Tema 1.6. Un lenguaje mínimo y su procesador: *Implementación*

Profesor
Federico Peinado

Elaboración del material
José Luis Sierra
Federico Peinado

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

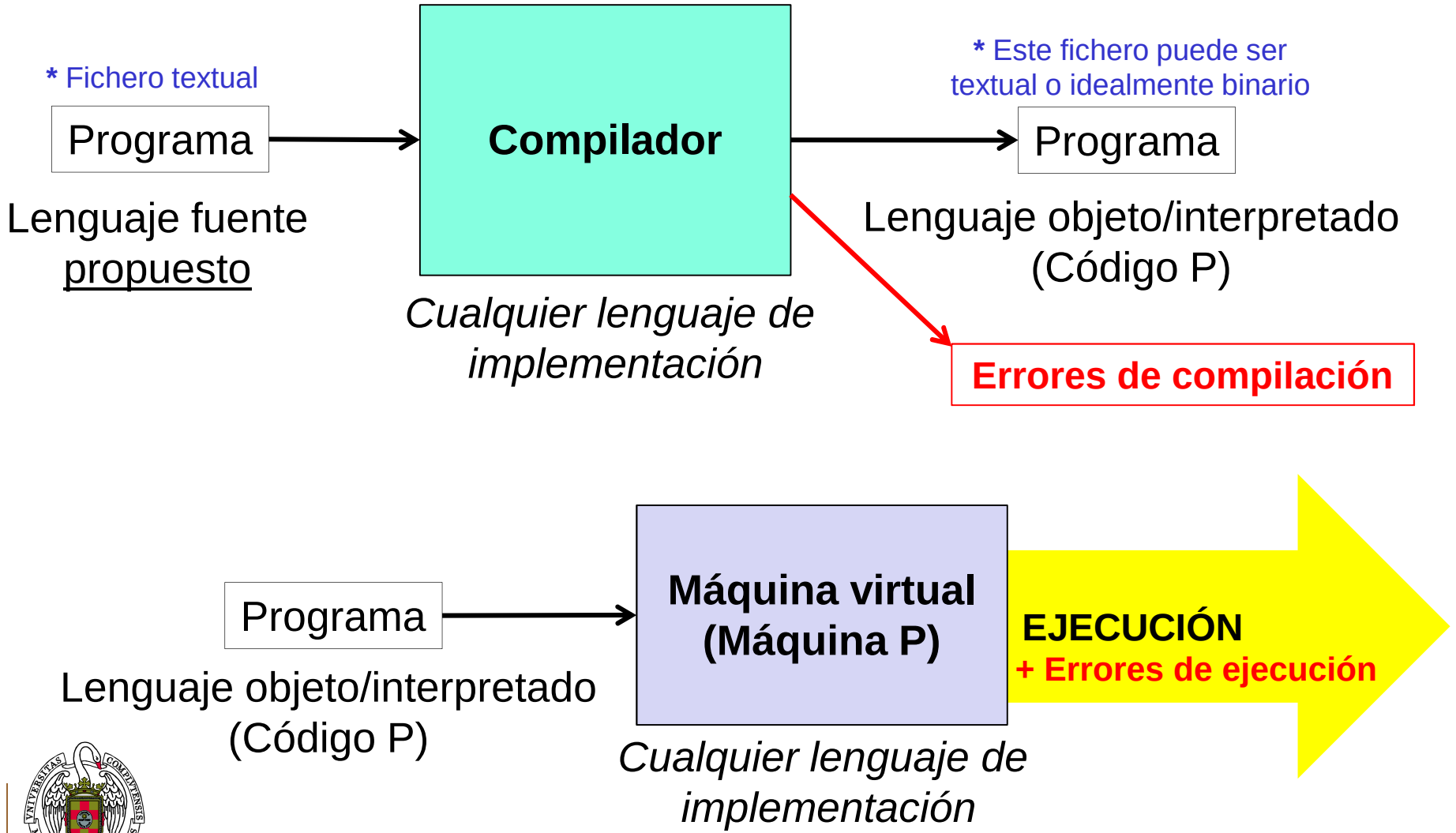
Implementación

- Una vez especificado el lenguaje de programación y sus procesadores, es hora de ponerse a **implementar**

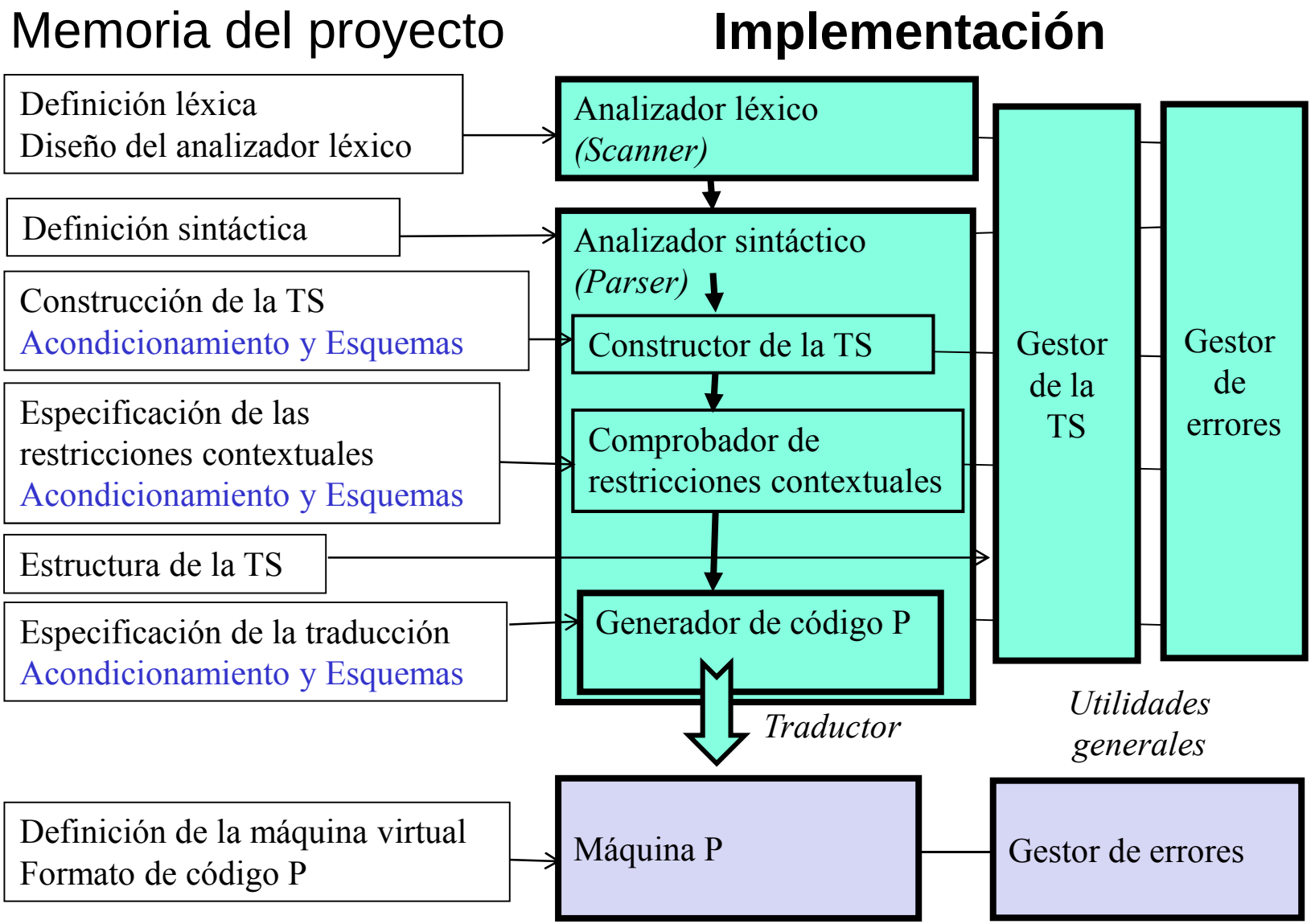
- Si se utiliza una herramienta de generación automática de procesadores de lenguaje bastará con *adaptar* la especificación al *formato de entrada* de la herramienta

