



Compiler Construction

Lecture 14: Semantic Analysis III (Attribute Evaluation)

Winter Semester 2018/19

Thomas Noll

Software Modeling and Verification Group

RWTH Aachen University

<https://moves.rwth-aachen.de/teaching/ws-1819/cc/>

Recap: The Circularity Check

Outline of Lecture 14

Recap: The Circularity Check

Attribute Evaluation

Attribute Evaluation by Topological Sorting

Attribute Evaluation by Recursive Functions

L-Attributed Grammars

Recap: The Circularity Check

Circularity of Attribute Grammars

Goal: **unique solvability** of equation system

⇒ avoid cyclic dependencies

Definition (Circularity)

An attribute grammar $\mathcal{A} = \langle G, E, V \rangle \in AG$ is called **circular** if there exists a syntax tree t such that the attribute equation system E_t is recursive (i.e., some attribute variable of t depends on itself). Otherwise it is called **noncircular**.

Remark: because of the division of Var_π into Int_π and Ext_π , cyclic dependencies cannot occur at production level.

Recap: The Circularity Check

The Circularity Check

Algorithm (Circularity check for attribute grammars)

Input: $\mathfrak{A} = \langle G, E, V \rangle \in AG$ with $G = \langle N, \Sigma, P, S \rangle$

Procedure: 1. for every $A \in N$, **iteratively construct** $IS(A)$ as follows:

i. if $\pi = A \rightarrow w \in P$, then

$$is[\pi] \in IS(A)$$

ii. if $\pi = A \rightarrow w_0 A_1 w_1 \dots A_r w_r \in P$ and $is_i \in IS(A_i)$ for every $i \in [r]$, then

$$is[\pi; is_1, \dots, is_r] \in IS(A)$$

2. **test whether** \mathfrak{A} **is circular** by checking if there exist $\pi = A \rightarrow w_0 A_1 w_1 \dots A_r w_r \in P$ and $is_i \in IS(A_i)$ for every $i \in [r]$ such that the following relation is cyclic:

$$\rightarrow_{\pi} \cup \bigcup_{i=1}^r \{(\beta.p_i, \alpha.p_i) \mid (\beta, \alpha) \in is_i\}$$

(where $p_i := \sum_{j=1}^i |w_{j-1}| + i$)

Output: “yes” or “no”

Attribute Evaluation

Outline of Lecture 14

Recap: The Circularity Check

Attribute Evaluation

Attribute Evaluation by Topological Sorting

Attribute Evaluation by Recursive Functions

L-Attributed Grammars

Attribute Evaluation

Attribute Evaluation Methods

- Given:
- noncircular attribute grammar $\mathfrak{A} = \langle G, E, V \rangle \in AG$
 - syntax tree t of G
 - valuation $v : Syn_{\Sigma} \rightarrow V$ for $Syn_{\Sigma} := \{\alpha.k \mid k \text{ labelled by } a \in \Sigma, \alpha \in \text{syn}(a)\} \subseteq Var_t$

Attribute Evaluation

Attribute Evaluation Methods

- Given:
- noncircular attribute grammar $\mathfrak{A} = \langle G, E, V \rangle \in AG$
 - syntax tree t of G
 - valuation $v : Syn_{\Sigma} \rightarrow V$ for $Syn_{\Sigma} := \{\alpha.k \mid k \text{ labelled by } a \in \Sigma, \alpha \in \text{syn}(a)\} \subseteq Var_t$

Goal: extend v to (partial) **solution** $v : Var_t \rightarrow V$

Attribute Evaluation

Attribute Evaluation Methods

- Given:
- noncircular attribute grammar $\mathfrak{A} = \langle G, E, V \rangle \in AG$
 - syntax tree t of G
 - valuation $v : Syn_{\Sigma} \rightarrow V$ for $Syn_{\Sigma} := \{\alpha.k \mid k \text{ labelled by } a \in \Sigma, \alpha \in \text{syn}(a)\} \subseteq Var_t$

Goal: extend v to (partial) **solution** $v : Var_t \rightarrow V$

Methods: 1. **Topological sorting** of D_t (for arbitrary noncircular AGs):

- i. start with variables which depend at most on Syn_{Σ}
- ii. proceed by successive substitution

Attribute Evaluation

Attribute Evaluation Methods

- Given:
- noncircular attribute grammar $\mathfrak{A} = \langle G, E, V \rangle \in AG$
 - syntax tree t of G
 - valuation $v : Syn_{\Sigma} \rightarrow V$ for $Syn_{\Sigma} := \{\alpha.k \mid k \text{ labelled by } a \in \Sigma, \alpha \in \text{syn}(a)\} \subseteq Var_t$

Goal: extend v to (partial) **solution** $v : Var_t \rightarrow V$

Methods: 1. **Topological sorting** of D_t (for arbitrary noncircular AGs):

- i. start with variables which depend at most on Syn_{Σ}
- ii. proceed by successive substitution

2. Using **recursive functions** (for **strongly noncircular** AGs; later)

Attribute Evaluation

Attribute Evaluation Methods

- Given:
- noncircular attribute grammar $\mathfrak{A} = \langle G, E, V \rangle \in AG$
 - syntax tree t of G
 - valuation $v : Syn_{\Sigma} \rightarrow V$ for $Syn_{\Sigma} := \{\alpha.k \mid k \text{ labelled by } a \in \Sigma, \alpha \in \text{syn}(a)\} \subseteq Var_t$

Goal: extend v to (partial) **solution** $v : Var_t \rightarrow V$

Methods: 1. **Topological sorting** of D_t (for arbitrary noncircular AGs):

- i. start with variables which depend at most on Syn_{Σ}
 - ii. proceed by successive substitution
2. Using **recursive functions** (for **strongly noncircular** AGs; later)
 3. **Integration with top-down parsing** (for **L-attributed** grammars; later)

Attribute Evaluation

Attribute Evaluation Methods

- Given:
- noncircular attribute grammar $\mathfrak{A} = \langle G, E, V \rangle \in AG$
 - syntax tree t of G
 - valuation $v : Syn_{\Sigma} \rightarrow V$ for $Syn_{\Sigma} := \{\alpha.k \mid k \text{ labelled by } a \in \Sigma, \alpha \in \text{syn}(a)\} \subseteq Var_t$

Goal: extend v to (partial) **solution** $v : Var_t \rightarrow V$

Methods: 1. **Topological sorting** of D_t (for arbitrary noncircular AGs):

- i. start with variables which depend at most on Syn_{Σ}
 - ii. proceed by successive substitution
2. Using **recursive functions** (for **strongly noncircular** AGs; later)
 3. **Integration with top-down parsing** (for **L-attributed** grammars; later)
 4. **Integration with bottom-up parsing** (for **S-attributed grammars**, i.e., $Inh = \emptyset$):
yacc/bison

Attribute Evaluation by Topological Sorting

Outline of Lecture 14

Recap: The Circularity Check

Attribute Evaluation

Attribute Evaluation by Topological Sorting

Attribute Evaluation by Recursive Functions

L-Attributed Grammars

Attribute Evaluation by Topological Sorting

Attribute Evaluation by Topological Sorting I

Algorithm 14.1 (Evaluation by topological sorting)

Input: noncircular $\mathfrak{A} = \langle G, E, V \rangle \in AG$, syntax tree t of G , $v : \text{Syn}_\Sigma \rightarrow V$

Attribute Evaluation by Topological Sorting

Attribute Evaluation by Topological Sorting I

Algorithm 14.1 (Evaluation by topological sorting)

Input: noncircular $\mathcal{A} = \langle G, E, V \rangle \in AG$, syntax tree t of G , $v : \text{Syn}_\Sigma \rightarrow V$

Procedure: 1. let $\text{Var} := \text{Var}_t \setminus \text{Syn}_\Sigma$ (* attributes to be evaluated *)

2. while $\text{Var} \neq \emptyset$ do

i. let $x \in \text{Var}$ such that $\{y \in \text{Var} \mid y \rightarrow_t x\} = \emptyset$

ii. let $x = f(x_1, \dots, x_n) \in E_t$

iii. let $v(x) := f(v(x_1), \dots, v(x_n))$

iv. let $\text{Var} := \text{Var} \setminus \{x\}$

Attribute Evaluation by Topological Sorting

Attribute Evaluation by Topological Sorting I

Algorithm 14.1 (Evaluation by topological sorting)

Input: noncircular $\mathfrak{A} = \langle G, E, V \rangle \in AG$, syntax tree t of G , $v : \text{Syn}_\Sigma \rightarrow V$

Procedure: 1. let $\text{Var} := \text{Var}_t \setminus \text{Syn}_\Sigma$ (* attributes to be evaluated *)

2. while $\text{Var} \neq \emptyset$ do

i. let $x \in \text{Var}$ such that $\{y \in \text{Var} \mid y \rightarrow_t x\} = \emptyset$

ii. let $x = f(x_1, \dots, x_n) \in E_t$

iii. let $v(x) := f(v(x_1), \dots, v(x_n))$

iv. let $\text{Var} := \text{Var} \setminus \{x\}$

Output: solution $v : \text{Var}_t \rightarrow V$

Attribute Evaluation by Topological Sorting

Attribute Evaluation by Topological Sorting I

Algorithm 14.1 (Evaluation by topological sorting)

Input: noncircular $\mathfrak{A} = \langle G, E, V \rangle \in AG$, syntax tree t of G , $v : \text{Syn}_\Sigma \rightarrow V$

Procedure: 1. let $\text{Var} := \text{Var}_t \setminus \text{Syn}_\Sigma$ (* attributes to be evaluated *)

2. while $\text{Var} \neq \emptyset$ do

i. let $x \in \text{Var}$ such that $\{y \in \text{Var} \mid y \rightarrow_t x\} = \emptyset$

ii. let $x = f(x_1, \dots, x_n) \in E_t$

iii. let $v(x) := f(v(x_1), \dots, v(x_n))$

iv. let $\text{Var} := \text{Var} \setminus \{x\}$

Output: solution $v : \text{Var}_t \rightarrow V$

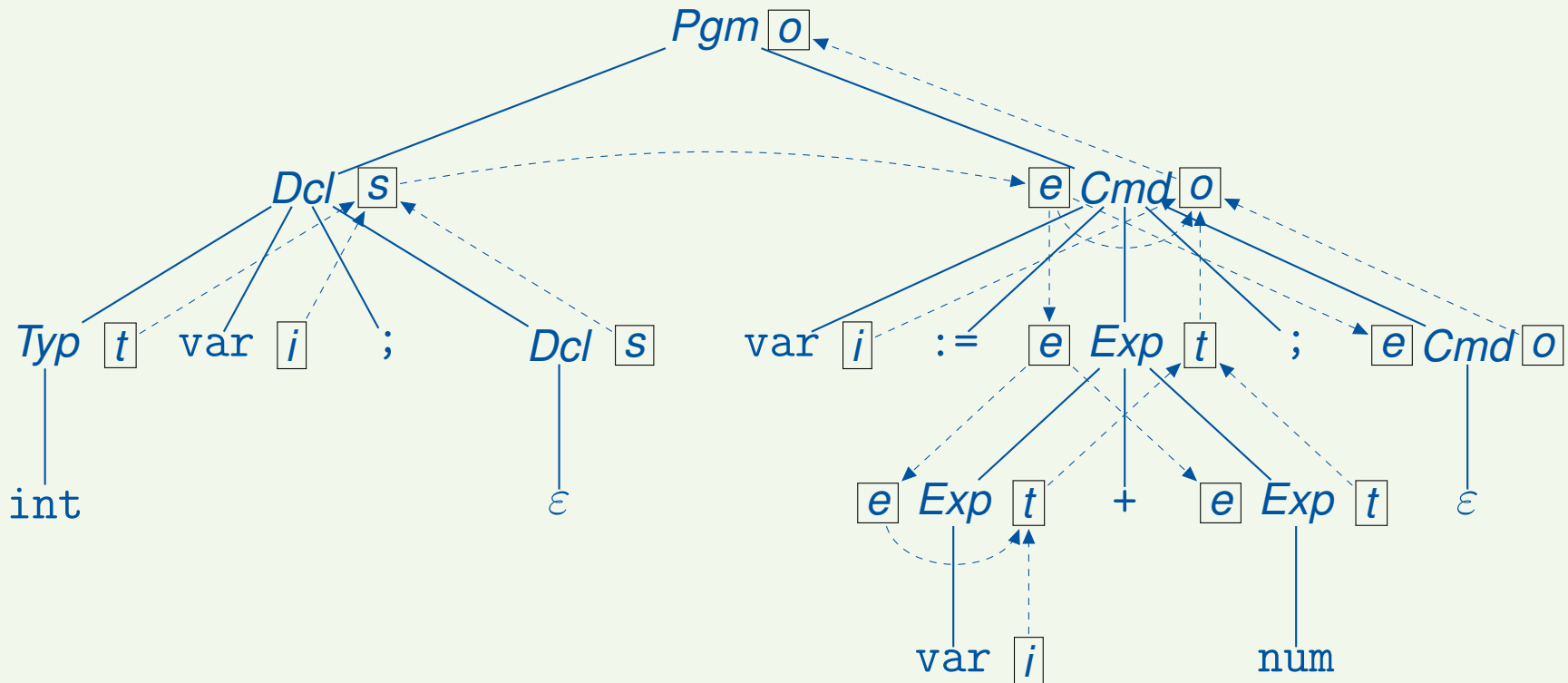
Remark: noncircularity guarantees that in step 2(i) at least one such x is available

Attribute Evaluation by Topological Sorting

Attribute Evaluation by Topological Sorting II

Example 14.2 (cf. Example 12.16)

