Tema 1.2. Un lenguaje mínimo y su procesador: *Gramáticas de atributos y tabla de símbolos*

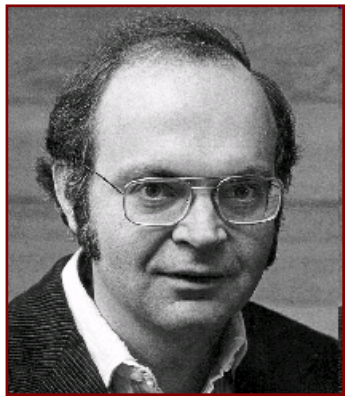
Profesor
Federico Peinado

Elaboración del material
José Luis Sierra
Federico Peinado

Ingeniería en Informática
Facultad de Informática – Universidad Complutense de Madrid
Curso 2009-2010

Semántica de un lenguaje de programación

- Hasta ahora hemos hablado de la forma (**sintaxis**) de las sentencias de un lenguaje de programación, pero ¿cómo podemos expresar su significado (**semántica**)?
- Donald E. Knuth propuso el formalismo de las **gramáticas de atributos** para expresar la semántica de cualquier lenguaje incontextual, basándose en estas ideas:



  - **Semántica *de traducción***: Cada sentencia se asocia con otra sentencia en un "lenguaje objeto de los significados"
    - Ej. `4 * 5` $\rightarrow_{\text{significado}}$ `apila 4, apila 5, multiplica`
  - **Principio de composicionalidad**: La "sentencia significado" asociada a cada categoría sintáctica dependerá de las "sentencias significado" de sus subcategorías sintácticas
    - Ej. El valor de una expresión (33 + 4*5) se compone con el valor de los operandos (33 y 4*5) y de la operación (+)

Gramáticas de atributos

- Se parte de la *gramática incontextual* y se le añade:
  - Un conjunto de **atributos semánticos** que se asociarán a las categorías sintácticas
    categoriaX.$\boldsymbol{a_1}$, *categoriaX.*$\boldsymbol{a_2}$, **…**, *categoriaX.*$\boldsymbol{a_n}$
    categoriaY.$\boldsymbol{a_1}$, *categoriaY.*$\boldsymbol{a_2}$, **…**, *categoriaY.*$\boldsymbol{a_n}$
    …
  - Un conjunto de **ecuaciones semánticas** que servirán para calcular el valor de los atributos de *la categoría sintáctica del lado izquierdo* de las producciones usando los valores de los atributos de *las categorías sintácticas del derecho*
    $\boldsymbol{a_1 = f_1 (a_x, a_{x+1}, a_{x+2} \ldots, a_y)}$
    $\boldsymbol{a_2 = f_2 (a_x, a_{x+1}, a_{x+2} \ldots, a_y)}$
    …
    - Las funciones $\mathbf{f_1}$, $\mathbf{f_2}$,… se llaman **funciones semánticas**

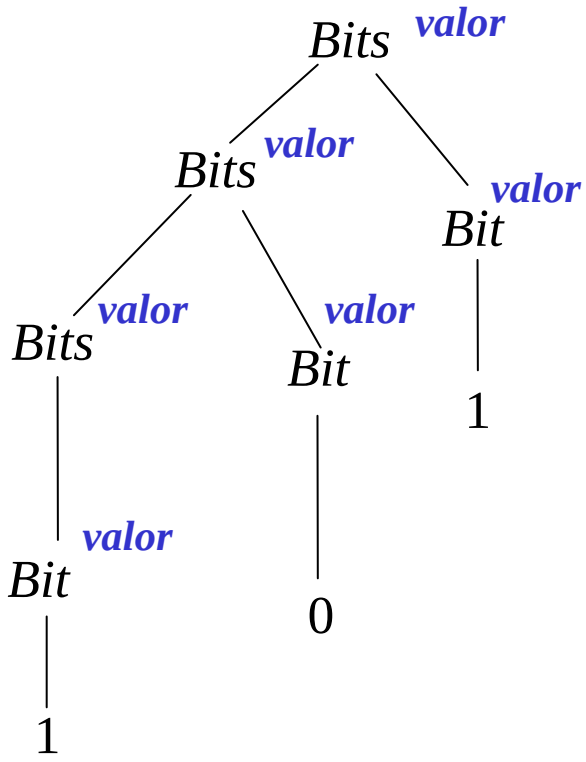
Ejemplo de gramática de atributos

```
Bits ::= Bits Bit
   Bits0.valor = Bits1.valor * 2 + Bit.valor
Bits ::= Bit
   Bits.valor = Bit.valor
Bit ::=  0
   Bit.valor = 0
Bit ::= 1
   Bit.valor = 1
```