- Si no se utiliza, entonces es necesario:
  1. Elegir un **lenguaje de implementación** concreto para los procesadores (aquí asumiremos paradigma **imperativo**)
  2. **Convertir las *especificaciones* de los procesadores** a *programas* en el lenguaje de implementación escogido
     - Esta conversión suele ser **manual**, ya que depende del lenguaje y el entorno de desarrollo escogidos, aunque debe realizarse de la forma más **sistemática** posible

Conexión entre el compilador y la máquina virtual propuestos



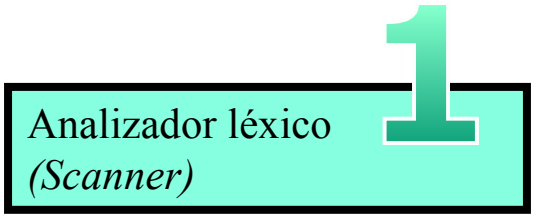
Estructura modular del compilador y la máquina virtual propuestos



Metodología a seguir para la implementación

- Una metodología sencilla, manual pero sistemática, para implementar el procesador, se basa en el **algoritmo de análisis descendente predictivo recursivo**
  - Algoritmo de **análisis** (sintáctico) que, como en otras metodologías, guía la **síntesis** también (ej. la traducción)
  - Esta **traducción dirigida por sintaxis** se lleva a cabo en la misma pasada en que se realiza el análisis del programa
- Etapas:
  1. Implementación del analizador léxico del lenguaje
  2. Implementación del analizador sintáctico del lenguaje y del generador de código
     1. Acondicionamiento de las gramáticas de atributos de la definición sintáctica del lenguaje y su traducción
     2. Especificación de los esquemas de traducción
     3. Implementación del generador a partir de los esquemas
  3. Implementación de la máquina virtual
     (para el caso del compilador propuesto, que genera un código para el que no se dispone de intérprete)

Implementación del analizador léxico



- Convierte el programa de entrada (cadena de *símbolos* alfanuméricos) en una secuencia de componentes léxicos (*ocurrencias* de *categorías léxicas*)
  - Cada componente léxico lleva **el identificador de su categoría léxica** y un conjunto de **atributos léxicos** (ej. **lexema** concreto que ha sido analizado)
- Se usa la definición léxica (expresiones regulares) para diseñar un reconocedor del lenguaje (autómata finito determinista) y después implementarlo según el diseño
  - La implementación es **manual**, usando el diseño sólo como "guía" (existen *algoritmos formales*, como los que usa la herramienta automática LEX, pero son muy tediosos…)
- En teoría, primero se realiza el análisis léxico y luego el sintáctico, pero ***en la práctica ambos se van realizando en la misma pasada***

Diseñar el reconocedor usando la definición léxica

- Considerar las ERs de las definiciones regulares de cada categoría léxica
- Añadir otra ER con lo necesario para tratar separadores, comentarios, reconocer el final del fichero, etc.
- Obtener un AFND para reconocer cada una de las ERs
- Solapar todos los AFNDs en un solo AFND (como haciendo la disyunción entre todas las ERs)
  - Un único estado inicial para todo
  - Uno o varios estados finales por cada categoría léxica
  - Mínimo número de estados intermedios que se puedan (tratando, por lo tanto, de *minimizar el tamaño del autómata*)
- Modificar el AFND resultante para convertirlo en un AFD *(si es que todavía tiene características no deterministas)*

Ejemplo de implementación de un analizador léxico

e_1 Identificador

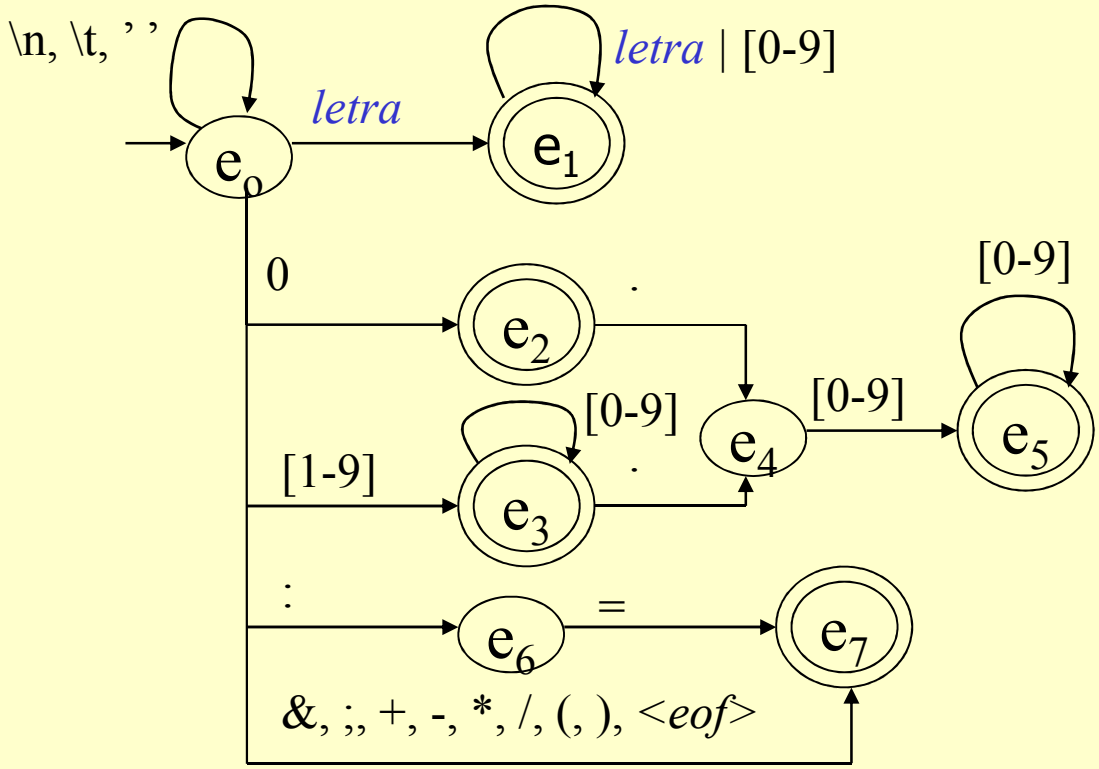
e_2 Entero

e_3 Entero

e_5 Real

…

(en realidad e_7 contiene varios "subestados")



***** En vez de **letra** podría ser cualquier símbolo de un cierto estándar

Implementación del analizador léxico según el diseño

- Hay varias estrategias a seguir para la implementación manual de un analizador léxico, proponemos una de las más habituales y sencillas:

1. Representar el AFD del reconocedor en forma de tabla de transición de estados
2. Codificar el autómata como un programa que…
   1. Tiene un buffer de entrada con la cadena de caracteres alfanuméricos a procesar
   2. Tiene un estado activo en cada momento
   3. Tiene un ciclo de ejecución con una instrucción *case*, que realiza la función de transición de estado. En cada caso del *case* se consume la parte correspondiente de la entrada del autómata y se cambia de estado