Attribute evaluation for `int x; x := x+1;`

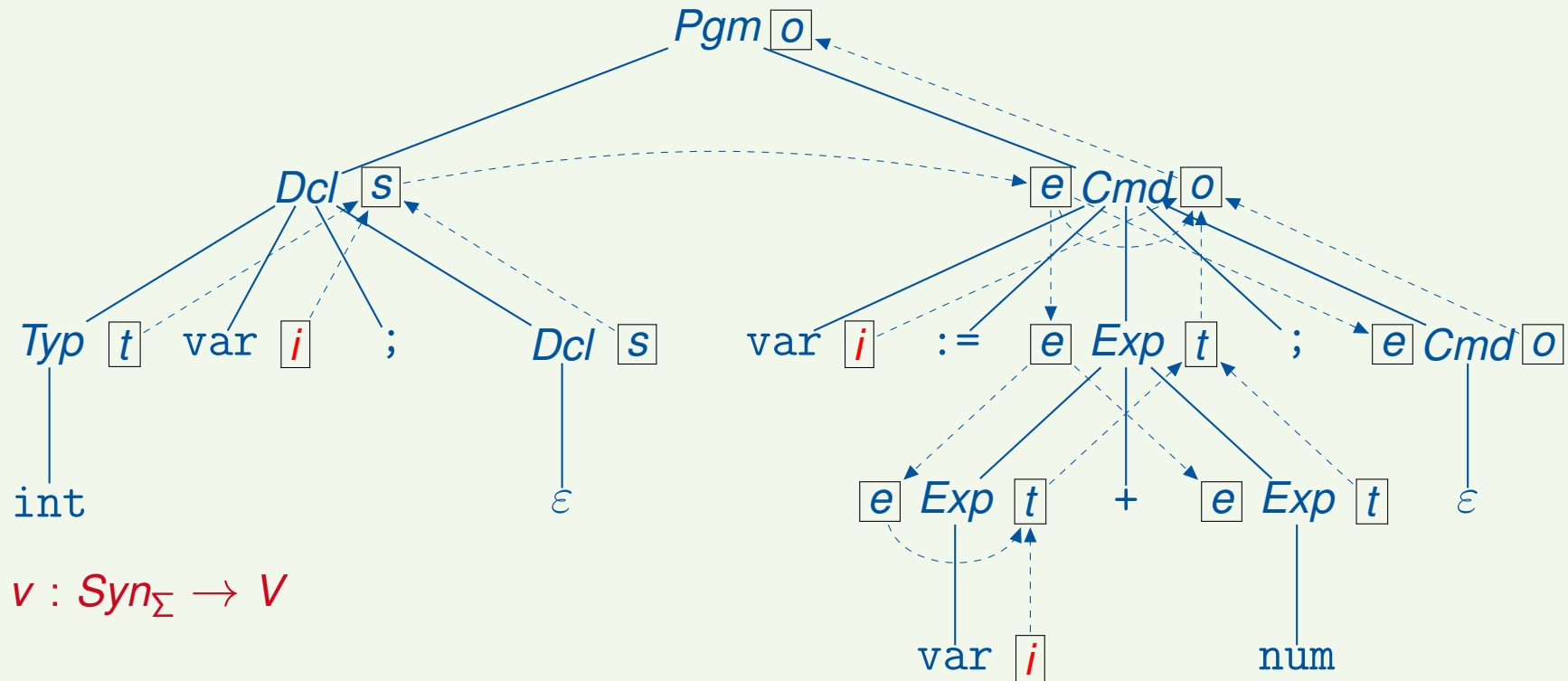


Attribute Evaluation by Topological Sorting

Attribute Evaluation by Topological Sorting II

Example 14.2 (cf. Example 12.16)

Attribute evaluation for `int x; x := x+1;`

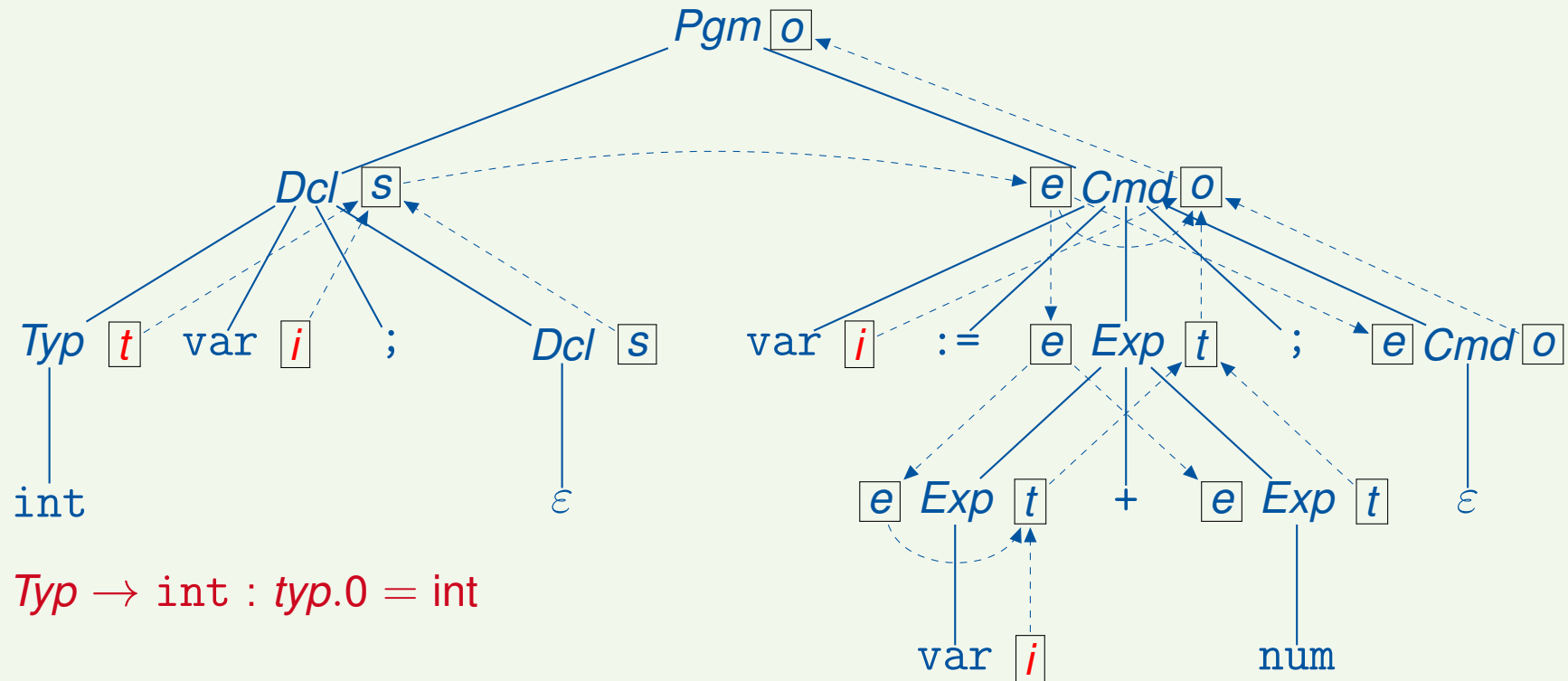


Attribute Evaluation by Topological Sorting

Attribute Evaluation by Topological Sorting II

Example 14.2 (cf. Example 12.16)

Attribute evaluation for `int x; x := x+1;`

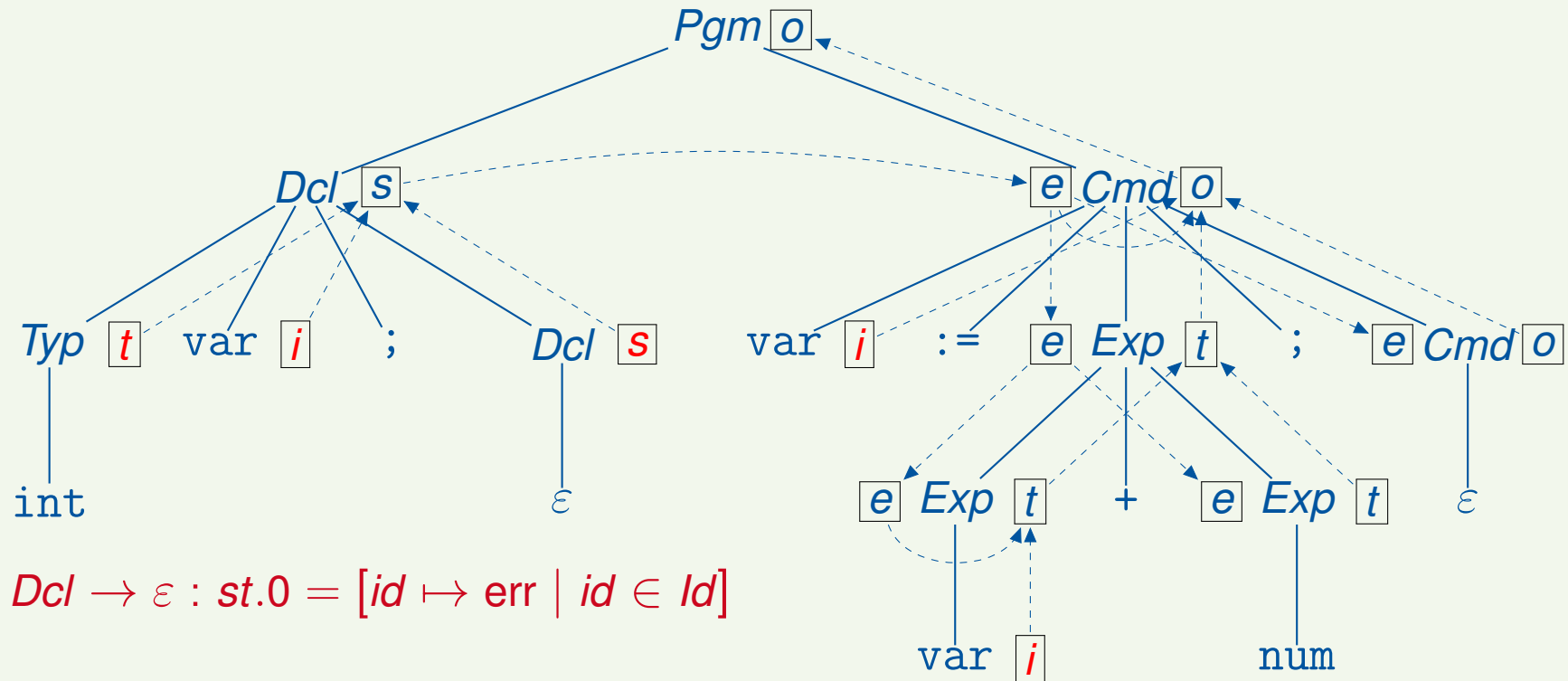


Attribute Evaluation by Topological Sorting

Attribute Evaluation by Topological Sorting II

Example 14.2 (cf. Example 12.16)

Attribute evaluation for `int x; x := x+1;`

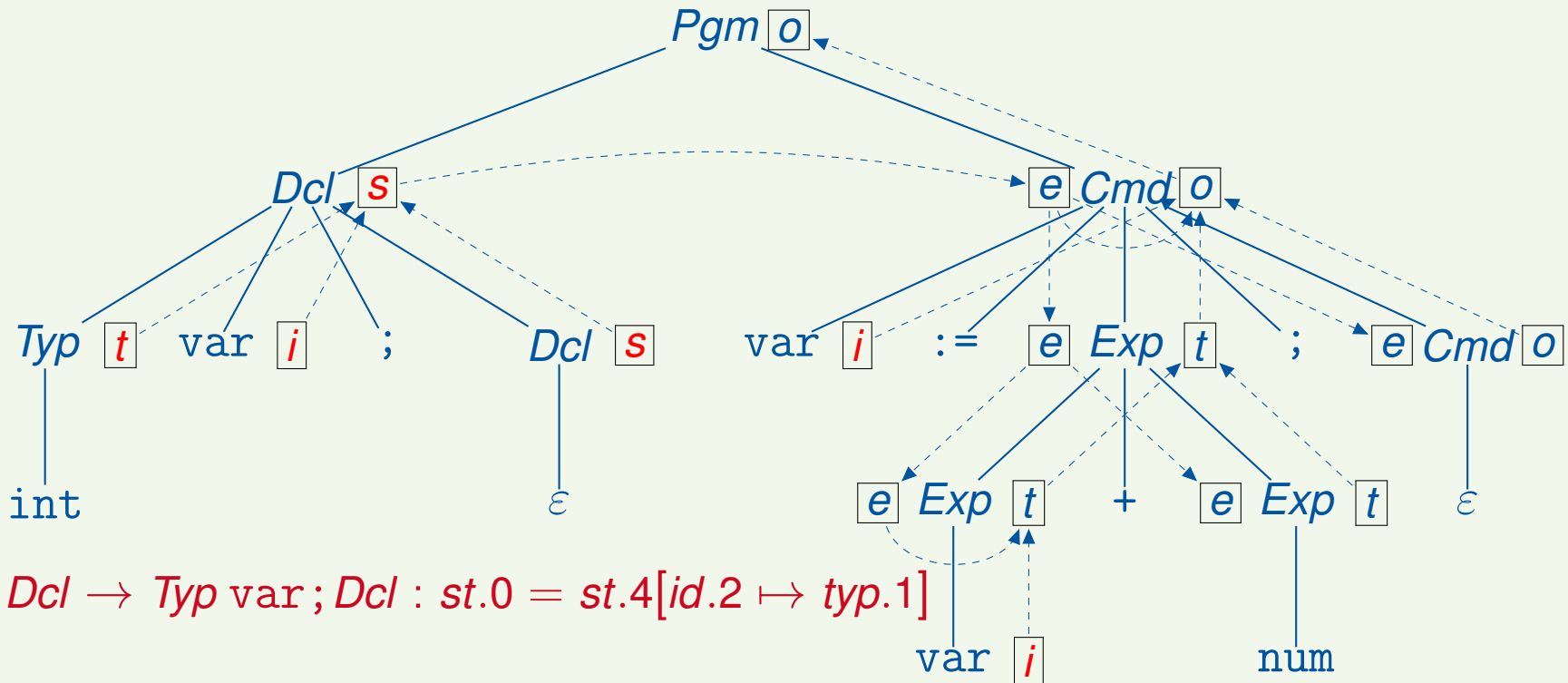


Attribute Evaluation by Topological Sorting

Attribute Evaluation by Topological Sorting II

Example 14.2 (cf. Example 12.16)

