



UNIVERSIDAD TÉCNICA
FEDERICO SANTA MARÍA

DEPARTAMENTO
DE INFORMÁTICA

LENGUAJES DE PROGRAMACIÓN

Equipo de Profesores:

Jorge Díaz Matte – José Luis Martí Lara – Rodrigo Salas Fuentes

Unidad 2

Fundamentos de los Lenguajes de Programación

2.1 Elementos Básicos de Lenguajes Formales

2.2 Descripción de Sintaxis

2.3 Nombres, Ligado y Ámbito

2.1 Elementos Básicos de Lenguajes Formales

Lenguajes Formales, Sintaxis, Semántica, Gramática libre de Contexto, BNF y eBNF, Expresiones regulares

Conceptos 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

- 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 el correcto funcionamiento de los traductores y programas (ej.: usando BNF).

Sintaxis: Tipos

- **Sintaxis léxica**: Define reglas para símbolos básicos o palabras denominados *tokens* (ej.: 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 (ej.: elimina puntuación). Útil para la traducción posterior y optimización en la generación de código.

Sintaxis: Especificación Formal

- 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

- 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.

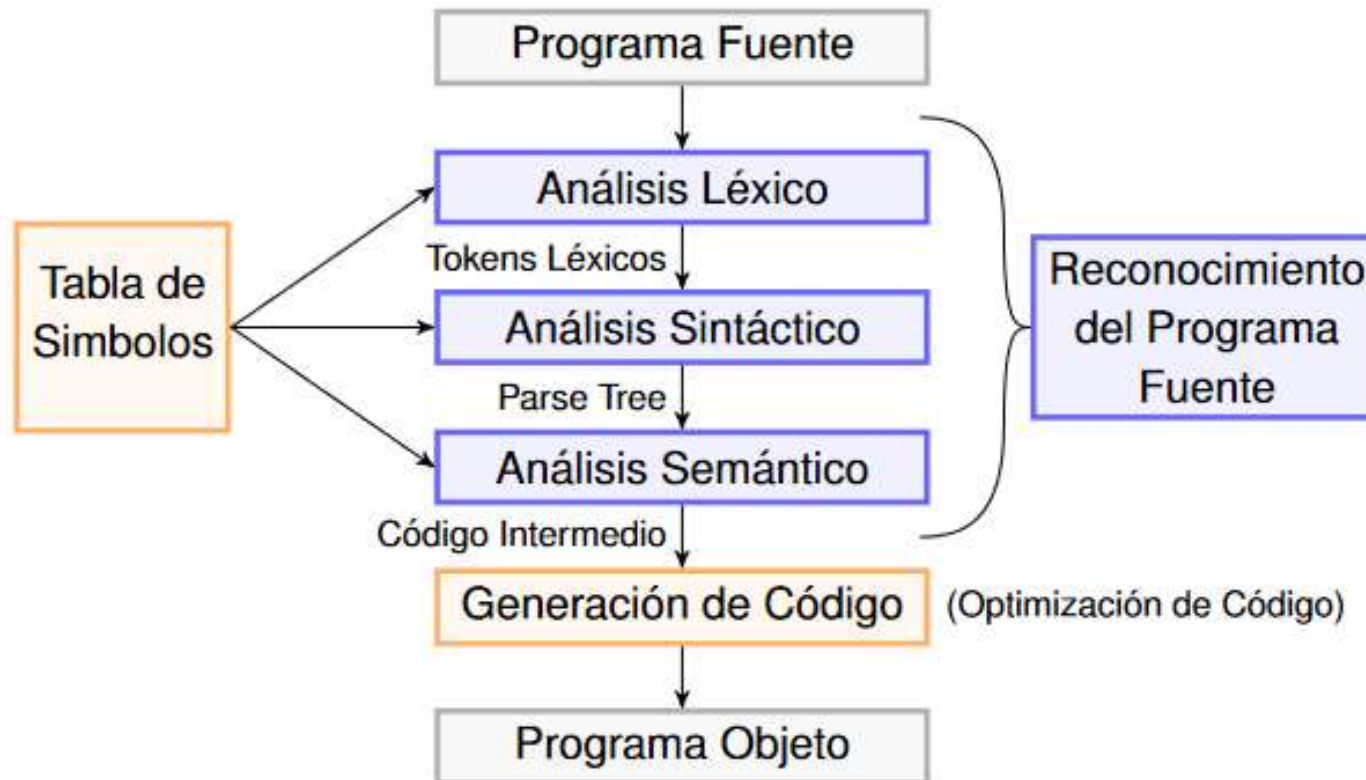
Semántica: Tipos

- **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.
Ejemplos: 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 Dinámica: Métodos de Especificación

- [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



Jerarquía de Chomsky

DEFINICIÓN: 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.