Ejemplo de implementación de un analizador léxico

- Pseudocódigo

```
fun scanner()
  lex ← ""
  estado ← e0
  repetir
    caso estado de
      e0: caso buff de
            {\t,\n,' '}: transita(e0); lex ←"";
            [a-z]|[A-Z]: transita(e1);
            0:           transita(e2);
            [1-9]:       transita(e3);
            ':':         transita(e6);
            {&,;,+,-,*,/,(,),<eof>}: tok ← token(buff);
                                     transita(e7);
           si no error(e0)
          fcaso
      e1: caso
            [a-z]|[A-Z]: transita(e1);
            si no devuelve Iden
          fcaso
```

* En **azul** las categorías léxicas (valores enumerados)

Ejemplo de implementación de un analizador léxico

Continuación…

```
e2: caso
      .: transita(e4);
      si no devuelve Num
    fcaso
e3: caso
      [0-9]: transita(e3);
      .: transita(e4)
      si no devuelve Num
    fcaso
e4: caso
      [0-9]: transita(e5)
      si no error(e4)
    fcaso
e5: caso
      [0-9]: transita(e5)
      si no devuelve Num
    fcaso
e6: caso
      =: tok ← Asig; transita(e7);
      si no error(e6)
    fcaso
```

Ejemplo de implementación de un analizador léxico

Continuación…

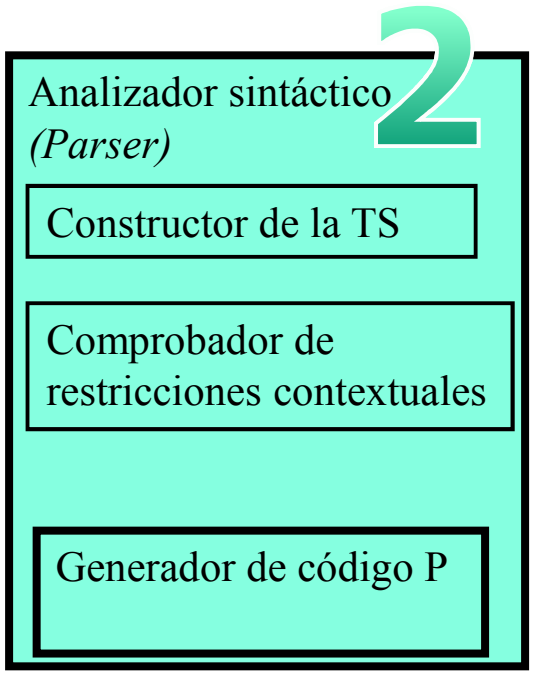
```
      e7: devuelve tok;
    fcaso
  frepite
ffun


fun transita(s)
  lex ← lex ++ buff; buff ← sigCar(); estado ← s
  (* puede mantenerse más información del estado, como
     por ejemplo un contador de líneas para mejorar la
     emisión de mensajes de error, etc. *)
ffun


fun error(s)
  (* Tratamiento del error dependiente del estado. Se
     informa del error y se aborta la ejecución *)
  mensajeError("Caracter inesperado: " ++ buff);
  abortar;
ffun
```

Ejemplo de implementación de un analizador léxico

Continuación…

```
fun token(car)
(* Una especie de tabla con “caracteres reservados” *)
  caso car
    &: devuelve Sep;
    ;: devuelve PuntoyComa;
   (* Y así para cada operador, separador, etc. *)
  fcaso
ffun


fun inicioScan()
  (* Tras abrir el fichero e inicializar todo... *)
  buff ← sigCar();
ffun
```

Implementación del analizador sintáctico



- Dada la secuencia de componentes léxicos de entrada decide si el programa que representa es válido o no en el lenguaje propuesto

- Se usa la definición sintáctica (gramática incontextual + restricciones contextuales) directamente para implementarlo (no hace falta llegar a diseñar el *autómata a pila* que reconoce el lenguaje)
  - La implementación es **manual**, usando la gramática incontextual y las restricciones contextuales sólo como "guías" (existen *algoritmos formales*, como los que usa la herramienta automática YACC, que se verán más tarde…)

Algoritmo de análisis sintáctico descendente predictivo recursivo

- Metodología sencilla para implementar manualmente el análisis (y de paso la traducción y el resto de aspectos):
  1. Considerar la gramática incontextual y **cambiar el axioma** para reconocer el final del programa
  2. Convertir los *no terminales* en **procedimientos**
  3. Convertir las *producciones* asociadas con dichos *no terminales* en el **cuerpo de dichos procedimientos**
     1. Si hay varias producciones de un *no terminal* en el cuerpo *se elige cual aplicar* según el **elemento de preanálisis**, el componente léxico de la entrada (existen otras estrategias, como el *backtracking* o los *algoritmos formales* que veremos en temas posteriores)
     2. Cada aparición de un *no terminal* significa **llamar al procedimiento** correspondiente
     3. Cada aparición de un *terminal* significa o bien **consumir el componente léxico de la entrada** o bien **informar de que hay un error sintáctico** en el programa

Algoritmo de análisis sintáctico descendente predictivo recursivo

- **Descendente** porque simula la expansión del árbol sintáctico, empezando por la raíz (el axioma) y "descendiendo" hasta las hojas

- **Predictivo** porque usa el *elemento de preanálisis* para "predecir" que producción debe aplicarse cuando hay varias asociadas a un mismo *no terminal*

- **Recursivo** porque si hay recursión en la gramática incontextual, habrá llamadas recursivas en el código

Ejemplo de implementación de un analizador sintáctico

- Si partimos de una gramática que tiene como *axioma* el *no terminal* S, con unas ciertas producciones:

```
S ::= a S b
S ::= c
...
```

- Debemos añadir otro *no terminal* y otra *producción* para **cambiar el axioma** y poder detectar así cuando se ha reconocido un programa correcto **y llegado a su final**:

```
SFin ::= S fin
S ::= a S b
S ::= c
...
```

* Siendo **fin** la categoría léxica correspondiente al fin de fichero (<*eof*>)

Ejemplo de implementación de un analizador sintáctico

- Pseudocódigo

```
fun SFin()
  (* SFin ::= S fin *)
  S();
  reconoce(fin);
  escribe("Programa correcto")
ffun;


fun S;
  si entrada = a entonces (* Elemento de preanálisis *)
    (* S ::= a S b *)
    reconoce(a);
    S;
    reconoce(b)
  si no
    (* S ::= c *)
    reconoce(c)
  fsi
ffun
```

Ejemplo de implementación de un analizador sintáctico

Continuación…

```
fun reconoce(token);
  si actual = token entonces
    actual ← sigTok();
  si no
    escribe("Error: Programa incorrecto")
    abortar;
  fsi
ffun


principal:
  actual ← sigTok();
  SFin();
```

Acondicionamiento de la gramática incontextual

- "Acondicionar" una gramática incontextual es **modificarla** para evitar problemas que surgen en la implementación
- Modificar una gramática incontextual implica **modificar todas las gramáticas de atributos** que hemos hecho, tanto las producciones como las ecuaciones semánticas (queremos preservar tanto la sintaxis como la semántica)
  - Las gramáticas de atributos tienen que ser l-atribuidas
  - Tenemos gramáticas de atributos hechas para especificar la construcción de la tabla de símbolos, la comprobación de las restricciones contextuales y toda la traducción
- Estudiaremos algún algoritmo formal para acondicionar gramáticas, pero los más complejos no los estudiaremos y habrá que resolver ciertos casos "a mano"