** Para distinguir varias apariciones de una misma categoría sintáctica en una producción de una gramática de atributos conviene ponerles subíndices (0, 1, …)*

Árboles sintácticos atribuidos

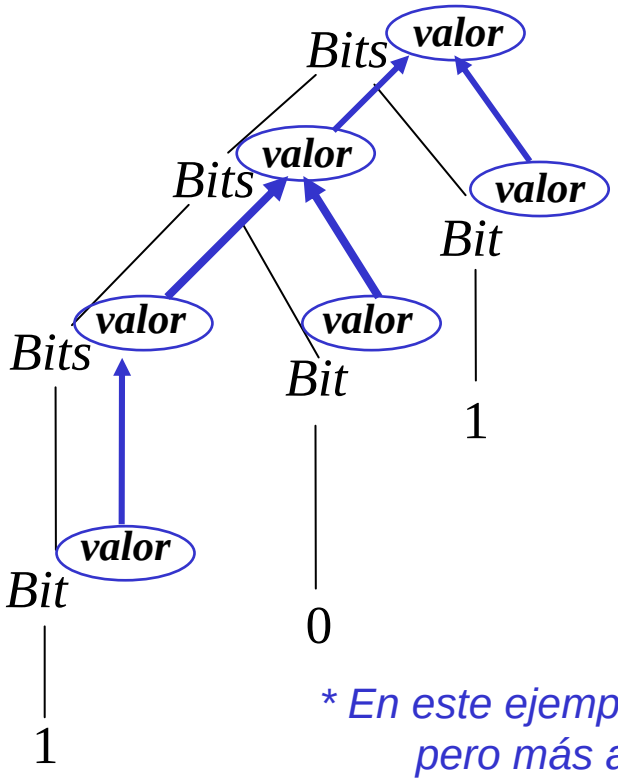
- Los nodos-padre de los árboles sintácticos, *decorados* por los atributos, se vuelven **árboles sintácticos atribuidos**
- Ej.



Grafos de dependencias de atributos

- Las ecuaciones semánticas introducen dependencias que se muestran como **grafos de dependencias de atributos**
  - Los nodos son los atributos y los arcos van de los *atributos de la derecha* de una ecuación al *atributo de la izquierda*

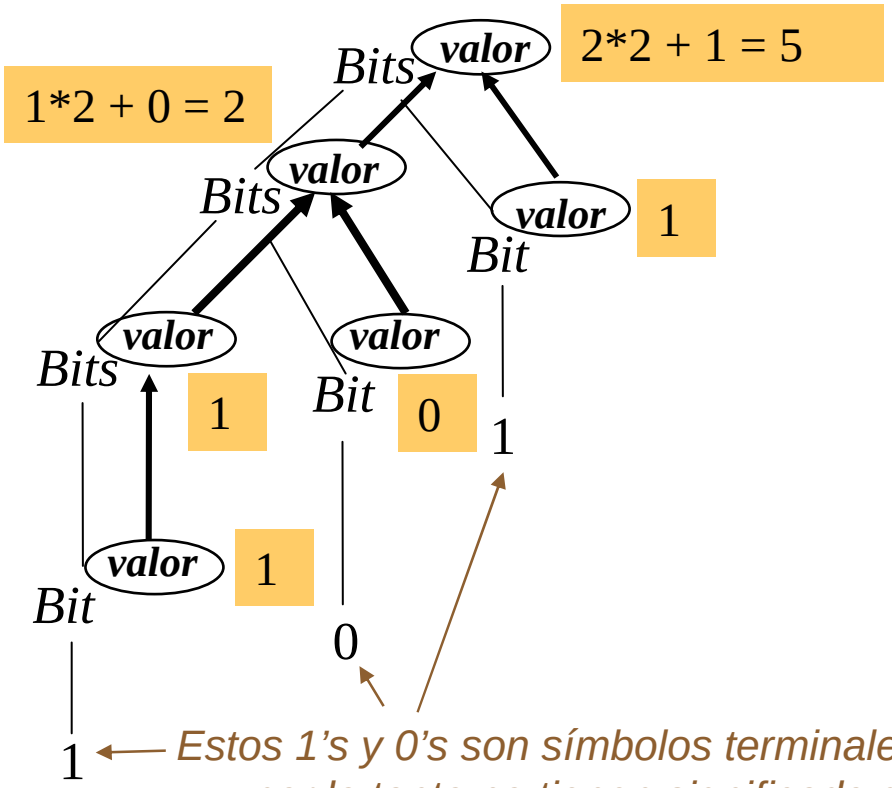
- Ej.