Attribute evaluation for `int x; x := x+1;`

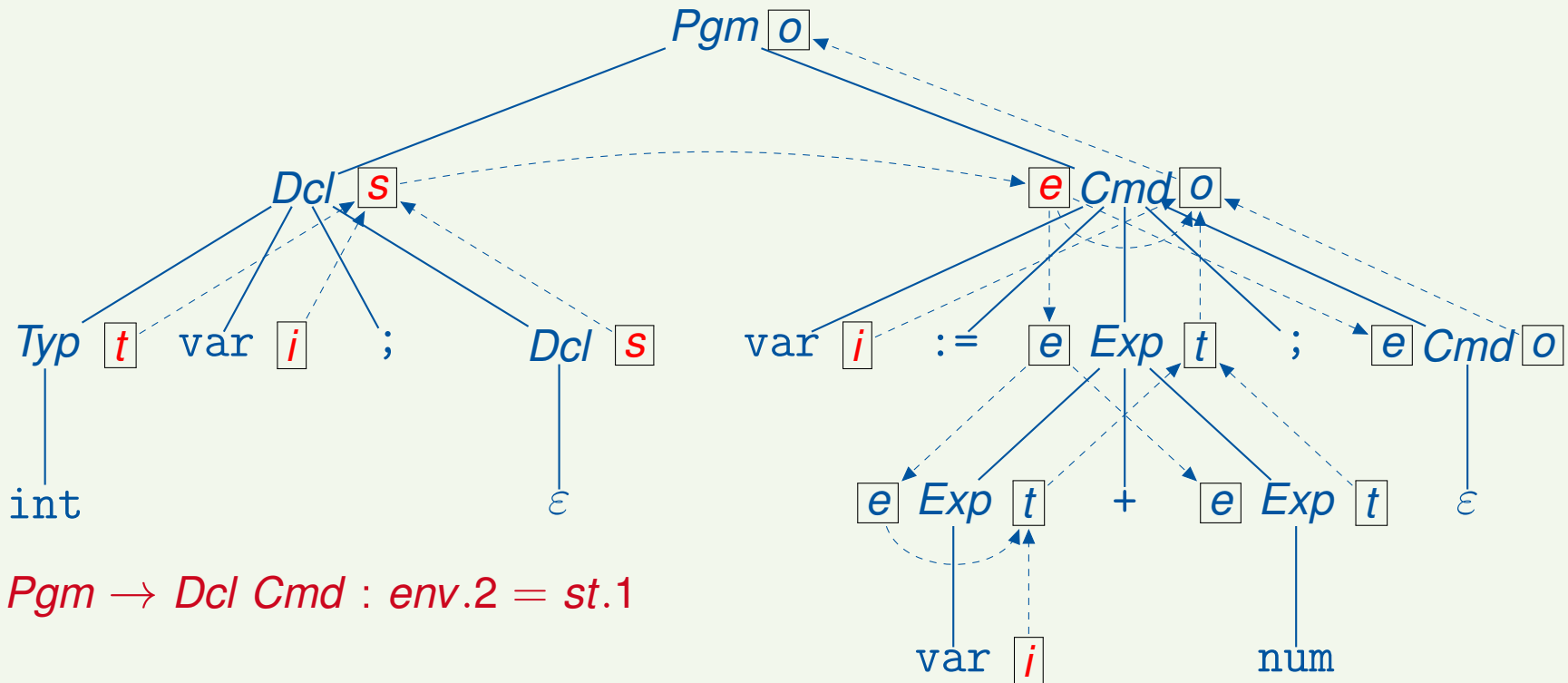


Attribute Evaluation by Topological Sorting

Attribute Evaluation by Topological Sorting II

Example 14.2 (cf. Example 12.16)

Attribute evaluation for `int x; x := x+1;`

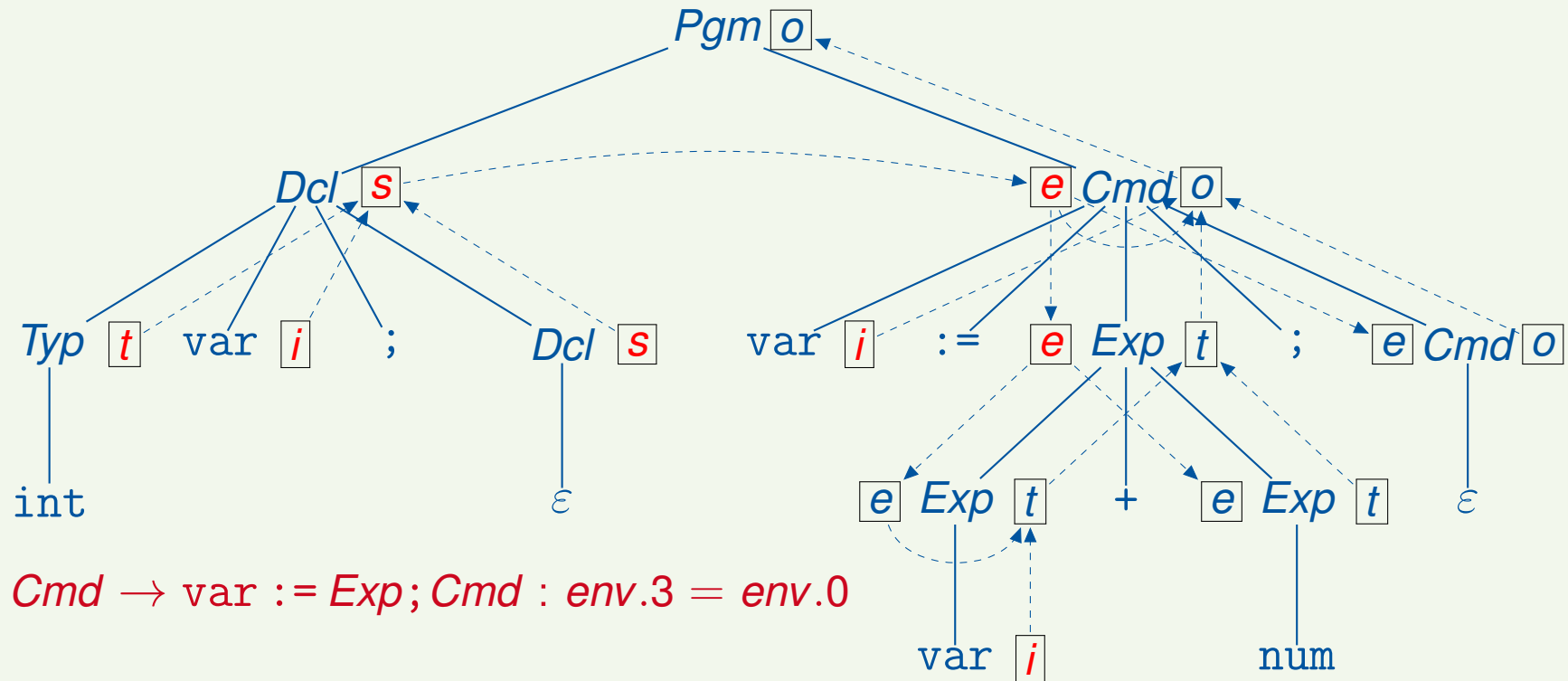


Attribute Evaluation by Topological Sorting

Attribute Evaluation by Topological Sorting II

Example 14.2 (cf. Example 12.16)

Attribute evaluation for `int x; x := x+1;`

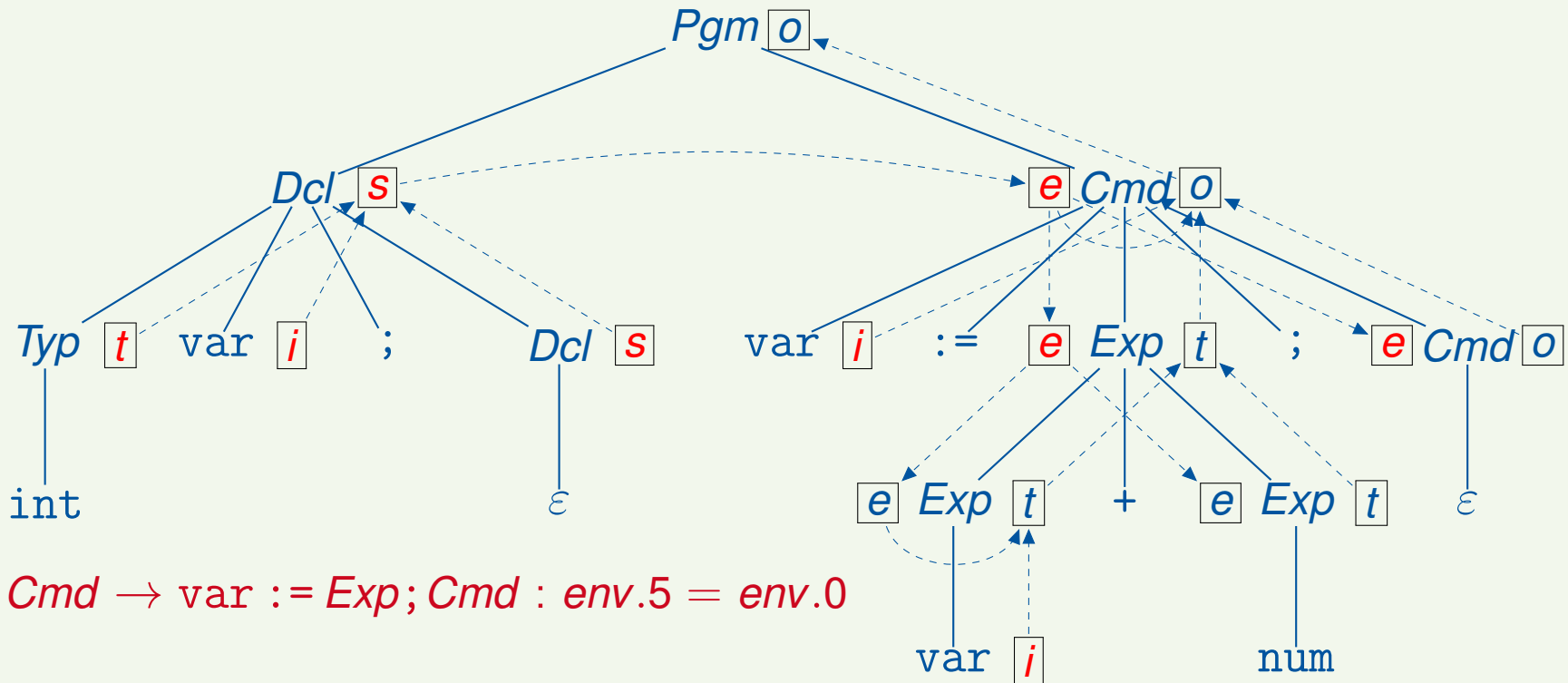


Attribute Evaluation by Topological Sorting

Attribute Evaluation by Topological Sorting II

Example 14.2 (cf. Example 12.16)

Attribute evaluation for `int x; x := x+1;`

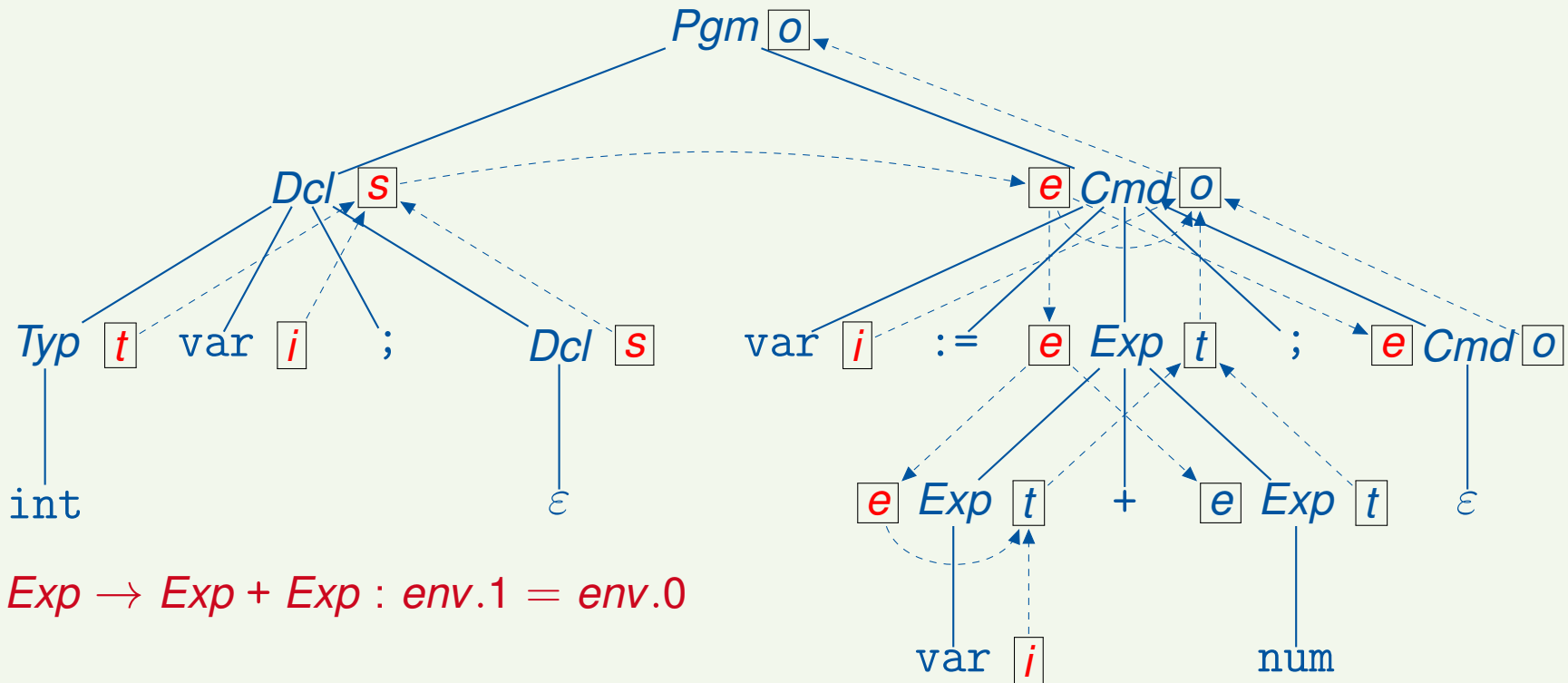


Attribute Evaluation by Topological Sorting

Attribute Evaluation by Topological Sorting II

Example 14.2 (cf. Example 12.16)

