UNIVERSITAT JAUME•I

# 4º Ingeniería Informática

## II26 Procesadores de lenguaje

Análisis semántico

## Esquema del tema

1. Introducción
2. Esquemas de traducción dirigidos por la sintaxis
3. El árbol de sintaxis abstracta
4. Comprobaciones semánticas
5. Interpretación
6. Introducción a metacomp
7. Algunas aplicaciones
8. Resumen del tema

## 1. Introducción

Hay determinadas características de los lenguajes de programación que no pueden ser modeladas mediante gramáticas incontextuales y que es necesario comprobar en una fase posterior al análisis sintáctico. Por otro lado, las fases posteriores de la compilación o interpretación necesitan una representación de la entrada que les permita llevar a cabo sus funciones de manera adecuada.

Estas dos vertientes —detección de errores y representación de la información— están muy relacionadas y se solapan en la práctica. Supongamos, por ejemplo, que nuestro lenguaje permite asignaciones según la regla

$$\langle\text{Asignación}\rangle \rightarrow \textbf{id}\,\texttt{:=}\langle\text{Expresión}\rangle\,\texttt{;}$$

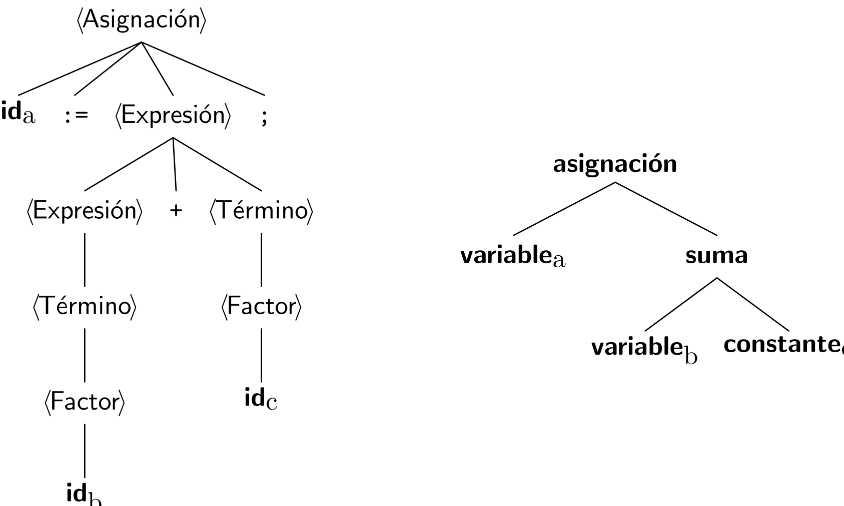
Es habitual que se impongan ciertas restricciones. En nuestro caso, estas podrían ser:

- El identificador de la parte izquierda debe estar declarado previamente.
- El tipo de la expresión debe ser compatible con el del identificador.

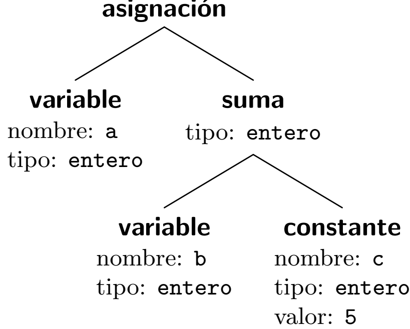
El analizador semántico deberá comprobar que estas dos restricciones se cumplen antes de declarar que la sentencia de asignación está bien formada. Pero sucede que la información necesaria para comprobarlas es útil también para generar código. Esto quiere decir que si tuviéramos una separación estricta entre las fases del compilador, para generar código deberíamos volver a mirar el identificador para saber a qué objeto (variable, función, constante, etc.) corresponde y qué tipo tiene y también deberíamos volver a comprobar los tipos de la expresión para generar el código adecuado.

Por otro lado, aunque en teoría el árbol de análisis sería suficiente para fases posteriores de la compilación o interpretación, es una representación que contiene mucha información redundante. Por ejemplo, el árbol correspondiente a la expresión `a:= b+c;` bien podría tener el aspecto del árbol de la izquierda, cuando el de la derecha contiene esencialmente la misma información y

resulta más cómodo para trabajar:




El segundo árbol se conoce como *árbol de sintaxis abstracta* (o AST, de las iniciales en inglés). Como hemos comentado, durante el análisis semántico se recoge una serie de informaciones que resultan de utilidad para fases posteriores. Estas informaciones se pueden almacenar en el árbol, "decorándolo":




Así, el objetivo de la fase de análisis semántico será doble: por un lado detectaremos errores que no se han detectado en fases previas y por otro lado obtendremos el AST decorado de la entrada.

Para ello utilizaremos *esquemas de traducción dirigidos por la sintaxis*, que permitirán asociar acciones a las reglas de la gramática. Estas acciones realizarán comprobaciones y construirán el AST que después se recorrerá para terminar las comprobaciones y será la base para la interpretación o la generación de código.

## 2. Esquemas de traducción dirigidos por la sintaxis

Nuestro objetivo es especificar una serie de acciones que se realizarán durante el análisis de la entrada y tener un mecanismo que nos permita obtener la información necesaria para realizar estas acciones. Para esto añadimos a las gramáticas dos elementos nuevos:

- Acciones intercaladas en las reglas.
- Atributos asociados a los no terminales de la gramática.

### 2.1. Acciones intercaladas en las reglas

Supongamos que tenemos el no terminal ⟨A⟩ que deriva dos partes diferenciadas y queremos que se escriba el mensaje "Voy a la parte 2" tras analizarse la primera. Podemos describir esto de

forma sencilla si aumentamos nuestra gramática incluyendo la acción en la parte derecha de la regla de ⟨A⟩:

$$\langle\mathsf{A}\rangle \quad\rightarrow\quad \langle\mathsf{Parte1}\rangle\ \{\texttt{escribe}(\text{“Voy a la parte 2”});\ \}\ \langle\mathsf{Parte2}\rangle$$

Podemos entender esta regla extendida como la receta: “analiza la primera parte, una vez encontrada, escribe el mensaje y después analiza la segunda parte”.

## 2.2. Atributos asociados a los no terminales

Si nuestras acciones se limitaran a escribir mensajes que indicaran la fase del análisis en la que nos encontramos, no necesitaríamos mucho más. Sin embargo, lo normal es que queramos que las acciones respondan al contenido de los programas que escribimos. Para ello, vamos a añadir a los no terminales una serie de *atributos*. Estos no son más que una generalización de los atributos que tienen los terminales. Vamos a ver un ejemplo.

Supongamos que en un lenguaje de programación se exige que en las subrutinas aparezca un identificador en la cabecera que debe coincidir con el que aparece al final. Tenemos el siguiente fragmento de la gramática del lenguaje:

$$\begin{aligned}\langle\mathsf{Subrutina}\rangle &\rightarrow \langle\mathsf{Cabecera}\rangle\langle\mathsf{Cuerpo}\rangle\langle\mathsf{Fin}\rangle\\ \langle\mathsf{Cabecera}\rangle &\rightarrow \textbf{subrutina id}\ \text{“(”}\langle\mathsf{Parámetros}\rangle\text{“)”};\\ \langle\mathsf{Fin}\rangle &\rightarrow \textbf{fin id}\end{aligned}$$

En la regla correspondiente a ⟨Fin⟩ querremos comprobar que el identificador es el mismo que en la cabecera. Vamos a definir un atributo del no terminal ⟨Fin⟩ que será el que nos diga qué nombre esperamos encontrar al final de la función:

$$\langle\mathsf{Fin}\rangle \quad\rightarrow\quad \textbf{fin id}\ \{\textbf{si id}.\text{lexema} \neq \langle\mathsf{Fin}\rangle.\text{esperado}\ \textbf{entonces}\ \textit{error}\ \textbf{fin si}\}$$

El atributo esperado es lo que se conoce como un *atributo heredado*: su valor se calculará en las reglas en las que ⟨Fin⟩ aparezca en la parte derecha y se podrá utilizar en las reglas en las que ⟨Fin⟩ aparezca en la izquierda; será información que “heredará” de su entorno. Fíjate en cómo hemos empleado un atributo de **id** para hacer comprobaciones. El analizador semántico es el que emplea la información contenida en los atributos de los componentes léxicos. Recuerda que el sintáctico sólo ve las etiquetas de las categorías. En cuanto a *error*, suponemos que representa las acciones necesarias para tratar el error.