** En este ejemplo el grafo es simplemente un árbol, pero más adelante los hay más complejos*

Cálculo del valor de los atributos

- El valor de un atributo se calcula evaluando la función semántica correspondiente sobre los otros atributos, según *el orden del grafo de dependencias de atributos*
- Ej.

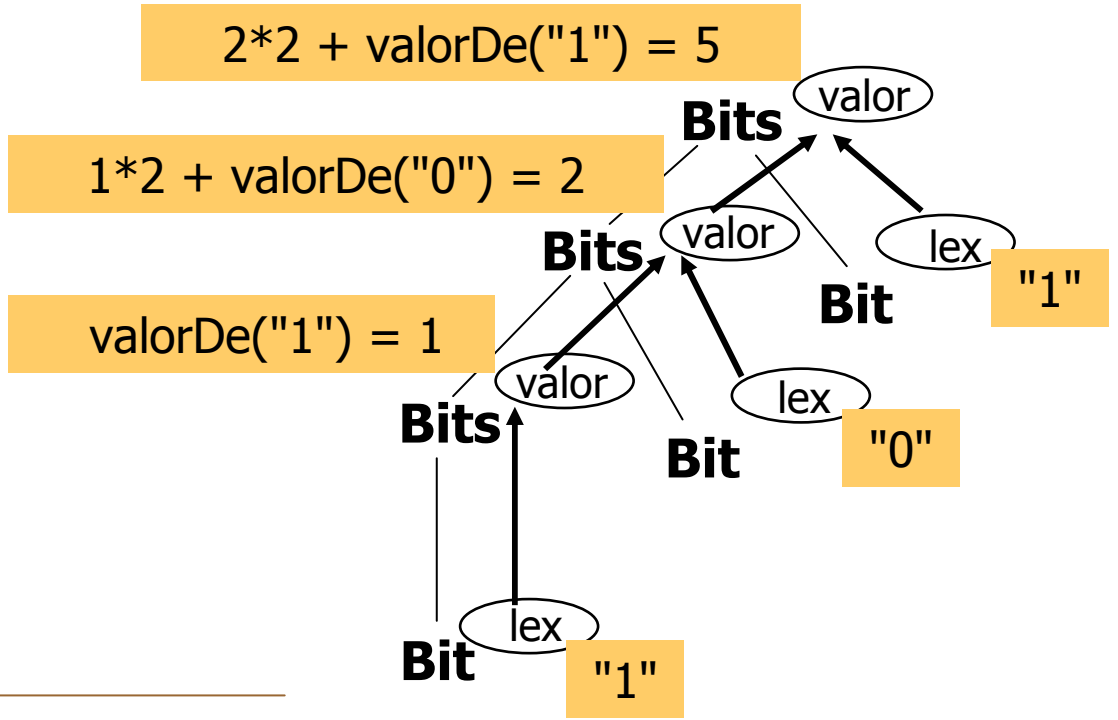


Atributos léxicos

- Hay varios tipos de atributos interesantes para definir la gramática de atributos de un lenguaje de programación
- Los **atributos léxicos** se asocian a todos los terminales de la gramática (= categorías léxicas) y contienen el valor concreto de los *lexemas* que han sido reconocidos
  *categoria-léxicaX.**lex***
  *categoria-léxicaY.**lex***
  …
  - Como la gramática no captura la microsintaxis (nos quedamos en las categorías léxicas) hace falta alguna manera de acceder a la *cadena concreta* que hay por debajo
    - Ej. El nombre de un *Identificador,* los dígitos de un *Número*, etc.
  - Asumiremos que vienen prefijados "de manera externa" a la gramática de atributos
    - Ej. Lo hace el analizador léxico

