que ALGOL60 debió haber reemplazado a fortran en la década de los 60, sin embargo, programadores y managers de esa época no entendían bien el diseño conceptual de ALGOL60, encontrando su descripción difícil de leer y entender. No apreciaban beneficios como la estructura de bloques y la recursión, por lo que no veían las ventajas de ALGOL60 sobre fortran.

En general, si quienes eligen lenguajes estuvieran bien informados, tal vez mejores lenguajes podrían reemplazar a los más inferiores.

▼ Explique cómo afectan la simplicidad y ortogonalidad de un lenguaje de programación a su facilidad de escritura.

La mayoría de características que afectan a la legibilidad también impactan la facilidad de escritura, ya que escribir un programa implica releer el código escrito.

Un lenguaje con demasiados constructos puede resultar en que los programadores no estén familiarizados con todos ellos, lo que puede llevar

al mal uso de ciertas características o al desuso de otras opciones más eficientes o elegantes.

Es preferible contar con un conjunto reducido de constructos primitivos y reglas coherentes para combinarlos, ya que esto facilita el aprendizaje y permite resolver problemas complejos de manera más eficiente.

Sin embargo, un exceso de ortogonalidad, donde casi cualquier combinación de primitivas es válida, también puede ser contraproducente. Esto puede permitir que errores en los programas pasen desapercibidos, llevando a absurdos en el código que no pueden ser detectados por el compilador.

▼ En que consiste la ortogonalidad en un lenguaje de programación. Provea un ejemplo de ortogonalidad, falta de ortogonalidad y exceso de ortogonalidad (++).

La ortogonalidad en un lenguaje de programación significa que unas pocas construcciones primitivas pueden combinarse de varias formas para crear estructuras de control y datos. Por ejemplo, si un lenguaje tiene 4 tipos de datos y 2 operadores de tipo, si los dos operadores de tipo pueden aplicarse entre sí y a los cuatro tipos de datos, se pueden definir un gran número de estructuras de datos.

Como ejemplo de falta de ortogonalidad en C, donde tenemos arreglos y registros, los registros pueden ser devueltos por funciones, los arreglos no. Los parámetros se pasan por valor, a excepción de los arreglos, los cuales se pasan por referencia. Un miembro de una estructura puede ser cualquier tipo de dato, excepto void o una estructura del mismo tipo. Un elemento de un arreglo puede ser cualquier tipo, excepto void o una función.

Un ejemplo de exceso de ortogonalidad es ALGOL68, el cual es probablemente considerado el lenguaje de programación más ortogonal. En este lenguaje la mayoría de los constructos producen valores, lo que permite combinaciones complejas. Por ejemplo, una condicional puede ser el lado izquierdo de una asignación:

(if (A<B) then C else D) := 3

▼ Explique los criterios de evaluación de lenguajes propuestos por Sebesta y presente una evaluación de algún

lenguaje de programación que conozca, utilizando los criterios y características, propuestos por Sebesta (+).

▼ Describe como ha influido la arquitectura informática conocida Von Neumann.

La arquitectura de Von Neumann ha tenido un profundo impacto en el diseño de lenguajes de programación durante los últimos 60 años. Estos lenguajes se llaman lenguajes imperativos. Casi todas las computadoras construidas desde la década de los 40 se han basado en esta arquitectura. En una computadora Von Neumann, los datos y programas se almacenan en la misma memoria, y como la CPU está separada de la memoria, las instrucciones y datos deben ser transmitidos desde la memoria a la CPU, así como los resultados de las operaciones.

Debido a esta arquitectura, las características centrales de los lenguajes imperativos son las variables, que modelan celdas de memoria; las declaraciones de asignación, que se basan en la operación de canalización; y la forma iterativa de repetición, que es la forma más eficiente de implementar la repetición en esta arquitectura..

▼ Indique un uso de los siguientes métodos de implementación: compilación, interpretación pura y sistemas de implementación híbridos.

Compilación: Los programas se traducen a lenguaje máquina, permitiendo una ejecución muy rápida una vez completada la ejecución. C, C++, Cobol

Interpretación Pura: Los programas se ejecutan directamente por un intérprete sin realizar ninguna traducción previa. APL, Lisp, JavaScript

Sistema de interpretación híbrida: Traducen programas de lenguajes de alto nivel a un lenguaje de nivel intermedio para facilitar la interpretación. Perl, Java

▼ Indique los seis motivos para estudiar las estructuras de los lenguajes de programación según Sebesta.

- Aumento de la capacidad para expresar ideas: Aquellos con una comprensión débil del lenguaje en el que se comunican están limitados en la complejidad de sus pensamientos. Los programadores enfrentan

las mismas limitaciones. El lenguaje de programación impone límites a las estructuras de control, datos y abstracciones que se pueden usar. Conocer una mayor variedad de características de lenguajes de programación puede reducir estas limitaciones.

En resumen, el estudio de conceptos de lenguajes de programación genera una apreciación por las características valiosas y fomenta su uso, incluso cuando el lenguaje no las soporta directamente.

- Mejor capacidad de elegir lenguajes apropiados: Muchos programadores sin educación formal en ciencias de la computación conocen solamente uno o dos lenguajes. Así, al elegir lenguajes para nuevos proyectos, tienden a usar los que ya conocen, aunque no sean los más adecuados. Si estuviesen familiarizados con más lenguajes y sus constructos, podrían seleccionar mejor el lenguaje apropiado.
- Aumento de la capacidad de aprender nuevos lenguajes: Una comprensión profunda de los conceptos generales de programación facilita el aprendizaje de nuevos lenguajes. Por ejemplo, los programadores con conocimientos de programación orientada a objetos encuentran más fácil aprender Ruby.

  Los desarrolladores deben estar preparados para aprender varios lenguajes debido a la cantidad de lenguajes en uso y a la naturaleza cambiante de las estadísticas.

  Es crucial que los programadores conozcan los conceptos básicos de los lenguajes de programación para entender descripciones, evaluaciones y literatura promocional, lo que les permite elegir y aprender nuevos lenguajes.
- Comprensión de la implementación: Es esencial abordar los problemas de implementación que afectan a los conceptos de los lenguajes de programación. Esto nos puede ayudar a entender por qué los lenguajes están diseñados de cierta manera y permite usarlos de forma más inteligente. Entender las opciones entre los constructos del lenguaje y sus consecuencias nos hace mejores programadores. Además, este conocimiento nos permite visualizar cómo una computadora ejecuta

distintos constructos del lenguaje e identificar la eficiencia de alternativas.

- Mejor uso de los lenguajes que ya se conocen: La mayoría de los lenguajes contemporáneos son grandes y complejos. En consecuencia, es poco común que un programador este familiarizado y utilice todas las características de un lenguaje. Al estudiar los conceptos de los lenguajes de programación, los programadores pueden aprender sobre partes previamente desconocidas de los lenguajes que ya usan y comenzar a utilizar estas características.
- Avance general de la computación: Aunque a menudo se puede entender por qué un lenguaje se volvió popular, muchos creen que los lenguajes más populares no siempre son los mejores. A veces, un lenguaje se usó ampliamente porque quienes elegían no estaban familiarizados con los conceptos de los lenguajes de programación. En general, si quienes eligen lenguajes estuvieran bien informados, tal vez mejores lenguajes reemplazarían a los inferiores.

▼ Como afecta la simplicidad general de un lenguaje de programación a su legibilidad? Explique detalladamente.

La simplicidad de un lenguaje de programación afecta su legibilidad. Un lenguaje con muchas construcciones básicas es más difícil de aprender. Los programadores a menudo aprenden solo un subconjunto del lenguaje, lo que puede causar problemas de legibilidad si otros conocen un subconjunto diferente.

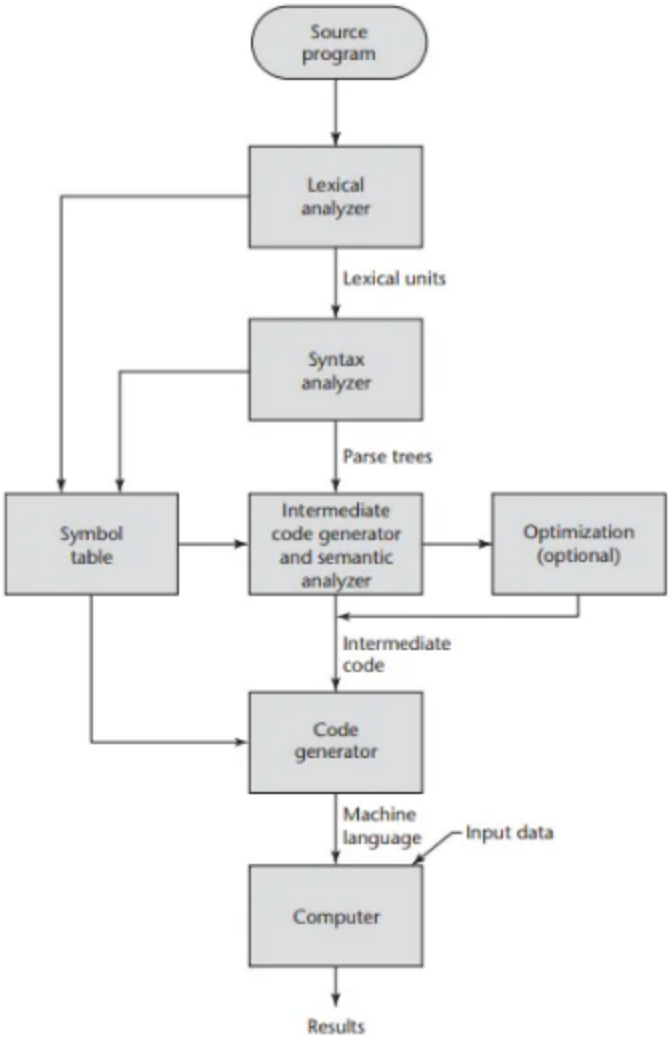
Una segunda característica complicante es la multiplicidad de características, es decir, tener más de una forma de realizar una operación en particular.

Un tercer problema potencial es la sobrecarga de operadores, donde un símbolo tiene múltiples significados, aunque esto a menudo es útil, puede reducir la legibilidad si se permite a los usuarios crear su propia sobrecarga y no lo hacen de manera sensata.

▼ Indique 3 criterios de diseño de 3 lenguajes distintos según Sebesta.

▼ **Describe el contexto histórico, las contribuciones claves, características innovadoras y el objetivo principal de los primeros compiladores FORTRAN.**
▼ **Presente una breve evaluación del impacto que ha tenido el lenguaje LISP sobre el diseño de los lenguajes.**
▼ **Describe las diferentes etapas del proceso de compilación y realice un gráfico del proceso de compilación. (+++)**

Figure 1.3
The compilation process



Primero, el analizador léxico agrupa los caracteres del programa fuente en unidades léxicas como identificadores y operadores, ignorando los comentarios.

Luego, el analizador sintáctico utiliza estas unidades para construir estructuras jerárquicas llamadas parse trees, que representan la estructura sintáctica del programa.

El generador de código intermedio produce un programa en un lenguaje a un nivel intermedio entre el programa fuente y el programa en lenguaje máquina. Durante este proceso, el analizador semántico verifica errores como los de tipo. Luego, la optimización mejora los programas haciéndolos más pequeños o más rápidos, o ambos.

