

Sintaxis → Semántica

Compilador, para a lenguaje máquina

- Sintaxis = texto del programa

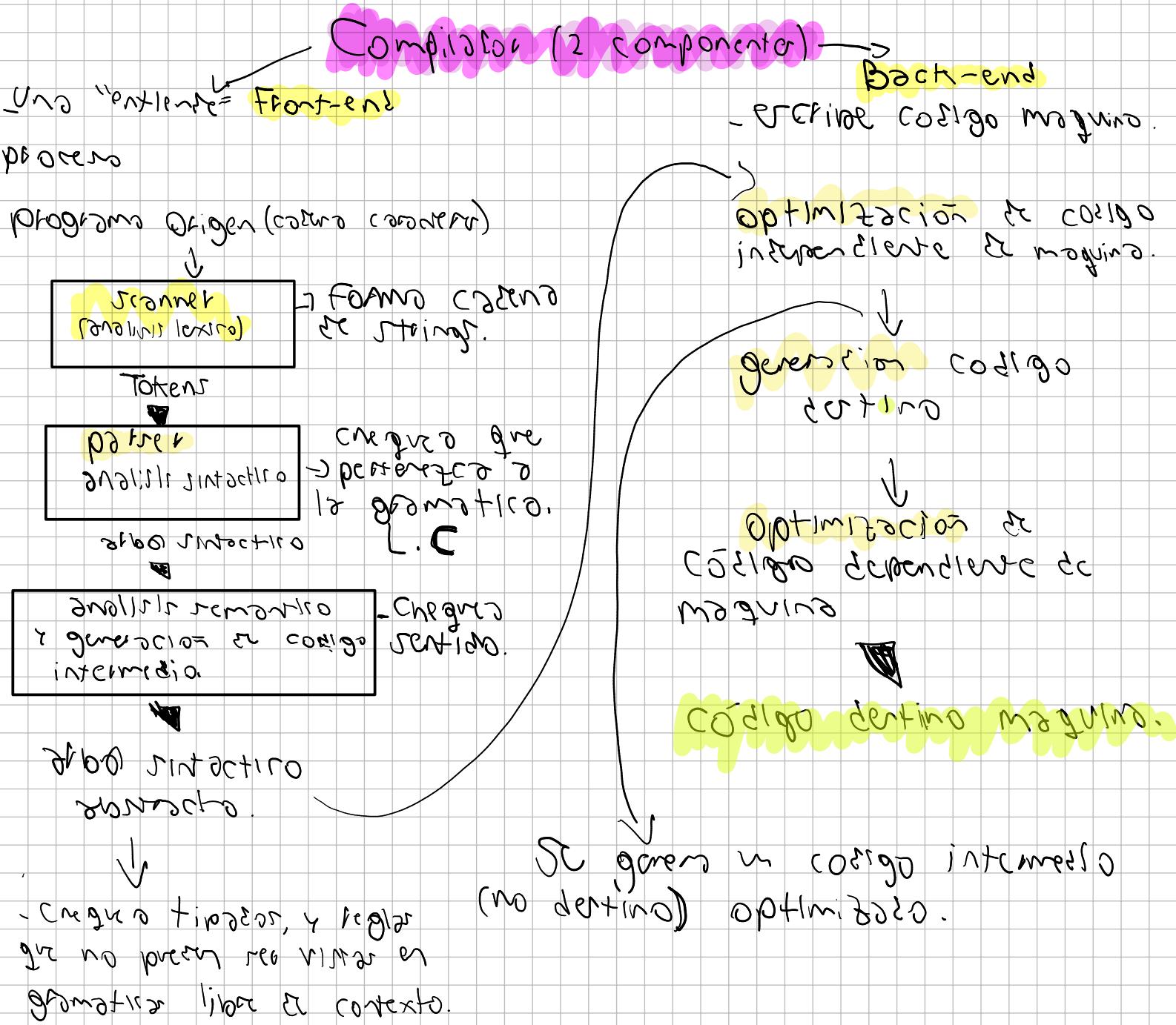
- Semántica = cosa que hace → código máquina.

