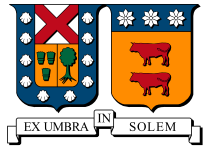
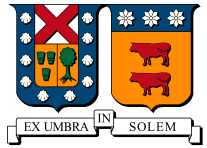


Fundamentos de Lenguajes de Programación

Roberto Díaz
roberto.diazu@usm.cl

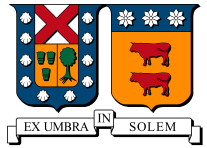
Universidad Técnica Federico Santa María
Departamento de Informática – Santiago, Chile

2021-2



Elementos Básicos de Lenguajes Formales

Concepto Básicos



Lenguaje (formal): Conjunto de cadenas de símbolos restringidas por reglas que son específicas a un lenguaje formal particular.

Sintaxis: Conjunto de reglas que definen las secuencias correctas de los elementos de un lenguaje de programación.

Semántica: Estudio del significado de los signos lingüísticos y de sus combinaciones.

Alfabeto: Conjunto de símbolos, letras o tokens con el cual se puede formar la cadena de un lenguaje.

Gramática (formal): Estructura matemática con un conjunto de reglas de formación que definen las cadenas de caracteres admisibles en un determinado lenguaje formal.

Sintaxis y Semántica

La **sintaxis** de un lenguaje de programación es la descripción precisa de todos los programas gramaticalmente correctos.

> Métodos formales para definir precisamente la sintaxis de un lenguaje son importantes para la correcta implementación de los traductores y programas (p.e. usando BNF).

Por otro lado, **semántica** se refiere al significado del lenguaje, y no a la forma (sintaxis). Provee reglas para interpretar la sintaxis, estableciendo restricciones para la interpretación de lo declarado.

Sintaxis léxica: Define reglas para símbolos básicos o palabras denominados tokens (p.e. identificadores, literales, operadores y puntuación).

Sintaxis concreta: Se refiere a una representación real de un programa usando símbolos del alfabeto. Una tarea es reconocer si el programa está gramaticalmente correcto.

Sintaxis abstracta: Lleva sólo la información esencial del programa (p.e. elimina puntuación). Útil para la traducción posterior y optimización en la generación de código.

Reconocimiento: Reconoce si una cadena de entrada pertenece al lenguaje. Formalmente, si se tiene un lenguaje **L** y un alfabeto Σ, una máquina **R** capaz de leer una cadena cualquiera de símbolos de Σ es un "reconocedor" si acepta toda cadena válida y rechaza toda cadena inválida de **L**.

> Ejemplo: El analizador sintáctico de un lenguaje es en principio un reconocedor de que un programa está sintácticamente correcto (pertenece al lenguaje).

Generación: Genera todas las posibles cadenas que pertenecen al lenguaje. No es en general práctico, sin embargo es útil para especificar formalmente una sintaxis entendible.

> Ejemplo: Notaciones Backus Nauer Form (BNF) y Extended BNF (EBNF) permiten generar todos los programas válidos.

Semántica Estática: Define restricciones sobre la estructura de los textos válidos que son difíciles de expresar en el formalismo estándar de la sintaxis. Define reglas que se pueden verificar en tiempo de compilación. “Gramática con Atributos” se aplica para especificarla.

> Ejemplo: Verificar que variable ha sido declarada antes de usarla; que los rótulos de una sentencia **switch** sean diferentes; que un identificador es usado en el contexto correcto; etc.

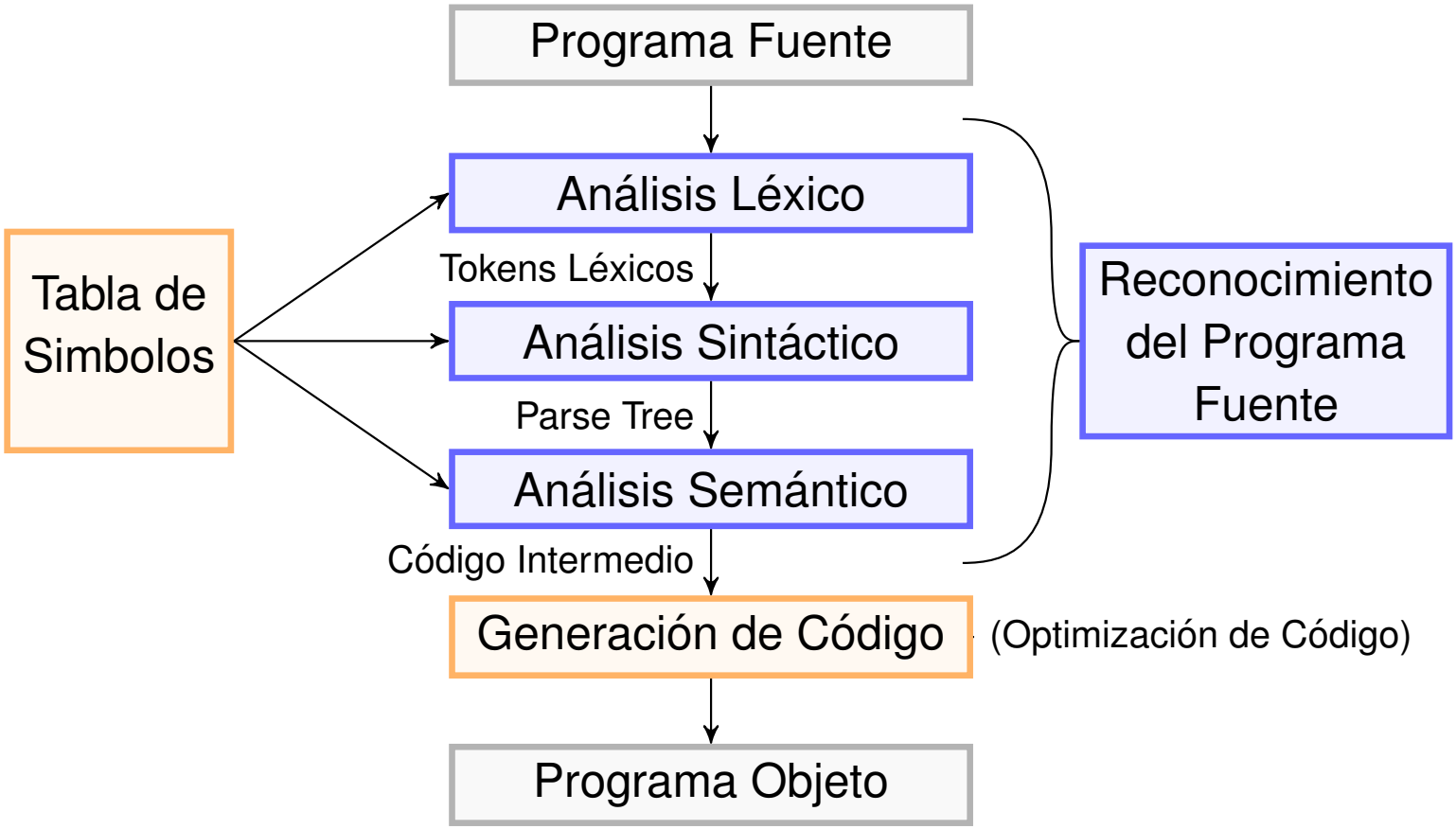
Semántica Dinámica (o de ejecución): Define el comportamiento que producen los diferentes constructos del lenguaje. Se puede especificar (informalmente) mediante lenguaje natural, sin embargo es deseable formalizarla para mayor precisión (métodos formales tienen en la industria aplicaciones limitadas para el diseño e implementación de lenguajes de programación).