Gramática Libre de Contexto

Se define por una 4-tupla $G = (V, \Sigma, P, S)$:

- 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 Libre de Contexto: Ejemplo

$G = (V, \Sigma, P, S)$, con $V = \{S, X, Y, Z\}$, $\Sigma = \{a, b\}$, $S = S$, donde P tiene 4 reglas de producción:

1) $S \rightarrow X \mid Y$

2) $X \rightarrow ZaX \mid ZaZ$

3) $Y \rightarrow ZbY \mid ZbZ$

4) $Z \rightarrow aZbZ \mid bZaZ \mid \epsilon$

OBSERVACIÓN: Esta gramática genera todas las cadenas con un número desigual de símbolos “a” y “b”.

Backus-Naur Form (BNF)

- La notación BNF es un metalenguaje, creado por John Backus (1959), que permite especificar formalmente gramáticas libres de contexto.
- Las reglas de producción se especifican como:
 <símbolo> ::= <expresión con símbolos>

Backus-Naur Form (BNF)

Características:

- Lado izquierdo corresponde a un símbolo no terminal.
- Lado derecho (expresión) representa posible sustitución para símbolo 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

Reglas de Producción:

R1: $\langle \text{program} \rangle ::= \text{begin } \langle \text{stmt_list} \rangle \text{ end}$
R2: $\langle \text{stmt_list} \rangle ::= \langle \text{stmt} \rangle \mid \langle \text{stmt} \rangle ; \langle \text{stmt_list} \rangle$
R3: $\langle \text{stmt} \rangle ::= \langle \text{id} \rangle := \langle \text{exp} \rangle$
R4: $\langle \text{id} \rangle ::= A \mid B \mid C$
R5: $\langle \text{exp} \rangle ::= \langle \text{id} \rangle \mid \langle \text{exp} \rangle + \langle \text{exp} \rangle \mid \langle \text{id} \rangle * \langle \text{exp} \rangle \mid (\langle \text{exp} \rangle)$

Gramática:

$G = G = (V, \Sigma, P, S)$
 $V = \{ \text{program, stmt_list, stmt, id, exp} \}$
 $\Sigma = \{ \text{"begin", "end", ";", ":=", "+", "*", "(", ")"}, A, B, C \}$
 $P = \{ \text{R1, R2, R3, R4, R5} \}$
 $S = \text{"program"}$

eBNF (BNF extendida)

Extensiones:

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

eBNF (BNF extendida)

Notación:

- Tokens (terminales) se indican de diferentes maneras: en negrita, entre cremillas.
- Nombres de reglas (no terminales) se colocan en texto normal o entre paréntesis angulares.

eBNF: ejemplo de una Especificación

<sintaxis-bnf> ::= <regla> [<sintaxis-bnf>]

<regla> ::= "<" <regla-nombre> ">" "::=" <expresion>

<expresion> ::= <expresion-alt>

<expresion-alt> ::= <expresion-lista> ["|" <expresion>]

<expresion-lista> ::= ("| "\" <token> "\"
| "(" <expresion> ")"
| "[" <expresion> "]") [<expresion-lista>]

eBNF: ejemplo concreto

<if_stmt> ::= if <condition> then <stmts> [else <stmts>]

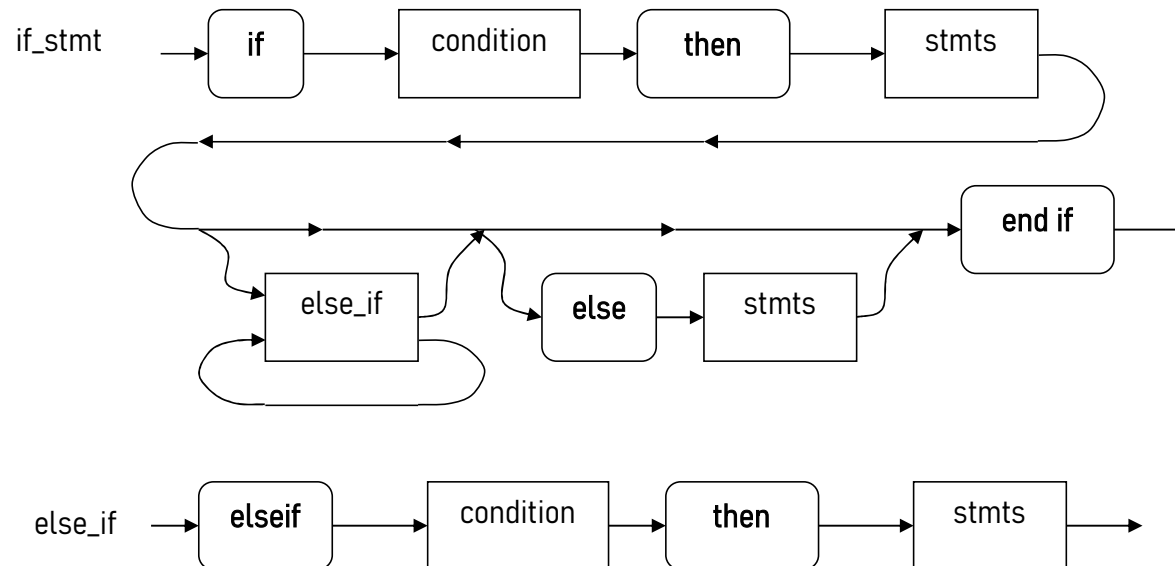
<identifier> ::= <letter> { <letter> | <digit> }

