Generación de Código

CI4722 – Lenguajes de Programación III

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo escoger las mejores instrucciones...

- ...si la arquitectura tiene muchos modos de direccionamiento?
- ...tiene instrucciones de propósito específico?
- ...si tiene varias instrucciones para un mismo propósito?

Otro problema combinatorio complejo...

UNIVERSIDAD SIMÓN BOLÍVAR

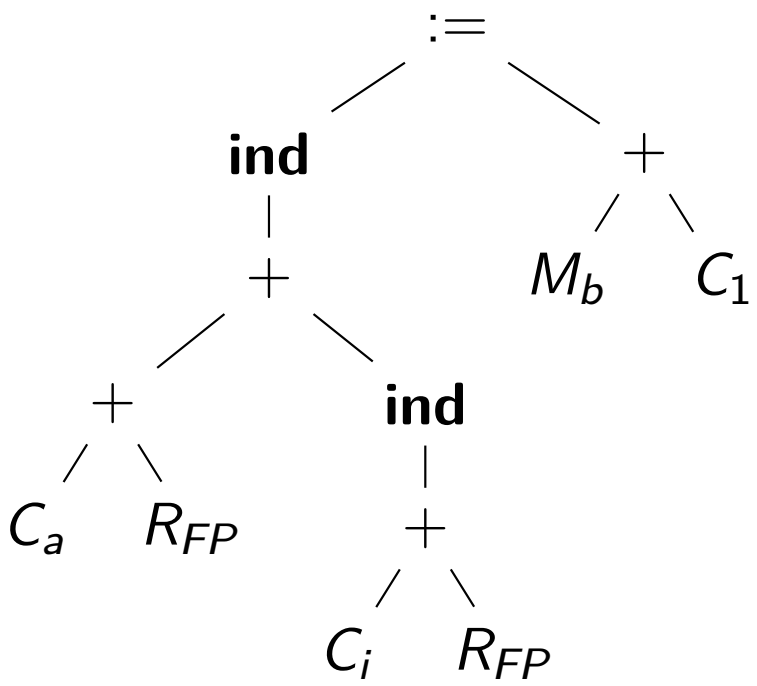
Generación basada en árboles decorados

- El generador de código trabaja sobre *árboles*.
- Los árboles representan las instrucciones.
  - TAC transformado en código destino.
  - Nombres sustituidos por ubicación en memoria – global o pila.
  - Modos de direccionamiento explícitos en nodos internos – típicamente el indirecto para denotar l-values.
- Las hojas contienen atributos
  - C_v para un valor constante – v es el valor.
  - R_r para un registro – r es el nombre del registro.
  - M_n para una dirección de memoria – n es el nombre asociado.

UNIVERSIDAD SIMÓN BOLÍVAR

Un árbol típico

```
a[i] := b + 1
```



- `b` es una variable global.
- `a` es un arreglo local.
- `i` es una variable local.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo generamos código?

Tree Translation Scheme

- Aplicaremos *reglas de reescritura* de la forma

$$replacement \leftarrow template \ \{ \ action \ \}$$

- El método de generación consiste en:
  - Identificar un árbol que coincida con *template*.
  - Sustituirlo por *replacement*.
  - Posiblemente emitr código como efecto de *action*.
  - Colapsar el árbol paulatinamente con las reescrituras.
  - Repetir hasta que reste solamente un nodo.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo generamos código?

Tree Translation Scheme

- Aplicaremos *reglas de reescritura* de la forma

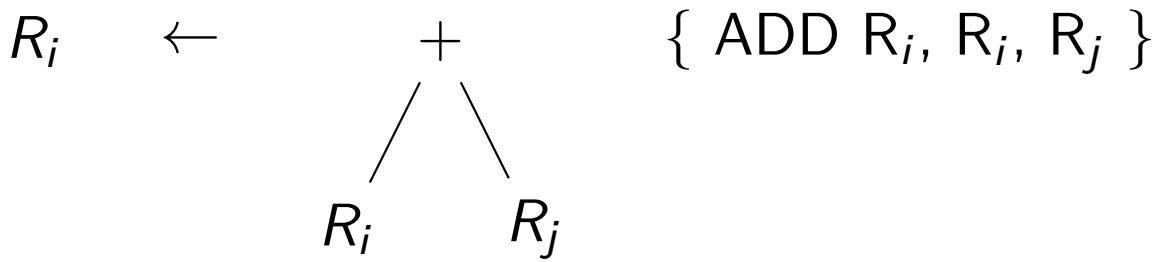
$$replacement \leftarrow template \ \{ \ action \ \}$$

- El método de generación consiste en:
  - Identificar un árbol que coincida con *template*.
  - Sustituirlo por *replacement*.
  - Posiblemente emitr código como efecto de *action*.
  - Colapsar el árbol paulatinamente con las reescrituras.
  - Repetir hasta que reste solamente un nodo.

"Ciertas restricciones aplican" –
¿qué quiere decir "coincidir"?

UNIVERSIDAD SIMÓN BOLÍVAR

Reescribiendo un árbol

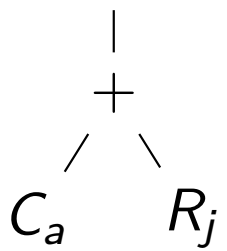


- Cuando el árbol de entrada contiene un subárbol que coincide:
  1. Se poda (*prune*) el subárbol.
  2. Se cuelga el reemplazo en su lugar.
  3. Se emite el código según la acción.
- El proceso se conoce como *subtree tiling*.
- Habrá un conflicto si más de una plantilla coincide – implantaremos algunos mecanismos para decidir.

UNIVERSIDAD SIMÓN BOLÍVAR

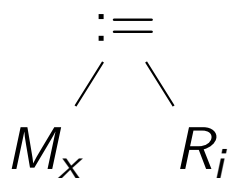
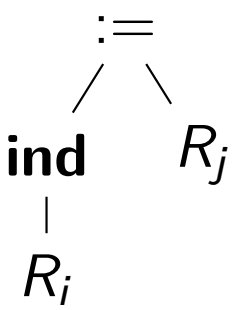
Esquema de Traducción con Arboles

Instrucciones de Carga

Regla	Reemplazo	$\leftarrow$	Patrón	Acción
L1	R_i	$\leftarrow$	C_a	{ LD R_i, $\#a$ }
L2	R_i	$\leftarrow$	M_x	{ LD R_i, x }
L3	R_i	$\leftarrow$		{ LD R_i, $a(R_j)$ }

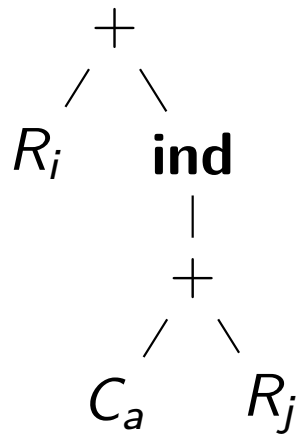
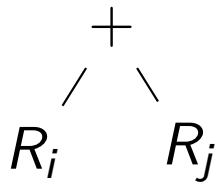
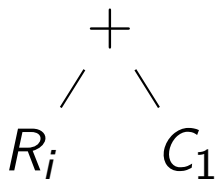
Esquema de Traducción con Arboles

Instrucciones de Almacenamiento

Regla	Reemplazo $\leftarrow$	Patrón	Acción
S1	$M \leftarrow$		{ ST x, R$_i$ }
S2	$M \leftarrow$		{ ST *R$_i$, R$_j$ }

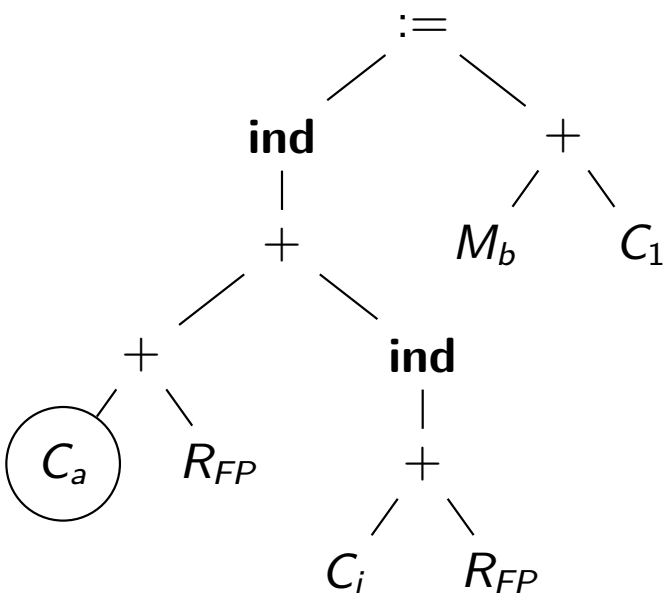
Esquema de Traducción con Arboles

Instrucciones para Operaciones

Regla	Reemplazo ← Patrón	Acción
O1	$R_i \leftarrow$ 	{ ADD R_i, R_i, $a(R_j)$ }
O2	$R_i \leftarrow$ 	{ ADD R_i, R_i, R_j }
O3	$R_i \leftarrow$ 	{ INC R_i }

