Introducción y Evaluación de Lenguajes

Criterios para evaluar los lenguajes de programación:

Simplicidad y legibilidad

Escribir programas fáciles de leer, escribir también que sea fácil aprenderlo y enseñarlo. Ejemplos que atentan contra este criterio son:

- Muchos componentes elementales.
- Conocer subconjuntos de componentes.
- Que tenga el mismo concepto semántico, pero su sintaxis es distinta.
- Que difiera el concepto semántico, pero la sintaxis sea la misma
- Abuso de operadores sobrecargados.

Claridad en las ligaduras

Los elementos de los lenguajes se ligan a sus atributos o propiedades en diferentes momentos de forma clara, ejemplos son:

- Definición del lenguaje
- Implementación del lenguaje
- En escritura del programa
- Compilación del programación
- Cargado del programa
- En ejecución

Confiabilidad

Se relaciona con la seguridad como el chequeo de tipos para encontrar errores en etapas tempranas y excepciones para evitar situaciones anómalas en tiempo de ejecución, actuando y tratando dichas anomalías para poder continuar.

Soporte

Debería ser accesible, multiplataforma y que tenga recursos para familiarizarse con el como documentación, tutoriales, cursos, etc.

Abstracción

Capacidad de definir y usar estructuras u operaciones complicadas de manera que sea posible ignorar muchos de los detalles. Abstracción de procesos y de datos.

Ortogonalidad

Significa que un conjunto pequeño de constructores primitivos puede ser combinado en número relativamente pequeño a la hora de construir estructuras de control y datos. Cada combinación es legal y con sentido.

Eficiencia

Tiempo y espacio, esfuerzo humano, optimizable.

Sintaxis

Definición de sintaxis

Conjunto de reglas que definen como componer letras, dígitos y otros caracteres (llamadas “palabra”) para formar sentencias y programas.

Ayuda al programador a escribir programas correctos sintácticamente. Establece reglas para que el programador pueda comunicarse con el procesador y debe contemplar soluciones a características tales como la legibilidad, verificabilidad, traducción, falta de ambigüedad.

Elementos de la sintaxis

Alfabeto o conjunto de caracteres

Es importante que la implementación de los caracteres soporte los que use el lenguaje.

Identificadores

Cadena de letras y dígitos que deben comenzar con una letra.

Operadores

Suma, resta, etc. Cada lenguaje puede definir diferentes operadores para la misma funcionalidad (`^` puntero en Pascal contra `*` puntero en C).

Palabra clave y reservada

La palabra clave tiene un significado dentro de un contexto, la palabra reservada es un palabra clave que ademas no puede ser utilizada por el programador con identificador de otra entidad.

Comentarios y uso de blancos

Reglas léxicas y sintácticas

Reglas léxicas

Conjunto de reglas que definen como formar una palabra a través de caracteres del alfabeto.

Reglas sintácticas

Conjunto de reglas que definen como formar sentencias y expresiones utilizando palabras.

Tipos de sintaxis

Abstracta (Estructura)

La estructura del código de un lenguaje es igual a la de otro lenguaje. El “if” en Java es estructuralmente igual al `if` en C.

Concreta (Parte léxica)

Un lenguaje utiliza las mismas palabras que otro lenguaje para determinar el mismo contexto. Como las llaves en C y Java.

Pragmática (Uso practico)

Relacionado con la legibilidad y cuantos usos tiene en la practica. En Ada toda estructura de control termina con un `end`, en C una estructura de control puede no llevar llaves si solo tiene una sentencia.

Formas para definir una sintaxis

Lenguaje natural

Gramática

Conjunto de reglas finita que define un conjunto infinito de posibles sentencias validas en el lenguaje, una gramatica G esta formada por 4-tupla, G = (N, T, S, P).

`N`: Conjunto de no terminales;

`T`: Conjunto de terminales;

`S`: No terminal distinguido;

`P`: Conjunto de producciones;

Características:

- Gramáticas ambiguas: Una gramática es ambigua si se puede derivar en mas de una forma.
- Subgramáticas: Gramáticas que están compuestas por una o mas gramáticas se conocen como subgramáticas.

Tipos de gramáticas:

- Libres de contexto: Estas no realizan un análisis del contexto. `C = A + B;` es sintacticamente correcto, pero `A` es carácter y `B` es un numero, semanticamente no seria correcto.
- Sensibles al contexto: Estas realizan un análisis, ocasionando que problemas como los del ejemplo anterior no sean permitidos.

BNF (Backus Naun Form)

Notación formal para describir la sintaxis que define reglas por medio de producciones.

`<numeros>`: No terminal;

`a | b | ... | z`: Terminales;

`<letras> ::= a | ... | z`: Producción;

EBNF (Extended Backus Naun Form):

- Remplaza las producciones recursivas por producciones iteradas.
- `{ ... }*`: Repite 0 o mas veces; `{ ... }+`: Repite 1 o mas veces; `[ ... ]`: Opcional;
- (... | ...) : Terminales;
- Ej: `<enteroconsigno> ::= [ ( “+” | “-” ) ]{ <digito> }+`

Arboles sintácticos (Árbol de derivación)

Resultado del método de análisis “parsing” para determinara que un string es una cadena sintacticamente valida, hay 2 maneras de construirlo.

- **Top-Down:** Desde arriba hacia abajo y puede ser leído de izquierda a derecha o viceversa.
- **Bottom-Up:** Desde abajo hacia arriba y puede ser leído de izquierda a derecha o viceversa.

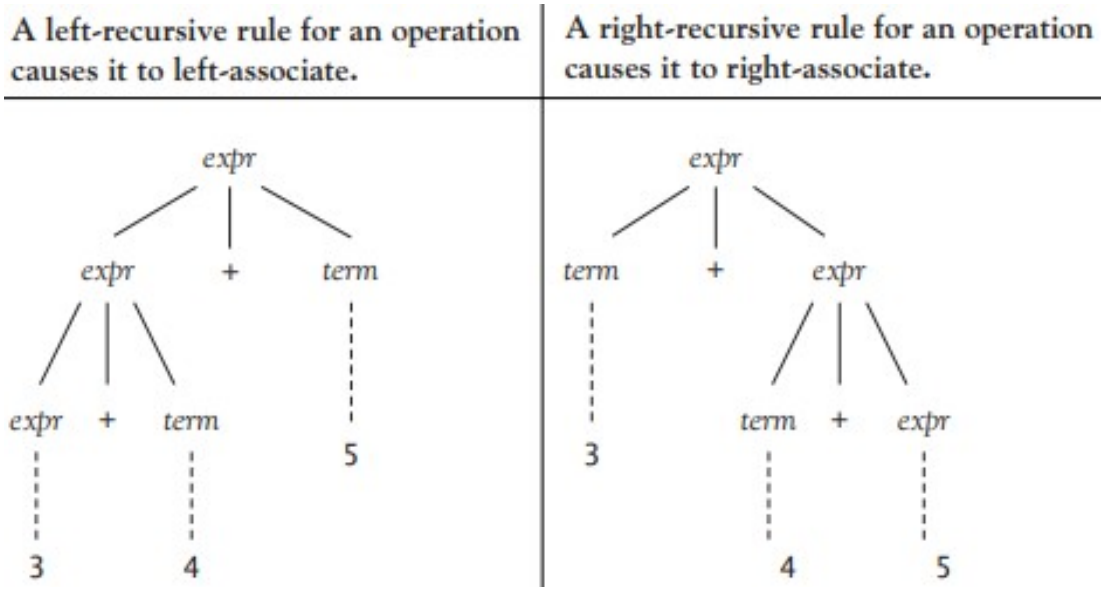
Producciones Recursivas

Son las que hacen el conjunto de sentencias descripto sea infinito.

