

Especificación de un Lenguaje de Programación Mínimo y Construcción de su Procesador (7)

Procesadores de Lenguaje

Construcción del procesador

Especificación formal de la traducción

1. (cont.) Especificación de un lenguaje de programación mínimo y construcción de su procesador

- Lenguaje objeto y máquina virtual.
- Especificación formal de la traducción.
 - Gramáticas de atributos y semántica operacional
- **Construcción del procesador.**
 - Analizadores recursivos descendentes y procesadores de un solo paso.
 - Transformaciones en las gramáticas de atributos.
 - Esquemas de traducción.
- **Fases de un compilador.**
- Reingeniería de dos ejemplos de procesadores de lenguajes sencillos.

Implementación

- Una vez que se han definido los distintos aspectos del procesador, debe abordarse su implementación.
- Para ello:
 - Debe seleccionarse un lenguaje de implementación concreto.
 - Debe *traducirse* la especificación del procesador a un programa en dicho lenguaje.
- La complejidad de la implementación dependerá del nivel y de la especificidad del lenguaje de implementación.
 - En el mejor de los casos, el procesador podrá generarse automáticamente a partir de la especificación (*realizando algunas adaptaciones, la especificación puede ser la entrada a una herramienta de generación de procesadores de lenguajes*).
 - En otros casos, la especificación deberá traducirse a un programa manualmente. Para facilitar dicha traducción se seguirá una metodología, dependiente del lenguaje y entorno de implementación.

3

Implementación

- Introduciremos una metodología para construir de forma manual el procesador:
 - Tomaremos como base un *algoritmo de análisis sintáctico* conocido como *análisis descendente predictivo recursivo*. En esta metodología (como en otras) el analizador sintáctico dirige el proceso de traducción.
- Esta metodología incluye las siguientes etapas :
 - Diseño y construcción del analizador léxico mediante expresiones regulares y *autómatas finitos* (*en ciertos entornos mediante gramáticas regulares*).
 - Eliminar la recursión izquierda directa si la gramática tiene producciones de este tipo, trasformando también las ecuaciones semánticas correspondientes, de forma que se preserve el lenguaje y su semántica.
 - Formular los esquemas de traducción (*los esquemas de traducción no son exclusivos de esta metodología*).
 - Implementar el procesador a partir de los esquemas de traducción (incluyendo el resto de los elementos del procesador)

4

Implementación Construcción de Analizadores Léxicos

- La función de un analizador léxico es segmentar el programa de entrada en una secuencia de *tokens o elementos del lenguaje fuente*:
 - convertir la secuencia de códigos carácter (alfabeto) en la entrada, en la secuencia de símbolos terminales -tokens- del vocabulario utilizado por la gramática incontextual que define el lenguaje fuente.
- Cada elemento de dicha secuencia incluye:
 - El token, es decir, el código del token.
 - Atributos del token (p.e. lexema, valor, dirección, etc.).
- El analizador léxico puede construirse sistemáticamente a partir de la definición léxica del lenguaje. Para ello:
 - Se traduce la definición léxica del lenguaje a un autómata finito determinista.
 - Se implementa el analizador léxico tomando como base dicho autómata.

5

Implementación Construcción de Analizadores Léxicos Obtención del AFD

- Objetivo: obtener un autómata finito (determinista) que reconozca los tokens del lenguaje.
- Dicho autómata se obtiene:
 - Formulando la expresión regular (ER) que resulta de la disyunción de las ERs asociadas con las categorías léxicas ...
 - ... añadiendo una ER para tratar los separadores, comentarios, etc. ...
 - ... y aplicando algoritmos de la teoría de lenguajes formales para obtener un AFD mínimo equivalente a la ER.
- Este proceso puede realizarse automáticamente (herramientas de generación de analizadores léxicos, como LEX)...
- ... o manualmente. En este caso puede ser más conveniente obtener un AF para cada categoría léxica, unir los estados iniciales de dichos AFs, y retocar, si fuera necesario, el AF resultante para transformarlo en un AFD.

6

Implementación

Construcción de Analizadores Léxicos

Obtención del AFD

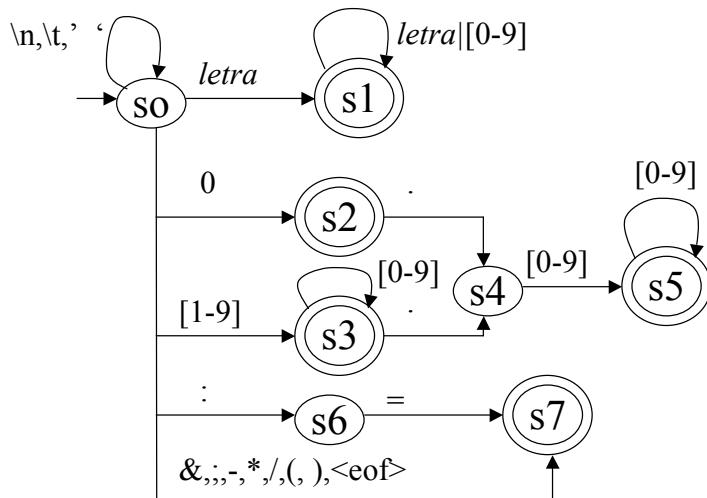
- Un posible método para obtener el AFD mínimo equivalente a una ER:
 - Se obtiene un *autómata finito no determinista* (AFN) equivalente a dicha expresión (*construcción de Thompson*).
 - Se obtiene un AFD equivalente al AFN. Para ello se aplica un algoritmo (*construcción de subconjuntos*).
 - Se obtiene un AFD mínimo equivalente al anterior, fusionando los estados equivalentes y eliminando los estados inaccesibles.
- Los generadores de analizadores léxicos se basan en optimizaciones de este método.

