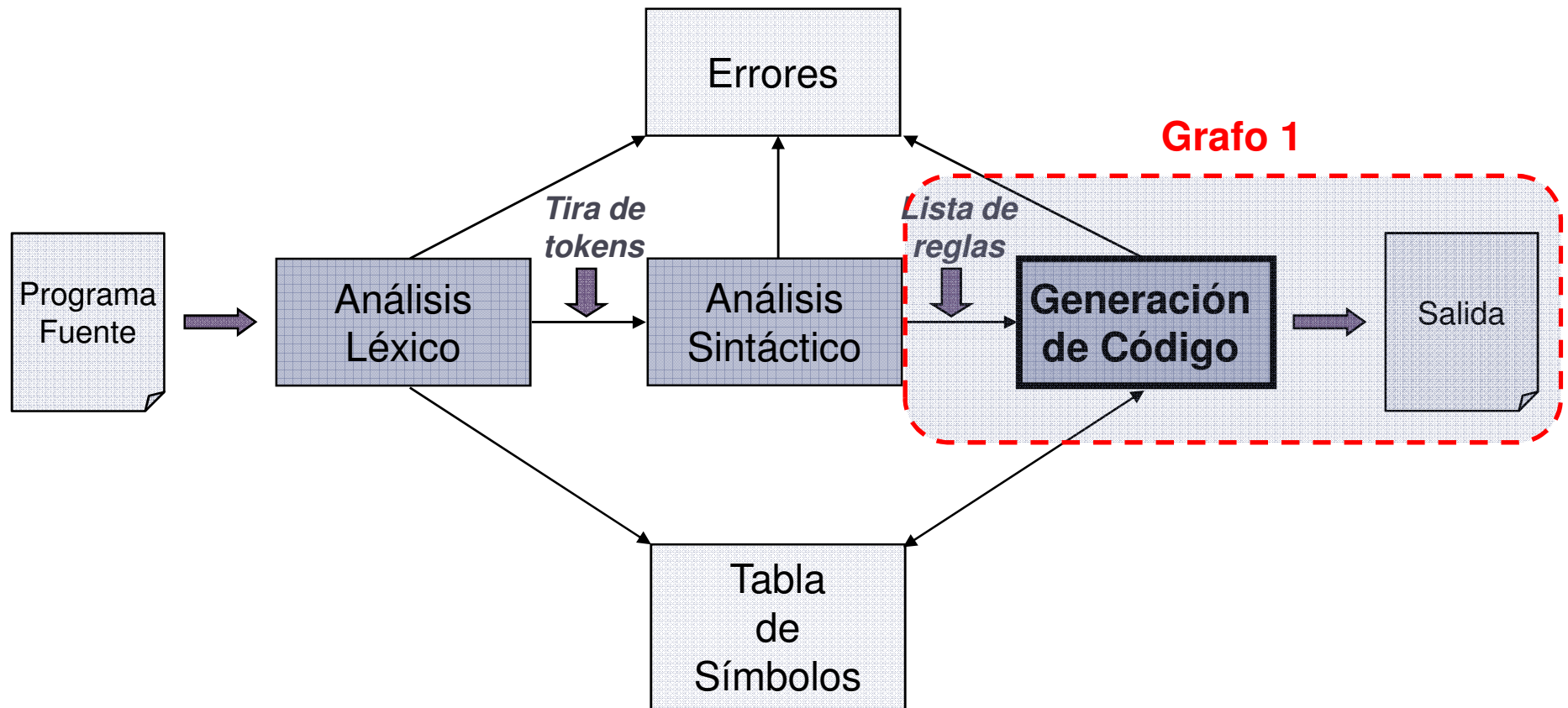


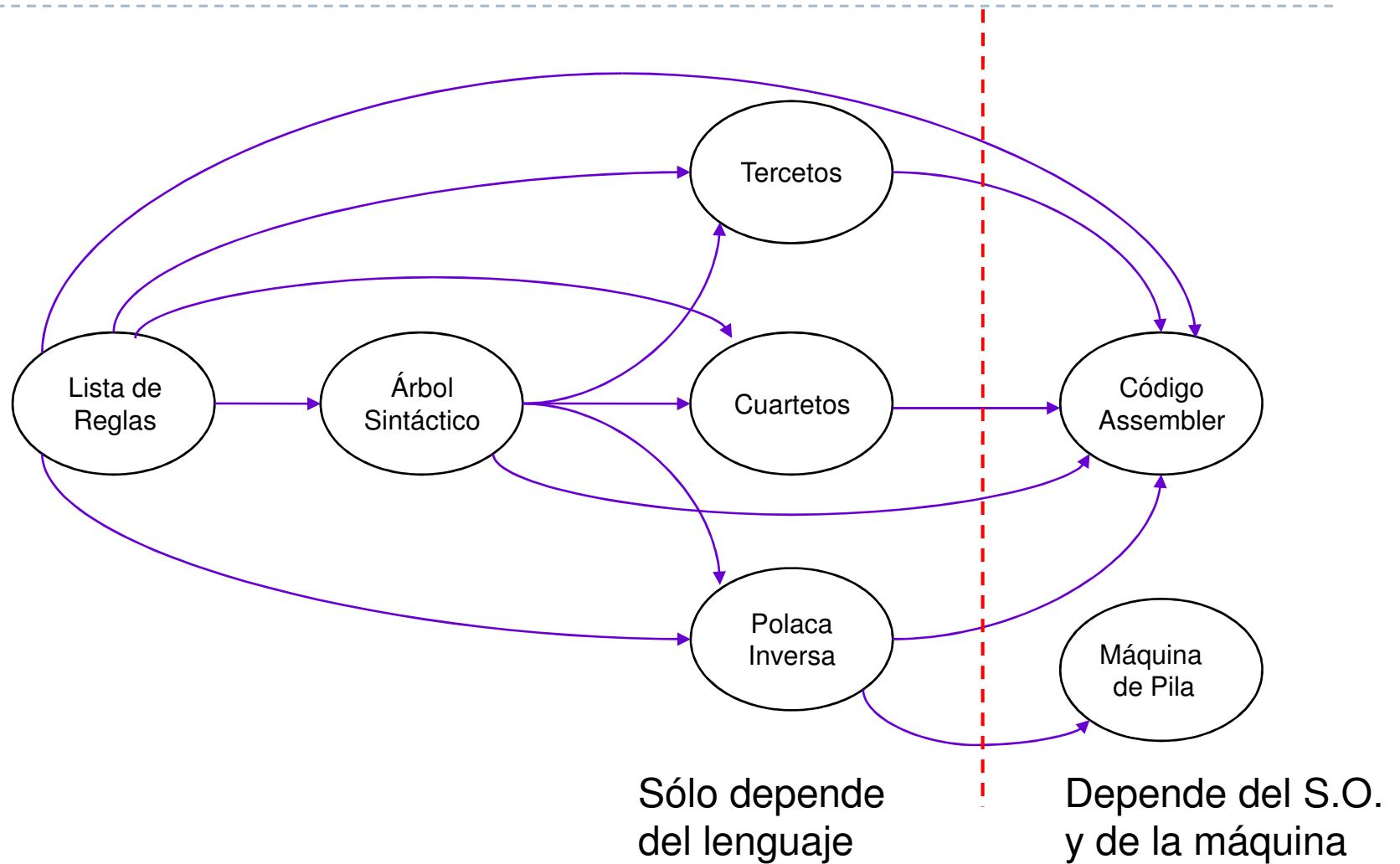
# Diseño de Compiladores I

Optimizaciones

# Fases de la Compilación



# Generación de Código



Generación de Código

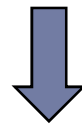
# Comparación de caminos

---

Compilador rápido (el más rápido)