Ejemplo con atributos léxicos

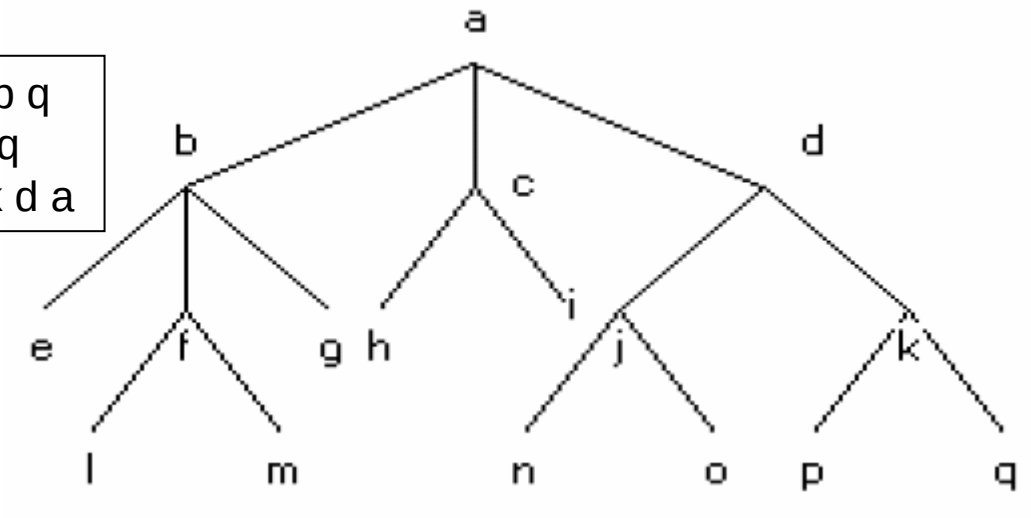
```
Bits ::= Bits Bit
   Bits0.valor = Bits1.valor * 2 + valorDe(Bit.lex)
Bits ::= Bit
   Bits.valor = valorDe(Bit.lex)
Bit ::= 0
   Bit.lex = “0”
Bit ::= 1
   Bit.lex = “1”
```



Atributos sintetizados y gramáticas s-atribuidas

- Todos los atributos que se calculan según el *principio de composicionalidad* se llaman **atributos sintetizados**
  - Ej. Los atributos anteriores **lex** y **valor**
- Las gramáticas de atributos que únicamente poseen *atributos sintetizados* se llaman **gramáticas s-atribuidas**
- La evaluación en una gramática s-atribuida se puede hacer con un **recorrido en postorden** de los árboles sintácticos atribuidos
- **Ejemplo (repaso):**

Preorden: a b e f l m g c h i d j n o k p q
Inorden: e b l f m g a h c i n j o d p k q
Postorden: e l m f g b h i c n o j p q k d a



Gramáticas de atributos (contextuales)

- Muchas veces el cálculo del valor de los atributos necesita ***información adicional de contexto*** → Se necesita *generalizar* el formalismo de las gramáticas de atributos
  - *Ej. Para interpretar el identificador de una variable (sustituirlo por su valor) hay que conocer su dirección en memoria, que se estableció en la sección de declaración de variables*
- De esta forma una gramática de atributos ya puede ser **gramática contextual** (que define un lenguaje contextual)

Atributos heredados

- La gramática de atributos se extiende con *información del contexto* (= parte no inferior del árbol sintáctico atribuido)
  - **Principio de composicionalidad generalizado**: La "sentencia significado" asociada a cada categoría sintáctica dependerá de las "sentencias significado" de sus subcategorías sintácticas y de las categorías sintácticas que formen parte de su contexto
- Todos los atributos que se calculan según el *principio de composicionalidad generalizado* (usando al menos un atributo del contexto) se llaman **atributos heredados**

* Los atributos sintetizados son como **valores de salida** de las categorías sintácticas (suben hacia la raíz –el programa-), mientras que los atributos heredados son como **valores de entrada** (bajan hacia las hojas –las expresiones, etc.-)