7

Implementación

Construcción de Analizadores Léxicos

Obtención del AFD



8

Implementación

Construcción de Analizadores Léxicos

- La implementación se guía por el AFD.
- Múltiples estrategias (representación tabular del AFD, traducción del AFD a código,...).
- Una estrategia de implementación manual usual es codificar la función de transición del AFD mediante:
 - Un CASE sobre los estados.
 - En cada opción del CASE se introducen las acciones de transición.
 - Un *buffer* de entrada con el siguiente carácter a procesar.
 - Tener en cuenta las posibles acciones asociadas al estado final y a las transiciones
 - Analizador léxico/sintáctico <-> Productor/consumidor

9

Implementación del analizador léxico: Ejemplo

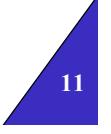
```
PROCEDURE Scan;
  TYPE
    TEstado = (S0,S1,S2,S3,S4,S5,S6,S7);
  ...
  (* Operaciones basicas *)
PROCEDURE Inicia; (* Inicia el AFD *) ...
PROCEDURE Transita(a: TEstado); (* Realiza una transicion *)
...
PROCEDURE Descarta; (* Descarta la cadena reconocida *) ...
PROCEDURE Acepta(token: TToken); (* Acepta una cadena *) ...

  (* Acciones de transicion *)
PROCEDURE AInicio; ....
PROCEDURE ARestoIden; ...
PROCEDURE ADecimal; ...
PROCEDURE AInicioParteDecimal; ...
PROCEDURE ARestoParteDecimal; ...
PROCEDURE ARestoAsig; ...
  (* Continua *)
```

10

Implementación del analizador léxico: Ejemplo

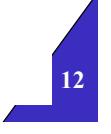
```
(* Cuerpo del analizador *)
BEGIN
    Inicia;
    WHILE NOT Fin DO
        CASE estado OF
            S0:      AInicio;
            S1:      ARestoIden;
            S2,S3:   ADecimal;
            S4:      AInicioParteDecimal;
            S5:      ARestoParteDecimal;
            S6:      ARestoAsig;
        END
    END;
```



11

Implementación del analizador léxico: Ejemplo

```
PROCEDURE AInicio;
BEGIN
    CASE info.token.sigCar OF (* info es una variable global
                                con información sobre el estado del traductor -
                                ficheros de entrada y errores, el token, etc... *)
        'a'..'z','A'..'Z': Transita(S1);
        '0':                 Transita(S2);
        '1' .. '9':          Transita(S3);
        ':' :                Transita(S6);
        '&':                BEGIN Transita(S7); Acepta(TKSEP) END;
        ....
        FL,TAB,' ':          BEGIN Transita(S0); Descarta; END;
        FF :                 Acepta(TKFF);
    ELSE BEGIN
        ErrorLexico(info,ECDESCONOCIDO);
        Transita(S0);
        Descarta
    END;
END;
END; (* TKSEP, TKPYCOMA, ... son valores enumerados que representan las
       categorias lexicas *)
```



12

Implementación

Introducción a los analizadores sintácticos descendentes-predictivos-recursivos

- Objetivo: Dada una gramática incontextual, construir un programa que decida si una frase pertenece o no al lenguaje generado por la gramática (*posteriormente el resto de los procesos y en particular el de traducción, se acoplarán con este proceso de análisis*).
- Dicho programa se denomina *analizador sintáctico (parser)*.
- Un método simple de construir analizadores sintácticos "descendentes predictivos recursivos" de forma manual es:
 - Interpretar los no terminales como procedimientos.
 - Interpretar las producciones asociadas con dichos no terminales como las definiciones de los procedimientos.
- Es un método de *análisis descendente*, puesto que simula la *expansión* del árbol de análisis sintáctico, comenzando por el axioma.
- Es una forma de implementar el autómata a pila que reconoce el lenguaje incontextual definido por la gramática.
- ver *Aho et al. 2.4 (se puede ampliar en 4.4)*

13

Implementación

Introducción a los analizadores sintácticos descendentes predictivos recursivos

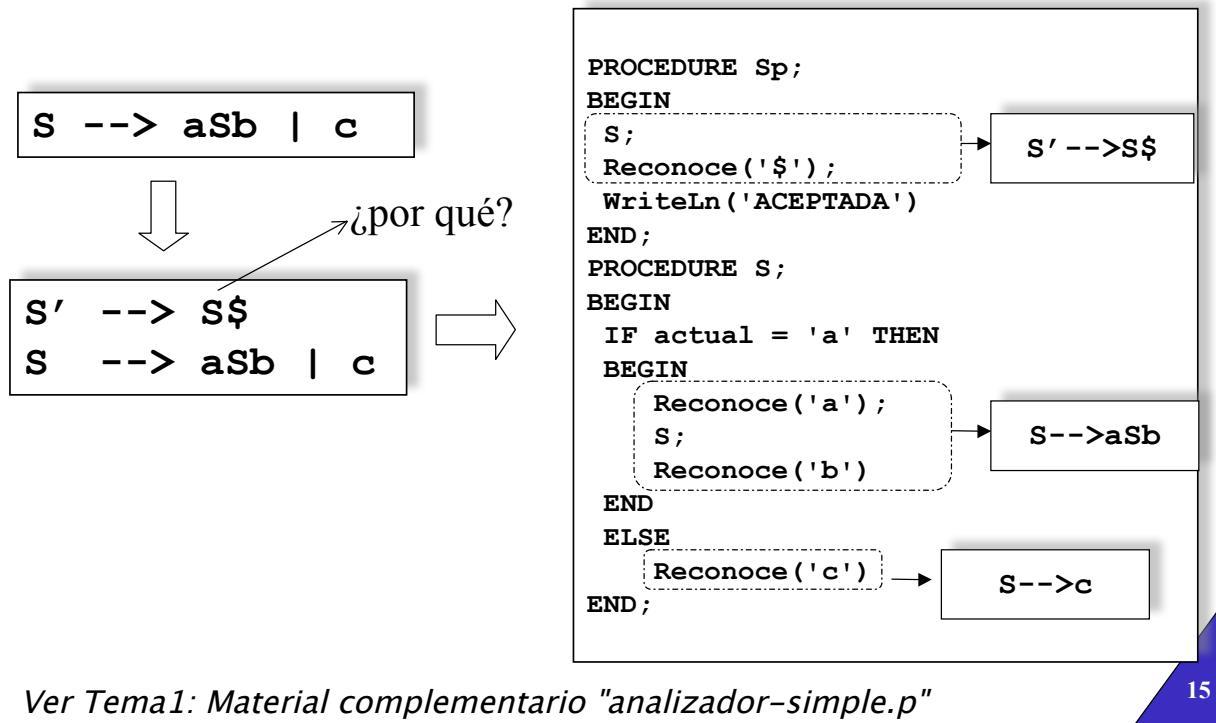
- Los no terminales en el lado derecho de las producciones se traducen en llamadas a los procedimientos correspondientes.
- Los terminales se traducen en un *consumo* del símbolo (*token*) de la frase y *obtención del siguiente*, o bien en la detección de que la frase no pertenece al lenguaje generado por la gramática.
- Cuando un no terminal tiene asociado más de una producción el carácter predictivo del analizador debe permitir decidir de forma determinista cuál aplicar:
 - Podría utilizarse una estrategia de retroceso (*backtracking*)...
 - ... pero es más eficiente tomar una decisión en base al símbolo (*token*) actual en la entrada (símbolo de *preanálisis*) => analizadores predictivos.
 - Existe un método sistemático para hacer esto, que funciona sobre una clase relativamente general de gramáticas. La teoría se estudiará en el tema 4.