Ejecutable grande y lento



Lista de Reglas → Assembler



# Comparación de caminos

---

Ejecutable rápido



Compilador lento



L.Reglas → Á.Sintáctico → Tercetos → Assembler  
(Optimizaciones)



# Comparación de caminos

---

**Desarrollo Rápido**



**Compilador relativamente rápido**



**Ejecutable relativamente bueno**



Lista de Reglas → Polaca Inversa → Assembler  
(Algunas Optimizaciones)



# Optimización

---

## Objetivo:

*Lograr que el código sea lo mejor posible*

## Resultados:

*Nunca se obtiene código óptimo*

---



# Optimización

---

## Propiedades:

- ▶ Una transformación debe preservar la semántica del programa.
- ▶ Debe acelerar el programa en una cantidad medible.
- ▶ Debe valer la pena.





# Optimización

---

## Causas:

- ▶ El programador comete errores de programación.

Ejemplo:

`a:=2*3*b` en lugar de `a:=6*b`

- ▶ El programador prefiere legibilidad.

Ejemplo:

```
#define LARGO 20
```

```
#define ANCHO 10
```

```
...
```

```
volumen := LARGO * ANCHO * alto;
```

- ▶ El programador no puede evitarlo.
- 



# Redundancia Simple

Tercetos

Cuartetos

Polaca Inversa

# Redundancia Simple

(subexpresiones comunes)

---

- ▶ Una expresión E es una subexpresión común si E ha sido calculada previamente, y los valores de las variables usadas en E no han cambiado desde dicho cálculo.
- ▶ Se puede evitar recalcular la expresión utilizando el valor calculado previamente (se eliminan operaciones duplicadas)
- ▶ Se tienen en cuenta propiedades de conmutatividad y asociatividad.

▶ Ejemplo:

$$a := b * c * d + b * c * e$$

$$a := b * c * (d + e)$$



# Redundancia en Tercetos

---

$x := (a * b) * c + e * f + (a * b) * d;$

- ▶ Se buscan tercetos repetidos hasta la asignación, se cambian las referencias, y se reenumera.

Tercetos:

- 28.  $(*, a, b)$
- 29.  $(*, [28], c)$
- 30.  $(*, e, f)$
- 31.  $(+, [29], [30])$
- 32.  $(*, a, b)$
- 33.  $(*, [32], d)$
- 34.  $(+, [31], [33])$
- 35.  $(:=, x, [34])$

Tercetos optimizados:

- 28.  $(*, a, b)$
- 29.  $(*, [28], c)$
- 30.  $(*, e, f)$
- 31.  $(+, [29], [30])$
- 32.  $(*, [28], d)$
- 33.  $(+, [31], [32])$
- 34.  $(:=, x, [33])$

---

¿INCONVENIENTES DE ESTA SOLUCIÓN?

# Redundancia en Tercetos

---

- ▶ Mejor solución: Cambiar la referencia en el o los tercetos que referencian al terceto redundante, y luego eliminarlo.

## Tercetos:

- 28.  $( *, a , b )$
- 29.  $( *, [28] , c )$
- 30.  $( *, e , f )$
- 31.  $( + , [29] , [30] )$
- 32.  $( *, a , b )$
- 33.  $( *, [32] , d )$
- 34.  $( + , [31] , [33] )$
- 35.  $( := , x , [34] )$

## Tercetos optimizados:

- 28.  $( *, a , b )$
- 29.  $( *, [28] , c )$
- 30.  $( *, e , f )$
- 31.  $( + , [29] , [30] )$
- 32.  ~~$( *, a , b )$~~
- 33.  $( *, [28] , d )$
- 34.  $( + , [31] , [33] )$
- 35.  $( := , x , [34] )$



# Redundancia en Cuartetos

---

$x := a * b * c + e * f + a * b * d;$

Cuartetos:

20. ( \* , a , b , aux1 )
21. ( \* , aux1 , c , aux2 )
22. ( \* , e , f , aux3 )
23. ( + , aux2 , aux3 , aux4 )
24. ( \* , a , b , aux5 )
25. ( \* , aux5 , d , aux6 )
26. ( + , aux4 , aux6 , aux7 )
27. ( := , x , aux7 , - )

Cuartetos optimizados:

20. ( \* , a , b , aux1 )
21. ( \* , aux1 , c , aux2 )
22. ( \* , e , f , aux3 )
23. ( + , aux2 , aux3 , aux4 )
24. ( \* , **aux1** , d , aux6 )
25. ( + , aux4 , aux6 , aux7 )
26. ( := , x , aux7 , - )



# Redundancia en Polaca Inversa

$x := a * b * c + e * f + a * b * d;$

Polaca Inversa:

x	a	b	*	c	*	e	f	*	+	a	b	*	d	*	+	:=
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	----

		f		b	d		
b	c	e	e*f	a	a*b	a*b*d	a*b*c+e*f+
a	a*b	a*b*c	a*b*c	a*b*c+e*f	a*b*c+e*f	a*b*c+e*f	a*b*d
x	x	x	x	x	x	x	x

Polaca Inversa optimizada:

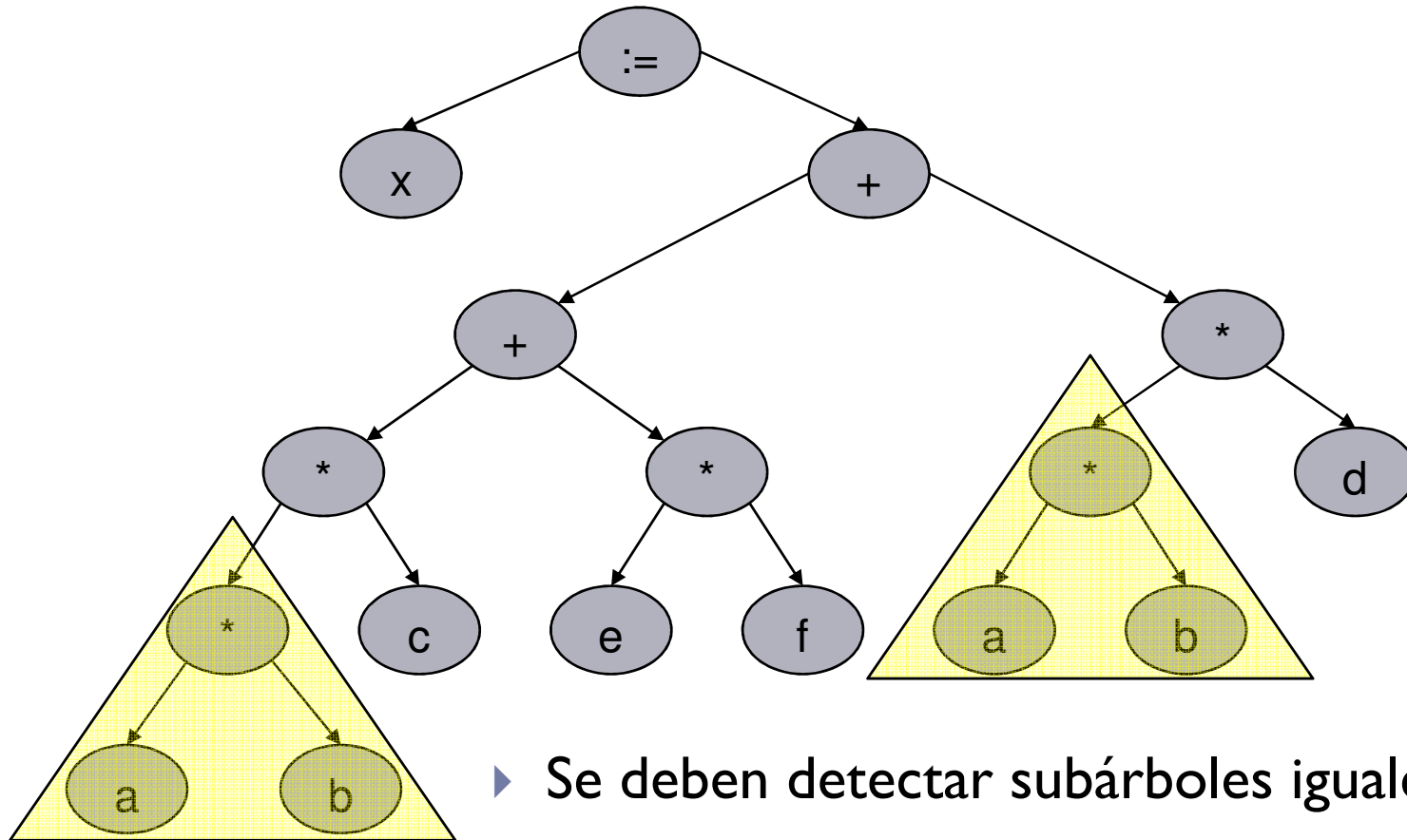
x	a	b	*	DUP	c	*	e	f	*	+	SWAP	d	*	+	:=
---	---	---	---	-----	---	---	---	---	---	---	------	---	---	---	----

		f			d		
b	c	e	e*f	a*b*c+e*f	a*b	a*b*d	a*b*c+e*f+
a	a*b	a*b*c	a*b	a*b	a*b*c+e*f	a*b*c+e*f	a*b*d
x	x	x	x	x	x	x	x

Diagram illustrating the optimization step: A yellow arrow points from the 'a\*b' value in the stack to the 'a\*b' value in the stack, indicating a swap operation to avoid redundant calculation.

# Redundancia en Árbol Sintáctico

$x := a * b * c + e * f + a * b * d;$



- ▶ Se deben detectar subárboles iguales
- ▶ Es muy costoso
- ▶ **No se hace**





# Reducción Simple

Árbol Sintáctico

Polaca Inversa

Tercetos

# Reducción Simple

---

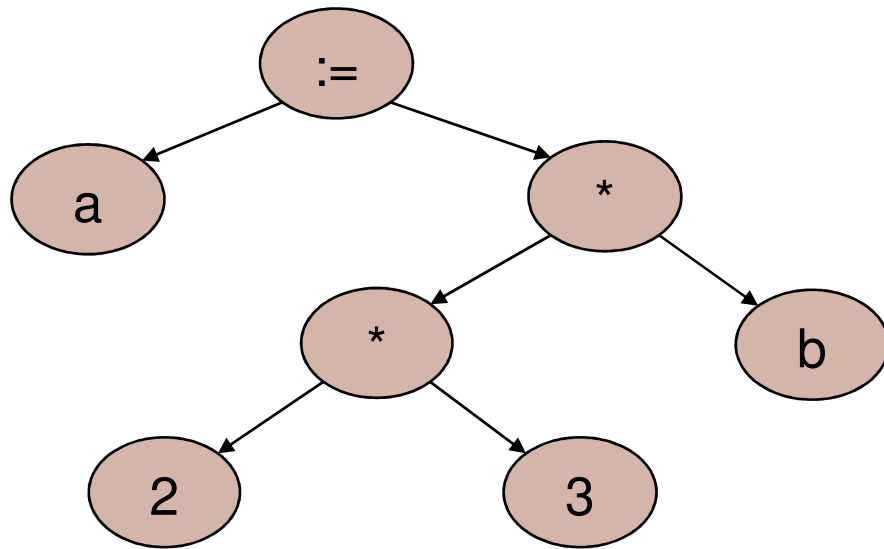
- ▶ Se evalúan expresiones que son conocidas en *tiempo de compilación* (entre constantes), y se sustituyen por su valor.



# Reducción Simple en Árbol Sintáctico

---

$a := 2 * 3 * b$



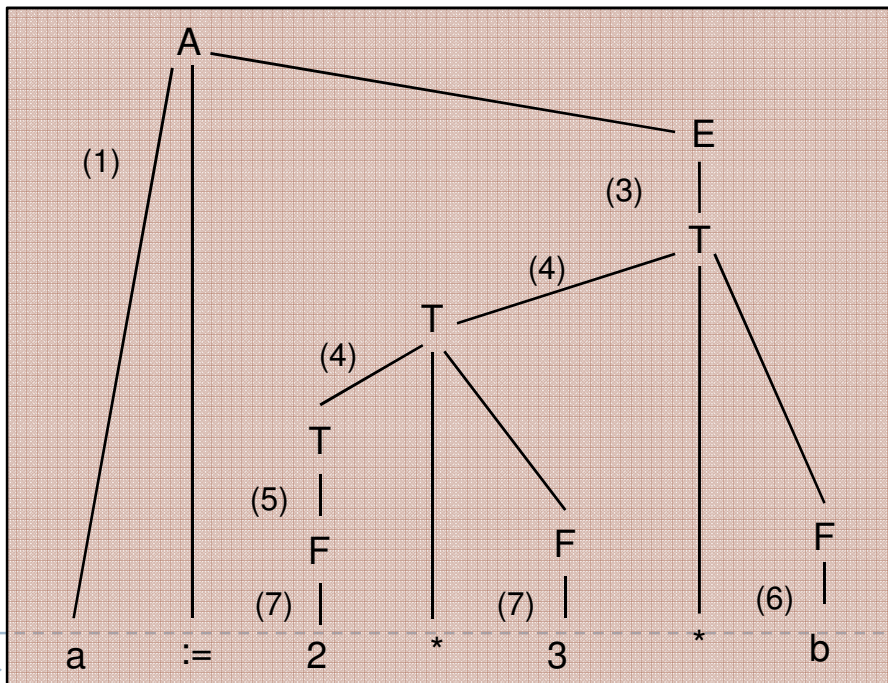
- ▶ Se puede aplicar:
  - ▶ Durante la construcción.
  - ▶ A la salida (hacia otra representación).
  - ▶ Sobre el árbol terminado.