- La implementación de un lenguaje transforma la sintaxis a assembler.

Existen interpretadores que combinan traducción y ejecución.

- El compilador traduce un programa en un lenguaje máquina.

- Existen compiladores de lenguajes a otros lenguajes. (Trascriptores)



Comprobaciones Semánticas del Compilador

- Comprobar si es típico.
- Comprueba si variables.
- Viso la identificación en contexto adecuado.
- Comprobar argumentos.
- Si no hay fallo, se genera error.

En tiempo de ejecución

- Valores que exceden límites.
- División por 0.
- Si hay error se levanta una excepción.

Tipo fuerte

- Tiene tipos fuertes en siempre la ejecución las errores de tipo.

- en tiempo de compilación o ejecución.
 - Frente = Java - ML - Haskell
 - Ruby - Factor - Parrot - C/C++ - Lisp

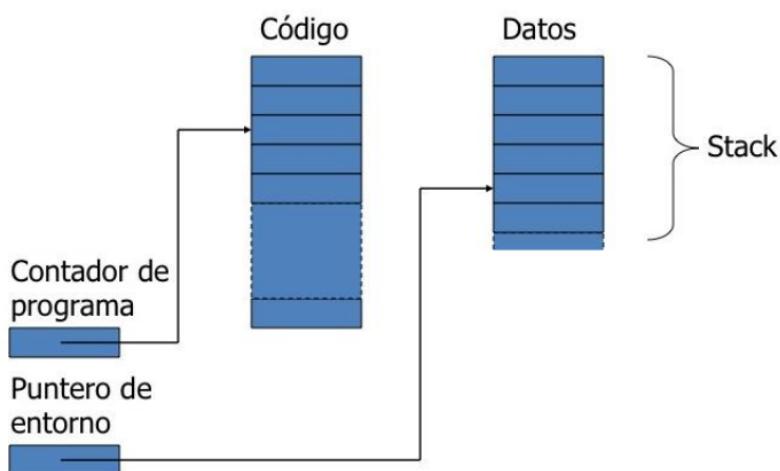
- hace que sea más seguro pero más lento por las comprobaciones dinámicas.

- En general los lenguajes europeos y la expresividad que tienen la gramática LC.

Semántica Operacional

- Interpretación abstracta de la ejecución de un programa, como secuencia de transiciones entre estados.

- Los estados son una descripción abstracta de la memoria.



- cada bloque agrega una activación nueva al stack para las variables locales y puntero de retorno.

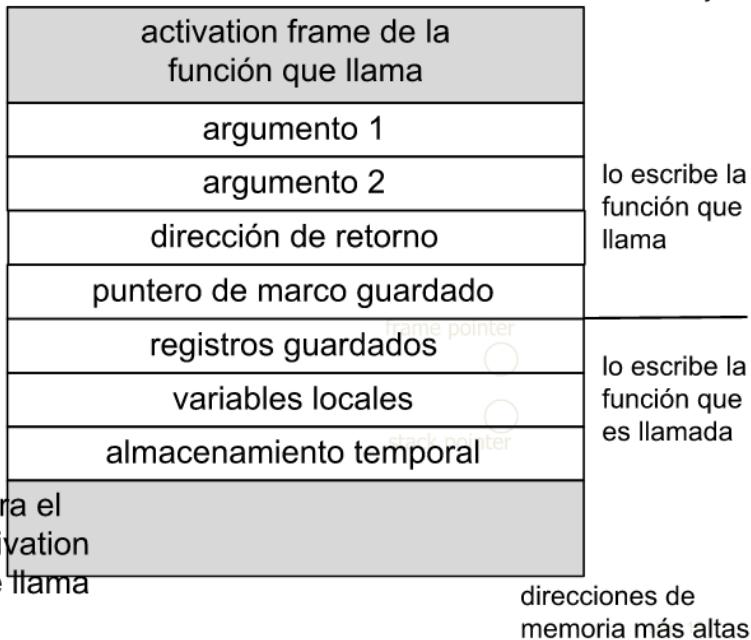
activation frame de la función que llama

activation frame de la función que es llamada

espacio para el siguiente activation frame que se llama

direcciones de memoria más bajas

ESTRUCTURAS DE UN ACTIVATION RECORD.