Ahora tenemos que completar las acciones de modo que ⟨Fin⟩ tenga algún valor para esperado. Podemos añadir una acción a la regla de ⟨Subrutina⟩ para calcular el valor de ⟨Fin⟩.esperado:

$$\langle\mathsf{Subrutina}\rangle \quad\rightarrow\quad \langle\mathsf{Cabecera}\rangle\langle\mathsf{Cuerpo}\rangle\ \{\langle\mathsf{Fin}\rangle.\text{esperado:=}\ \langle\mathsf{Cabecera}\rangle.\text{inicial}\}\langle\mathsf{Fin}\rangle$$

Lo que hacemos es copiar el atributo inicial que nos “dará” ⟨Cabecera⟩. El atributo inicial de ⟨Cabecera⟩ es un *atributo sintetizado*: su valor se calculará en las reglas en las que ⟨Cabecera⟩ aparezca en la parte izquierda y se utilizará en las reglas donde ⟨Cabecera⟩ aparezca en la parte derecha; es información que ⟨Cabecera⟩ “sintetiza” para su entorno. Debemos calcular el valor de inicial en la regla de ⟨Cabecera⟩:

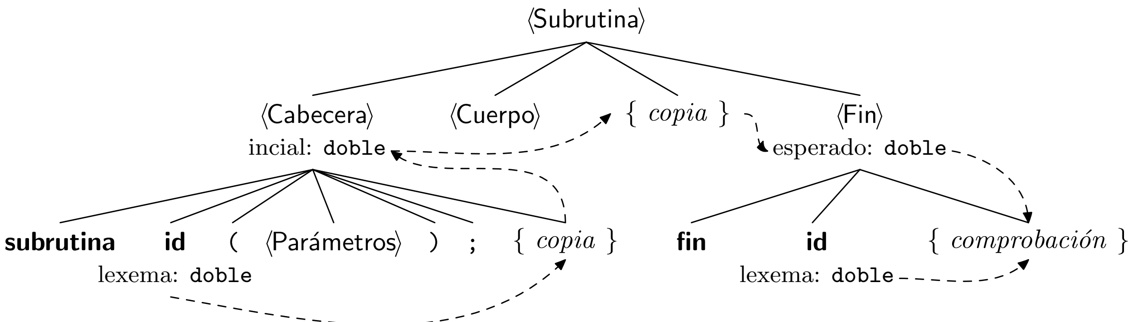
$$\langle\mathsf{Cabecera}\rangle \quad\rightarrow\quad \textbf{subrutina id}\ \text{“(”}\langle\mathsf{Parámetros}\rangle\text{“)”};\ \{\langle\mathsf{Cabecera}\rangle.\text{inicial:=}\ \textbf{id}.\text{lexema}\}$$

Vamos a intentar ver el flujo de información sobre el árbol de análisis de la sentencia:

`subrutina doble (` ⟨Parámetros⟩ `);` ⟨Cuerpo⟩ `fin doble;`

© Universitat Jaume I 2010-2011

El árbol que obtenemos es[1]:




Como puedes ver, cuando el atributo es sintetizado, la información "sube", mientras que cuando es heredado, "baja".

## 2.3. Otros elementos de las acciones

Como habrás comprobado, no hemos definido ningún lenguaje formal para escribir las acciones. Asumiremos que se emplean construcciones corrientes de los algoritmos tales como condicionales o bucles. También haremos referencia en ellos a subrutinas y a variables, que interpretaremos como variables locales a la regla que estamos analizando o como variables globales si lo indicamos explícitamente. Así, en:

⟨Valor⟩ → **opsum** {**si opsum**.lexema="+" **entonces** signo:= 1 **si no** signo:= -1 **fin si**}
⟨Valor⟩$_1$ {⟨Valor⟩.v:= signo*⟨Valor⟩$_1$.v}

no hay ninguna "interferencia" entre las distintas variables signo que pueden llegar a estar activas en un momento dado, por ejemplo, al analizar `--a`. Fíjate también en cómo hemos distinguido mediante un subíndice las dos apariciones del no terminal ⟨Valor⟩.

Un recurso muy útil son los atributos que contienen listas. Por ejemplo, para una declaración de variables del tipo:

⟨DeclVariables⟩ → ⟨ListaIds⟩ **:** ⟨Tipo⟩
⟨ListaIds⟩ → **id ,** ⟨ListaIds⟩ | **id**

podemos hacer lo siguiente:

⟨DeclVariables⟩ → ⟨ListaIds⟩ **:** ⟨Tipo⟩
{**para** $v \in$ ⟨ListaIds⟩.l **hacer** declara_var($v$, ⟨Tipo⟩.t) **fin para**}
⟨ListaIds⟩ → ⟨ListaIds⟩$_1$ **, id** {⟨ListaIds⟩.l:= ⟨ListaIds⟩$_1$.l; ⟨ListaIds⟩.l.añade(**id**.lexema)}
⟨ListaIds⟩ → **id** {⟨ListaIds⟩.l:= [**id**.lexema]}

## 2.4. Recopilación

Con lo que hemos visto, podemos decir que un esquema de traducción consta de:

- *Una gramática incontextual* que le sirve de soporte.
- Un conjunto de *atributos* asociados a los símbolos terminales y no terminales.
- Un conjunto de *acciones* asociadas a las partes derechas de las reglas.

[1]Por problemas de espacio, hemos abreviado el texto de las acciones.

Dividimos los atributos en dos grupos:

- Atributos heredados.
- Atributos sintetizados.

Sea ⟨A⟩ → ⟨X$_1$⟩ . . . ⟨X$_n$⟩ una producción de nuestra gramática. Exigiremos que:

- Las acciones de la regla calculen todos los atributos sintetizados de ⟨A⟩.
- Las acciones situadas a la izquierda de ⟨X$_i$⟩ calculen todos los atributos heredados de ⟨X$_i$⟩.
- Ninguna acción haga referencia a los atributos sintetizados de los no terminales situados a su derecha en la producción.

Las dos últimas reglas nos permitirán integrar las acciones semánticas en los analizadores descendentes recursivos. Cuando se emplean, se dice que el esquema de traducción está basado en una *gramática L-atribuida*. La L indica que el análisis y evaluación de los atributos se pueden hacer de izquierda a derecha.

Ejercicio 1

Las siguientes reglas representan parte de las sentencias estructuradas de un lenguaje de programación:

⟨Programa⟩ → ⟨ListaSentencias⟩
⟨ListaSentencias⟩ → ⟨Sentencia⟩⟨ListaSentencias⟩|⟨Sentencia⟩
⟨Sentencia⟩ → **mientras** ⟨Expresión⟩ **hacer** ⟨ListaSentencias⟩ **finmientras**
⟨Sentencia⟩ → **si** ⟨Expresión⟩ **entonces** ⟨ListaSentencias⟩ **sino** ⟨ListaSentencias⟩ **finsi**
⟨Sentencia⟩ → **interrumpir**
⟨Sentencia⟩ → **otros**

Añade las reglas necesarias para comprobar que la instrucción **interrumpir** aparece únicamente dentro de un bucle.

Ejercicio 2

Sea $G$ la siguiente gramática:

⟨A⟩ → ⟨B⟩⟨C⟩|a
⟨B⟩ → ⟨A⟩a⟨B⟩b|⟨C⟩a
⟨C⟩ → a⟨C⟩⟨C⟩|aba⟨C⟩|a

Añade a $G$ las reglas semánticas necesarias para que el atributo ia de ⟨A⟩ contenga el número de aes al inicio de la cadena generada. Por ejemplo, dadas las cadenas `aaaaba`, `abaaaa` y `aaa`, los valores de ia serían 4, 1 y 3, respectivamente.

Puedes utilizar los atributos adicionales que consideres necesarios, pero ninguna variable global. Además, los atributos que añadas deben ser de tipo entero o lógico.

## 2.5. Eliminación de recursividad por la izquierda y atributos

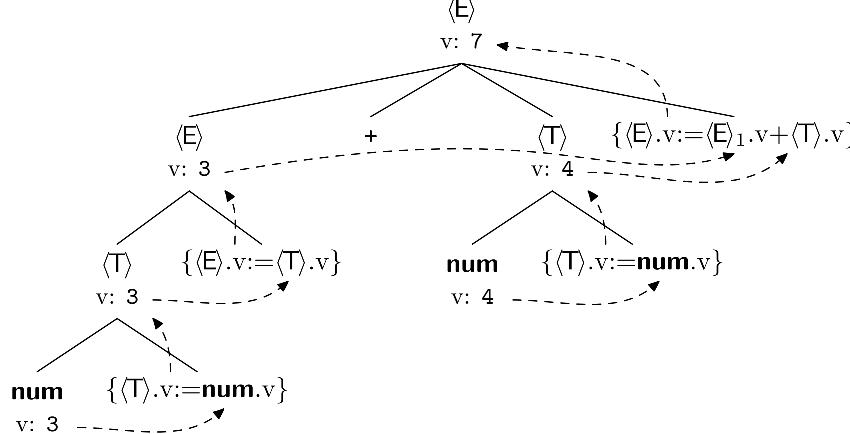
En el tema de análisis sintáctico vimos cómo se pueden modificar las producciones con recursividad por la izquierda para lograr que la gramática sea LL(1). El problema con las transformaciones es que dejan una gramática que tiene muy poca relación con la original, lo que hace que escribir los atributos y las reglas correspondientes sea difícil sobre la gramática transformada. Sin embargo, se pueden escribir los atributos sobre la gramática original y después convertirlos de una manera bastante mecánica.