Semántica Operacional: Significado se establece especificando los efectos de la ejecución de un programa o sentencia en una máquina. Interesa el cómo se produce el efecto de una computación. Típicamente se usa un lenguaje intermedio para especificar el comportamiento.

Semántica Denotacional: Significado se modela con objetos matemáticos que representan los efectos de una ejecución. Interesa el efecto de una computación, no el cómo.

Semántica Axiomática: Especifica propiedades de los efectos de ejecución mediante afirmaciones lógicas. Así puede que se ignoren ciertos aspectos de la ejecución. Se aplica en verificación (formal) de la correctitud de un programa.

Proceso de Compilación



Gramática

Una **gramática** se define por una 4−tupla G=(V, Σ, P, S)

- **V:** Conjunto finito de símbolos no terminales o variables. Cada variable define un sub-lenguaje de G.
- **Σ:** Conjunto finito de símbolos terminales, que define el alfabeto de G.
- **P:** Relación finita $V \rightarrow (V \cup \Sigma)^*$, donde cada miembro de P se denomina regla de producción de G.
- **S:** Símbolo de partida, donde $S \in V$.

Observación: Gramática permite expresar formalmente la sintaxis de un lenguaje (formal).

Gramática

G=(V, Σ, P, S) con:

- V=$\{$I, X, Y, Z$\}$, Σ=$\{$a, b$\}$, S=I

donde P tiene 4 reglas de producción:

1. $I \rightarrow X \mid Y$
2. $X \rightarrow ZaX \mid ZaZ$
3. $Y \rightarrow ZbY \mid ZbZ$
4. $Z \rightarrow aZbZ \mid bZaZ \mid \varepsilon$

Observación: Esta gramática genera todas las cadenas con un número desigual de simbolos ‘a’ y ‘b’.

Jerarquía de Chomsky

Clasificación de diferentes tipos de gramáticas formales que generan lenguajes formales (Noam Chomsky, 1956).

- Tipo 0: Lenguajes Recursivamente Enumerables (la categoría más general que puede ser reconocida).
- Tipo 1: Lenguajes Sensibles al Contexto.
- Tipo 2: Lenguajes Independientes de Contexto (basado en una gramática libre de contexto).
- Tipo 3: Lenguajes Regulares (basado en gramática regular y que se puede obtener mediante expresiones regulares).

En lenguajes de programación, los dos últimos son los más usados.

Backus-Naur Form (BNF)

La notación BNF es un metalenguaje, creado por John Backus (1959), que permite especificar formalmente gramáticas libres de contexto.
Reglas de producción se especifican como:

<símbolo> = <expresión con símbolos>

Características:

- Lado izquierdo corresponde a un símbolo no terminal.
- Lado derecho (expresión) representa posible sustitución para símbolo no terminal.
- Expresión usa "|" para indicar alternación (opciones).
- Símbolos terminales nunca aparecen a la izquierda.
- Normalmente, primera regla corresponde a símbolo no terminal de partida.

BNF
Ejemplo

```
R1: <program>   ::= "begin" <stmt_list> "end"
R2: <stmt_list> ::= <stmt> | <stmt> ";" <stmt_list>
R3: <stmt>      ::= <id> "=" <exp>
R4: <id>        ::= "A" | "B" | "C"
R5: <exp>       ::= <id> | <exp>"+"<exp> | <id>"*"<exp>
                  | "("<exp>")"
```

BNF
Ejemplo

```
R1: <program>   ::= "begin" <stmt_list> "end"
R2: <stmt_list> ::= <stmt> | <stmt> ";" <stmt_list>
R3: <stmt>      ::= <id> "=" <exp>
R4: <id>        ::= "A" | "B" | "C"
R5: <exp>       ::= <id> | <exp>"+"<exp> | <id>"*"<exp>
                  | "("<exp>")"
```

```
G = (V, A, P, S)
V = {program, stmt_list, stmt, id, exp}
A = {'begin','end',';','=','+','*','(',')','A','B','C'}
P = {R1, R2, R3, R4, R5}
S = program
```

BNF Extendido (EBNF)

Extensiones:

- Elementos opcionales son presentados entre corchetes `[...]`
- Paréntesis (redondos) permiten agrupar elementos
- Alternación puede usarse en una regla entre paréntesis `(exp | exp | ...)`
- Repetición de elementos (0 ó más) se indican con llaves `{...}`

Notación:

- Concatenación puede ser simple o usando coma (`,`)
- Tokens (terminales) se indican de diferentes maneras: en negrita, entre comillas.
- Nombres de reglas (no terminales) se colocan en texto normal o entre paréntesis angulares.
- Reglas pueden terminar con punto y coma (`;`)

EBNF
Ejemplo

```
R1: program   = 'begin' , stmt_list , 'end' ;
R2: stmt_list = stmt | stmt , ';' , stmt_list ;
R3: stmt      = id , '=' , exp ;
R4: id        = 'A' | 'B' | 'C' ;
R5: exp       = id | exp , '+' , exp | id , '*' , exp
              | '(' , exp , ')' ;
```