Attribute evaluation for `int x; x := x+1;`

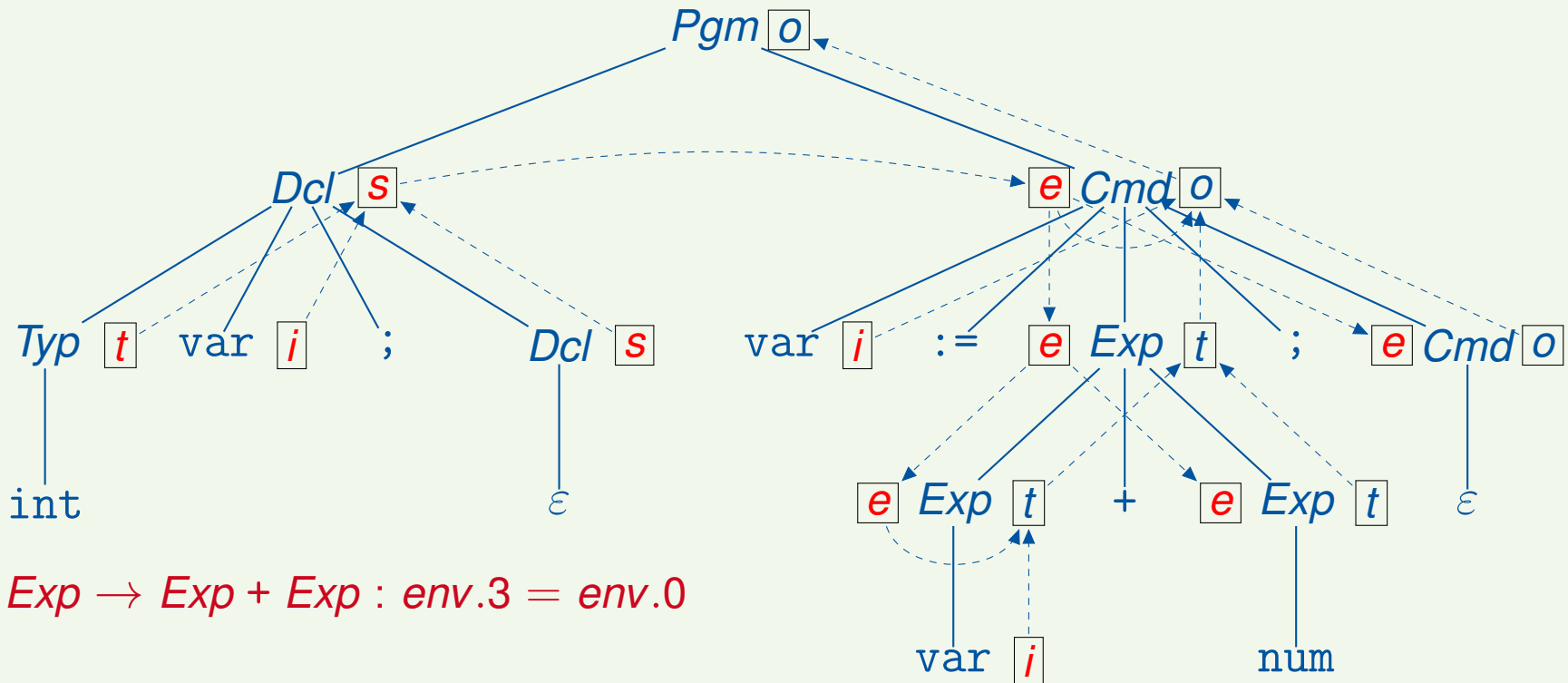


Attribute Evaluation by Topological Sorting

Attribute Evaluation by Topological Sorting II

Example 14.2 (cf. Example 12.16)

Attribute evaluation for `int x; x := x+1;`

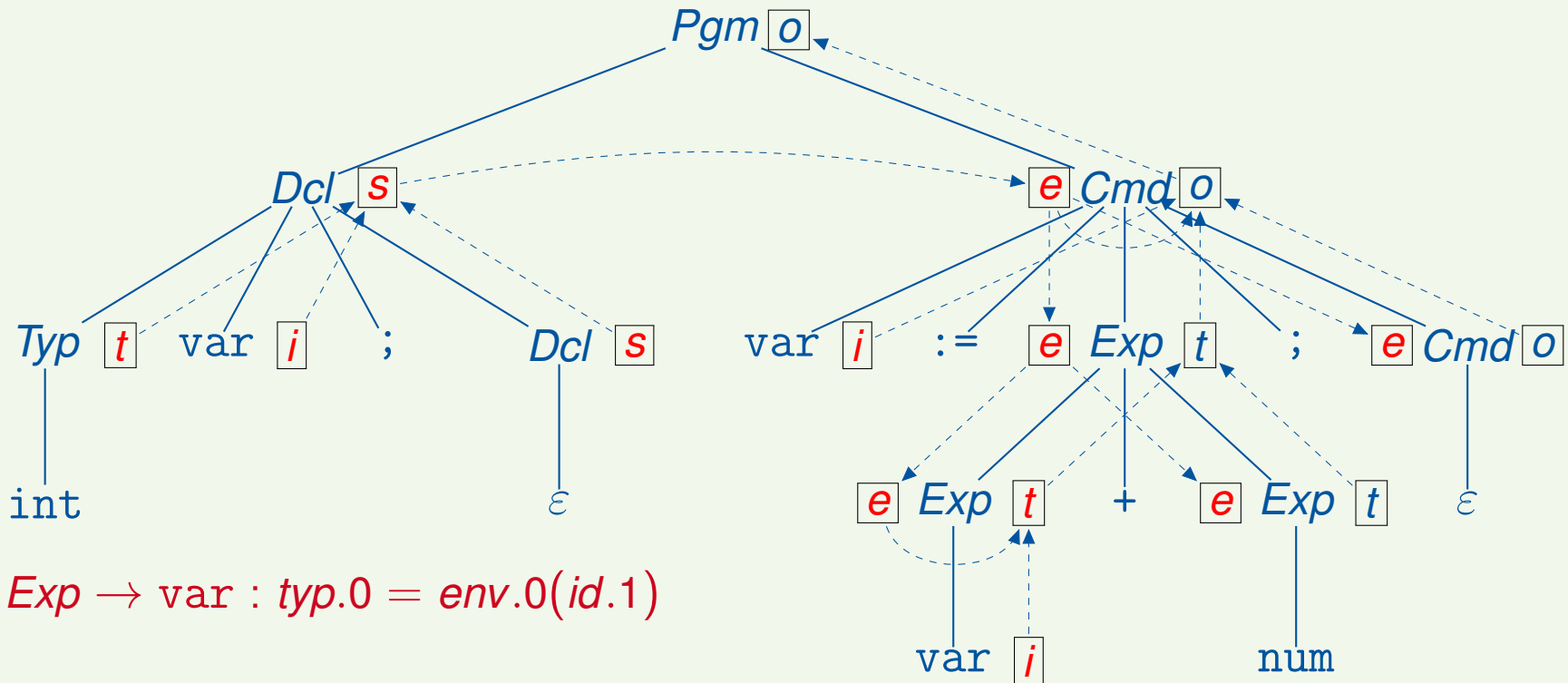


Attribute Evaluation by Topological Sorting

Attribute Evaluation by Topological Sorting II

Example 14.2 (cf. Example 12.16)

Attribute evaluation for `int x; x := x+1;`

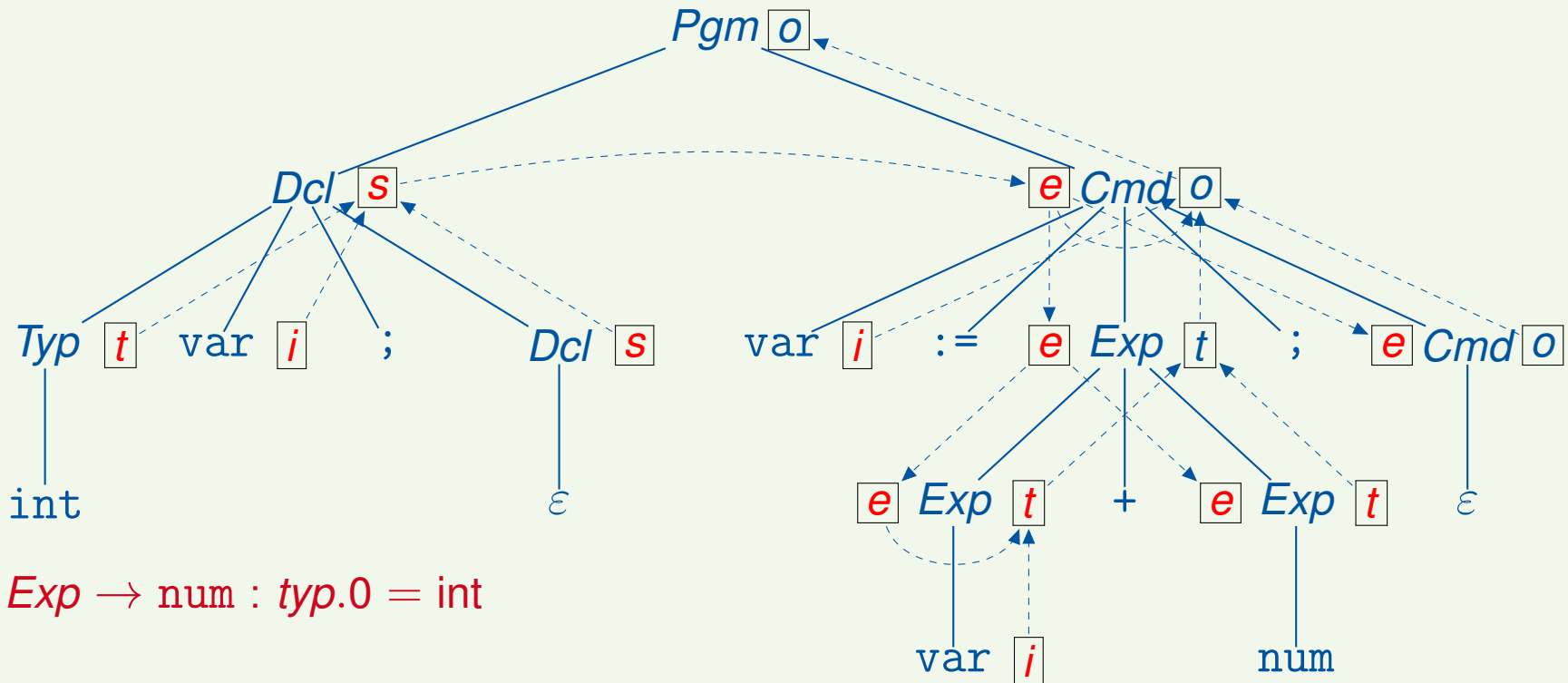


Attribute Evaluation by Topological Sorting

Attribute Evaluation by Topological Sorting II

Example 14.2 (cf. Example 12.16)

Attribute evaluation for `int x; x := x+1;`

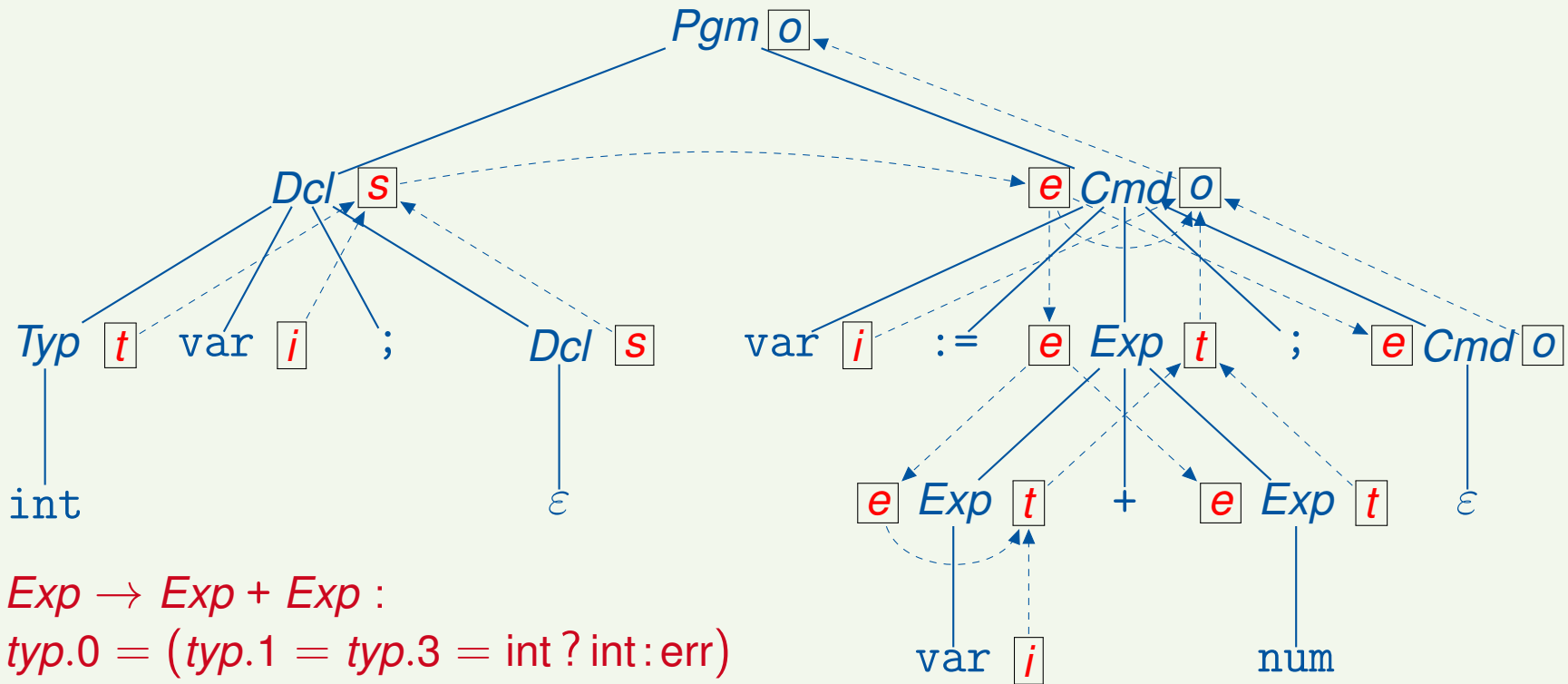


Attribute Evaluation by Topological Sorting

Attribute Evaluation by Topological Sorting II

Example 14.2 (cf. Example 12.16)