© Universitat Jaume I 2010-2011

Como las GPDRs no suelen necesitar recursividad por la izquierda, es fácil que no tengas que utilizar esta transformación. Pese a todo, vamos a ver cómo se hace la transformación sobre un ejemplo. Partimos del siguiente esquema de traducción:

$$\begin{array}{lcl}
\langle \mathsf{E}\rangle & \rightarrow & \langle \mathsf{E}\rangle_1\mathsf{+}\langle \mathsf{T}\rangle\ \{\langle \mathsf{E}\rangle.\text{v:= }\langle \mathsf{E}\rangle_1.\text{v}+\langle \mathsf{T}\rangle.\text{v}\}\\
\langle \mathsf{E}\rangle & \rightarrow & \langle \mathsf{T}\rangle\ \{\langle \mathsf{E}\rangle.\text{v:= }\langle \mathsf{T}\rangle.\text{v}\}\\
\langle \mathsf{T}\rangle & \rightarrow & \textbf{num}\ \{\langle \mathsf{T}\rangle.\text{v:= }\textbf{num}.\text{v}\}
\end{array}$$

Primero, veamos cómo se propagan los valores con un ejemplo. Si analizamos la expresión `3+4` obtenemos el siguiente árbol decorado:




Para eliminar la recursividad por la izquierda, comenzamos por transformar la gramática:

$$\begin{array}{lcl}
\langle \mathsf{E}\rangle & \rightarrow & \langle \mathsf{T}\rangle\langle \mathsf{E'}\rangle\\
\langle \mathsf{E'}\rangle & \rightarrow & \mathsf{+}\langle \mathsf{T}\rangle\langle \mathsf{E'}\rangle\\
\langle \mathsf{E'}\rangle & \rightarrow & \lambda\\
\langle \mathsf{T}\rangle & \rightarrow & \textbf{num}
\end{array}$$

Al plantearnos las reglas en la nueva gramática, vemos que en la regla de la suma tenemos el sumando derecho en $\langle \mathsf{E'}\rangle$ pero no el izquierdo. Para arreglarlo, creamos un atributo heredado que guarda el valor del sumando izquierdo. Ya que es un atributo heredado, tenemos que calcular su valor en todas las reglas donde aparece $\langle \mathsf{E'}\rangle$ en la parte derecha: en la primera regla nos limitamos a copiar el valor de $\langle \mathsf{T}\rangle$; en la segunda, sumamos el valor heredado recibido con el de $\langle \mathsf{T}\rangle$:

$$\begin{array}{lcl}
\langle \mathsf{E}\rangle & \rightarrow & \langle \mathsf{T}\rangle\ \{\langle \mathsf{E'}\rangle.\text{h:= }\langle \mathsf{T}\rangle.\text{v}\}\langle \mathsf{E'}\rangle\\
\langle \mathsf{E'}\rangle & \rightarrow & \mathsf{+}\langle \mathsf{T}\rangle\ \{\langle \mathsf{E'}\rangle_1.\text{h:= }\langle \mathsf{E'}\rangle.\text{h}+\langle \mathsf{T}\rangle.\text{v}\}\langle \mathsf{E'}\rangle_1
\end{array}$$

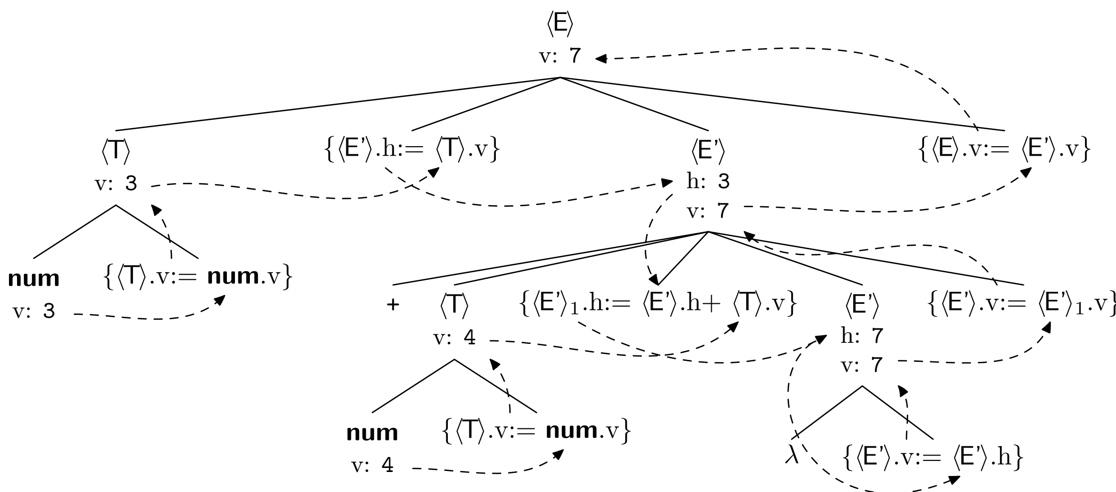
Con esto, el atributo h contendrá siempre la suma, que es el valor que tenemos que devolver finalmente. Por eso, hay que "transmitir" el nuevo valor al atributo v:

$$\begin{array}{lcl}
\langle \mathsf{E}\rangle & \rightarrow & \langle \mathsf{T}\rangle\ \{\langle \mathsf{E'}\rangle.\text{h:= }\langle \mathsf{T}\rangle.\text{v }\}\langle \mathsf{E'}\rangle\ \{\langle \mathsf{E}\rangle.\text{v:= }\langle \mathsf{E'}\rangle.\text{v}\}\\
\langle \mathsf{E'}\rangle & \rightarrow & \mathsf{+}\langle \mathsf{T}\rangle\ \{\langle \mathsf{E'}\rangle_1.\text{h:= }\langle \mathsf{E'}\rangle.\text{h}+\langle \mathsf{T}\rangle.\text{v}\}\langle \mathsf{E'}\rangle_1\ \{\langle \mathsf{E'}\rangle.\text{v:= }\langle \mathsf{E'}\rangle_1.\text{v}\}
\end{array}$$

¿Qué hacemos cuando $\langle \mathsf{E'}\rangle$ se reescribe como la cadena vacía? En este caso, basta con devolver como sintetizado el valor que se hereda. El esquema de traducción completo queda:

$$\begin{array}{lcl}
\langle \mathsf{E}\rangle & \rightarrow & \langle \mathsf{T}\rangle\ \{\langle \mathsf{E'}\rangle.\text{h:= }\langle \mathsf{T}\rangle.\text{v}\}\langle \mathsf{E'}\rangle\ \{\langle \mathsf{E}\rangle.\text{v:= }\langle \mathsf{E'}\rangle.\text{v}\}\\
\langle \mathsf{E'}\rangle & \rightarrow & \mathsf{+}\langle \mathsf{T}\rangle\ \{\langle \mathsf{E'}\rangle_1.\text{h:= }\langle \mathsf{E'}\rangle.\text{h}+\langle \mathsf{T}\rangle.\text{v}\}\langle \mathsf{E'}\rangle_1\ \{\langle \mathsf{E'}\rangle.\text{v:= }\langle \mathsf{E'}\rangle_1.\text{v}\}\\
\langle \mathsf{E'}\rangle & \rightarrow & \lambda\ \{\langle \mathsf{E'}\rangle.\text{v:= }\langle \mathsf{E'}\rangle.\text{h}\}\\
\langle \mathsf{T}\rangle & \rightarrow & \textbf{num}\ \{\langle \mathsf{T}\rangle.\text{v:= }\textbf{num}.\text{v}\}
\end{array}$$

Podemos ver cómo funciona este esquema retomando el ejemplo de `3+4`:




Ejercicio 3

Transforma el siguiente esquema de traducción para eliminar la recursividad por la izquierda:

$$
\begin{array}{lll}
\langle E\rangle & \rightarrow & \langle E\rangle_1\texttt{+}\langle T\rangle\ \{\langle E\rangle.\text{arb:= NodoSuma}(\langle E\rangle_1.\text{arb}, \langle T\rangle.\text{arb});\} \\
\langle E\rangle & \rightarrow & \langle T\rangle\ \{\langle E\rangle.\text{arb:= } \langle T\rangle.\text{arb}\} \\
\langle T\rangle & \rightarrow & \langle T\rangle_1\texttt{*}\langle F\rangle\ \{\langle T\rangle.\text{arb:= NodoProducto}(\langle T\rangle_1.\text{arb}, \langle F\rangle.\text{arb});\} \\
\langle T\rangle & \rightarrow & \langle F\rangle\ \{\langle T\rangle.\text{arb:= } \langle F\rangle.\text{arb}\} \\
\langle F\rangle & \rightarrow & \text{“(”}\langle E\rangle\text{“)”}\ \{\langle F\rangle.\text{arb:= } \langle E\rangle.\text{arb}\} \\
\langle F\rangle & \rightarrow & \textbf{num}\ \{\langle F\rangle.\text{v:= NodoEntero}(\textbf{num}.\text{v});\}
\end{array}
$$

El siguiente ejercicio presenta la transformación de una manera más general.

Ejercicio* 4

Si tenemos las siguientes reglas en un esquema de traducción

$$
\begin{array}{lll}
\langle A\rangle & \rightarrow & \langle A\rangle_1\langle Y\rangle\ \{\langle A\rangle.\text{a:= g}(\langle A\rangle_1.\text{a}, \langle Y\rangle.\text{y})\} \\
\langle A\rangle & \rightarrow & \langle X\rangle\ \{\langle A\rangle.\text{a:= f}(\langle X\rangle.\text{x})\}
\end{array}
$$

podemos transformarlas en

$$
\begin{array}{lll}
\langle A\rangle & \rightarrow & \langle X\rangle\ \{\langle A'\rangle.\text{h:= f}(\langle X\rangle.\text{x})\}\ \langle A'\rangle\ \{\langle A\rangle.\text{a:= }\langle A'\rangle.\text{s}\} \\
\langle A'\rangle & \rightarrow & \langle Y\rangle\ \{\langle A'\rangle_1.\text{h:= g}(\langle A'\rangle.\text{h}, \langle Y\rangle.\text{y})\}\ \langle A'\rangle_1\ \{\langle A'\rangle.\text{s:= }\langle A'\rangle_1.\text{s}\} \\
\langle A'\rangle & \rightarrow & \lambda\ \{\langle A'\rangle.\text{s:= }\langle A'\rangle.\text{h}\}
\end{array}
$$

Comprueba que la transformación es correcta analizando $\langle X\rangle\langle Y\rangle_1\langle Y\rangle_2$ mediante las dos versiones y comparando el valor de a.