Ejemplo con atributos heredados

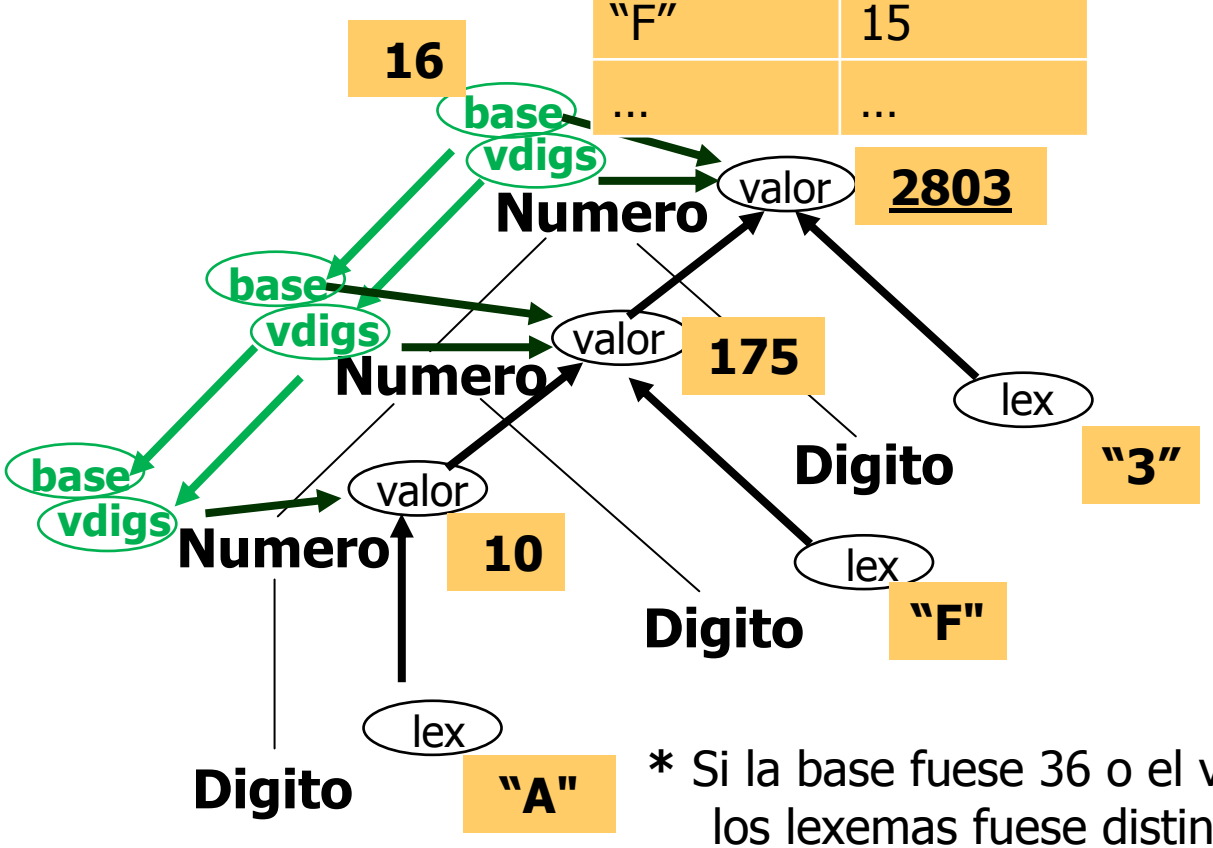
- *Números en base X*: Para interpretar un número hay que saber interpretar sus dígitos y además conocer la base
  - Ej. AF3 = $2803)_{10}$ porque A=10, F=15, 3=3 y la base es 16
  - Ej. AF3 = $13503)_{10}$ porque A=10, F=15, 3=3 y la base es 36

```
Número ::= Número Dígito
   Número0.valor = Número1.valor*Número0.base +
                   valorDe(Dígito.lex,Número0.vdigs)
   Número1.base = Número0.base
   Número1.vdigs = Número0.vdigs
Número ::= Dígito
   Número.valor = valorDe(Dígito.lex,Número.vdigs)
```

* **vdigs** es un atributo en forma de tabla con el valor de los dígitos

Ejemplo con atributos heredados

Lexema	Valor
"3"	3
"A"	10
"F"	15
...	...

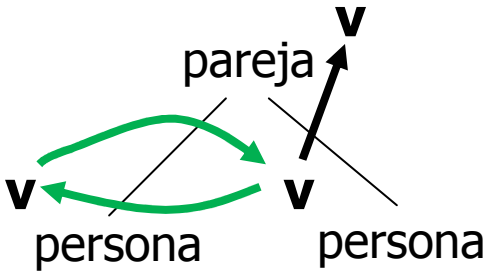


* Si la base fuese 36 o el valor asociado a los lexemas fuese distinto, todos los valores del árbol serían otros...

Ciclos en los grafos de dependencias de atributos

- Normalmente los grafos de dependencias de atributos no contienen ciclos (**acíclicos)**
  →El orden de evaluación es sencillamente *el orden topológico del grafo*
- Sin embargo una gramática de atributos puede generar grafos de dependencias de atributos con ciclos (**cíclicos**)
  - Ej.

```
pareja ::= persona persona
    persona0.v = f(persona1.v)
    persona1.v = f(persona0.v)
    pareja.v = persona1.v
```



El problema de la circularidad

- Una gramática de atributos es **circular** cuando da lugar a grafos de dependencias de atributos cíclicos
- **Evitaremos la circularidad** a la hora de hacer gramáticas de atributos para un lenguaje de programación
  - Aunque, al igual que la ambigüedad, la circularidad es un problema que puede ser tratable en algunos casos… en esta primera parte del curso no lo admitiremos
- Existen algoritmos para comprobar la circularidad de una gramática de atributos, que también se usan en la **generación automática de procesadores del lenguaje eficientes** partiendo de su gramática de atributos