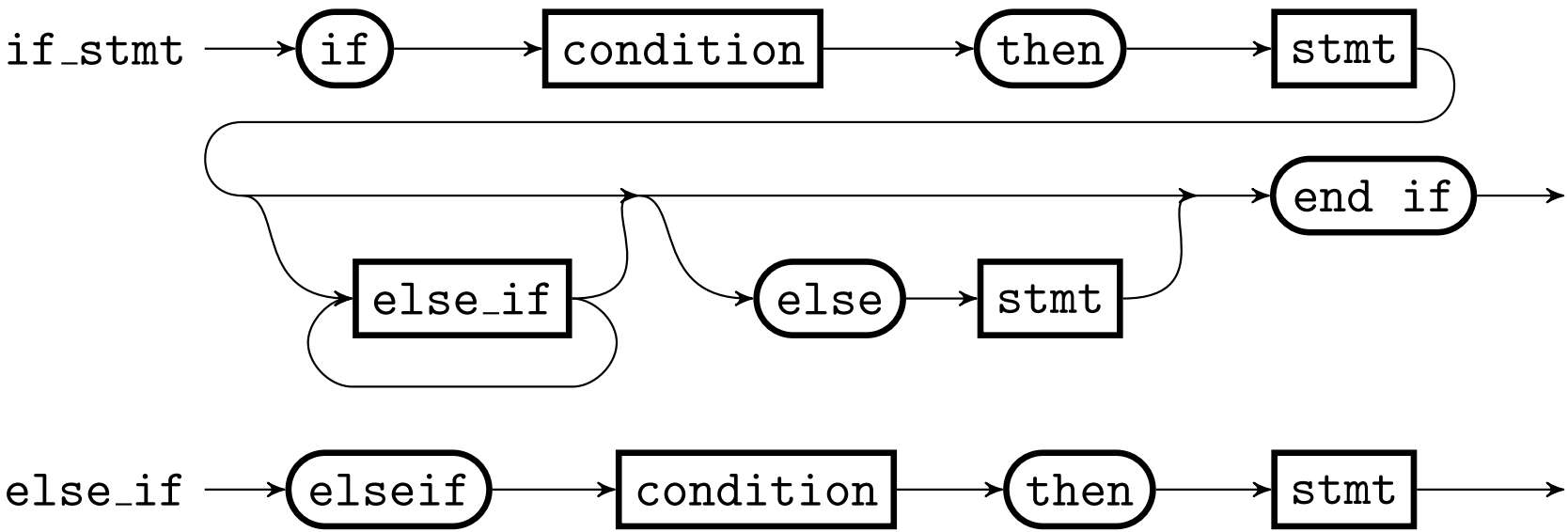
```
G = (V, A, P, S)
V = {program, stmt_list, stmt, id, exp}
A = {'begin','end',';','=','+','*','(',')','A','B','C'}
P = {R1, R2, R3, R4, R5}
S = program
```

EBNF
Ejemplo

```
if_stmt = 'if', condition, 'then', stmts
          , ['else', stmts] ;

identifier = letter , {letter|digit} ;

for_stmt = 'for' , identifier , 'in' , identifier , ':'
          , stmt ;
```

Diagramas Sintácticos

```
if_stmt = 'if' , condition , 'then' , stmts , {else_if}
          , ['else' stmts] , 'end' , 'if' ;

else_if = 'elseif' , condition , 'then' , stmts ;
```



Expresiones Regulares

Las **expresiones regulares** permiten describir patrones de cadenas de caracteres. Son útiles para especificar y reconocer tokens. Cada expresión regular tiene asociado un autómata finito.

Operaciones básicas:

- **Concatenación:** expresada por la secuencia de los elementos.
- **Cuantificación:** Permite especificar frecuencias; * (0 ó más veces), + (al menos una vez) y ? (a lo más una vez).
- **Alternación:** Permite elegir entre alternativas (|)
- **Agrupación:** Permite agrupar varios elementos con paréntesis redondos.

Ejemplo:

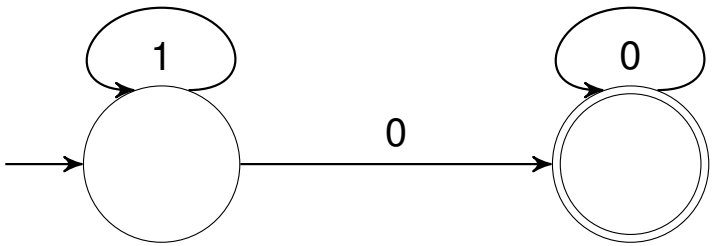
- Expresión **(a|b)*c** reconoce **c**, **ac**, **bc**, **aaaabbbc**, etc.

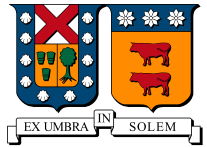
Autómata Finito

Un Autómata Finito es una 5-tupla (Q, Σ, δ, q_0, F)

- Q: Conjunto finito de estados (circulo)
- Σ: un alfabeto finito (que definen una cadena de entrada)
- δ: Función de transición Q X $\Sigma \rightarrow$ Q (flecha con etiqueta)
- q_0: Estado inicial (con una flecha de entrada)
- F: Estados finales o de aceptación F $\subseteq$ Q (con doble circulo)

Ejemplo: Autómata Finito equivalente a expresión regular: `1*0+`





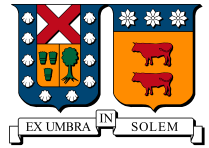
Descripción de sintaxis

Proceso de Análisis Léxico y Sintáctico

Proceso de reconocimiento de sintaxis tiene típicamente dos fases:

- ***Scanning* (análisis léxico):** El traductor lee secuencia de caracteres de programa de entrada, y se reconocen ***tokens***.
  - Se asume conocido conjunto de caracteres válidos.
  - Expresiones regulares permiten reconocer ***tokens***.
- ***Parsing* (análisis sintáctico):** El traductor procesa los ***tokens***, determinando si el programa está sintácticamente correcto.
  - Construcción de un **Árbol Sintáctico** (*parse tree*), que permite analizar (y reconocer) si el programa es sintácticamente correcto.

Tokens



Tipos de tokens:

- Palabras claves y reservadas (p.e. `if`, `while`, `struct`)
- Literales y constantes (p.e. `3.14` y `"hola"`)
- Identificadores (p.e. `i`, `suma`, `getBalance`)
- Símbolos de operadores (p.e. `+`, `*=` y `<=`)
- Símbolos especiales (p.e. blancos, delimitadores y paréntesis)
- Comentarios (p.e. `/*un comentario*/`)

Aspectos prácticos:

- No confundir palabras reservadas con identificadores predefinidos (e.g. `int` en C)
- Reconocimiento del largo de un identificador
- Formato Libre vs. Formato Fijo (e.g. fin de línea, posición o indentación afectan el significado y reconocimiento de tokens).
- Tokens se pueden reconocer fácilmente aplicando expresiones regulares.