Attribute evaluation for `int x; x := x+1;`

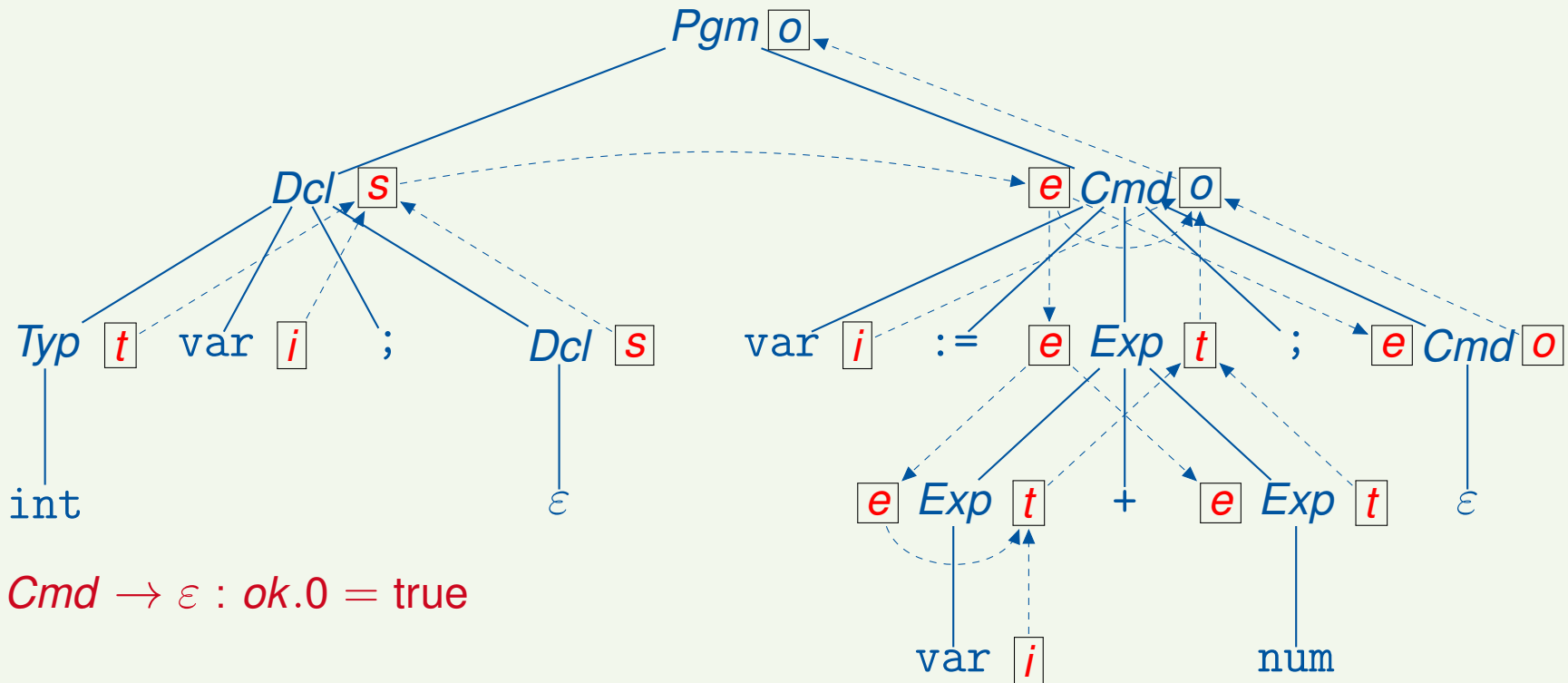


Attribute Evaluation by Topological Sorting

Attribute Evaluation by Topological Sorting II

Example 14.2 (cf. Example 12.16)

Attribute evaluation for `int x; x := x+1;`

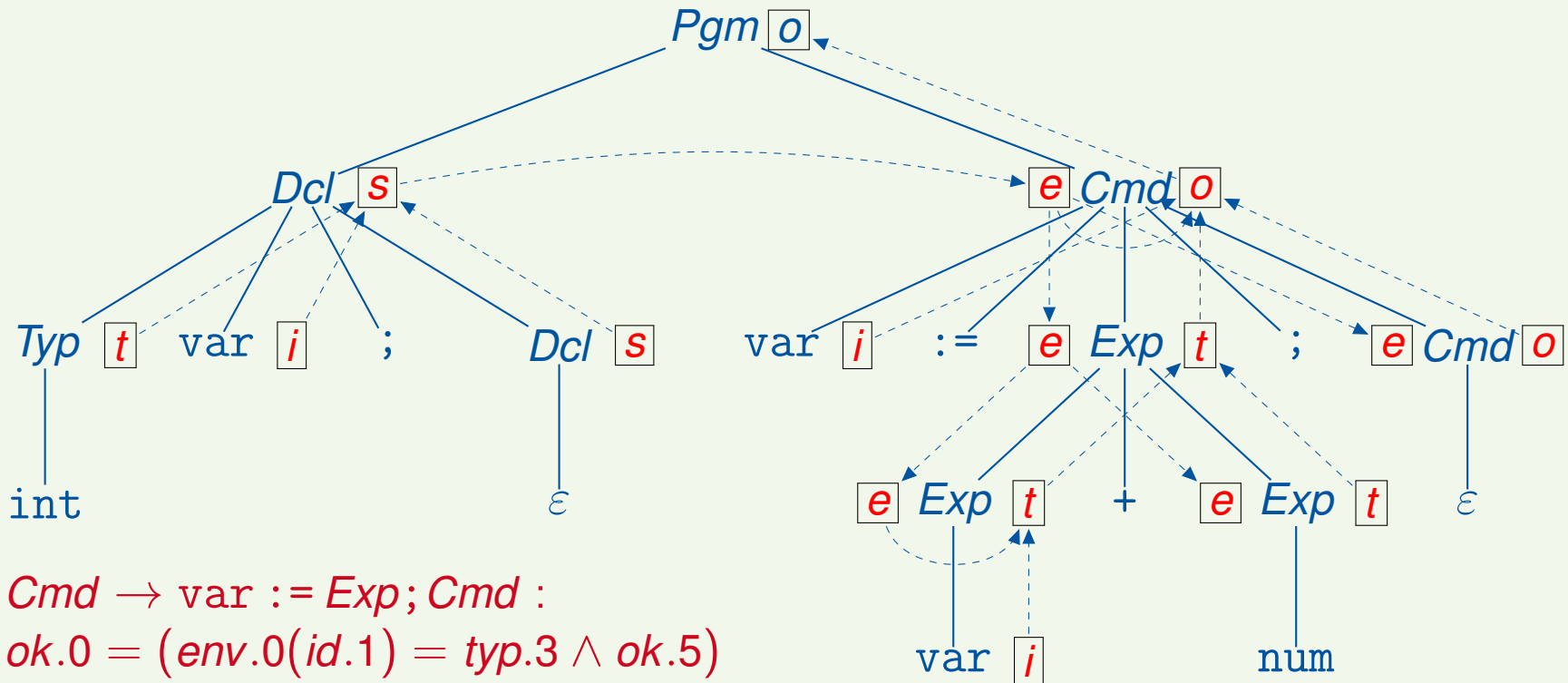


Attribute Evaluation by Topological Sorting

Attribute Evaluation by Topological Sorting II

Example 14.2 (cf. Example 12.16)

Attribute evaluation for `int x; x := x+1;`



Attribute Evaluation by Recursive Functions

Outline of Lecture 14

Recap: The Circularity Check

Attribute Evaluation

Attribute Evaluation by Topological Sorting

Attribute Evaluation by Recursive Functions

L-Attributed Grammars

Attribute Evaluation by Recursive Functions

Strong Noncircularity I

Definition 14.3 (Strong Noncircularity)

Let $\mathcal{A} = \langle G, E, V \rangle \in AG$ with $G = \langle N, \Sigma, P, S \rangle$.

- **Reminder:** If t is a syntax tree with root label $A \in N$ and root node k , $\alpha \in \text{syn}(A)$, and $\beta \in \text{inh}(A)$ such that $\beta.k \rightarrow_t^+ \alpha.k$, then α is dependent on β below A in t (notation: $\beta \xrightarrow{A} \alpha$).

Attribute Evaluation by Recursive Functions

Strong Noncircularity I

Definition 14.3 (Strong Noncircularity)

Let $\mathcal{A} = \langle G, E, V \rangle \in AG$ with $G = \langle N, \Sigma, P, S \rangle$.

- **Reminder:** If t is a syntax tree with root label $A \in N$ and root node k , $\alpha \in \text{syn}(A)$, and $\beta \in \text{inh}(A)$ such that $\beta.k \rightarrow_t^+ \alpha.k$, then α is dependent on β below A in t (notation: $\beta \xrightarrow{A} \alpha$).
- For every $A \in N$, $IS'(A) := \{(\beta, \alpha) \mid \beta \xrightarrow{A} \alpha \text{ in some } t \text{ with root label } A\} \subseteq \text{Inh} \times \text{Syn}$

Attribute Evaluation by Recursive Functions

Strong Noncircularity I

Definition 14.3 (Strong Noncircularity)

Let $\mathcal{A} = \langle G, E, V \rangle \in AG$ with $G = \langle N, \Sigma, P, S \rangle$.

- **Reminder:** If t is a syntax tree with root label $A \in N$ and root node k , $\alpha \in \text{syn}(A)$, and $\beta \in \text{inh}(A)$ such that $\beta.k \rightarrow_t^+ \alpha.k$, then α is **dependent on β below A in t** (notation: $\beta \xrightarrow{A} \alpha$).
- For every $A \in N$, $IS'(A) := \{(\beta, \alpha) \mid \beta \xrightarrow{A} \alpha \text{ in some } t \text{ with root label } A\} \subseteq \text{Inh} \times \text{Syn}$
- \mathcal{A} is **strongly noncircular** if for each $\pi = A \rightarrow w_0 A_1 w_1 \dots A_r w_r \in P$ the relation

$$\rightarrow_\pi \cup \bigcup_{i=1}^r \{(\beta.p_i, \alpha.p_i) \mid (\beta, \alpha) \in IS'(A_i)\}$$

is **acyclic** (where $p_i := \sum_{j=1}^i |w_{j-1}| + i$)

Attribute Evaluation by Recursive Functions

Strong Noncircularity I

Definition 14.3 (Strong Noncircularity)

Let $\mathfrak{A} = \langle G, E, V \rangle \in AG$ with $G = \langle N, \Sigma, P, S \rangle$.

- **Reminder:** If t is a syntax tree with root label $A \in N$ and root node k , $\alpha \in \text{syn}(A)$, and $\beta \in \text{inh}(A)$ such that $\beta.k \rightarrow_t^+ \alpha.k$, then α is **dependent on β below A in t** (notation: $\beta \xrightarrow{A} \alpha$).
- For every $A \in N$, $IS'(A) := \{(\beta, \alpha) \mid \beta \xrightarrow{A} \alpha \text{ in some } t \text{ with root label } A\} \subseteq \text{Inh} \times \text{Syn}$
- \mathfrak{A} is **strongly noncircular** if for each $\pi = A \rightarrow w_0 A_1 w_1 \dots A_r w_r \in P$ the relation

$$\rightarrow_\pi \cup \bigcup_{i=1}^r \{(\beta.p_i, \alpha.p_i) \mid (\beta, \alpha) \in IS'(A_i)\}$$

is **acyclic** (where $p_i := \sum_{j=1}^i |w_{j-1}| + i$)

Corollary 14.4

For every $A \in N$, $IS'(A) = \bigcup \{is \mid is \in IS(A)\}$.

Attribute Evaluation by Recursive Functions

Strong Noncircularity I

Definition 14.3 (Strong Noncircularity)

Let $\mathfrak{A} = \langle G, E, V \rangle \in AG$ with $G = \langle N, \Sigma, P, S \rangle$.

- **Reminder:** If t is a syntax tree with root label $A \in N$ and root node k , $\alpha \in \text{syn}(A)$, and $\beta \in \text{inh}(A)$ such that $\beta.k \rightarrow_t^+ \alpha.k$, then α is **dependent on β below A in t** (notation: $\beta \xrightarrow{A} \alpha$).
- For every $A \in N$, $IS'(A) := \{(\beta, \alpha) \mid \beta \xrightarrow{A} \alpha \text{ in some } t \text{ with root label } A\} \subseteq \text{Inh} \times \text{Syn}$
- \mathfrak{A} is **strongly noncircular** if for each $\pi = A \rightarrow w_0 A_1 w_1 \dots A_r w_r \in P$ the relation

$$\rightarrow_\pi \cup \bigcup_{i=1}^r \{(\beta.p_i, \alpha.p_i) \mid (\beta, \alpha) \in IS'(A_i)\}$$

is **acyclic** (where $p_i := \sum_{j=1}^i |w_{j-1}| + i$)

Corollary 14.4

For every $A \in N$, $IS'(A) = \bigcup \{is \mid is \in IS(A)\}$.

Remark: $IS'(A)$ is computable in polynomial time using a simplification of Alg. 13.6

Strong Noncircularity II

Lemma 14.5

1. *Every strongly noncircular attribute grammar is noncircular.*
2. *There are noncircular attribute grammars which are not strongly noncircular.*

Attribute Evaluation by Recursive Functions

Strong Noncircularity II

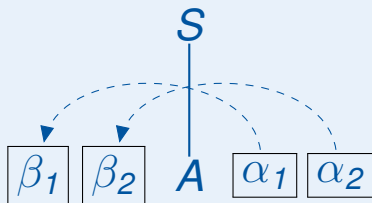
Lemma 14.5

1. Every strongly noncircular attribute grammar is noncircular.
2. There are noncircular attribute grammars which are not strongly noncircular.

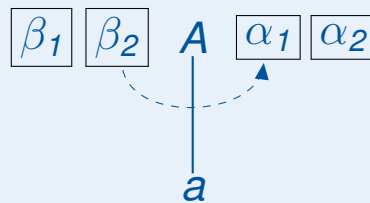
Proof.