14

Implementación

Introducción a los analizadores sintácticos

descendentes predictivos recursivos



Ver Tema1: Material complementario "anализатор-simple.p"

15

Implementación

Introducción a los analizadores sintácticos descendentes predictivos recursivos

- El resto del programa

```

PROGRAM EjemploDescendente;
VAR
  actual: Char;

PROCEDURE Reconoce(que: Char);
BEGIN
  IF actual = que THEN
    Read(actual)
  ELSE
    BEGIN
      writeln('NO ACEPTADA');
      Halt
    END;
END;
  
```

```

PROCEDURE S; forward;

PROCEDURE Sp; ...
PROCEDURE S; ...

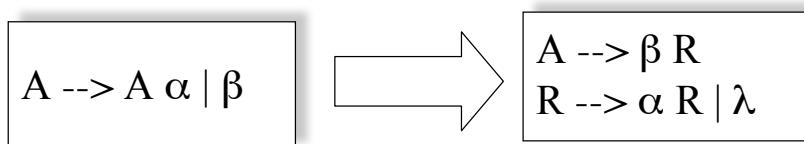
BEGIN
  Read(actual);
  Sp
END.
  
```

16

Implementación

Introducción a los analizadores sintácticos descendentes predictivos recursivos

- ¿Qué le pasaría a un analizador predictivo recursivo obtenido a partir de la siguiente gramática
 $S \rightarrow S a \mid a \quad ?$
- Los analizadores descendentes no funcionan con gramáticas con recursión a izquierdas (*recursión a izquierdas directa o indirecta*).
- Solución: transformar la gramática a una equivalente recursiva a derechas.
- Se considerará la siguiente transformación, útil en la mayoría de los casos prácticos:



(donde α y β son secuencias de terminales y no terminales y β no comienza por A).

17

Implementación

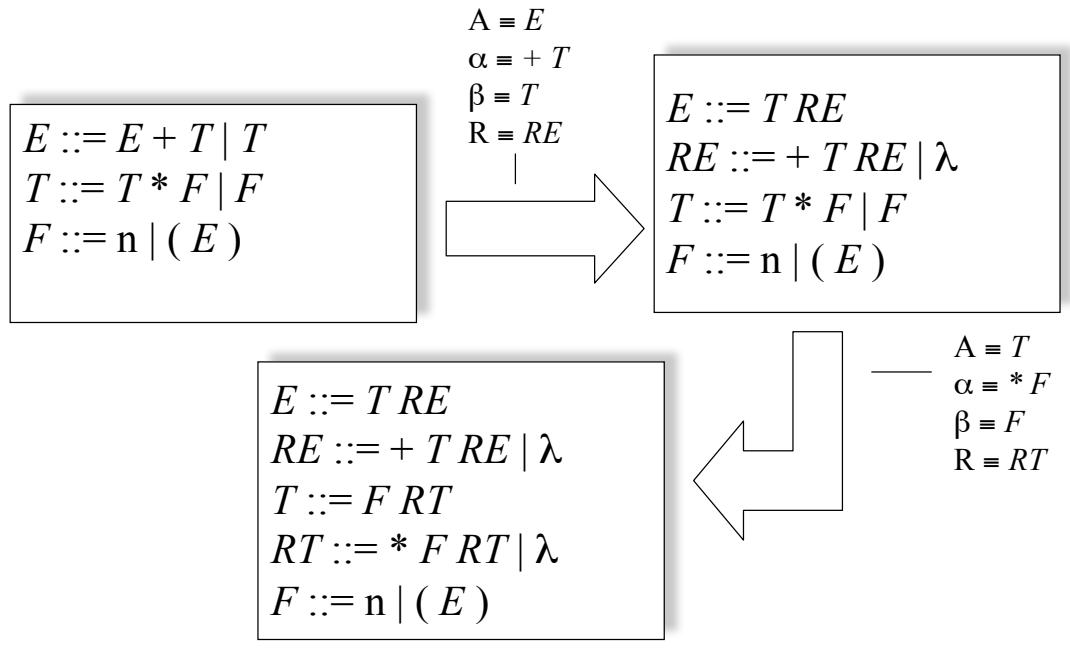
Introducción a los analizadores sintácticos descendentes predictivos recursivos

- Desde el punto de vista de los procesadores de lenguaje, ¿bastará con esta transformación de las producciones?
- ¡No! También será necesario transformar las ecuaciones semánticas, para preservar no sólo el lenguaje sino también su semántica.
- También se puede aplicar un algoritmo para eliminar la recursión izquierda indirecta (*consultar Aho et al.*) pero la dificultad que implica transformar en este caso las ecuaciones semánticas hace que sea muy poco utilizado.
- Luego se estudiará la transformación de las ecuaciones semánticas asociada a la eliminación de la recursión izquierda directa.