# Reducción Simple en Árbol Sintáctico

## Durante la construcción

- 1)  $A \rightarrow id := E$      $A.ptr := crear\_nodo( := , crear\_hoja( id.ref\_TS ) , E.ptr )$
- 2)  $E \rightarrow E + T$      $E.ptr := crear\_nodo( + , E.ptr , T.ptr )$
- 3)  $E \rightarrow T$      $E.ptr := T.ptr$
- 4)  $T \rightarrow T * F$      $T.ptr := crear\_nodo( * , T.ptr , F.ptr )$
- 5)  $T \rightarrow F$      $T.ptr := F.ptr$
- 6)  $F \rightarrow id$      $F.ptr := crear\_hoja( id.ref\_TS )$
- 7)  $F \rightarrow cte$      $F.ptr := crear\_hoja( cte.ref\_TS )$



$a := 2 * 3 * b$

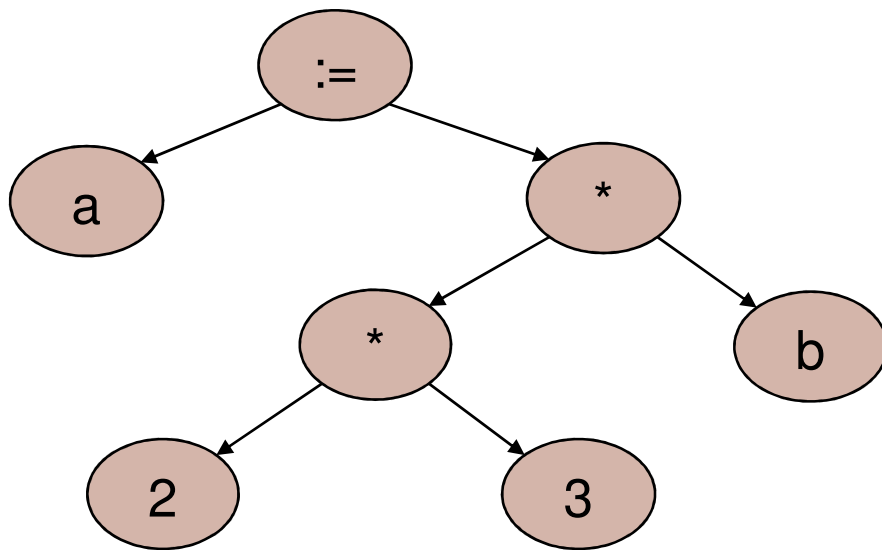
← Árbol de Parsing

**Lista de Reglas: 7 5 7 4 6 4 3 |**

# Reducción Simple en Árbol Sintáctico

## Durante la construcción

- 1)  $A \rightarrow id := E$      $A.ptr := crear\_nodo( := , crear\_hoja( id.ref\_TS ) , E.ptr )$
- 2)  $E \rightarrow E + T$      $E.ptr := crear\_nodo( + , E.ptr , T.ptr )$
- 3)  $E \rightarrow T$      $E.ptr := T.ptr$
- 4)  $T \rightarrow T * F$      $T.ptr := crear\_nodo( * , T.ptr , F.ptr )$
- 5)  $T \rightarrow F$      $T.ptr := F.ptr$
- 6)  $F \rightarrow id$      $F.ptr := crear\_hoja( id.ref\_TS )$
- 7)  $F \rightarrow cte$      $F.ptr := crear\_hoja( cte.ref\_TS )$



$a := 2 * 3 * b$

← Árbol Sintáctico

**Lista de Reglas: 7 5 7 4 6 4 3 |**



# Reducción Simple en Árbol Sintáctico

## Durante la construcción

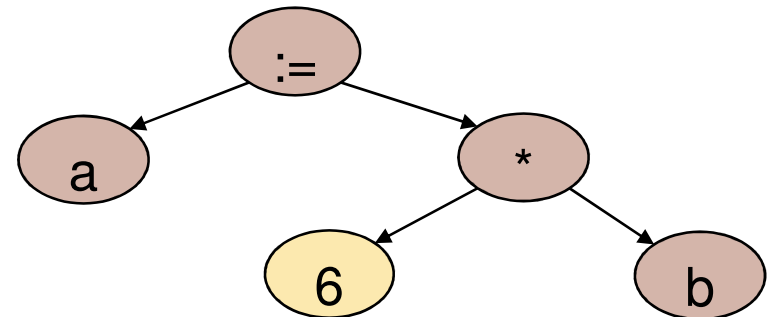
- 1)  $A \rightarrow id := E$      $A.ptr := crear\_nodo( := , crear\_hoja( id.ref\_TS , E.ptr )$
- 2)  $E \rightarrow E + T$      $\leftarrow$  Idem regla 4, para la suma
- 3)  $E \rightarrow T$      $E.ptr := T.ptr$
- 4)  $T \rightarrow T * F$     **IF ((es\_cte(T.ptr) AND es\_cte(F.ptr)) THEN**  
                  **calculo = valor(T.ptr) \* valor(F.ptr)**  
                  **Borrar(T.ptr)**  
                  **Borrar(F.ptr)**  
                  **T.ptr := crear\_hoja(Alta\_TS(calculo))**  
                  **ELSE**  
                  **T.ptr := crear\_nodo( \* , T.ptr , F.ptr )**
- 5)  $T \rightarrow F$      $T.ptr := F.ptr$
- 6)  $F \rightarrow id$      $F.ptr := crear\_hoja( id.ref\_TS )$
- 7)  $F \rightarrow cte$      $F.ptr := crear\_hoja( cte.ref\_TS )$