El generador de código traduce la versión optimizada del código intermedio del programa al lenguaje máquina, listo para ser ejecutado.

La tabla de símbolos actúa como base de datos durante la compilación, almacenando el tipo y la información de atributo de cada nombre definido por el usuario en el programa. Esta información es colocada en la tabla de símbolos por los analizadores léxico y sintáctico y es utilizada por el analizador semántico y el generador de código.

▼ Describe elementos del desarrollo de uno de los siguientes lenguajes: Fortran, Lisp, Cobol, Algol. Presenta los siguientes puntos: Principales actores, proceso de diseño, características generales del lenguaje, objetivos, evaluación.

Fortran

Uno de los mayores avances de la informática fue la introducción de la IBM 704 en 1954, lo cual impulsó el desarrollo de Fortran. El desarrollo de Fortran comenzó incluso antes de la IBM 704, liderado por John Backus en IBM. En noviembre de 1954, John Backus y su equipo habían producido un informe el cual describía la primera versión de Fortran, Fortran 0. En dicho documento, se afirmaba que Fortran proporcionaría la eficiencia de los programas codificados a mano y la facilidad de programación de los sistemas de pseudocódigo interpretativo. En otro momento de optimismo, el documento afirmaba que Fortran eliminaría los errores de codificación y el proceso de depuración.

Fortran I, lanzado en 1957, introdujo características innovadoras como el formato de entrada/salida, nombres de variables de hasta seis caracteres, subrutinas definidas por el usuario, y las declaraciones If y

Do. Todas las declaraciones de control de Fortran I se basaban en las instrucciones del 704. Su éxito inicial se debió a la notable optimización del compilador, que logró producir código casi tan eficiente como el escrito a mano.

El efecto que Fortran ha tenido en el uso de los computadores, junto con el hecho de que todos los lenguajes posteriores le deben algo a Fortran, es realmente impresionante a la luz de los modestos objetivos de sus diseñadores. Una de las características de Fortran I, que permite la existencia de compiladores altamente optimizados, era que los tipos y el almacenamiento de todas las variables estaban fijos antes del tiempo de ejecución.
El éxito general de Fortran es difícil de exagerar: cambió drásticamente la forma en que se usan las computadoras. Esto se debe en gran parte, a que fue el primer lenguaje de alto nivel ampliamente utilizado.
Alan Perlis, uno de los diseñadores de ALGOL60, dijo: “Fortran es la lingua franca del mundo de la programación”.

Lisp

El interés en la Inteligencia artificial surgió a mediados de los 50. Investigadores de varias disciplinas llegaron a la conclusión de que era necesario desarrollar métodos para que las computadoras puedan procesar datos simbólicos en listas enlazadas.

En 1958, John McCarthy, investigó la computación simbólica y desarrolló una lista de requerimientos para realizar dichas computación, entre ellos estaban la recursión, las expresiones condicionales, y la necesidad de listas enlazadas asignadas dinámicamente y algún tipo de desasignación implícita. El lenguaje existente, FLPL, no cumplía con estos requisitos, lo que llevo a McCarthy a crear un nuevo lenguaje: Lisp, un lenguaje puramente funcional diseñado para procesar listas.

Lisp solo utiliza dos tipos de estructuras de datos: átomos y listas. Los átomos pueden ser símbolos o literales numéricos, y las listas son estructuras de datos enlazadas, cuyos elementos pueden ser átomos o sublistas. Internamente las listas se representan como listas enlazadas simples, donde cada nodo contiene dos punteros: uno al elemento de la lista y otro al siguiente nodo. Esto permite la inserción y eliminación en

cualquier punto de la lista.
Lisp fue diseñado como un lenguaje de programación funcional, donde toda la computación se realiza aplicando funciones a argumentos. Ni las sentencias de asignación ni las variables son necesarias en los programas de lenguajes funcionales. Además, permite que la repetición se especifique mediante llamadas recursivas a funciones, haciendo innecesario los bucles.
La sintaxis de Lisp es un modelo de simplicidad. El código del programa y los datos tienen exactamente la misma forma: listas entre paréntesis.

Lisp dominó completamente las aplicaciones de inteligencia artificial durante un cuarto de siglo. Gran parte de su reputación de ser altamente ineficiente ha sido eliminada. Además de su éxito en IA, Lisp fue pionero en la programación funcional, que ha demostrado ser un área vibrante de investigación en lenguajes de programación. Muchos investigadores creen que la programación funcional es un enfoque mucho mejor para el desarrollo de software que la programación procedural utilizando lenguajes imperativos.

ALGOL60

ALGOL60 se creó como un esfuerzo conjunto para desarrollar un lenguaje de programación universal destinado aplicaciones científicas. Este esfuerzo surgió en respuesta a la proliferación de lenguajes dependientes de máquinas, que dificultaban el intercambio de programas.
En 1957, varios grupos de usuarios de computadoras de Estados Unidos presentaron una petición a la ACM para formar un comité que estudiara y recomendara acciones para crear un lenguaje de programación científica independiente de la máquina. Previamente, en 1955, la sociedad GAMM en Europa formó un comité para diseñar un lenguaje algorítmico universal independiente de la máquina. Esto fue a raíz del miedo de los Europeos a ser dominados por IBM. Sin embargo, a finales de 1957, la aparición de varios lenguajes de alto nivel convenció al subcomité de GAMM que su esfuerzo debía ampliarse para incluir a los estadounidenses, y se envió una carta de invitación a la ACM.

La primera reunión de diseño se dio del 27 de mayo al 1 de junio de 1958, con cuatro miembros de la ACM y cuatro miembros de GAMM. Se establecieron tres objetivos principales:
La sintaxis del lenguaje debería ser lo más cercana posible a la notación matemática estándar, y los programas escritos en él deberían ser legibles con poca explicación adicional.
Debería ser posible usar el lenguaje para describir algoritmos en publicaciones impresas.
Los programas en el nuevo lenguaje deben ser traducibles mecánicamente a lenguaje de máquina.
Esta reunión logró producir un lenguaje que cumpliera con todos los requisitos, pero el proceso de diseño requirió innumerables compromisos.

El lenguaje diseñado en la reunión, cuyo nombre termino siendo ALGOL58, fue, en muchos aspectos, un descendiente de Fortran, generalizando muchas de sus características y añadiendo nuevas construcciones y conceptos. Algunas de las generalizaciones tenían que ver con el objetivo de no vincular el lenguaje a ninguna máquina en particular, y otras fueron intentos de hacer el lenguaje más flexible y poderoso. ALGOL58 formalizó el concepto de tipos de datos, añadió la idea de sentencias compuestas, y permitió características como identificadores de longitud ilimitada y cualquier número de dimensiones en matrices.

El informe de ALGOL58, publicado en 1958 por Perlis y Samelson, fue inicialmente bien recibido. El lenguaje se veía más como una colección de ideas que como un lenguaje estándar universal. En realidad, el informe no se concibió como un producto final, si no como un documento preliminar para la discusión internacional.
Al principio, tanto IBM como su principal grupo de usuarios SHARE, parecían adoptar ALGOL58. Sin embargo, pronto abandonaron ALGOL58 en favor de Fortran, debido a las complicaciones y costos asociados con la implementación y adopción de un nuevo lenguaje.

En 1959, ALGOL58 fue intensamente debatido, lo que llevó a numerosas sugerencias de modificación. Uno de los hitos de ese año fue la presentación del trabajo del comité de Zúrich, donde Backus presentó

su nueva notación para describir la sintaxis de los lenguajes de programación, que más tarde se conoció como BNF.
En enero de 1960, se celebró la segunda reunión de ALGOL, esta vez en Paris. El propósito de la reunión fue debatir las 80 sugerencias que habían sido presentadas para su consideración. Peter Naur jugo un papel importante al adaptar BNF para describir el nuevo lenguaje.

En la reunión de 1960, se realizaron modificaciones significativas a ALGOL58, dando lugar a ALGOL60. Entre los más importantes están los siguientes:
El concepto de estructura de bloques.
Dos métodos para pasar parámetros: por nombre y por valor.
Se permitió que los procedimientos fueran recursivos.
Se permitieron matrices dinámicas en pila.
Sin embargo, se omitieron características que podrían haber tenido un impacto drástico en el lenguaje, como las sentencias de entrada y salida, porque se pensaba que dependían de la máquina.

En algunos aspectos, ALGOL60 fue un gran éxito; en otros, fue un rotundo fracaso. Logró convertirse casi de inmediato en el único medio formal aceptable para comunicar algoritmos. Todos los lenguajes imperativos diseñados desde los 60 deben algo a ALGOL60. De hecho, la mayoría son descendientes directos o indirectos.
Fue pionero en muchos aspectos. Fue la primera vez que un grupo internacional intentó diseñar un lenguaje de programación. Fue el primer lenguaje que fue diseñado para ser independiente de la máquina.
También fue el primer lenguaje cuya sintaxis fue descrita formalmente.

Sin embargo, su implementación fue complicada, y la falta de soporte de IBM y la predominancia de Fortran contribuyeron a su fracaso para lograr un uso generalizado. Algunas de las características de ALGOL60 resultaron ser demasiado flexibles; dificultaban la comprensión y hacían ineficiente la implementación. Además, la falta de sentencias de entrada/salida y la complejidad percibida del BNF dificultaron su adopción..

COBOL

COBOL fue diseñado en 1959 por un comité de personas que se reunieron durante periodos relativamente cortos de tiempo, de manera similar a ALGOL60. En ese momento, la informática empresarial estaba en una etapa similar a la de la informática científica durante el desarrollo de Fortran. COBOL surgió en un contexto donde varios proyectos de diseño de lenguajes estaban en desarrollo.

El primer encuentro formal para crear un lenguaje común para aplicaciones comerciales, patrocinado por el Departamento de Defensa de EEUU, tuvo lugar en mayo de 1959. El consenso fue que el lenguaje debería tener las siguientes características:
Debía usar el inglés tanto como fuera posible.
El lenguaje debía ser fácil de usar, incluso a costa de ser menos poderoso.
El diseño no debía estar demasiado restringido por los problemas de su implementación.
Una de las preocupaciones principales de la reunión fue que se debía actuar rápidamente para crear este lenguaje, ya que se estaba haciendo mucho trabajo en crear otros lenguajes comerciales.
Se tomaron decisiones tempranas para separar las declaraciones del lenguaje en dos categorías: descripción de datos y operaciones ejecutables, separando estas en distintas partes del programa.
La especificación del lenguaje, denominado COBOL60, fue publicado en abril de 1960, y posteriormente revisado en varias ocasiones, con su evolución continua hasta el presente.

COBOL introdujo varios conceptos novedosos como el verbo DEFINE, utilizado para macros, y las estructuras de datos jerárquicas, que han sido incluidas en la mayoría de los lenguajes imperativos diseñados desde entonces. Su división de datos era la parte fuerte, ideal para tareas contables, pero su división de procedimientos era débil inicialmente, careciendo de funciones y soporte para subprogramas con parámetros.

COBOL fue el primer lenguaje de programación cuyo uso fue obligatorio por parte del Departamento de Defensa. A pesar de sus méritos, COBOL probablemente no habría sobrevivido sin ese mandato. El bajo rendimiento de los primeros compiladores simplemente hacía que el

lenguaje fuera demasiado costoso de usar.
Eventualmente, los compiladores se volvieron más eficientes y las computadoras se volvieron mucho más rápidas y baratas. Juntos, estos factores permitieron que COBOL tuviera éxito, dentro y fuera del DoD.