Conceptos

La **sintaxis** establece estructura, no significado; sin embargo el significado de un programa está relacionado con su **sintaxis**.

El proceso de asociar la semántica a un constructor lingüístico es denominada "**semántica dirigida por sintaxis**".

Un **árbol sintáctico** (*parse tree*) permite establecer una estructura sintáctica y asociarlo a un significado.
Veremos a continuación dos temas relevantes:

- Definición de árbol sintáctico (*parse tree*)
- Resolución de ambigüedad semántica mediante reglas de **asociatividad** y **precedencia**.

Arbol Sintáctico

Convenciones

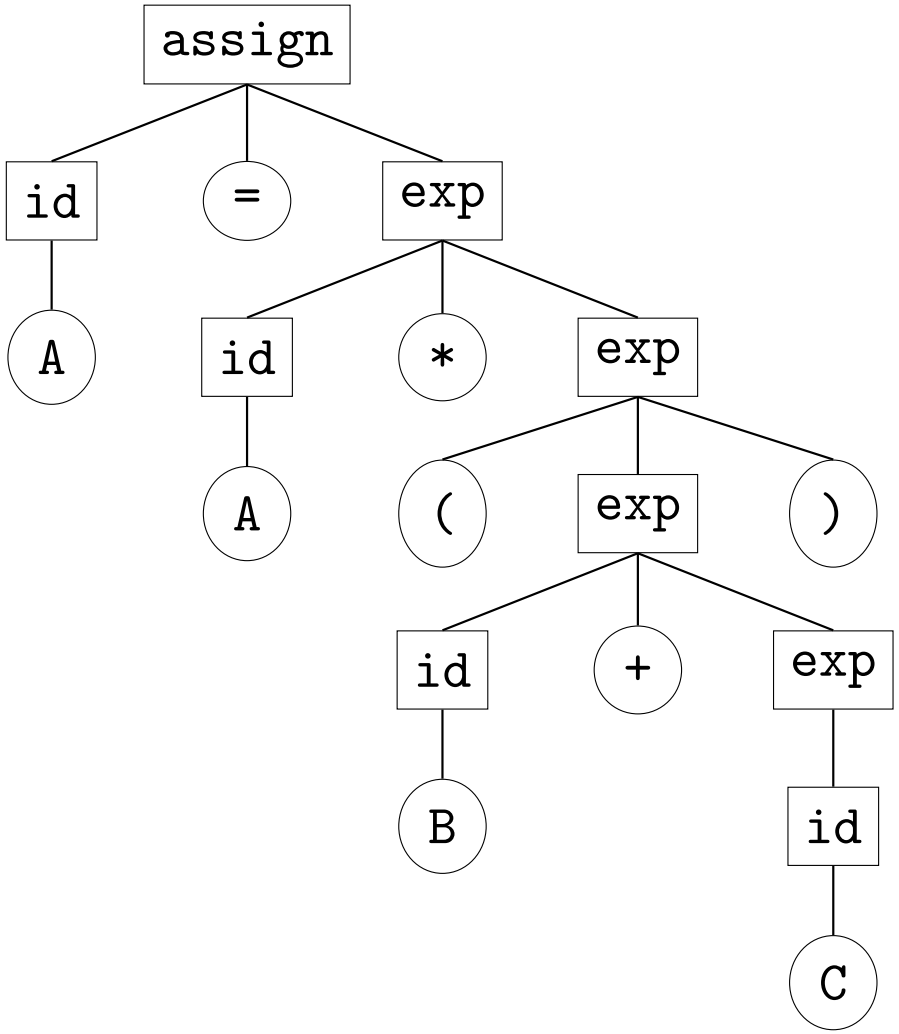
- Cuadrado : No Terminal
- Óvalo: Terminal

Una simple Gramática

```
assign = id, '=', exp;
id = 'A' | 'B' | 'C';
exp = id, '+', exp
      | id, '*', exp
      | '(', exp, ')'
      | id;
```

Expresión:

```
A = A * (B + C)
```



Arbol Sintáctico
Ambigüedad

Gramática

```
assign = id,'=',exp;
id = 'A' | 'B' | 'C';
exp = exp,'+',exp | exp,'*',exp | '(',exp,')' | id
```