$a := 2 * 3 * b$

**Lista de Reglas:**

**7 5 7 4 6 4 3 1**

**Árbol Sintáctico  
Optimizado**

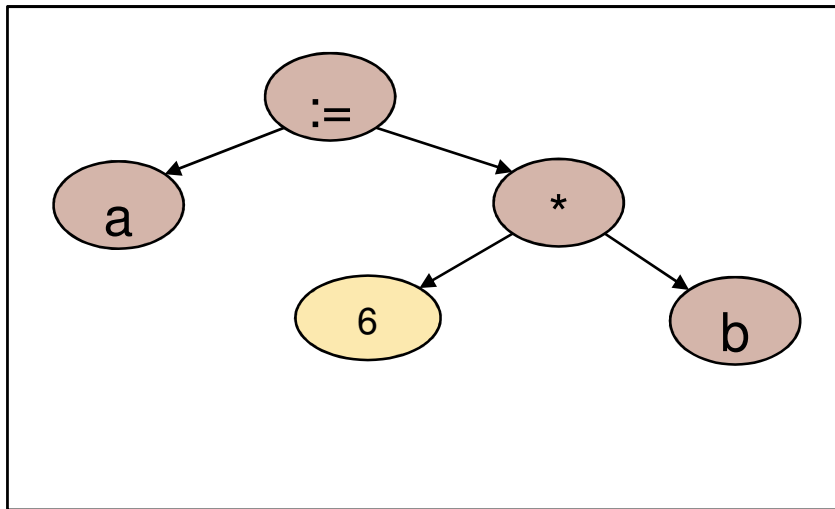


# Reducción Simple en Árbol Sintáctico

## A la salida

En la traducción a Assembler, u otra representación intermedia

$a := 2 * 3 * b$



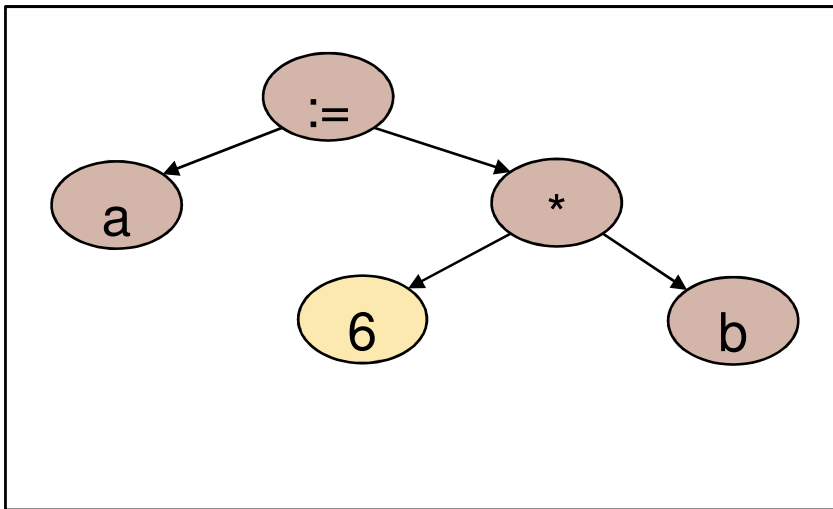
Árbol Sintáctico  
Optimizado

- ▶ Se busca el subárbol de más a la izquierda con hijos hojas
  - ▶ Si los hijos son constantes
    - ▶ Calcular
    - ▶ Reemplazar el subárbol por el resultado
  - ▶ Si los hijos no son constantes
    - ▶ Generar código (Assembler u otra repr. intermedia)
    - ▶ Reemplazar el subárbol por el lugar donde quedó el resultado (registro, variable auxiliar, nro. de terceto)

# Reducción Simple en Árbol Sintáctico

Con el árbol terminado

$a := 2 * 3 * b$



Árbol Sintáctico  
Optimizado

- ▶ Se recorre el árbol buscando subárboles cuyos hijos sean constantes:
  - ▶ Calcular
  - ▶ Reemplazar el subárbol por el resultado

*Se recorre 2 veces el árbol*





# Reducción Simple en Polaca Inversa

---

$a := 2 * 3 * b$

<b>a</b>	<b>2</b>	<b>3</b>	<b>*</b>	<b>b</b>	<b>*</b>	<b>:=</b>
----------	----------	----------	----------	----------	----------	-----------

- ▶ Se puede aplicar:
  - ▶ Durante la construcción.
  - ▶ A la salida (hacia otra representación).
  - ▶ Sobre la Polaca terminada.



# Reducción Simple en Polaca Inversa

Durante la construcción (desde Lista de Reglas)

1)	$A \rightarrow id := E$	Agregar (id.ref_TS );Agregar ( := )
2)	$E \rightarrow E + T$	Agregar ( + )
3)	$E \rightarrow T$	-
4)	$T \rightarrow T * F$	Agregar ( * )
5)	$T \rightarrow F$	-
6)	$F \rightarrow id$	Agregar ( id.ref_TS )
7)	$F \rightarrow cte$	Agregar ( cte.ref_TS )

a := 2 \* 3 \* b

**Lista de Reglas:**

**7 5 7 4 6 4 3 1**

<b>2</b>	<b>3</b>	<b>*</b>	<b>b</b>	<b>*</b>	<b>a</b>	<b>:=</b>
----------	----------	----------	----------	----------	----------	-----------



Polaca Inversa

**Nota:** Si se está generando PI a partir del árbol Sintáctico,

- ▶ se aplicará el algoritmo de RS en AS a la salida

# Reducción Simple en Polaca Inversa

Durante la construcción (desde Lista de Reglas)

1)	$A \rightarrow id := E$	Agregar (id.ref_TS ); Agregar ( := )
2)	$E \rightarrow E + T$	← Idem regla 4, para la suma
3)	$E \rightarrow T$	-
4)	$T \rightarrow T * F$	<b>IF (es_cte(PI(pos-1)) AND es_cte(PI(pos-2))) THEN</b> <b>calculo := valor(PI(pos-1)) * valor(PI(pos-2))</b> <b>Borrar(PI(pos-1))</b> <b>Borrar(PI(pos-2))</b> <b>Agregar(Alta_TS(calculo))</b> <b>ELSE</b> <b>Agregar ( * )</b>
5)	$T \rightarrow F$	-
6)	$F \rightarrow id$	Agregar ( id.ref_TS )
7)	$F \rightarrow cte$	Agregar ( cte.ref_TS )

<b>6</b>	<b>b</b>	<b>*</b>	<b>a</b>	<b>:=</b>
----------	----------	----------	----------	-----------



Polaca Inversa  
Optimizada

a := 2 \* 3 \* b

Lista de Reglas:

**7 5 7 4 6 4 3 1**



# Reducción Simple en Polaca Inversa

## A la salida

En la traducción a Assembler

$a := 2 * 3 * b$

2	3	*	b	*	a	:=
---	---	---	---	---	---	----

3	b	
2	6	RI

IF (es\_operando(PI(pos))) THEN

apilar (PI(pos))

ELSE

IF (es\_operador\_binario(PI(pos))) THEN

IF (es\_cte(tope) AND es\_cte(tope - 1)) THEN

calculo := valor(tope) \* valor(tope - 1)

// Operación según PI(pos)

desapilar 2

apilar(calculo)

ELSE

desapilar 2

generar\_codigo

apilar resultado (registro o variable auxiliar)



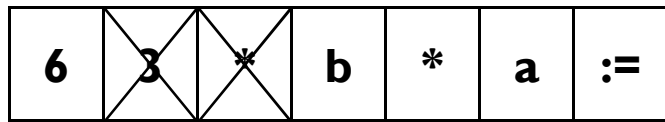
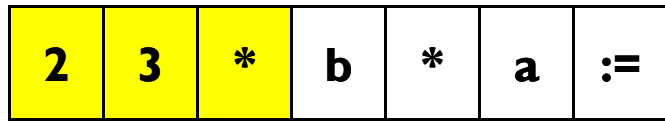
# Reducción Simple en Polaca Inversa

Con la Polaca terminada

---

$a := 2 * 3 * b$

- ▶ Se buscan ternas cte cte operador
- ▶ Se calcula el valor de la operación
- ▶ Se reemplaza el primer elemento de la terna por el resultado, y se anulan los otros 2 elementos de la terna



*Se recorre 2 veces la Polaca*



# Reducción Simple en Tercetos

---

$a := 2 * 3 * b$

Tercetos:

15. ( \* , 2 , 3 )
16. ( \* , [15] , b )
17. ( := , a , [16] )

- ▶ Se puede aplicar:
  - ▶ Durante la construcción.
  - ▶ A la salida (en la traducción a Assembler).
  - ▶ Sobre los Tercetos terminados.