Cáps 1, 2, 3, 5 & 17 guía.

Paradigma Imperativo.

Elementos básicos:

- Definición de tipos, declaración de variables (Tipizar o no)
 - Ubicación → valor de la variable:
 - Global, en la pila o stack, puede pertenecer a un bloque.
 - Variable = ubicación en memoria
 - Valor = valor guardado en la memoria.
 - Identificador = nombre.
- Expresión y sentencia de asignación:
 - El r-valor es un punto en el l-valor de una variable.
- Sintaxis control de flujo.
 - Un programa en estructuras ni el flujo de control es evidente en las estructuras sintácticas del programa.
- Bloque: agrupar código en bloques lógicos.
 - Llamado a memoria: se entra en bloque se guarda en memoria.
 - Se salen de código si se libera o no.
- Al entrar a un bloque se agrega Activation Record al stack, al salir se lo elimina.

declaración de funciones o procedimientos.

- Una función tiene varios de efectos.
- un procedimiento NO retorna valores, pero tiene un efecto secundario visible, como son los efectos de algún valor global.

Act. cerca para funciones, tiene lugar para:

- parámetros, variables locales, dirección de retorno.
- control lineal de quien hace la llamada.

Conceptos:

Parámetros y argumentos: Guía 6.2

- **Por valor:** - la función que llama para el valor (la copia) los argumentos a la función llamada.
 - no hay "aliasing" (no identifican la misma ubicación)
 - la func no puede cambiar el valor de la variable original.
 - método para estructurar códigos.
- **Por referencia:** - se pasa el L-VIOL de los argumentos a la función que es llamada.
 - se asigna la dirección de memoria del argumento al parámetro
 - como "aliasing" tengo que referirme a una ubicación.
 - la función puede modificar la variable que llama.
 - método para estructurar código.
- **Por valor-férutado:**
 - Hace una copia en los argumentos al principio, copia las variables locales a los propios argumentos al final del procedimiento, de forma que se modifican los argumentos.
 - Cuidado: el comportamiento depende del orden en que se copian las variables locales.

- no tiene aliasing.
- tiene efectos en los argumentos.

Por nombre:

- en el cuerpo de la función se sustituye textualmente el argumento para cada instancia de su parámetro

- es un ejemplo de ligado tardío
 - la evaluación del argumento se posterga hasta que efectivamente se ejecuta en el cuerpo de la función
 - asociado a evaluación perezosa en lenguajes funcionales (e.g., Haskell)

Por recambio:

Variación de *call-by-name* donde se guarda la evaluación del parámetro después del primer uso

Idéntico resultado a *call-by-name* (y más eficiente!) si no hay efectos secundarios
El mismo concepto que lazy evaluation

resumen de pasaje de parámetros

método	qué se pasa	lenguajes	comentarios
por valor (by value)	valor	C, C++	simple, los parámetros que se pasan no cambian, pero puede ser costoso
por referencia (by reference)	dirección	FORTRAN, C++	económico, pero los parámetros pueden cambiar!
por valor-resultado (by value-result)	valor + dirección	FORTRAN, Ada	más seguro que por referencia, pero más costoso
por nombre (by name)	texto	Algol	complicado, ya no se usa

Alcance 1 clausuras

6.3, 6.4, 6.5 guía.

variables locales y globales

x, y son locales al bloque exterior
z es local al bloque interior
x, y son globales al bloque interior

```
{ int x=0;
  int y=x+1;
  { int z=(x+y)*(x-y);
  };
}
```