Acondicionamiento: Eliminación de recursión a izquierdas

- **Problema**: Las producciones con recursión a la izquierda son *una recursión infinita* según nuestra implementación:

```
fun X()
  (* X ::= X a *)
  X();
  reconoce(a);
ffun;
```

- **Solución:** Eliminar cualquier recursión a izquierdas (directa o indirecta) de las producciones, transformando la gramática en otra equivalente con recursión a derechas
  - Patrón típico para eliminar recursión directa a izquierdas:

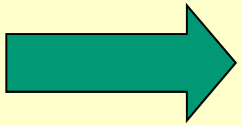
$$X ::= X\,\alpha \mid \beta \quad \Rightarrow \quad X ::= \beta\,RX \;\; RX ::= \alpha\,RX \mid \lambda$$

Siendo α y β formas sentenciales (= de terminales y no terminales) y cumpliéndose que β no comienza por X

Ejemplo de eliminación de recursión a izquierdas

- Aplicamos el patrón a la gramática de la izquierda, para quitar la recursión de E, considerando:
  - **E** hace las veces de X
  - **+ T** hace las veces de α
  - **T** hace las veces de β

```
E ::= E + T
E ::= T
T ::= T * F
T ::= F
F ::= n
F ::= ( E )
...
```

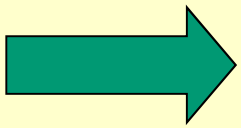


```
E ::= T RE
RE ::= + T RE
RE ::= λ
T ::= T * F
T ::= F
F ::= n
F ::= ( E )
...
```

Ejemplo de eliminación de recursión a izquierdas

- Volvemos a aplicar el patrón a la nueva gramática, esta vez para quitar la recursión de T, considerando:
  - **T** hace las veces de X
  - *** F** hace las veces de α
  - **F** hace las veces de β

```
E ::= T RE
RE ::= + T RE
RE ::= λ
T ::= T * F
T ::= F
F ::= n
F ::= ( E )
...
```



```
E ::= T RE
RE ::= + T RE
RE ::= λ
T ::= F RT
RT ::= * F RT
RT ::= λ
F ::= n
F ::= ( E )
...
```

- Y así seguiríamos hasta quitar toda la recursión a izquierdas que haya en la gramática incontextual…

Ejemplo de implementación de un analizador sintáctico

- Una vez tenemos la gramática incontextual de antes con el axioma cambiado y la recursión a izquierdas eliminada:

```
EFin ::= E fin
E ::= T RE
RE ::= + T RE
RE ::= λ
T ::= F RT
RT ::= * F RT
RT ::= λ
F ::= n
F ::= ( E )
```

Ejemplo de implementación de un analizador sintáctico

- Podemos implementar sin recursiones infinitas:

```
fun EFin()
  E;
  reconoce(fin);
ffun


fun E()
  T();
  RE();
ffun


fun RE()
  si token = Suma entonces
    reconoce(Suma);
    T();
    RE();
  fsi
ffun
```

Ejemplo de implementación de un analizador sintáctico

Continuación…

```
fun T()
  F();
  RT();
ffun


fun RT()
  si token = Mul entonces
    reconoce(Mul);
    F();
    RT();
  fsi
ffun


fun F()
  si token = Num entonces
    reconoce(Num)
  si no
    reconoce(PApertura);
    E();
    reconoce(PCierre);
  fsi
ffun
```

Acondicionamiento: Eliminación de recursión a izquierdas

- Para una **gramática de atributos s-atribuida** también hay una parte del patrón típico para eliminar recursión directa a izquierdas que explica cómo transformar las ecuaciones semánticas de los atributos sintetizados
  - Curiosamente es necesario añadir atributos heredados

$X ::= X\ \alpha$
$X_0.s = f(X_1.s, \alpha.s)$

$X ::= \beta$
$X_0.s = g(\beta.s)$

$X ::= \beta\ RX$
$RX.sh = g(\beta.s)$
$X.s = RX.s$

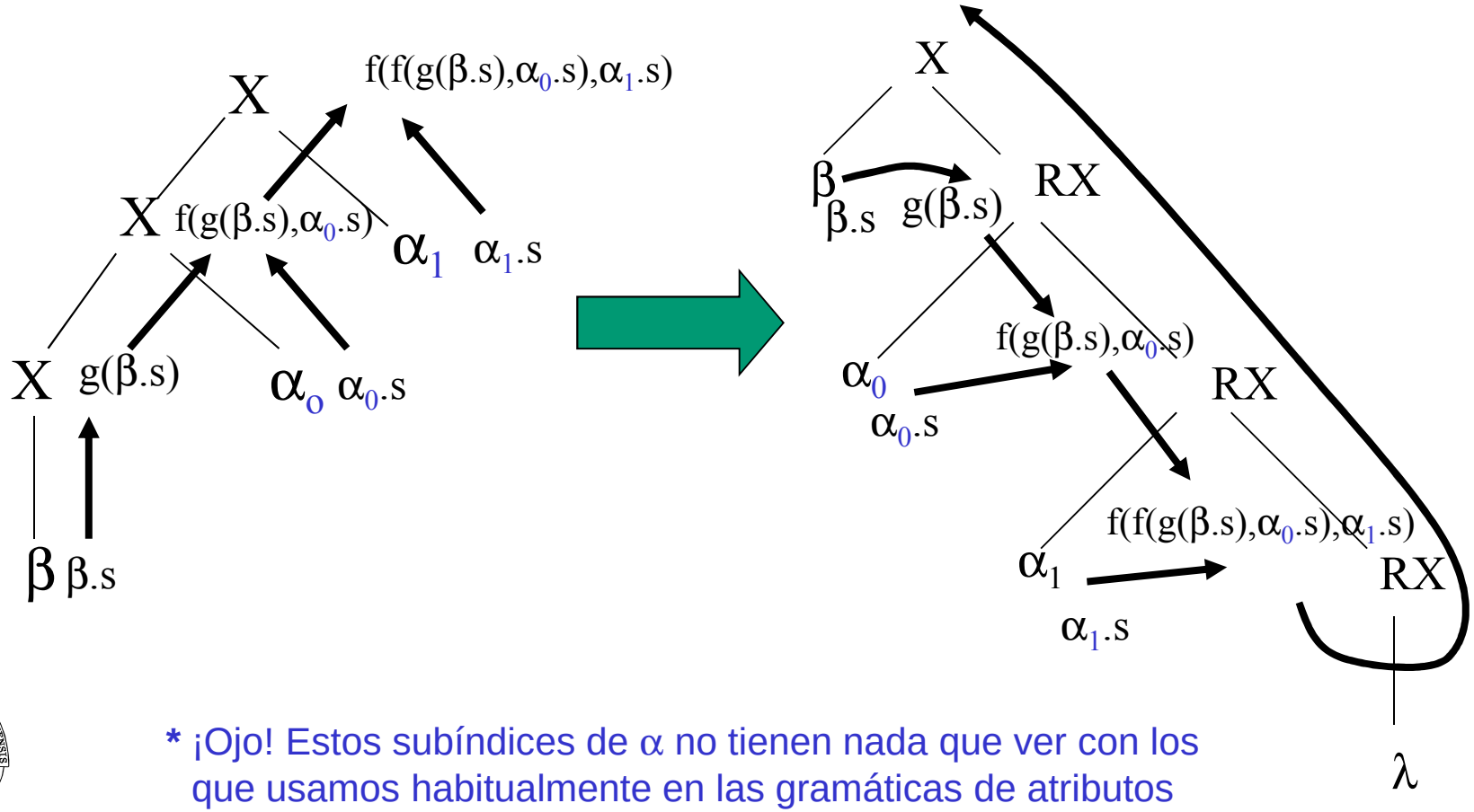
$RX ::= \alpha\ RX$
$RX_1.sh = f(RX_o.sh, \alpha.s)$
$RX_o.s = RX_1.s$