## 2.6. Implementación de los esquemas de traducción

La interpretación de las acciones como sentencias que se ejecutan al pasar el análisis por ellas permite implementar los esquemas de traducción de manera sencilla. Para ello se modifica la implementación del analizador recursivo descendente correspondiente a la gramática original de la siguiente manera:

© Universitat Jaume I 2010-2011

- Los atributos heredados del no terminal $\langle\mathsf{A}\rangle$ se interpretan como parámetros de entrada de la función `Analiza_A`.
- Los atributos sintetizados del no terminal $\langle\mathsf{A}\rangle$ se interpretan como parámetros de salida de la función `Analiza_A`.
- Las acciones semánticas, una vez traducidas al lenguaje de programación correspondiente, se insertan en la posición correspondiente según su orden en la parte derecha donde aparecen.

En la práctica, es frecuente que, si el lenguaje de programación (como C) no permite devolver más de un valor, los atributos sintetizados del no terminal se pasen por referencia.

En Python existe una solución bastante cómoda. Comenzamos por definir una clase vacía:

```
class Atributos:
  pass
```

Antes de llamar a una función o método de análisis, creamos un objeto de esta clase y le añadimos los atributos heredados del no terminal. La correspondiente función de análisis creará los atributos sintetizados. Este es el único parámetro que se pasa. Así, la traducción de la regla:

$$\langle\mathsf{E}\rangle \quad\rightarrow\quad \langle\mathsf{T}\rangle\ \{\langle\mathsf{Ep}\rangle.\mathrm{h}{:=}\ \langle\mathsf{T}\rangle.\mathrm{v}\}\ \langle\mathsf{Ep}\rangle\ \{\langle\mathsf{E}\rangle.\mathrm{v}{:=}\ \langle\mathsf{Ep}\rangle.\mathrm{v}\}$$

es la siguiente:

```
def analiza_E(E):
  T= Atributos()  # Atributos de T
  Ep= Atributos() # Atributos de Ep
  analiza_T(T)
  Ep.h= T.v       # Creamos un atributo heredado de Ep
                  # a partir de uno sintetizado de T
  analiza_Ep(Ep)
  E.v= Ep.v       # Creamos un atributo sintetizado de E
```

Por claridad, hemos omitido el código de control de errores.

## 2.7. Atributos en GPDR

La interpretación que hemos hecho de los esquemas de traducción se traslada de forma natural a las GPDR. Por ejemplo, la traducción de:

$$\langle\mathsf{E}\rangle \quad\rightarrow\quad \langle\mathsf{T}\rangle_1\ \{\langle\mathsf{E}\rangle.\mathrm{v}{:=}\ \langle\mathsf{T}\rangle_1.\mathrm{v}\}(\texttt{+}\langle\mathsf{T}\rangle_2\ \{\langle\mathsf{E}\rangle.\mathrm{v}{:=}\ \langle\mathsf{E}\rangle.\mathrm{v}{+}\langle\mathsf{T}\rangle_2.\mathrm{v}\})^*$$

es simplemente:

```
def analiza_E(E):
  T1= Atributos() # Atributos de T1
  T2= Atributos() # Atributos de T2
  analiza_T(T1)
  E.v= T1.v
  while token.cat=="suma":
    token= alex.siguiente()
    analiza_T(T2)
    E.v= E.v+T2.v
```

Como antes, hemos omitido el control de errores.

# 3. El árbol de sintaxis abstracta

Como hemos comentado en la introducción, una de las posibles representaciones semánticas de la entrada es el árbol de sintaxis abstracta o AST.

Aunque es similar a los árboles de análisis sintáctico, tiene algunas diferencias importantes:

- No aparecen todos los componentes léxicos del programa. Por ejemplo:
  - No es necesario incluir los paréntesis de las expresiones.
  - No se necesitan los separadores o terminadores de las sentencias.
  - ...
- Pueden aparecer otros componentes no estrictamente sintácticos, como acciones de coerción de tipos.

## 3.1. Construcción

Para construir los árboles, debemos comenzar por definir qué elementos emplearemos para cada estructura:

| Estructura | Representación | Estructura | Representación |
|---|---|---|---|
| **if** ⟨E⟩ **then** ⟨LS⟩ **end** | **si** (hijos: ⟨E⟩ ⟨LS⟩) | **begin** $\langle S\rangle_1 \ldots \langle S\rangle_n$ **end** | **sentencias** (hijos: $\langle S\rangle_1 \ldots \langle S\rangle_n$) |
| **while** ⟨C⟩ **do** ⟨LS⟩ **end** | **mientras** (hijos: ⟨C⟩ ⟨LS⟩) | **id**:=⟨E⟩; | **asignación** (hijos: **id** ⟨E⟩) |
| **repeat** ⟨LS⟩ **until** ⟨C⟩; | **repetir** (hijos: ⟨LS⟩ ⟨C⟩) | $\langle E\rangle_1$+$\langle E\rangle_2$ | **suma** (hijos: $\langle E\rangle_1$ $\langle E\rangle_2$) |
| ... | ... | ... | ... |

Fíjate que el árbol resultante puede representar programas en diversos lenguajes de programación del estilo C, Pascal, etc.

Ahora debemos utilizar los atributos para construir el árbol. Utilizando el atributo arb para devolver el árbol que construye cada no terminal, podemos hacer algo parecido a:

⟨Sentencia⟩ → **if** ⟨Expresión⟩ **then** ⟨Sentencias⟩ **end**
  {⟨Sentencia⟩.arb:= NodoSi(⟨Expresión⟩.arb, ⟨Sentencias⟩.arb)}
⟨Sentencia⟩ → **while** ⟨Expresión⟩ **do** ⟨Sentencias⟩ **end**
  {⟨Sentencia⟩.arb:= NodoMientras(⟨Expresión⟩.arb, ⟨Sentencias⟩.arb)}
⟨Sentencia⟩ → **repeat** ⟨Sentencias⟩ **until** ⟨Expresión⟩;
  {⟨Sentencia⟩.arb:= NodoRepetir(⟨Sentencias⟩.arb, ⟨Expresión⟩.arb)}
⟨Sentencia⟩ → **id**:= ⟨Expresión⟩;
  {⟨Sentencia⟩.arb:= NodoAsignación(**id**.lexema,⟨Expresión⟩.arb)}

Para la implementación hay dos opciones principales:

- Utilizar funciones (NodoSi, NodoMientras, ...) que devuelvan una estructura de datos adecuada.

© Universitat Jaume I 2010-2011

- Utilizar objetos (NodoSi, NodoMientras, …) para cada uno de los nodos.

Probablemente, la segunda opción sea la más cómoda para el trabajo posterior con el árbol.

En cuanto a la lista de sentencias, podemos utilizar nodos que tengan un grado variable, para eso almacenamos una lista con los hijos:

$$\langle\text{Sentencias}\rangle \rightarrow \{\text{l} := [\,]\}\ (\langle\text{Sentencia}\rangle\ \{\text{l.añade}(\langle\text{Sentencia}\rangle.\text{arb})\})^*$$
$$\{\langle\text{Sentencias}\rangle.\text{arb} := \text{NodoSentencias(l)}\}$$

El tratamiento de las expresiones es sencillo. Por ejemplo, para la suma podemos hacer:

$$\langle\text{Expresión}\rangle \rightarrow \langle\text{Término}\rangle_1\ \{\text{arb:=}\langle\text{Término}\rangle_1.\text{arb}\}$$
$$(\ +\ \langle\text{Término}\rangle_2\ \{\text{arb:=NodoSuma(arb,}\langle\text{Término}\rangle_2.\text{arb)}\})^*$$
$$\{\langle\text{Expresión}\rangle.\text{arb:= arb}\}$$

Las restas, productos, etc, se tratarían de forma similar. Puedes comprobar que de esta manera el AST resultante respeta la asociatividad por la izquierda.

EJERCICIO* 5

En el caso de la asociatividad por la derecha tenemos dos opciones: crear una lista de operandos y recorrerla en orden inverso para construir el árbol o utilizar recursividad por la derecha, que no da problemas para el análisis LL(1). Utiliza ambas posibilidades para escribir sendos esquemas de traducción que construyan los AST para expresiones formadas por sumas y potencias de identificadores siguiendo las reglas habituales de prioridad y asociatividad.

## 3.2. Evaluación de atributos sobre el AST

Un aspecto interesante del AST es que se puede utilizar para evaluar los atributos sobre él, en lugar de sobre la gramática inicial. Si reflexionas sobre ello, es lógico. Todo el proceso de evaluación de los atributos se puede ver como el etiquetado de un árbol (el árbol de análisis), pero no hay nada que impida que el árbol sobre el que se realiza la evaluación sea el AST.

