Sintaxis → Semantica

Compilador, para a lenguaje maquina

- Sintaxis = texto del programa
- Semantica = cosa que hace → codigo maquina.
- La implementacion de un lenguaje transforma la sintaxis a ensamblador.

Existen interpretes que combinan traduccion y ejecucion

- El compilador traduce a un programa escrito en lenguaje maquina.
- Existen compiladores de lenguaje a otro lenguaje. (transpiladores)

Compilador (2 componentes)

- Una "entrada" = Front-end

Programa Origen (cadena caracteres)

↓

Scanner (analisis lexico) → Forma cadena de strings.

Tokens

↓

Parser (analisis sintactico) → Comprueba que pertenezca a la gramatica. L.C

arbol sintactico

↓

Analisis semantico y generacion de codigo intermedio. - Comprueba sentido.

arbol sintactico abstracto

↓

- Comprueba tipados, y reglas que no pueden ser vistas en gramatica libre de contexto.

Back-end

- Escribe codigo maquina.

Optimizacion de codigo independiente de maquina

↓

Generacion codigo destino

↓

Optimizacion de codigo dependiente de maquina

↓

Código destino maquina.

Se genera un codigo intermedio (no destino) optimizado.

Comprobaciones semánticas del compilador:

- Comprobación de tipos.
- Declaración de variables.
- Uso de identificadores en contexto adecuado.
- Comprobar argumentos.
- Si hay fallo, se genera error.

En tiempo de ejecución.

- Valores dentro de límites.
- División por 0.
- Si hay error se lanza una excepción.

Tipado fuerte

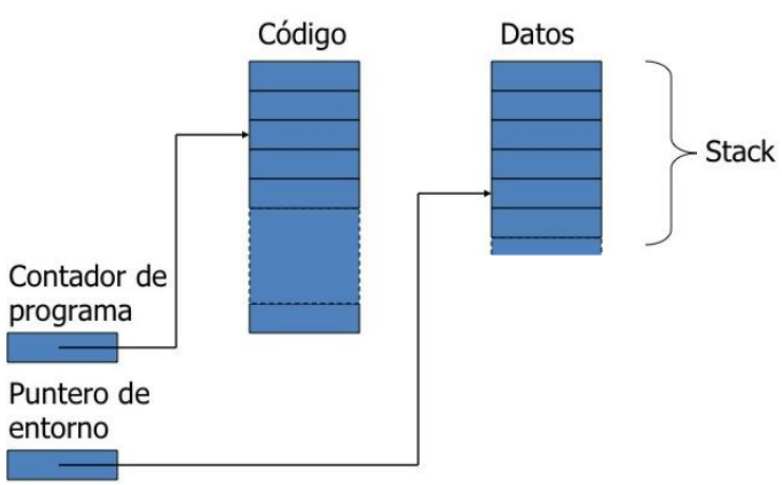
- Tiene tipado fuerte si siempre se detectan los errores de tipo.
  - en tiempo de compilación o ejecución.
    - fuerte = ada - Java - ML - Haskell
    - débil = Fortran - Pascal - C/C++ - Lisp.
  - hace que sea más seguro pero más lento por las comprobaciones dinámicas.

- En general los lenguajes exceden de la expresividad que tienen las gramáticas LC.

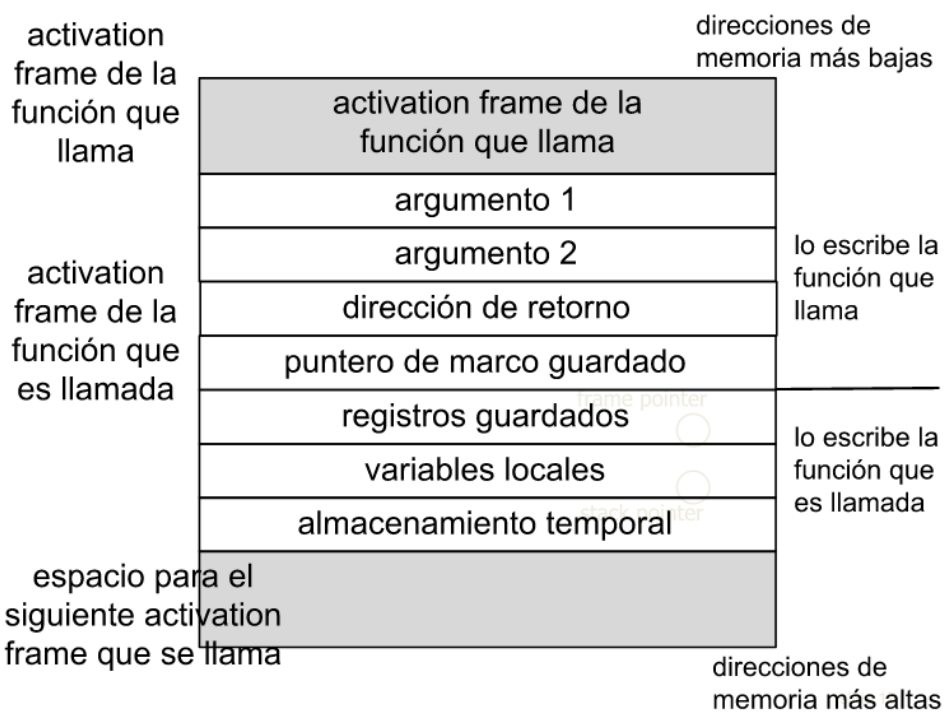
Semántica Operacional.