<for_stmt> ::= for <var> := <expr> (to | downto) <expr>
do <stmt>

Diagramas Sintácticos

**<if_stmt> ::= if <condition> then <stmts> {<else_if> }
[else <stmts>] end if**

<else_if> ::= elseif <condition> then <stmts>



Expresiones Regulares

- Permiten describir patrones de cadenas de caracteres. Son útiles para especificar y reconocer *tokens*. Cada expresión regular tiene asociado un autómata finito.

Expresiones Regulares

Operaciones Básicas:

- Concatenación: Expresada por la secuencia de los elementos.
- Cuantificación: Permite especificar frecuencias; * (0 o 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 (no permite recursión).

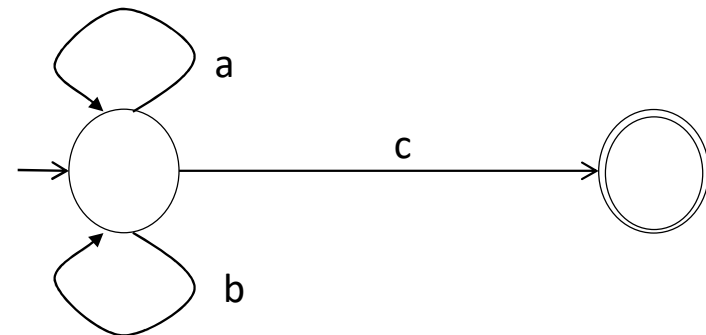
Ejemplo: la expresión $(a|b)^*c$ reconoce c , ac , bc , $aaaabbbbc$, etc.

Autómatas Finitos

Un Autómata Finito es una 5-tupla $(Q, \Sigma, \delta, q_0, F)$

- Q : Conjunto finito de estados (círculos)
- Σ : Un alfabeto finito (que define una cadena de entrada)
- δ : Función de transición $Q \times \Sigma \rightarrow Q$ (flecha con etiqueta)
- q_0 : Estado inicial (con una flecha de entrada)
- F : Estados finales o de aceptación $F \subseteq Q$ (con doble círculo)

Ejemplo: la expresión $(a|b)^*c$ se puede representar por el autómata finito:



2.2 Descripción de la Sintaxis

Elementos sintácticos, Análisis Léxico, Análisis Sintáctico, Resolución de Ambigüedad

Proceso de Análisis Léxico y Sintáctico

El proceso de reconocimiento de sintaxis tiene típicamente dos fases:

- Análisis Léxico: El traductor lee secuencia de caracteres de programa de entrada, y se reconocen *tokens*.
 - Se asume conocido el conjunto de caracteres válidos.
 - Expresiones regulares permiten reconocer *tokens*.
- Análisis Sintáctico: El traductor procesa los *tokens*, determinando si el programa está sintácticamente correcto.
 - Construcción de un **árbol sintáctico**, que permite analizar y reconocer si el programa es sintácticamente correcto.

Tokens

TIPOS DE TOKENS: Estos son:

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

Tokens

ASPECTOS PRÁCTICOS:

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

Tokens

Formato fijo:

- Cambio de línea puede actuar como separador de sentencias.
- Indentación (sangría) afecta significado sintáctico de programas en Python.

Expresiones regulares para reconocer un token:

- `[0-9]+(\.[0-9]+)?` representa un literal (constante) de punto flotante.