Gramáticas de atributos l-atribuidas

- Una gramática de atributos es **l-atribuida** (se hereda por la izquierda) cuando en cada producción **N ::= X_1 X_2 …X_n** los atributos heredados de cada **X_i** dependen sólo de:
  - Los atributos heredados de **N**
  - Los atributos sintetizados de **X_1, X_1, …, X_{i-1}**
- La evaluación en una gramática l-atribuida se puede hacer con un **recorrido en preorden** de los árboles sintácticos atribuidos para los atributos *heredados*, y un **recorrido en postorden** para los *sintetizados*
- Intentaremos trabajar siempre con gramáticas l-atribuidas, aunque también existen técnicas para tratar algunos casos donde se hereda *por la derecha*

Tabla de símbolos (TS)

Tabla Símbolos

Tabla de Tokens

Id Símbolo	ExprTipo	Direccion/Valor	Clase	Nivel	Tipo
C1	<t:Int, tam:1>	1	PVar	0	INTEGER
C2	<t:Int, tam:1>	-1	PVar	0	INTEGER
C3	<t:Int, tam:1>	1	PVar	0	INTEGER
T1	<t:Int, tam:1>	--	Tipo	0	INTEGER
TPUNTO	<t:Reg, camp...	--	Tipo	0	RECORD
T2	<t:Arr, nelms...	--	Tipo	0	ARRAY
T3	<t:Arr, nelms...	--	Tipo	0	ARRAY
PUNTO1	<t:Ref, id:tPu...	0	Var	0	REF
PUNTO2	<t:Ref, id:tPu...	7	Var	0	REF
ARRAY1	<t:Ref, id:t2, t...	14	Var	0	REF
METODO1	<t:Proc, para...	--	Proc	1	PROC
----------	----------	----------	----------	----------	----------
X	<t:Int, tam:1>	0	PVar	1	INTEGER
Y	<t:Int, tam:1>	1	PVar	1	INTEGER
Z	<t:Int, tam:1>	2	PVar	1	INTEGER
X2	<t:Char, tam:...	3	Var	1	CHAR
METODO1	<t:Proc, para...	--	Proc	1	PROC
C1	<t:Int, tam:1>	2	PVar	1	INTEGER
MITIPO	<t:Arr, nelms...	--	Tipo	1	ARRAY
V1	<t:Char, tam:...	4	Var	1	CHAR
SUBPROC	<t:Proc, para...	--	Proc	2	PROC
----------	----------	----------	----------	----------	----------
VARIABLE	<t:Int, tam:1>	0	PVar	2	INTEGER
SUBPROC	<t:Proc, para...	--	Proc	2	PROC

- Los lenguajes de programación incluyen mecanismos para definir *nuevos símbolos* junto con *el significado* que les da el usuario
  - Ej. Declaración de que X e Y son variables y son de tipo entero:
    var
    x,y: Integer;
- La **tabla de símbolos** (= el diccionario específico de un programa) es la estructura de datos que permite almacenar esa información semántica
- Sirve para dar soporte a las *restricciones contextuales* (ej. impedir que se usen variables sin declarar) y es de utilidad tanto en el *análisis* como en la *síntesis* de un programa

Estructura y construcción de la tabla de símbolos

- Al definir un lenguaje de programación, también hay que diseñar la tabla de símbolos que permite almacenar la *información expresable mediante declaraciones*
  - Especificando **estructura en abstracto**
  - Especificando **operaciones de acceso y modificación**
  - Especificando **algoritmo de construcción**
- Precisamente el **algoritmo de construcción** (que mientras analiza la sección de declaraciones va rellenando la tabla) utiliza una gramática de atributos

* Aquí estudiamos conceptos básicos sobre la estructura y la construcción de la tabla de símbolos, que serán ampliados al hablar sobre la *síntesis* en procesadores de lenguaje

Ampliación del ejemplo: Secuencia de instrucciones de asignación

1. Descripción informal del lenguaje
   - Cada asignación tiene la forma *`variable := expresión`*
   - Las expresiones pueden contener variables
   - Restricción contextual: las variables deben haber sido previamente declaradas antes de usarse en una asignación
   - … más todo lo dicho antes para las expresiones aritméticas

   - *Ejemplo de frase (= miniprograma) en este lenguaje:*

```
tiempo; espacio; velocidad
&
espacio := 25;
tiempo := 30;
velocidad := espacio / tiempo
```

Ampliación del ejemplo: Secuencia de instrucciones de asignación

2. Definición léxica
   - parte-entera ≡ 0|[1-9]{dígito}*
   - parte-decimal ≡ .{dígito}+
   - número ≡ {parte-entera}{parte-decimal}?
   - letra ≡ [a-z]|[A-Z]
   - dígito ≡ [0-9]
   - identificador ≡ {letra}({letra}|{dígito})*
   - + ≡ \+
   - - ≡ \-
   - * ≡ *
   - / ≡ /
   - (≡ \(
   -) ≡ \)
   - ~~asignación ≡ :=~~
   - ~~separador ≡ &~~
   - ~~punto-y-coma ≡ ;~~
   - := ≡ :=
   - & ≡ &
   - ; ≡ ;