Un ejemplo sería el cálculo de tipos. Si tenemos la expresión `(2+3.5)*4`, podemos calcular los tipos sobre el árbol de análisis decorándolo como en la figura **??**.

Para poder calcular los tipos, podríamos utilizar una gramática similar a la siguiente[2]:

$$\langle\text{Expresión}\rangle \rightarrow \langle\text{Término}\rangle_1\ \{\text{t:= }\langle\text{Término}\rangle_1.\text{t}\}(+\langle\text{Término}\rangle_2\ \{\text{t:= másgeneral(t, }\langle\text{Término}\rangle_2.\text{t)}\})^*$$
$$\{\langle\text{Expresión}\rangle.\text{t:= t}\}$$

…

$$\langle\text{Término}\rangle \rightarrow \langle\text{Factor}\rangle_1\ \{\text{t:= }\langle\text{Factor}\rangle_1.\text{t}\}(*\langle\text{Factor}\rangle_2\ \{\text{t:= másgeneral(t, }\langle\text{Factor}\rangle_2.\text{t)}\})^*$$
$$\{\langle\text{Término}\rangle.\text{t:= t}\}$$

…

$$\langle\text{Factor}\rangle \rightarrow \textbf{num}\ \{\langle\text{Factor}\rangle.\text{t:= }\textbf{num}.\text{t}\}$$
$$\langle\text{Factor}\rangle \rightarrow \text{“(”}\langle\text{Expresión}\rangle\text{“)”}\ \{\langle\text{Factor}\rangle.\text{t:= }\langle\text{Expresión}\rangle.\text{t}\}$$

…

[2] Utilizamos la función másgeneral que devuelve el tipo más general de los que se le pasan como parámetros (el tipo real se considera más general que el entero).

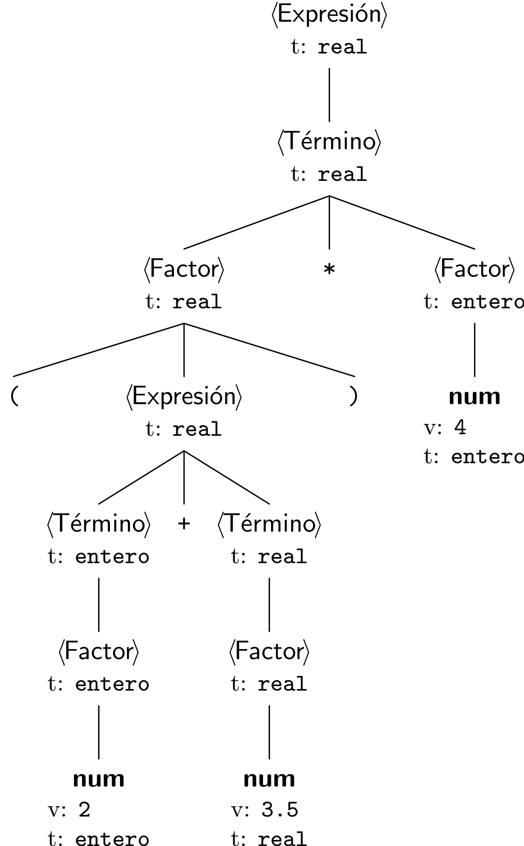
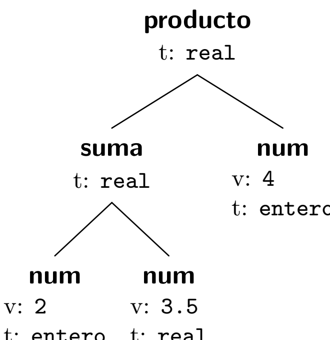



**Figura 1:** Árbol de análisis de la expresión `(2+3.5)*4` decorado con los tipos de los nodos.

También podemos realizar el cálculo sobre el AST:




Esto se haría junto con el resto de comprobaciones semánticas.

La elección acerca de si evaluar atributos en el AST o durante el análisis descendente es básicamente una cuestión de simplicidad. Generalmente, podemos decir que los atributos sintetizados tienen una dificultad similar en ambos casos. Sin embargo, cuando la gramática ha sufrido transformaciones o hay muchos atributos heredados, suele ser más fácil evaluarlos sobre el AST.

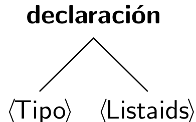
Otro aspecto interesante a la hora de decidir qué atributos evaluar sobre el árbol de análisis y cuáles sobre el AST está en los propios nodos del árbol. Supongamos que queremos tener nodos diferentes para la suma entera y la suma real. En este caso, tendremos que comprobar los tipos directamente sobre el árbol de análisis (o crear un AST y después modificarlo).

© Universitat Jaume I 2010-2011

# 4. Comprobaciones semánticas

Los atributos nos permitirán llevar a cabo las comprobaciones semánticas que necesite el lenguaje. En algunos casos, utilizaremos los atributos directamente (posiblemente evaluados sobre el AST), por ejemplo en la comprobación que hacíamos de que el identificador al final de la función era el mismo que al principio.

En otros casos, los atributos se utilizan indirectamente mediante estructuras globales, por ejemplo la tabla de símbolos. Un ejemplo sería la comprobación de que un identificador no se ha declarado dos veces. Si hemos utilizado un nodo similar a:




el método de comprobación sería:

**Objeto** NodoDeclaración:
  ...
  **Método** compsemánticas()
    ...
    **para** $v \in$ listaids **hacer**
      **si** TablaSímbolos.existe($v$) **entonces**
        *error*
      **si no**
        TablaSímbolos.inserta($v$, tipo)
      **fin si**
    **fin para**
    ...
  **fin** compsemánticas
  ...
**fin** NodoDeclaración

También podemos hacer las comprobaciones directamente en las acciones. Por ejemplo, podemos modificar el ejemplo de la sección **??**:

⟨DeclVariables⟩ → ⟨ListaIds⟩ : ⟨Tipo⟩
  {**para** $v \in$ ⟨ListaIds⟩.l **hacer**
    **si** TablaSímbolos.existe($v$) **entonces**
      *error*
    **si no**
      TablaSímbolos.inserta($v$, ⟨Tipo⟩.t)
    **fin si**
  **fin para**}
⟨ListaIds⟩ → ⟨ListaIds⟩$_1$ **,** **id** {⟨ListaIds⟩.l:= ⟨ListaIds⟩$_1$.l; ⟨ListaIds⟩.l.añade(**id**.lexema)}
⟨ListaIds⟩ → **id** {⟨ListaIds⟩.l:= [**id**.lexema]}

## 4.1. La tabla de símbolos

Durante la construcción del AST, las comprobaciones semánticas y, probablemente, durante la interpretación y la generación de código necesitaremos obtener información asociada a los distintos identificadores presentes en el programa. La estructura de datos que permite almacenar y recuperar esta información es la tabla de símbolos.

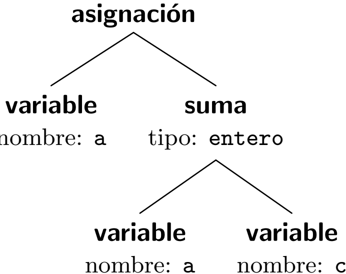
En principio, la tabla debe ofrecer operaciones para:

- Insertar información relativa a un identificador.
- Recuperar la información a partir del identificador.

La estructura que puede realizar estas operaciones de manera eficiente es la tabla *hash* (que en Python está disponible mediante los diccionarios). La implementación habitual es una tabla que asocia cada identificador a información tal como su naturaleza (constante, variable, nombre de función, etc.), su tipo (entero, real, booleano, etc.), su valor (en el caso de constantes), su dirección (en el caso de variables y funciones), etc.

Es importante tener únicamente una tabla; si tenemos varias, por ejemplo, una para constantes y otra para variables, el acceso se hace más difícil ya que para comprobar las propiedades de un identificador hay que hacer varias consultas en lugar de una.

Una cuestión importante es cómo se relacionan la tabla y el AST. Tenemos distintas posibilidades, que explicaremos sobre el siguiente árbol, correspondiente a la sentencia `a= a+c`:




La primera posibilidad es dejar el árbol tal cual y cada vez que necesitemos alguna información, por ejemplo el valor de `a`, consultar la tabla. Esta opción sería la más adecuada para situaciones en las que se vaya a recorrer el árbol pocas veces, por ejemplo en una calculadora donde se evalúe cada expresión una sola vez.

