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

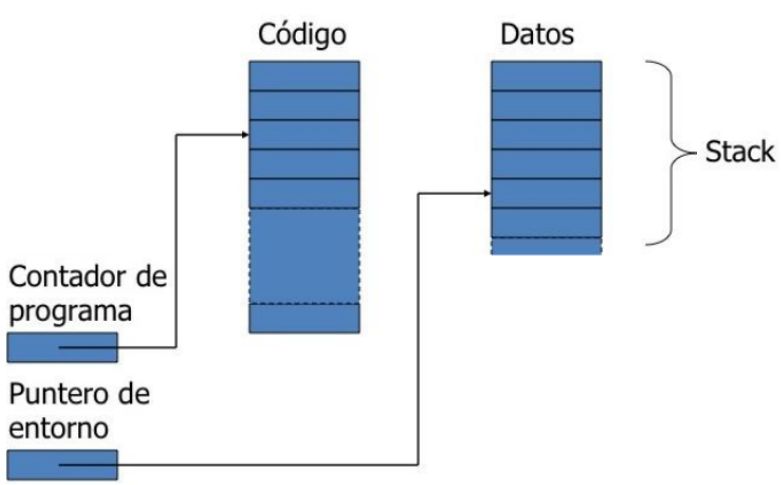
- Tiene tipado fuerte si siempre se detectan los errores de tipo.
  - en tiempo de compilación o ejecución.
    - fuerte = ada - Java - ML - Haskell
    - débil = Fortran - Pascal - C/C++ - Lisp.
  - hace que sea más seguro pero más lento por las comprobaciones dinámicas.

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

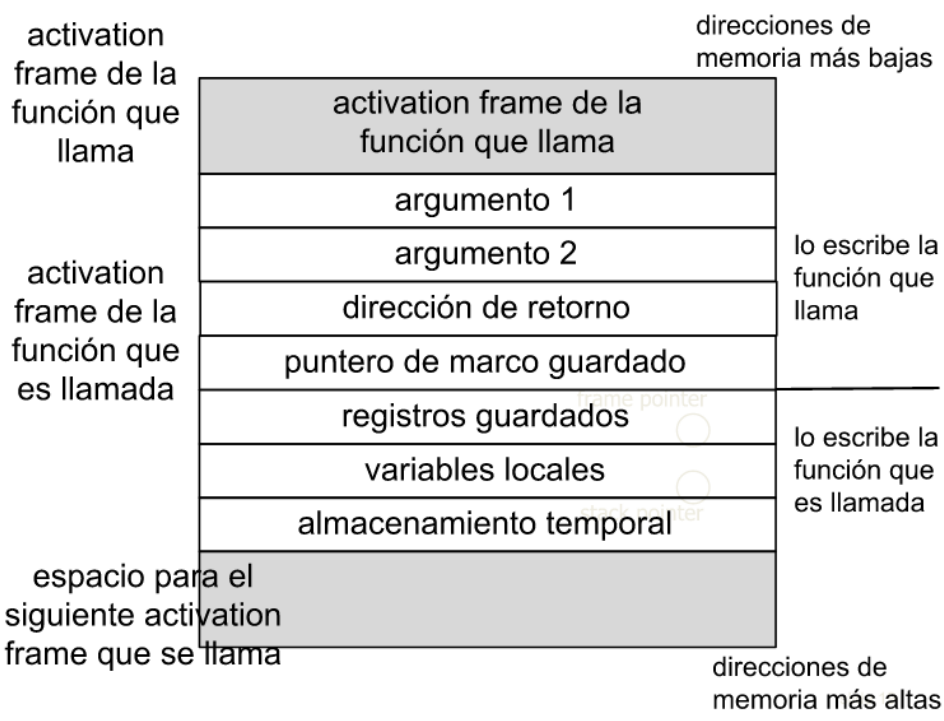
Semántica Operacional.