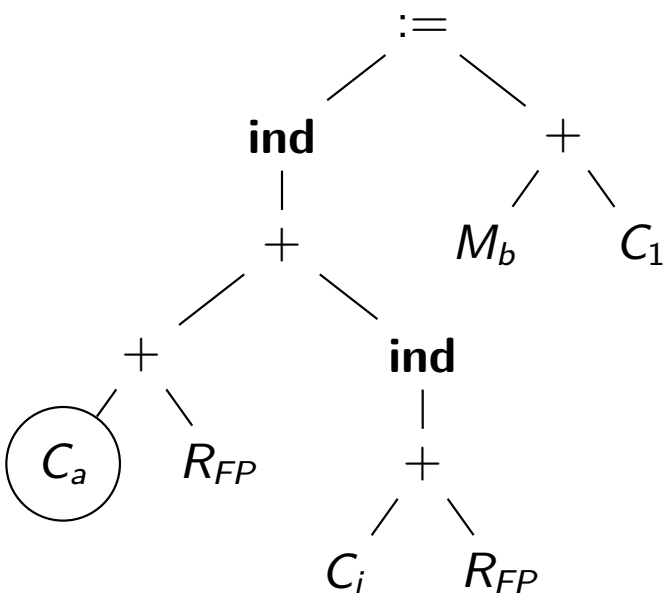
Tiling aplicado a nuestro ejemplo

Paso a Paso – *Tile* regla L1



Tiling aplicado a nuestro ejemplo

Paso a Paso – *Tile* regla L1

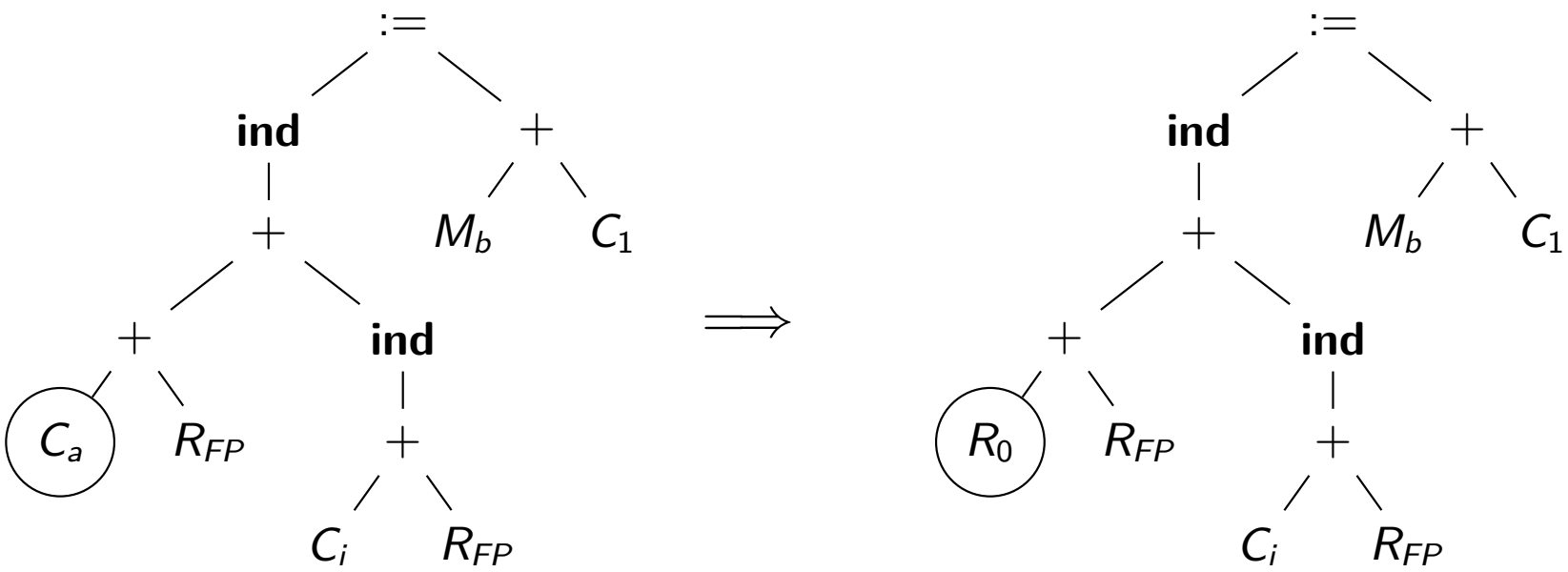


$$R_0 \leftarrow C_a \quad \{ \texttt{LD R0, \#a} \}$$

UNIVERSIDAD SIMÓN BOLÍVAR

Tiling aplicado a nuestro ejemplo

Paso a Paso – *Tile* regla L1

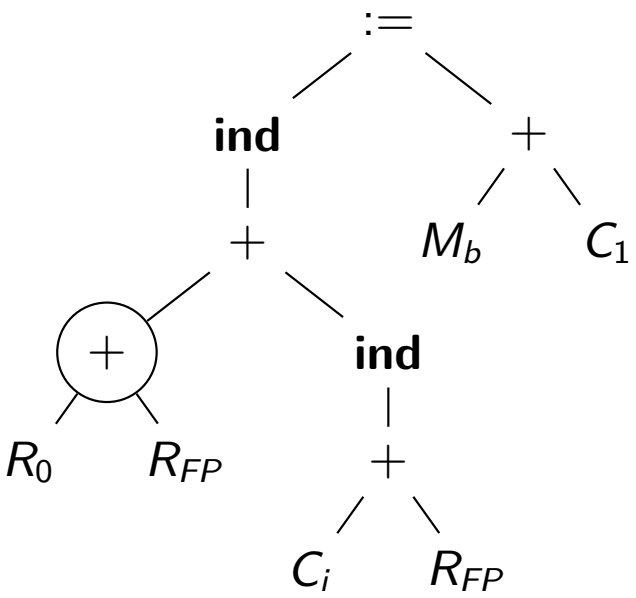


$$R_0 \leftarrow C_a \quad \{\ \texttt{LD R0, \#a}\ \}$$

UNIVERSIDAD SIMÓN BOLÍVAR

Tiling aplicado a nuestro ejemplo

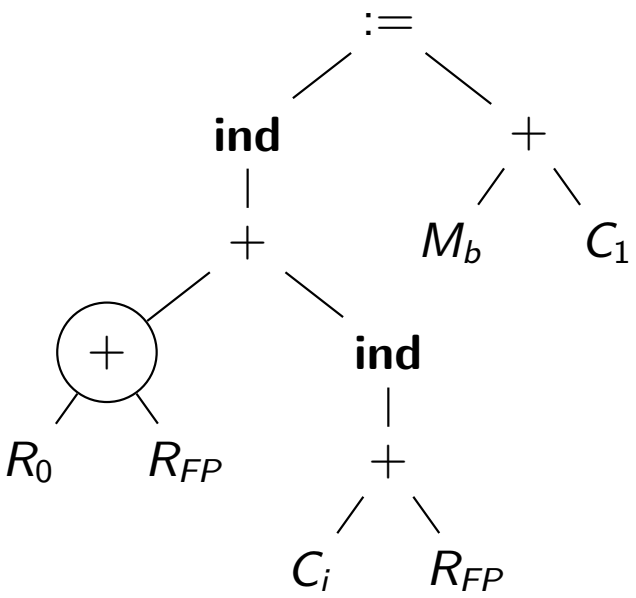
Paso a Paso – *Tile* regla O2

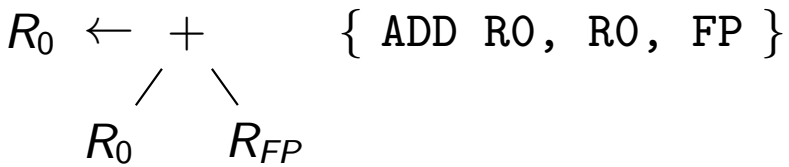


UNIVERSIDAD SIMÓN BOLÍVAR

Tiling aplicado a nuestro ejemplo

Paso a Paso – *Tile* regla O2

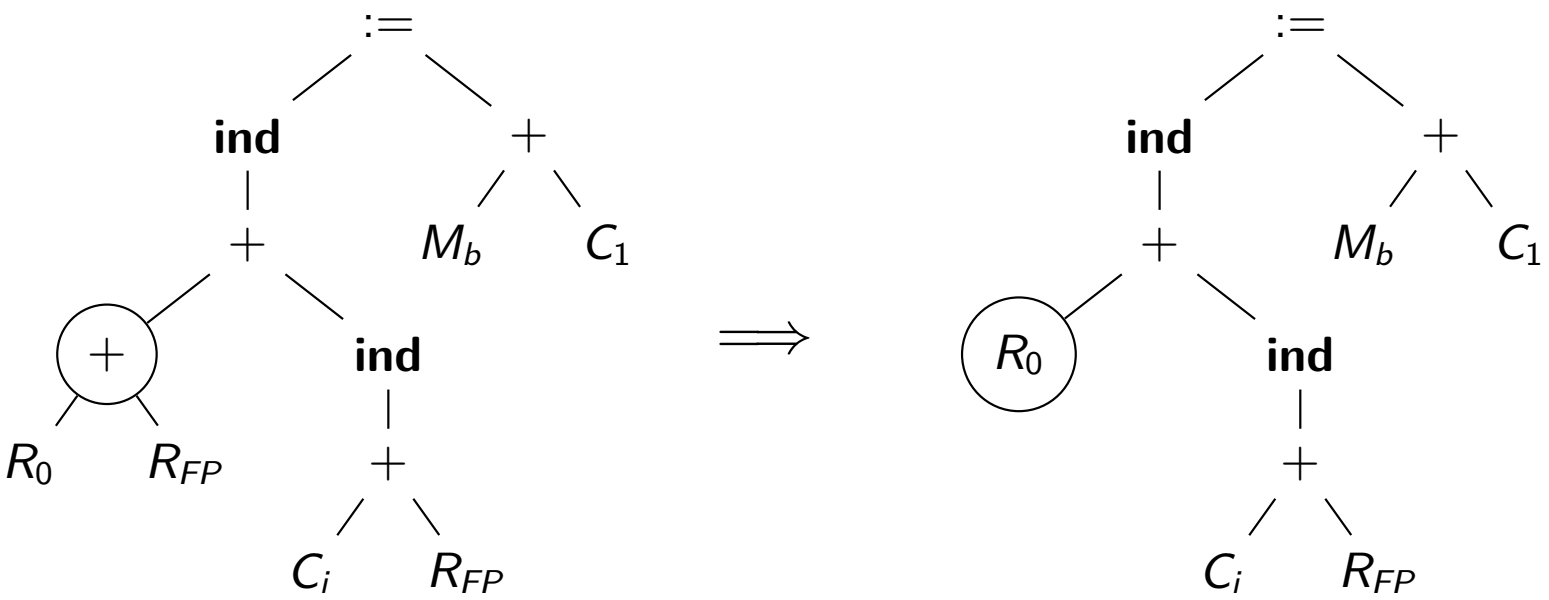


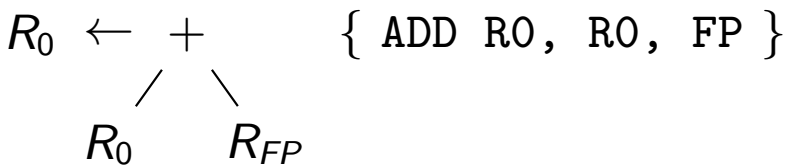


UNIVERSIDAD SIMÓN BOLÍVAR

Tiling aplicado a nuestro ejemplo

Paso a Paso – *Tile* regla O2

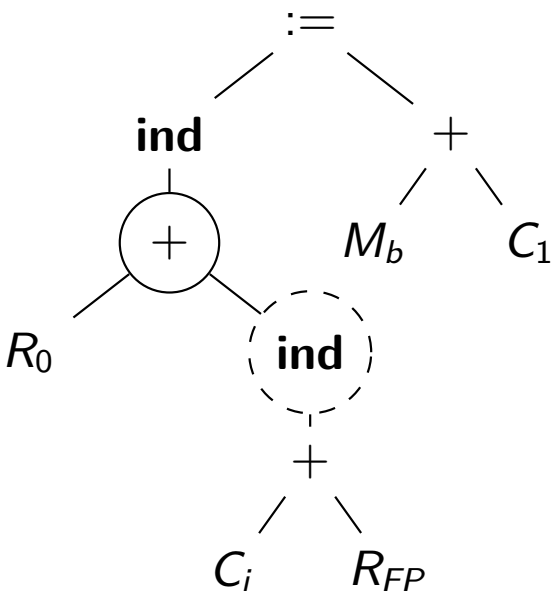




UNIVERSIDAD SIMÓN BOLÍVAR

Tiling aplicado a nuestro ejemplo

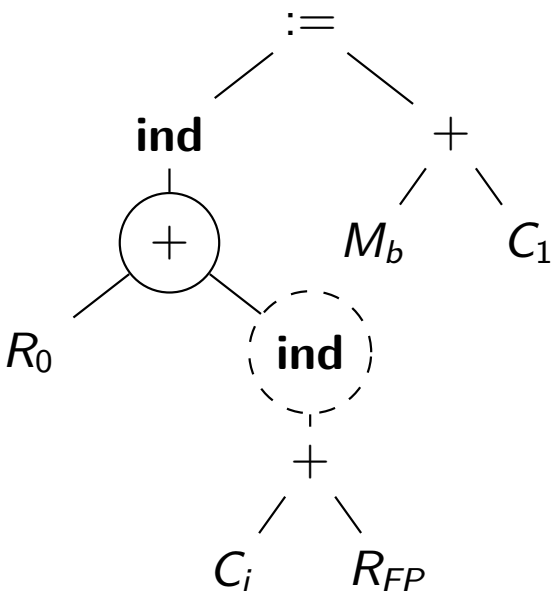
Paso a Paso – *Tile* O1 antes que L3 – "poda más"

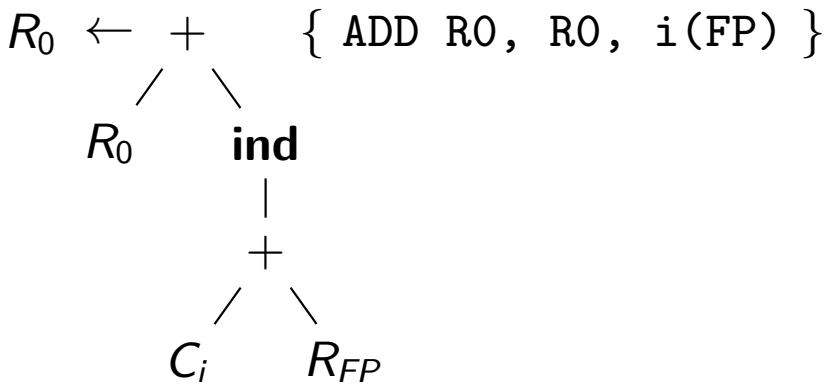


UNIVERSIDAD SIMÓN BOLÍVAR

Tiling aplicado a nuestro ejemplo

Paso a Paso – *Tile* O1 antes que L3 – "poda más"

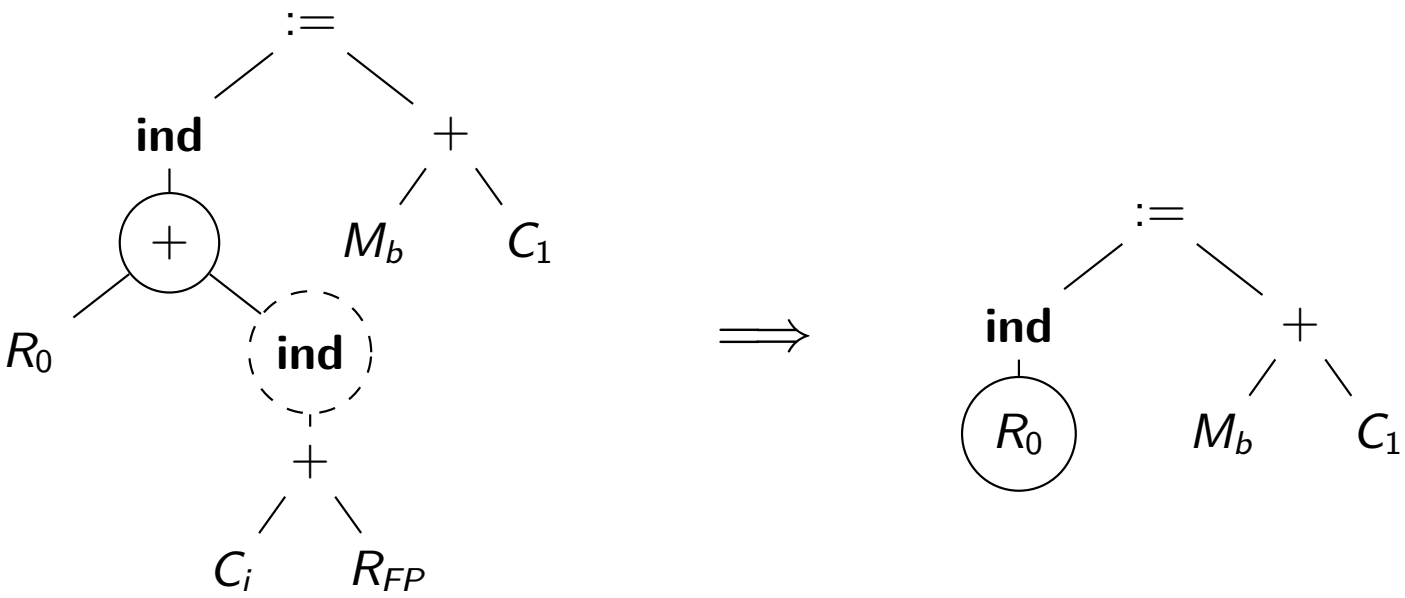


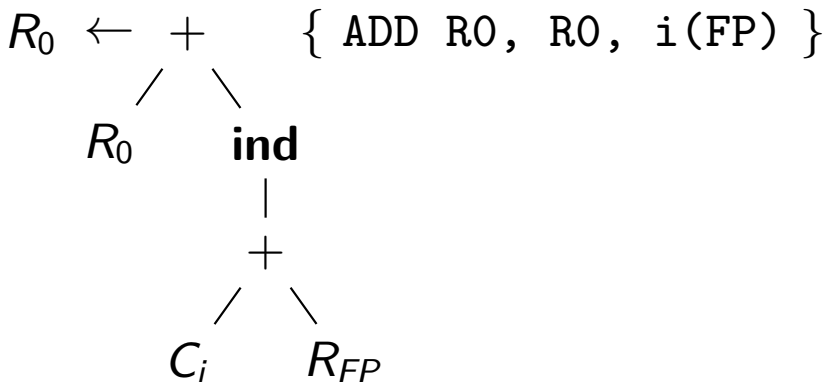


UNIVERSIDAD SIMÓN BOLÍVAR

Tiling aplicado a nuestro ejemplo

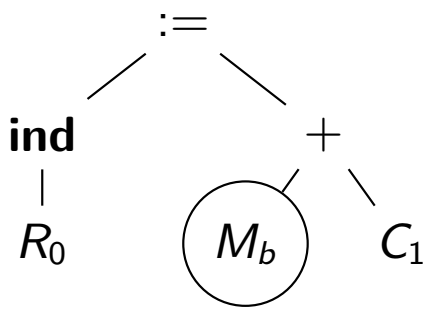
Paso a Paso – *Tile* O1 antes que L3 – "poda más"