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 se denominada “semántica dirigida por sintaxis”.
- Un árbol sintáctico permite establecer una estructura sintáctica y asociarlo a un significado.
- Dos temas relevantes:
 - Definición de árbol sintáctico
 - Resolución de ambigüedad semántica mediante reglas de asociatividad y precedencia.

Árbol Sintáctico: ejemplo

Expresión: $A := A * (B + C)$

Convenciones:

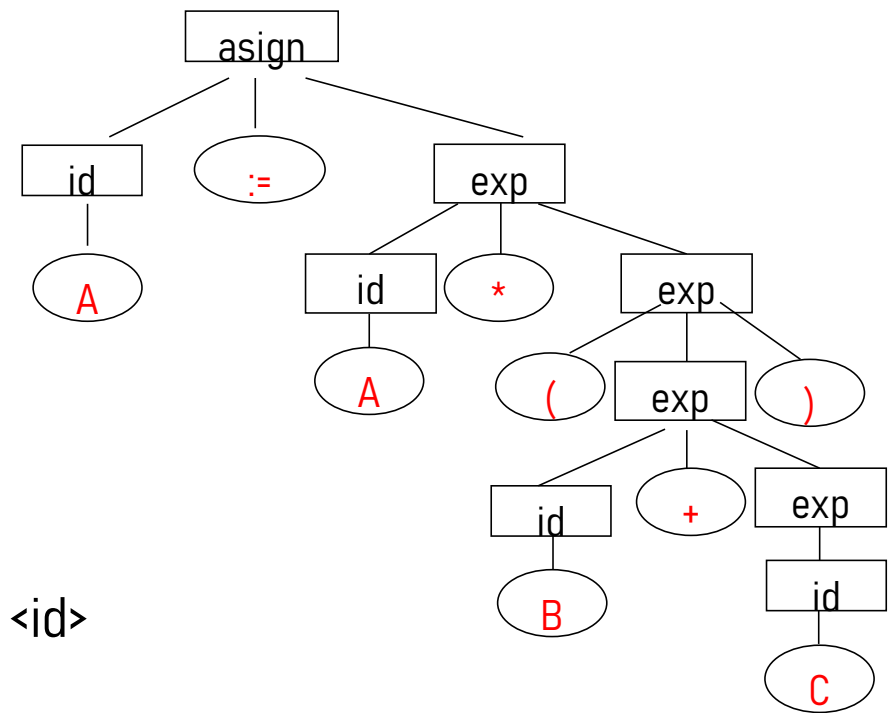
- \square : No terminal
- \odot : Terminal

Una gramática simple:

$\langle \text{assign} \rangle ::= \langle \text{id} \rangle := \langle \text{exp} \rangle$

$\langle \text{id} \rangle ::= A \mid B \mid C$

$\langle \text{exp} \rangle ::= \langle \text{id} \rangle + \langle \text{exp} \rangle \mid \langle \text{id} \rangle * \langle \text{exp} \rangle \mid (\langle \text{exp} \rangle) \mid \langle \text{id} \rangle$



Ambigüedad (1/2)

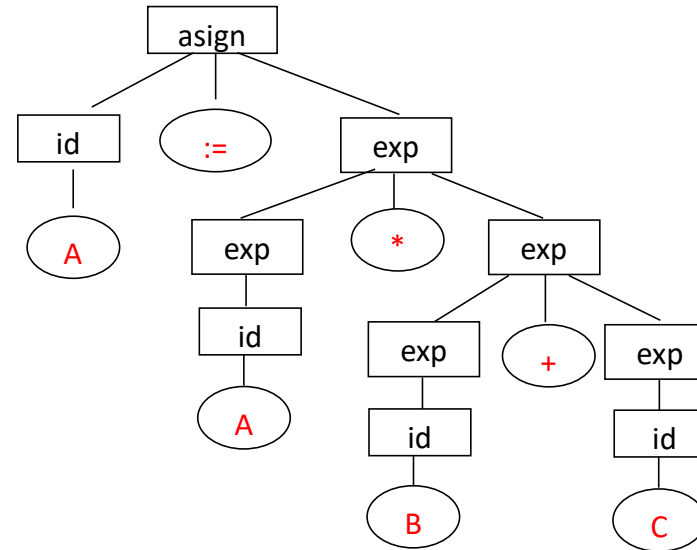
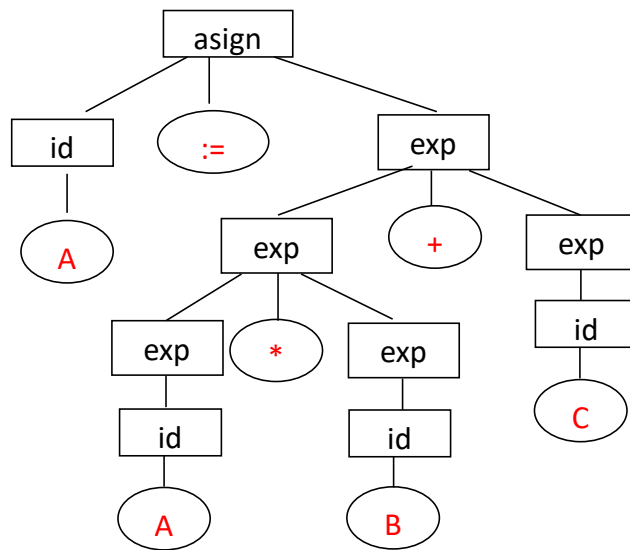
GRAMÁTICA AMBIGUA:

$\langle \text{assign} \rangle ::= \langle \text{id} \rangle := \langle \text{exp} \rangle$

$\langle \text{id} \rangle ::= A \mid B \mid C$

$\langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{exp} \rangle \mid \langle \text{exp} \rangle * \langle \text{exp} \rangle \mid (\langle \text{exp} \rangle) \mid \langle \text{id} \rangle$

Expresión: $A := A * (B + C)$



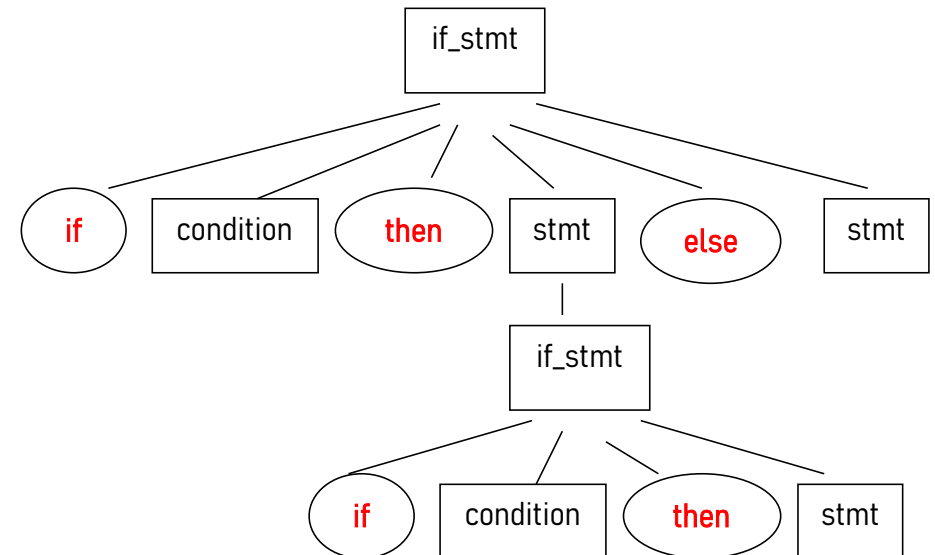
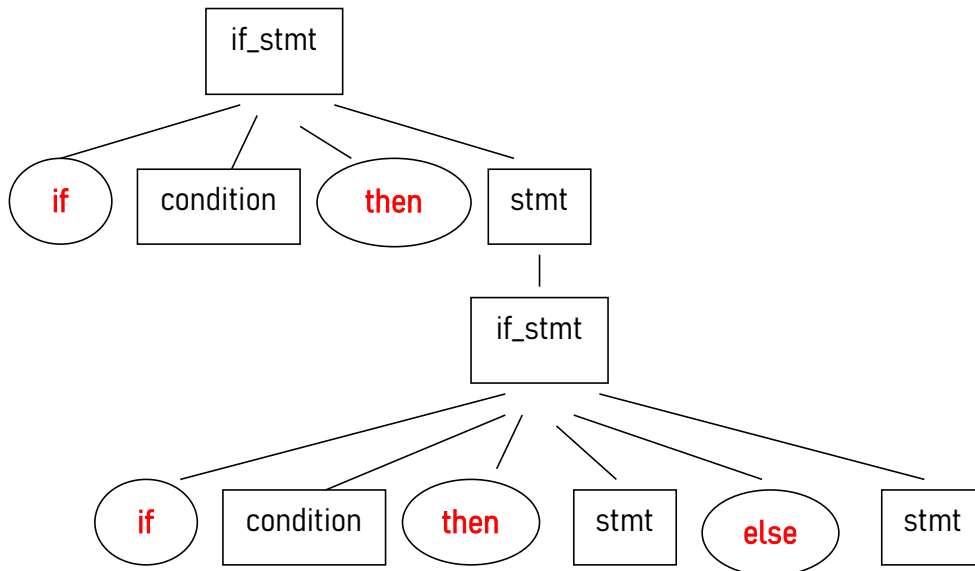
Ambigüedad (2/2)