$RX ::= \lambda$
$RX.s = RX.sh$

α y β son formas senteciales y β no comienza por X
s es cualquier información sintetizada (conjunto de atributos)

Acondicionamiento: Eliminación de recursión a izquierdas

- Esto es lo que estamos haciendo al aplicar el patrón para eliminar recursión directa a izquierdas sobre los atributos:

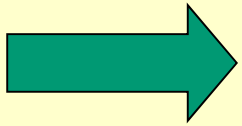


* ¡Ojo! Estos subíndices de α no tienen nada que ver con los que usamos habitualmente en las gramáticas de atributos

Ejemplo de eliminación de recursión a izquierdas

- Aplicamos el patrón a la gramática de la izquierda, para quitar la recursión de E, considerando:
  - **val** hace las veces del atributo sintetizado s
  - **x + y** hace las veces de la función f(x,y)
  - **x** hace las veces de la función g(x)

```
E ::= E + T
  E0.val = E1.val + T.val
E ::= T
  E.val = T.val
...
```



```
E ::= T RE
  RE.valh = T.val
  E.val = RE.val
RE ::= + T RE
  RE1.valh = RE0.valh + T.val
  RE0.val = RE1.val
RE ::=  λ
  RE.val = RE.valh
...
```

Ejemplo de eliminación de recursión a izquierdas

- Volvemos a aplicar el patrón a la nueva gramática, esta vez para quitar la recursión de T, considerando:
  - **val** hace las veces del atributo sintetizado s
  - **x + y** hace las veces de la función f(x,y)
  - **x** hace las veces de la función g(x)

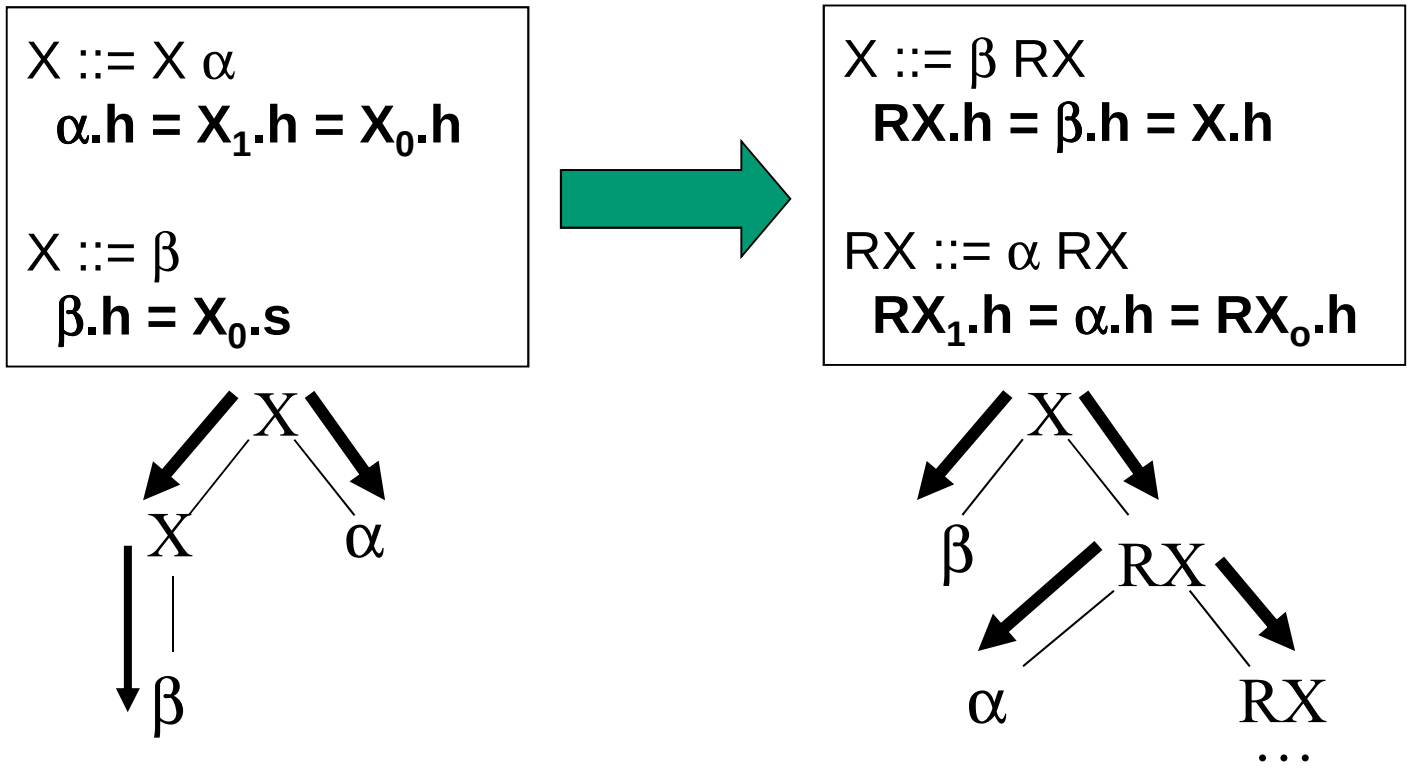
```
T ::= T * F
  T0.val = T1.val * F.val
T ::=  F
  T.val = F.val
...
```

```
T ::= F RT
  RT.valh = F.val
  T.val = RT.val
RT ::= * F RT
  RT1.valh = RT0.valh * F.val
  RT0.val = RT1.val
RT ::=  λ
  RT.val = RT.valh
...
```

- Y así seguiríamos hasta quitar toda la recursión a izquierdas que haya en la gramática de atributos…

Acondicionamiento: Eliminación de recursión a izquierdas

- Para transformar ecuaciones semánticas con atributos heredados hay que considerar cada caso…
  - Aunque también hay algún patrón trivial, como **transmitir un atributo heredado de un nodo a todos sus hijos** (por ejemplo, la tabla de símbolos):



Acondicionamiento: Factorización de las producciones

- **Problema**: Si varias producciones de un mismo *no terminal* empiezan con el mismo *terminal* o *no terminal*, según nuestra implementación, en el cuerpo del procedimiento correspondiente el *elemento de preanálisis* no nos ayuda a elegir cual aplicar:

```
E ::= - B
E ::= - C
```

***** Estando en el procedimiento **E** y siendo **-** el componente léxico de entrada ¿he de llamar al procedimiento **B** o al **C**?