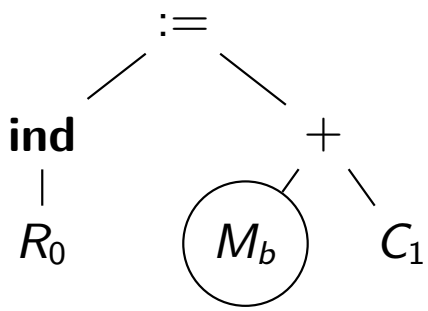
Tiling aplicado a nuestro ejemplo

Paso a Paso – *Tile* regla L2



Tiling aplicado a nuestro ejemplo

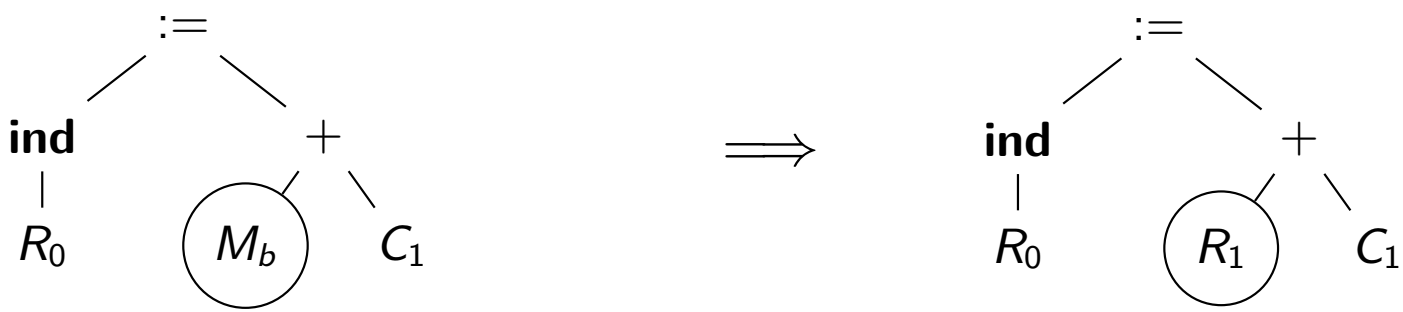
Paso a Paso – *Tile* regla L2



$$R_1 \leftarrow M_b \quad \{ \texttt{LD R1, b} \}$$

Tiling aplicado a nuestro ejemplo

Paso a Paso – *Tile* regla L2

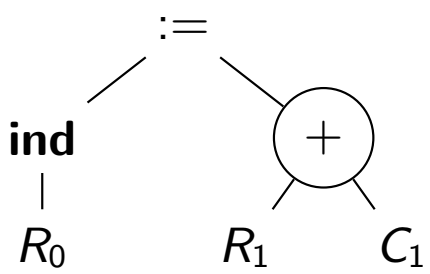


$$R_1 \leftarrow M_b \quad \{ \texttt{LD R1, b} \}$$

UNIVERSIDAD SIMÓN BOLÍVAR

Tiling aplicado a nuestro ejemplo

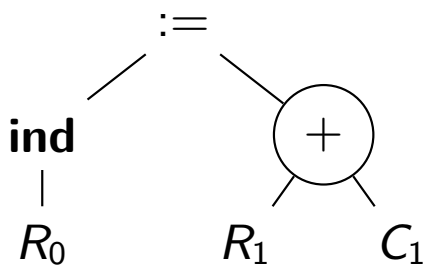
Paso a Paso – *Tile* regla O3

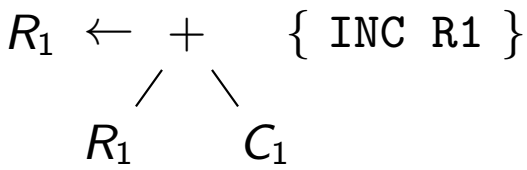


UNIVERSIDAD SIMÓN BOLÍVAR

Tiling aplicado a nuestro ejemplo

Paso a Paso – *Tile* regla O3

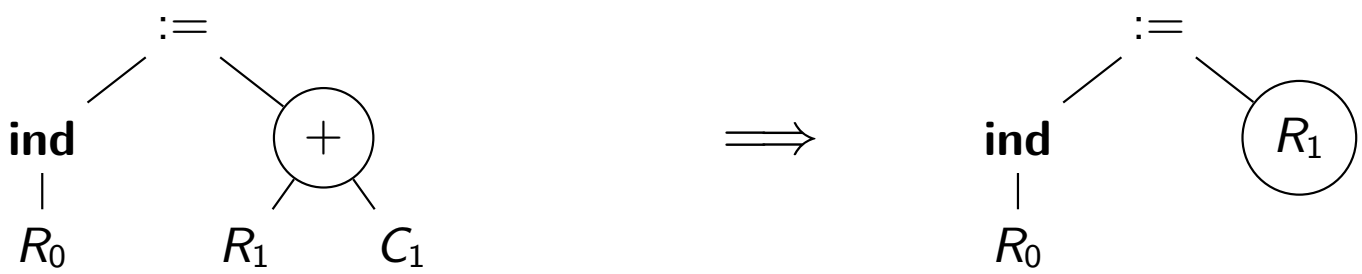


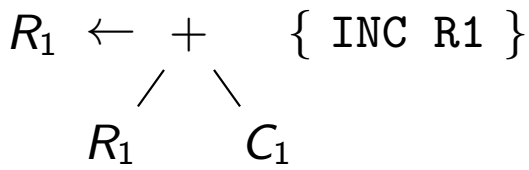


UNIVERSIDAD SIMÓN BOLÍVAR

Tiling aplicado a nuestro ejemplo

Paso a Paso – *Tile* regla O3

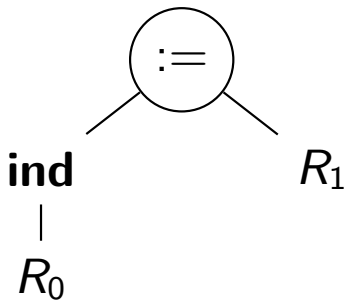




UNIVERSIDAD SIMÓN BOLÍVAR

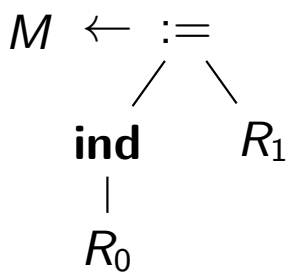
Tiling aplicado a nuestro ejemplo

Paso a Paso – *Tile* regla S2



$\Longrightarrow$

```
LD    R0, #a
ADD   R0, R0, FP
ADD   R0, R0, i(FP)
LD    R1, b
INC   R1
ST   *R0, R1
```



{ `ST *R0, R1` }

Refinación del proceso

¿Cómo encontrar el patrón?

- Generación será eficiente si ubica candidatos eficientemente.

Refinación del proceso

¿Cómo encontrar el patrón?

- Generación será eficiente si ubica candidatos eficientemente.
- Si no se encuentra ningún patrón, se bloquea el proceso.
  - No pasa si hay suficientes reglas.
  - Debe haber al menos una regla por cada operador.

UNIVERSIDAD SIMÓN BOLÍVAR

Refinación del proceso

¿Cómo encontrar el patrón?

- Generación será eficiente si ubica candidatos eficientemente.
- Si no se encuentra ningún patrón, se bloquea el proceso.
  - No pasa si hay suficientes reglas.
  - Debe haber al menos una regla por cada operador.
- Si más de un patrón aplica, ¿cuál reemplazo escoger? – diferentes secuencias de sustitución podrían generar código notablemente diferente.

UNIVERSIDAD SIMÓN BOLÍVAR

Refinación del proceso

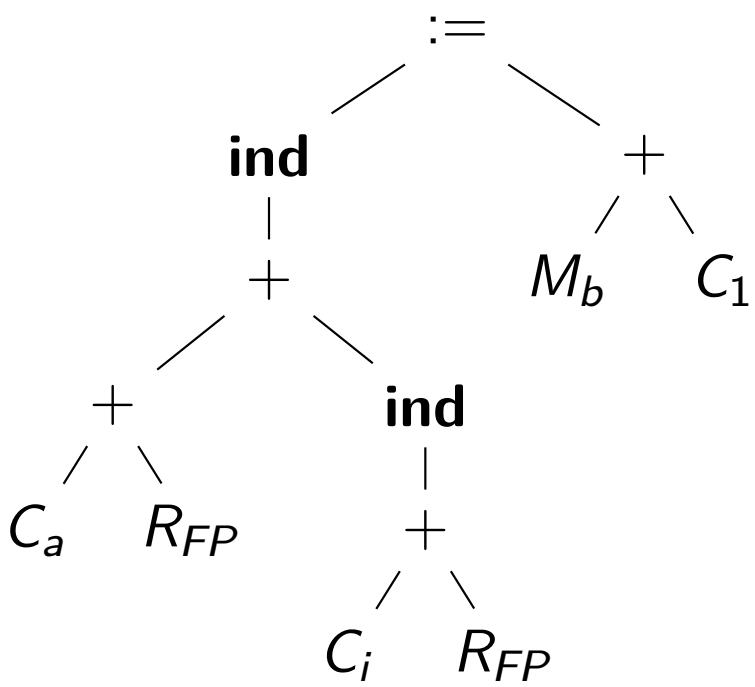
¿Cómo encontrar el patrón?

- Generación será eficiente si ubica candidatos eficientemente.
- Si no se encuentra ningún patrón, se bloquea el proceso.
  - No pasa si hay suficientes reglas.
  - Debe haber al menos una regla por cada operador.
- Si más de un patrón aplica, ¿cuál reemplazo escoger? – diferentes secuencias de sustitución podrían generar código notablemente diferente.
- ¿Cómo evitar una secuencia infinita de sustituciones?

Vamos a suponer que hay suficientes registros para calcular cada subárbol (quizás con *spill*)

UNIVERSIDAD SIMÓN BOLÍVAR

¿Qué tal si el árbol...



...es serializado a su forma prefija

$:=$ **ind** $+ + C_a\ R_{FP}$ **ind** $+ C_i\ R_{FP} + M_b\ C_1$

y lo tratamos como una *secuencia de tokens*?

UNIVERSIDAD SIMÓN BOLÍVAR

Generación de código via un reconocedor

Traductor dirigido por sintaxis *LR*

R_i	$\rightarrow$	$\mathbf{c}_a$	{ LD R_i, #a }
R_i	$\rightarrow$	M_x	{ LD R_i, x }
M	$\rightarrow$	:= M_x R_i	{ ST x, R_i }
M	$\rightarrow$	:= **ind** R_i R_j	{ ST *R_i, R_j }
R_i	$\rightarrow$	**ind** + c_a R_j	{ LD R_i, a(R_j) }
R_i	$\rightarrow$	+ R_i **ind** + $\mathbf{c}_a$ R_j	{ ADD R_i, R_i, a(R_j) }
R_i	$\rightarrow$	+ R_i R_j	{ ADD R_i, R_i, R_j }
R_i	$\rightarrow$	+ R_i $\mathbf{c}_1$	{ INC R_i }
R	$\rightarrow$	**fp**	
M	$\rightarrow$	**m**	

- Terminal **m** – dirección de memoria de algún nombre.
- Terminal **fp** – usos del registro `FP`.
- Terminal **c** – constantes.

UNIVERSIDAD SIMÓN BOLÍVAR

No tan rápido...

- La gramática de traducción tendrá *muchas* ambiguedades.
  - Resolver conflictos usando información de costos.
  - Si la información es insuficiente, preferir reducciones más largas.
  - Resultan bastante «artísticos» – pero así es la generación de código.

UNIVERSIDAD SIMÓN BOLÍVAR

No tan rápido...

- La gramática de traducción tendrá *muchas* ambiguedades.
  - Resolver conflictos usando información de costos.
  - Si la información es insuficiente, preferir reducciones más largas.
  - Resultan bastante «artísticos» – pero así es la generación de código.
- Ventajas
  - *LR* es eficiente y confiable – generadores prácticos y rápidos.
  - Relativamente fácil cambiar a otra arquitectura destino.
  - *Peephole* se reduce a incluir reglas especiales.

UNIVERSIDAD SIMÓN BOLÍVAR

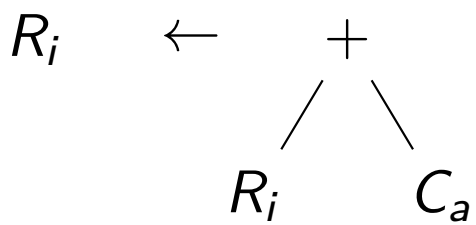
No tan rápido...

- La gramática de traducción tendrá *muchas* ambiguedades.
  - Resolver conflictos usando información de costos.
  - Si la información es insuficiente, preferir reducciones más largas.
  - Resultan bastante «artísticos» – pero así es la generación de código.
- Ventajas
  - *LR* es eficiente y confiable – generadores prácticos y rápidos.
  - Relativamente fácil cambiar a otra arquitectura destino.
  - *Peephole* se reduce a incluir reglas especiales.
- Desventajas
  - Expresiones *siempre* se evalúan de izquierda a derecha.
  - Arquitecturas con muchos modos de direccionamiento producen reconocedores muy grandes.
  - *Backtracking* o *GLR* para manejar decisiones erróneas en conflictos – evitar bloqueo y ciclos infinitos en el reconocedor.

UNIVERSIDAD SIMÓN BOLÍVAR

Explotar las verificaciones semánticas

- Especificar restricciones sobre los atributos como predicados en las acciones de contexto.
- Consolidar instrucciones con estructuras similares en menos reglas – simplificar un poco la gramática.



$\{$

if $a == 1$ **then**

INC R_i

else

ADD R_i, R_i, #a

end if

$\}$

- Resolver conflictos de la gramática combinando reglas y agregando predicados – difícil verificar la precisión de la traducción.

UNIVERSIDAD SIMÓN BOLÍVAR

Patrones generalizados para árboles

- Representación prefija obliga a evaluar desde la izquierda – representación postfija obliga a evaluar desde la derecha.