Dada la siguiente gramática:

$\langle \text{if_stmt} \rangle ::= \text{if } \langle \text{condition} \rangle \text{ then } \langle \text{stmt} \rangle$
 $\quad \quad \quad | \text{if } \langle \text{condition} \rangle \text{ then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$

$\langle \text{stmt} \rangle ::= \langle \text{if_stmt} \rangle | \dots$

Reconocer: **if** (C1) **then** **if** (C2) **then** S1 **else** S2



Ambigüedad: Resolución

Caso de operadores:

- Definir precedencia
- A igual precedencia, definir asociatividad por la izquierda o por la derecha.

Caso del if:

- Asociar el *else* con el *if* más cercano

```
<stmt>      := <match>|< unmtched >  
<matched>  := if <logic_expr> then <matched> else < unmtched >  
              | <any_non-if_statement>  
<unmtched> ::= if <logic_expr> then <stmt>  
              | if <logic_expr> then <matched> else < unmtched >
```

Precedencia y Asociatividad de Operadores

- DEFINICIÓN: En una expresión pueden aparecer diferentes operadores, los cuales pueden ser evaluados de distintas maneras, produciendo ambigüedad. Gráficamente, el operador que se evalúa primero queda en un nivel más abajo en el árbol sintáctico.
- ASOCIATIVIDAD: Se identifican los operadores en la entrada del analizador sintáctico, y se aplica una regla de ordenar la evaluación.
 - Asociatividad por la izquierda: Se prioriza la evaluación desde la izquierda a la derecha.
 - Asociatividad por la derecha: Se prioriza evaluación de derecha a izquierda.

Precedencia y Asociatividad de Operadores

- 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.

Precedencia y Asociatividad de Operadores: C y C++

Operador	Descripción	Asociatividad
::	Resolución de ámbito (solo C++)	Izquierda a derecha
++ -- () [] . -> typeid() const_cast dynamic_cast reinterpret_cast static_cast	Post- incremento y decremento Llamada a función Elemento de vector Selección de elemento por referencia Selección de elemento con puntero Información de tipo en tiempo de ejecución (solo C++) Conversión de tipo (solo C++) Conversión de tipo (solo C++) Conversión de tipo (solo C++) Conversión de tipo (solo C++)	
++ -- + - ! ~ (type) * & sizeof new new[] delete delete[]	Pre- incremento y decremento Suma y resta unitaria NOT lógico y NOT binario Conversión de tipo Indirección Dirección de Tamaño de Asignación dinámica de memoria (solo C++) Desasignación dinámica de memoria (solo C++)	Derecha a izquierda

Ambigüedad: Conclusiones (1/2)

Características:

- Si gramática permite generar diferentes árboles sintácticos, se tiene ambigüedad.
- El problema mayor es que el compilador puede generar diferentes árboles sintácticos, y por lo tanto modificar la semántica del programa.
- Sucede cuando para una misma sentencia de la gramática (o regla), se permite más de una derivación.

Ambigüedad: Conclusiones (2/2)

Enfoques de resolución:

- Algunos analizadores sintácticos se pueden basar en una gramática ambigua, pero cuando detectan una ambigüedad, usan otra información para construir/seleccionar el árbol correcto.
- A veces es posible reescribir la gramática para eliminar una ambigüedad.

Árbol Sintáctico: ejercicio

<query>	::= SELECT <columnlist> FROM <table> [<condition>] ;
<columnlist>	::= * count(<columnlist>) <column>[,<columnlist>]
<condition>	::= WHERE <exp>
<exp>	::= <exp> AND <exp> <exp> OR <exp>
<exp>	::= <column> > <value> <column> < <value> <column> == <value>
<value>	::= A B C
<column>	::= colA colB //Atributos o columnas de la tabla
<table>	::= tabla

SELECT * FROM tabla WHERE colA > A AND colA < B AND colB == C;

2.3 Nombres, Ligado y Ámbito

Nombres, Variables, Constantes, Ligado de Tipo, Ligado de Memoria, Ámbito

Nombres

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

Nombres

Aspectos de diseño:

- Largo (significativo) del nombre
- Tipos de caracteres aceptados (ej.: 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 (ej.: `$...` para variables en PHP)

Variables

DEFINICIÓN: Abstracción de un objeto de memoria, que tiene varios atributos asociados, tal como la siguiente 6-tupla:

- Nombre: Identificador como lo ya discutido.
- Dirección (l-value): Indica dónde está localizado, pudiendo 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

DEFINICIÓN: Proceso de asociación de un atributo a una entidad del lenguaje (ej.: variable). Se consideran diferentes tipos de ligado, tal 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).