1. Clear since $is \subseteq IS'(A)$ for every $A \in N$ and $is \in IS(A)$
2. The attribute grammar with the following dependency graphs is noncircular but not strongly noncircular (since $IS(A) = \{(\beta_2, \alpha_1), (\beta_1, \alpha_2)\}$ and $IS'(A) = \{(\beta_2, \alpha_1), (\beta_1, \alpha_2)\}$):

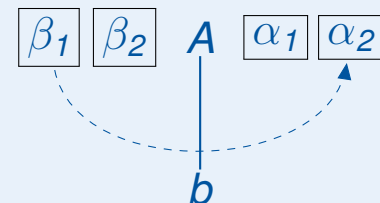
$D_{S \rightarrow A}$:



$D_{A \rightarrow a}$:



$D_{A \rightarrow b}$:



□

Attribute Evaluation by Recursive Functions

Attribute Evaluation by Recursive Functions I

Restriction: only for **strongly noncircular attribute grammars**

Attribute Evaluation by Recursive Functions

Attribute Evaluation by Recursive Functions I

Restriction: only for **strongly noncircular attribute grammars**

Principle: for every $A \in N$ and $\alpha \in \text{syn}(A)$, define **evaluation function** $\langle \alpha, A \rangle$

- parameters of $\langle \alpha, A \rangle$:
 - the **node k of t** where α has to be evaluated (which is labelled by A) and
 - all **inherited attributes of A** on which α (potentially) depends ($\{\beta \in \text{Inh} \mid (\beta, \alpha \in \text{IS}'(A))\}$)
- implementation of $\langle \alpha, A \rangle$: derived from semantic rule $\alpha.0 = f(\gamma_1.i_1, \dots, \gamma_n.i_n) \in E_\pi$, where $\pi = A \rightarrow X_1 \dots X_r$ is the production applied at node k
 - $\gamma_j \in \text{Inh}, i_j = 0$: use parameter γ of $\langle \alpha, A \rangle$
 - $\gamma_j \in \text{Syn}, 1 \leq i_j \leq r$:
 - $X_{i_j} \in N$: call $\langle \gamma_j, X_{i_j} \rangle$ with parameters determined by rules in E_π for $\text{inh}(X_{i_j})$
 - $X_{i_j} \in \Sigma$: use $v(\gamma_j.\text{child}_{i_j}(k))$ (where $\text{child}_i(k) = i$ th child of k)

Attribute Evaluation by Recursive Functions

Attribute Evaluation by Recursive Functions I

Restriction: only for **strongly noncircular attribute grammars**

Principle: for every $A \in N$ and $\alpha \in \text{syn}(A)$, define **evaluation function** $\langle \alpha, A \rangle$

- parameters of $\langle \alpha, A \rangle$:
 - the **node k of t** where α has to be evaluated (which is labelled by A) and
 - all **inherited attributes of A** on which α (potentially) depends ($\{\beta \in \text{Inh} \mid (\beta, \alpha \in \text{IS}'(A))\}$)
- implementation of $\langle \alpha, A \rangle$: derived from semantic rule $\alpha.0 = f(\gamma_1.i_1, \dots, \gamma_n.i_n) \in E_\pi$, where $\pi = A \rightarrow X_1 \dots X_r$ is the production applied at node k
 - $\gamma_j \in \text{Inh}, i_j = 0$: use parameter γ of $\langle \alpha, A \rangle$
 - $\gamma_j \in \text{Syn}, 1 \leq i_j \leq r$:
 - $X_{i_j} \in N$: call $\langle \gamma_j, X_{i_j} \rangle$ with parameters determined by rules in E_π for $\text{inh}(X_{i_j})$
 - $X_{i_j} \in \Sigma$: use $v(\gamma_j.\text{child}_{i_j}(k))$ (where $\text{child}_i(k) = i$ th child of k)

Evaluation: given a syntax tree t with root k_0 and $v : \text{Syn}_\Sigma \rightarrow V$, **evaluate** $\langle \alpha, S \rangle(k_0)$
for every $\alpha \in \text{syn}(S)$

Attribute Evaluation by Recursive Functions

Attribute Evaluation by Recursive Functions I

Restriction: only for **strongly noncircular attribute grammars**

Principle: for every $A \in N$ and $\alpha \in \text{syn}(A)$, define **evaluation function** $\langle \alpha, A \rangle$

- parameters of $\langle \alpha, A \rangle$:
 - the **node k of t** where α has to be evaluated (which is labelled by A) and
 - all **inherited attributes of A** on which α (potentially) depends ($\{\beta \in \text{Inh} \mid (\beta, \alpha \in \text{IS}'(A))\}$)
- implementation of $\langle \alpha, A \rangle$: derived from semantic rule $\alpha.0 = f(\gamma_1.i_1, \dots, \gamma_n.i_n) \in E_\pi$, where $\pi = A \rightarrow X_1 \dots X_r$ is the production applied at node k
 - $\gamma_j \in \text{Inh}, i_j = 0$: use parameter γ of $\langle \alpha, A \rangle$
 - $\gamma_j \in \text{Syn}, 1 \leq i_j \leq r$:
 - $X_{i_j} \in N$: call $\langle \gamma_j, X_{i_j} \rangle$ with parameters determined by rules in E_π for $\text{inh}(X_{i_j})$
 - $X_{i_j} \in \Sigma$: use $v(\gamma_j.\text{child}_{i_j}(k))$ (where $\text{child}_i(k) = i$ th child of k)

Evaluation: given a syntax tree t with root k_0 and $v : \text{Syn}_\Sigma \rightarrow V$, **evaluate** $\langle \alpha, S \rangle(k_0)$ **for every** $\alpha \in \text{syn}(S)$

Result: evaluates synthesized attribute variables at root of t and all attribute variables on which they depend

- **eager** evaluation: **possible** dependencies (according to IS' sets)
- **lazy** evaluation: **actual** dependencies (according to E_t)

Attribute Evaluation by Recursive Functions

Attribute Evaluation by Recursive Functions II

Example 14.6 (Attribute grammar for type checking; cf. Example 12.1)

$Cmd \rightarrow \varepsilon$	$ok.0 = true$		
$var := Exp; Cmd$	$ok.0 = (env.0(id.1) = typ.3 \wedge ok.5)$	$env.3 = env.0$	$env.5 = env.0$
$Exp \rightarrow num$	$typ.0 = int$		
var	$typ.0 = env.0(id.1)$		
$Exp + Exp$	$typ.0 = (typ.1 = typ.3 = int ? int : err)$	$env.1 = env.0$	$env.3 = env.0$
\vdots			

Attribute Evaluation by Recursive Functions

Attribute Evaluation by Recursive Functions II

Example 14.6 (Attribute grammar for type checking; cf. Example 12.1)

$Cmd \rightarrow \varepsilon$	$ok.0 = \text{true}$		
$ \text{ var := Exp; Cmd}$	$ok.0 = (env.0(id.1) = \text{typ}.3 \wedge ok.5)$	$env.3 = env.0$	$env.5 = env.0$
$Exp \rightarrow \text{num}$	$typ.0 = \text{int}$		
$ \text{ var}$	$typ.0 = env.0(id.1)$		
$ \text{ Exp + Exp}$	$typ.0 = (typ.1 = typ.3 = \text{int} ? \text{int} : \text{err})$	$env.1 = env.0$	$env.3 = env.0$
\vdots			

yields

$\langle ok, Cmd \rangle(k, env) = \text{case production}(k) \text{ of}$

$Cmd \rightarrow \varepsilon :$

true

$Cmd \rightarrow \text{var := Exp; Cmd} : env(v(id.child_1(k))) = \langle typ, Exp \rangle(child_3(k), env) \wedge \langle ok, Cmd \rangle(child_5(k), env)$

$\langle typ, Exp \rangle(k, env) = \text{case production}(k) \text{ of}$

$Exp \rightarrow \text{num} :$

int

$Exp \rightarrow \text{var} :$

$env(v(id.child_1(k)))$

$Exp \rightarrow \text{Exp + Exp} : (\langle typ, Exp \rangle(child_1(k), env) = \langle typ, Exp \rangle(child_3(k), env) = \text{int} ? \text{int} : \text{err})$

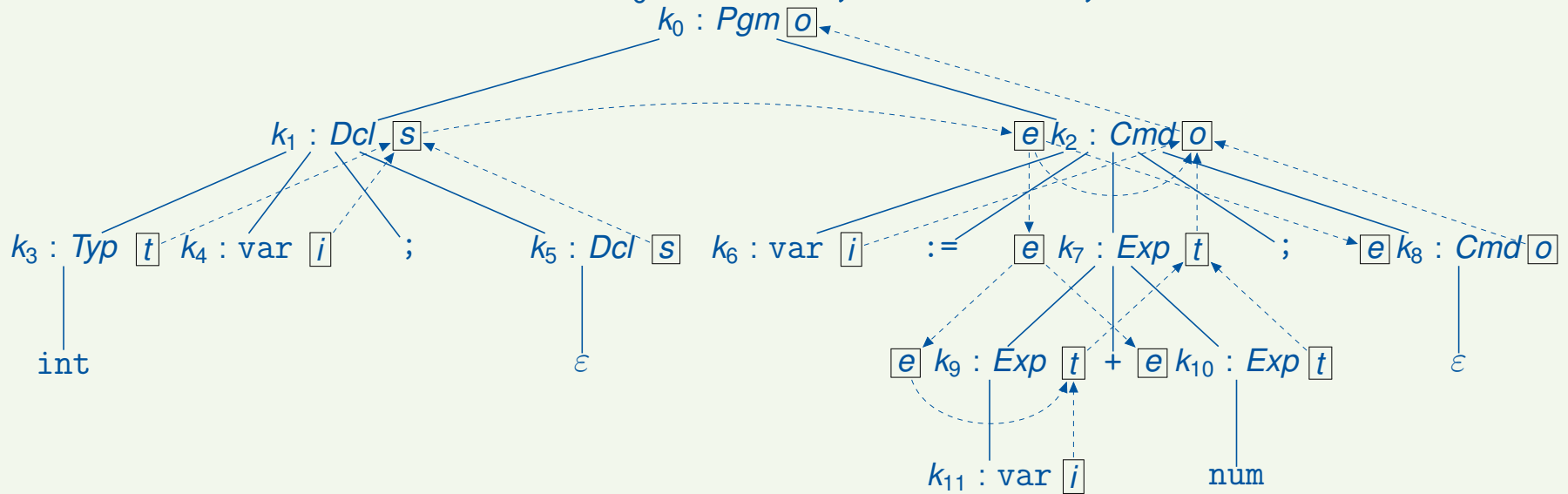
(where $production(k) =$ production applied at k and $child_i(k) =$ i th child of node k)

Attribute Evaluation by Recursive Functions

Attribute Evaluation by Recursive Functions III

Example 14.7 (cf. Example 12.16)

Evaluation of *ok* attribute at root k_0 of `int x; x := x+1;;`

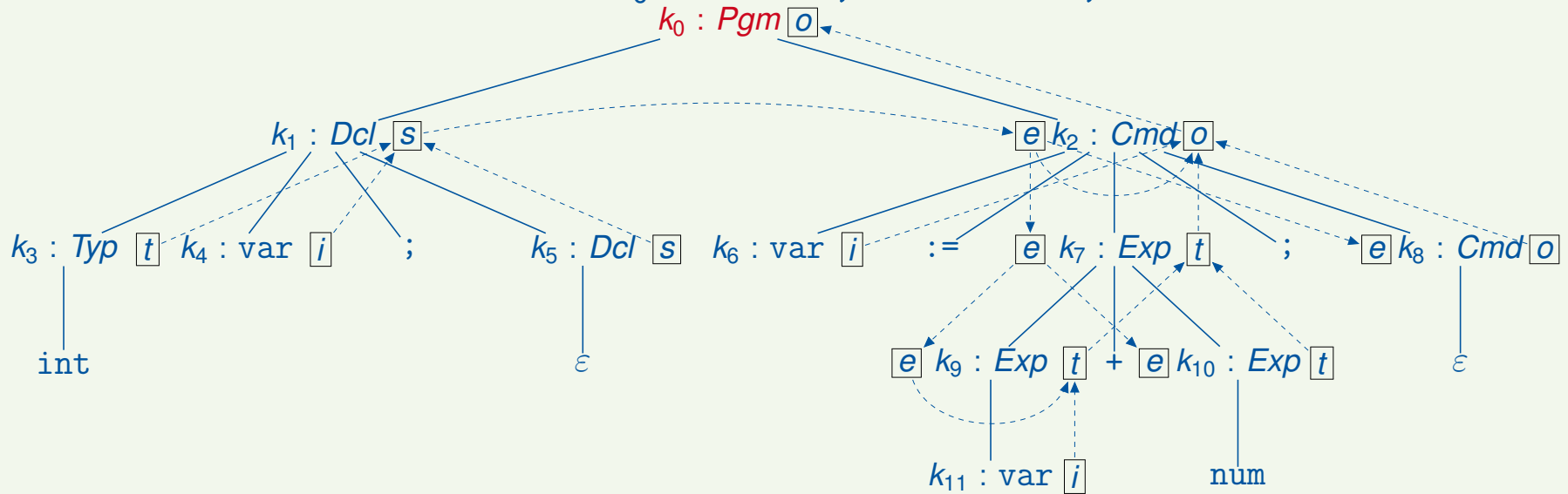


Attribute Evaluation by Recursive Functions

Attribute Evaluation by Recursive Functions III

Example 14.7 (cf. Example 12.16)