Alcance la declaración en el bloque es programado más tarde, m. dentro para devueltos en el texto.

alcance estático: el valor de las variables globales se obtiene del bloque inmediatamente contenedor

alcance dinámico: el valor de las variables globales se obtiene del activation record más reciente

→ Sigue el punto de la memoria en el AR más reciente.

val x = 4;

fun f(y) = x*y;

fun g(h) =

let

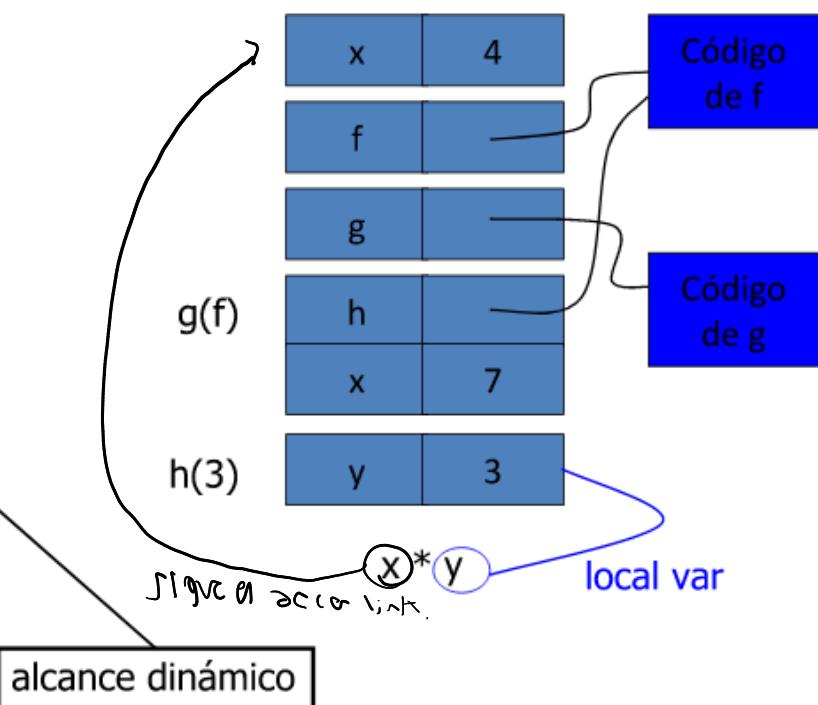
val x=7

in

h(3) + x;

g(f);

alcance estático



Control link
Access link
Return address
Return result addr
Parameters
Local variables
Intermediate results

- Control link
 - link al activation record del bloque anterior (el que llama al actual)
 - depende del comportamiento dinámico del programa
- **Access link**
 - link al activation record del bloque que incluye de más cerca al actual, léxicamente, en el texto del programa
 - depende del texto estático del programa

paso en
→ Diccionario
entorno.

Puntero de programa

AR → borrar en valor de x. que necesita.

en ese caso lo x que una f, esto es
el bloq. inmediatamente contiene la f. para

29

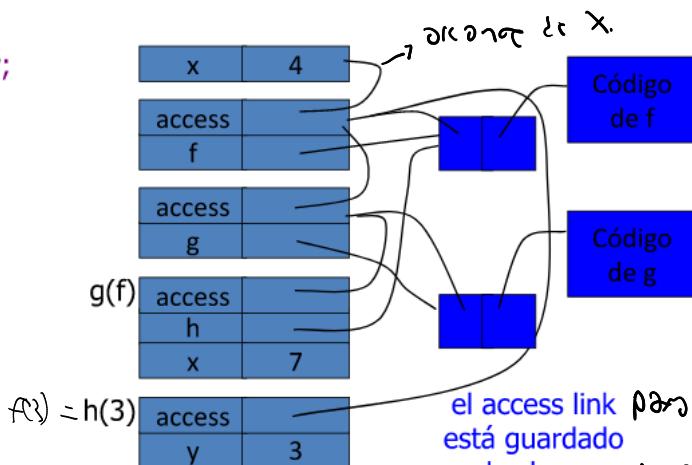
lo que uno en el access link, para ir a ese

función como parámetro:

Cuando pones una función como parámetro en la lista para
que sea "clausura" que es una función que es parámetro) y otro en el código de
la AR de la función (que es parámetro) y otro en el código de
la misma. Entonces, código

pila de ejecución con clausuras

```
val x = 4;
fun f(y) = x*y;
fun g(h) =
  let
    val x=7
    in
      h(3) + x;
  g(f);
```



cuando en g se llama a

f, se va a la AR de f, para
obtener la semántica de el
dicho acceso entorno.

el access link para obtener el valor de x

f, contiene a x.

para obtener el valor de x

funciones & alto orden: una f puede ser argumento o resultado.

- declarar en código abierto.

- Recut Non a la cola.**
- Si g tiene retorno f(x) entonces se "recuerda" el activation record.
- la función g hace una **llamada a la cola** a la función f si el valor de retorno de la función f es el valor de retorno de g
 - ejemplo llamada a la cola no llamada a la cola
 $\text{fun } g(x) = \text{if } x > 0 \text{ then } f(x) \text{ else } f(x)*2$
 - optimización: se puede desapilar el activation record actual en una llamada a la cola
 - especialmente útil para llamadas a la cola recursivas porque el siguiente activation record tiene exactamente la misma forma

Funcional - Imperativo / Declarativo vs Imperativo.

Jar combinaciones imperativas cambian en valor y los declarativos crean en valor.

- designación imperativa: - efectuar secuencias, en función a la llamada funcional.
- Op declarativa o determinística = mismo igual que SIEMPRE.

- una operación declarativa es:
 - **independiente** (depende sólo de sus argumentos)
 - **sin estado** (no recuerda ningún estado entre llamados)
 - **determinística** (los llamados con los mismos argumentos siempre dan los mismos resultados)

- más fácil de razonar en programación declarativa.

Programación a gran escala: una componente declarativa se puede escribir, testear y verificar independientemente de otras componentes.

- la complejidad de razonar sobre un programa compuesto de componentes no declarativas explota por la combinatoria de la interacción entre componentes

Transparencia referencial.

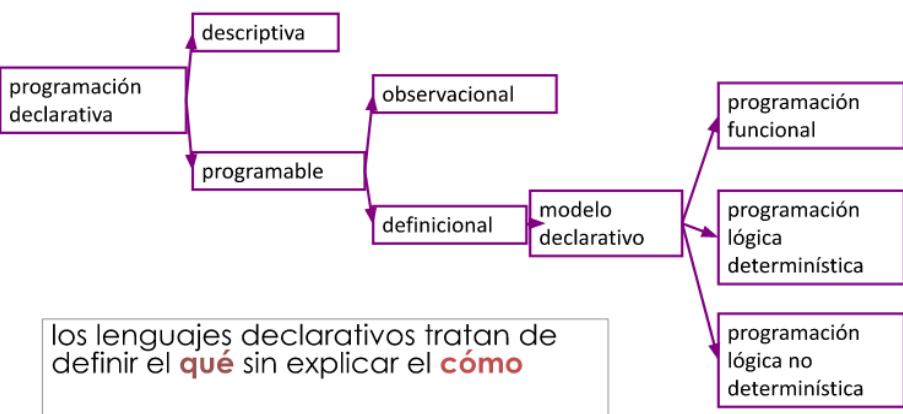
Una expresión transparente, se puede sustituir por su valor y no afecta la semantics en programa.