Ej: <numero> ::= <dígito><numero>

- **Regla recursiva por la izquierda:** <numero> ::= <numero><dígito>
- **Regla recursiva por la derecha:** <numero> ::= <dígito><numero>

Dependiendo de la regla y operadores puede ocasionar problemas, como la resta “8-4-2”, por izquierda el resultado seria ((8-4)-2) = 2, por derecha seria (8-(4-2)) = 6



Programming Languages - Kenneth C.Louden Third Edition - Capitulo 6.4 - Hoja 219

Diagrama Sintáctico (Conway)

Es un grafo sintáctico o carta sintáctica, cada diagrama es una regla o producción que tiene una entrada, una salida, el camino determina el análisis, para que una sentencia sea valida debe haber un camino desde la entrada hasta la salida que la describa.

Semántica

Definición de semántica

La semántica describe el significado de los símbolos, palabras y frases de un lenguaje ya sea lenguaje natural o lenguaje informático que es sintácticamente válido.

Semántica estática

Definición

Se las llama así porque el análisis para el chequeo se hace en compilación (antes de la ejecución). No esta relacionada con el significado de la ejecución del programa, está más relacionado con las formas válidas (con la sintaxis). Este análisis se ubica entre el análisis sintáctico y el análisis de la semántica dinámica.

BNF/EBNF no sirven para detectar errores son gramáticas libres de contexto (no se meten con el significado si con las formas), para describir la sintaxis y la semántica estática formalmente sirven las denominadas gramáticas de atributos.

Gramática de atributos

Son gramáticas sensibles al contexto (GSC), si se relacionan con el significado. La usan los compiladores, antes de la ejecución. Generalmente resuelven los aspectos de la sintaxis y semántica estática. El funcionamiento es el siguiente.

Las construcciones de un lenguaje se les asocia un atributo mediante el símbolo asociado de la gramática (terminal o no terminal) para poder detectar errores, el atributo puede ser un valor de una variable, un tipo de la variable, una expresión, lugar que ocupa una variable, dígitos significativos de un numero, etc. Estos valores de los atributos se obtienen mediante “ecuaciones o reglas semánticas” asociadas a las producciones gramaticales.

Las reglas sintácticas son similares a BNF. Las reglas semánticas (ecuaciones) permiten detectar errores y obtener valores de atributos. Los atributos están directamente relacionados a los símbolos gramaticales. Las gramáticas de atributos se suelen expresar en forma de tabla para obtener el valor del atributo.

Buscando en la tabla:

- Busca una regla gramatical en la tabla y busca el atributo para terminales y no terminales asociado a la regla.
- Si lo encuentra el atributo recupera mediante una ecuación el valor del atributo.

De la ejecución de las ecuaciones:

- Ingresan los símbolos a la tabla de símbolos.
- Chequean variables iguales, combinaciones no permitidas, semántica estática y dan mensaje de error.
- Controlar tipo y variables de igual tipo.
- Generar un código para el siguiente paso.

Semántica dinámica

Definición

Es la que describe el significado de ejecutar las diferentes construcciones del lenguaje de programación. Su efecto se ve durante la ejecución del programa. Influirá la interacción con el usuario y errores de la programación. Los programas solo se podrán ejecutar si correctos en sintaxis y semántica estática.

Formas de describir la semántica dinámica

Sirven para comprobar la ejecución, la exactitud de un lenguaje, comparar funcionalidades de distintos programas. Se pueden usar combinados, no todos sirven para todos los tipos de lenguajes de programación.

Formales y complejas:

Semántica axiomática:

Se considera al programa como una maquina de estados donde cada instrucción provoca un cambio de estado, parte de un axioma que sirve para verificar estados y condiciones a probar, los constructores del lenguaje se formalizan para describir como su ejecución provoca un cambio de estado, la notación utilizada es el “Calculo de predicados (Lógica del primer orden)”, el estado se describe con un predicado y este predicado describe los valores de las variables en ese estado, existe el estado anterior y posterior al estado actual, cada sentencia se precede y continua con una expresión lógica que describe las restricciones y relaciones entre datos.

Operadores	Precondición	Sentencia	Postcondición
a: Dividendo c: Cosciente x: Divisor r: Resto	x != 0	y/x	(y==x*c+r) && (r < x)

Semántica denotacional:

Basado en funciones recursivas y modelos matemáticos, lo que mejora la exactitud de los resultados a costa de legibilidad, a diferencia del axiomático este utiliza funciones recursivas en vez de predicados para definir sus estados, define una correspondencia entre los constructores sintácticos y sus significados, describe la dependencia funcional entre los resultados de la ejecución y sus datos iniciales, lo que hace es buscar producciones que se aproximen a las producciones sintácticas.

Tabla sintáctica y significados	Convertir binario a decimal: 110
<Nbin>::= 0 | 1 | <Nbin>0 | <Nbin>1	FNbin(<Nbin>0)
	2* FNbin(<Nbin>1)
FNbin(0) = 0	2*[2*FNbin(1)+1]
FNbin(1) = 1	2*[2*1+1]
FNbin(<Nbin>0) = 2 * FNbin(**<Nbin>**)	2*[3]
FNbin(<Nbin>1) = 2 * FNbin(**<Nbin>**) + 1	6

<Nbin>: Es la siguiente regla sintáctica en la cadena.

No formal:

Semántica operacional:

El significado de un programa se describe mediante otro lenguaje de bajo nivel implementado sobre una máquina abstracta, cuando se ejecuta una sentencia del lenguaje de programación los cambios de estado de la máquina abstracta definen su significado, es un método informal porque se basa en otro lenguaje de bajo nivel y puede llevar a errores.

Lenguajes	Maquina Abstracta

for i := pri to ul do
begin
....
end
	i = pri
	lazo if > ul goto sal
	....
	i := i + 1
	goto lazo
	sal ....

Procesamiento de un lenguaje

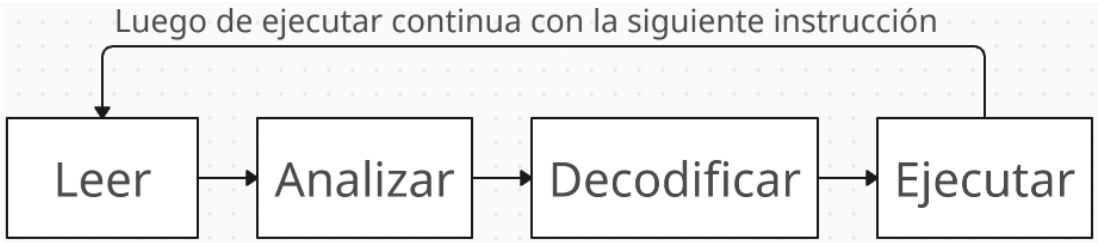
Como las computadoras eran programadas utilizando 0´s y 1´s y esto era complejo y propenso a errores al programar la solución fue remplazar estas secuencias de bits por código conocido como “código mnemotécnico”. Con esto surge la solución:

- **Lenguaje ensamblador:** Utiliza estos códigos mnemotécnicos para programar.
- **Programa ensamblador:** Traduce los códigos mnemotécnicos a lenguaje de maquina.