Patrones generalizados para árboles

- Representación prefija obliga a evaluar desde la izquierda – representación postfija obliga a evaluar desde la derecha.
- ¡Convertir cada plantilla en un conjunto de cadenas!
  - Eliminar los subíndices – el valor no es necesario para el patrón, solamente el atributo.
  - Cada camino de raíz hasta hoja es una cadena.
  - Incluyen indicadores de posición para hijos, de izquierda a derecha.

Patrones generalizados para árboles

- Representación prefija obliga a evaluar desde la izquierda – representación postfija obliga a evaluar desde la derecha.
- ¡Convertir cada plantilla en un conjunto de cadenas!
  - Eliminar los subíndices – el valor no es necesario para el patrón, solamente el atributo.
  - Cada camino de raíz hasta hoja es una cadena.
  - Incluyen indicadores de posición para hijos, de izquierda a derecha.
- Escribir un reconocedor *ad-hoc*
  - Método *top-down*.
  - Identificar prefijos comunes sobre el conjuntos de cadenas – exclusión rápida de caminos imposibles.
  - Técnicas de reconocimiento de cadenas – Rabin-Karp, Aho-Korasick, regexes ...

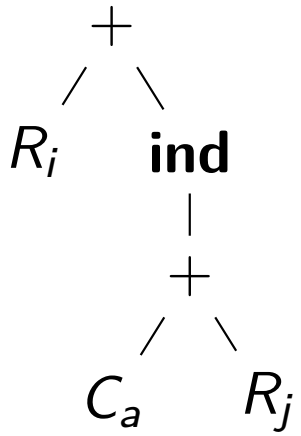
UNIVERSIDAD SIMÓN BOLÍVAR

Tree «Stringification»

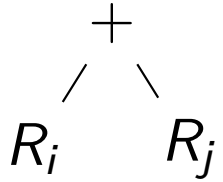
De árboles a conjunto de cadenas

$R_i \leftarrow C_a$

$R_i \leftarrow$



$R_i \leftarrow$



C	El único con C en la raíz.
$+\ 1\ R$	Los *dos* con hijo izquierdo R.
$+\ 2$ **ind** $1 + 1\ C$	
$+\ 2$ **ind** $1 + 2\ R$	
$+\ 2\ R$	Los *dos* con hijo derecho R.

UNIVERSIDAD SIMÓN BOLÍVAR

Generación óptima para una expresión

Un caso borde frecuente

- Se cuenta con un número *fijo* de registros asignados a la generación de código para expresiones del lenguaje.
- Suponemos suficiente generar código *una* expresión a la vez.
  - Porque el bloque contiene una sola expresión.
  - Porque nos ocuparemos una expresión por vez en un bloque que contiene varias.
- Suponemos que ya se ha completado la optimización que elimina cualquier subexpresión común.

UNIVERSIDAD SIMÓN BOLÍVAR

Generación óptima para una expresión

Un caso borde frecuente

- Se cuenta con un número *fijo* de registros asignados a la generación de código para expresiones del lenguaje.
- Suponemos suficiente generar código *una* expresión a la vez.
  - Porque el bloque contiene una sola expresión.
  - Porque nos ocuparemos una expresión por vez en un bloque que contiene varias.
- Suponemos que ya se ha completado la optimización que elimina cualquier subexpresión común.

¿Cuántos registros son necesarios para evaluar un nodo *sin* almacenar temporales?

UNIVERSIDAD SIMÓN BOLÍVAR

Números de Ershov

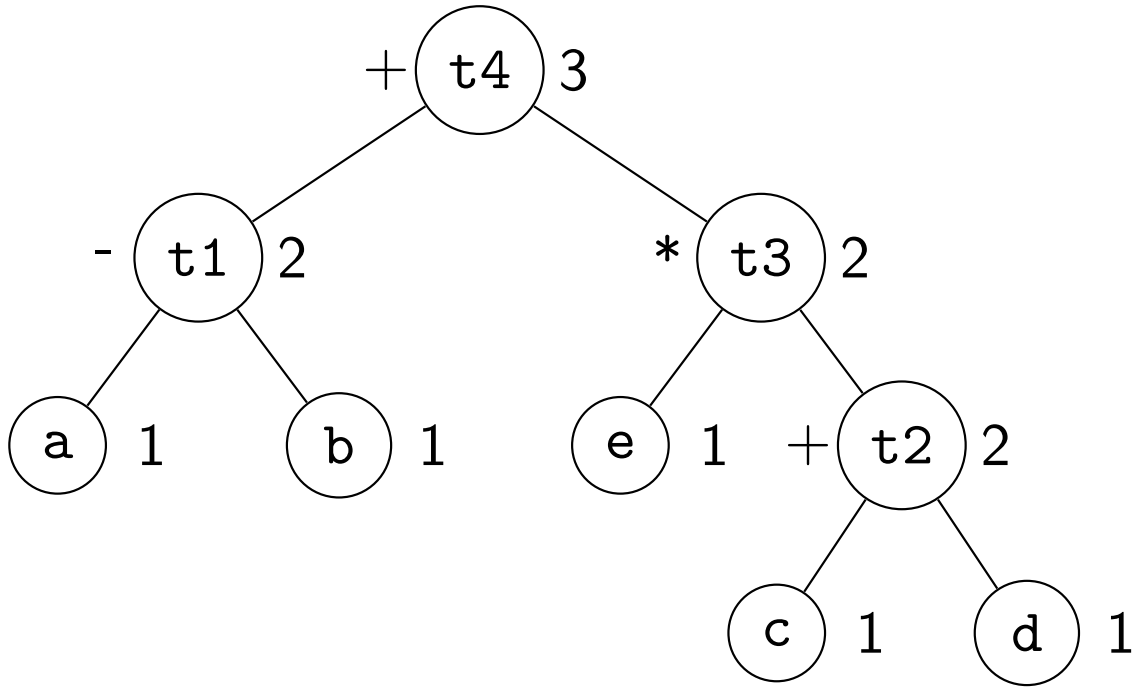
- Regla de numeración para indicar la cantidad de registros necesarios para evaluar un nodo sin almacenar temporales.
- Específica para cada modelo de máquina – para la nuestra
  - $label(n) \leftarrow 1$ – cuando n es una hoja.
  - $label(n) \leftarrow label(c)$ – cuando n tiene hijo único c.
  - Sean c_{left} y c_{right} los hijos de n
    - $label(n) \leftarrow max(label(c_{left}), label(c_{right}))$ cuando $label(c_{left}) \neq label(c_{right})$
    - $label(n) \leftarrow label(c_{left}) + 1$ cuando $label(c_{left}) = label(c_{right})$

UNIVERSIDAD SIMÓN BOLÍVAR

Etiquetado de Ershov

```
(a - b) + e * (c + d)
```

```
t1 := a - b
t2 := c + d
t3 := e * t2
t4 := t1 + t3
```



¿Cómo generar código?

Algoritmo – caso base

- Partir desde la raíz del árbol etiquetado Ershov hacia las hojas.
- Definiremos $b \geq 1$ registro "base" para generación por nodo – la raíz del árbol comienza con $b = 1$.
- Al procesar un nodo la etiqueta k indica que:
  - Se necesitan k registros para el código.
  - Serán los registros $R_b, R_{b+1}, \ldots R_{b+k-1}$.
  - El resultado debe quedar en R_{b+k-1}.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo generar código?

Algoritmo – caso base

- Partir desde la raíz del árbol etiquetado Ershov hacia las hojas.
- Definiremos $b \geq 1$ registro "base" para generación por nodo – la raíz del árbol comienza con $b = 1$.
- Al procesar un nodo la etiqueta k indica que:
  - Se necesitan k registros para el código.
  - Serán los registros $R_b, R_{b+1}, \ldots R_{b+k-1}$.
  - El resultado debe quedar en R_{b+k-1}.
- Caso Base – una hoja
  - Representa el operando `x`.
  - Siendo b la base de generación, basta generar `LD R`$_b$`, x`

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo generar código?

Algoritmo – casos recursivos

- Generación para un nodo interior con b como base de generación.

¿Cómo generar código?

Algoritmo – casos recursivos

- Generación para un nodo interior con b como base de generación.
- Caso recursivo – nodo interior con dos hijos c_l y c_r de clave idéntica
  - Sea k la etiqueta del nodo – los hijos tienen $k-1$.
  - Generar código para c_l usando base $b+1$ – resultado estará en R_{b+k-1}
  - Generar código para c_r usando base b – resultado estará en R_{b+k-2}
  - Generar `OP` $\mathtt{R}_{b+k-1}$, $\mathtt{R}_{b+k-2}$, $\mathtt{R}_{b+k-1}$

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo generar código?

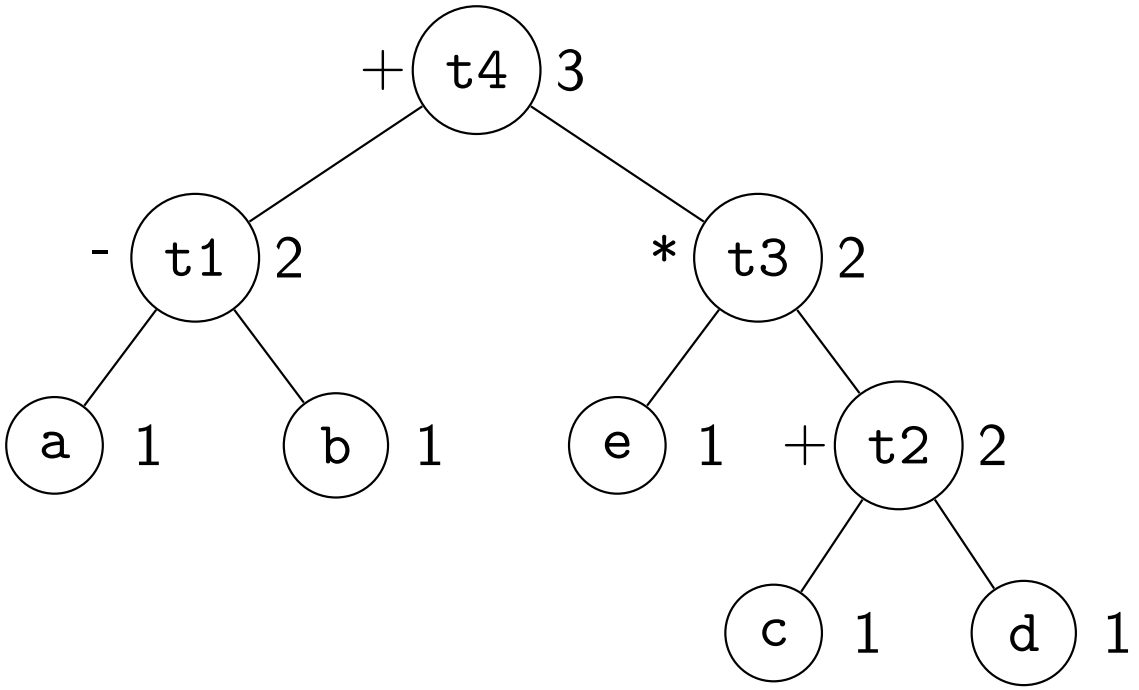
Algoritmo – casos recursivos

- Generación para un nodo interior con b como base de generación.
- Caso recursivo – nodo interior con dos hijos c_l y c_r de clave idéntica
  - Sea k la etiqueta del nodo – los hijos tienen $k-1$.
  - Generar código para c_l usando base $b+1$ – resultado estará en R_{b+k-1}
  - Generar código para c_r usando base b – resultado estará en R_{b+k-2}
  - Generar `OP` $\mathtt{R}_{b+k-1}$, $\mathtt{R}_{b+k-2}$, $\mathtt{R}_{b+k-1}$
- Caso recursivo – nodo interior con dos hijos c_b y c_s de clave diferente
  - Sean $k > m$ las etiquetas de los nodos c_b y c_s, respectivamente.
  - Generar código para c_b usando base b – resultado estará en R_{b+k-1}
  - Generar código para c_s usando base b – resultado estará en R_{b+m-1}
  - Si c_b es el hijo izquierdo, generar `OP` $\mathtt{R}_{b+k-1}$, $\mathtt{R}_{b+k-1}$, $\mathtt{R}_{b+m-1}$
  - Si c_b es el hijo derecho, generar `OP` $\mathtt{R}_{b+k-1}$, $\mathtt{R}_{b+m-1}$, $\mathtt{R}_{b+k-1}$

UNIVERSIDAD SIMÓN BOLÍVAR

Generación recursiva

Para nuestro ejemplo

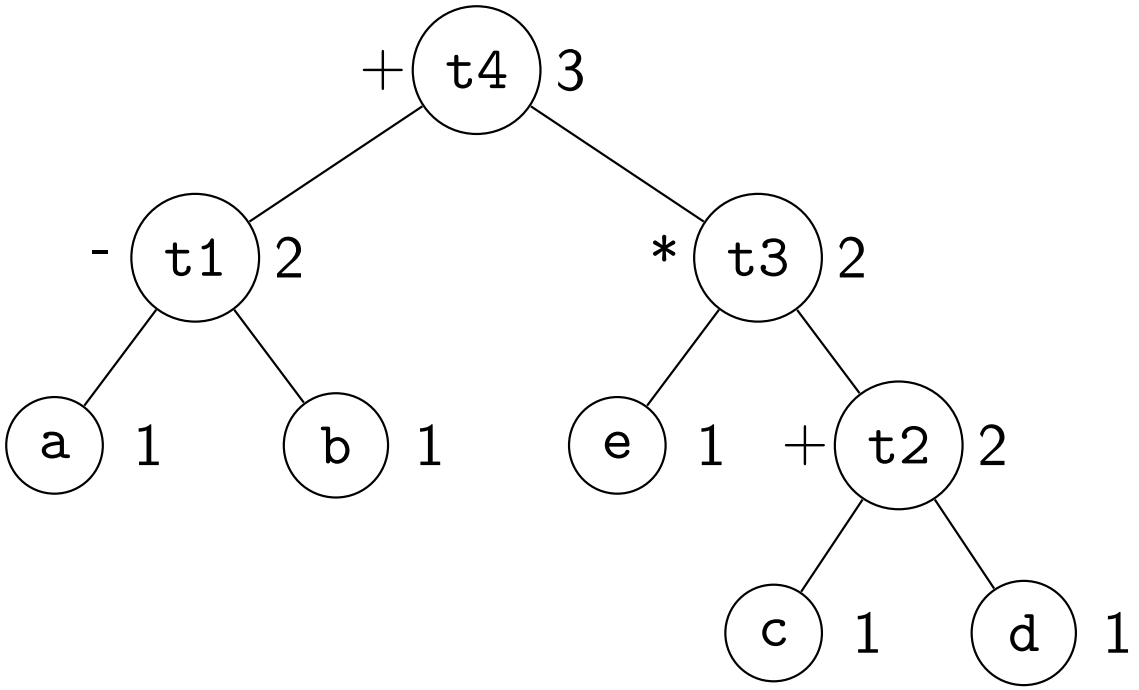


UNIVERSIDAD SIMÓN BOLÍVAR

Generación recursiva

Para nuestro ejemplo

- Raíz con etiqueta 3
  - Usará `R1`, `R2` y `R3`
  - Resultado en `R3`
  - $label(t1) = label(t3)$