18

Implementación

Introducción a los analizadores sintácticos descendentes predictivos recursivos



19

Implementación

Introducción a los analizadores sintácticos descendentes predictivos recursivos

$E' ::= E <\text{eof}>$

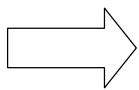
$E ::= T RE$

$RE ::= + T RE \mid \lambda$

$T ::= F RT$

$RT ::= * F RT \mid \lambda$

$F ::= n \mid (E)$



```

PROCEDURE Ep; BEGIN E; Rec(TKFF) END;
PROCEDURE E; BEGIN T; RE END;
PROCEDURE RE;
BEGIN
  IF info.token.token = TKMAS THEN
    BEGIN rec(TKMAS); T; RE END
END;
PROCEDURE T; BEGIN F; RT END;
PROCEDURE RT;
BEGIN
  IF info.token.token = TKPOR THEN
    BEGIN rec(TKPOR); F; RT END
END;
PROCEDURE F;
BEGIN
  IF info.token.token = TKNUM THEN
    rec(TKNUM)
  ELSE BEGIN Rec(TKPAP); E;
          Rec(TKPCIERRE) END
END;

```

20

Implementación

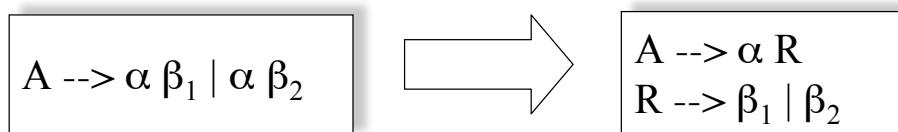
Introducción a los analizadores sintácticos descendentes predictivos recursivos

¿es posible construir un analizador predictivo recursivo para la siguiente gramática?

$S \rightarrow a \mid a S ?$

Las gramáticas que para una misma categoría sintáctica tienen producciones con un prefijo común no admiten un análisis descendente predictivo.

Este problema se puede solucionar factorizando las producciones: una transformación que retrasa la decisión sobre que producción emplear hasta que estas producciones presentan un subfijo diferente



Ver Tema1: Material complementario "analizador-expresiones.p"

21

Implementación: Traducción

Esquemas de Traducción

- La elección de un algoritmo de análisis sintáctico supone un recorrido implícito del árbol de análisis sintáctico.
- Por ejemplo, los analizadores descendentes predictivos recursivos recorren implícitamente el árbol de análisis sintáctico en preorden (de izquierda a derecha, y en profundidad) equivalente a realizar la derivación más a la izquierda.
- El recorrido que el analizador hace del árbol de análisis puede aprovecharse para evaluar *al vuelo*, en una sola pasada por el árbol, las ecuaciones semánticas, siempre y cuando pueda encontrarse un orden de evaluación de las ecuaciones semánticas que sea compatible con dicho recorrido.
- ¿Cuáles son las ventajas y cuáles los inconvenientes de esta aproximación de evaluación *al vuelo*?
- Para expresar la evaluación *al vuelo* de las ecuaciones semánticas se utilizará una notación intermedia que incorpora a la gramática las acciones semánticas como si fuesen símbolos del lenguaje, conocida como esquemas de traducción

22

Implementación: Traducción

Esquemas de Traducción

- Los **esquemas de traducción** expresan explícitamente *cómo y cuando* computar los valores de los atributos en cada producción.
- Para ello intercalan las *acciones semánticas* (como fragmentos de código escritos en un lenguaje de programación) entre los símbolos de dichas producciones.
- Existen herramientas que procesan automáticamente esquemas de traducción para generar traductores (p.e. YACC con una estrategia de análisis ascendente y las DCGs con una descendente).

```
E ->{$1.tsh:=$.ts_h}E + {$3.tsh:=$.tsh}T {$err:=$1.err OR $3.err}
E ->{$1.tsh:=$.tsh} T {$err:=$1.err}
T ->{$1.tsh:=$.tsh} T * {$3.tsh:=$.tsh}F {$err:=$1.err OR $3.err}
T ->{$1.tsh:=$.tsh} F {$err:=$1.err}
F -> n {$err:=false}
F -> id {$err:=NOT existeID($.tsh,$1.lex)}
F --> ( {$2.tsh:=$.tsh} E ) {$err:=$2.err}
```

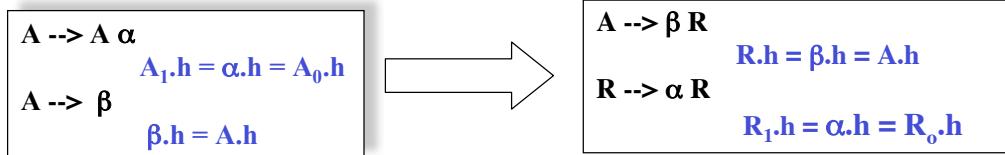
Especificación de la traducción basada en el algoritmo de análisis descendente predictivo recursivo, gramáticas de atributos y esquemas de traducción:

- Se parte de la(s) gramática(s) de atributos utilizada(s) en la definición del lenguaje:
 - La gramática incontextual utilizada para construir el analizador debe ser apropiada para el análisis descendente predictivo recursivo (ADPR)....
 - ... y *l-atribuida*. Una gramática de atributos se dice *l-atribuida* cuando, para cada producción $n \rightarrow X_1 X_2 \dots X_n$ los atributos heredados de cada símbolo X_j dependen sólo de:
 - Los atributos de los símbolos $X_1 \dots X_{j-1}$
 - Los atributos heredados de n .
 - Nota: Es posible tratar algunos casos de atributos heredados por la derecha.
 - Nota: Si sólo hay atributos sintetizados, la gramática se dice *s-atribuida*.
 - Si la gramática no es apropiada para el ADPR, p.e. incluye producciones con recursión izquierda directa (en principio la indirecta no la trataremos) deberemos eliminar estas producciones preservando el lenguaje y su semántica
- ...
será necesario además, *transformar las ecuaciones semánticas*.