▼ Evaluación de ALGOL 60.

▼ Nombre una persona que ha tenido una gran influencia en el desarrollo del:

El modelo imperativo: John Backus

El modelo funcional: John McCarthy

La orientación a objetos: Alan Key

El modelo de programación lógica: Robert Kowalski

▼ Evaluación de LISP.

▼ Define formalmente un DFA, un NFA, cuales son las diferencias entre ambos?

Un DFA M se compone de un alfabeto $\sum$, un conjunto de estados S, una función de transición $T : S \times \sum \to S$, un estado inicial $s_0 \in S$ y un conjunto de estados de aceptación $A \subset S$. El lenguaje aceptado por M, escrito como $L(M)$, se define como el conjunto de cadenas de caracteres $c_1, c_2 ... c_n$ con cada $c_i \in \sum$, tal que existen estados $s_1 = T(s_0, c_1)$, $s_2 = T(s_1, c_2)$, ..., $s_n = T(s_{n-1}, c_n)$, con s_n como un elemento de A.

Un NFA M consta de un alfabeto $\sum$, un conjunto de estados S, y una función de transición $T : S \times (\sum \cup \{\epsilon\}) \to \delta(S)$, así como de un estado inicial s0 de S y un conjunto de estados de aceptación A de S. El lenguaje aceptado por M, escrito como $L(M)$, se define como el conjunto de cadenas de caracteres $c_1, c_2 ... c_n$ con cada $c_i \in \sum \cup \{\epsilon\}$, tal que existen estados s1 en $T(s_0, c_1)$, s2 en $T(s_1, c_2)$, ..., sn en (s_{n-1}, c_n), con s_n como un elemento de A.

▼ Cite y explique brevemente los criterios que afectan a la legibilidad.

▼ Explique como influye el manejo de excepciones en la confiabilidad de un lenguaje para crear programas.

La capacidad para interceptar errores en tiempo de ejecución (así como otras condiciones inusuales detectables por el programa), tomar medidas correctivas y luego continuar es una ayuda evidente para la confiabilidad. Esta funcionalidad se llama manejo de excepciones.

▼ Definición formal de DFA.

Un DFA M se compone de un alfabeto Z, un conjunto de estados S, una función de transición T:SxZ → S, un estado inicial so e S, y un conjunto de estados de aceptación A U S. El lenguaje aceptado por M, escrito como L(S), se define como el conjunto de cadenas de caracteres c1, c2, ..., cn con cada ci en Z, tal que existen estados s1 = T(s0, c1), s2 = T(s1, c2) ..... con sn como un elemento de A.

▼ Definición formal de NFA.

Un NFA M se compone de un alfabeto Z, un conjunto de estados S, una función de transición T: Sx(Zu{e} → deltaS, un estado inicial so de S, y un conjunto de estados de aceptación A de S. El lenguaje aceptado por M, escrito como L(M) se define como el conjunto de cadenas de caracteres c1, c2, ... cn con cada ci en Zue tal que existen estados s1 en T(s0, c1) ..... con sn como un elemento de A.

▼ Cuales son tres de los “language design trade-offs” señalados por Sebesta.

Un conflicto común es entre la fiabilidad y costo de ejecución. Por ejemplo, Java exige que todas las referencias a elementos de matrices se verifiquen, lo que aumenta el costo de ejecución, mientras que C no realiza estas verificaciones, haciendo que sus programas sean más rápidos pero menos fiables. Los diseñadores de Java sacrificaron la eficiencia por la fiabilidad.

Otro ejemplo es APL, que tiene un conjunto poderoso de operadores para matrices, requiriendo nuevos símbolos y permitiendo expresiones complejas y compactas. Un resultado de este alto grado de expresividad es que para aplicaciones que involucren muchas operaciones con matrices, APL es muy fácil de escribir. Otro resultado es que los programas en APL tienen una legibilidad muy pobre. Una expresión compacta y concisa tiene una cierta belleza matemática, pero es difícil de entender para cualquiera

que no sea el programador. El diseñador de APL sacrificó la legibilidad por la facilidad de escritura.

El conflicto entre facilidad de escritura y fiabilidad es común en el diseño de lenguajes. Por ejemplo, C++ permite la manipulación flexible de punteros, lo que no se incluye en Java debido a problemas potenciales de fiabilidad.

▼ Segundo Parcial

▼ 2024-2

▼ 1.1. ¿Cuál es la definición más precisa de un enlace?

a) El tiempo en que se declara una variable.

b) El tiempo durante el cual una variable tiene un valor asignado.

c) La asociación de una entidad a una propiedad.

d) La localización de una variable en memoria.

▼ 1.2. ¿Qué describe mejor el alcance de una variable?

a) La porción del programa donde la variable se puede referenciar.

b) El tiempo durante el cual la variable tiene un valor válido.

c) La cantidad de memoria asignada a la variable.

d) El tipo de dato asociado a la variable.

▼ 1.3. En el contexto de lenguajes de programación, ¿qué significa que una variable tenga un "enlace dinámico"?

a) Su valor puede cambiar durante la ejecución.

b) Su tipo es determinado en tiempo de ejecución.

c) La variable es accesible en todo el programa.

d) La variable solo puede usarse en su función de origen.

▼ 1.4. ¿Cuál es la principal ventaja del enlace estático sobre el enlace dinámico?

a) Mayor flexibilidad durante la ejecución.

b) Menor uso de memoria.

c) Facilidad para comprender y depurar el programa.

d) Capacidad para cambiar el tipo de dato en tiempo de ejecución.

▼ 1.5. ¿Qué sucede con una variable global cuando se declara una variable local con el mismo nombre dentro de un bloque?

a) Ambas variables comparten el mismo valor.

b) La variable global se sobrescribe.

c) La variable local oculta a la global dentro del bloque.

d) La variable local es ignorada y se utiliza la global.

▼ 1.6. ¿Cuáles son las dos cuestiones de diseño más importantes específicas de los tipos de cadenas de caracteres?

a) ¿Las cadenas deben ser un tipo especial de arreglo de caracteres o un tipo primitivo?

b) ¿Las cadenas deben tener longitud estática o dinámica?

c) ¿Las cadenas deben ser inmutables o mutables?

d) ¿Las cadenas deben ser manejadas mediante punteros o referencias?

▼ 1.7. ¿Cuáles son las cuestiones de diseño clave para los tipos de enumeración?

a) ¿Se permite que una constante de enumeración aparezca en más de una definición de tipo, y si es así, cómo se verifica el tipo de esa constante en el programa?

b) ¿Los valores de enumeración se convierten a enteros?

c) ¿Se convierten otros tipos a un tipo de enumeración?

d) ¿Las constantes de enumeración deben ser mutables?

e) ¿Las constantes de enumeración pueden ser utilizadas en operaciones aritméticas?

f) ¿Se permite la redefinición de constantes de enumeración dentro del mismo programa?

g) ¿El orden de declaración de las constantes afecta su valor en la enumeración?

▼ **1.8. ¿Cuáles de las siguientes son cuestiones de diseño para arrays?**

a) ¿Qué tipos son legales para los subíndices?

b) ¿Se verifican los rangos de los subíndices en las referencias a los elementos?

c) ¿Cuándo se vinculan los rangos de los subíndices?

d) ¿Cuándo tiene lugar la asignación de memoria para los arrays?

e) ¿Qué tipos de slices están permitidos, si es que se permite alguno?

f) ¿Se permiten arrays multidimensionales irregulares o rectangulares, o ambos?

▼ **1.9. ¿Cuál es una de las limitaciones del tipado fuerte en lenguajes como Java y C#?**

a) No permiten la coerción entre tipos de datos.

b) Incluyen reglas de unión que no se verifican.

c) Permiten la coerción explícita, lo que puede provocar errores de tipo.

d) No verifican los errores de tipo en tiempo de compilación.

▼ **1.10. ¿Cuál es una de las principales diferencias entre el chequeo de tipos estático y dinámico?**

a) El chequeo de tipos estático permite más flexibilidad al programador que el dinámico.

b) El chequeo de tipos dinámico detecta errores en tiempo de compilación, mientras que el estático lo hace en tiempo de ejecución.

c) El chequeo de tipos estático detecta errores en tiempo de compilación, mientras que el dinámico lo hace en tiempo de ejecución.

d) Ambos tipos de chequeo permiten la misma detección de errores, pero en diferentes momentos.

▼ 1.11. ¿Cuáles son los enfoques principales para definir la equivalencia de tipos en lenguajes de programación?

a) Equivalencia por nombre y equivalencia por estructura.

b) Equivalencia por coerción y equivalencia por conversión.

c) Equivalencia por valor y equivalencia por referencia.

d) Equivalencia por asignación y equivalencia por función.

▼ 1.12. ¿Cuál es una de las principales cuestiones de diseño para las expresiones aritméticas en los lenguajes de programación según Sebesta?

a) ¿Se permite la sobrecarga de operadores definida por el usuario?

b) ¿Cuál es el tamaño máximo de una expresión aritmética?

c) ¿Cuál es el orden de evaluación de operadores booleanos?

d) ¿Cuántos operandos pueden tener los operadores aritméticos?

▼ 1.13. ¿Cuál es la regla general de precedencia de operadores en los lenguajes imperativos comunes?

a) La multiplicación y la división tienen mayor precedencia que la exponenciación.

b) La adición y la sustracción tienen mayor precedencia que la multiplicación.

c) La exponenciación tiene la mayor precedencia, seguida por la multiplicación y la división, y luego la adición y la sustracción.

d) Los operadores unarios tienen siempre la menor precedencia.

▼ **1.14. ¿Qué es la transparencia referencial en un programa?**

a) La capacidad de una función para modificar variables globales.

b) La propiedad de que cualquier dos expresiones con el mismo valor pueden ser sustituidas entre sí sin afectar el comportamiento del programa.

c) La posibilidad de que una función dependa de variables externas al mismo tiempo que mantiene su estado.

d) La capacidad de una función para almacenar y modificar su estado interno.

▼ **1.15. ¿Cuál es una de las principales conclusiones sobre las estructuras de control en lenguajes de programación según el estudio de Böhm y Jacopini?**

a) Todos los algoritmos pueden ser expresados usando únicamente estructuras de control basadas en goto.

b) Es suficiente con tener una declaración de control que permita seleccionar entre dos caminos de ejecución y otra para iteraciones controladas lógicamente.

c) Se requiere una variedad extensa de estructuras de control para asegurar la máxima flexibilidad en los lenguajes.

d) Las estructuras de control deben permitir múltiples entradas para aumentar la legibilidad del código.

▼ **1.16. ¿Cuál es una de las principales categorías de control de iteración en las declaraciones iterativas?**

a) Control basado en excepciones.

b) Control lógico, conteo, o una combinación de ambos.

c) Control basado en la recursión.

d) Control manual a través de funciones externas.

▼ **1.17. ¿Qué significa el término "pretest" en el contexto de una declaración iterativa?**

a) Que el cuerpo del bucle se ejecuta al menos una vez antes de evaluar la condición.

b) Que la evaluación de la condición para completar el bucle ocurre antes de ejecutar el cuerpo del bucle.

c) Que la evaluación de la condición después de ejecutar el cuerpo del bucle.

d) Que la condición del bucle se evalúa en cada iteración, independientemente de su ubicación.