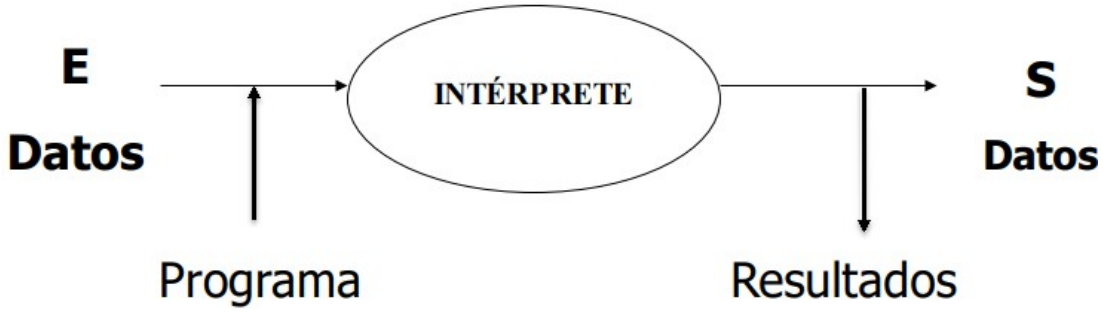
Esta solución tenia sus problemas, ya que los programas ensamblados no eran intercambiables entre familia de procesadores o por el código de maquina, un procesador con otra versiones podía tener set incompatibles a su vez de agregar nuevas instrucciones, para solucionar esto se introdujo los programas traductores del lenguaje (Interpretadores y Compiladores)

Interpretación

Hay un programa que está escrito en lenguaje de programación interpretado y este es traducido por un programa llamado interprete al momento de ejecución. El proceso que realiza cuando se ejecuta sobre cada una de las sentencias del programa es:



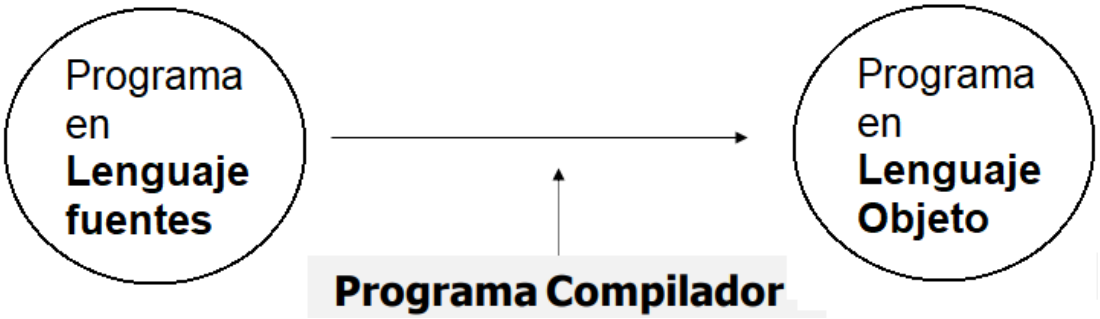
Pasa por ciertas instrucciones, no por todas, según sea la ejecución. Por cada posible acción hay un subprograma en lenguaje de máquina que ejecuta esa acción. La interpretación se realiza llamando a estos subprogramas en la secuencia adecuada hasta generar el resultado de la ejecución. Un interprete realiza la siguiente secuencia de acciones:



- Obtiene la próxima sentencia
- Determina la acción a ejecutar
- Ejecuta la acción

Compilación

Tenemos un programa escrito en lenguaje de programación de alto nivel el cual es traducido a lenguaje de maquina por un programa llamado compilador antes de la ejecución, este pasa por todas sus instrucciones. El código que se genera se guarda y se puede volver a utilizar ya compilado.



El compilador toma todo el programa escrito en un lenguaje de alto nivel que llamamos lenguaje fuente antes de su ejecución. Luego de la compilación va a generar: O un lenguaje objeto que es generalmente el ejecutable (en lenguaje de máquina) o un lenguaje de nivel intermedio (lenguaje ensamblador).

Comparación entre Compilador e Intérprete

Interprete	Compilación
No necesariamente recorre todo el código, lo que reduce su espacio	Recorre todo el código, lo que incrementa su espacio en memoria
Por cada sentencia que pasa realiza el proceso de decodificación para determinar las operaciones y sus operandos, las sentencias iterativas se decodifican tantas veces como sean necesarias. La velocidad de proceso se puede ver afectada por esto	Pasa por todas las sentencias, no repite lazos, traduce todo una única vez y genera un código objeto ya compilado. La velocidad de compilar dependerá del tamaño del código
Más lento en ejecución. Se repite el proceso cada vez que se ejecuta el mismo programa o pasa por las mismas instrucciones	Más rápido ejecutar desde el punto de vista del hardware porque ya está en un lenguaje de más bajo nivel
Para ser ejecutado en otra máquina se necesita tener si o si el intérprete instalado y el programa fuente será publico.	Está listo para ser ejecutado. Ya compilado es más eficiente. Detectá más errores al pasar por todas las sentencias. Tarda más en compilar porque se verifica todo previamente. El programa fuente no será publico
Cada sentencia se deja en la forma original y las instrucciones interpretadas necesarias para ejecutarlas se almacenan en los subprogramas del interprete en memoria	Una sentencia puede ocupar decenas o centenas de sentencias de máquina al pasar a código objeto
Tablas de símbolos, variables y otros se generan cuando se usan en forma dinámica	Cosas cómo tablas de símbolos, variables, etc. se generan siempre se usen o no
Las sentencias del código fuente pueden ser relacionadas directamente con la sentencia en ejecución entonces se puede ubicar donde se produjo el error, haciendo mas fácil detectarlo por donde pasan en la ejecución y corregirlo	Se pierde la referencia entre el código fuente y el código objeto, ocasionando que sea casi imposible ubicar el error

Combinación de técnicas de traducción

En la práctica muchos lenguajes combinan ambas técnicas para sacar provecho a cada una, los compiladores y los intérpretes se diferencian en como reportan los errores de ejecución, combinándolos genera una mejor detección de errores en el programa.

Interpreto → Compilo:

- Se utiliza el intérprete en la etapa de desarrollo para facilitar el diagnóstico de errores.
- Con el programa validado se compila para generar un código objeto más eficiente.

Compilo → Interpreto:

- Se hace traducción a un código intermedio a bajo nivel que luego se interpretará.
- Sirve para generar código portable, es decir, código fácil de transferir a diferentes máquinas y con diferentes arquitecturas.