Ampliación del ejemplo: Secuencia de instrucciones de asignación

3. Definición sintáctica incontextual

```
Prog  ::= Decs & Is
Decs  ::= Dec | Decs ; Dec
Dec   ::= iden
Is    ::= I | Is ; I
I     ::= IAsig
IAsig ::= iden := Exp
Exp   ::= Exp OpAd Term
Exp   ::= Term
Term  ::= Term OpMul Fact
Term  ::= Fact
Fact  ::= num | iden | ( Exp )
OpAd  ::= + | -
OpMul ::= * | /
```

Ampliación del ejemplo: Secuencia de instrucciones de asignación

4. y 5. Definición sintáctica contextual y definición semántica

- **Estructura** de la tabla de símbolos
  - Conjunto de identificadores que implícitamente se asume que son todos de tipo real
  - Ej.

Identificador	Dirección (celda de memoria)
“tiempo”	16058
“espacio”	16060
…	…

Ampliación del ejemplo: Secuencia de instrucciones de asignación

4. y 5. Definición sintáctica contextual y definición semántica

- **Operaciones** de la tabla de símbolos
  - Las tres operaciones más básicas posibles:

creaTS Crea la tabla inicialmente

añadeID Añade un nuevo identificador a la tabla

existeID Nos dice si un identificador está en la tabla

Ampliación del ejemplo: Secuencia de instrucciones de asignación

4. y 5. Definición sintáctica contextual y definición semántica

- **Operaciones** de la tabla de símbolos

Géneros: TS,ID **(alias de** String**),** BOOL

Operaciones privadas:

[]: TS

.: TS × ID → TS

Operaciones públicas:

creaTS: TS

añadeID: TS × ID → TS

existeID: TS × ID → BOOL

Ecuaciones:

creaTS = []

añadeID(TS,id) = .(TS,id)

existeID(.(TS,id_o),id_1) = (id_o == id_1) ∨ existeID(TS,id_1)

existeID([],_) = false

- En vez de usar una *signatura formal* (como la especificación algebraica) nos conformaremos con una *descripción informal* de la semántica de las operaciones