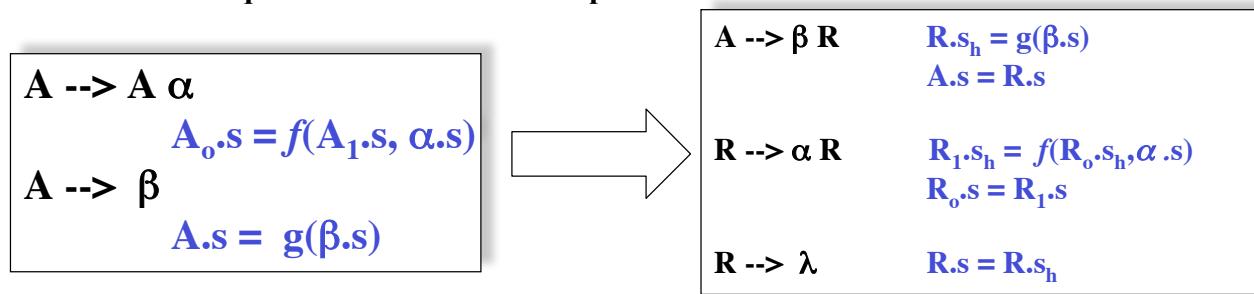
Implementación

Traducción Descendente

- Se estudiarán algunas transformaciones de gramáticas y de ecuaciones semánticas que preservan la semántica del lenguaje al eliminar recursividad a izquierdas.
- La propagación de un atributo heredado (ej: la tabla de símbolos) no presenta demasiados problemas.

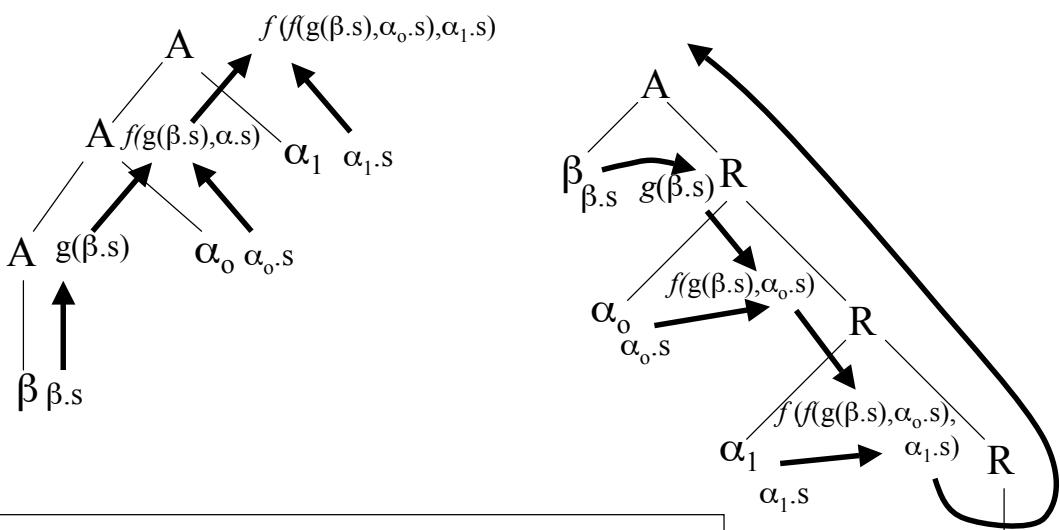


- El tratamiento de la síntesis es más complicado. Deben introducirse atributos heredados que *acumulen* los cálculos parciales realizados.



Implementación

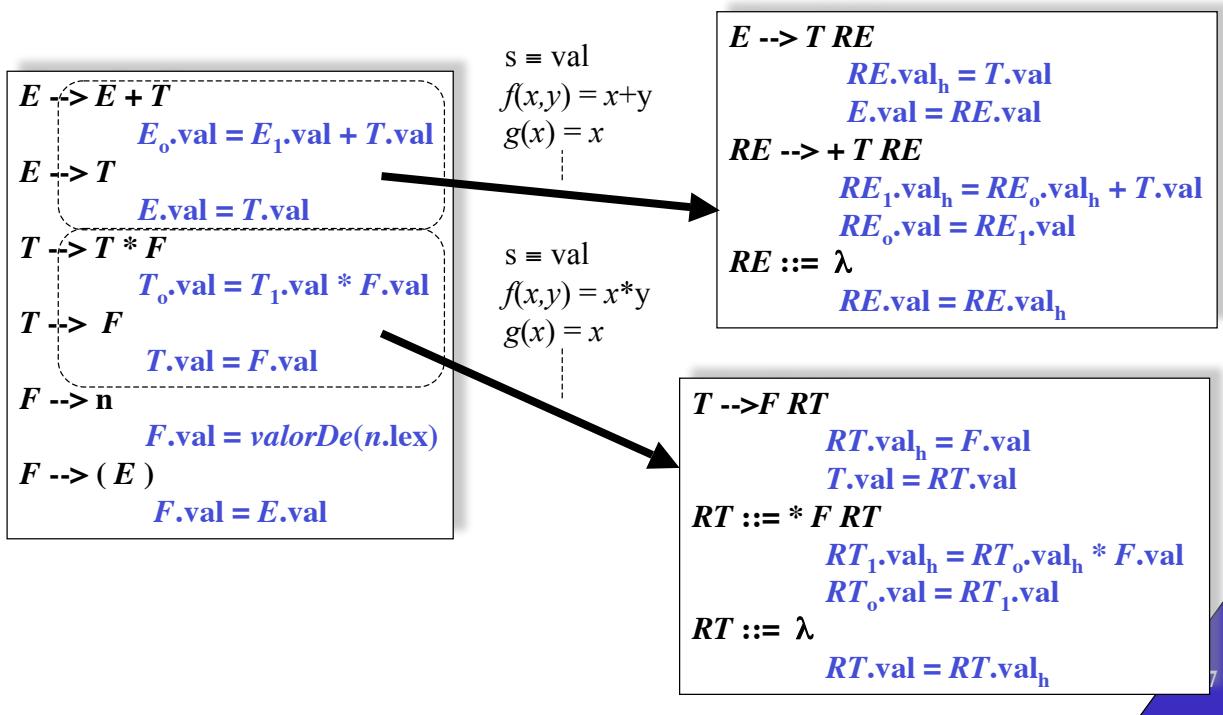
Traducción Descendente



¡ventaja! en el caso de definiciones s-atribuidas, la transformación a aplicar es siempre la misma

Implementación

Traducción Descendente



Ejercicio

3.1 Explica, haciendo un esquema de los árboles generados por las gramáticas transformada y sin transformar, el algoritmo de eliminación de la recursión a izquierdas directa, y de transformación de las ecuaciones semánticas de una gramática **s-atribuida**.