- 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 clausuras

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);
    };
};
```

busca la declaración en el bloque de programa más cercano, mirando para arriba en el texto.

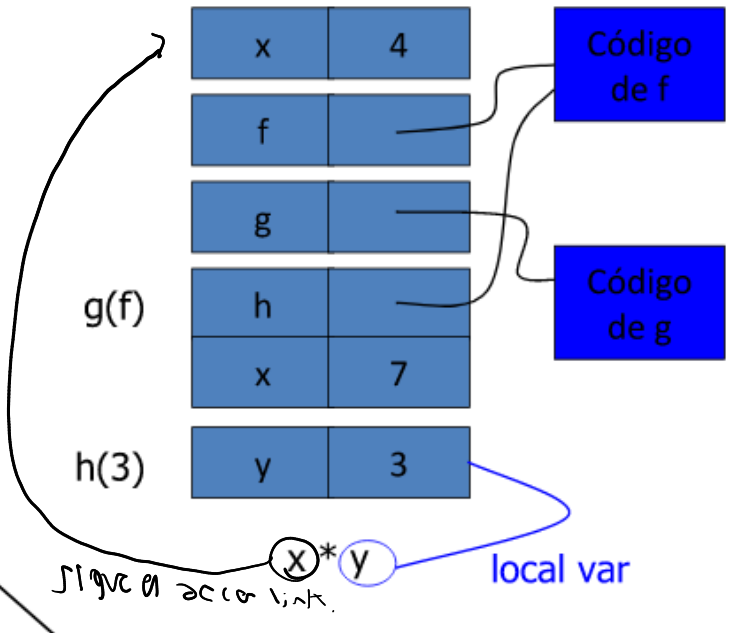
alcance estático: el valor de las variables globales se obtiene del bloque inmediatamente contenedor

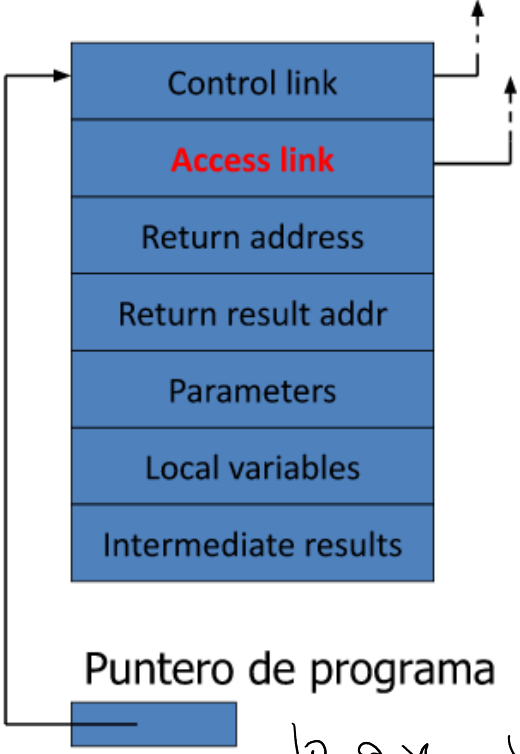
alcance dinámico: el valor de las variables globales se obtiene del activation record más reciente

→ sube por la pila buscando la declaración en el AR más reciente.

→ en la pila.

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





- 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

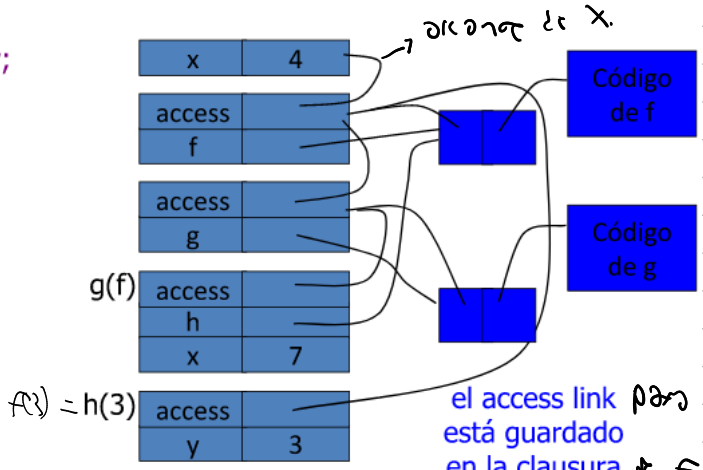
para el → alcance estático.

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. (entorno, 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);
```



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.

cuando en g se llama a f, se usa el AR de f, para mantener la semántica del alcance estático.

funciones de alto orden: una f pasa su argumento o resultado.
- declaradas en cualquier ámbito.

Recursión a la cola.

Si g vuelve actual f(x) entonces se "recicla" el activation record.

Cap 7.3 Mitchell.

- 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 - Imperativa / Declarativo vs Imperativo.

Las construcciones imperativas cambian el valor y las declarativas crean el valor.

- programación imperativa: - efectos secundarios, en funcional se llama destructivo.
- Op declarativa o determinística = mismo in mismo out 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 programas declarativos.

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

- Componentes declarativas ⇒ func. matemáticas ⇒ razonamiento algebraico.

Transparencia referencial.