▼ **1.18. ¿Cuál es una de las cuestiones de diseño clave para los mecanismos de control de bucles con salidas ubicadas por el usuario?**

a) ¿El bucle debe reinicializar automáticamente después de una salida?

b) ¿El mecanismo condicional debe ser una parte integral de la salida?

c) ¿El bucle debe ser siempre de tamaño fijo?

d) ¿Se deben permitir salidas solo al final del bucle?

▼ **1.19. ¿Cuál es una ventaja de pasar datos a un subprograma a través de parámetros en lugar de acceder directamente a variables no locales?**

a) Mejora la flexibilidad del subprograma al permitirle procesar diferentes datos en cada llamada.

b) Reduce la necesidad de declarar variables locales dentro del subprograma.

c) Evita la necesidad de declarar variables globales en el programa.

d) Permite que el subprograma acceda a datos en cualquier parte del programa sin restricciones.

▼ **1.20. ¿Cuál de las siguientes afirmaciones describe correctamente un diseño importante en el manejo de excepciones?**

a) Los manejadores de excepciones deben ser locales a la unidad donde se detecta la excepción.

b) La continuación después de la ejecución de un manejador de excepciones siempre debe ser en el punto de la excepción.

c) La propagación de excepciones permite que solo un manejador maneje excepciones en varias unidades del programa.

d) Las excepciones predefinidas deben ser siempre manejadas automáticamente por el sistema.

▼ Por temas

▼ ¿Qué tipo de datos tiene un orden secuencial?

A) Enumeración

B) Subrango

C) Ordinales

D) Puntero

▼ Describe una situación en la cual el operador de suma en un lenguaje de programación no sería asociativo.

▼ U5 - NAMES, BINDINGS, TYPE CHECKING, AND SCOPES

▼ ¿Cuál es una característica clave de las variables en la programación imperativa en relación con la arquitectura de la computadora von Neumann?

a) Simulan el comportamiento del procesador

b) Son abstracciones de las celdas de memoria de la máquina

c) Representan las instrucciones del programa

d) Actúan como puertas lógicas en circuitos

▼ En la programación funcional, las expresiones nombradas se asemejan a:

a) Variables mutables

b) Punteros

c) Constantes nombradas

d) Argumentos de funciones

▼ Explique qué es la dirección de una variable.

La dirección de una variable es la dirección de memoria de la máquina con la que está asociada. La dirección de una variable también se conoce como “l-value”, ya que la dirección es lo que se necesita cuando una variable aparece en el lado izquierdo de una asignación.

▼ En términos de aliasing, ¿Qué afirmación es cierta respecto a la fiabilidad?

a) El aliasing es deseable para la claridad del código

b) El aliasing aumenta la fiabilidad al permitir múltiples nombres para una variable

c) La restricción del aliasing aumenta la fiabilidad del programa

d) El aliasing no tiene impacto en la fiabilidad del programa

▼ Cuáles son los 6 atributos de las variables? Explique brevemente cada uno de ellos.

Nombre: Es una cadena de caracteres utilizada para identificar a la variable.

Dirección: La dirección de una variable es la dirección de memoria de la máquina con la que está asociada.

Tipo: El tipo de una variable determina el rango de valores que la variable puede almacenar y el conjunto de operaciones que están definidas para los valores de este tipo.

Valor: Es el contenido de la o las celdas de memoria asociadas con la variable.

Tiempo de vida: Es el periodo durante el cual una variable esta asociada a una ubicación de memoria especifica.

Alcance: Es el rango de sentencias en las que una variable es visible, es decir, donde puede ser referenciada o asignada.

▼ ¿Cuál es una desventaja principal del uso de variables estáticas en la programación?

a) No pueden ser utilizadas en subprogramas recursivos

b) Requieren una mayor cantidad de memoria

c) Son menos eficientes en tiempo de ejecución

d) No son compatibles con la mayoría de los lenguajes modernos

▼ Explique de forma detallada la diferencia entre variables de pila dinámicas (stack-dynamic) y variables de montón explícito dinámicas (explicit heap-dynamic). Proporcione ejemplos de lenguajes que utilizan cada tipo y discuta las ventajas y desventajas de cada tipo de variable.

▼ ¿Qué caracteriza a una variable 'heap-dinámica explícita' en programación?

a) Su ciclo de vida y tipo son dinámicamente vinculados durante la ejecución

b) Son liberadas automáticamente por el recolector de basura

c) Son vinculadas durante la compilación del programa

d) Tienen un alcance global y estático en el programa

▼ Cuáles son los tipos de enlaces de almacenamiento (storage bindings). Explique detalladamente cada uno de

ellos.

Variable estática: Son las variables en las que tanto su almacenamiento como su tipo se asocian en tiempo de compilación. Estos no pueden cambiar a lo largo de la ejecución del programa. Son utilizadas normalmente en variables globales. Su principal ventaja es la eficiencia, ya que permiten direccionamiento directo y no generan sobrecarga en tiempo de ejecución.

Variable dinámica en pila: Las variables dinámicas en pila tienen su tipo asociado en tiempo de compilación, pero su almacenamiento se asocia cuando se elabora su declaración, o sea, cuando la ejecución alcanza el código asociado a la declaración. Por ejemplo, las variables que se declaran al inicio de un método se elaboran cuando se llama al método, y se desasignan cuando el método completa su ejecución. Son útiles para programas recursivos, ya que cada copia activa del subprograma tiene su propia versión de variables locales.

Variable dinámica explicita en el heap: Una variable dinámica explícita en el heap es una variable sin nombre la cual se asigna y desasigna explícitamente mediante instrucciones en tiempo de ejecución. Estas variables solo pueden referenciarse a través de punteros o referencias. Por ejemplo, en C++

```
int *intnode;
intnode = new int;
```

las siguientes declaraciones crean una variable dinámica explicita en el heap.

Estas variables son útiles para construir estructuras dinámicas como listas enlazadas o árboles, las cuales necesitan crecer o disminuirse durante la ejecución.

Variable dinámica implícita en el heap: Son las variables en las que se asocian al almacenamiento en el heap solo cuando se les asigna un valor. De hecho, todos sus atributos se asocian cada vez que se les asigna un valor. Su ventaja es que ofrecen el mayor grado de flexibilidad, permitiendo escribir código altamente genérico.

▼ En el contexto de los lenguajes de programación, ¿Cómo se define el 'alcance' de una variable?

a) La duración de la memoria asignada a la variable

b) El rango de operaciones posibles en la variable

c) El rango de instrucciones en las que la variable es visible

d) La compatibilidad de la variable con diferentes tipos de datos

▼ Defina static scoping y dynamic scoping. Cuáles son las reglas que se aplican en cada caso.

El alcance estático permite determinar el alcance de una variable de manera estática, es decir, antes de la ejecución, permitiendo a los lectores y compiladores determinar el tipo de cada variable simplemente examinando su código fuente. Esto se realiza buscando la variable referenciada en el subprograma actual, y si no se encuentra, la búsqueda se extiende a sus padres estáticos. Uno de sus problemas en que en la mayoría de los casos permite más acceso a variables y a subprogramas del que es necesario.

El alcance dinámico se basa en la secuencia de llamadas de subprogramas, no en su relación espacial entre sí. Por lo tanto, se determina en tiempo de ejecución. Esto se logra buscando la variable referenciada en las declaraciones locales. Si no se encuentra, la búsqueda continúa en el padre dinámico de esa función, y así sucesivamente, hasta que se encuentra la declaración a la variable. El alcance dinámico hace que los programas sean más difíciles de leer y menos confiables, ya que una variable no local puede referirse a diferentes variables según la secuencia de llamadas. Además, no se pueden verificar los tipos de las variables no locales en tiempo de compilación.

▼ Qué es un entorno de referencia?

El entorno de referencia de una sentencia es el conjunto de variables visibles en esa sentencia. Este entorno se necesita al compilar la sentencia para manejar las referencias a variables no locales en tiempo de ejecución.

El entorno de referencia de una sentencia en un lenguaje estático consiste de las variables de su alcance local y las visibles en alcances ascendentes.

En lenguajes de alcance dinámico el entorno de referencia de una sentencia está compuesto por las variables declaradas localmente, más las variables de todos los subprogramas que estén actualmente activos.

▼ Explique detalladamente qué es el entorno de referencia de una sentencia, cómo se usa?

El entorno de referencia de una sentencia es el conjunto de variables que son visibles para esa sentencia. Se utiliza al momento de compilar una sentencia para manejar referencias a variables de otro alcance en tiempo de ejecución.

▼ En que consiste el entorno de referencia en un lenguaje de alcance estático?

En un lenguaje estático, el entorno de referencia de una sentencia consta de las variables locales y las variables en alcances estáticos, o sea, en sus ancestros estáticos.

▼ Porqué se requiere del entorno de referencia al momento de la compilación?

El entrono de referencia al momento de la compilación se necesita para manejar las referencias a variables de otro alcance en tiempo de ejecución.

▼ U6 - DATA TYPES

▼ ¿Qué es una coerción?

A) Error de tipo en un programa

B) Conversión explícita de un tipo de dato a otro

C) Conversión implícita de un tipo de dato a otro

D) Función especial en programación

▼ ¿Qué mecanismos se utilizan para almacenar enteros negativos en una computadora?

Notación signo-magnitud, en donde el bit de signo se establece para indicar negativo, y el resto de la cadena de bits representa el valor

absoluto del número.

Notación complemento a dos, en donde la representación de un entero negativo se forma tomando el complemento lógico de la versión positiva del número y sumando uno.

Notación complemento a uno, el entero negativo se almacena como el complemento lógico de su valor absoluto. Desventaja de que tiene dos representaciones de cero.

▼ ¿Cuál es una ventaja de los tipos de datos decimales?

A) Ocupan menos memoria que los tipos enteros

B) Son más rápidos que los tipos enteros

C) Permiten una representación más precisa de números fraccionarios

D) No tienen desventajas

▼ Qué tipo de datos puede ser utilizado para representar de manera precisa el valor e (la base para el logaritmo natural) o el valor de pi? Presente una breve descripción del tipo de datos que corresponde.

No hay ningún tipo de dato con el que se pueda almacenar correctamente los valores e ni pi, ya que estos números tienen infinitos decimales, lo cual sería imposible guardar en las computadoras debido a la naturaleza finita de la memoria.

▼ ¿Qué lenguajes permiten subíndices negativos para acceder a elementos de un arreglo?

A) C++

B) Python

C) Java

D) C#

▼ Cuál es la diferencia entre un static array y un heap dynamic array?

En un arreglo estático, tanto el rango de subíndices como la asignación de almacenamiento son enlazados de manera estática, es decir, antes del tiempo te ejecución. Estos se mantienen fijos y no pueden cambiar durante la ejecución del programa.

En un arreglo dinámico en el montón, los rangos de subíndice y la asignación de almacenamiento son dinámicos y pueden cambiar cualquier cantidad de veces durante la vida útil del arreglo.