Funcionamiento de los compiladores

Etapa de análisis (Vinculado al código fuente):

- **Análisis léxico:** Análisis a nivel de palabra, divide el programa en elementos/categorías, analiza el tipo de cada uno para determinar si son tokens válidos, lleva una tabla para la especificación del análisis léxico con cada categoría, conjunto de atributos y acciones asociadas, si una entrada no coincide con ninguna categoría genera un error.
- **Análisis sintáctico:** Análisis a nivel sentencia/estructura, usa los tokens del análisis léxico, se identifican estructuras de sentencias, declaraciones y expresiones, utiliza una gramática para construir el árbol sintáctico.
- **Análisis semántico:** Procesa las estructuras sintácticas, agrega información implicita, realiza comprobación de tipos, duplicados, nombres

Etapa de síntesis (Opcional)

- Optimización y generación de código final (vinculado a las características del código objeto, hardware y arquitectura).

Semántica Operacional

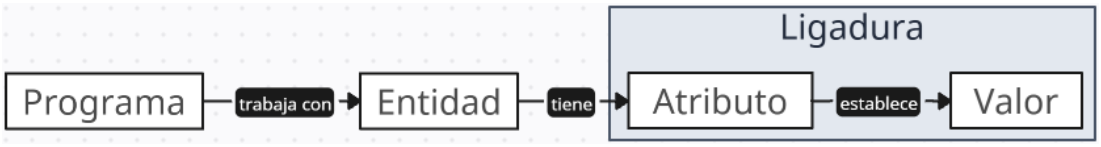
Propósito

La semántica operacional es fundamental para diversos aspectos del proceso de desarrollo de software, como el diseño de lenguajes de programación, la verificación de programas y la comprensión de cómo se ejecutan los programas en un nivel más bajo. Se describen las siguientes entidades principales.

Entidades	Atributos
Variables	Nombre, tipo, área de memoria, etc
Rutinas	Nombre, parámetros formales y reales, convección de pasaje de parámetros, etc
Sentencias	Acción asociada
Descriptor: Lugar(repositorio) donde se almacena la información de los atributos anteriores.	

Ligadura

La ligadura (Binding) es un concepto central en la definición de la semántica de los lenguajes de programación. Es el momento en el que el atributo se asocia con un valor.



Diferencias entre los distintos lenguajes: Cantidad de entidades, cantidad de atributos a ligar, momento de ligadura (Estática o Dinámica), Estabilidad (Puede modificarse).

	Tipos de ligadura	
	Estática	**Dinámica**
Momento de ligadura	Se establece antes de la ejecución. Ej. Definición/Implementación del lenguaje, Compilación/Traducción	Se establece durante la ejecución
Estabilidad	No se puede modificar	Se puede modificar durante ejecución de acuerdo a alguna regla especifica del lenguaje. (Las constantes son una excepción, se liga en ejecución y no puede modificarse)

Variables

Una Variable es una 5-tupla que tiene los atributos: Nombre, Alcance, Tipo, L-Valor y R-Valor

Nombre

Cadena de caracteres que se usa para referenciar a la variable. (identificador).

- **Aspectos de diseño:** Sensitivos a mayúsculas, Palabras clave y reservadas

Alcance

Es el rango de instrucciones en el que se conoce el nombre, es visible, y puede ser referenciada (visibilidad).

- **Ligadura por alcance estático:** Busco en la declaración del que me contiene.
- **Ligadura por alcance dinámico:** Busco en la declaración del que me invoco.

Clasificación de variables por su alcance		
Global	**Local**	**No local**
Referencia creadas en el programa principal	Referencia dentro de la unidad y esta tiene una declaración de la referencia	Referencia dentro de la unidad y esta no tiene una declaración de la referencia

Tipo

Es el tipo de variables definidas, tiene asociadas rango de valores y operaciones permitidas. Ayuda a a detectar errores en forma temprana y a la confiabilidad del código. Se detalla los momentos de ligadura:

- Estático (en traducción), el tipo se liga en compilación y no puede modificarse, la ligadura se realiza con la declaración y el chequeo de tipo es estático, la ligadura puede ser realizada en forma:

Explícita	Implícita	Inferida
Directamente se declara el tipo	Indirectamente se declara el tipo	Se deduce del dato provisto

- Dinámico (en ejecución), el tipo se liga en ejecución y puede modificarse.

L-Valor

Es el área de memoria ligada a la variable durante la ejecución, está asociado al tiempo de vida (variables se alocan y desalocan) y su alocacion (reserva de memoria).

Tiempo de vida: El tiempo de vida es el tiempo en que está aloca da la variable en memoria y la ligadura existe. Es desde que se solicita hasta que se libera.

Alocación: Es el momento en que se reserva la memoria para una variable. La alocación depende de los lenguajes y encontramos tipos como:

- **Estática:** Se hace en compilación.
- **Dinámica:** Se hace en tiempo de ejecución y hay dos tipos: La automática que es cuando aparece una declaración en ejecución y la explicita que es requerida por el programador a través de una sentencia.
- **Persistente:** Persisten más allá de la memoria.

R-Valor

Es el valor codificado almacenado en la ubicación de la variable. El tipo de ligadura es dinámica, puede modificarse siempre salvo que se declare como constante.

Estrategias de inicialización: Por defecto, En la declaración

Variables anónimas

Algunos lenguajes permiten que variables sin nombre sean accedidas por el r-valor de otra variable, ese r-valor se denomina referencia o puntero a la variable, la referencia puede ser a el r-valor de una variable nombrada o el de una variable referenciada llamada acces path de longitud arbitraria, algunos lenguajes permiten que el r-valor de una variable sea una referencia al l-valor de otra variable.

Concepto de sobrecarga y alias

Alias: Distintos nombres tienen una entidad, dos punteros a una misma referencia.

Sobrecarga: Un nombre tiene distintas entidades, operador de suma distinto por tipo de dato.

Unidades de Programas

Unidades

Los lenguajes de programación permiten que un programa este compuesto por unidades (Acción abstracta). Las unidades se las conoce normalmente como rutinas (Procesos, Funciones). Se detallan los atributos de una unidad:

Nombre

Cadena de caracteres que se usa para invocar a la rutina (Identificador).

Alcance: Rango de instrucciones donde se conoce su nombre. El alcance se extiende desde el punto de su declaración hasta algún constructor de cierre. Según el lenguaje puede ser estático o dinámico. La llamada puede estar solo dentro del alcance de la rutina. Algunos lenguajes (C, C++, Ada) hacen distinción entre definición y declaración de las rutinas.

Tipo

El encabezado de la rutina define el tipo de los parámetros y el tipo del valor de retorno (si lo hay). La signatura especifica el tipo de la rutina. Ej, func: T1xT2 → res

L-Valor

Es el lugar de memoria en el que se almacena el cuerpo de la rutina.

R-Valor

La llamada a la rutina causa la ejecución su código, eso constituye su r-valor.

- Estático: El caso mas usual.
- Dinámica: Variables de tipo rutina.

Representación en ejecución

Instancias de la unidad

Segmento de código: Instrucciones de la unidad se almacena en la memoria de instrucción (Contenido fijo).

Registro de activación: Datos locales de la unidad se almacena en la memoria de datos (Contenido cambiante).