Expresión:

```
A = A * B + C
```

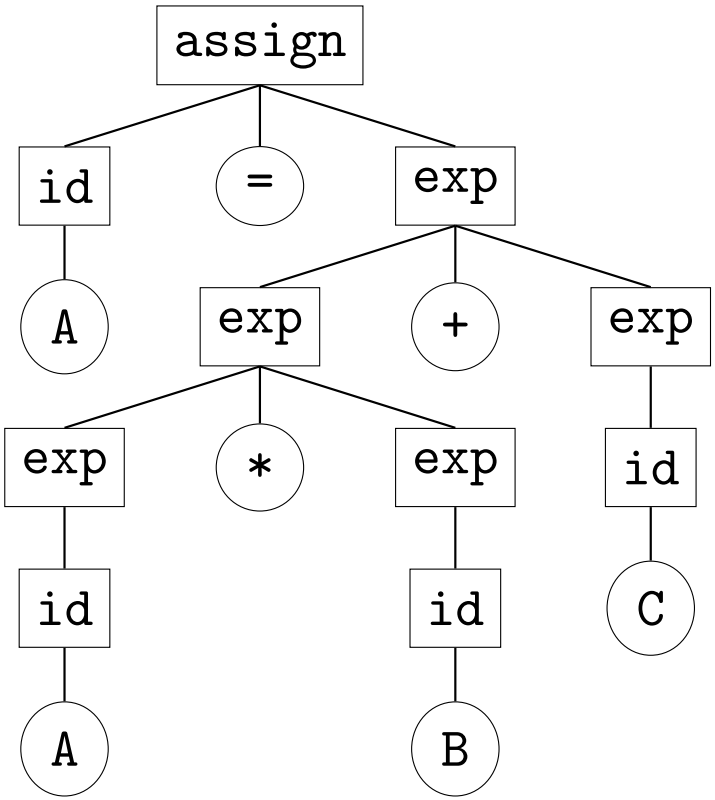
Arbol Sintáctico
Ambigüedad

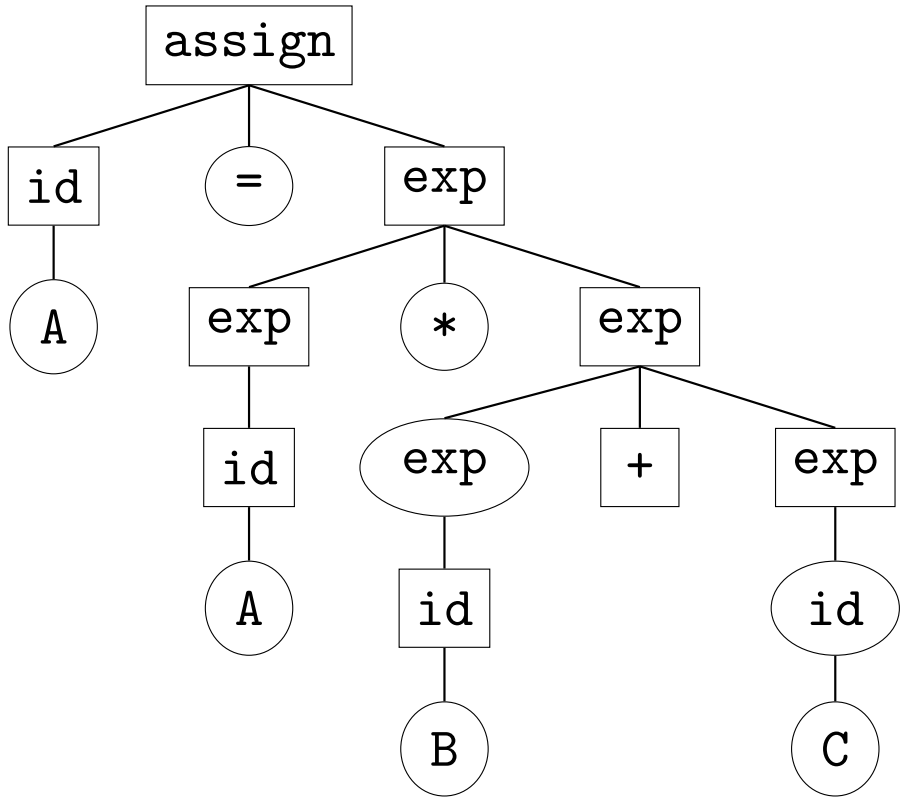
Gramática

```
assign = id,'=',exp;
id = 'A' | 'B' | 'C';
exp = exp,'+',exp | exp,'*',exp | '(',exp,')' | id
```

Expresión:

```
A = A * B + C
```





Precedencia y Asociatividad de Operadores

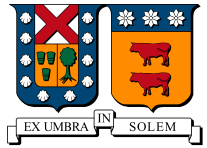
En una expresión pueden aparecer diferentes operadores, los cuales pueden ser evaluados en diferentes órdenes, produciendo ambigüedad. Operador que se evalúa primero queda más abajo en el árbol.

Asociatividad: Se identifican los operadores en la entrada del parser y se aplica una regla de ordenar la evaluación.

- **Asociatividad por la izquierda:** Se prioriza la evaluación de izquierda a derecha.
- **Asociatividad por la derecha:** Se prioriza evaluación de derecha a izquierda.

Precedencia: Se clasifican categorías de operadores y se ordenan de mayor a menor prioridad estas categorías.

- Para un programa la evaluación se hace en orden de mayor a menor precedencia.
- A igual precedencia de dos operadores, se puede resolver por asociatividad.



Nombres, Ligado y Ámbito

Nombres

Abstracción básica que permite mediante una cadena de caracteres referenciar entidades o constructos del lenguaje (e.g. variables, constantes, procedimientos y parámetros). Se les denomina también “identificadores”.

Aspectos de Diseño:

- Largo del nombre
- Tipos de caracteres aceptados (e.g. conector _)
- Sensibilidad a mayúsculas y minúsculas
- Palabras reservadas y palabras claves (las últimas a veces se pueden redefinir o representan identificadores predefinidos)
- Convención especial para ciertas entidades (e.g. $... para variables en PHP)

Variables

Una variable es una abstracción de un objeto de memoria, que tiene varios atributos asociados, tal como los siguientes:

- **Nombre:** identificador como lo ya discutido.
- **Dirección (l-value):** indica dónde está localizado. La dirección puede cambiar en el tiempo. Puede haber varias variables asociadas a una misma dirección.
- **Valor (r-value):** contenido (abstracto) de la memoria.
- **Tipo:** tipo de dato que almacena y define operaciones válidas.
- **Tiempo de vida:** ¿cuándo se crea y se destruye? Se vincula al ligado de memoria.
- **Ámbito:** ¿dónde es visible y se puede referenciar?

Ligado (*Binding*)

El ligado es el proceso de asociación de un atributo a una entidad del lenguaje (e.g. variable). Se consideran diferentes tipos de ligado, como

- **Ligado de tipo:** asociación de un tipo (de datos).
- **Ligado de memoria:** dirección asociada (determina valor de variable o código de un procedimiento u operador).

Ejemplo:

- **Variable:** <Dirección, tipo> $\rightarrow$ <Nombre>
- **Operador:** <Código> $\rightarrow$ <Símbolo>

Tiempo de Ligado

Según cuándo suceda la asociación entre una variable u otra entidad nombrada; se clasifica según si sucede antes o durante la ejecución del programa como:

- **Estático**: ligado ocurre durante la compilación, enlace (*linking*) o carga del programa.
- **Dinámico**: ligado se realiza durante la ejecución del programa (*runtime*), pudiendo cambiar en el intertanto.

Ligado de Tipo

Variables deben ser ligadas a un tipo de datos antes de usarlas. El tipo puede ser ligado por declaración explícita (sentencia) o implícita (convenciones).

- C y C++ hacen la diferencia entre declaración y definición.

Ligado de Tipo
Clasificación

Ligado Estático: Puede ser con declaración explícita o implícita

- Lenguajes modernos compilados usan preferentemente declaración explícita (mejor confiabilidad).
- Algunos lenguajes asocian por convención de nombres un tipo (e.g. PERL identificadores que comienzan con $ son escalares).

Ligado Dinámico: Ligado con declaración implícita en el momento de la asignación. Usado preferentemente por lenguajes de scripting (e.g. Python).

- **Ventaja:** Permite programar en forma genérica (procesar datos con tipo asociado desconocido hasta su uso).
- **Desventaja:** Disminuye capacidad de detección de errores y aumento del costo (de ejecución).

Ligado de Memoria

Características

- Un **nombre** puede ser asociado con diferentes **direcciones** en diferentes partes (**ámbito**) y tiempo de ejecución de un programa.
- El mismo **nombre** puede ser usado para referirse a diferentes objetos de memoria (**direcciones**), según el **ámbito**.
- Diferentes **nombres** pueden referirse a un mismo objeto (**alias**), lo que puede provocar **efectos laterales**.