▼ Un array se puede almacenar en row-major-order como se hace en C++ o en column-major-order como ocurre en FORTRAN. Escriba una función de acceso para row-major-order.

a[i, j] = a[0, 0] + (i*n+j) * element_size

donde i es el número de filas por encima del elemento deseado, n el tamaño de la fila, y j el número de elementos a la izquierda de la columna deseada

▼ ¿Cuál es el problema de diseño en arrays asociativos?

La única cuestión de diseño es cual es la forma de referenciar a sus elementos?

▼ Cite las cuestiones de diseño de los registros.

- Cual es la forma sintáctica de las referencias a los campos?
- Se permiten referencias elípticas?

▼ ¿Cuál es la principal diferencia entre un registro y una tupla?

A) Los registros tienen campos con nombres y las tuplas no

B) Las tuplas tienen campos con nombres y los registros no

C) Los registros pueden contener diferentes tipos de datos, pero las tuplas no

D) No hay diferencia

▼ Defina unión, unión libre y unión discriminada.

A) Unión: varios tipos en el mismo espacio de memoria, Unión libre: unión sin discriminador de tipo, Unión discriminada: unión con discriminador de tipo

B) Unión: unión con discriminador de tipo, Unión libre: unión sin discriminador de tipo, Unión discriminada: varios tipos en el mismo espacio de memoria

C) Unión: unión sin discriminador de tipo, Unión libre: varios tipos en el mismo espacio de memoria, Unión discriminada: unión con discriminador de tipo

D) Unión: varios tipos en el mismo espacio de memoria, Unión libre: unión con discriminador de tipo, Unión discriminada: unión sin discriminador de tipo

▼ ¿Cuáles son los dos problemas comunes con los punteros?

A) Punteros rápidos y punteros lentos

B) Punteros nulos y punteros colgantes

C) Punteros largos y punteros cortos

D) Punteros seguros y punteros inseguros

▼ Defina error de tipo.

A) Ocurre cuando se realiza una operación en un tipo de dato que es compatible con esa operación

B) Ocurre cuando se realiza una operación en un tipo de dato que no es compatible con esa operación

C) Ocurre cuando se utiliza un tipo de dato incorrecto en una declaración

D) Ocurre cuando se declara una variable con un tipo incorrecto

▼ Cite y explique detalladamente los dos enfoques existentes para la equivalencia de tipo.

La equivalencia de tipos por nombre significa que dos variables tienen tipos equivalentes si están definidas ya sean en la misma

declaración o en declaraciones que usan el mismo nombre de tipo.

La equivalencia de tipos por estructura significa que dos variables tienen tipos equivalentes si sus tipos tienen estructuras idénticas.

▼ Defina named type equivalence y structure type equivalence

La equivalencia de tipos por nombre significa que dos variables tienen tipos equivalentes si están definidas ya sean en la misma declaración o en declaraciones que usan el mismo nombre de tipo.

La equivalencia de tipos por estructura significa que dos variables tienen tipos equivalentes si sus tipos tienen estructuras idénticas.

▼ U7 - EXPRESIONES Y SENTENCIAS DE ASIGNACIÓN

▼ Cite las 6 cuestiones de diseño en las expresiones aritméticas según Sebesta.

- Cuales son las reglas de precedencia de operadores?
- Cuales son las reglas de asociatividad a de operadores?
- Cual es el orden de evaluación de los operandos?
- Existen restricciones en los efectos secundarios en la evaluación de operandos?
- Permite el lenguaje la sobrecarga de operadores definida por el usuario?
- Qué mezcla de tipos esta permitida en las expresiones?

▼ Cuáles son las 6 cuestiones de diseño fundamentales de las expresiones aritméticas según Sebesta? Explique brevemente cada una de ellas.

- **Cuales son las reglas de precedencia de los operadores?** Las reglas de precedencia de los operadores definen el orden en el que se evalúan los operadores en una expresión, con base en la jerarquía de operadores que establezca el diseñador del lenguaje.

- **Cuales son las reglas de asociatividad de los operadores?** Las reglas de asociatividad de los operadores definen en que orden se evaluarán los operadores en una expresión en la que hayan dos operadores adyacentes con la misma precedencia.
- **Cual es el orden de evaluación de los operandos?** El orden de evaluación de los operandos es importante cuando alguno de los operandos de un operador tiene efectos secundarios, ya que que un operando se evalúe primero puede afectar el valor del otro.
- **Existen restricciones en los efectos secundarios de la evaluación de operandos?** Hay dos formas de restringir los efectos secundarios, uno, prohibiendo por completo los efectos secundarios funcionales, por ejemplo, prohibiendo el acceso a variables globales en funciones, y dos, imponiendo un orden de evaluación de los operandos, lo que puede limitar técnicas de optimización utilizadas por los compiladores, ya que algunas de estas implican reordenar las evaluaciones de los operandos.
- **Permite el lenguaje la sobrecarga de operadores definida por el usuario?** La sobrecarga de operadores significa que un un operador puede tener múltiples significados dependiendo del contexto, la sobrecarga de operadores puede beneficiar la legibilidad cuando se utiliza de manera sensata, pero cuando no puede ser bastante perjudicial para esta.
- **Que mezcla de tipos esta permitida en las expresiones?** Una expresión podría tener operandos de diferentes tipos, los lenguajes que permiten tales expresiones, deben definir restricciones y convenciones para las conversiones implícitas de los operados.

▼ ¿Cuál de las siguientes opciones mejor define la precedencia de operadores?

A) Orden en el que se ejecutan los programas

B) Orden en el que se evalúan los operadores en una expresión

C) Cantidad de operadores en una expresión

D) Velocidad de ejecución de los operadores

▼ Precedencia y asociatividad de operadores.

▼ Explica en al menos 120 palabras, qué es la transparencia referencial y cómo se relaciona la transparencia referencial con los efectos colaterales [side effects] de las funciones?

Un programa tiene la propiedad de transparencia referencial si cualquier dos expresiones en el programa que tengan el mismo valor pueden ser sustituidas entre sí sin afectar la acción del programa.

Su relación con los efectos secundarios se demuestra en el siguiente ejemplo:

```
result1 = fun(a) + b;
temp = fun(a);
result2 = temp + b;
```

Si fun no tiene efectos secundarios, entonces result1 y result2 serán iguales, ya que las expresiones asignadas a ellas son equivalente. Sin embargo, si fun tiene el efecto secundario de sumar 1 a b, entonces result1 no sería igual a result2. Así que ese efecto secundario viola la transparencia referencial del programa.

▼ Side-effects de una función. Ejemplo.

Un efecto secundario de una función ocurre cuando la función cambia uno de sus parámetros o una variable global, por ejemplo

```
int fun(a)
    a = a + 1
    return 5
int main
    a = 5
    result = a + fun(a)
```

el siguiente código proporcionaría valores distintos dependiendo del orden de evaluación de los operadores, por ejemplo, si se evalúa de

izquierda a derecha, la variable result tendría un valor de 10, pero si se evaluará de derecha a izquierda, tendría un valor de 11

▼ ¿Qué es la transparencia referencial?

A) Propiedad de las funciones que dependen del estado del programa

B) Propiedad de las funciones que siempre producen el mismo resultado para los mismos argumentos

C) Capacidad de una función para cambiar su entorno

D) Tipo especial de variable

▼ Qué es la transparencia referencial y cómo se relaciona con los efectos colaterales? (Referential Transparency and side Effects)

Un programa tiene la propiedad de transparencia referencial si cualquier dos expresiones en el programa que tengan el mismo valor pueden ser sustituidas entre sí sin afectar la acción del programa.

Su relación con los efectos secundarios se demuestra en el siguiente ejemplo:

```
result1 = fun(a) + b;
```

```
temp = fun(a);
```

```
result2 = temp + b;
```

Si fun no tiene efectos secundarios, entonces result1 y result2 serán iguales, ya que las expresiones asignadas a ellas son equivalente. Sin embargo, si fun tiene el efecto secundario de sumar 1 a b, entonces result1 no sería igual a result2. Así que ese efecto secundario viola la transparencia referencial del programa.

▼ Define conversión narrowing y conversión widening. Mencione un ejemplo de cada conversión.

una conversión de reducción (narrowing) convierte un valor a un tipo cuyo rango de valores posibles es más pequeño. Por ejemplo, convertir un double a un float es una conversión de reducción, ya que el rango de double es mucho mayor que el de float.

Una conversión de ampliación (widening) convierte un valor a un tipo que puede incluir al menos aproximaciones de todos los valores del tipo original. Por ejemplo, convertir un int a un float.

▼ ¿Qué es una coerción?

A) Error de tipo en un programa

B) Conversión explícita de un tipo de dato a otro

C) Conversión implícita de un tipo de dato a otro

D) Función especial en programación

▼ ¿Qué es un cast?

A) Conversión implícita de un tipo de dato a otro

B) Conversión explícita de un tipo de dato a otro realizada por el programador

C) Error de tipo en un programa

D) Función especial en programación

▼ Por qué los diseñadores de los lenguajes no están de acuerdo con respecto a la coerción en expresiones aritméticas?

Aquellos diseñadores de los lenguajes que se oponen a un amplio rango de coerciones están preocupados por los problemas de confiabilidad que pueden surgir de estas coerciones, ya que reducen los beneficios de la comprobación de tipos.

▼ ¿Crees que la eliminación de operadores sobrecargados en tu lenguaje de programación favorito sería beneficiosa? ¿Por qué sí o por qué no?

▼ Presenta tus propios argumentos a favor y en contra de permitir expresiones aritméticas de modo mixto.

▼ U8 - ESTRUCTURAS DE CONTROL A NIVEL DE SENTENCIA

▼ Cuáles son las únicas estructuras de control necesarias?

Aunque solamente una declaración de control, un goto seleccionable, es mínimamente suficiente, según Böhm y Jacopini se necesitan al menos dos mecanismos adicionales para que las computaciones en los programas sean flexibles y potentes: una para elegir dos caminos de flujo de control y otra para iteraciones controladas lógicamente.

▼ Cuáles son tres consideraciones de diseño relacionadas a los Selectores de dos vías según Sebesta?

- Cual es la forma y el tipo de la expresión que controla la selección?
- Como se especifican las clausulas then y else?
- Como debe especificarse el significado de los selectores anidados?

▼ Cite las cuestiones de diseño de los selectores de 2 vías.

- Cual es la forma y el tipo de la expresión que controla la selección?
- Como se especifican las clausulas then y else?
- Como debe especificarse el significado de los selectores anidados?

▼ ¿Cuál es la forma general de un selector de dos vías?

if *expresion_de_control*

then *clausula*

else *clausula*

▼ Cuáles son cinco consideraciones de diseño relacionadas a los selectores múltiples según Sebesta?

- Cual es la forma y el tipo de la expresión que controla la selección?
- Cómo se especifican los segmentos seleccionables?
- Está restringido el flujo de ejecución para incluir solo un único segmento seleccionable?
- Como se especifican los valores de los casos?
- Como deben manejarse, si es que deben de hacerlo, los valores de expresión de selector no representados?

▼ ¿Qué mecanismo sigue un compilador cuando el número de casos en una declaración de selección es 10 o más para optimizar el tiempo requerido de ejecución?

El compilador construye una tabla hash de las etiquetas de los segmentos, lo que resultaría en un tiempo aproximadamente igual y corto de elegir cualquiera de los segmentos seleccionables. Si el lenguaje permite rangos de valores para las expresiones de los casos, como en Ruby, una tabla hash no es lo adecuado. En este caso es mejor realizar una búsqueda binaria de valores de casos y direcciones de segmentos.