- **Solución:** Añadir otra producción para "factorizar" (sacar el factor común de) las conflictivas
  - Patrón típico para factorizar:

$$X ::= \alpha\ \beta \quad X ::= \alpha\ \gamma \quad \Rightarrow \quad X ::= \alpha\ FX \quad FX ::= \beta \quad FX ::= \gamma$$

Siendo α, β y γ formas sentenciales

Ejemplo de factorización de las producciones

- Aplicamos el patrón a la gramática de la izquierda, para factorizar sus producciones, considerando:
  - **E** hace las veces de X
  - **T** hace las veces de α
  - **% E** hace las veces de β
  - **num** hace las veces de γ

```
E ::= T % E
E ::= T num
...
```

```
E ::= T FE
FE ::= % E
FE ::= num
...
```

Otro ejemplo de factorización de las producciones

- Aplicamos el patrón a la gramática de la izquierda, para factorizar sus producciones, considerando:
  - **C** hace las veces de X
  - **I** hace las veces de α
  - **; Is** hace las veces de β
  - λ hace las veces de γ

```
C ::= I ; Is
C ::= I
...
```

```
C ::= I FC
FC ::= ; Is
FC ::= λ
...
```

Acondicionamiento: Factorización de las producciones

- Para una **gramática de atributos s-atribuida** también hay una parte del patrón típico para factorizar las producciones que explica cómo transformar las ecuaciones semánticas de los atributos sintetizados
  - También es necesario añadir atributos heredados

$X ::= \alpha\ \beta$
$X.s = f(\alpha.s, \beta.s)$

$X ::= \alpha\ \gamma$
$X.s = g(\alpha.s, \gamma.s)$

$X ::= \alpha\ FX$
$FX.sh = \alpha.s$
$X.s = FX.s$

$FX ::= \beta$
$FX.s = f(FX.sh, \beta.s)$

$FX ::= \gamma$
$FX.s = g(FX.sh, \gamma.s)$

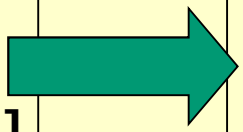
α, β y γ son formas sentenciales

s es cualquier información sintetizada (conjunto de atributos)

Ejemplo de factorización de las producciones

- Aplicamos el patrón a la gramática de la izquierda, para factorizar sus producciones, considerando:
  - **val** hace las veces del atributo sintetizado s
  - **x ^ y** hace las veces de la función f(x,y)
  - **x mod y** hace las veces de la función g(x,y)

```
E ::= T and E
  E0.val = T.val ^ E1.val
E ::= T % num
  E.val = T.val mod num.val
...
```



```
E ::= T FE
  E.val = FE.val
  FE.valh = T.val
FE ::= and E
  FE.val = FE.valh ^ E.val
FE ::= % num
  FE.val = FE.valh mod num.val
...
```

Acondicionamiento: Factorización de las producciones

- Para transformar ecuaciones semánticas con atributos heredados también hay que considerar cada caso…
  - Y hay algún patrón trivial, como **transmitir un atributo heredado de un nodo a todos sus hijos** (por ejemplo, la tabla de símbolos):

$X ::= \alpha\ \beta$
$\beta.h = \alpha.h = X.h$

$X ::= \alpha\ \gamma$
$\gamma.h = \alpha.h = X.h$

$X ::= \alpha\ FX$
$FX.h = \alpha.h = X.h$

$FX ::= \beta$
$\beta.h = FX.h$

$FX ::= \gamma$
$\gamma.h = FX.h$

*** Aviso**: A veces hacen falta las dos cosas: eliminar la recursión izquierda y factorizar… entonces conviene *factorizar primero* y *luego eliminar la recursión* (y acordarse de recolocar todas las ecuaciones semánticas por completo)

Esquemas de traducción

- Una vez *acondicionada* la gramática (cambiado el axioma, eliminada la recursión izquierda y factorizadas las producciones) ya podríamos implementar...

  … aunque falta saber cómo **implementar el cálculo del valor de los atributos semánticos** según las ecuaciones semánticas de la gramática de atributos

- Para implementar estos cálculos es muy habitual empezar haciendo unos **esquemas** llamados **de traducción** (porque típicamente los usamos para calcular, entre otros, el valor de **cod**, la traducción del programa fuente)
  - Aprovechando el recorrido implícito del árbol sintáctico que hace el *algoritmo descendente predictivo recursivo* (**preorden**) para evaluar en una sola pasada todas las ecuaciones semánticas de la gramática de atributos

Esquemas de traducción

- Estos esquemas son una notación intermedia en forma de *dos modificaciones sucesivas* a la gramática de atributos:
  1. Esquemas orientados a la gramática de atributos
     - Colocar las *ecuaciones semánticas* entre los símbolos de las producciones de la gramática, para indicar **cuando** se calculan los atributos semánticos (establecer un *orden de evaluación* en las ecuaciones semánticas)
  2. Esquemas orientados a la traducción
     - Convertir los símbolos de las producciones en **encabezados o llamadas a procedimientos** y las ecuaciones en **acciones semánticas** (*fragmentos de pseudocódigo describiendo la implementación*) para concretar **cómo** se calculan los atributos semánticos
- Existen *herramientas automáticas* que reciben esquemas de traducción y devuelven el código del traductor deseado
  - JavaCC lo hace con un algoritmo descendente y YACC con uno ascendente que estudiaremos más adelante

Ejemplo de esquemas de traducción

- Partimos de esta gramática de atributos y asumimos el uso del algoritmo descendente predictivo recursivo (es decir, recorrido en preorden del árbol sintáctico)…

```
Número ::= Dígito RNúmero
  Número.valor = RNúmero.valor +
         valorDe(Dígito.lex, Número.vdigs) * Número.peso
  Número.peso = RNúmero.peso * Número.base
  RNúmero.base = Número.base
  RNúmero.vdigs = Número.vdigs
RNúmero ::= Número
  RNúmero.valor = Número.valor
  RNúmero.peso =  Número.peso
  Número.base = RNúmero.base
  Número.vdigs = RNúmero.vdigs
RNúmero ::= λ
  RNúmero.peso = 1
  RNúmero.valor = 0
```

*** Aviso**: Recordad que **base** y **vdigs** son atributos heredados

Esquemas orientados a la gramática de atributos

1. Las ecuaciones semánticas se colocan dentro de las producciones según estas normas:
   - Las ecuaciones se agrupan y se escriben entre llaves **{ }**
   - Las ecuaciones que calculan el valor de los atributos heredados de los símbolos que están en la parte derecha de las producciones deben colocarse **antes** de dichos símbolos
   - Las ecuaciones no pueden hacer referencia a atributos sintetizados de símbolos que estén colocados **después** *(más adelante se estudiará cómo evitar esta restricción)*
   - Las ecuaciones que calculan el valor de los atributos sintetizados del *no terminal* de la izquierda de la producción se colocan **después** del cálculo de todos los atributos a los que referencian (normalmente **al final de la producción**)

Ejemplo de esquemas orientados a la gramática de atributos

1. Colocando las ecuaciones semánticas entre los símbolos adecuados de las producciones de la gramática se obtienen estos esquemas:

```
Número ::= Dígito
           {RNúmero.base = Número.base;
            RNúmero.vdigs = Número.vdigs}
           RNúmero
           {Número.peso = RNúmero.peso * Número.base;
            Número.valor = RNúmero.valor +
                   valorDe(Dígito.lex, Número.vdigs) *
                   Número.peso}
RNúmero ::= {Número.base = RNúmero.base;
             Número.vdigs = RNúmero.vdigs}
            Número
            {RNúmero.peso = Número.peso;
             RNúmero.valor = Número.valor}
RNúmero ::= λ
            {RNúmero.peso = 1;
             RNúmero.valor = 0}
```

Esquemas orientados a la traducción

2. Los símbolos de las producciones se convierten en encabezados o llamadas a procedimientos, y las ecuaciones en acciones semánticas (*pseudocódigo del traductor*) según estas normas:
   - Los *no terminales* de la izquierda de las producciones serán los **encabezados** de los procedimientos y las demás apariciones serán **llamadas** a dichos procedimientos
   - Los atributos se convierten en variables de los procedimientos correspondientes a cada *no terminal*
     - Parámetros **de entrada** si son *heredados* y **de salida** si son *sintetizados* (tenerlo en cuenta tanto en los encabezados como en las llamadas a procedimientos)
     - **Variables locales** si son atributos que deban ser inicializados o evaluados
   - Pueden declararse *otras variables locales* (auxiliares) en cada producción.
   - Pueden definirse *otros procedimientos/funciones* (auxiliares) para toda la implementación