Ejemplos:

- Unión y punteros en C y C++
- Variables dinámicas en Python

Taxonomía de la Memoria

El carácter fundamental de un lenguaje está determinado por cómo se administra la memoria, para establecer el tipo de ligado de memoria que se realiza con las variables.

Los “objetos de memoria” asignados, que pueden corresponder a datos, pero también a código, se crean y destruyen en diferentes momentos de la ejecución (ciclo de vida).

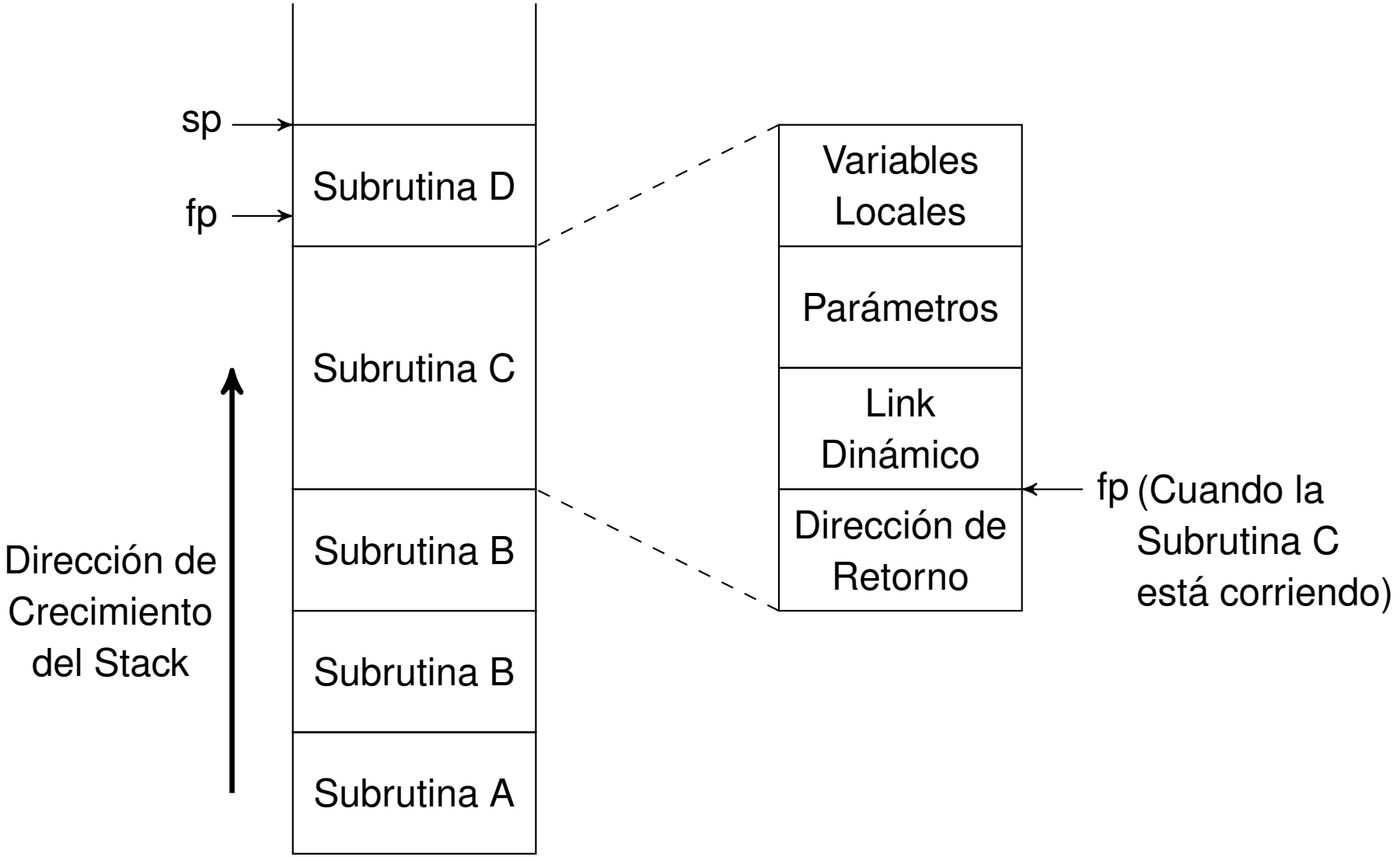
Taxonomía de la Memoria
Clasificación

Memoria estática: área de memoria que permite realizar ligado estático de una variable a la memoria.

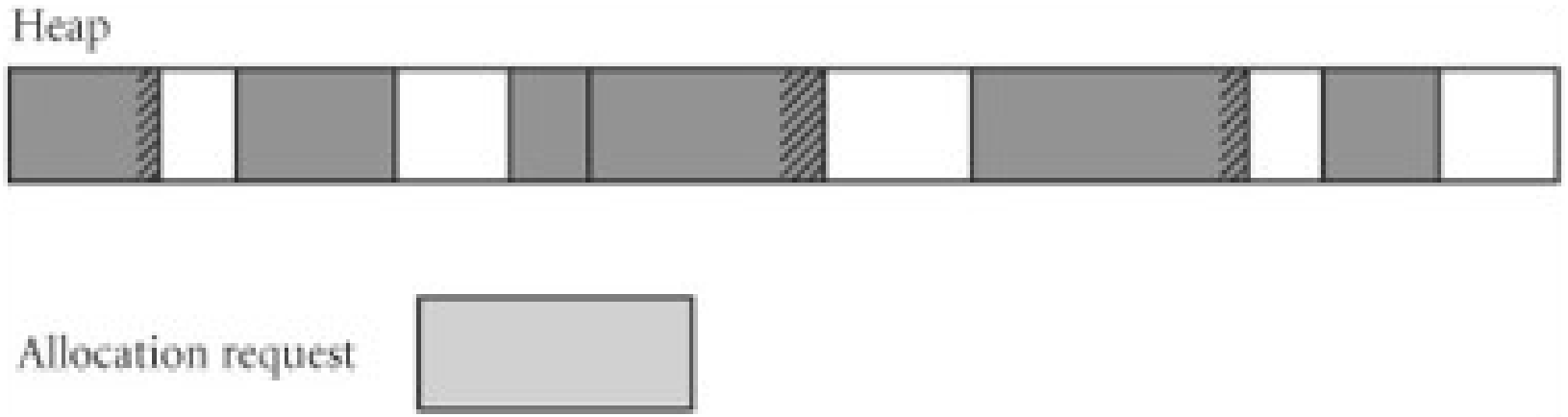
Memoria stack: área dinámica de memoria que permite mantener objetos que se asignan y liberan automáticamente al activar o desactivar un ambiente de ejecución (p.e. invocación a procedimiento o ejecución de un bloque).