▼ Cuáles son las categorías en las que pueden clasificarse las sentencias de control de iteraciones? Cuales son las preguntas que sirven para definir estas clasificaciones?

Las principales categorías en las que pueden clasificarse las sentencias de control por iteraciones son los bucles controlados por contador y los bucles controlados lógicamente.

Las dos básicas de diseño que sirven para definir estas categorías son:

- Como se controla la iteración?
- Donde debe aparecer el mecanismo de control en la iteración?

▼ Cuáles son tres consideraciones de diseño relacionadas a las iteraciones controladas por contador

en Sebesta?

- Cual es el tipo y el alcance de la variable de bucle?
- Debería ser legal que la variable de bucle o los parámetros del bucle sean modificados dentro del bucle, y de ser así, el cambio afecta al control del bucle?
- Debe evaluarse los parámetros de bucle solo una vez o una vez por cada iteración?

▼ ¿En qué sentido es la declaración `for` de C más flexible que en muchos otros lenguajes?

A) C permite múltiples declaraciones en la inicialización.

B) C solo permite enteros en la declaración `for`.

C) C permite cualquier expresión en las secciones de inicialización, condición y actualización.

D) C no permite bucles `for`.

E) C permite cualquier expresión en las secciones de inicialización, condición y actualización.

▼ ¿Cómo un lenguaje funcional implementa la repetición?

Ya que los lenguajes funcionales no tienen variables, lo cual es importante en los lenguajes imperativos para los bucles controlados por contador, controlan la repetición con recursión.

Los bucles con contador pueden ser simulados de la siguiente manera:

El contador puede ser un parámetro de una función que ejecuta repetidamente el cuerpo del bucle, el cual puede estar especificado en una segunda función enviada a la función de bucle como un parámetro. Así, una función de bucle toma la función del cuerpo y el número de repeticiones como parámetros.

Un ejemplo de un bucle controlado por contador en F#:

```
let rec forLoop loopBody reps =
    if reps <= 0 then
        ()
    else:
        loopBody()
        forLoop loopBody, (reps-1);;
```

▼ ¿Cuáles son los problemas de diseño para las declaraciones de bucle controladas lógicamente?

- Debe el control ser de preevaluación o postevaluación?
- Debe el bucle controlado lógicamente ser una forma especial de un bucle con contador o una instrucción separada?

▼ ¿Qué ventaja tiene la declaración `break` de Java sobre la declaración `break` de C?

A) Java no tiene declaración `break`.

B) Java permite salir de bloques anidados específicos con etiquetas.

C) Java no permite usar `break` en bloques de `switch`.

D) No hay diferencia entre Java y C en cuanto a la declaración `break`.

▼ Seleccione la opción que describe correctamente tres situaciones en las que se necesita una declaración de bucle combinada de conteo y lógica.

A) 1) Cuando necesitas repetir una acción un número específico de veces, pero también necesitas una condición específica para continuar. 2) Cuando estás procesando elementos de una lista y necesitas parar cuando encuentres un valor específico. 3) Cuando estás esperando a que un recurso esté disponible, pero solo quieres intentarlo por un tiempo limitado.

B) 1) Cuando necesitas repetir una acción un número infinito de veces. 2) Cuando solo necesitas una declaración de bucle basada en una condición lógica. 3) Cuando quieres iterar a través de todos los elementos de una colección sin ninguna condición específica.

C) 1) Cuando estás realizando una búsqueda y quieres detenerte cuando encuentres el elemento o después de un cierto número de intentos. 2) Cuando estás intentando conectarte a un servidor y quieres reintentar un número específico de veces antes de rendirte. 3) Cuando estás validando entrada del usuario y quieres darle un número limitado de intentos para ingresar datos válidos.

D) 1) Cuando solo necesitas un bucle que se ejecute un número fijo de veces. 2) Cuando estás realizando una operación que no requiere condiciones de parada. 3) Cuando estás realizando cálculos matemáticos que no involucran iteraciones.

▼ Mencione un uso común inadecuado de los resultados de Böhm y Jacopini.

Un uso indebido del resultado de Böhm y Jacopini es argumentar en contra de la inclusión de otras estructuras de control más allá de las selecciones y los bucles lógicos con preevaluación. Ningún lenguaje de uso general ha dado ese paso todavía, esto debido al impacto negativo en la capacidad de escritura y legibilidad que esto conllevaría. Los programas escritos solo con selecciones y bucles lógicos con preevaluación generalmente tienen una escritura menos natural, son más complejos y, por lo tanto, más difíciles de escribir y leer.

▼ U9 - SUBPROGRAMAS

▼ ¿Cuáles son las 3 características generales de los subprogramas?

Cada subprograma tiene un punto único de entrada.

La unidad de programa que llamó, se mantendrá suspendida hasta que el subprograma llamado terminé, por lo que solo habrá un subprograma en ejecución en un momento dado.

El control siempre regresa a quien hizo la llamada cuando el subprograma termina.

▼ ¿Qué es una definición de subprograma?

Una definición de subprograma describe la interfaz y las acciones de la abstracción del subprograma.

▼ Cuáles son las tres características de una subrutina? Cuáles son 11 consideraciones de diseño principales de las subrutinas?

Características

Los subprogramas deben tener un único punto de entrada.

Cuando un programa llame a un subprograma, este programa debe quedar suspendido hasta que el subprograma termine su ejecución, por lo que habrá solo un subprograma en ejecución en un momento dado.

El control siempre regresa a quien hizo la llamada cuando el subprograma termina.

Cuestiones de Diseño

- Qué método o métodos de paso de parámetros se utilizan?
- Se verifican los tipos de los parámetros reales con respecto a los tipos de los parámetros formales?
- Las variables locales se asignan de manera estática o dinámica?
- Pueden aparecer definiciones de subprogramas dentro de otras definiciones de subprogramas?
- Si los subprogramas pueden pasarse como parámetros y pueden anidarse, cuál es el entorno de referencia de un subprograma pasado como parámetro?
- Se permiten efectos secundarios funcionales?
- Qué tipos de valores pueden devolver las funciones?
- Cuántos valores pueden devolver las funciones?
- Pueden sobrecargarse los subprogramas?
- Pueden ser genéricos los subprogramas?
- Si el lenguaje permite subprogramas anidados, se admiten closures?

▼ Cite y explique brevemente los 5 tipos de paso de parámetros? Explica en al menos 50 palabras en cada caso.

Paso por Valor:

Paso por Resultado:

Paso por Valor-Resultado:

Paso por Referencia:

Paso por Nombre:

▼ Describe correctamente el problema de pasar arrays multidimensionales como parámetros.

La función de mapeo de almacenamiento de una matriz dada con elementos de tamaño 1 es la siguiente:

```
address[i, j] = address[0, 0] + i*n + j
```

Con n siendo el tamaño de fila, o sea, el número de columnas. Como está función debe construirse antes de que el subprograma sea llamado, debe conocer el número de columnas al momento de compilar el subprograma. Para esto, el número de columnas debe incluirse en el parámetro formal.

El problema con esto es que no permite que un programador escriba una función que acepte matrices con diferentes números de columnas, reduciendo así la flexibilidad.

▼ Cuáles son las complicaciones que existen con el paso de parámetros que son subprogramas? Cuáles son las opciones que se han desarrollado para lidiar con la cuestión de la determinación del entorno en referencia, explique brevemente. Explica en al menos 50 palabras.

Para el paso de parámetros que son subprogramas, surgen dos complicaciones:

Primero, la verificación de tipos de los parámetros del subprograma que se pasó como parámetro. En C y C++, las funciones no se pueden pasar directamente como parámetros, pero si se pueden

pasar punteros a funciones. El tipo de puntero a función incluye el protocolo de la función. Debido a que el protocolo incluye todos los tipos de los parámetros, estos parámetros pueden ser verificados en cuanto a tipos.

La segunda complicación se da cuando hay subprogramas anidados, y es cual es el entorno de referencia que debe usarse para ejecutar el subprograma pasado. Para esto hay tres opciones:

Usar el entorno de la instrucción de llamada al subprograma pasado (shallow binding)

Usar el entorno de la definición del subprograma pasado (deep binding)

Usar el entorno de la instrucción que pasó el subprograma como parámetro real (ad hoc binding)

▼ Defina shallow y deep binding para entornos de referencia de subprogramas que han sido pasados como parámetros. Explique detalladamente.

Shallow binding se refiere a utilizar el entorno de referencia de la instrucción de llamada al subprograma pasado como parámetro.

Deep binding se refiere a usar el entorno de referencia de la definición del subprograma pasado.

▼ ¿Qué es el enlace ad hoc?

Ad hoc binding es una solución al problema de cual entorno de referencia debe usarse para un subprograma pasado como parámetro. En programas con ad hoc binding, el entorno utilizado por un subprograma que fue pasado como parámetro es el entorno de la llamada que pasó el subprograma como un parámetro

▼ Qué es polimorfismo de subtipo?

El polimorfismo de subtipo significa que una variable de tipo T puede acceder a cualquier objeto de tipo T o de cualquier tipo derivado de T.

▼ U10 - IMPLEMENTACIÓN DE SUBPROGRAMAS

▼ Los pasos que se siguen al ser llamado un procedimiento.

En un subprograma simple, es decir, en el que los su programas no pueden anidarse y todas las variables son locales estáticas, los pasos al realizar una llamada son:

1. Guardar el estado de ejecución del programa actual.
2. Calcular y pasar los parámetros.
3. Pasar la dirección de retorno al subprograma llamado.
4. Transferir el control al subprograma llamado.

Pasos a realizar en la llamada de un subprograma en un lenguaje con variables locales dinámicas en pila

Programa que llama

1. Crear una instancia del registro de activación
2. Guardar el estado de ejecución del programa actual
3. Calcular y pasar los parámetros
4. Pasar la dirección de retorno al subprograma llamado
5. Transferir el control al subprograma llamado

Subprograma llamado

1. Guardar el antiguo EP en la pila como el enlace dinámico y crear el nuevo EP
2. Asignar las variables locales

▼ Los pasos que se siguen cuando el procedimiento sale de escena.

Los pasos que se siguen cuando un subprograma simple sale de la escena son los siguientes:

1. Si hay parámetros de modo salida o entrada salida, se copian los valores actuales de los parámetros formales a los parámetros

reales

2. Si el subprograma es una función, el valor funcional se coloca en un lugar accesible para el programa que hizo la llamada.
3. Se restaura el estado de ejecución del programa que llama.
4. El control se transfiere de vuelta al programa que llama.

Lenguajes con variables locales dinámicas en pila

Subprograma llamado

1. Si hay parámetros en modo de salida o de valor resultado, mover los valores de los parámetros formales a los parámetros reales.
2. Si el subprograma es una función, colocar el valor funcional en un lugar accesible para el subprograma que llamo
3. Restaurar el puntero de la pila configurándolo al valor del EP actual menos uno y configurar el EP al antiguo enlace dinámico
4. Restaurar el estado de ejecución del programa que llama
5. Transferir el control de vuelta al programa que llama

▼ ¿Qué debe almacenarse para la vinculación (linkage) a un subprograma?