creaTS():TS

El resultado es una TS vacía

añadeID(ts:TS, id:String):TS

El resultado es la TS resultante de añadir *id* a *ts*

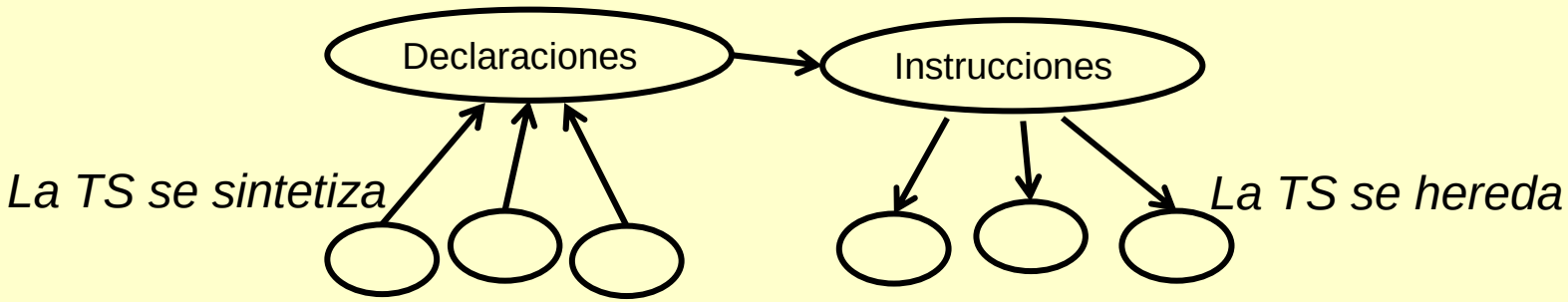
existeID(ts: TS, id:String):Boolean

El resultado es *true* si *id* aparece en *ts*, *false* en caso contrario

Ampliación del ejemplo: Secuencia de instrucciones de asignación

4. y 5. Definición sintáctica contextual y definición semántica

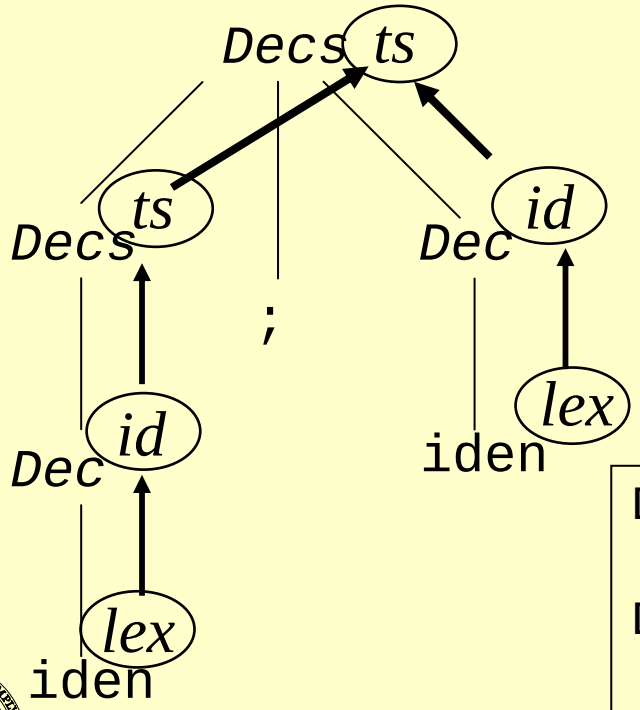
- **Construcción** de la tabla de símbolos
  - La tabla de símbolos puede verse *como un atributo sintetizado* de una *gramática de atributos* que vamos a definir para expresar **el sublenguaje de la sección de declaraciones** de un lenguaje de programación
    - Luego en el **sublenguaje de la sección de código** curiosamente se ve al revés, como *atributo heredado*



Ampliación del ejemplo: Secuencia de instrucciones de asignación

4. y 5. Definición sintáctica contextual y definición semántica

- **Construcción** de la tabla de símbolos
  - Ej. Sección de declaración de variables



* **id** es otro atributo sintetizado,
con información del identificador declarado

```
Decs ::= Dec
  Decs.ts = añadeID(creaTS(),Dec.id)
Decs ::= Decs ; Dec
  Decs0.ts = añadeID(Decs1.ts,Dec.id)
Dec ::= iden
  Dec.id = iden.lex
```

Sintaxis abstracta

- La **sintaxis abstracta** de un lenguaje es el *tipo abstracto de datos* que representa las características sintácticas esenciales para el procesamiento de sus sentencias

- La tabla de símbolos es, por lo tanto, la sintaxis abstracta del sublenguaje de la sección de declaraciones de un lenguaje de programación

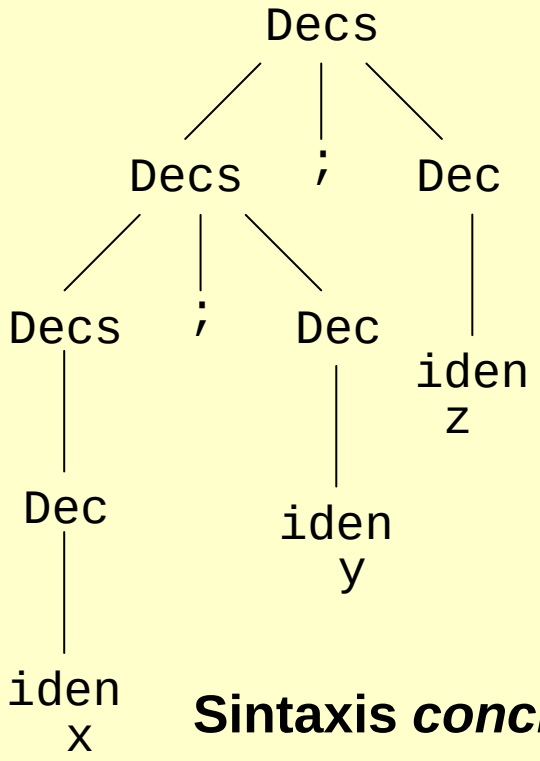
Ampliación del ejemplo: Secuencia de instrucciones de asignación

4. y 5. Definición sintáctica contextual y definición semántica

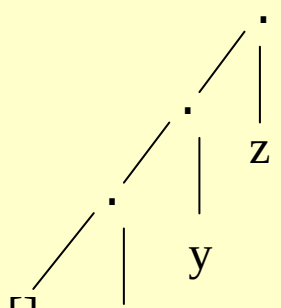
- **Construcción** de la tabla de símbolos
  - Ej.

x;y;z

Sentencia



Sintaxis *concreta*



Sintaxis *abstracta*
(se prescinde de toda la estructura de no terminales, etc.)

Críticas, dudas, sugerencias...

Federico Peinado
www.federicopeinado.es