Evaluation of *ok* attribute at root k_0 of `int x; x := x+1;;`



Production: $Pgm \rightarrow Dcl\ Cmd$

$ok.0 = ok.2 \quad env.2 = st.1$

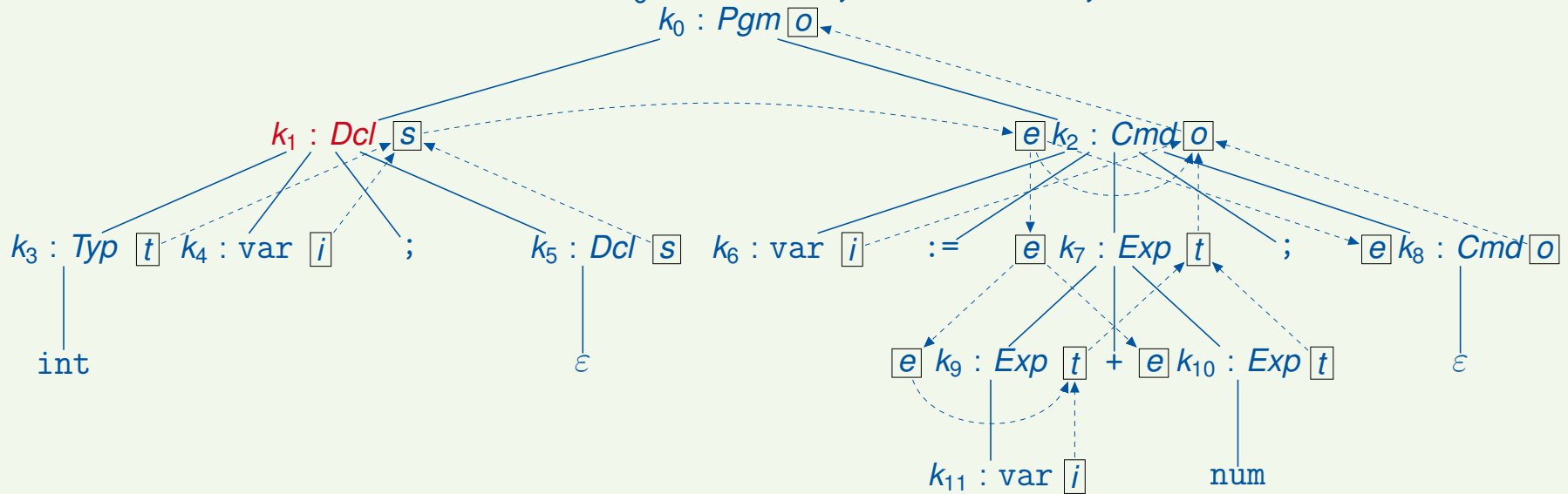
Evaluation: $\langle ok, Pgm \rangle(k_0) = \langle ok, Cmd \rangle(k_2, \langle st, Dcl \rangle(k_1))$

Attribute Evaluation by Recursive Functions

Attribute Evaluation by Recursive Functions III

Example 14.7 (cf. Example 12.16)

Evaluation of *ok* attribute at root k_0 of `int x; x := x+1;;`



Production: $Dcl \rightarrow Typ \text{ var } ; Dcl$
 $st.0 = st.4[id.2 \mapsto typ.1]$

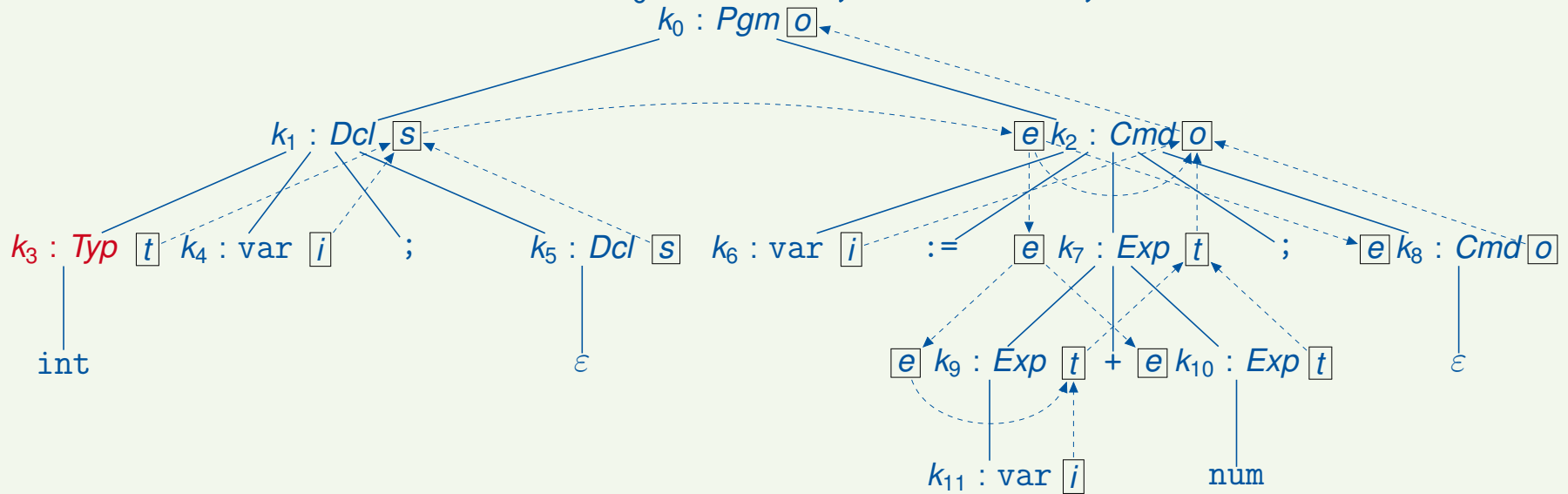
Evaluation: $\langle ok, Pgm \rangle(k_0) = \langle ok, Cmd \rangle(k_2, \langle st, Dcl \rangle(k_5)[v(id.k_4) \mapsto \langle typ, Typ \rangle(k_3)])$

Attribute Evaluation by Recursive Functions

Attribute Evaluation by Recursive Functions III

Example 14.7 (cf. Example 12.16)

Evaluation of *ok* attribute at root k_0 of `int x; x := x+1;;`



Production: $Typ \rightarrow int$
 $typ.0 = int$

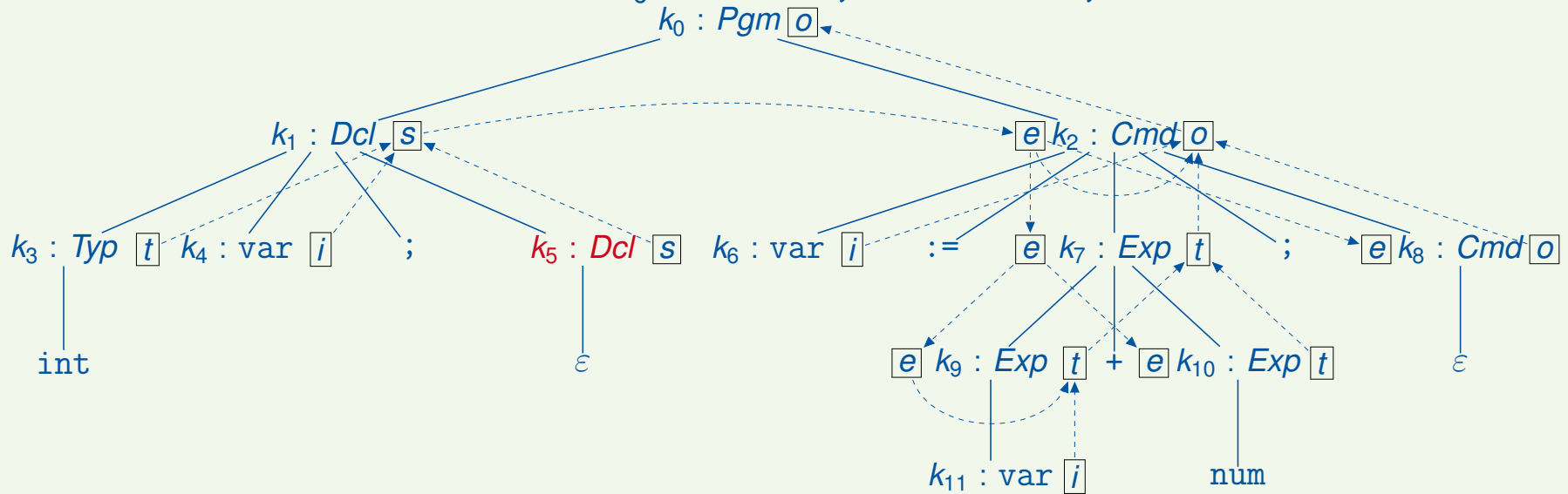
Evaluation: $\langle ok, Pgm \rangle(k_0) = \langle ok, Cmd \rangle(k_2, \langle st, Dcl \rangle(k_5)[x \mapsto int])$

Attribute Evaluation by Recursive Functions

Attribute Evaluation by Recursive Functions III

Example 14.7 (cf. Example 12.16)

Evaluation of *ok* attribute at root k_0 of `int x; x := x+1;;`



Production: $Dcl \rightarrow \varepsilon$

$st.0 = [id \mapsto err \mid id \in Id]$

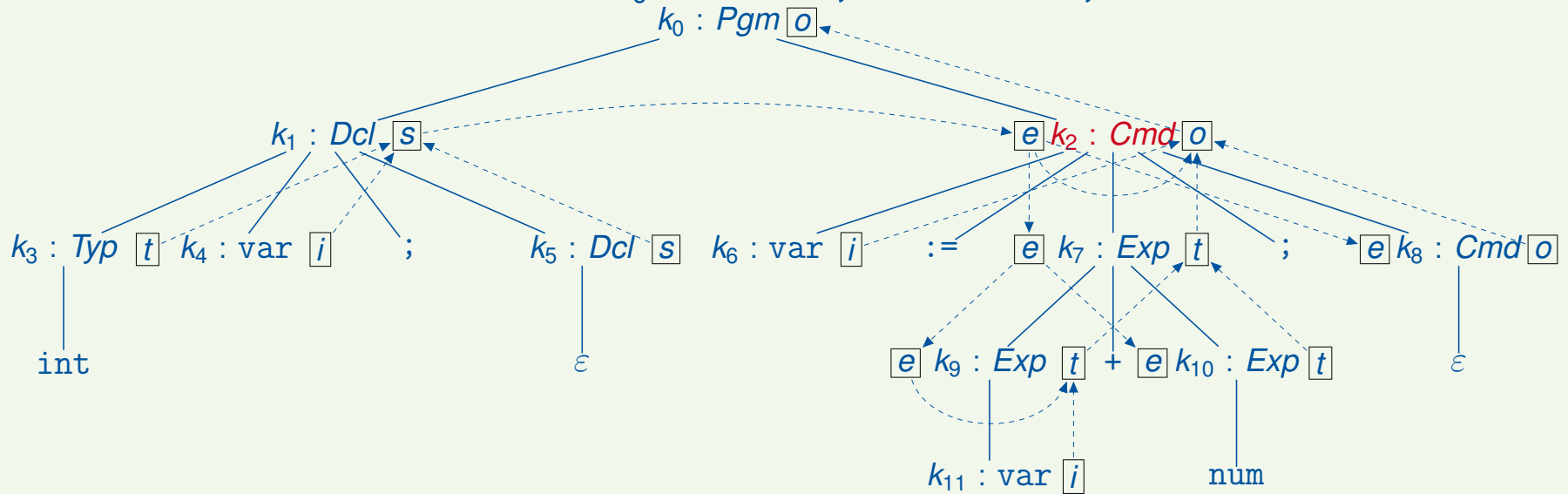
Evaluation: $\langle ok, Pgm \rangle(k_0) = \langle ok, Cmd \rangle(k_2, \eta)$ for $\eta := [x \mapsto int]$

Attribute Evaluation by Recursive Functions

Attribute Evaluation by Recursive Functions III

Example 14.7 (cf. Example 12.16)

Evaluation of *ok* attribute at root k_0 of `int x; x := x+1;;`



Production: $Cmd \rightarrow var := Exp; Cmd$

$ok.0 = (env.0(id.1) = typ.3 \wedge ok.5) \quad env.3 = env.0 \quad env.5 = env.0$

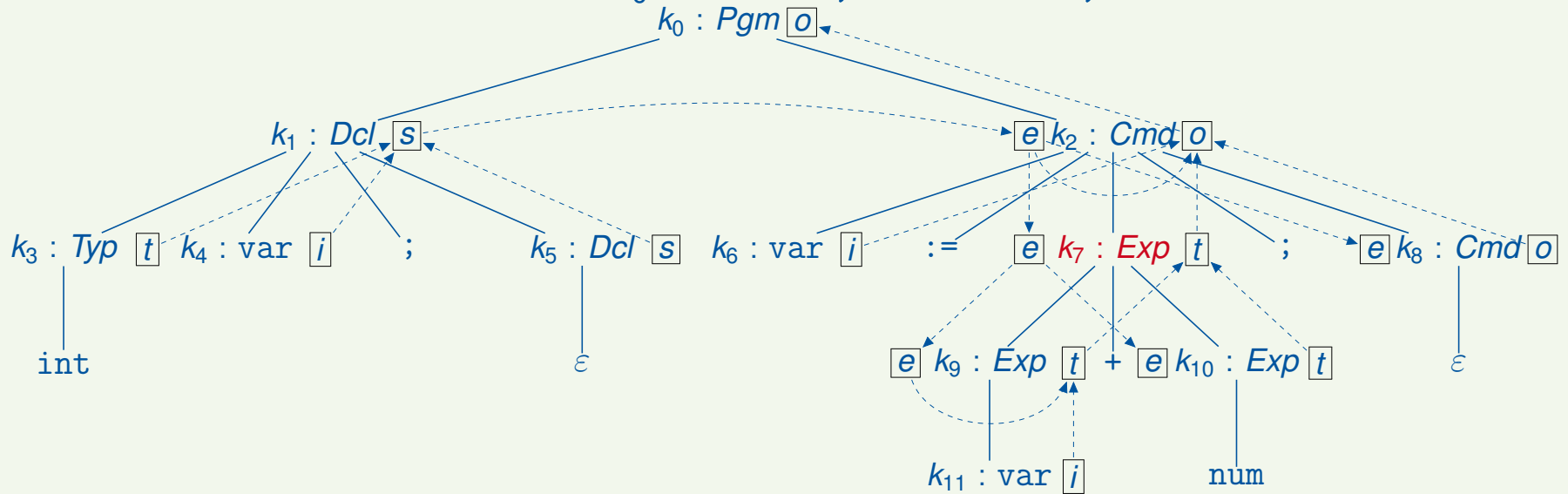
Evaluation: $\langle ok, Pgm \rangle(k_0) = (\eta(v(id.k_6)) = \langle typ, Exp \rangle(k_7, \eta) \wedge \langle ok, Cmd \rangle(k_8, \eta))$

Attribute Evaluation by Recursive Functions

Attribute Evaluation by Recursive Functions III

Example 14.7 (cf. Example 12.16)

Evaluation of *ok* attribute at root k_0 of `int x; x := x+1;;`



Production: $Exp \rightarrow Exp + Exp$

$typ.0 = (typ.1 = typ.3 = \text{int} ? \text{int} : \text{err}) \quad env.1 = env.0 \quad env.3 = env.0$