Para la vinculación de un subprograma debe almacenarse:

1. Información sobre el estado de ejecución del programa que llama.
2. Parámetros.
3. Valores devueltos por funciones.
4. Dirección de retorno.
5. Variables temporales utilizadas por los subprograma.

▼ ¿Cuál es la diferencia entre un registro de activación y una instancia de registro de activación?

Un registro de activación es el formato de la parte no relacionada con el código de un subprograma, la cual describe datos que son relevantes solo durante la activación o ejecución del subprograma. Una instancia de registro de activación es un ejemplo concreto de un registro de activación, el cual se crea cada que un subprograma es llamado.

▼ Dibuje un registro de activación. Describa brevemente sus partes

Subprograma Simple

Variables Locales	Variables temporales utilizadas dentro del subprograma llamado
Parámetros	Son los valores o direcciones proporcionados por el programa que llama
Dirección de Retorno	puntero a la instrucción que sigue a llamada

Lenguajes con variables locales dinámicas en pila

Variables Locales	Variables temporales utilizadas dentro del subprograma llamado
Parámetros	Son los valores o direcciones proporcionados por el programa que llama
Enlace Dinámico	Puntero a la base de la instancia de la instancia del registro de activación del que llama
Dirección de Retorno	puntero a la instrucción que sigue a llamada

▼ ¿Cómo se representan las referencias a las variables en el método static-chain?

Las referencias a variables no locales en el método static-chain puede ser representada por un par de ordenado de enteros (chain_offset, local_offset), donde chain offset es el número de enlaces a la instancia correcta de registro de activación y local offset es desplazamiento necesario para llegar a la variable desde el inicio del registro de activación.

▼ Describe correctamente el método shallow-access para implementar el alcance dinámico.

▼ U14 - EXCEPTION HANDLING and EVENT HANDLING

▼ Cuestiones de diseño para el manejo de excepciones.

- Cómo y donde se especifican los manejadores de excepciones y cuál es su alcance?
- Cómo se vincula una ocurrencia a un manejador de excepciones?
- Se puede pasar información sobre una excepción al manejador?
- Dónde continúa la ejecución, si es que continúa, después de que un manejador de excepciones completa su ejecución?
- Se proporciona alguna forma de finalización?
- Cómo se especifican las excepciones definidas por el usuario?
- Si hay excepciones predefinidas, deberían existir manejadores de excepciones predeterminados para los programas que no proporcionen los suyos?
- Pueden levantarse explícitamente las excepciones predefinidas?
- Se tratan los errores detectables por hardware como excepciones que pueden ser manejadas?
- Hay alguna excepción predefinida?

▼ Louden U7 - AMBIENTES DE EJECUCIÓN

▼ Dibuje la organización del almacenamiento en tiempo de ejecución. Describa brevemente sus partes.

área de código	
área global/estatica	
pila	
espacio libre	
heap	

▼ Describa un ambiente de ejecución basado en pila sin procedimientos locales.

▼ Final

▼ 2023-2.1

▼ ¿Qué característica contribuye significativamente a la legibilidad de un lenguaje de programación?

a) Multiplicidad de características

b) Sobrecarga de operadores

c) Simplicidad general

d) Uso de tipos de datos inadecuados

▼ ¿Cómo afecta la ortogonalidad a la legibilidad de un lenguaje de programación?

a) Disminuyendo la cantidad de excepciones a las reglas del lenguaje

b) Aumentando la complejidad del lenguaje

c) Limitando el número de tipos de datos disponibles

d) Creando una sobrecarga de operadores

▼ En el diseño de la sintaxis, ¿qué factor mejora la legibilidad?

a) Uso de palabras especiales que pueden ser nombres de variables

b) Diseño de sentencias que sugieran su propósito

c) Uso de un gran número de palabras reservadas

d) Permitir múltiples formas de realizar una misma operación

▼ En relación con la simplicidad y ortogonalidad, ¿qué afirmación es correcta para mejorar la escritura?

a) Una alta cantidad de constructos primitivos es preferible

b) La ortogonalidad excesiva puede ser perjudicial

c) La simplicidad se logra con un gran número de reglas

d) La consistencia en las reglas de combinación no es importante

▼ **¿Qué característica del lenguaje de programación es crucial para la fiabilidad?**

a) La presencia de múltiples formas de realizar una operación

b) La comprobación de tipos durante la compilación

c) El uso de operadores sobrecargados

d) La limitación en los tipos de datos disponibles

▼ **En términos de aliasing, ¿qué afirmación es cierta respecto a la fiabilidad?**

a) El aliasing es deseable para la claridad del código

b) El aliasing aumenta la fiabilidad al permitir múltiples nombres para una variable

c) La restricción del aliasing aumenta la fiabilidad

d) El aliasing no tiene impacto en la fiabilidad del programa

▼ **¿Cuál es una característica clave de las variables en la programación imperativa en relación con la arquitectura de la computadora von Neumann?**

a) Simulan el comportamiento del procesador

b) Son abstracciones de las celdas de memoria de la máquina

c) Representan las instrucciones del programa

d) Actúan como puertas lógicas en circuitos

▼ **En el contexto de los lenguajes de programación, ¿cómo se define el 'alcance' de una variable?**

a) La duración de la memoria asignada a la variable

b) El rango de operaciones posibles en la variable

c) El rango de instrucciones en las que la variable es visible

d) La compatibilidad de la variable con diferentes tipos de datos

▼ **En la programación funcional, las expresiones nombradas se asemejan a:**

a) Variables mutables

b) Punteros

c) Constantes nombradas

d) Argumentos de funciones

▼ **¿Cuál es una desventaja principal del uso de variables estáticas en la programación?**

a) No pueden ser utilizadas en subprogramas recursivos

b) Requieren una mayor cantidad de memoria

c) Son menos eficientes en tiempo de ejecución

d) No son compatibles con la mayoría de los lenguajes modernos

▼ **¿Qué caracteriza a una variable 'heap-dinámica explícita' en programación?**

a) Su ciclo de vida y tipo son dinámicamente vinculados durante la ejecución

b) Son gestionadas automáticamente por el recolector de basura

c) Deben ser liberadas explícitamente en el código

d) Tienen un alcance global y estático en el programa

▼ **¿Cómo afecta la sensibilidad (case-sensitive) al caso en los nombres de variables a la legibilidad en la programación?**

a) Mejora significativamente la legibilidad

b) La sensibilidad al caso es irrelevante

c) Disminuye la legibilidad

d) Solo es útil en lenguajes de bajo nivel

▼ 2018-2.1

▼ **Factores que determinan el costo según Sebesta**

1. Capacitación de programadores
2. Escritura de programas
3. Compilación de programas

4. Ejecución de programas
5. Costo del sistema de implementación
6. Costo de la baja confiabilidad
7. Mantenimiento de programas

▼ 6 Razones para estudiar estructura de los lenguajes

1. Aumento en la capacidad para expresar ideas
2. Mejorar la capacidad de elegir lenguajes apropiados
3. Aumento de la capacidad para aprender nuevos lenguajes
4. Comprensión de la implementación
5. Mejor uso de los lenguajes que ya se conocen
6. Avance general de la computación

▼ Cuestiones de diseño de sentencias de elección doble

1. Cual es la forma y el tipo de expresión que controla la selección?
2. Como se especifican las clausulas then else?
3. Como debe especificarse el significado de los selectores anidados?

▼ Cuestiones de diseño de punteros

1. Cual es el alcance y la vida útil de una variable puntero?
2. Cual es la vida útil de una variable dinámica en el heap?
3. Están los punteros restringidos en cuanto al tipo de valor al que pueden apuntar?
4. Se utilizan los punteros para la gestión de almacenamiento dinámico, la dirección indirecta, o ambos?
5. Debe el lenguaje soportar tipos de puntero, tipos de referencia, o ambos?

▼ Completa

- El primer lenguaje orientado a objetos fue smalltalk, cumplió con todas las características de POO las cuales son herencia, clases y

métodos. Desarrollado principalmente por Alan Key en la década de 70.

- Las tuplas son estructuras similares a los registros pero los campos no tienen nombre.
- Las uniones pueden clasificarle en discriminadas y libres.
- Métodos de recolección de basura en los lenguaje de programación pueden ser lápidas y lock-and-keys que funcionan _____ y _____ respectivamente.
- El display es una alternativa a las cadenas estáticas.

▼ **Explique brevemente las características de las funciones.**

▼ **Defina formalmente un lenguaje regular.**

Una expresión regular es una de las siguientes:

1. Una expresión regular básica constituida por solo un carácter a, donde a proviene de un alfabeto Z de caracteres legales; el metacarácter e; o el metacarácter o. En el primer caso, L(a)={a}; en el segundo L(e)={e}; en el tercer L(o)={}.
2. Una expresión de la forma r|s, donde r y s son expresiones regulares. En este caso L(r|s) = L(r)UL(s)
3. Una expresión de la forma rs, donde r y son expresiones regulares. En este caso, L(rs) = L(r)L(s).
4. Una expresión de la forma r*, donde r es una expresión regular. En este caso L(r*) = L(r)*.
5. Una expresión de la forma (r), donde r es una expresión regular. En este caso. L((r)) = L(r). De este modo los parentesis no cambian el lenguaje solo se utilizan para ajustar la precedencia de las operaciones.

▼ **Explique detalladamente el proceso de llamada y retorno de una función.**

Llamada

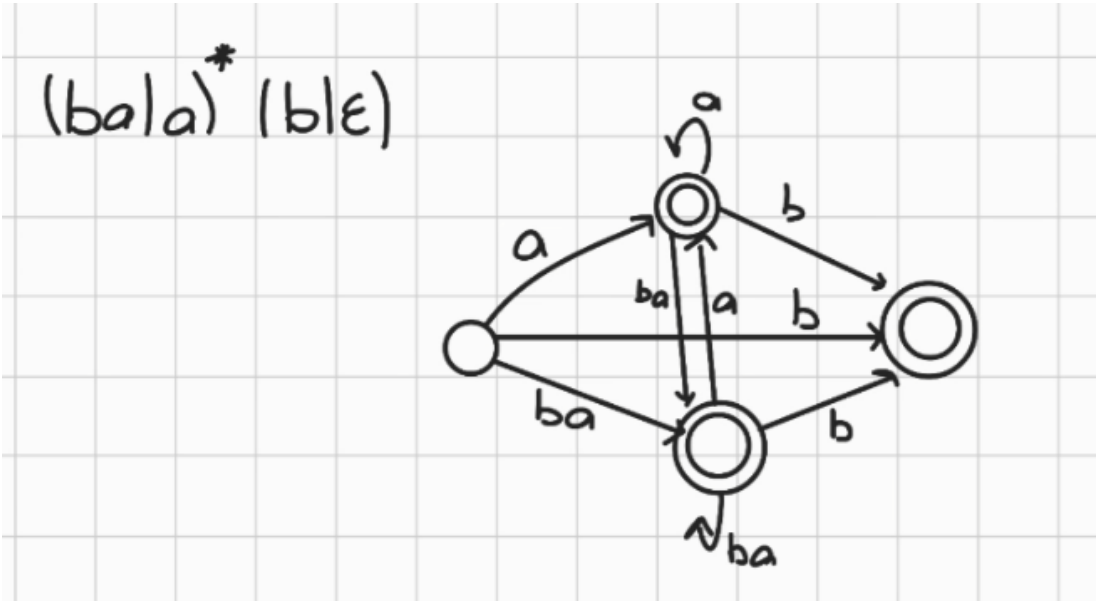
1. Guardar el estado de ejecución del programa actual.