Estructura de ejecución de los lenguajes de programación

Estático: C1, C2, C2'	**Basado en pila:** C3, C4, C5'	**Dinámico:** C5'', C6
El espacio necesario para la ejecución se deduce del código	El espacio se deduce del código	Lenguajes con impredecible uso de memoria
Todo los requerimientos de memoria necesarios se	Programas más potentes cuyos requerimientos de	Los datos son alocados dinámicamente solo

conocen antes de la ejecución	memoria no puede calcularse en traducción	cuando se los necesita durante la ejecución
La alocación puede hacerse estáticamente	Se utiliza una estructura de pila para modelizarlo, una pila no es parte de la semántica del lenguaje, es parte de nuestro modelo semántico	No pueden modelizarse con una pila, el programador puede crear objetos de dato en cualquier punto arbitrario durante la ejecución del programa
No puede haber recursión	Las variables se alocan automáticamente y se desalocan cuando el alcance se termina.	Los datos se alocan en la zona de memoria heap
	La memoria a utilizarse es predecible y sigue una disciplina last-in-first-out	

Clasificación de lenguajes

- C1: Programa sencillo sin bloques internos, solo código programa principal..
- C2: Datos globales, con unidades sin anidamientos, punto de retorno a la rutina principal.
- C2': Unidades compiladas separadas, variables externas.
- C3: Rutinas con capacidad de llamarse a si misma (recursión directa) o a otra rutina de forma recursiva (recursión indirecta), rutinas con la capacidad de devolver valores (funciones), espacio fijo y conocido, no se sabe cuantas instancias de una unidad se necesitaran durante ejecución, se necesita el valor de retorno en el esquema, link dinámico, free (siguiente posición libre en la pila) y current (dirección del registro de activación actual en ejecución)
- C4: Estructura de bloque, permite sentencias compuestas en declaraciones locales y permite la definición de rutinas dentro de otras rutinas. se introduce el link estatico
- C5': Registro de activación cuyo tamaño se conoce cuando se activa la unidad. Ej, arreglos dinámicos, se reserva lugar en el registro de activación para los descriptores de los arreglos dinámicos.
- C5'': Los datos pueden alocarse durante la ejecución. Se alocan explícitamente durante la ejecución mediante instrucciones de alocación
- C6: Se trata de aquellos lenguajes que adoptan más reglas dinámicas que estáticas. Usan tipado dinámico y reglas de alcance dinámicas. Se podrían tener reglas de tipado dinámicas y de alcance estático, pero en la practica las propiedades dinámicas se adoptan juntas. Una propiedad dinámica significa que las ligaduras correspondientes se llevan a cabo en ejecución y no en compilación.

Formas de comunicación entre las rutinas

Rutinas

Definición y tipos

También llamadas subprogramas. Son una Unidad de Programa (función, procedimiento) que están formadas por un conjunto de sentencias que representan una acción abstracta. Permiten al programador definir una nueva operación a semejanza de las operaciones primarias ya integradas en el lenguaje. Permiten ampliar a los lenguajes, dan modularidad, claridad y buen diseño. Se lanzan con una llamada explícita (se invocan por su nombre) y luego retornan a algún punto de la ejecución (responden al esquema call/return).

- **Procedimiento:** Conjunto de sentencias que resuelve un subproblema y puede no devolver valores.
- **Función:** Conjunto de sentencias que resuelve y siempre devuelve un valor en el punto en el que se llamo.

Comunicación de subprogramas:

Ambiente no local implícito:

- Regla de alcance dinámico: Quién me llamó y busco el identificador.
- Regla de alcance estático: Dónde está contenido y busco el identificador.

Ambiente común explicito: Permite definir áreas comunes de código. El programador debe especificar que es lo comparte. Cada lenguaje tiene su forma de realizarlo.

Pasaje de parámetros: Parámetro Formal, Parámetro Real.

Parámetros

¿Los parámetros formales son variables locales?

Si, un parámetro formal es una variable local a su entorno.

¿Qué datos pueden ser los parámetros reales?

Un parámetro real puede ser un valor, entidad, expresión, y otros, que pueden ser locales, no locales o globales, y que se especifican en la llamada a una función/rutina dentro de la unidad llamante.

Ventajas del uso de parámetros

Protección: El uso del ambiente no local cause un decremento en la seguridad.

Legibilidad

Modificable

Vinculación de los Parámetros

Momento de vinculación de parámetro real y formal: comprende la evaluación de los parámetros reales y la ligadura con los parámetros formales.

Evaluación

En general antes de la invocación primero se evalúan los parámetros reales, y luego se hace la ligadura. Se verifica que todo esté bien antes de transferir el control a la unidad llamada.

Momento de ligadura

Por posición: Se corresponden con la posición que ocupan en la lista de parámetros. Van en el mismo orden

Por nombre o palabra clave: Se corresponden con el nombre por lo tanto pueden estar colocados en distinto orden en la lista de parámetros.

Tipos de parámetros

Parámetros de datos

In: Valor, Valor constante.

Out: Resultado, Resultado de Funciones.

In / Out: Valor / Resultado, Referencia, Nombre.

Parámetros de subprograma

El parametro real es una referencia a un proceso o funcion.

- Deep: Donde esta declarado el subprograma.
- Shallow: Donde esta declarado el parametro formal que recibe el subprograma.
- Ad Hoc: Donde se hace el llamado a la función con en subprograma.

Unidades genéricas

Que son las unidades genericas?

Son una opción para lenguajes que no permiten pasar parámetros de tipo subprogramas. Una unidad genérica es una unidad que puede instanciarse con parámetros formales de distinto tipo.

Tipos de datos

Concepto

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

<table>
<tr><th colspan="3">Tipos de Datos</th></tr>
<tr><th></th><th>Elementales</th><th>Compuestos</th></tr>
<tr><td rowspan="4">Predefinidos</td><td>Enteros</td><td rowspan="4">String</td></tr>
<tr><td>Reales</td></tr>
<tr><td>Caracteres</td></tr>
<tr><td>Booleanos</td></tr>
<tr><td rowspan="3">Definidos por el usuario</td><td rowspan="3">Enumerados</td><td>Arreglos</td></tr>
<tr><td>Registros</td></tr>
<tr><td>Listas</td></tr>
<tr><td colspan="3">Dominio del tipo determina los valores posibles</td></tr>
</table>

Tipos predefinidos

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

Las ventajas de los tipos predefinidos son la invisibilidad de la representación, la verificación estática, el desambiguar operadores y el control de precisión

Tipos definidos por el usuario

Compuestos Constructores:

- **Producto Cartesiano:** Elemento del conjunto entre el producto cartesiano de dos o mas conjuntos.
- **Correspondencia Finita:** Es una función de un conjunto finito de valores de un tipo de dominio de datos con valores del tipo del dominio de retorno. Ej: fun: [dominio de datos] → [domimio de retorno]
- **Unión y union discriminada:** La declaración es muy similar a la del producto cartesiano. La diferencia es que sus campos son mutuamente excluyentes. Solo puede existir un campo asignado. En la union discriminada, agrega un discriminante, el cual indica el tipo que tiene almacenado la unión, haciendo que sea mas seguro al operar
- **Recursión:** Un tipo de dato recursivo T se define como una estructura que puede contener componentes del tipo T, los datos pueden crecer arbitrariamente y su estructura puede ser arbitrariamente compleja.

Punteros y manejo de memoria

Que es un puntero?