# Reducción Simple en Tercetos

Durante la construcción (desde Lista de Reglas)

- 
- |    |                         |  |
|----|-------------------------|--|
| 1) | $A \rightarrow id := E$ | $A.ptr := crear\_terceto( := , id.ref\_TS , E.ptr )$ |
| 2) | $E \rightarrow E + T$   | $E.ptr := crear\_terceto( + , E.ptr , T.ptr )$       |
| 3) | $E \rightarrow T$       | $E.ptr := T.ptr$                                     |
| 4) | $T \rightarrow T * F$   | $T.ptr := crear\_terceto( * , T.ptr , F.ptr )$       |
| 5) | $T \rightarrow F$       | $T.ptr := F.ptr$                                     |
| 6) | $F \rightarrow id$      | $F.ptr := id.ref\_TS$                                |
| 7) | $F \rightarrow cte$     | $F.ptr := cte.ref\_TS$                               |

$a := 2 * 3 * b$

15.  $( * , 2 , 3 )$

16.  $( * , [15] , b )$

17.  $( := , a , [16] )$

 Tercetos

**Lista de Reglas: 7 5 7 4 6 4 3 1**

**Nota:** Si se están generando tercetos a partir del árbol Sintáctico,

▶ se aplicará el algoritmo de RS en AS a la salida

# Reducción Simple en Tercetos

Durante la construcción

---

- |    |                         |  |
|----|-------------------------|--|
| 1) | $A \rightarrow id := E$ | $A.ptr := crear\_terceto( := , id.ref\_TS , E.ptr )$   |
| 2) | $E \rightarrow E + T$   | $\leftarrow$ <b>Idem regla 4, para la suma</b>   |
| 3) | $E \rightarrow T$       | $E.ptr := T.ptr$   |
| 4) | $T \rightarrow T * F$   | <b>IF ((es_cte(T.ptr) AND es_cte(F.ptr)) THEN</b><br><b>    calculo = valor(T.ptr) * valor(F.ptr)</b><br><b>    T.ptr := Alta_TS(calculo)</b><br><b>ELSE</b><br><b>    T.ptr := crear_terceto( * , T.ptr , F.ptr )</b> |
| 5) | $T \rightarrow F$       | $T.ptr := F.ptr$   |
| 6) | $F \rightarrow id$      | $F.ptr := id.ref\_TS$  |
| 7) | $F \rightarrow cte$     | $F.ptr := cte.ref\_TS$   |

$a := 2 * 3 * b$

**Lista de Reglas:**  
**7 5 7 4 6 4 3 1**

15.  $( * , 6 , b )$   
16.  $( := , a , [15] )$



Tercetos  
Optimizados

---





# Reducción Simple en Tercetos

## A la salida

En la traducción a Assembler

---

$a := 2 * 3 * b$

15.  $( *, 2, 3 ) 6$

16.  $( *, [15], b ) @aux23$

17.  $( :=, a, [16] )$

▶ Para cada terceto:

- ▶ IF (es\_cte(operando1) AND (es\_cte(operando2)) THEN
  - ▶ calculo = valor(operando1) \* valor(operando2)
  - ▶ Agregar calculo al terceto
- ▶ ELSE
  - ▶ Generar\_codigo (terceto)
  - ▶ Agregar registro o auxiliar al terceto



# Reducción Simple en Tercetos

Con los tercetos terminados

---

$a := 2 * 3 * b$

15. ( \* , 2 , 3 )

16. ( \* , [15] , b )

17. ( := , a , [16] )



~~15. ( \* , 2 , 3 )~~

16. ( \* , **6** , b )

17. ( := , a , [16] )

---

▶ Recorrer los tercetos

▶ Para cada terceto:

▶ IF (es\_cte(operando1) AND (es\_cte(operando2))) THEN

▶ calculo = valor(operando1) \* valor(operando2)

▶ Reemplazar las referencias al terceto por calculo

▶ Eliminar terceto

▶ ELSE

▶ -

*Se recorren 2 veces los tercetos*



# Reordenamiento de Código

Árbol Sintáctico

# Reordenamiento de código

---

- ▶ Intenta minimizar el uso de registros.
- ▶ Aún habiendo registros suficientes, reduce la cantidad de operaciones.



# Reordenamiento de código

---

$a + b + c * d * (e + f + g * h * i) \rightarrow (g * h * i + e + f) * c * d + a + b$

MOV R1, a

ADD R1, b

MOV R2, c

MUL R2, d

MOV R3, e

ADD R3, f

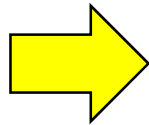
MOV R4, g

MUL R4, h

MUL R4, i

ADD R3, R4

MUL R2, R3



MOV R1, g

MUL R1, h

MUL R1, i

ADD R1, e

ADD R1, f

MUL R1, c

MUL R1, d

ADD R1, a

ADD R1, b

---

▶ ADD R1, R2

# Reordenamiento de código

---

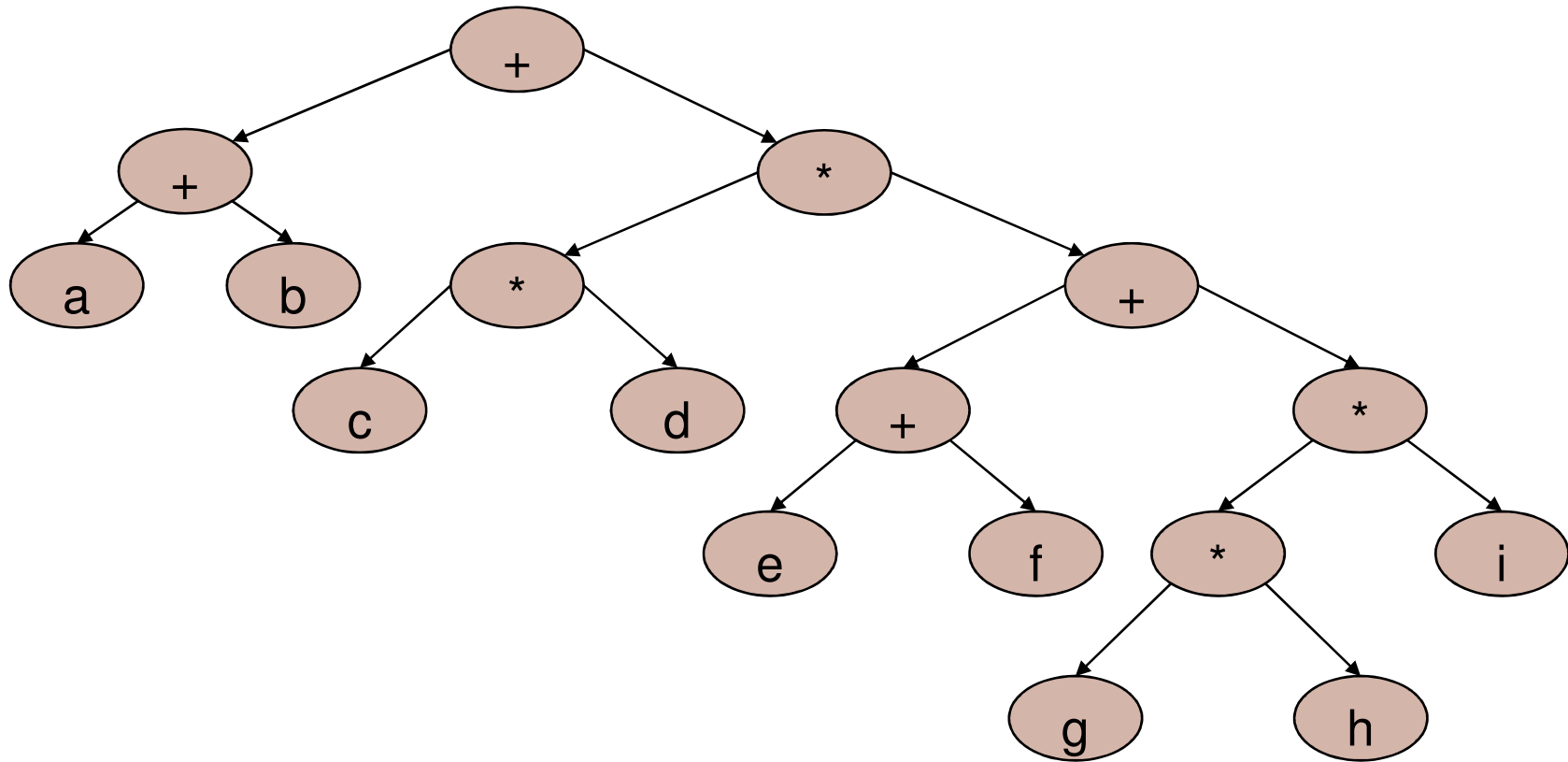
- ▶ Se aplica sobre el **Árbol Sintáctico**