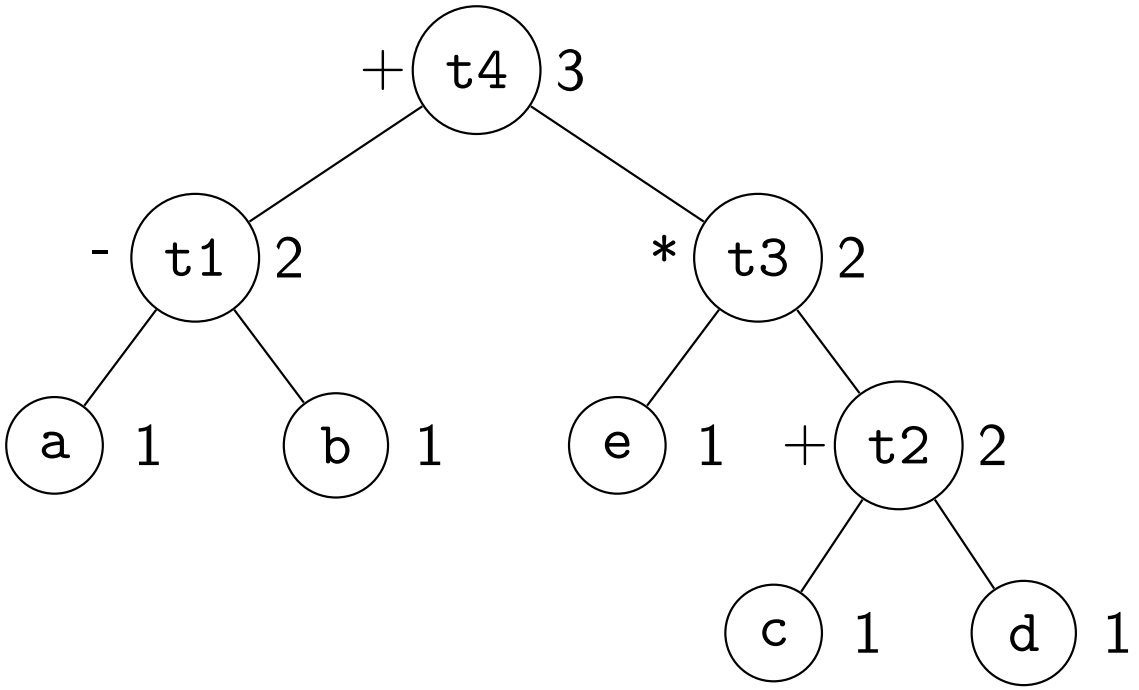


UNIVERSIDAD SIMÓN BOLÍVAR

Generación recursiva

Para nuestro ejemplo

- Raíz con etiqueta 3
  - Usará R1, R2 y R3
  - Resultado en R3
  - $label(t1) = label(t3)$
- Generar derecho (t3)
  - Con $b = 2$
  - Usará R2 y R3
  - Resultado en R3
  - $label(t2) > label(e)$

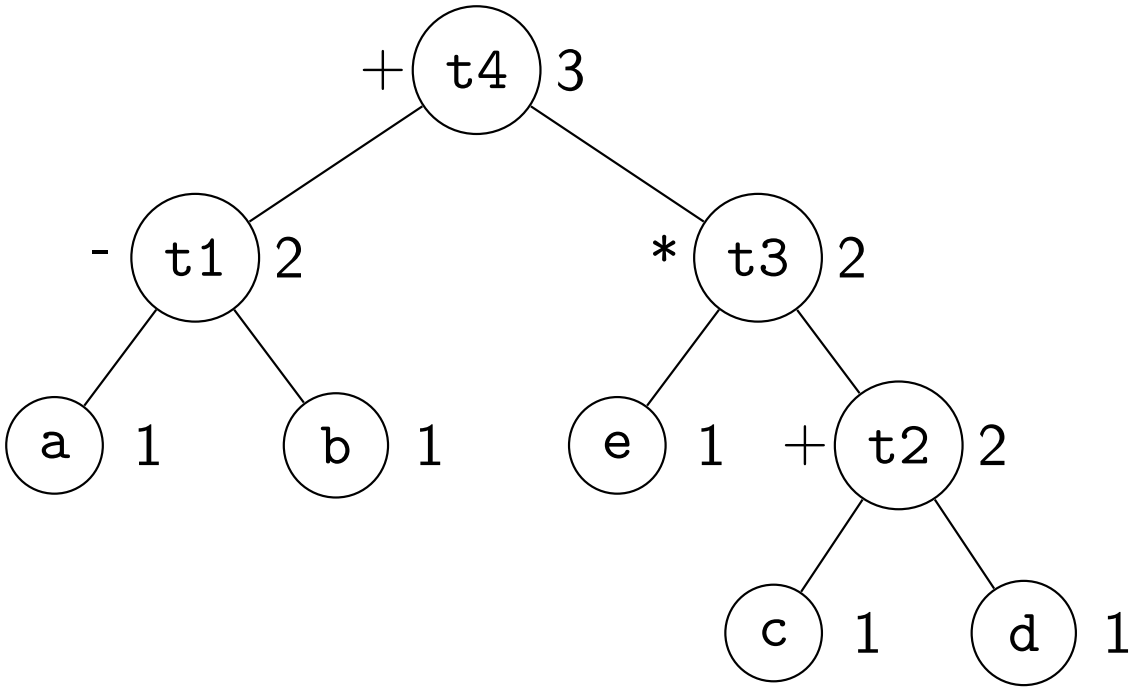


UNIVERSIDAD SIMÓN BOLÍVAR

Generación recursiva

Para nuestro ejemplo

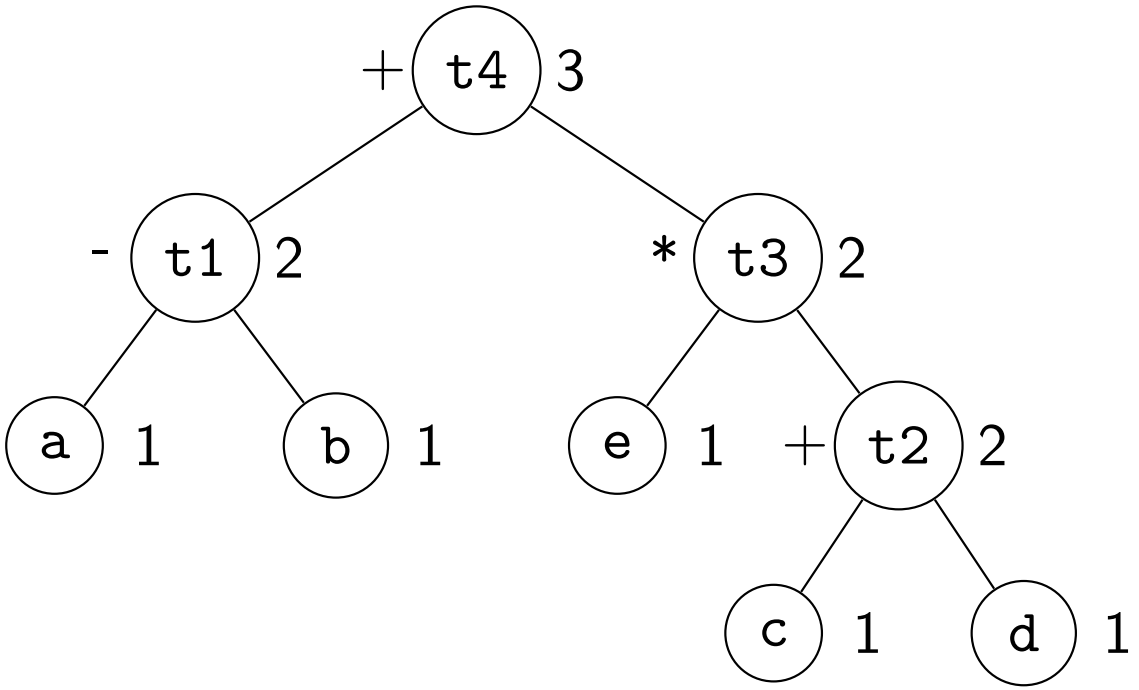
- Raíz con etiqueta 3
  - Usará R1, R2 y R3
  - Resultado en R3
  - $label(t1) = label(t3)$
- Generar derecho (t3)
  - Con $b = 2$
  - Usará R2 y R3
  - Resultado en R3
  - $label(t2) > label(e)$
- Generar derecho (t2)
  - Con $b = 2$
  - Usará R2 y R3
  - Resultado en R3



UNIVERSIDAD SIMÓN BOLÍVAR

Generación recursiva

Para nuestro ejemplo

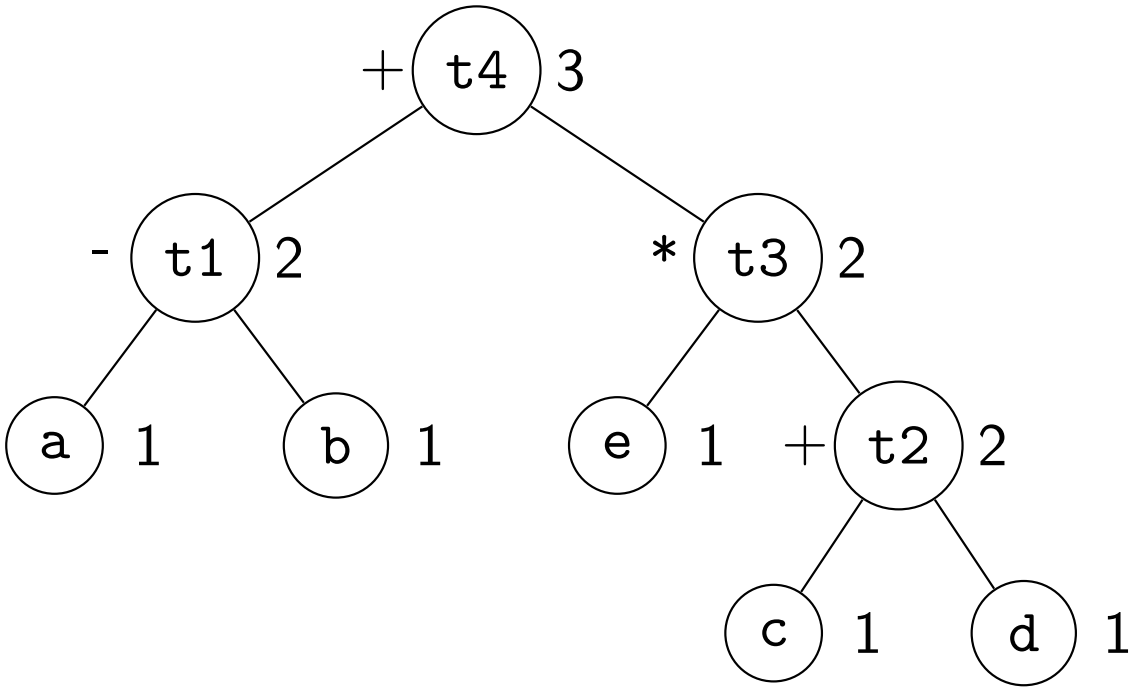


UNIVERSIDAD SIMÓN BOLÍVAR

Generación recursiva

Para nuestro ejemplo

- Generar derecho (t2)
  - Con $b = 2$
  - Usará R2 y R3
  - Resultado en R3
  - $label(c) = label(d)$



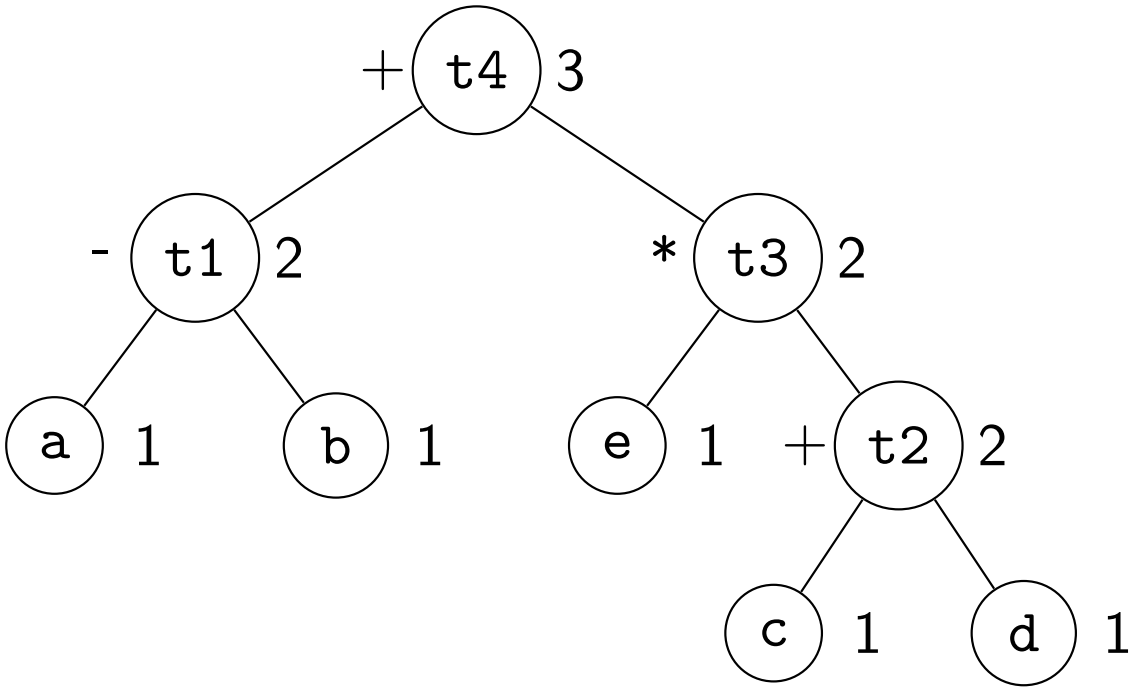
UNIVERSIDAD SIMÓN BOLÍVAR

Generación recursiva

Para nuestro ejemplo

- Generar derecho (t2)
  - Con $b = 2$
  - Usará `R2` y `R3`
  - Resultado en `R3`
  - $label(c) = label(d)$
- Hoja derecha (d)