Un puntero es una referencia a un objeto, una variable puntero es una variable cuyo r-valor es una referencia a un objeto.

Inseguridad de punteros

Violación de tipos: Cuando un puntero termina operando sobre un tipo distinto al que fue declarado.

Referencias sueltas: Una referencia suelta o dangling es un puntero que contiene una dirección de una variable dinámica que fue desalocada.

Punteros no inicializados: Peligro de acceso descontrolado a posiciones de memoria, la solución es el valor especial nulo (null, nil, void).

Punteros y uniones discriminadas: Si un puntero esta en una unión se debe tener cuidado, ya que el cambiar el valor dentro de la unión ocasionara que se pierda la referencia del puntero, dejándolo inaccesible

Alias: Dos punteros tienen la misma referencia, cambiar la referencia en uno afectara al otro.

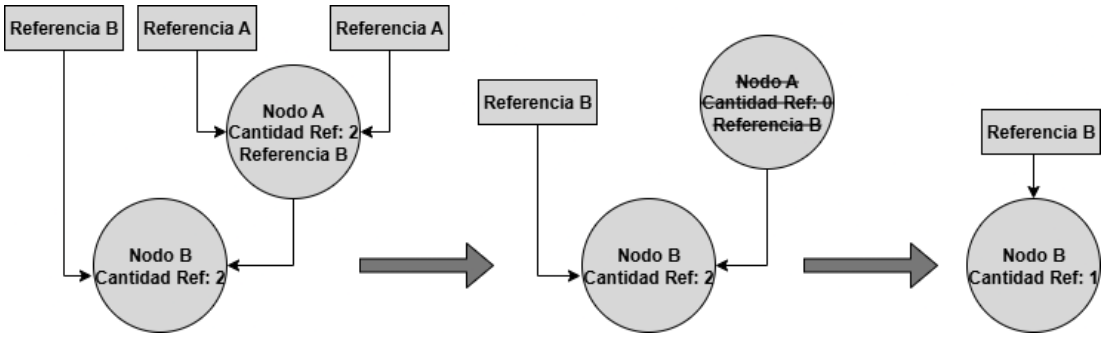
Liberación de memoria: Memoria que esta alocada pero no hay ninguna referencia. Si los objetos en el heap dejan de ser accesibles esa memoria podria liberarse por lo que se implementan mecanismos para desalocar memoria:

- **Explicita:** Dispose (Pascal), Delete (C++)
- **Implícita:** Colector de basura (Garbage Collector)

Colector de basura (Garbage Collector)

Es un mecanismo de liberación de memoria de forma implícita, cuando la memoria alocada ya no tiene referencias pasa a ser candidata para ser desalocada. Se cuenta con varias estrategias entre ellas estan:

Conteo de Referencias: Cada objeto tiene contado los punteros que lo referencian, cuando llega a cero, se considera basura y es candidato a eliminacion, este cuando es eliminado los objetos que referenciaba el candidato se les decrementa el contador en uno.



Recolección por Trazado: Se asignan celdas libres hasta que el espacio libre se agote o llegue a una determinada cantidad. Solo en ese momento el sistema entra en una fase de recolección de basura.

Stop and Copy: La heap se divide en dos, la primer mitad se asignan los espacios que pueden variar ocasionando fragmentación externa, cuando esta por llenarse las

asignaciones alcanzables se pasan a la segunda mitad para ser compactadas, una vez compactadas el colector cambia la noción de la primer y segunda mitad de la heap permitiendo continuar con la ejecución.

Recolección Generacional: La heap se divide en regiones, joven, adulta, anciana como ejemplo, los objetos se crean como jóvenes, cuando escasea el espacio el recolector examina la región joven, si estos sobreviven a la región joven pasan a la siguiente, si no se puede reclamar espacio de una región se pasa a la siguiente.

Recolector de basura de Rust: Mantener un registro de qué partes del código están utilizando qué datos en el heap, minimizar la cantidad de datos duplicados en el heap y limpiar los datos no utilizados en el heap para que no se quede sin espacio son todos problemas que el concepto de ownership (en rust) aborda. Las reglas que define rust son:

- Cada valor en Rust tiene un propietario (una variable).
- Solo puede haber un propietario a la vez.
- Cuando el propietario sale del alcance, el valor se descartará.

Datos Abstractos (TAD)

Motivo para los TAD

Abstraer es representar algo descubriendo sus características esenciales y suprimiendo las que no lo son. El principio básico de la abstracción es la información oculta.

TAD = Representación (Datos) + Operaciones (Funciones y Procedimientos).

Los tipos de datos son abstracciones y el proceso de construir nuevos tipos se llama abstracción de datos. Los nuevos tipos de datos definidos por el usuario se llaman tipos abstractos de datos.

Tipo abstracto de dato (TAD) es el que satisface:

- **Encapsulamiento:** la representación del tipo y las operaciones permitidas para los objetos del tipo se describen en una única unidad sintáctica.
- **Ocultamiento de la información:** la representación de los objetos y la implementación del tipo permanecen ocultos.

Especificación de un TAD

La especificación formal proporciona un conjunto de axiomas que describen el comportamiento de todas las operaciones. Ha de incluir una parte de sintaxis y una parte de semántica.

Especificación Formal	TAD nombre del tipo (Textualmente que valores que toma los datos del tipo)
Especificación de Sintaxis	Operación(Tipo argumento, ...) → Tipo resultado
Especificación de Semántica	Operación(Valores particulares argumentos) → Expresión resultado

Sistema de tipos

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

Tipado Fuerte: Un lenguaje se dice fuertemente tipado (type safety) si el sistema de tipos impone restricciones que aseguran que no se producirán errores de tipo en ejecución. Lo contrario establece un sistema débil de tipos.

Reglas de Equivalencia y Conversión

Reglas de equivalencia

Equivalencia por nombre: Dos variables son del mismo tipo si y solo si están declaradas juntas o si están declaradas con el mismo nombre de tipo.

Equivalencia por estructura: Dos variables son del mismo tipo si los componentes de su tipo son iguales.

Un tipo es compatible si:

- Es equivalente
- Se puede convertir

Coerción

Significa convertir un valor de un tipo a otro. Reglas del lenguaje de acuerdo al tipo de los operandos y a la jerarquía.

Conversión

Widening (ensanchar): cada valor del dominio tiene su correspondiente valor en el rango. (Entero a Real). Pascal solo widening de entero a real.

Narrowing (estrechar): cada valor del dominio puede no tener su correspondiente valor en el rango. En este caso algunos lenguajes producen un mensaje avisando la pérdida de información. (Real a Entero). En C Depende del contexto y utiliza un sistema de coersión simple.

Cláusula de casting: conversiones explicitas, se fuerza a que se convierta.

Reglas de Inferencia y nivel de polimorfismo

Inferencia

La inferencia de tipos permite que el tipo de una entidad declarada se “infiera” en lugar de ser declarado. La inferencia puede realizarse de acuerdo al tipo de:

- **Un operador predefinido:** `fun f1(n,m)=(n mod m=0)`
- **Un operando:** `fun f2(n) = (n*2)`
- **Un argumento:** `fun f3(n) = n*n`
- **El tipo del resultado:** `fun f4(n) = (n*n)`

Monomorfismo

Un lenguaje se dice mono-mórfico si cada entidad se liga a un único tipo (estáticos). En un lenguaje de programación mono-mórfico la función disjuntos deberá implementarse para cada tipo particular de conjunto.