Memoria heap: permite crear y eliminar objetos de memoria dinámicamente, y son referenciadas indirectamente mediante una variable como un puntero o una referencia.

Ejemplo Memoria de Stack



Ejemplo Memoria de Heap

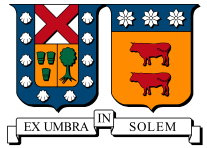


Taxonomía de Variables según Ligado de Memoria

El tipo de ligado a memoria de una variable se clasifica fundamentalmente por el tiempo de vida del ligado y el tipo de memoria asignado. Se reconocen las siguientes categorías:

- **Variable Estática**
- **Variable Dinámica de Stack (o automática)**
- **Variable Dinámica de Heap**

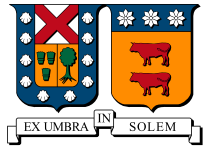
Taxonomía de Variables según Ligado de Memoria
Variables Estáticas



Variables con ligado estático que usan la memoria estática. Su acceso puede ser global, pero también se puede restringir a un ámbito local (e.g. calificador `static` en C).

Ventajas:

Taxonomía de Variables según Ligado de Memoria
Variables Estáticas



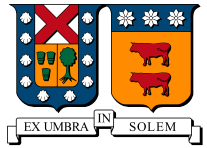
Variables con ligado estático que usan la memoria estática. Su acceso puede ser global, pero también se puede restringir a un ámbito local (e.g. calificador `static` en C).

Ventajas:

- Útil para definir variables globales y compartidas.
- Para variables sensibles a la historia de un subprograma (e.g. uso de `static` en variables de funciones C y C++).
- Acceso directo permite mayor eficiencia.

Desventajas:

Taxonomía de Variables según Ligado de Memoria
Variables Estáticas



Variables con ligado estático que usan la memoria estática. Su acceso puede ser global, pero también se puede restringir a un ámbito local (e.g. calificador `static` en C).

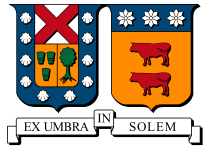
Ventajas:

- Útil para definir variables globales y compartidas.
- Para variables sensibles a la historia de un subprograma (e.g. uso de `static` en variables de funciones C y C++).
- Acceso directo permite mayor eficiencia.

Desventajas:

- Falta de flexibilidad.
- No soportan recursión.
- Impide compartir memoria entre diferentes variables (no se reasigna).

Taxonomía de Variables según Ligado de Memoria
Variable Dinámica de Stack



Se asigna memoria dinámicamente desde el *runtime system support*, específicamente desde el registro de activación, cuando se ejecuta el ámbito de declaración asociado. Soporta muy bien la recursión. Ligado de tipo es normalmente estático.

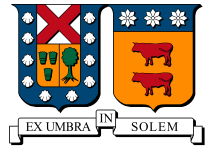
Ámbitos de declaración:

- Procedimiento, función, método o bloque.

Ejemplos:

- En C y C++ las variables definidas en una función son de este tipo (denominadas variables automáticas).
- En Pascal una variable se puede declarar en un bloque, y se activa sólo durante la ejecución del bloque.

Taxonomía de Variables según Ligado de Memoria
Variable Dinámica de Heap (Explícitas)

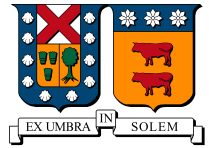


La memoria se asigna y libera en forma explícita desde el Heap por el programador, usando un operador del lenguaje o una llamada al sistema. Su acceso ocurre sólo mediante variables de puntero o referencia.

- C++ dispone del operador `new` y `delete`.
- C usa llamada al sistema `malloc()` y `free()`.
- Java posee asignación explícita mediante operador `new`, pero no permite la liberación explícita.

Ventajas:

Taxonomía de Variables según Ligado de Memoria
Variable Dinámica de Heap (Explícitas)



La memoria se asigna y libera en forma explícita desde el Heap por el programador, usando un operador del lenguaje o una llamada al sistema. Su acceso ocurre sólo mediante variables de puntero o referencia.

- C++ dispone del operador `new` y `delete`.
- C usa llamada al sistema `malloc()` y `free()`.
- Java posee asignación explícita mediante operador `new`, pero no permite la liberación explícita.

Ventajas:

- Útil para estructuras dinámicas usando punteros.
- Gran control del programador.

Desventajas:

Taxonomía de Variables según Ligado de Memoria
Variable Dinámica de Heap (Explícitas)

La memoria se asigna y libera en forma explícita desde el Heap por el programador, usando un operador del lenguaje o una llamada al sistema. Su acceso ocurre sólo mediante variables de puntero o referencia.

- C++ dispone del operador `new` y `delete`.
- C usa llamada al sistema `malloc()` y `free()`.
- Java posee asignación explícita mediante operador `new`, pero no permite la liberación explícita.

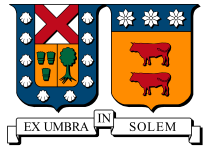
Ventajas:

- Útil para estructuras dinámicas usando punteros.
- Gran control del programador.

Desventajas:

- Dificultad en su uso correcto y en la administración de la memoria.

Taxonomía de Variables según Ligado de Memoria
Variable Dinámica de Heap (Implícitas)

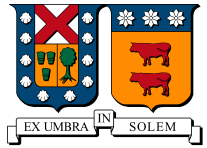


Se liga dinámicamente a la memoria del Heap cada vez que ocurre una asignación.

- Ejemplos: Python y JavaScript.

Ventajas:

Taxonomía de Variables según Ligado de Memoria
Variable Dinámica de Heap (Implícitas)



Se liga dinámicamente a la memoria del Heap cada vez que ocurre una asignación.

- Ejemplos: Python y JavaScript.

Ventajas:

- Alto grado de flexibilidad.
- Un nombre sirve para cualquier cosa.
- Permite escribir código genérico.

Desventajas:

Taxonomía de Variables según Ligado de Memoria
Variable Dinámica de Heap (Implícitas)

Se liga dinámicamente a la memoria del Heap cada vez que ocurre una asignación.

- Ejemplos: Python y JavaScript.

Ventajas:

- Alto grado de flexibilidad.
- Un nombre sirve para cualquier cosa.
- Permite escribir código genérico.

Desventajas:

- Alto costo de ejecución para mantener atributos de la variable.
- Pérdida de cierta capacidad de detección de errores.