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

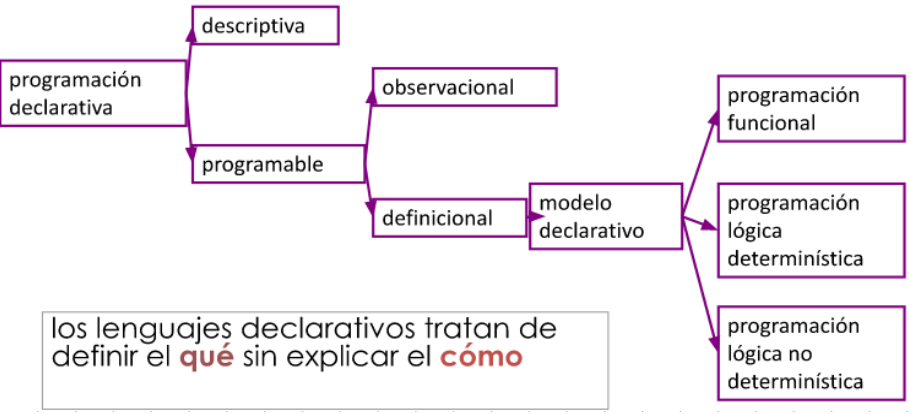
- Todas las comp. declarativas son transparentes, por ende se pueden usar como valor en cualq. lado.

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



los lenguajes declarativos tratan de definir el **qué** sin explicar el **cómo**

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)

El estado siempre está en la computación, incluso en los funcionales.

- implícitos: integran el estado de forma explícita.

para ello los funcionales pueden acceder al estado
- mónadas
- pasar estado como parámetro

Usamos estado
- queremos eficiencia en memoria
- el entorno es determinante.

Mónadas.

- Crea un alcance aislado en programa
- Se permiten op. con efectos secundarios.

No necesariamente los leng declarativos son mejores para representar problemas. Existen problemas que requieren el uso de estados y además la computación y la vida tienen estados. Pero aún no todos los problemas tienen una buena solución declarativa (funcional).

Cap 9 Apunte

* Estática: tiempo comp. Dinámica: tiempo de ejec.

Tipado

tipo: conj de entidades que comparten propiedad.

Sirven para la org. del programa y que es controlado por el compilador (type checking)

errores de tipo:

- Ocurren cuando alguna entidad se usa de una manera que no es coherente con la prop de su tipo.
  - Errores de hardware y de semántica.
  - ej: un patron guardado para representar un tipo, es usado para representar otro tipo.

- Algunas operaciones de tipos (por lo general tipos "básicos") pueden ser optimizadas por el compilador. Si el tipo de la variable es sabido en "compile time".

Lenguajes "Type Safe".

Un lenguaje es type safe. Si se respeta la distinción de tipos. Casos que hacen un lenguaje poco seguro:

- Conteo de tipos, aritmética de punteros. (NO seguro)
- Alloc y Free explícito, dangling pointers: (Con seguro)
- Chequeo de tipos completo. (seguro)

Run-time-check: el código generado por el compilador, hace chequeo de tipos antes de cada operación. Encuentra errores de tipo de cualquier clase. pero no es eficiente.

Compile_Time_check: chequea expresiones por errores potenciales. no compila programas con errores de tipo. Permite código más eficiente.

Encuentra errores que se producirían en run-time. pero también no a todos posibles errores en run-time.

no todos los compile-time-check convergen.

type-check

Run-time

compile-time.

Ventajas

Detecta errores de tipo

||

Elimina runtime tests.

Encuentra errores antes de correr.

Desventajas.

Ineficiente.

Restrictivo por ser conservador.

Inferencia de Tipos.

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

Es una manera lógica para encontrar el tipo de la exp primero.

Soporta polimorfismo.

Algoritmo de inferencia de tipos:

1. asigna tipo a la expresión y cada subexpresión. Por cada expr compuesta, usa una variable de tipo. Para expresiones conocidas: (+, 3) usa el tipo conocido.
2. genera ecuaciones/restricciones de tipos. usando el árbol de la expresión, ej: si una func se le aplica a un arg. y si el arg es tipo `a` pero la func tiene `Int` entonces a es tipo Int.
3. resolver estas igualdades/restricciones. por sustitución desde las hojas de las ramas hasta la raíz.

Polimorfismo y Overloading

3 formas de polimorfismo:

- polimorfismo paramétrico: una función se aplica a cualq argumento, cuyos tipos coinciden con con la expresión y variables de tipos.
- ad hoc polimorfismo (overload): dar implementaciones con dif tipos se definen con el mismo nombre.

Se resuelve cuál se usa, en tiempo de compilación.

Ej el operador `+` puede ser para int -> int -> int o real -> real -> real.