Polimorfismo

Un lenguaje se dice polimórfico si las entidades pueden estar ligadas a más de un tipo. Las variables polimórficas pueden tomas valores de diferentes tipos. Las operaciones polimórficas son funciones que aceptan operandos de varios tipos. Los tipos polimórficos tienen operaciones polimórficas.

El **polimorfismo ad-hoc** permite que una función se aplique a distintos tipos con un comportamientos sustancialmente diferente en cada caso.

- El término sobrecarga se utiliza para referirse a conjuntos de abstracciones diferentes que están ligadas al mismo símbolo o identificador.
- La coerción permite que un operador que espera un operando de un determinado tipo T puede aplicarse de manera seguro sobre un operando de un tipo diferente al esperado.

El **polimorfismo universal** permite que una única operación se aplique uniformemente sobre un conjunto de tipos relacionados.

- Si la uniformidad de la estructura de tipos está dada a través de parámetros, hablamos de polimorfismo paramétrico.
- El polimorfismo por inclusión es otra forma de polimorfismo universal que permite modelar subtipos y herencia.

Estructuras de control

Que son las estructuras de control?

Definición

Son el medio por el cual los programadores pueden especificar el flujo de ejecución entre los componentes de un programa. Las estructuras de control se dividen en dos niveles: a nivel de unidad y a nivel de sentencia.

A Nivel de Unidad y Sentencia

A nivel de unidad

Cuando el flujo de control se pasa entre unidades (rutinas, funciones, procedimientos Etc.) también es necesario estructurar el flujo entre ellas. Las formas de control son: Pasajes de Parámetros, Call-Return, Excepciones entre otras.

A nivel de sentencia

Estas son divididas en tres grupos: Secuencia, Selección e Iteración

Secuencia

Estructuras de control que permiten ejecutar una serie de instrucciones en un orden específico, de arriba hacia abajo, sin ningún tipo de desviación. Algunes lenguajes no delimitan la la sentencia como python, en una linea puede haber una o mas de una instrucción.

Sentencias compuestas

Se pueden agrupar varias sentencias en una con el uso delimitadores.

Ej: (`Begin ... End` , `{ ... }`)

Expresion

Devuelve el valor obtenido como resultado de combinar otros valores a través del uso de operadores. Ej: `(x + 3)*2`

Asignacion

Devuelve el r-valor de una expresión y lo asigna al l-valor modificando así el valor de esa posición de memoria. Ej: `y = (x + 3)*2;`

Asignación múltiple: Caso particular de C donde se asigna de izquierda a derecha con multiples variables. Ej: `A=B=C=10`

La mayoría de los lenguajes de programación requieren que sobre el lado izquierdo de la asignación aparezca un l-valor y no un r-valor.

Selección

Estructuras de control que permiten tomar decisiones basadas en ciertas condiciones. Estas estructuras dirigen el flujo del programa hacia diferentes caminos según el resultado de la evaluación de las condiciones especificadas.

Estructura de control IF ELSE

Estructura de control que permite expresar una elección entre un cierto número posible de sentencias alternativas (ejecución condicional)

```
if x>0 then if x<10 then x:=0 else x:=1000
```

La sentencia anterior presenta una ambigüedad, que valor devuelve x?

X	Caso A (else al 2do if)	Caso B (else al 1er if)
10	Cambia 1000	Deja sin cambio 10
0	Deja sin cambio 0	Cambia 1000

Por lo que se definieron formas para eliminar este tipo de ambigüedad.

- Que cada else se empareje con la instrucción if solitaria más próxima buscando de derecha a izquierda.
- Usar sentencias de cierre explícitas del bloque condicional if.
- Sentencias compuestas (Bloque de instrucciones).

Selección con expresión corta (Operador ternario)

Reducción del if para que quede una sentencia en una sola linea. Ej:

```
if (cond) then begin
   return b;
end else begin
   return c;
end
```

Se traduce a un if corto como:

```
cond ? b : c ;
```

Evaluación de expresiones

Circuito corto: Significa que los operandos de una expresión lógica se evalúan de izquierda a derecha, pero solo hasta encontrar el primero que determina el resultado final. En ese punto, la evaluación se detiene y no se evalúan los operandos restantes.

Circuito Largo: Significa que todos los operandos se evalúan, sin importar si el primero ya determina el resultado final.

Estructura de control CASE-ELSE

Los lenguajes incorporan distintos tipos de sentencias de selección múltiple para poder elegir entre dos o más opciones/caminos posibles, y evitar de If anidados.

Como el else es opcional no declararlo puede llevar a inconsistencias e inseguridad.

Funcionamiento del "Falling Through"

Cuando se evalúa una expresión en un switch-case en C, el compilador busca un case cuyo valor coincida con el valor de la expresión. Una vez que encuentra un case con coincidencia, ejecuta el código asociado a ese case y luego continúa ejecutando el código de los case siguientes que no tienen break hasta encontrar un break, Default o llegar al final del switch.

Interacion

Estructuras de control que permiten repetir un bloque de código múltiples veces hasta que se cumpla una condición de salida. Esto permite la ejecución repetida de una serie de instrucciones sin tener que escribir las mismas instrucciones una y otra vez. Ejemplos for, while, do-while y similares.

Estructura de control FOR

Bucles en los que se conoce el número de repeticiones al inicio del bucle. Se repiten un cierto número de veces.

Estructura de control WHILE

Bucles en los que el cuerpo se ejecuta repetidamente siempre que se cumpla una condición.

Estructura de control UNTIL-REPEAT y DO-WHILE

Estructuras que permite repetir un proceso "hasta" que se cumpla una condición. En definitiva, permite ejecutar un bloque de instrucciones mientras no se cumpla una condición dada (o sea falso). La condición/expresión se evalúa al final del proceso, por lo que por lo menos 1 vez el proceso se realiza

Excepciones

Qué es una excepción?

Definición

Condición inesperada o inusual, que ocurre durante la ejecución del programa y no puede ser manejada en el contexto local. La excepción interrumpe el flujo normal de ejecución y ejecuta un controlador de excepciones registrado previamente.

Requisitos para tratarlas

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

Tipos de excepciones

Implicitas

Definidas por el lenguaje (built-in).

Explicitas

Definidas por el programador.

Controlador de excepciones

Que es?

Es una sección de código que se encarga de manejar una excepción en un programa. Su objetivo principal es proporcionar una forma de recuperarse de un error o falla, permitiendo que el programa continúe ejecutándose en lugar de detenerse abruptamente. Puede tomar distintas acciones según la situación:

- Imprimir un mensaje de error.
- Realizar acciones correctivas.
- Lanzar otra excepción.
- Finalizar la ejecución del programa.

Los lenguajes deben proveer instrucciones en su controlador para:

- Definición de la excepción.
- Levantamiento de una excepción.
- Manejador de la excepción.