Otra posibilidad es ir decorando cada una de las hojas con toda la información que se vaya recopilando. Por ejemplo, si durante el análisis averiguamos el tipo y dirección de las variables, pasaríamos a almacenar la nueva información:

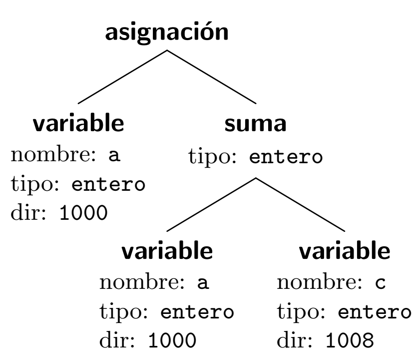


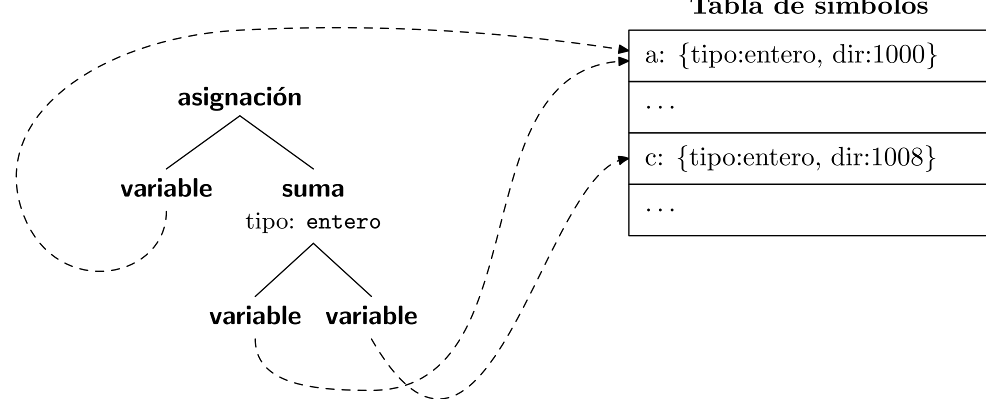

**Tabla de símbolos**

| |
|---|
| a: {tipo:entero, dir:1000} |
| ... |
| c: {tipo:entero, dir:1008} |
| ... |

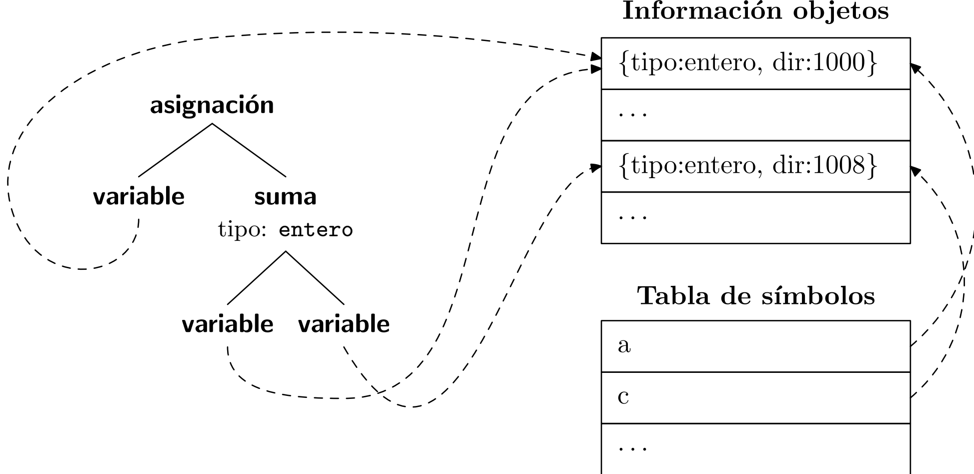
Esto tiene costes muy elevados en tiempo —tendremos que actualizar repetidamente un número de nodos que puede ser elevado— y espacio —hay mucha información que se almacena repetida—. Podemos reducir estos costes guardando en cada nodo un puntero a la entrada correspondiente en

© Universitat Jaume I 2010-2011

la tabla:




Finalmente, la opción más adecuada cuando tenemos lenguajes que presenten ámbitos anidados (veremos en otro tema cómo representar los ámbitos en la tabla) es guardar la información sobre los objetos en otra estructura de datos a la que apunta tanto la tabla como los nodos del árbol:




## 4.2. Comprobaciones de tipos

Un aspecto importante del análisis semántico es la comprobación de los tipos de las expresiones. Esta comprobación se hace con un doble objetivo:

- Detectar posibles errores.
- Averiguar el operador o función correcto en casos de sobrecarga y polimorfismo.

Para representar los tipos en el compilador, se emplean lo que se denominan *expresiones de tipo* (ET). La definición de estas expresiones es generalmente recursiva. Un ejemplo sería:

- Los tipos básicos: real, entero,..., además de los tipos especiales *error_de_tipo* y ausencia de tipo (*void*) son ETs.
- Si $n_1$ y $n_2$ son enteros, $\mathit{rango}(n_1, n_2)$ es una ET.
- Si $T$ es una ET, también lo es $\mathit{puntero}(T)$.
- Si $T_1$ y $T_2$ son ETs, también lo es $T_1 \rightarrow T_2$.
- Si $T_1$ y $T_2$ son ETs, también lo es $\mathit{vector}(T_1, T_2)$.
- Si $T_1$ y $T_2$ son ETs, también lo es $T_1 \times T_2$.

- Si $T_1,\ldots,T_k$ son ETs y $N_1,\ldots,N_k$ son nombres de campos, $\mathit{registro}((N_1,T_1),\ldots,(N_k,T_k))$ es una ET.

Ten en cuenta que según los lenguajes, estas definiciones cambian. Es más, los lenguajes pueden restringir qué expresiones realmente resultan en tipos válidos para los programas (por ejemplo, en Pascal no se puede definir un vector de funciones).

Algunos ejemplos de declaraciones y sus expresiones correspondientes serían:

| Declaración | *Expresión de tipo* |
|---|---|
| `int v[10];` | $\mathit{vector}(\mathit{rango}(0,9),\mathit{entero})$ |
| `struct {`<br>` int a;`<br>` float b;`<br>`} st;` | $\mathit{registro}((a,\mathit{entero}),(b,\mathit{real}))$ |
| `int *p;` | $\mathit{puntero}(\mathit{entero})$ |
| `int f(int, char);` | $\mathit{entero} \times \mathit{carácter} \rightarrow \mathit{entero}$ |
| `void f2(float);` | $\mathit{real} \rightarrow \mathit{void}$ |

### 4.2.1. Equivalencia de tipos

Comprobar si dos expresiones de tipo, $T_1$ y $T_2$, son equivalentes es muy sencillo. Si ambas corresponden a tipos elementales, son equivalentes si son iguales. En caso de que correspondan a tipos estructurados, hay que comprobar si son el mismo tipo y si los componentes son equivalentes.

En muchos lenguajes se permite dar nombre a los tipos. Esto introduce una sutileza a la hora de comprobar la equivalencia. La cuestión es, dada una declaración como

```
typedef int a;
typedef int b;
```

¿son equivalentes `a` y `b`? La respuesta depende del lenguaje (o, en casos como el Pascal, de la implementación). Existen dos criterios para la equivalencia:

**Equivalencia de nombre:** dos expresiones de tipo con nombre son equivalentes si y sólo si tienen el mismo nombre.

**Equivalencia estructural:** dos expresiones de tipo son equivalentes si y sólo si tienen la misma estructura.

Hay argumentos a favor de uno y otro criterio. En cualquier caso, hay que tener en cuenta que para comprobar la equivalencia estructural ya no sirve el método trivial presentado antes. En este caso, puede haber ciclos, lo que obliga a utilizar algoritmos más complicados.

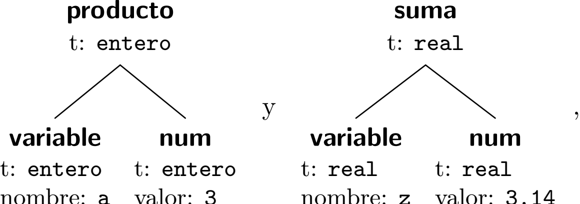
### 4.2.2. Comprobación de tipos en expresiones

Para comprobar los tipos en las expresiones podemos utilizar un atributo que indique el tipo de la expresión. Este tipo se infiere a partir de los distintos componentes de la expresión y de las reglas de tipos del lenguaje.

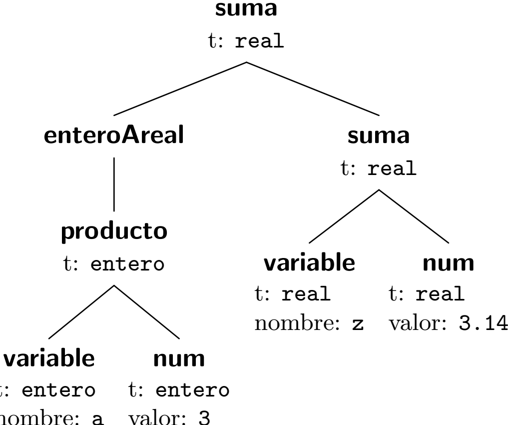
A la hora de calcular el tipo hay varias opciones: podemos calcularlo sobre la gramática, sobre el AST o al construir el AST. En cualquier caso, puede ser útil tener una función similar a másgeneral (la que usábamos en la sección **??**) que refleje las peculiaridades de los tipos del lenguaje. Por ejemplo, si tenemos tipos entero y real con las reglas habituales de promoción, esta función devolvería los resultados según esta tabla:

| Primero | Segundo | Resultado |
|---|---|---|
| *entero* | *entero* | *entero* |
| *entero* | *real* | *real* |
| *real* | *entero* | *real* |
| *real* | *real* | *real* |
| *otros* | | *error_tipo* |

El cálculo de atributos de tipo debería ser trivial sobre la gramática o sobre el AST. En cuanto al cálculo al construir el AST, se puede, por ejemplo, hacer que el constructor compruebe los tipos de los componentes que recibe. En este caso, si es necesario, se puede añadir un nodo de promoción de tipos. Por ejemplo, si vamos a crear un nodo suma a partir de los árboles:




podemos añadir un nodo intermedio para indicar la promoción:




También podemos conseguir este efecto en las comprobaciones semánticas. Para ello, una vez averiguado el tipo, se introduce uno o varios nodos intermedios. Para el NodoSuma podríamos hacer:

**Objeto** NodoSuma:
    ...
    **Método** compsemánticas()
        ...
        **si** izdo.tipo= *entero* **y** tipo= *real* **entonces**
            izdo:= NodoEnteroAReal(izdo)
        **fin si**
        **si** dcho.tipo= *entero* **y** tipo= *real* **entonces**
            dcho:= NodoEnteroAReal(izdo)
        **fin si**
        ...
    **fin** compsemánticas
    ...
**fin** NodoSuma

Respecto a la propagación de los errores de tipo, hay que intentar evitar un exceso de mensajes de error. Para ello se pueden utilizar distintas estrategias:

- Se puede hacer que *error_tipo* sea compatible con cualquier otro tipo.
- Se puede intentar inferir cuál sería el tipo en caso de no haber error. Por ejemplo: si el operador en que se ha detectado el error es el y-lógico, se puede devolver como tipo el tipo lógico.