2. Calcular y pasar los parámetros.
3. Pasar la dirección de retorno del subprograma llamado.
4. Transferir el control al subprograma llamado.

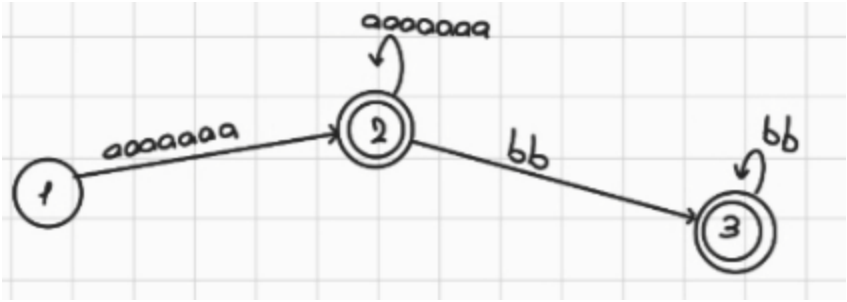
Retorno

1. Si hay parámetros de modo salida o entrada/salida, se copian los valores actuales de los parámetros formales a los parámetros reales.
2. Si el subprograma es una función, el valor funcional se coloca en un lugar accesible para el subprograma que hizo la llamada.
3. Se restaura el estado de ejecución del programa que llama.
4. El control se transfiere de vuelta al programa que llama.

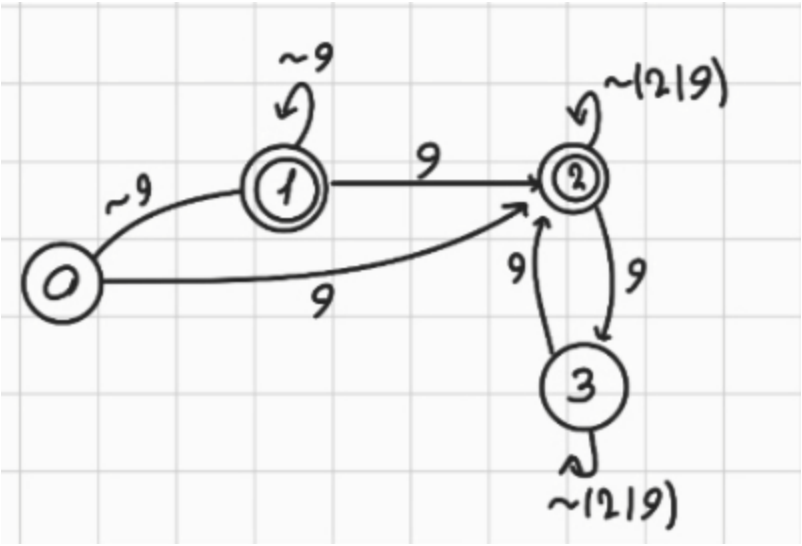
▼ **Diseñe un dfa para el un lenguaje (a,b) de todas las cadenas que no contengan dos b´s consecutivas.**



▼ **Diseñe un dfa para cadenas que tengan múltiplo de 7 de a´s seguido de múltiplo de 2 de b´s.**



▼ **Diseñe un dfa para cadenas (0,1,2..,9) de manera a que todos los 2's se presenten primero antes que los 9´s y la cantidad de 9´s sea impar.**



▼ 2020-2.1

Cada tema deberá ser explicado en mínimo 300 palabras

▼ **¿Por qué piensa Usted que es importante que los lenguajes de programación contengan un conjunto enriquecido de sentencias? Explicar claramente en qué consiste, ventajas y desventajas. Proveer ejemplos significativos.**

▼ **¿Por qué piensa Usted que es importante que los lenguajes de programación provean un conjunto de tipos de datos? Proporcione ejemplos significativos.**

▼ **¿Qué argumentos puede presentar para la idea de un lenguaje único para todos los dominios de programación? Fundamente su opinión.**

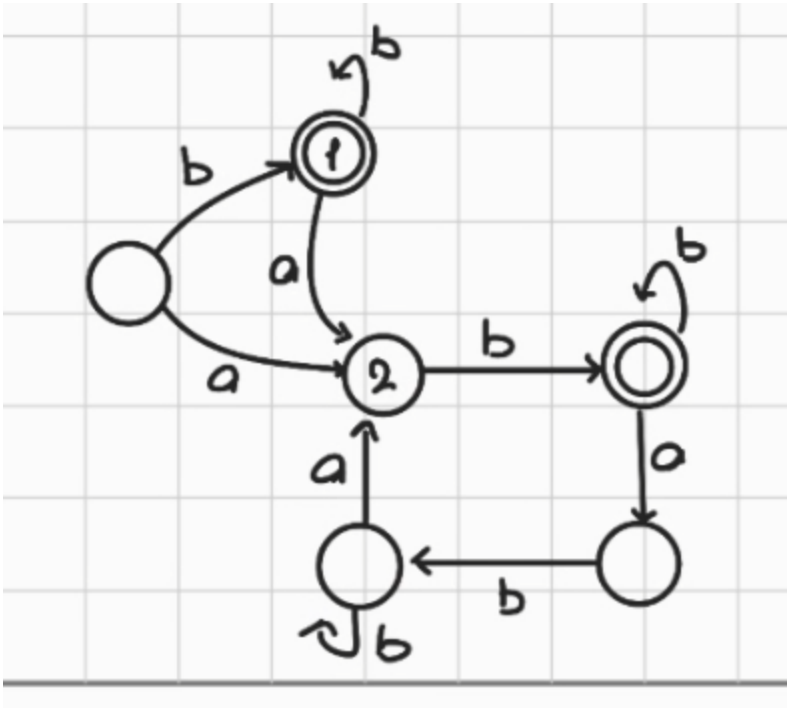
▼ **Escribe una evaluación comparativa sobre la sentencia iterativa de los lenguajes Python y R, utilizando los criterios de evaluación indicados**

por Sebesta. Explicar claramente en base a todos los criterios, ventajas y desventajas. Proveer ejemplos significativos.

▼ **Explique detalladamente cómo las reglas de coerción pueden disminuir los beneficios del tipado fuerte. Explicar claramente en qué consiste, ventajas y desventajas. Proveer ejemplos significativos.**

▼ 2016-2.R

▼ **Dado un alfabeto {a, b}, construya un DFA capaz de reconocer el siguiente lenguaje: capaz de reconocer un número impar de a y cada letra a es seguida por al menos una b.**



▼ **Explique brevemente los dos factores que constituyen las influencias principales en el diseño de lenguajes de programación.**

La arquitectura de Von Neumann ha tenido un profundo impacto en el diseño de lenguajes de programación durante los últimos 60 años. Estos lenguajes se llaman lenguajes imperativos. La mayoría de computadoras construidas desde 1940 utilizan la arquitectura de Von Neumann.

Otro factor influyente ha sido la evolución de las metodologías de diseño de programas. A finales de los 70, el cambio del costo principal de la computación del hardware al software impulsó un análisis profundo del desarrollo de software y el diseño de lenguajes. En los años 70, surgieron metodologías como el diseño top-down y el refinamiento escalonado.
A finales de los 70, se produjo un cambio de metodologías de diseño orientadas a procedimientos hacia metodologías orientadas a datos.
A principios de los 80, el diseño orientado a objetos emergió como un avance importante.

▼ **Cuales son tres de los "language design trade-offs" señalados por Sebesta.**

▼ **Explique detalladamente qué es el entorno de referencia de una sentencia, como se usa?**

El entorno de referencia de una sentencia es el conjunto de variables visibles en esa sentencia. El entorno de referencia en un lenguaje de alcance estático incluye las variables de su alcance local y las visibles en alcances ascendentes. El entorno de referencia de una sentencia en un lenguaje de alcance dinámico está compuesto por las variables declaradas localmente, más las variables de todos los otros subprogramas que están actualmente activos.

▼ **Cuál es la relación entre el alcance y el tiempo de vida? Explique una situación cuando existe esta aparente relación y cuándo no lo hay.**

▼ **Cuáles son los tipos de enlaces de almacenamiento (storage bindings). Explique detalladamente cada uno de ellos.**

▼ **Cite y explique detalladamente los dos enfoques existentes para la equivalencia de tipo.**

▼ **Qué es la transparencia referencial y cómo se relaciona con los efectos colaterales?**

▼ **Mencione un uso común inadecuado de los resultados de Böhm y Jacopini.**

▼ **Defina shallow y deep binding para entornos de referencia de subprogramas que han sido pasados como parámetros. Explique detalladamente.**

▼ **Qué debe ser almacenado para la vinculación a un subprograma?**

▼ No se

▼ **Dibuje un DFA que corresponda a la expresión regular ϕ**

▼ **Para qué se utilizan los registros de activación de procedimiento? Cuáles son sus partes?**

▼ **Cuál es otro nombre con el cual se conocen a los registros de activación?**

▼ **Cuáles son los registros típicos que se utilizan para mantener las activaciones de procedimiento?**

▼ 2016-2.1

▼ **Para un alfabeto {a, b}, construya un DFA capaz de reconocer el siguiente lenguaje: L = {w | w tiene exactamente dos a y al menos dos b}**

▼ **Realice una tabla con los criterios de evaluación para los lenguajes de programación y los criterios que afectan.**

▼ **Explique que es el static scope y el dinamic scope. Cuáles son las ventajas y desventajas de cada uno?**

▼ **Cuál es la relación entre el alcance y el tiempo de vida? Explique una situación cuando existe esta aparente relación y cuándo no lo hay.**

▼ **Cuáles son las ventajas y desventajas del enlace de tipos dinámico. Explique detalladamente.**

▼ **Cuando se dice que un lenguaje es fuertemente tipado, cuáles son los requisitos?**

▼ **Cuáles son las 6 consideraciones de diseño principales para las expresiones aritméticas?**

▼ **Qué es la transparencia referencial y cómo se relaciona con los efectos colaterales?**

▼ **Defina shallow y deep binding para entornos de referencia de subprogramas que han sido pasados como parámetros. Explique detalladamente.**

▼ **Qué debe ser almacenado para la vinculación a un subprograma?**

▼ 2016-2.2

▼ **Defina static scoping y dynamic scoping. Cuáles son las reglas que se aplican en cada caso.**

▼ **Escriba una función de acceso para row-major-order.**

▼ **Defina named type equivalence y structure type equivalence.**

▼ **Cite las cuestiones de diseño de los selectores de 2 vías.**

▼ **Qué es un heap-dynamic array? Cuál es su ventaja? Cuál es su desventaja?**

▼ **Cuál es la diferencia entre un static array y un heap dynamic array?**

▼ **Cite las 6 cuestiones de diseño de las expresiones aritméticas según Sebesta.**

▼ **Cite las cuestiones de diseño de los registros.**

▼ **Dibuje la organización del almacenamiento en tiempo de ejecución. Describa brevemente sus partes.**

▼ **Dibuje un registro de activación. Describa brevemente sus partes.**

▼ **Pasos que siguen al ser llamado un procedimiento.**

▼ **Pasos que siguen cuando el procedimiento sale de la escena.**

▼ **Describa un ambiente de ejecución basado en pila sin procedimientos locales.**