Punto de retorno

Después de atender a una excepción, el punto de retorno dependerá del flujo de ejecución del programa y de cómo se haya diseñado el manejo de excepciones en el código. También va a depender del lenguaje. Se puede tener en cuenta:

Continuar la ejecución normal del programa:

- Si después de manejar una excepción el programa puede continuar la ejecución del código restante sin problemas.
- El punto de retorno será definido por el lenguaje (por ejemplo, el siguiente bloque de código después del bloque de manejo de excepciones o siguiente instrucción).

Retornar a un estado anterior:

- Cuando el manejo de excepciones puede requerir que el programa regrese a un estado anterior o deshaga acciones realizadas antes de que se produjera la excepción.

Propagar la excepción:

- En el controlador de excepciones que no puede manejar completamente la excepción, puede optar por propagarla a un nivel superior en la jerarquía de llamadas.
- El punto de retorno sería el controlador de excepciones en el nivel superior que pueda manejar la excepción o decidir cómo manejarla.

Terminar la ejecución del programa:

- En situaciones excepcionales o críticas, es posible que el controlador de excepciones determine que no se puede continuar ejecutando el programa de manera segura.
- El punto de retorno puede ser la finalización del programa o alguna acción específica de cierre antes de la finalización.

Modelos de manejo de excepciones

Reasunción

Se refiere a la posibilidad de retomar la ejecución normal del programa después de manejar una excepción. El controlador de excepciones realiza las acciones necesarias para manejar la excepción (medidas correctivas) y luego el programa continúa su ejecución a partir del punto donde se produjo la excepción. Este fue implementado solo por PL/1

Terminación

El controlador de excepciones realiza las acciones necesarias para manejar la excepción, pero no se retorna al punto donde se produjo la excepción (invocador), continúa su ejecución a partir de la finalización del manejador.

Propagación: Si la unidad que genera la excepción no proporciona un manejador, se debe buscar ese manejador dinámicamente. Esto significa que se vuelve a levantar en otro ámbito.

Implementación en distintos lenguajes

PL/1

Criterio: Reasunción

Formato:

```
1. prog Main
2. begin
3.     on condition MayorError begin ... end;
4.     ...
5.     if (condError) then begin signal condition MayorError end;
6. end
```

Características:

- Este lenguaje tiene una serie de excepciones ya predefinidas con su manejador asociado. Son las Built-in exceptions.
- Los manejadores se ligan dinámicamente con las excepciones. Una excepción siempre estará ligada con el último manejador definido. (Manejo de pila de manejadores de excepciones). Ej: Si una excepción `e1` se define en una unidad `a` y en una unidad `b` si ocurriera una excepción `e1` antes de la de la definición en `b`, la excepción usara el manejador de `a`.
- El alcance de un manejador termina cuando finaliza la ejecución de la unidad donde fue declarado.

ADA

Criterio: Terminación

Formato:

```
1. package Main is
2.     e1: exception
3. begin
4.     if (condError) then raise e1;
5.     exception
6.     when e1 => begin ... end;
7.     when others => begin ... end;
8.     end
9. end Main;
```

Características:

- La asociación de la excepción con el manejador es determinístico (variable → manejador de esa variable). Asigno mismo nombre a ambos
- Si se deseara continuar ejecutando las instrucciones de un bloque donde se lanza una excepción, es preciso “crear un bloque más interno”. Se usa Declare para amar una unidad, como se muestra en el ejemplo y luego agregar instrucciones restantes abajo. (Simular a reasunción).

C++

Criterio: Terminación

Formato:

```
1. int main() {
2.     try {
3.         throw 125;
4.     } catch(int) {
5.         ...
6.     }
7. }
```

Características:

- Si una excepción se propaga repetidamente y nunca se encuentra un manejador coincidente, llama automáticamente a una función especial llamada `terminate()`. `terminate()` puede ser redefinido por el programador. Su comportamiento predeterminado, eventualmente aborta la ejecución del programa.

CLU

Criterio: Terminación

Formato:

```
1. procedure main() {
2.     procedure uno() signals e1; begin ... end;
3.     begin
4.         uno();
5.         exception
6.             when e1: ... end;
7.     end
8. end
```

Características:

- Solamente pueden ser lanzadas por los procedimientos. Si una instrucción genera una excepción, el procedimiento que contiene la instrucción retorna anormalmente al generar la excepción. Un procedimiento no puede manejar una excepción generada por su ejecución, quien llama al procedimiento debe encargarse de manejarla.
- La excepciones que un procedimiento puede lanzar se declaran en su encabezado.
- Si el manejador no se encuentra en ese lugar la excepción se propaga estáticamente en las sentencias asociadas. Esto significa que el proceso se repite para las sentencias incluidas estáticamente.

Java

Criterio: Terminación

Formato:

```
1. class Main {
2.     public static void main(args[]) {
3.         try { if(condError) throw new Exception(); }
4.         catch (Exception e) { ... }
5.         finally { ... }
6.     }
7. }
```

Características:

- Se debe poner cualquier código que el programador desee que se ejecute siempre, en la clausula `finally`.

Python

Criterio: Terminación

Formato:

```
1. try:
2.     raise Exception
3. except Exception:
4.     ...
5. else
6.     print("No se disparo una excepción")
7. finally
8.     print("Siempre soy ejecutado")
```

Características:

- Existe la clausula `else` la cual ejecuta codigo solo si no se disparo una excepcion.

PHP

Criterio: Terminación

Formato: Similar a java.

Características:

- El objeto lanzado debe ser una instancia de la clase Exception o de una subclase de Exception. Intentar lanzar un objeto que no lo es resultará en un Error Fatal de PHP.

Paradigmas de Programación

Definición

Un paradigma de programación es un estilo de desarrollo de programas, un modelo para resolver problemas computacionales. Los lenguajes de programación, necesariamente, se encuadran en uno o varios paradigmas a la vez, a partir del tipo de órdenes que permiten implementar, tiene una relación directa con su sintaxis.

Paradigmas principales

Imperativo

Paradigma que consiste en una clara definición de las instrucciones para un ordenador

Declarativo

Los programas describen los resultados esperados sin listar explícitamente los pasos a llevar a cabo para alcanzarlos.

Lógico

Basado en aserciones lógicas que consiste en la definición de hechos y reglas, también están las consultas las cuales verifican si ciertas condiciones son verdaderas, los objetos son representados por términos, los cuales contienen constantes y variables, se considera como un subparadigma del paradigma declarativo.

Funcional

Los programas se componen de funciones.

Características de los lenguajes funcionales:

- Provee un conjunto de funciones primitivas.
- Provee un conjunto de formas funcionales.
- Semántica basada en valores.
- Transparencia referencial.
- Regla de mapeo basada en combinación o composición.
- Las funciones de primer orden.

Orientado a Objetos

El programa se compone de objetos que interactuan entre si a través de métodos y mensajes.

Dirigido a Eventos

El flujo del programa está determinado por sucesos externos (por ejemplo, una acción del usuario).

Orientado a Aspectos

Apunta a dividir el programa en módulos independientes, cada uno con un comportamiento y responsabilidad bien definido.

Lenguajes basados en script

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

- Uso de scripts para “pegar” o combinar programas.
- Desarrollo y evolución rápida.
- Asociado a editores livianos.
- Interpretados – (modestos requerimientos de eficiencia).
- Alto nivel de funcionalidad en aplicaciones de áreas específicas.