# 5. Interpretación

La utilización del AST hace que la interpretación sea bastante sencilla. Una vez lo hemos construido, tenemos dos aproximaciones para la interpretación:

- Podemos representar la entrada mediante una serie de instrucciones similares a un lenguaje máquina con algunas características de alto nivel. Este es el caso, por ejemplo, de Java y el Java bytecode. Esta aproximación representa prácticamente una compilación.
- La otra alternativa es representar la entrada de manera similar al AST y recorrerlo para ejecutar el programa.

Utilizaremos la segunda opción. Los nodos correspondientes a las expresiones tendrán un método al que llamaremos *evalúa* y que devolverá el resultado de evaluar el subárbol correspondiente. Por ejemplo, en el nodo de la suma tendríamos:

```
Objeto NodoSuma:
    ...
    Método evalúa()
        devuelve i.evalúa() + d.evalúa(); // i y d son los hijos del nodo
    fin evalúa
    ...
fin NodoSuma
```

Aquellos nodos que representen sentencias tendrán el método *ejecuta*. Por ejemplo, para un nodo que represente un mientras, el método correspondiente sería:

```
Objeto NodoMientras:
    ...
    Método ejecuta()
        mientras condición.evalúa()=cierto:
            sentencias.ejecuta()
        fin mientras
    fin ejecuta
    ...
fin NodoMientras
```

Las variables se representarían mediante entradas en una tabla (que, en casos sencillos, puede ser la tabla de símbolos) que contenga el valor correspondiente en cada momento. Una asignación consiste simplemente en evaluar la expresión correspondiente y cambiar la entrada de la tabla.

La ejecución del programa consistirá en una llamada al método *interpreta* del nodo raíz del programa. Alternativamente, si hemos guardado el programa como una lista de sentencias, la ejecución consistirá en un bucle que recorra la lista haciendo las llamadas correspondientes.

Lógicamente, existen otras maneras de recorrer el árbol. En particular, si no lo hemos representado con objetos, podemos recorrerlo mediante funciones recursivas o de manera iterativa. En este último caso, podemos emplear una pila auxiliar o ir "enhebrando" el árbol durante su construcción. Esta es probablemente la opción más rápida con una implementación cuidadosa, pero es también la que más esfuerzo exige (especialmente si hay que recorrer el árbol en distintos órdenes).

© Universitat Jaume I 2010-2011

# 6. Introducción a metacomp

Mediante `metacomp` se pueden traducir esquemas de traducción a programas Python que implementan los correspondientes analizadores descendentes recursivos.

Un programa `metacomp` se divide en varias secciones. Cada sección se separa de la siguiente por un carácter "%" que ocupa la primera posición de una línea. La primera sección contiene la especificación del analizador léxico. Las secciones pares contienen código de usuario. La tercera y sucesivas secciones impares contienen el esquema de traducción. Es decir, un programa `metacomp` se escribe en un fichero de texto siguiendo este formato:

*Especificación léxica*
%
*Código de usuario*
%
*Esquema de traducción*
%
*Código de usuario*
%
*Esquema de traducción*
⋮

Las secciones de "código de usuario" se utilizan para declarar estructuras de datos y variables, definir funciones e importar módulos útiles para el procesador de lenguaje que estamos especificando. Estos elementos deberán codificarse en Python. La herramienta `metacomp` copia literalmente estos fragmentos de código en el programa que produce como salida.

## 6.1. Especificación léxica

La especificación léxica consiste en una serie de líneas con tres partes cada una:

- Un nombre de categoría o la palabra `None` si la categoría se omite.
- Un nombre de función de tratamiento o `None` si no es necesario ningún tratamiento.
- Una expresión regular.

Los analizadores léxicos generados por `metacomp` dividen la entrada siguiendo la estrategia avariciosa y, en caso de conflicto, prefiriendo las categorías que aparecen en primer lugar. Por cada lexema encontrado en la entrada se llama a la correspondiente función de tratamiento, que normalmente se limita a calcular algún atributo adicional a los tres que tienen por defecto todos los componentes: lexema, nlinea y cat, que representan, respectivamente, el lexema, el número de línea y la etiqueta de categoría del componente.

Un ejemplo de especificación léxica para una calculadora sencilla sería:

| categoría | expresión regular | acciones | atributos |
|---|---|---|---|
| **num** | [0–9]$^+$ | calcular valor<br>emitir | valor |
| **opad** | `[-+]` | copiar lexema<br>emitir | lexema |
| **abre** | `\(` | emitir | |
| **cierra** | `\)` | emitir | |
| **espacio** | `[␣\t]`$^+$ | omitir | |
| **nl** | `\n` | emitir | |

En `metacomp`, lo podemos escribir así:

```
num    calculaValor [0-9]+
opad   None [-+]
```

```
abre   None \(
cierra None \)
None   None [ \t]+
nl     None \n
```

Donde `calculaValor` sería una función como la siguiente:

```
def calculaValor(componente):
  componente.v = int(componente.lexema)
```

No necesitamos ningún tratamiento para `opad` ya que `metacomp` añade por defecto el atributo lexema (que es el que hemos usado para calcular el valor de los enteros).

## 6.2. Esquema de traducción

Las secciones de esquema de traducción contienen las reglas de la GPDR utilizando una sintaxis muy similar a la de la asignatura. Cada regla tiene una parte izquierda, un símbolo `->`, una parte derecha y un punto y coma. Los no terminales se escriben marcados mediante `<` y `>`. Los terminales tienen que coincidir con los declarados en la especificación léxica[3]. Se pueden emplear los operadores regulares de concatenación (implícita), disyunción (mediante `|`), opcionalidad (mediante `?`), clausura (mediante `*`) y clausura positiva (mediante `+`). Por ejemplo, la regla

$$\langle \mathsf{E} \rangle \quad \rightarrow \quad \langle \mathsf{T} \rangle (\textbf{opad} \langle \mathsf{T} \rangle)^*$$

se escribiría en `metacomp` así:

```
<E> -> <T> ( opad <T> )*;
```

Las acciones de las reglas se escriben encerradas entre arrobas (`@`) y sin fin de línea entre ellas. Para poder referirse a los terminales y no terminales, estos están numerados por su orden de aparición en la parte derecha de la regla. El no terminal de la izquierda no tiene número y, si no hay ambigüedad, la primera aparición de cada símbolo no necesita número. Si tenemos las acciones de la regla anterior:

$$\begin{aligned} \langle \mathsf{E} \rangle \quad \rightarrow \quad & \langle \mathsf{T} \rangle_1 \; \{\text{v:= } \langle \mathsf{T} \rangle_1.\text{v}\} \\ & (\textbf{opad} \; \langle \mathsf{T} \rangle_2 \; \{\textbf{si } \textbf{opad}.\text{lexema= ``+'' } \textbf{entonces} \text{ v:= v+} \langle \mathsf{T} \rangle_2.\text{v } \textbf{si no} \text{ v:= v}-\langle \mathsf{T} \rangle_2.\text{v } \textbf{fin si}\})^* \\ & \{\langle \mathsf{E} \rangle.\text{v:= v}\} \end{aligned}$$

podemos escribirlas en `metacomp` así:

```
<E> -> <T> @v= T.v@
       ( opad <T>
         @if opad.lexema=="+":@
         @  v+= T2.v@
         @else:@
         @  v-= T2.v@
       )*
       @E.v= v@;
```

# 7. Algunas aplicaciones

Vamos a aprovechar `metacomp` para escribir un par de utilidades de procesamiento de ficheros: una para sumar las columnas de un fichero con campos separados por tabuladores y otra para leer ficheros de configuración.

[3] Existe también la posibilidad de utilizar una sintaxis especial para categorías con un sólo lexema, pero no la veremos ahora.

### 7.1. Ficheros organizados por columnas

Supongamos que tenemos un fichero de texto en el que aparecen secuencias de dígitos separadas por tabuladores que definen columnas. Queremos sumar la columna $n$ de la tabla.