  `LD R3, d`

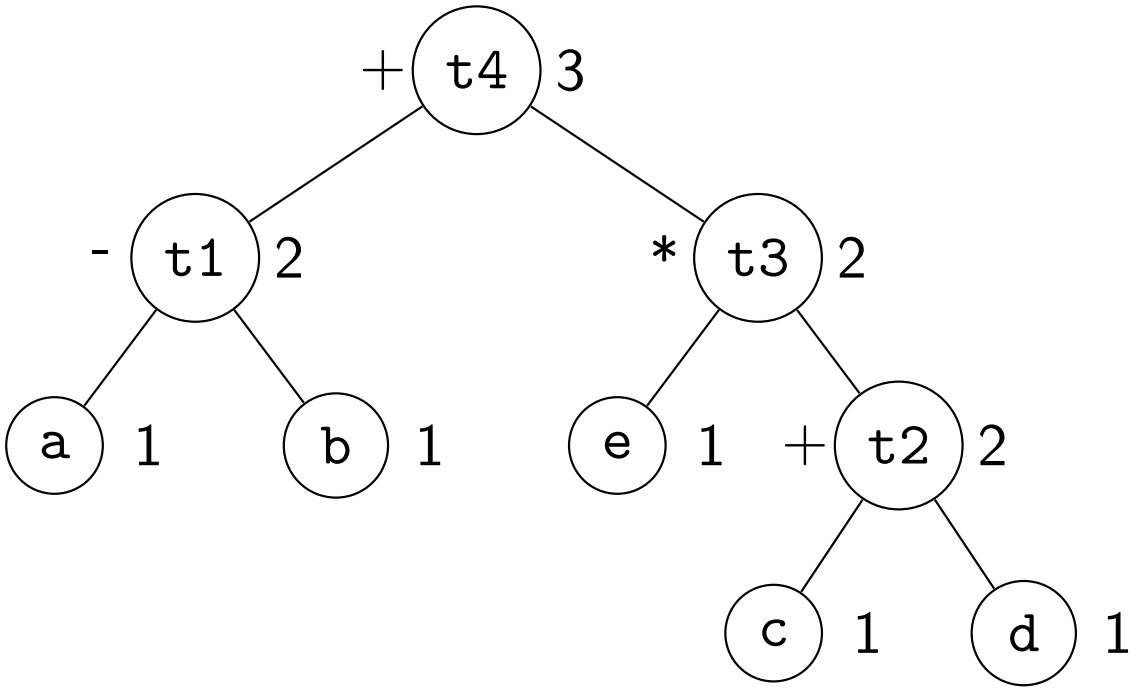


UNIVERSIDAD SIMÓN BOLÍVAR

Generación recursiva

Para nuestro ejemplo

- Generar derecho (t2)
  - Con $b = 2$
  - Usará `R2` y `R3`
  - Resultado en `R3`
  - $label(c) = label(d)$
- Hoja derecha (d)
  `LD R3, d`
- Hoja izquierda (c)
  `LD R2, c`

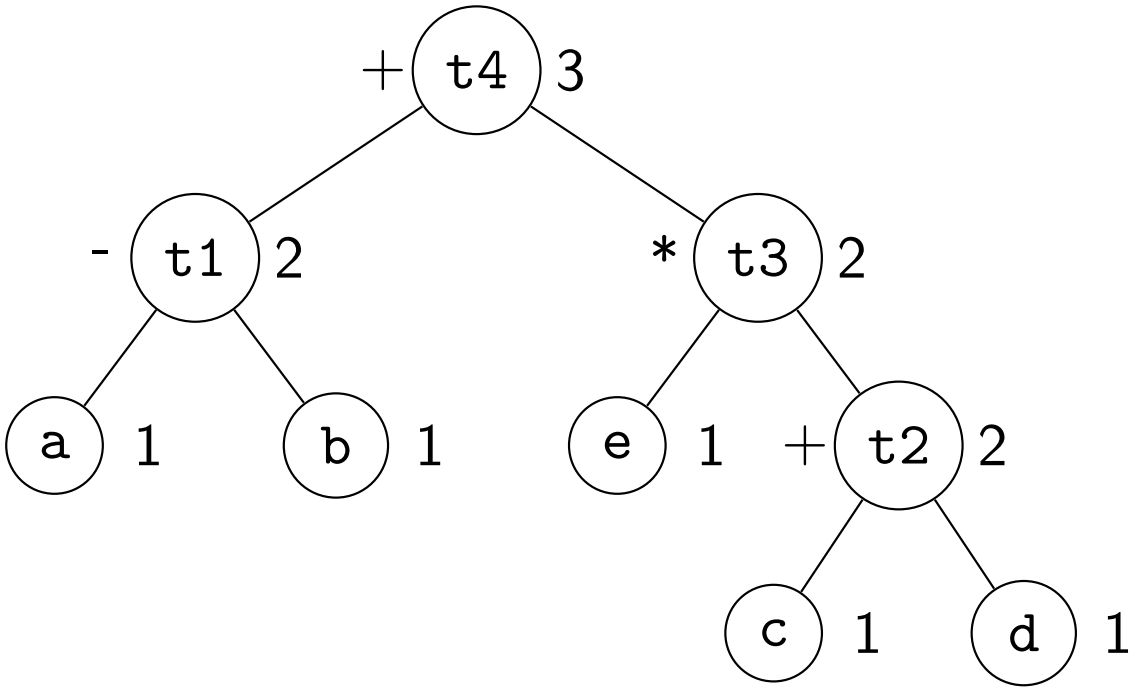


UNIVERSIDAD SIMÓN BOLÍVAR

Generación recursiva

Para nuestro ejemplo

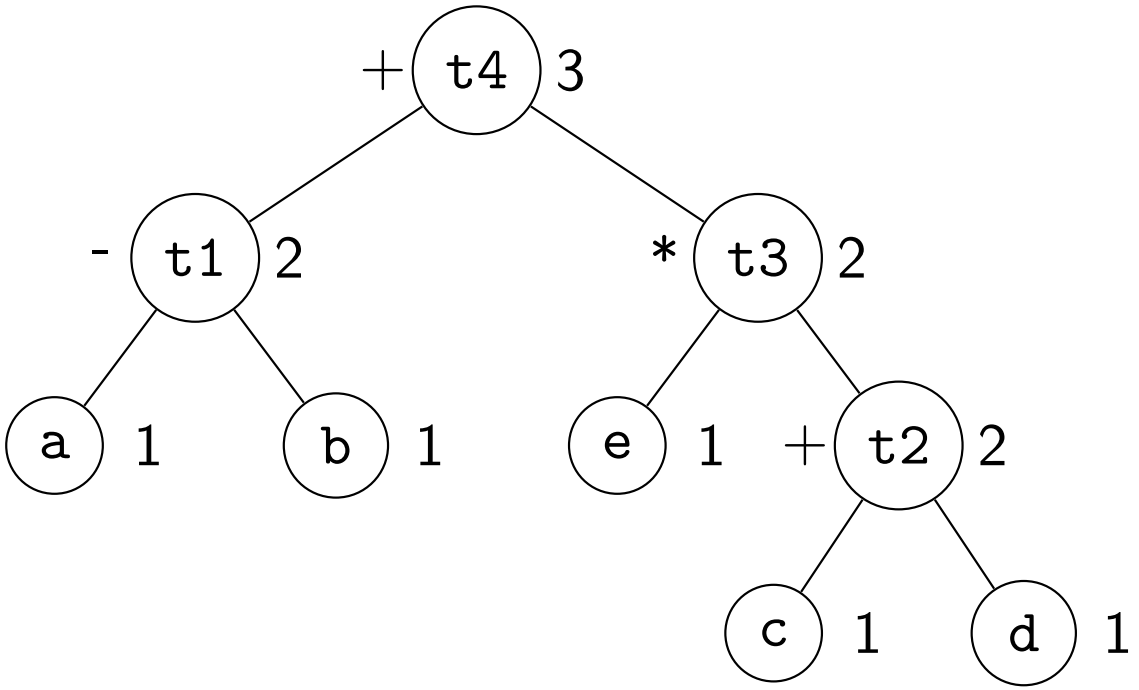
- Generar derecho (t2)
  - Con $b = 2$
  - Usará `R2` y `R3`
  - Resultado en `R3`
  - $label(c) = label(d)$
- Hoja derecha (d)
  `LD R3, d`
- Hoja izquierda (c)
  `LD R2, c`
- De nuevo en (t2)
  `ADD R3, R2, R3`



UNIVERSIDAD SIMÓN BOLÍVAR

Generación recursiva

Para nuestro ejemplo

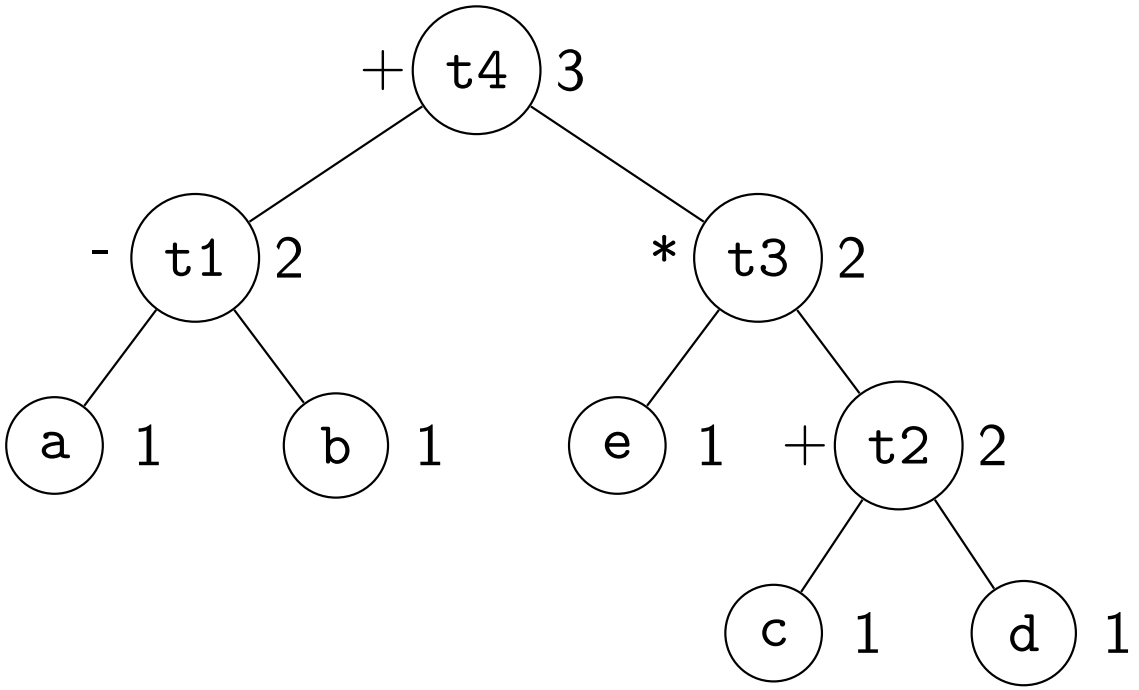


UNIVERSIDAD SIMÓN BOLÍVAR

Generación recursiva

Para nuestro ejemplo

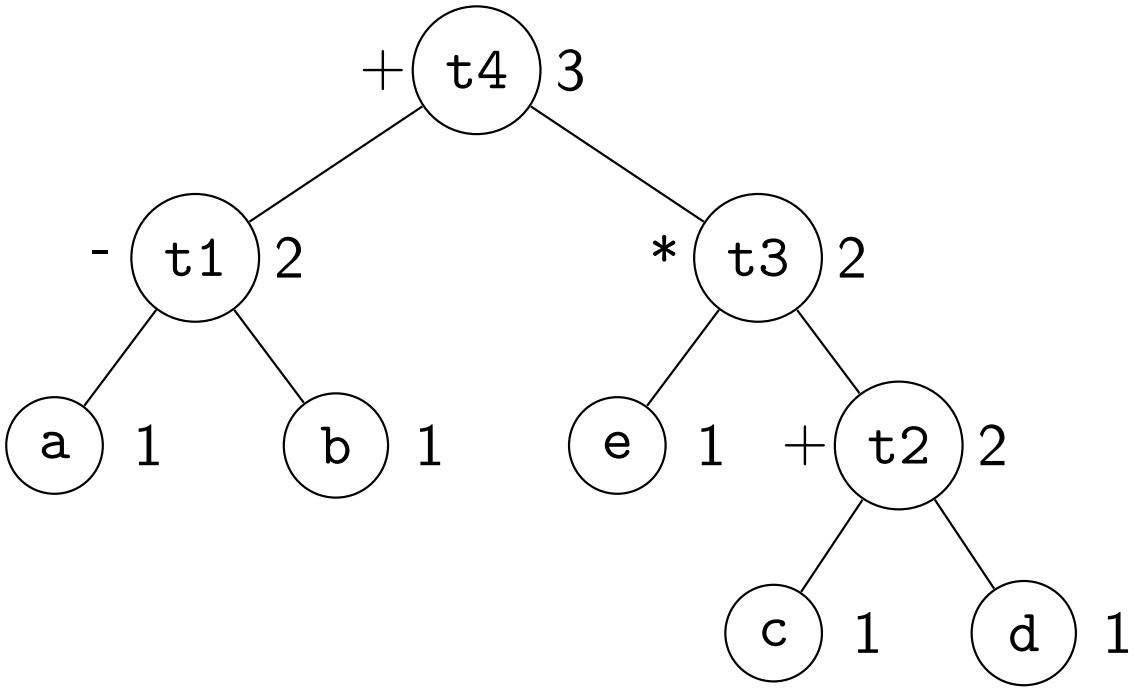
- De nuevo en (t3)
  - Tenemos $b = 2$
  - Listo hijo derecho (t2)