3.2 Aplica el algoritmo anterior para eliminar la recursión a izquierdas directa de la siguiente gramática de atributos:

$D \rightarrow id \quad D.p = buscarId(id.lex)$

$D.c = apila D.p->dir$

$D.t = D.p->tipo$

$D \rightarrow D . id \quad D_1.campo = buscarCampo(id.lex, D_2.t)$

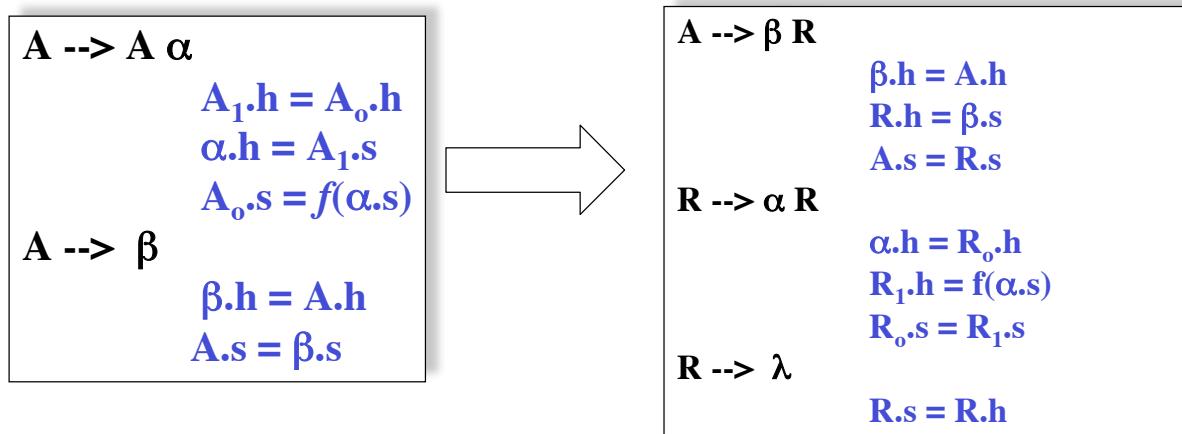
$D_1.c = D_2.c \parallel apila D_1.campo->offset \parallel suma$

$D_1.t = D_1.campo->tipo$

Implementación

Traducción Descendente

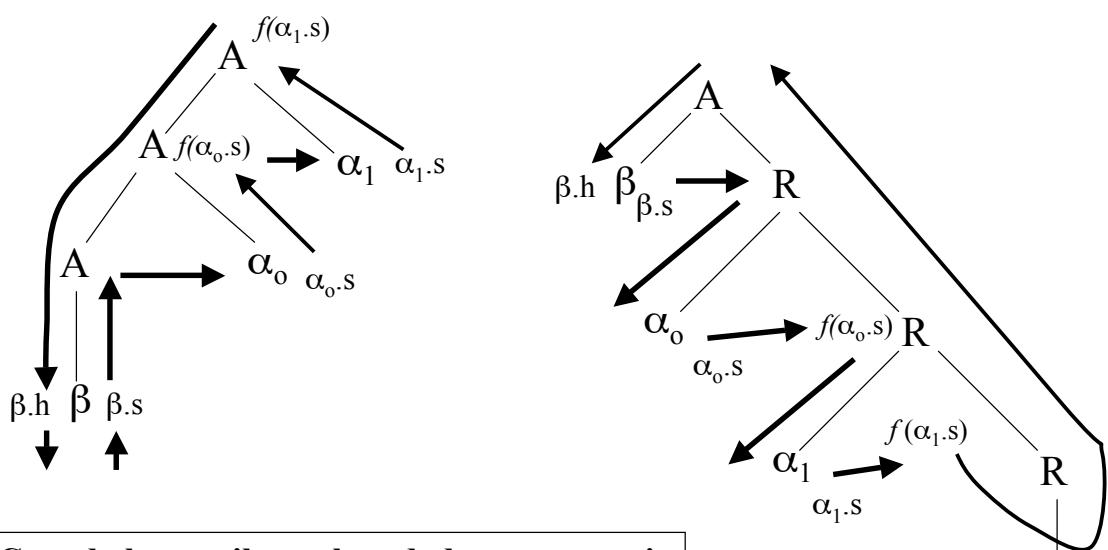
- Una situación diferente surge cuando la síntesis depende de un atributo heredado.



29

Implementación

Traducción Descendente

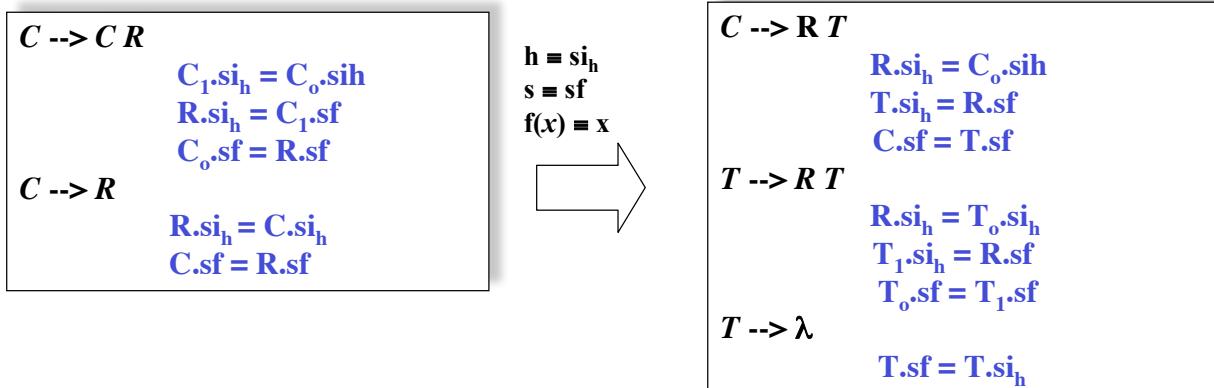


Cuando hay atributos heredados es necesario diseñar transformaciones específicas

30

Implementación

Traducción Descendente



expresión regular (concatenación de símbolos) a autómata finito utilizando como estado inicial el estado final anterior

31

Implementación

Traducción Descendente

Como en el caso de la eliminación de la recursión a izquierdas, la factorización de las producciones requiere preservar la semántica del lenguaje y transformar de forma adecuada las ecuaciones semánticas.