Ejemplo de esquemas orientados a la traducción

2. Convirtiendo símbolos en procedimientos y ecuaciones en acciones semánticas se obtienen estos esquemas:

```
Número(in base1,vdigs1; out peso1,valor1) ::= Dígito(out lex)
                    {base2 ← base1;
                     vdigs2 ← vdigs1}
                    RNúmero(in base2,vdigs2; out peso2,valor2)
                    {peso1 ← peso2*base1;
                     valor1 ← valor2+peso1*valorDe(lex,vdigs1)}
RNúmero(in base1,vdigs1; out peso1,valor1) ::=
                    {base2 ← base1;
                     vdigs2 ← vdigs1}
                    Número(in base2,vdigs2; out peso2,valor2)
                    {peso1 ← peso2;
                     valor1 ← valor2}
RNúmero(in base,vdigs; out peso,valor) ::=
                    {peso ← 1;
                     valor ← 0}
```

***** Los atributos ahora son **parámetros** de los procedimientos correspondientes a cada producción y para distinguir los que se llaman igual, se numeran *en local*

*** x ← y** significa *asignar* a x el valor de y

Esquemas orientados a la traducción (Optimizaciones)

2. … y si el entorno de implementación escogido lo permite pueden hacerse algunas **optimizaciones** para que nuestra implementación quede más clara y concisa:
   - Hacer que varios atributos hagan **referencia a la misma estructura de datos**, para ahorrar espacio en memoria (por ejemplo: **ts** y **tsh**)
   - Hacer que algunos atributos sean **variables globales**, para ahorrarnos su propagación explícita por todas partes (por ejemplo: **cod** o cualquier atributo heredado que se transmita a todos los subárboles de un nodo, como **ts**/**tsh**)
   - En general, cualquier otra **simplificación** que ahorre variables y sentencias de asignación innecesarias *(dependientes del entorno y el lenguaje de implementación)*

Ejemplo de esquemas orientados a la traducción

2. … y finalmente optimizando (en este caso con variables globales), conseguimos los esquemas definitivos:

```
global base,vdigs;

Número(out peso1,valor1) ::= Dígito(out lex)
                RNúmero(out peso2,valor2)
                {peso1 ← peso2*base;
                 valor1 ← valor2+peso1*valorDe(lex,vdigs)}
RNúmero(out peso1,valor1) ::= Número(out peso2,valor2)
                {peso1 ← peso2;
                 valor1 ← valor2}
RNúmero(out peso,valor) ::=
                {peso ← 1;
                 valor ← 0}
```

Ejemplo de implementación de un analizador/traductor

- Partimos de una gramática de atributos como esta…

```
...
Exp ::= Exp OpAd Term

  // Propagación de la tabla de símbolos
  Exp1.tsh = Term.tsh = Expo.tsh

  // Gestión de errores (restricciones contextuales)
  Expo.err = Exp1.err ∨ Term.err

  // Generación de código (traducción)
  Expo.cod = Exp1.cod || Term.cod || OpAd.op
...
```

Ejemplo de implementación de un analizador/traductor

- Acondicionamos la gramática (en este ejemplo no hace falta factorizar, sólo eliminar la recursión a izquierdas)…

```
...
Exp ::= Term RExp
  Exp.tsh = Term.tsh = RExp.tsh
  RExp.errh = Term.err
  Exp.err = Rexp.err
  RExp.codh =  Term.cod
  Exp.cod = RExp.cod
RExp ::= OpAd Term RExp
  RExp_o.tsh = Term.tsh = RExp_1.tsh
  RExp_1.errh = RExp_o.errh ∨ Term.err
  RExp_o.err = RExp_1.err
  RExp_1.codh = RExp_o.codh || Term.cod || OpAd.op
  RExp_o.cod = RExp_1.cod
RExp ::= λ
  RExp.err = RExp.errh
  RExp.cod = RExp.codh
...
```

Ejemplo de implementación de un analizador/traductor

1. Hacemos el esquema orientado a la gramática de atributos…

```
...
Exp ::= {Term.tsh = Exp.tsh}
        Term
        {RExp.tsh = Exp.tsh;
         RExp.errh = Term.err;
         RExp.codh = Term.cod}
        RExp
        {Exp.err = RExp.err;
         Exp.cod = RExp.cod}
RExp ::= OpAd
         {Term.tsh = RExp_o.tsh}
         Term
         {RExp_1.tsh = RExp_o.tsh;
          RExp_1.errh = RExp_o.errh ∨ Term.err;
          RExp_1.codh = RExp_o.codh || Term.cod || OpAd.op}
         RExp
         {RExp_o.err = RExp_1.err;
          RExp_o.cod = RExp_1.cod}
RExp ::= λ
         {RExp.err = RExp.errh;
          RExp.cod = RExp.codh}
...
```

Ejemplo de implementación de un analizador/traductor

2. Hacemos el esquema orientado a la traducción…

```
...
Exp(in tsh0; out err0,cod0) ::= {tsh1 ← tsh0}
                        Term(in tsh1; out err1,cod1)
                        {tsh2 ← tsh0; errh2 ← err1; codh2 ← cod1}
                        RExp(in tsh2,errh2,codh2; out err2,cod2)
                        {err0 ← err2; cod0 ← cod2}
RExp(in tsh0,errh0,codh0; out err0,cod0) ::= OpAd(out op)
                        {tsh1 ← tsh0}
                        Term(in tsh1; out err1,cod1)
                        {tsh2 ← tsh0; errh2 ← errh0 ∨ err1;
                         codh2 ← codh0 || cod1 || op}
                        RExp(in tsh2,errh2,codh2; out err2,cod2)
                        {err0 ← err2; cod0 ← cod2}
RExp(in tsh,errh,codh; out err,cod) ::= λ
                        {err ← errh;cod ←codh}
...
```

Ejemplo de implementación de un analizador/traductor

2. … y las optimizaciones que consideremos convenientes (mejor es tener la *tabla de símbolos* y el *código traducido* en global), ¡y ya tenemos los esquemas de traducción definitivos!

```
global ts,cod;

...
Exp(out err0) ::= Term(out err1)
                  {errh2 ← err1}
                  RExp(in errh2; out err2)
                  {err0 ← err2}
RExp(in errh0; out err0) ::= OpAd(out op)
                             Term(out err1)
                             {errh2 ← errh0 ∨ err1;
                              emite(op)}
                             RExp(in errh2; out err2)
                             {err0 ← err2}
RExp(in errh; out err) ::= λ
                           {err ← errh}
...
```

*** emite()** es una función que podemos inventarnos para que se encargue de ir añadiendo *líneas concretas de código* a la variable global **cod**

Implementación de un analizador/traductor

- Implementar el analizador/traductor partiendo de los esquemas orientados a la traducción optimizados es bastante obvio
- Y después se pueden seguir haciendo simplificaciones y mejoras *directamente sobre el código final*
  - Convirtiendo llamadas recursivas en bucles
  - Fusionando varios procedimientos en uno
  - …

Ejemplo de implementación de un un analizador/traductor

- El pseudocódigo correspondiente a los esquemas de traducción (aún no optimizados) del ejemplo anterior:

```
...

fun Exp(in tsh0; out err0,cod0)
  var tsh1,codh1,err1,cod1,tsh2,errh2,codh2,err2,cod2;
  tsh1 ← tsh0;
  Term(tsh1, err1, cod1);
  tsh2 ← tsh1;
  errh2 ← err1;
  codh2 ← cod1;
  RExp(tsh2, errh2, codh2, err2, cod2)
  err0 ← err2;
  cod0 ← cod2
ffun
```