Generación recursiva

Para nuestro ejemplo

- De nuevo en (t3)
  - Tenemos $b = 2$
  - Listo hijo derecho (t2)
- Hoja izquierda (e)
  `LD R2, e`

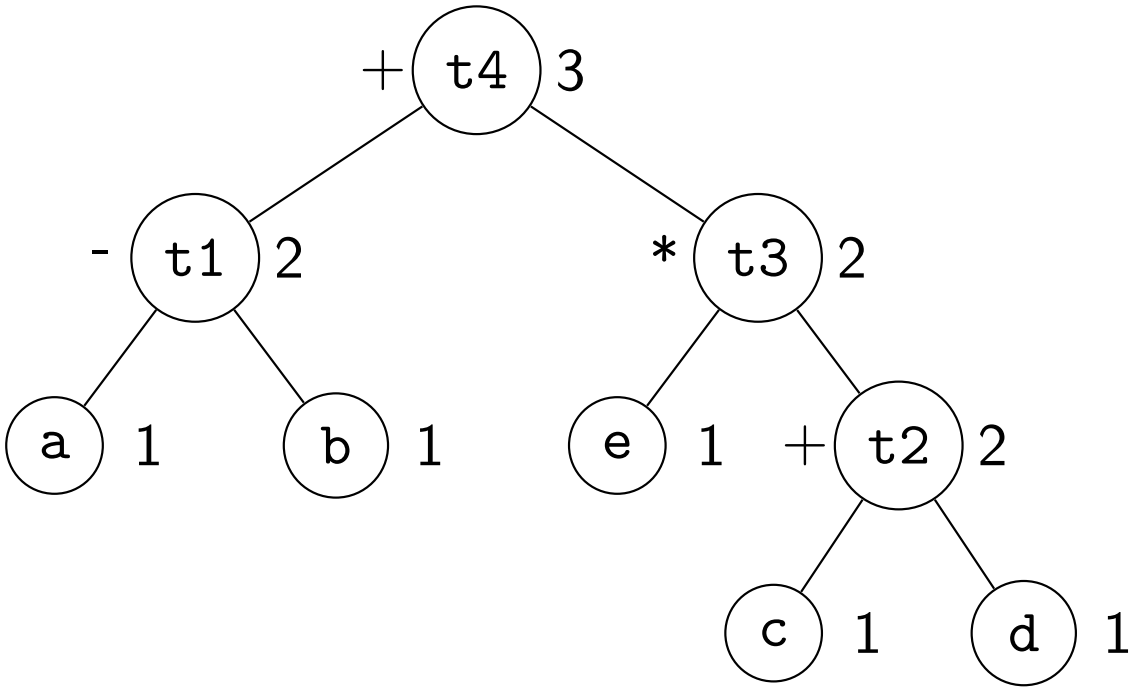


UNIVERSIDAD SIMÓN BOLÍVAR

Generación recursiva

Para nuestro ejemplo

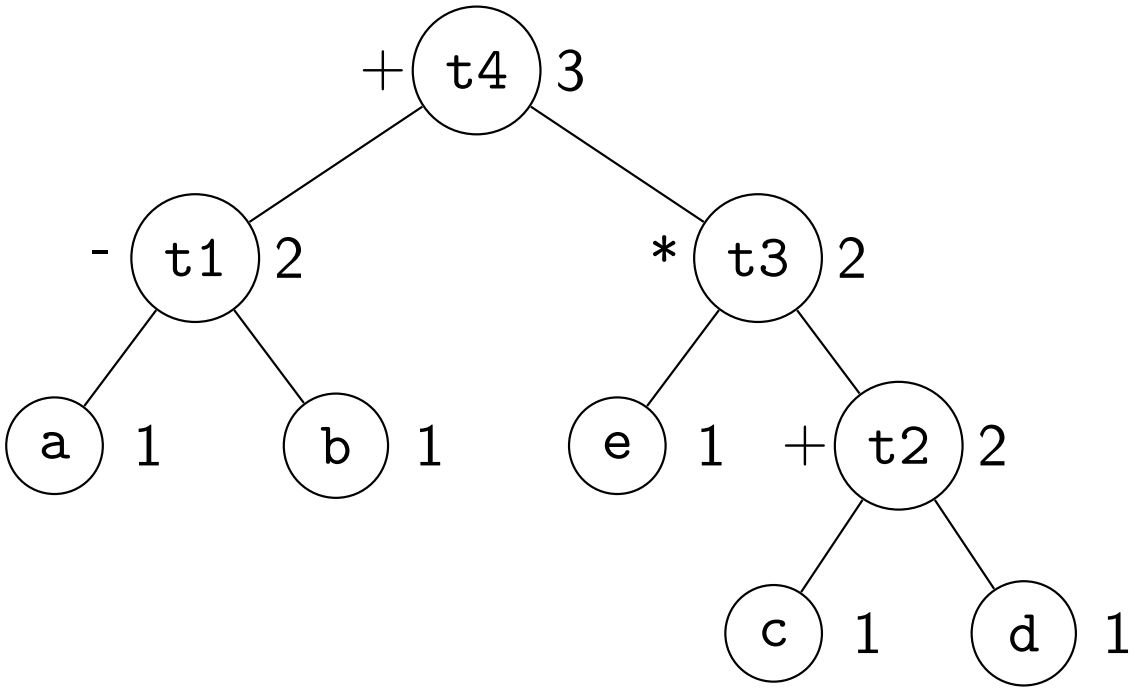
- De nuevo en (t3)
  - Tenemos $b = 2$
  - Listo hijo derecho (t2)
- Hoja izquierda (e)
  `LD R2, e`
- De nuevo en (t3)
  `MUL R3, R2, R3`



UNIVERSIDAD SIMÓN BOLÍVAR

Generación recursiva

Para nuestro ejemplo

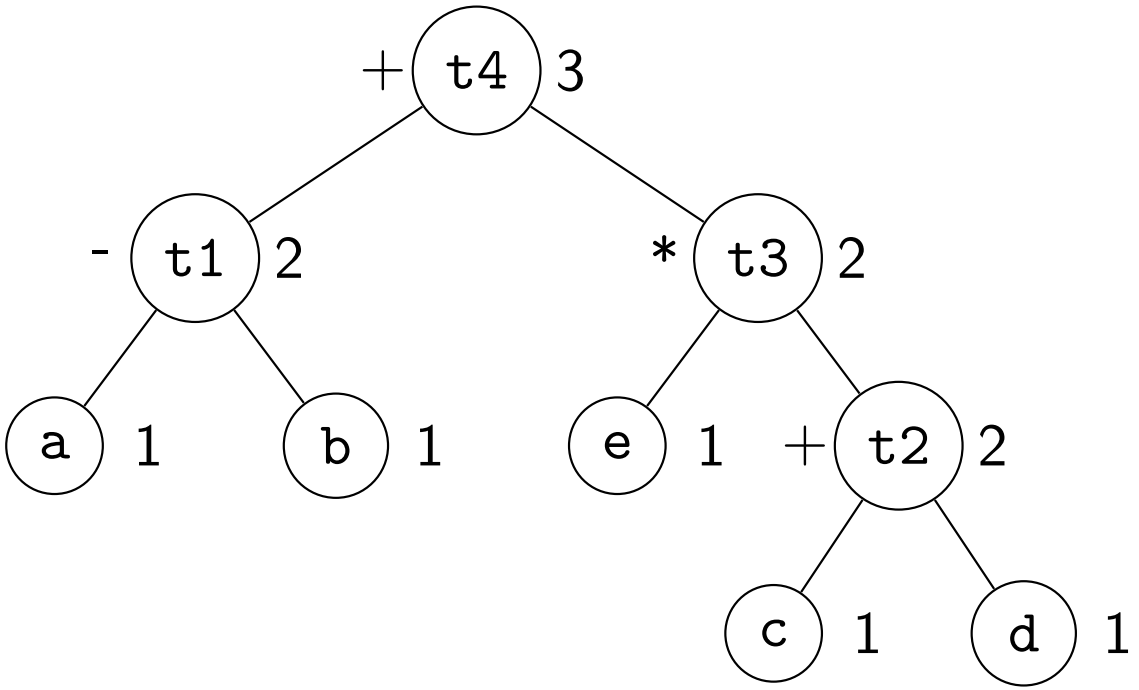


UNIVERSIDAD SIMÓN BOLÍVAR

Generación recursiva

Para nuestro ejemplo

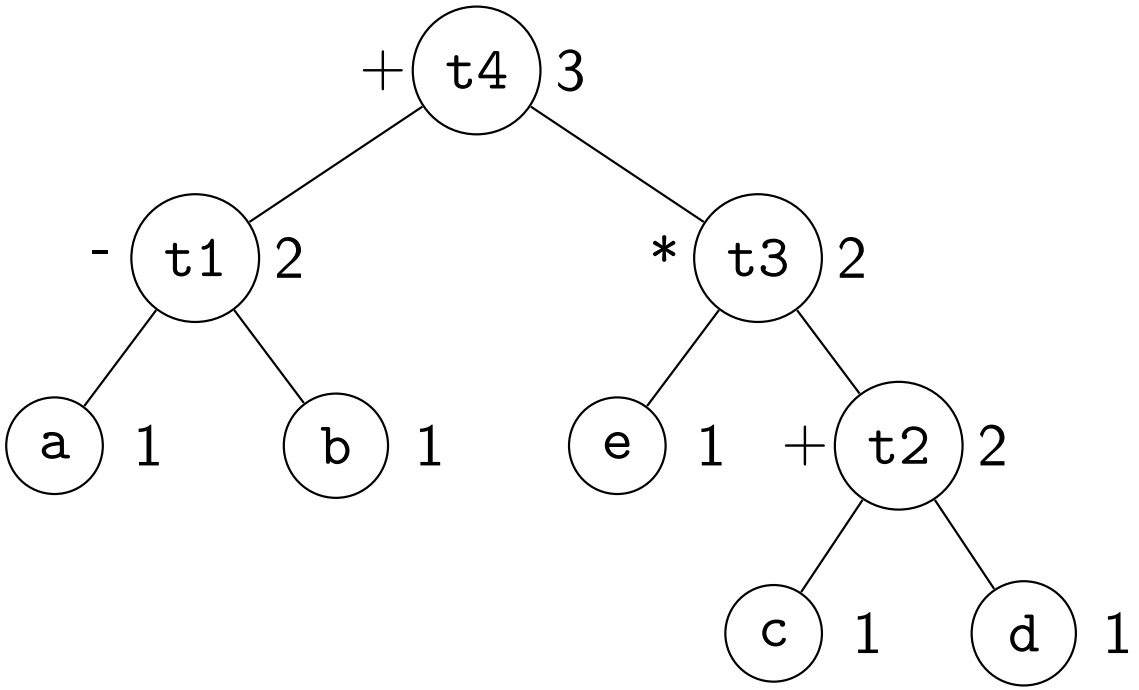
- De nuevo en (t4)
  - Tenemos $b = 1$
  - Listo hijo derecho (t3)



Generación recursiva

Para nuestro ejemplo

- De nuevo en (t4)
  - Tenemos $b = 1$
  - Listo hijo derecho (t3)
- Generar izquierdo (t1)
  - Con $b = 1$
  - Usará `R1` y `R2`
  - Resultado en `R2`
  - $label(a) = label(b)$



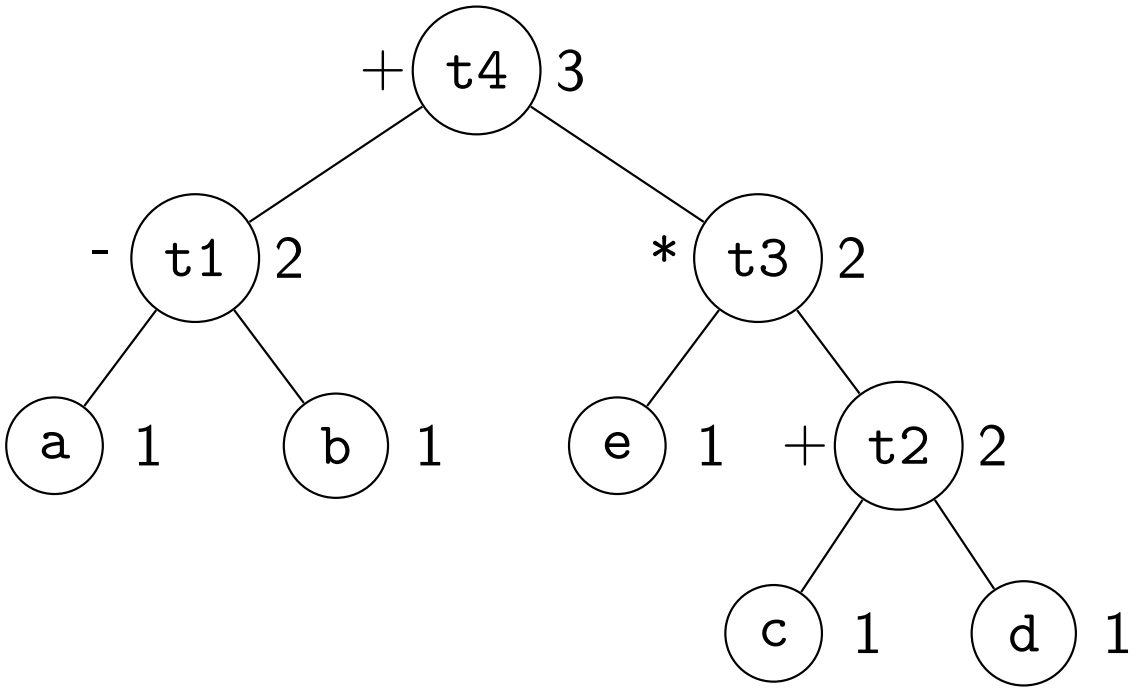
UNIVERSIDAD SIMÓN BOLÍVAR

Generación recursiva

Para nuestro ejemplo

- De nuevo en (t4)
  - Tenemos $b = 1$
  - Listo hijo derecho (t3)
- Generar izquierdo (t1)
  - Con $b = 1$
  - Usará `R1` y `R2`
  - Resultado en `R2`
  - $label(a) = label(b)$
- Hoja derecha (b)