Todos los comp. declarativos son transparentes, no existe orden entre como valor en cuad. 120.

componentes declarativos vs. lenguajes declarativos

- en todos los lenguajes se pueden escribir componentes declarativos, que tendrán las propiedades mencionadas
- algunos lenguajes proveen sintaxis y semántica más fuertemente declarativas
- de más declarativo a menos declarativo:
Prolog puro, Haskell, OCaml, Scheme/Lisp, Python, Javascript, C--, Perl, PHP, C++, Pascal, C, Fortran, Assembly

clasificación de lenguajes declarativos



cuándo queremos usar el estado explícito

cuando la asignación destructiva convierte un problema intratable en tratable

cuando queremos guardar resultados temporales (por ejemplo, en **programación dinámica**)

- encontrar el camino más corto (Dijkstra) (sabemos cuál es el camino más corto entre los puntos intermedios)

No necesariamente los lenguajes declarativos representan problemas. Existe problemas que requieren la computación, y la vida tiene estados. No todo los problemas tienen una buena solución ejecutiva (funcional).

Cap 9 Apunte

* Estático: tiempo comp.

Dinámico: tiempo & etc.

El ésto es siempre entra en la computación, incluso en las funciones.

- imperativas: integran a ésto en forma explícita.

Para ello las funciones deben acceder al entorno

- mutación

- pjm's entre como pjm's

Usamos estados

- queremos ap. memoria

- el entorno es determinante.

Mutación.

- crea un diccionario para tu programa

- te permiten op. con efectos secundarios.

Tipado cap. 6.1.2.3.4 Mitchell

Tipo: Conjunto de entidades que comparten propiedades.

Muchos tipos lo org. del programa & que el control de tipos en compilador (type checking)

Errores de tipos:

- Ocurren cuando algunas entidades no cumplen las normas que los tipos establecen por su propia naturaleza.

Ejemplo de hardware 1 es semanticos.

Ejemplo: un patron guardado para representar un tipo, cuando se representa otro tipo.

- Algunas operaciones de tipos (por lo general tipos "cajones") pueden ser optimizadas por el compilador, si el tipo es la variable o se obliga en "compilacion".

Lenguaje Type Safe.

Un lenguaje es type safe. Si se respeto la distincion de tipos.

Características que hacen un lenguaje poco seguro:

- Casting de tipos, sumando los punteros. (NO seguro)

- Alloc y free explicito, dangling pointers. (CON seguros)

- CINEOS de tipos compactos. (SEGUROS)

Run-time-check: al código generado por el compilador, hace cheques de tipos entre las expresiones. Encuentra errores de tipo de manera clara. pero no es eficiente.

Compile-Time-Check: chequea expresiones para evitar posibles errores en el código programado con errores de tipos. Permite código más eficiente.

Evita errores que se produzcan en runtime. pero también no es posible encontrar errores en run-time.

no mucha ljs compile-time-check conservadora.

type-checking	VENTAJAS	DESVENTAJAS.
RUN-TIME	crea errores tipos	ineficiente.
compile-time.	elimina runtime errors. encuentra errores antes de correr.	en reactivos por ser conservador.

Inferencia de tipos.

proceso de determinar el tipo de expresiones a partir de otros tipos que aparecen en ellas.

Se usa inferencia lógica para encontrar el tipo de la expresión.

Lenguaje polimórfico.

Algoritmo de inferencia de tipos:

1. asigna tipo a la expresión o como subexpresión. Por cada expri completamente. Una variable es tipo. Poco expresión conocida. ($t, 3$) es tipo conocido.
2. genera inferencia de tipos. Usando el algoritmo de la expresión. Si una func. se le aplica a un arg. $x : t$ el arg es tipo t pero la func tipo 'Int' entonces es un tipo int.
3. devolver entre igualdades/ pertenencias. por sustitución func. las horas de las variables a la raíz.

Polimorfismo y Overloading

3 formas de polimorfismo

- Polimorfismo paramétrico: uno función se adapta a cualquier argumento, cuyos tipos coinciden con los de las expresiones y variables de tipos.