Ejemplo de implementación de un un analizador/traductor

Continuación…

```
fun RExp(in tsh0,errh0,codh0; out err0,cod0)
  si token ∈ {Suma,Resta} entonces
    var tsh1,codh1,err1,cod1,tsh2,errh2,codh2,err2,cod2,op;
    OpAd(op);
    tsh1 ← tsh0;
    Term(tsh1, err1, cod1);
    tsh2 ← tsh0;
    errh2 ← errh0 ∨ err1;
    codh2 ← codh0 || cod1 || op;
    RExp(tsh2, errh2, codh2, err2, cod2)
    err0 ← err2;
    cod0 ← cod2
  si no
    err0 ← errh0;
    cod0 ←codh0
  fsi
ffun

...
```

Ejemplo de implementación de un un analizador/traductor

- Unificamos variables con nombres distintos pero que en realidad son la misma, y ahorramos variables metiendo directamente expresiones en las llamadas a función:

```
...

fun Exp(in tsh0; out err0,cod0)
  var err1,cod1;
  Term(tsh0, err1, cod1);
  RExp(tsh0, err1, cod1, err0, cod0)
ffun
```

Ejemplo de implementación de un un analizador/traductor

Continuación…

```
fun RExp(in tsh0,errh0,codh0; out err0,cod0)
  si token ∈ {Suma,Resta} entonces
    var err1,cod1,op;
    OpAd(op);
    Term(tsh0, err1, cod1);
    RExp(tsh0, errh0 ∨ err1, codh0||cod1||op, err0, cod0)
  si no
    err0 ← errh0;
    cod0 ←codh0
  fsi
ffun

...
```

Ejemplo de implementación de un un analizador/traductor

- Tenemos en cuenta las optimizaciones de los esquemas de traducción, como el uso de variables globales o la función auxiliar `emite()` para ir generando código:

```
global ts; cod;

...

fun Exp(out err0)
  var err1;
  Term(err1);
  RExp(err1, err0)
ffun
```

Ejemplo de implementación de un un analizador/traductor

Continuación…

```
fun RExp(in errh0; out err0)
  si token ∈ {Suma,Resta} entonces
    var err1,op;
    OpAd(op);
    Term(err1);
    emite(op);
    RExp(errh0 ∨ err1, err0)
  si no
    err0 ← errh0
  fsi
ffun

...
```

Ejemplo de implementación de un un analizador/traductor

- Convertimos procedimientos recursivos a iterativos (allí donde lo podamos hacer):

```
global ts; cod;

...

fun Exp(out err0)
  var err1;
  Term(err1);
  RExp(err1, err0)
Ffun
```

Ejemplo de implementación de un un analizador/traductor

Continuación…

```
fun RExp(in errh0; out err0)
  mientras token ∈ {Suma,Resta}
    var err1,op;
    OpAd(op);
    Term(err1);
    emite(op);
    errh0 ← errh0 ∨ err1
  fmientras
  err0 ← errh0
ffun

...
```

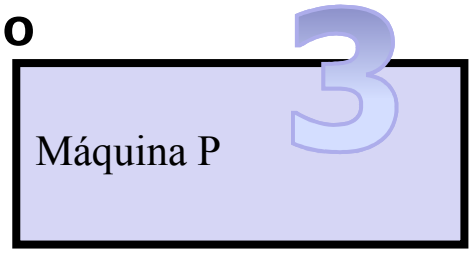
Ejemplo de implementación de un un analizador/traductor

- Fusionamos procedimientos similares y los volvemos a simplificar (por ejemplo: renombrando variables)…
  … ¡y el *código final* quedará más conciso y elegante!

```
global ts; cod;

...

fun Exp(out err)
  Term(err);
  mientras token ∈ {Suma,Resta}
     var errTerm,op;
     OpAd(op);
     Term(errTerm);
     emite(op);
     err ← err ∨ errTerm
  fmientras
ffun

...
```

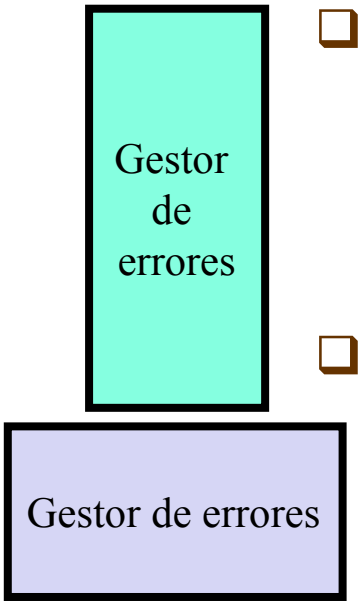
Implementación de la máquina virtual

- La máquina virtual es un **intérprete sencillo**
  - Se implementa su **arquitectura** y las **estructuras de datos** que mantienen el estado, los distintos registros y la memoria
  - Tendrá un **analizador léxico**, pero la sintaxis del código máquina suele tener una estructura trivial por lo que se puede analizará e interpretará *al vuelo* (línea a línea)



- La ejecución es un bucle en cuyo cuerpo **se ejecuta la instrucción leída** y luego se discrimina la **transición de estado a aplicar** según la instrucción leída
- Cuando la máquina virtual es un programa independiente del compilador, el programa en código máquina suele estar *serializado* en un **fichero** (idealmente mediante **cadenas de bytes** que codifican *instrucción* y *operandos*)

Gestión de errores



- La estrategia de gestión más sencilla al encontrar un error es **interrumpir la ejecución** y **mostrar un mensaje** al usuario con **información significativa sobre el error**: dónde ocurre, qué elementos están implicados, etc.
  (más adelante se estudiarán estrategias más avanzadas)
- Ejemplos
  - **Error léxico:** "El lexema *X* no se reconoce como un componente léxico del lenguaje" (tanto el analizador léxico del compilador como el de la máquina virtual no interrumpen la ejecución: pueden mostrar todos los errores de golpe)
  - **Error sintáctico incontextual:** "Encontrado el componente léxico *X* cuando se esperaba el *Y*" (tanto en el analizador sintáctico del compilador como en la máquina virtual)
  - **Error sintáctico contextual (violación de una restricción contextual):** "La variable *X* no ha sido declarada"
    (no se interrumpe la ejecución del analizador sintáctico del compilador, pero sí se *inhibe* la generación de código)
  - **Error de ejecución:** "División por cero" (en la máquina virtual)

Gestión de errores

*** Consejo:** Cuando se usan los dos atributos **err** y **tipo** en una gramática, en las expresiones basta con usar únicamente **tipo** (y en caso de error asignarle el valor **tipoError**)

```
global ts; cod;
...
fun Exp(out tipo)
  var tipo1,tipo2;
  Term(tipo1);
  mientras token ∈ {Suma,Resta}
     var tipo,op;
     OpAd(op);
     Term(tipo2);
     tipo ← tipoExpAd(tipo1,tipo2)
     emite(op);
  fmientras
ffun
...
```

```
fun tipoExpAd(in tipo1,tipo2)
  si tipo1=tipo2=tipoNumérico entonces
    devuelve tipoNumérico
  sino
    errorContextual(ExPad,tipo1,tipo2);
    devuelve tipoError
  fsi
ffun
```

- La información de error podría enriquecerse **mostrando una "traza"** de todos los *contextos afectados* desde que se detecta el error original en un punto del programa fuente (o **guardándola** en un atributo **err** más sofisticado)

Críticas, dudas, sugerencias...

Federico Peinado
www.federicopeinado.es