  `LD R2, b`

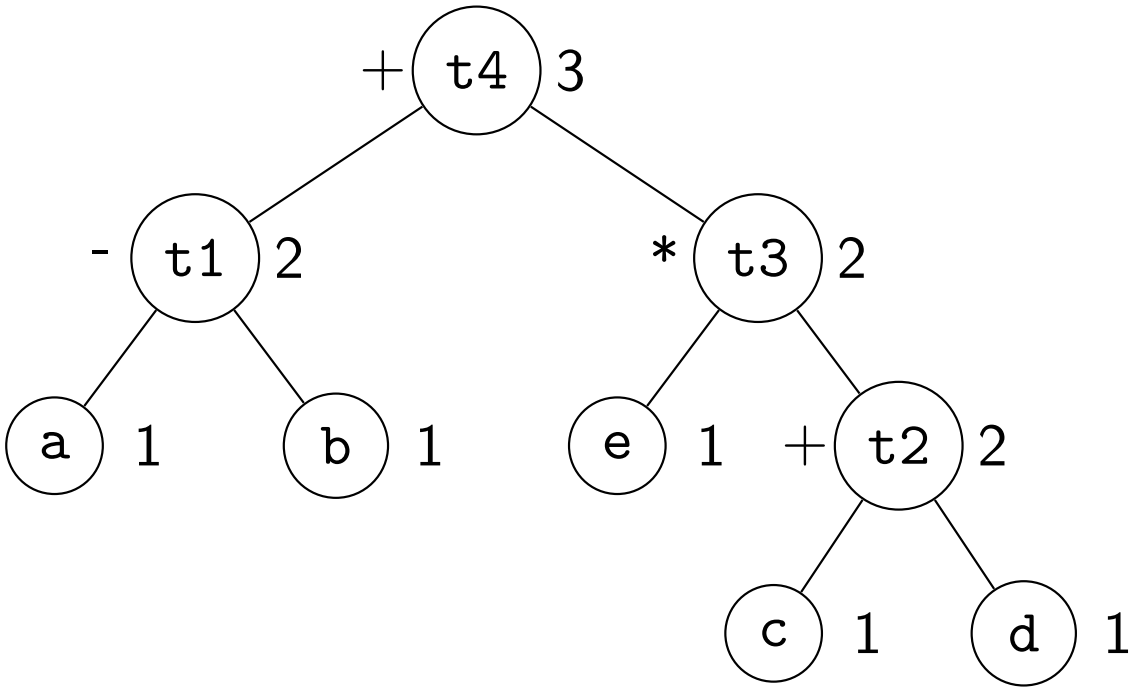


UNIVERSIDAD SIMÓN BOLÍVAR

Generación recursiva

Para nuestro ejemplo

- De nuevo en (t4)
  - Tenemos $b = 1$
  - Listo hijo derecho (t3)
- Generar izquierdo (t1)
  - Con $b = 1$
  - Usará `R1` y `R2`
  - Resultado en `R2`
  - $label(a) = label(b)$
- Hoja derecha (b)
  `LD R2, b`
- Hoja izquierda (a)
  `LD R1, a`

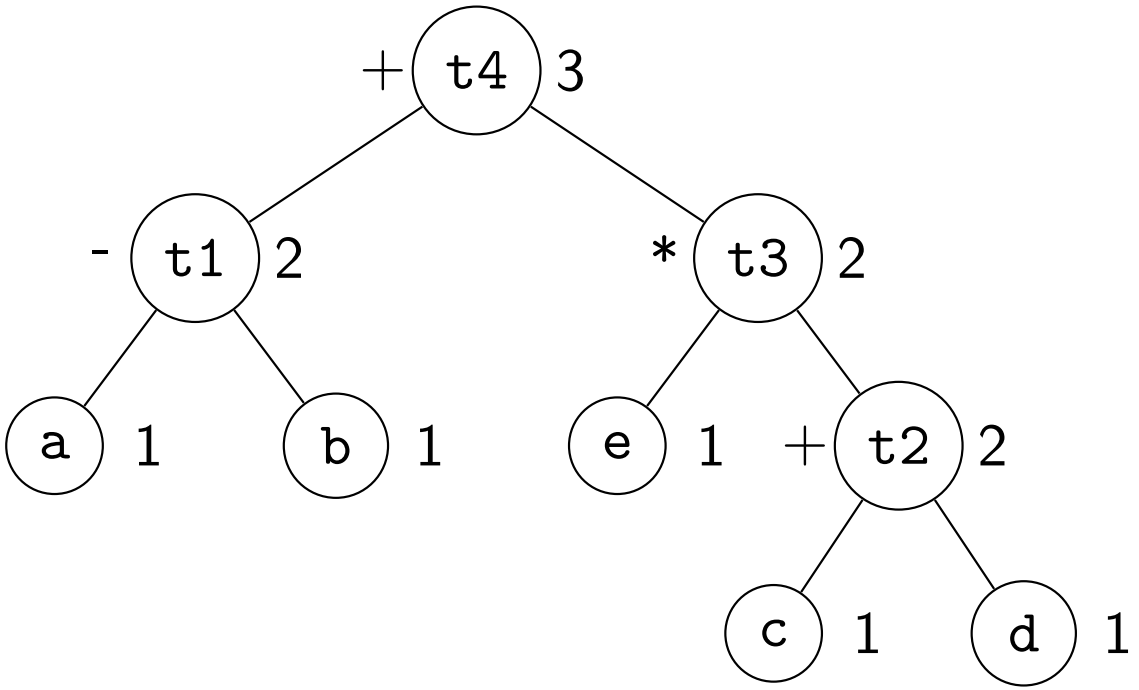


UNIVERSIDAD SIMÓN BOLÍVAR

Generación recursiva

Para nuestro ejemplo

- De nuevo en (t4)
  - Tenemos $b = 1$
  - Listo hijo derecho (t3)
- Generar izquierdo (t1)
  - Con $b = 1$
  - Usará `R1` y `R2`
  - Resultado en `R2`
  - $label(a) = label(b)$
- Hoja derecha (b)
  `LD R2, b`
- Hoja izquierda (a)
  `LD R1, a`
- De nuevo en (t1)
  `SUB R2, R1, R2`



UNIVERSIDAD SIMÓN BOLÍVAR

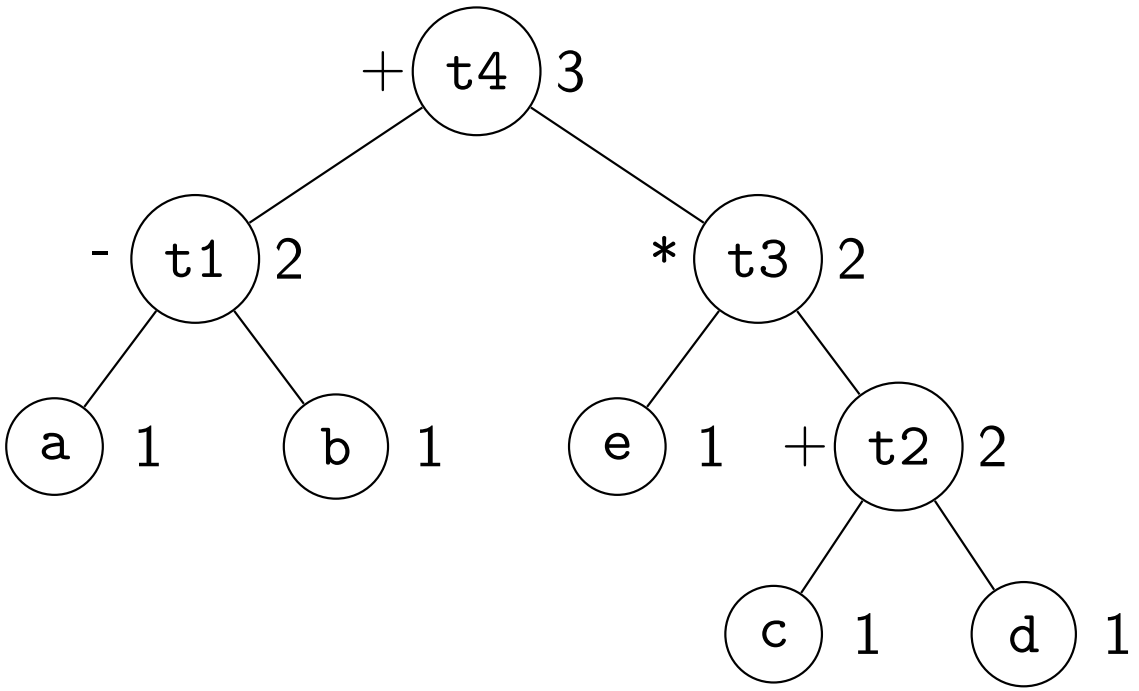
Generación recursiva

Para nuestro ejemplo

- De nuevo en (t4)
  - Tenemos $b = 1$
  - Listos ambos hijos

  `ADD R3, R2, R3`

```
LD   R3, d
LD   R2, c
ADD  R3, R2, R3
LD   R2, e
MUL  R3, R2, R3
LD   R2, b
LD   R1, a
SUB  R2, R1, R2
ADD  R3, R2, R3
```



UNIVERSIDAD SIMÓN BOLÍVAR

¿Qué hacer ante la escasez de registros?

Incluiremos *spills*

- El algoritmo se modifica para el caso que la etiqueta k de un nodo interior sea mayor a la cantidad de registros disponibles r.
  - Generar código para el sub-árbol c_b con etiqueta mayor – usar $b = 1$ para que el resultado esté en R_r.
  - Se completa el *spill* a espacio local en pila generando `ST` $\texttt{t}_k$`,` $\texttt{R}_r$
  - Generar código para el sub-árbol c_s con etiqueta menor
    - $b = 1$ – si $label(c_s) \geq r$
    - $b = r - j$ – si $label(c_s) = j < r$
    - El resultado estará en R_r.
  - Cargar el intermedio nuevamente `LD` $\texttt{R}_{r-1}$`,` $\texttt{t}_k$
  - Si c_b es el hijo izquierdo, generar `OP` $\texttt{R}_r$`,` $\texttt{R}_{r-1}$`,` $\texttt{R}_r$
  - Si c_b es el hijo derecho, generar `OP` $\texttt{R}_r$`,` $\texttt{R}_r$`,` $\texttt{R}_{r-1}$

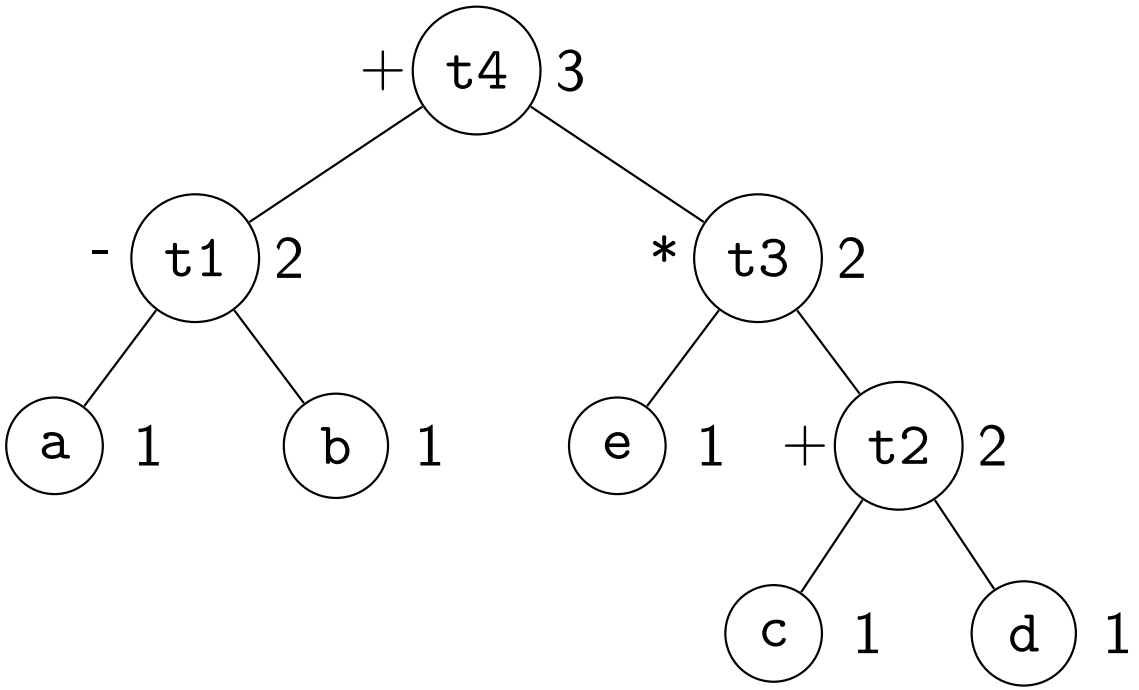
UNIVERSIDAD SIMÓN BOLÍVAR

Generación recursiva con escasez

Para nuestro ejemplo

- Escasez en (t4)
  - Digamos $c_b = t3$
  - t1 global o en pila

```
LD   R2, d
LD   R1, c
ADD  R2, R1, R2
LD   R1, e
MUL  R2, R1, R2
ST   t3, R2
LD   R2, b
LD   R1, a
SUB  R2, R1, R2
LD   R1, t3
ADD  R2, R2, R1
```



Bibliografía

- [*Aho*]
  - Secciones 8.9 y 8.10
  - Ejercicios 8.9.1 a 8.9.4, y 8.10.1 a 8.10.7
- Efficient Tree Pattern Matching: an Aid to Code Generation
  Aho & Ganapathi

UNIVERSIDAD SIMÓN BOLÍVAR