# Reordenamiento de código

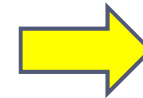
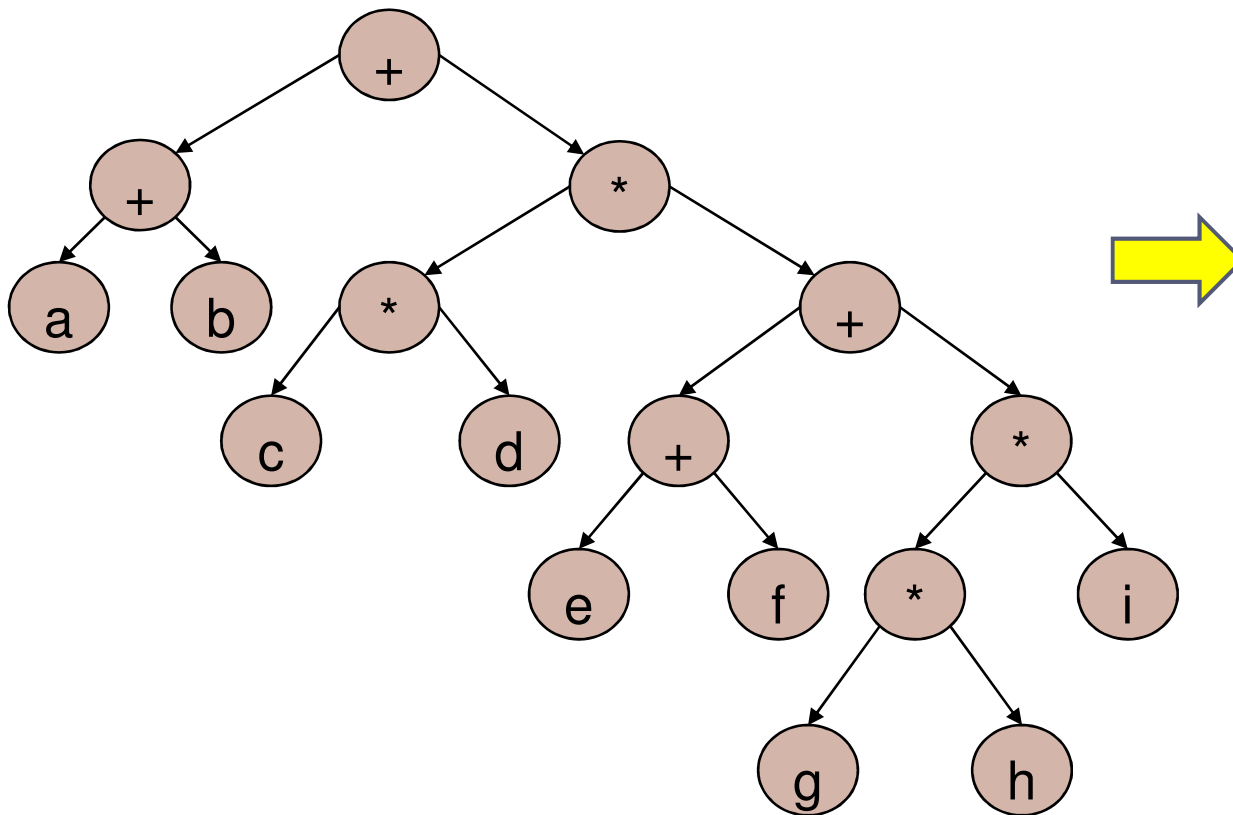
---

$a + b + c * d * (e + f + g * h * i)$



# Reordenamiento de código

$a + b + c * d * (e + f + g * h * i)$



```
MOV R1, a
ADD R1, b
MOV R2, c
MUL R2, d
MOV R3, e
ADD R3, f
MOV R4, g
MUL R4, h
MUL R4, i
ADD R3, R4
MUL R2, R3
ADD R1, R2
```



# Reordenamiento de código – Paso 1

---

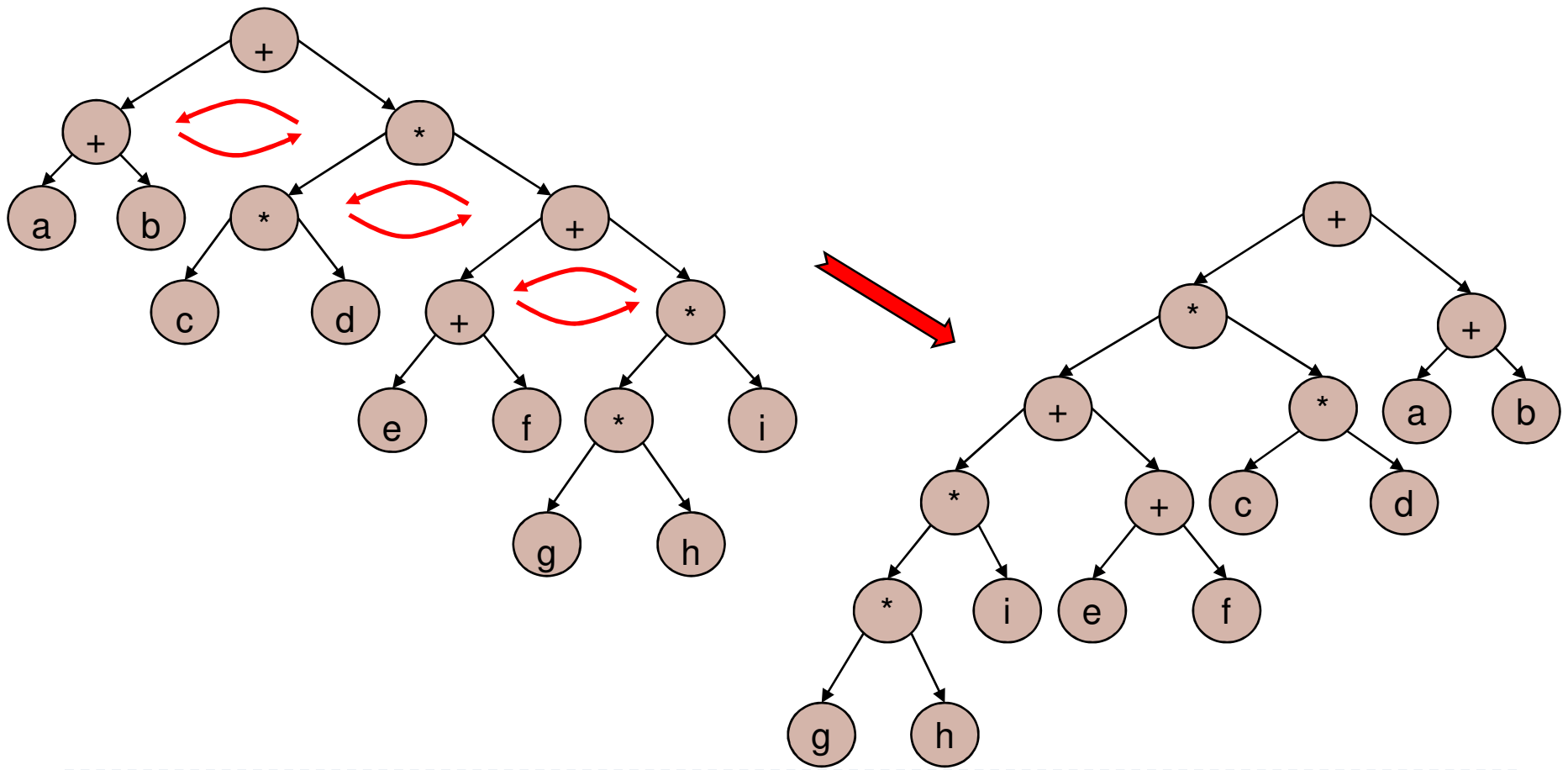
- ▶ Se evalúa la profundidad de las ramas de cada nodo (algoritmo de Nakata)
  - ▶ Si la rama izquierda tiene menor profundidad que la derecha, se invierten las ramas, siempre que el operador sea conmutativo.
- ▶ Se repite hasta que no se produzcan más inversiones.