Las técnicas a utilizar son similares a las vistas para la eliminación de la recursión izquierda.

Ejercicio:

Factorizar y transformar las ecuaciones semánticas de las siguientes gramáticas de atributos

$S \rightarrow AB$ $A \rightarrow aA$ cA $B \rightarrow bB$ b	$S.n = A.n + B.n$ $A_0.n = A_1.n + 1$ $A.n = 1$ $\text{if } (A_1.n > 0) \text{ then } A_0.n = A_1.n - 1$ $\text{else } A_0.n = A_1.n$ $B.n = B.n + 1$ $B.n = 1$
--	---

32

Implementación

Traducción Descendente

Ejercicio:

Factorizar y transformar las ecuaciones semánticas de las siguientes gramáticas de atributos

```
S -> Seq # Seq      S.error = (Seq0.c = Seq1.c)
Seq -> 0 Seq        Seq0.c = "0" || Seq1.c
                  | 0           Seq.c = "0"
                  | 1 Seq        Seq0.c = "1" || Seq1.c
                  | 1           Seq.c = "1"
```

```
Lid->id,Lid  Lid1.tipoh = Lid0.tipoh
                Lid0.ts = Lid1.ts
                Lid1.tsh = añadeID(Lid0.tsh, id.lex,Lid0.tipoh)
Lid->id        Lid.ts = añadeID(Lid.tsh, id.lex,Lid.tipoh)
```

33

Implementación

Traducción Descendente

Ejercicio:

Factorizar y transformar las ecuaciones semánticas de las siguientes producciones de una gramática de atributos genérica

```
A --> α β1      A.n = f1( α.n, β1.n)
A --> α β2      A.n = f2( α.n, β1.n)
```

34

Implementación Traducción Descendente

- En un entorno de implementación que utilice un lenguaje imperativo, la especificación del procesador de un lenguaje definido mediante una gramática *l-atribuida* puede hacerse mediante *esquemas de traducción* de acuerdo con las siguientes normas (*el orden es importante*):
 - Los atributos heredados de los símbolos en las partes derechas de las producciones deben calcularse en acciones que precedan a dichos símbolos.
 - En principio, las acciones no pueden referirse a atributos sintetizados de símbolos que estén a su derecha (más adelante veremos técnicas para eliminar esta restricción).
 - Los atributos sintetizados del no terminal de la izquierda sólo pueden calcularse cuando se hayan calculado todos los atributos de los que dependen. Los cálculos de estos atributos sintetizados se llevan a cabo en una acción que se sitúa al final de la producción.

35

Implementación Traducción Descendente

- Así mismo, y dependiendo del entorno de implementación, es posible ahorrar espacio en el almacenamiento de los valores de los atributos, haciendo que estos hagan referencia al mismo objeto.
- Esta situación ocurre en la práctica con el manejo de la tabla de símbolos y con el resultado de la traducción (la secuencia de código objeto):
 - La tabla de símbolos puede modificarse destrutivamente, y, de hecho, puede disponerse de ella globalmente, lo que ahorra su propagación explícita.
 - Cuando el resultado de la traducción es una secuencia de instrucciones simples, dicho resultado puede almacenarse en algún tipo de secuencia (e.g. una lista o un array) del que también puede disponerse de manera global.
- *Discusión: ¿en qué se apoyan estas optimizaciones?*

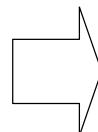
36

Implementación Traducción Descendente

```
...
Exp --> Exp OpAd Term
    Exp1.tsh = Term.tsh = Expo.tsh
    Expo.cod = Exp1.cod || Term.cod || OpAd.op

Exp --> Term
    Term.tsh = Exp.tsh
    Exp.cod = Term.cod
...

```



Eliminar
recursión a
izquierdas

37

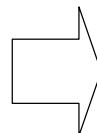
Implementación Traducción Descendente

```
...
Exp --> Term RExp
    Term.tsh = RExp.tsh = Exp.tsh
    RExp.codh = Term.cod
    Exp.cod = RExp.cod

RExp --> OpAd Term RExp
    Term.tsh = RExp1.tsh = RExpo.tsh
    RExp1.codh = RExpo.codh || Term.cod || OpAd.op
    RExpo.cod = RExp1.cod

RExp --> λ
    RExp.cod = RExp.codh
...

```



Obtener
esquema de
traducción

38

Implementación Traducción Descendente

- Si el código y la tabla de símbolos se asumen globales (los *atributos remotos en el sentido de Knuth*, se implementan como variables globales), el esquema de traducción se simplifica, ya que puede obviarse la propagación explícita de los atributos asociados.

```
...
Exp --> Term RExp
RExp --> OpAd Term {genOpAdd(opDeOpAd); } RExp
RExp --> λ
...
```

- *discusión: si el atributo código y tabla de símbolos no se implementasen como globales ¿cómo sería el esquema de traducción?*

39

Implementación Traducción Descendente

- La conversión del esquema de traducción a un traductor descendente predictivo recursivo es muy simple:
 - Los atributos sintetizados se añaden como parámetros de salida a los correspondientes procedimientos, y los heredados como parámetros de entrada.
 - En el cuerpo se declararán, como variables locales, los atributos que deban ser inicializados o evaluados en la correspondiente categoría sintáctica.
 - Las llamadas a los procedimientos incluirán los parámetros apropiados.
 - como hemos visto en el ejemplo anterior, en muchas ocasiones, puede hacerse alguna simplificación, que ahorré algunas variables y/o sentencias de asignación.