Ligado: de Tipo

DEFINICIÓN: 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

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 (ej.: PERL identificadores que comienzan con % son escalares).

DINÁMICO: Con declaración implícita en el momento de la asignación. Usado de preferencia por lenguajes de *scripting* (ej.: 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:

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

Tiempo de Ligado

DEFINICIÓN: 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 se realiza durante el diseño o implementación del lenguaje; o en la programación, compilación; o enlace (*linking*) o carga del programa.
- Dinámico: ligado se realiza durante la ejecución del programa (*runtime*), pudiendo cambiar en el intertanto.

Taxonomía de Memoria

DEFINICIÓN: 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.

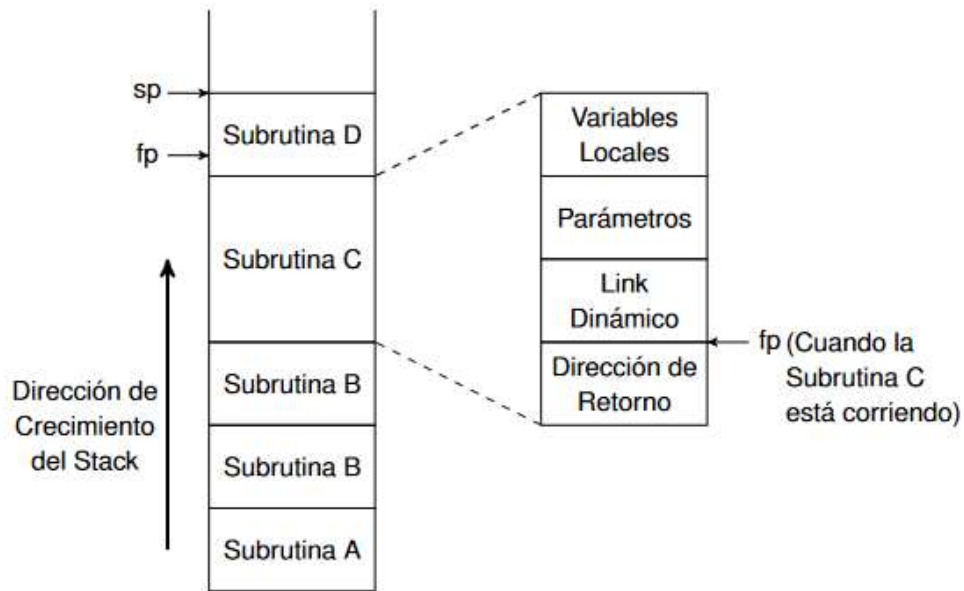
- Los “objetos de memoria” asignados, que pueden corresponder a datos, pero también a código, se crean y destruyen en diferentes tiempo (ciclo de vida).

Taxonomía de Memoria: Clasificación

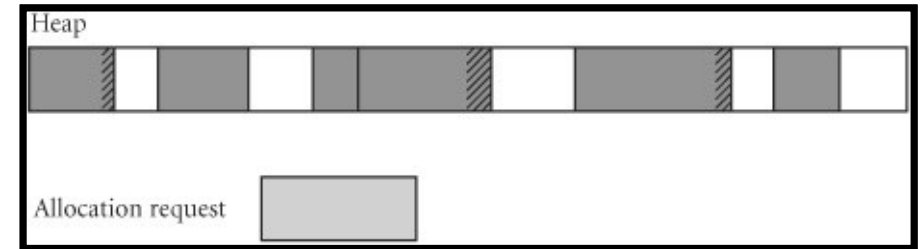
CLASIFICACIÓN:

- 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 (ejs.: invocación a procedimiento, ejecución de un bloque).
- Memoria heap: Área dinámica que permite crear y eliminar objetos de memoria que son referenciadas indirectamente mediante una variable como un puntero o una referencia.

Taxonomía de Memoria



Memoria Stack



Memoria Heap

Ligado de Memoria: Tipos de Variables

DEFINICIÓN: 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: ligado estático a la memoria estática. El ligado permanece durante todo el tiempo de ejecución del programa.
- Variable Dinámica de Stack (o automática): ligado ocurre cuando la ejecución alcanza el ámbito de su declaración y se libera memoria cuando se abandona este ámbito.
- Variable Dinámica de Heap: se asigna memoria dinámicamente desde el heap, donde la asignación y liberación puede ser explícita o implícita.