- El otro polimorfismo (Overload): las implementaciones con distinto tipo se asocian con el mismo nombre.

Se resuelve cuál se usará en tiempo de compilación.

C) si operador '+' opera. si para $\text{int} \rightarrow \text{int} \rightarrow \text{int}$ o $\text{real} \rightarrow \text{real} \rightarrow \text{real}$.

Excepciones. Guía 6.6.

- forma estructural de salto para saltar de bloque o normal a función
- se pegan temporales activation records que ya no se necesitan.

Usa el controlador linguístico try-catch.

try: ejecuta una excepción, como para dar comienzo a programación en salto.

catch/handler: mecanismo de control, permite declaraciones, excepto. El llamador a función proveerá respuesta a excepciones.

permite saltar a bloques de programa más avanzados, sin código intermedio, salta al que tiene una parte del código ya cargado en la pila.

- para saltar como parte de otra.

- se establece como un mecanismo de control para casos como, siguiente diccionario.

6.1. En las siguientes funciones en ML:

```
exception Excpt of int;  
1 fun twice(f,x) = f(f(x)) handle Excpt(x) => x;  
2 fun pred(x) = if x = 0 then raise Excpt(x) else x-1;  
3 fun dumb(x) = raise Excpt(x);  
4 fun smart(x) = 1 + pred(x) handle Excpt(x) => 1;
```

Analizar:

59

a) twice(pred, 1) = pred(pred(1)) → no levanta excepción.
- pred(0) → levanta Excpt(0)

6.2)

```
1 class Ejemplo1 {
2     public static void main(String args[]){
3         try{
4             System.out.println("primera sentencia del bloque
try");
5             int num=45/0;
6             System.out.println(num);
7         }
8         catch(ArrayIndexOutOfBoundsException e){
9             System.out.println("ArrayIndexOutOfBoundsException");
10            }
11        finally{
12            System.out.println("bloque finally");
13        }
14        System.out.println("fuera del bloque
try-catch-finally");
15    }
}
```

No se ejecuta
try y finally
porque no
excepción que
levanta el yes/no
no se lanza
después catch

en q b) la excepción creada o ArithmeticException
cañoncina tiene ejecución la 3 parte.

```
class Ejemplo3 {
    public static void main(String args[]){
        try{
            System.out.println("primera sentencia del bloque
try");
            int num=45/3;
            System.out.println(num);
        }
        catch(ArrayIndexOutOfBoundsException e){
            System.out.println("ArrayIndexOutOfBoundsException");
        }
        finally{
            System.out.println("bloque finally");
        }
        System.out.println("fuera del bloque
try-catch-finally");
    }
}
```

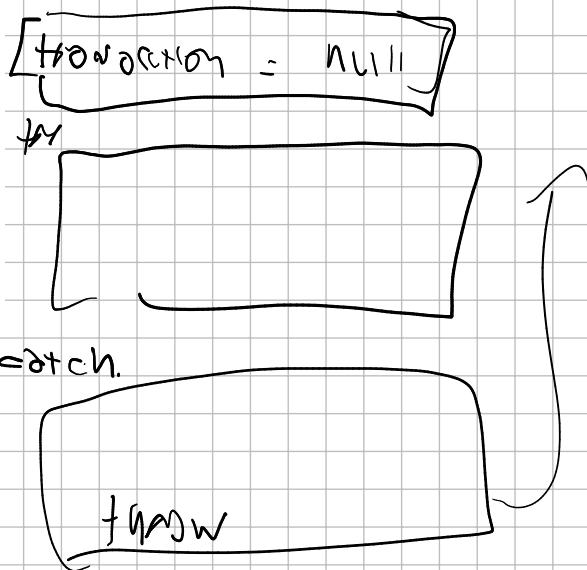
try no levanta
ninguna excepción,
por lo que
se ejecuta
try > finally.