# Reordenamiento de código – Paso 1

---

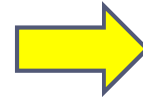
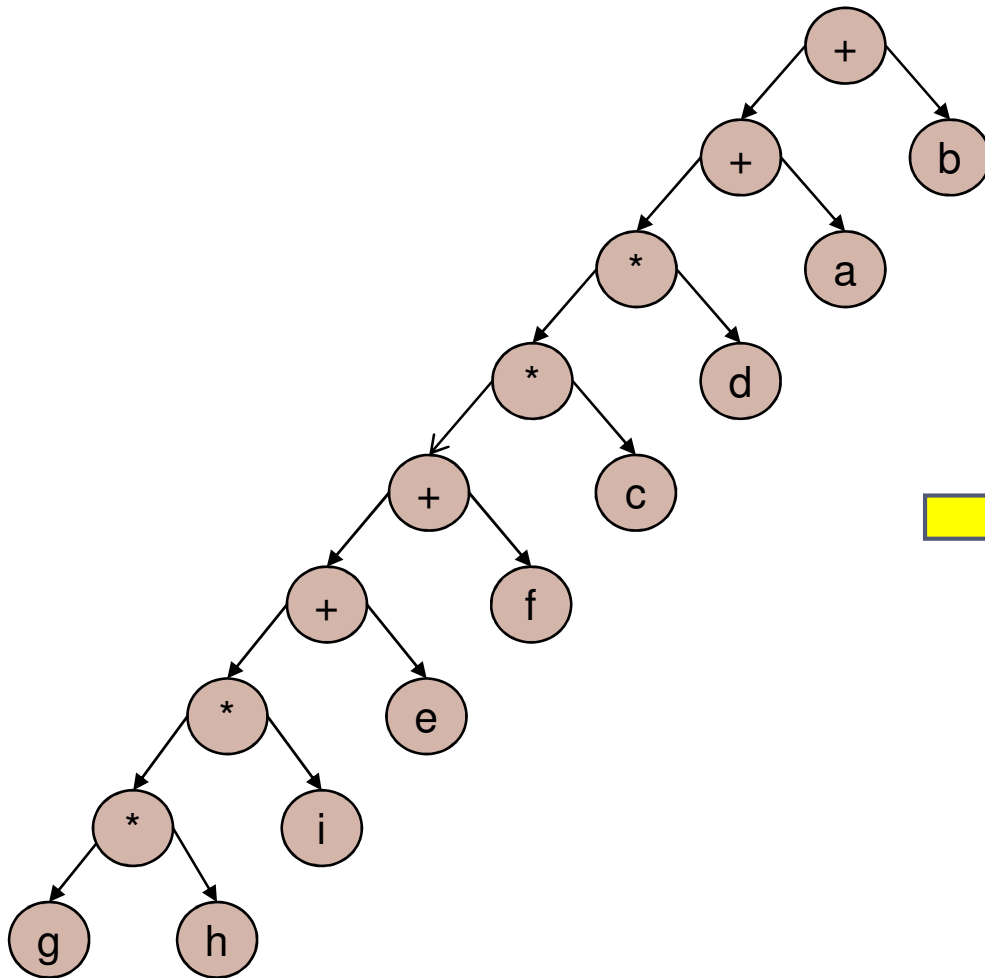
$$a + b + c * d * (e + f + g * h * i)$$





# Reordenamiento de código

---



```
MOV RI, g  
MUL RI, h  
MUL RI, i  
ADD RI, e  
ADD RI, f  
MUL RI, c  
MUL RI, d  
ADD RI, a  
ADD RI, b
```

# Reordenamiento de código

---

- ▶ Desventaja:

- ▶ Puede cambiar situaciones de overflow

- ▶ Ejemplo:  $a + b + c * d * e$    $d * e * c + a + b$

con

a=30000

b=-30000

c=5000

d=2

e=2

***OVERFLOW!!***



# Otras optimizaciones

---

- ▶ Reducción no simple
  - ▶  $2 * a * 3$
- ▶ Redundancia no simple
  - ▶  $z = a * b + 7 + b * a$
- ▶ Propagación de copias de constantes
  - ▶  $a := 2$
  - ▶  $b := a * 3 \rightarrow b := 6$
- ▶ Reducción de esfuerzo
  - ▶  $a^2$  es menos costoso hacer  $a * a$
- ▶ Código muerto (IF, WHILE, Asignación)



# Ejercicios

---

- ▶ Considerar las siguientes expresiones en las que existen operaciones redundantes:
  - ▶  $a * b * c - d + a * b$
  - ▶  $a * b - d + c * a * b$
  - ▶  $a * b * c * f - d + b * c$
- ▶ ¿Qué optimizaciones de redundancia se pueden hacer en cada expresión?
- ▶ ¿Y si la gramática fuera recursiva a derecha?



# Ejercicios

---

- ▶ Considerar la optimización de redundancia realizada sobre los tercetos terminados.
- ▶ Si existen más de dos ocurrencias de la operación redundante en la expresión:
  - ▶ ¿Se podrán optimizar todas las ocurrencias?
  - ▶ ¿Cómo se realizaría?
  - ▶ ¿Tendrá algún impacto la optimización en la Traducción a Assembler?
    - ▶ Usando variables auxiliares
    - ▶ Usando seguimiento de registros





¿PREGUNTAS?