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

  Los estados son una descripción abstracta de la memoria.



- cada bloque apila un activation record al stack. para las variables locales y puntero de retorno.



estructura de un activation record.

Caps 1, 2, 3 y 5 de la guía.

Paradigma Imperativo.

elementos básicos:

- definición de tipos, declaración de variables (tipadas o no)
  - ubicación y valor de la variable:
    - global, en la pila o stack, puede pertenecer a un bloque).
    - l-valor = ubicación en memoria
    - r-valor = valor guardado en la memoria.
    - identificador = nombre.
- Expresiones y sentencia de asignación:
  - el r-valor de un puntero es el l-valor de otra variable.
- Sentencia control de flujo.
  - Un programa es estructurado si el flujo de control es evidente en la estructura sintáctica del programa.
- Bloque: agrupa código en bloques lógicos.
  - manejo de memoria: al entrar a un bloque se guarda espacio para las variables.
    - al salir se decide si se libera o no.
- Al entrar a un bloque se crea un Activation Record, al salir se lo elimina.

- declaración de funciones o procedimientos.

- Una función tiene valor de retorno.
- un procedimiento NO retorna valor, pero tiene un efecto recordable visible, como cambio en estado de algún valor global.

Act. records para función, tiene lugar para:
- parámetros, variables locales, dirección de retorno.
- control link al AR de quien la llama.

Conceptos:

pasaje de parámetros:

- por valor: la función que llama pasa el r-value (la copia) del argumento a la función llamada.
  - no hay "aliasing" (dos identificadores para una ubicación)
  - la func no puede cambiar el valor de la variable original.
  - mejor para entender cosas.
- por referencia: se pasa el L-value del argumento a la función que es llamada.
  - se asigna la dirección de memoria del argumento al parámetro
  - como "aliasing" tengo dos referencias a una ubicación.
  - la función puede modificar la variable que llama.
  - mejor para entender código.
- por valor-resultado:
  - 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 necesidad:

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 y closures

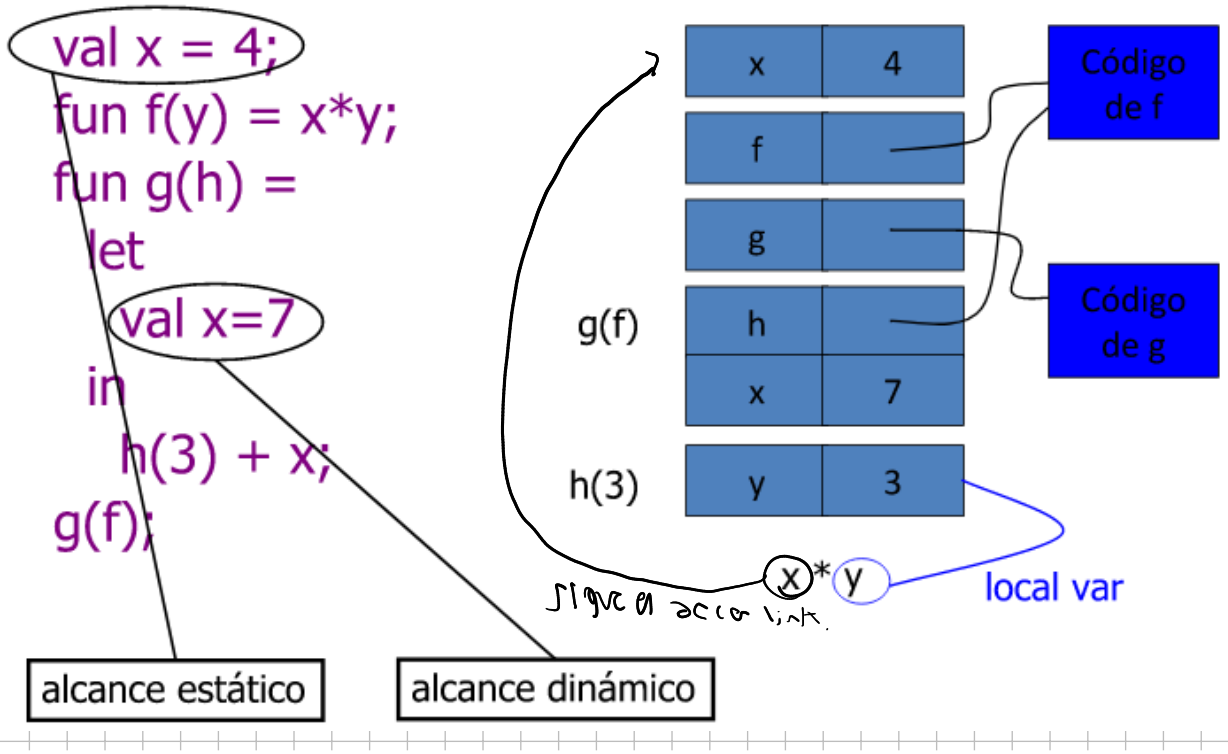
variables locales y globales

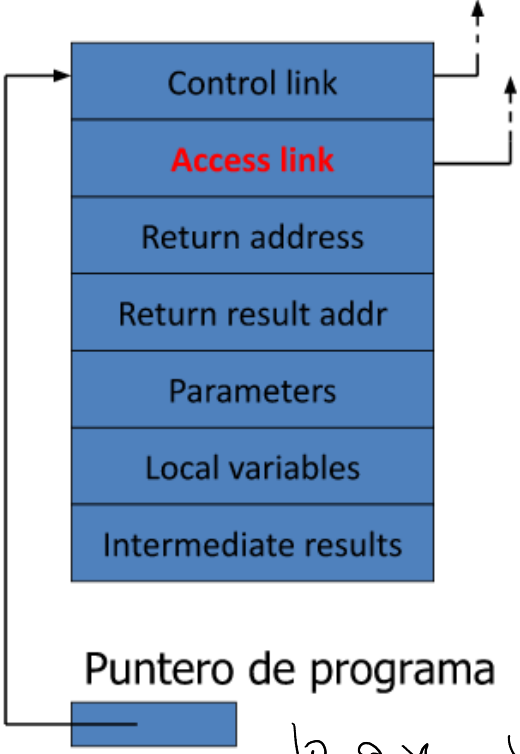
x, y son locales al bloque exterior
z es local al bloque interior
x, y son globales al bloque interior

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

alcance 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





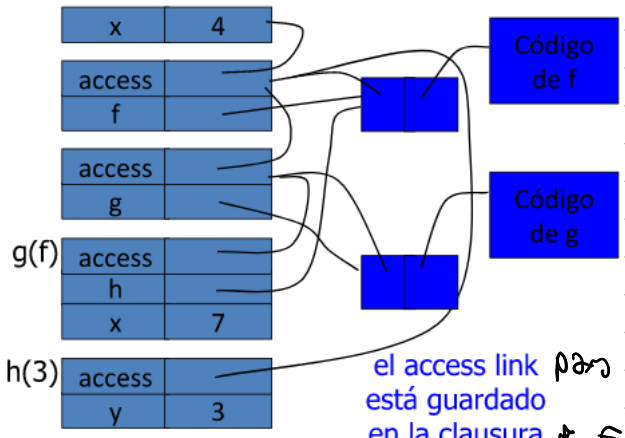
- Control link
  - link al activation record del bloque anterior (el que llama al actual)
  - depende del comportamiento dinámico del programa
- **Access link**
  - link al activation record del bloque que incluye de más cerca al actual, léxicamente, en el texto del programa
  - depende del texto estático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

en ese caso la x que usa f, está en el bloque inmediatamente contenedor de f. por lo que usa el access link para ir a ese AR a buscar el valor de x. que necesita.

Funciones como parámetros:
Cuando pasamos una función como parámetro en realidad pasamos una "clausura" que es una tupla de punteros, uno al AR de la función (que es parámetro) y otro al código de la misma.

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);
```



el access link para obtener el valor de x está guardado en la clausura de f, ya que el bloque que contiene a f, contiene a x.

Funciones de alto orden: una f pasa su argumento o resultado.

Recursión a la cola.

Si g tiene actual f(x) entonces se "recicla" 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

```
fun g(x) = if x>0 then f(x) 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.