Podemos emplear la siguiente especificación léxica:

| categoría | expresión regular | acciones | atributos |
|---|---|---|---|
| número | $[0–9]^+$ | calcular valor<br>emitir | valor |
| separación | `\t` | emitir | |
| linea | `\n` | emitir | |
| blanco | $␣^+$ | omitir | |

La representación en `metacomp` de la especificación léxica podría ser:

```
1   numero     calculaValor   [0-9]+
2   separacion None    \t
3   nl         None    \n
4   None       None    [ ]+
```

La función `calculaValor` se define en la zona de auxiliares junto con dos funciones para tratamiento de errores:

```
5   %
6   def calculaValor(componente):
7     componente.v = int(componente.lexema)
8
9   def error(linea, mensaje):
10    sys.stderr.write("Error en línea %d: %s\n" % (linea, mensaje))
11    sys.exit(1)
12
13  def error_lexico(linea, cadena):
14    error(linea, "no he podido analizar %s." % repr(cadena))
15
16  ncol= 0
```

La función `error` la utilizamos para escribir mensajes genéricos. La función `error_lexico` es llamada por `metacomp` cuando encuentra un error léxico. Los dos parámetros que recibe son la línea donde se ha producido el error y el carácter o caracteres que no se han podido analizar.

Podemos emplear el siguiente esquema de traducción:

⟨Fichero⟩ → {suma:=0}(⟨Línea⟩ **nl** {suma:= suma+⟨Línea⟩.v})* {⟨Fichero⟩.suma:= suma}

⟨Línea⟩ → **numero**$_1$ {**global** ncol; col:=1; **si** col= ncol **entonces** ⟨Línea⟩.v:= **numero**.v **fin si**}
(**separación número**$_2$
{col:= col+1; **si** col= ncol **entonces** ⟨Línea⟩.v:= **numero**$_2$.v **fin si**}
)*

Hemos supuesto que el símbolo inicial tiene un atributo sintetizado que leerá el entorno. Además, la columna deseada está en la variable global *ncol*. Representamos esto en `metacomp` así:

```
17  %
18  <Fichero>->   @suma = 0@
19              ( <Linea> @suma += Linea.v@ nl )*
20                @Fichero.suma = suma@ ;
```

```
21
22   <Fichero>-> error @error(mc_al.linea(), "línea mal formada")@ ;
23
24   <Linea>-> numero
25               @col = 1@
26               @if col == ncol:@
27               @  Linea.v = numero.v@
28             ( separacion numero
29                 @col += 1@
30                 @if col == ncol:@
31                 @  Linea.v = numero2.v@
32             )*
33             @if col < ncol:@
34             @  error(numero.nlinea, "no hay suficientes columnas")@
35             ;
36
```

La regla `<Fichero>-> error` nos permite capturar los errores que se produzcan durante el análisis. Hay varias maneras de tratar los errores, pero hemos optado por abortar el programa. La variable `mc_al` contiene el analizador léxico y la usamos para averiguar el número de línea del error.

Finalmente, el programa tendrá un parámetro que indique qué columna sumar. Si no se especifica ningún parámetro adicional, se leerá la entrada estándar. En caso de que haya varios parámetros, se sumará la columna deseada de cada uno de los ficheros especificados, escribiéndose el total. Con esto, el programa principal es:

```
37   %
38   def main():
39     global ncol
40     if len(sys.argv) == 1:
41       sys.stderr.write("Error, necesito un número de columna.\n")
42       sys.exit(1)
43     try:
44       ncol = int(sys.argv[1])
45     except:
46       sys.stderr.write("Error, número de columna mal formado.\n")
47       sys.exit(1)
48
49     if len(sys.argv) == 2:
50       A = AnalizadorSintactico(sys.stdin)
51       suma = A.Fichero.suma
52     else:
53       suma = 0
54       for arg in sys.argv[2:]:
55         try:
56           f = open(arg)
57         except:
58           sys.stderr.write("Error, no he podido abrir el fichero %s.\n" % arg)
59           sys.exit(1)
60         A = AnalizadorSintactico(f)
61         suma += A.Fichero.suma
62     print "La suma es %d." % suma
```

Por defecto, `metacomp` genera una función `main` que analiza `sys.stdin`. Si queremos modificar ese comportamiento, debemos crear nuestra propia función. Como ves, la mayor parte del código se

© Universitat Jaume I 2010-2011

dedica a analizar posibles errores en los parámetros. La construcción más interesante está en las dos líneas que crean el analizador. La asignación

```
A= AnalizadorSintactico(f)
```

hace que se cree un analizador que inmediatamente analizará el fichero `f` (que debe estar abierto). Cuando termina el análisis, el objeto `A` tiene un atributo con el nombre del símbolo inicial que contiene sus atributos sintetizados. Así es como transmitimos el resultado.

## 7.2. Ficheros de configuración

Vamos ahora a programar ahora un módulo de lectura de ficheros de configuración. Supondremos que estos ficheros tienen líneas con el formato: *variable* = *valor*. Los nombres de variable son secuencias de letras y dígitos que comienzan por una letra. Los valores pueden ser cadenas entre comillas (sin comillas en su interior) o números enteros. Se permiten comentarios que comienzan por el carácter `#` y terminan al final de la línea y también se permiten líneas vacías.

Haremos que el resultado del análisis sea un diccionario en el que las claves sean las variables y los valores asociados sean los leídos del fichero.

Utilizamos la siguiente especificación léxica:

| categoría | expresión regular | acciones | atributos |
|---|---|---|---|
| **variable** | [a–zA–Z][a–zA–Z0–9]* | copiar lexema<br>emitir | lexema |
| **igual** | = | emitir | |
| **valor** | [0–9]+\|"[^"\n]*" | copiar lexema<br>emitir | lexema |
| **linea** | (#[^\n]*)?\n | emitir | |
| **blanco** | [␣\t]+ | omitir | |

Con lo que el analizador léxico se escribiría así:

```
1   variable   None    [a-zA-Z][a-zA-Z0-9]*
2   igual      None    =
3   valor      None    [0-9]+|"[^"\n]*"
4   nl         None    (#[^\n]*)?\n
5   None       None    [ \t]+
```

Los únicos auxiliares que necesitamos son las funciones de tratamiento de errores:

```
6   %
7   def error(linea, mensaje):
8     sys.stderr.write("Error en línea %d: %s\n" % (linea, mensaje))
9     sys.exit(1)
10
11  def error_lexico(linea, cadena):
12    error(linea, "no he podido analizar %s." % repr(cadena))
```

Nuestro esquema de traducción será:

$$\langle\text{Fichero}\rangle \rightarrow \{\text{dic:= }\{\}\}((\langle\text{Línea}\rangle\ \{\text{dic}[\langle\text{Línea}\rangle.\text{izda}]\text{:= }\langle\text{Línea}\rangle.\text{dcha}\}|\lambda)\ \mathbf{nl})^*\ \{\langle\text{Fichero}\rangle.\text{dic:= dic}\}$$

$$\langle\text{Línea}\rangle \rightarrow \mathbf{variable}\ \{\langle\text{Línea}\rangle.\text{izda:= }\mathbf{variable}.\text{lexema}\}\ \mathbf{igual}\ \mathbf{valor}\ \{\langle\text{Línea}\rangle.\text{dcha:= }\mathbf{valor}.\text{lexema}\}$$

Pasado a `metacomp`:

```
13   %
14   <Fichero>->   @dic = {}@
15               ( (<Linea> @dic[Linea.izda] = Linea.dcha@)? nl )*
16                 @Fichero.dic = dic@ ;
17
18   <Fichero>-> error @error(mc_al.linea(), "línea mal formada")@ ;
19
20   <Linea>-> variable
21               @Linea.izda = variable.lexema@
22             igual valor
23               @Linea.dcha = valor.lexema@
24             ;
25
```

Como ves, hemos utilizado el operador de opcionalidad para representar la disyunción ($\langle$Linea$\rangle|\lambda$).

# 8. Resumen del tema

- El analizador semántico tiene dos objetivos:
  - Hacer comprobaciones que no se hagan durante el análisis léxico o sintáctico.
  - Crear una representación adecuada para fases posteriores.
- Implementaremos el análisis semántico en dos partes:
  - Mediante esquemas de traducción dirigidos por la sintaxis.
  - Recorriendo el AST.
- Un esquema de traducción dirigido por la sintaxis añade a las gramáticas:
  - Acciones intercaladas en las partes derechas de las reglas.
  - Atributos asociados a los no terminales.
- Dos tipos de atributos: heredados y sintetizados.
- Las acciones deben garantizar que se evalúan correctamente los atributos.
- Se pueden implementar los esquemas de traducción sobre los analizadores sintácticos interpretando los atributos como parámetros y añadiendo el código de las acciones al código del analizador.
- El cálculo de algunos atributos y algunas comprobaciones semánticas son más fáciles sobre el AST.
- La tabla de símbolos se puede implementar eficientemente mediante una tabla *hash*.
- Las comprobaciones de tipos se complican si se introducen nombres. Dos criterios:
  - Equivalencia de nombre.
  - Equivalencia estructural.
- La interpretación se puede realizar mediante recorridos del AST.
- `metacomp` permite implementar cómodamente esquemas de traducción dirigidos por la sintaxis.

© Universitat Jaume I 2010-2011