- 6.3. En este fragmento de código se captura una excepción y se vuelve a lanzar la misma excepción. Explique qué sucede en este fragmento de código, ayudándose de un diagrama de una secuencia de estados de la pila de ejecución, mostrando cómo se apilan y desapilan los diferentes *activation records* a medida que se va ejecutando el programa. Para mayor claridad, puede acompañarlo de una descripción verbal. Describa también verbalmente cuál sería el objetivo de este programa. ¿Qué sentido tiene capturar una excepción para volver a lanzarla? ¿Es eso lo único que se hace?

```

1 ITransaccion transaccion = null;
2 try
3 {
4     transaccion = sesion.EmpiezaTransaccion();
5     // hacer algo
6     transaccion.Commit();
7 }
8 catch
9 {
10    if (transaccion != null) {
11        transaccion.RestaurarEstadoPrevio(); }
12    throw;
13 }

```



Vuelve a intentar para hacer otra broma otra vez en la pila. A ver si veces mejor la mejor manera.

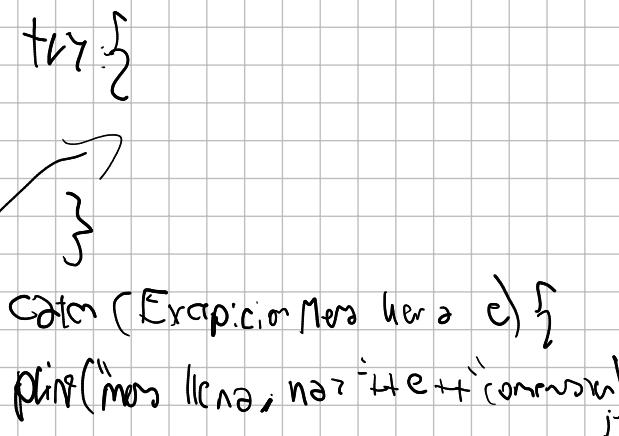
- 6.4. En el siguiente código java, inserte en el main los mecanismos de manejo de excepciones necesarios para capturar la excepción de forma adecuada.

```

public class RepartirComensales
{
    public static void main(String [] args)
    {
        Mesa mesa = new Mesa(1);
        System.out.println("vamos a llenar la mesa 1....");
        mesa.anadirComensal("Ana");
        mesa.anadirComensal("Juan");
        mesa.anadirComensal("Maria");
        mesa.anadirComensal("Pedro");
        mesa.anadirComensal("Juana");
        mesa.anadirComensal("Esteban");
        mesa.anadirComensal("Lola");
    }
}

public class Mesa
{
    private int numeroDeComensales;
    private int numeroDeMesa;
    public Mesa(int numeroDeMesa)
    {
        this.numeroDeMesa = numeroDeMesa;
    }
    public void anadirComensal(string comensal) throws
    ExcepcionMesaLlena
    {
        if(numeroDeComensales > 5)
        {
            throw new ExcepcionMesaLlena(numeroDeComensales);
        }
        else
        {
            numeroDeComensales += 1;
        }
    }
}

```



Garbage Collection

Basura = ubicaciones de memoria que si el programa las de fija, ocupo, no afecta su ejecución.

Recollection de basura en un proceso se detectan basuras en runtime y hace esa memoria disponible para otros programas.

- Los programas deben memoria marcada en modo "free-list".
Cuando era free-list le quedó poco espacio, se "fija" la ejecución y se llama al recolector de basura.
(o veces como un paro)

Mark-and-Sweep

- Primero marca todos los direcciones de memoria accesibles dentro del programa.
 - Luego limpia (sweeps) todos los no marcados.
 - Una un bit de cada dirección de memoria para marcar y desmarcar.
- 1- marca todos los direcciones de memoria con 0.
 - 2- donde el lugar de ejecución sigue teniendo los accesos de memoria y links → los no marcados con 1.
 - 3- los que siguen con 0, los pone en free-list.
- Recolector basura agrega SJ a overhead. de ejecución.