40

Implementación Traducción Descendente

```
PROCEDURE Exp;  
  
BEGIN  
    Term;  
    RExp  
END;  
  
PROCEDURE RExp;  
VAR  
    opDeOpAd: Char;  
  
BEGIN  
    IF info.token.token = TKMAS THEN  
        BEGIN  
            OpAd(opDeOpAd);  
            Term;  
            genOpAdd(opDeOpAd);  
            RExp  
        END  
    END;  
END;
```

41

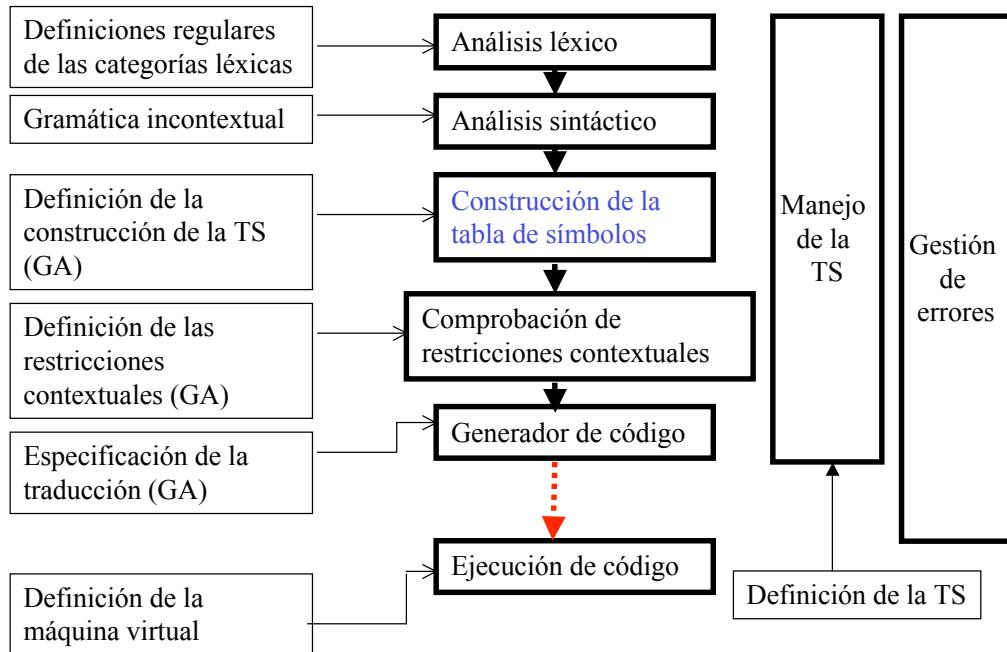
Implementación Implementación de la máquina virtual

- La implementación de la máquina virtual como un intérprete es sencilla:
 - Se decide una estructura de datos adecuada para implementar el estado.
 - El intérprete se organiza como:
 - La creación del estado inicial a partir del programa.
 - Un bucle en cuyo cuerpo se discrimina la regla a aplicar, y se actualiza el estado de acuerdo con dicha regla.
 - En un compilador se debe separar la máquina virtual del traductor. El programa, la frase del lenguaje objeto, se *serializa* en un fichero. En el caso de lenguajes sencillos (secuencias de instrucciones), el programa puede representarse de manera eficiente en forma de *código de bytes* (cada instrucción se codifica mediante uno o más bytes, indicando en el primero el código de la instrucción, y en el resto los operandos).

42

Implementación

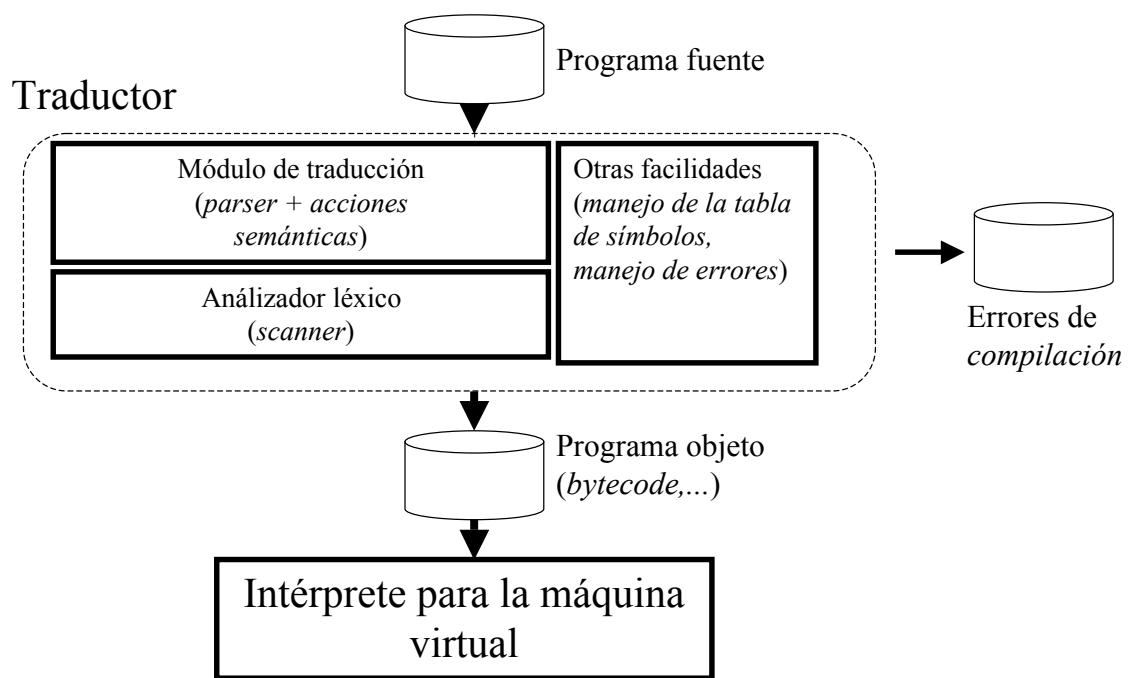
Estructura Detallada: Fases de un Procesador de Lenguaje



43

Implementación

Estructura Detallada de un Procesador de Lenguaje



44