Ligado de Memoria: Variables Estáticas

DEFINICIÓN: Su acceso puede ser global, pero también se puede restringir a un ámbito local (ej.: calificador *static* en C).

- Ventajas:
 - Útil para definir variables globales y compartidas.
 - Para variables sensibles a la historia de un subprograma (ej.: 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).

Ligado de Memoria: Variables Dinámicas de *Stack*

DEFINICIÓN: 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.

Ligado de Memoria: Variables Dinámicas de *Heap*

DEFINICIÓN DEL CASO EXPLÍCITO: 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.

Ligado de Memoria: Variables Dinámicas de *Heap*

DEL CASO EXPLÍCITO:

Ventajas:

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

Desventaja:

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

Ligado de Memoria: Variables Dinámicas de *Heap*

DEFINICIÓN DE CASO IMPLÍCITO: 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

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

Ventajas:

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

Ejemplos:

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

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

Ámbito

DEFINICIÓN: 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.
- El 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

DEFINICIÓN: Ámbito puede ser determinado antes de la ejecución, es decir 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 (ejs.: Python, JavaScript, Java y Scheme).
- No anidados: no se permite la declaración de un subprograma dentro de otro (ej.: C).

Ámbito: Estático

Ejemplo en Pascal:

```
program main;
var x: integer;

  procedure p1;
  begin { p1}
  ... x ...
  end; {p1}

  procedure p2;
  var x: integer;
  begin { p2}
  ... x ...
  end; {p2}

begin {main}
...
end. {main}
```

The diagram illustrates static scope nesting. A large bracket on the right groups the entire program body (from the first procedure to the end) under the label **main**. Inside this, a bracket groups the first procedure under the label **p1**. Another bracket groups the second procedure under the label **p2**. This shows that **p1** and **p2** are nested within the scope of **main**, and **p2** is nested within the scope of **p1**.

Anidamiento y Ocultamiento

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

- 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 jerárquico.
- Ejemplo: operador de ámbito de C++:

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

Bloques

DEFINICIÓN: Permite la declaración de nombres dentro de un bloque de sentencias, creando ámbitos estáticos dentro de un subprograma.

- Se permiten en C y C++, no así en Java ni C# por seguridad (pues se induce a errores), pero sí dentro de una sentencia *for*.
- Ejemplo de C:

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

Ámbito: Global (1/2)

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

- Ejemplos: C, C++, PHP, JavaScript y Python.

Ámbito: Global (2/2)

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 objeto de programa (unidades compiladas).

Ámbito: Global

Ejemplo en C:

```
extern int i;  
int j = 100;  
static int k;  
  
void f1 (...)  
{ int i,m; ... }  
  
void f2 (...)  
{ int static k=0;  
  
    if (!k) { printf("primera vez\n"); k=1};  
    for (int j=0; j++; j<99) { "..."};  
}
```

Ámbito: Dinámico

DEFINICIÓN: 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.


Lenguajes modernos en general no lo usan porque:

- No permite proteger bien la visibilidad de variables locales.
- No se pueden verificar estáticamente los tipos.
- Los programas son difíciles de leer y son más lentos.

Ámbito: Dinámico

Ejemplo:

```
program main;  
var x: integer;  
  
  procedure p1;  
  begin { p1 }  
    ... x ...  
  end; { p1 }  
  
  procedure p2;  
  var x: integer;  
  begin { p2 }  
    ... x ...  
  end; { p2 }  
  
begin { main }  
...  
end. { main }
```



Suponga secuencia de llamadas:
main → p2 → p1

¿Referencia x en p1 se refiere a:
main::x ó p2::x?

Referencias Múltiples

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

- Ejemplos: paso por referencia en C++, uniones 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.

Modularidad (1/2)

DEFINICIÓN: 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.

Modularidad (1/2)

Ejemplos:

- C usa archivos y controla con `extern` y `static`.
- Espacios de nombre en C++ (ej.: `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 (tal como comprobación de interfaces, variables, etc.) de acuerdo a lo que exporta e importa (ej.: ADA y Java).
- Compilación Independiente: se compilan unidades de programas sin información de otras (ej.: C, C++ y Fortran).

Unidad 2: FIN

Fundamentos de los Lenguajes de Programación

FIN

2.1 Elementos Básicos de Lenguajes Formales

2.2 Descripción de Sintaxis

2.3 Nombres, Ligado y Ámbito