Constantes con Nombre

Variable que se liga a un valor sólo una vez. Permanece luego inmutable. Se usa el mismo procedimiento de inicialización.

Ejemplos:

- **C y C++:** `const double pi = 3.14159265;`
- **Java:** `final int largo = 256;`

Observación: si expresión de inicialización contiene una variable, ésta debiera realizarse dinámicamente.

Ventajas:

Constantes con Nombre

Variable que se liga a un valor sólo una vez. Permanece luego inmutable. Se usa el mismo procedimiento de inicialización.

Ejemplos:

- **C y C++:** `const double pi = 3.14159265;`
- **Java:** `final int largo = 256;`

Observación: si expresión de inicialización contiene una variable, ésta debiera realizarse dinámicamente.

Ventajas:

- Mejora la legibilidad y confiabilidad de los programas. También ayuda a mejorar control de parámetros.

Ámbito

Un ámbito corresponde al rango de sentencias en el cual un nombre es visible.

- Una variable es visible en una sentencia si ésta puede allí ser referenciada.
- Lenguajes definen reglas de visibilidad para ligar un nombre a determinada declaración o definición.
- Ámbito normalmente está determinado por un subprograma o bloque.

Ámbito

Extensión de visibilidad:

- **Nombre local:** nombre es declarado en determinado bloque o unidad de programa, y sólo puede ser referenciado en él.
- **Nombre no local:** es visible dentro de un bloque o unidad de programa, pero ha sido declarado fuera de él. Una categoría especial es un nombre global.

Tiempo de determinación del ámbito del nombre:

- **Estático:** puede ser determinado en tiempo de compilación.
- **Dinámico:** sólo se determina en tiempo de ejecución.

Ámbito Estático

Un lenguaje posee ámbito estático cuando el ámbito puede ser determinado antes de la ejecución, p.e. en tiempo de compilación, analizando código fuente.

- Introducido por ALGOL 60, y usado en la mayoría de los lenguajes imperativos, pero también en algunos no imperativos.

Tipos de ámbitos para subprogramas:

- **Anidados:** se permite el anidamiento de subprogramas en una jerarquía de ámbitos, que define una ascendencia estática (e.g. Python. JavaScript, Java y Scheme).
- **No anidados:** no se permite la declaración de un subprograma dentro de otro (e.g. C).

Anidamiento y Ocultamiento

En ámbitos anidados la correspondencia entre una referencia a un nombre y su declaración se busca desde el ámbito más cercano al más externo.

- Un elemento con nombre se oculta en la medida que exista otro elemento con igual nombre en un ámbito más cercano.

Algunos lenguajes permiten acceder elementos con nombre ocultos usando una estructura de nombres jerárquica.

- Ejemplo: operador de ámbito de C++:

```
<nombre-ambito>::<nombre-elemento>
```

Bloques

Permite la declaración de nombres dentro de un bloque de sentencias, creando ámbitos estáticos dentro de un subprograma. Introducido en ALGOL60.

- Lo permiten C y C++,
- No lo permiten Java ni C# por seguridad (induce a errores), pero sí dentro de una sentencia for.

Ejemplo de C:

```
void sub() {
    int count=3;
    while (...) {
        int count=1;
        count++;
        ...
        }
    ...
}
```

Ámbito Global

Se define una estructura del programa como una secuencia de definiciones de funciones, donde las variables pueden aparecer definidas fuera de ellas.

- **Ejemplo:** C, C++, PHP, JavaScript y Python.

Caso Especial: En C y C++ existe diferencia entre la **declaración** y **definición** de una variable de datos globales.

- Una **declaración** especifica el nombre de la variable y liga tipo, entre otros atributos, pero no asigna memoria. Sirve para referenciar un nombre global que es definido en otra parte. Una **definición** sí realiza la asignación de memoria.
- Útil para compilación separada y enlazado de objetos de programa (unidades compiladas).

Ámbito Dinámico

Un lenguaje posee ámbito dinámico cuando el ámbito de nombres no locales está definido por secuencia de llamadas a subprogramas, no por organización del código fuente.

- Anidamiento de llamadas define ascendencia dinámica.
- Referencias se resuelven en el orden de anidamiento de las llamadas.

Problemas: Lenguajes modernos en general no lo usan porque:

- No permite proteger bien la visibilidad de variables locales.
- No se pueden verificar estáticamente los tipos.
- Programas son difíciles de leer y son más lentos debido un manejo dinámico de ámbitos.

Ejemplo de Ámbitos en C

```c
int x = 1;
void p1(){
    printf("%d\n",x);
    x = 4;
}
void p2(int x){
    p1();
    printf("%d\n",x);
}

int main(){
    p2(5);
    printf("%d\n",x);
    return 0;
}
```

Referencias Multiples

Alias: cuando más de un nombre se refiere a un mismo objeto de memoria.

- Ejemplos: paso por referencia en C++.
- Puede hacer más difícil la lectura.

Sobrecarga (*overloading*): un mismo nombre se refiere a diferentes objetos.

- Ejemplos: operador + en C; funciones/métodos con diferentes parámetros en C++ y Java.
- Se debe resolver ambigüedad en el análisis semántico.

Modularización

Apoya el desarrollo de grandes programas y la división del trabajo en equipos de desarrollo, agrupando objetos de programa, tales como subprogramas, variables, tipos y otros.

- Permite reducir riesgos de conflicto entre nombres y compartimentar los errores, facilitando el desarrollo y la mantención.
- Módulos proveen mecanismos para importar y exportar objetos de programa usando sus nombres.

Ejemplos:

- C usa archivos y controla con `extern` y `static`.
- Espacios de nombre en C++ (e.g. `clase::metodo()`)
- Paquetes en Java y Ada.
- Módulos y paquetes en Python.

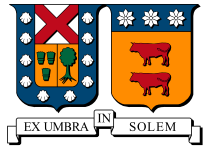
Tipos de Compilación

En grandes programas es deseable compilar por partes tanto en el desarrollo como la mantención del software.

Se definen dos tipos de compilación:

- **Compilación Separada**: unidades de programas pueden compilarse en diferentes tiempos, pero se consideran dependencias para verificar regularidad en el uso de interfaces (e.g. ADA y Java).
- **Compilación Independiente**: se compilan unidades de programa sin información de otras (e.g. C, C++ y Fortran).

Referencias



- Capítulos III y V de [Sebesta, 2011]
- Capítulos VI y VII de [Louden et. al., 2011]
- Capítulo II, III de [Tucker et. al., 2006]