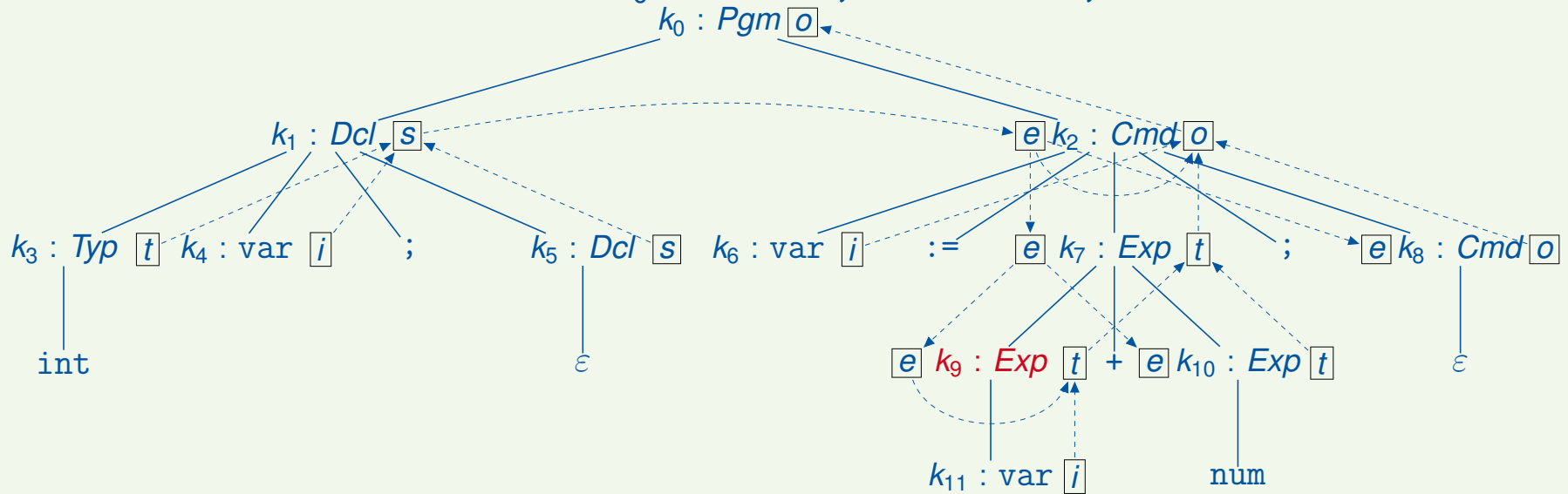
Evaluation: $\langle ok, Pgm \rangle(k_0) = (\text{int} = (\langle typ, Exp \rangle(k_9, \eta) = \langle typ, Exp \rangle(k_{10}, \eta) ? \text{int} : \text{err}) \wedge \langle ok, Cmd \rangle(k_8, \eta))$

Attribute Evaluation by Recursive Functions

Attribute Evaluation by Recursive Functions III

Example 14.7 (cf. Example 12.16)

Evaluation of *ok* attribute at root k_0 of `int x; x := x+1;;`



Production: $Exp \rightarrow var$

$typ.0 = env.0(id.1)$

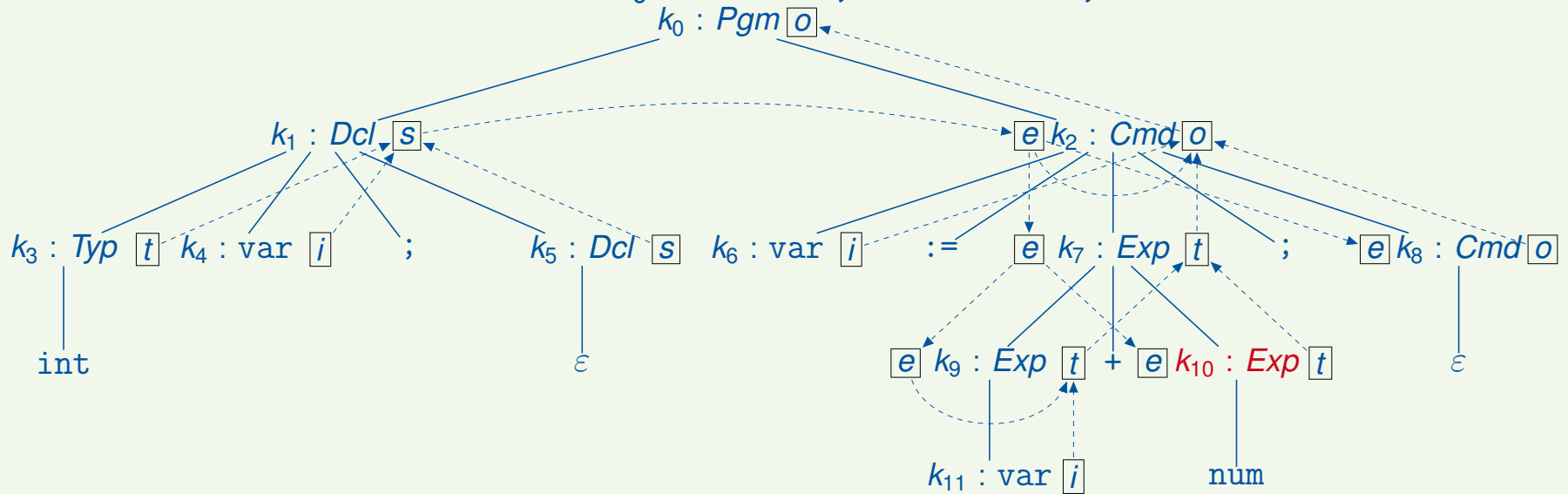
Evaluation: $\langle ok, Pgm \rangle(k_0) = (int = (\eta(v(id.k_{11}))) = \langle typ, Exp \rangle(k_{10}, \eta) ? int : err) \wedge \langle ok, Cmd \rangle(k_8, \eta)$

Attribute Evaluation by Recursive Functions

Attribute Evaluation by Recursive Functions III

Example 14.7 (cf. Example 12.16)

Evaluation of *ok* attribute at root k_0 of `int x; x := x+1;;`



Production: $Exp \rightarrow num$
 $typ.0 = int$

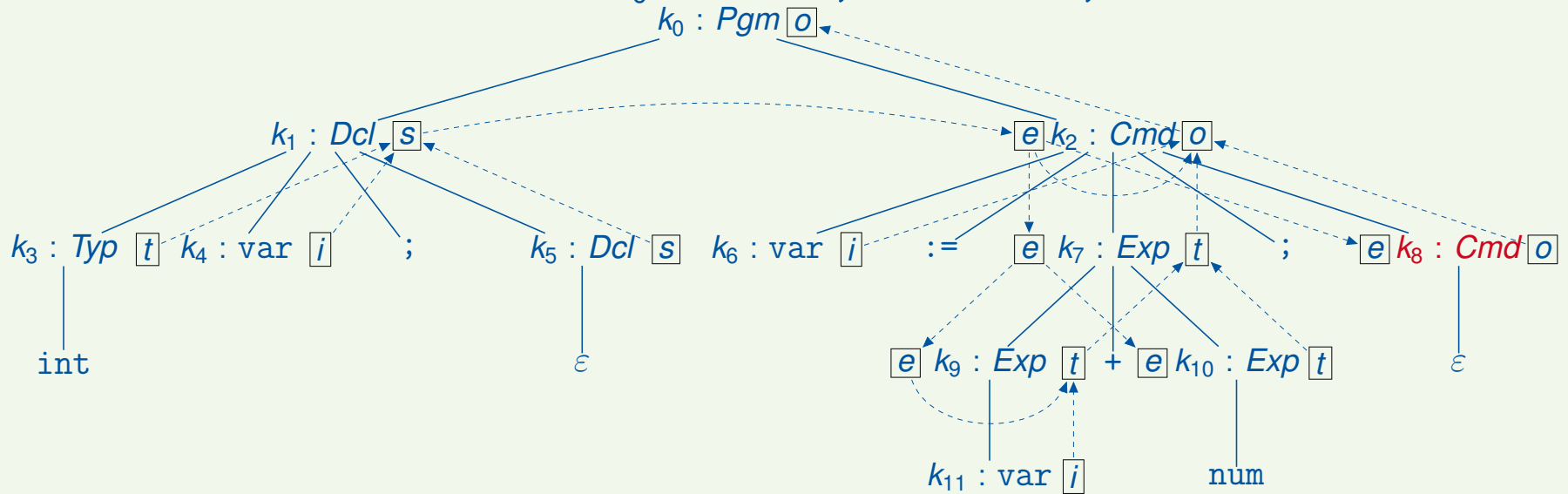
Evaluation: $\langle ok, Pgm \rangle(k_0) = (int = (int = int ? int : err) \wedge \langle ok, Cmd \rangle(k_8, \eta))$

Attribute Evaluation by Recursive Functions

Attribute Evaluation by Recursive Functions III

Example 14.7 (cf. Example 12.16)

Evaluation of *ok* attribute at root k_0 of `int x; x := x+1;;`



Production: $Cmd \rightarrow \varepsilon$
 $ok.0 = true$

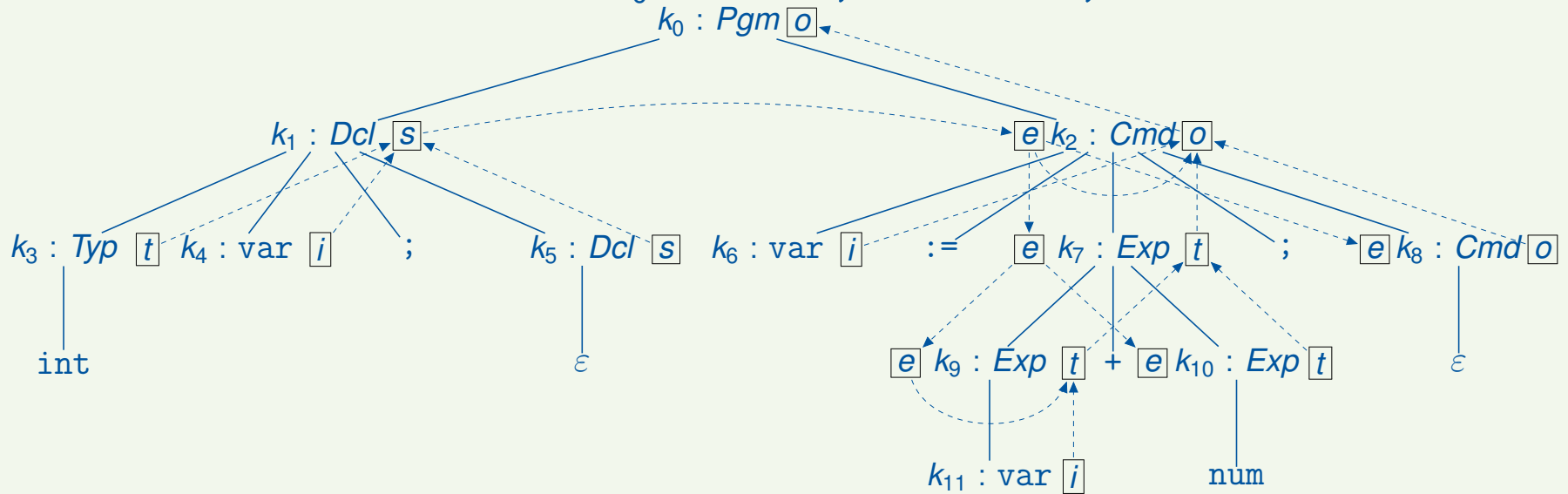
Evaluation: $\langle ok, Pgm \rangle(k_0) = (int = int \wedge true)$

Attribute Evaluation by Recursive Functions

Attribute Evaluation by Recursive Functions III

Example 14.7 (cf. Example 12.16)

Evaluation of *ok* attribute at root k_0 of `int x; x := x+1;;`



Production:

Evaluation: $\langle ok, Pgm \rangle(k_0) = \text{true}$

L-Attributed Grammars

Outline of Lecture 14

Recap: The Circularity Check

Attribute Evaluation

Attribute Evaluation by Topological Sorting

Attribute Evaluation by Recursive Functions

L-Attributed Grammars

L-Attributed Grammars

L-Attributed Grammars I

In an L-attributed grammar, attribute dependencies on the right-hand sides of productions are only allowed to run **from left to right**.

L-Attributed Grammars

L-Attributed Grammars I

In an L-attributed grammar, attribute dependencies on the right-hand sides of productions are only allowed to run **from left to right**.

Definition 14.8 (L-attributed grammar)

Let $\mathfrak{A} = \langle G, E, V \rangle \in AG$ such that, for every $\pi \in P$ and $\beta.i = f(\dots, \alpha.j, \dots) \in E_\pi$ with $\beta \in Inh$ and $\alpha \in Syn, j < i$. Then \mathfrak{A} is called an **L-attributed grammar** (notation: $\mathfrak{A} \in LAG$).

Remark: note that no restrictions are imposed for $\beta \in Syn$ (for $i = 0$) or $\alpha \in Inh$ (for $j = 0$). Thus, in an L-attributed grammar,

- synthesized attributes of the left-hand side can depend on any external variable and
- every internal variable can depend on any inherited attribute of the left-hand side.

L-Attributed Grammars

L-Attributed Grammars I

In an L-attributed grammar, attribute dependencies on the right-hand sides of productions are only allowed to run **from left to right**.

Definition 14.8 (L-attributed grammar)

Let $\mathfrak{A} = \langle G, E, V \rangle \in AG$ such that, for every $\pi \in P$ and $\beta.i = f(\dots, \alpha.j, \dots) \in E_\pi$ with $\beta \in Inh$ and $\alpha \in Syn, j < i$. Then \mathfrak{A} is called an **L-attributed grammar** (notation: $\mathfrak{A} \in LAG$).

Remark: note that no restrictions are imposed for $\beta \in Syn$ (for $i = 0$) or $\alpha \in Inh$ (for $j = 0$). Thus, in an L-attributed grammar,

- synthesized attributes of the left-hand side can depend on any external variable and
- every internal variable can depend on any inherited attribute of the left-hand side.

Corollary 14.9

Every $\mathfrak{A} \in LAG$ is noncircular.

L-Attributed Grammars

L-Attributed Grammars II

Example 14.10

L-attributed grammar:

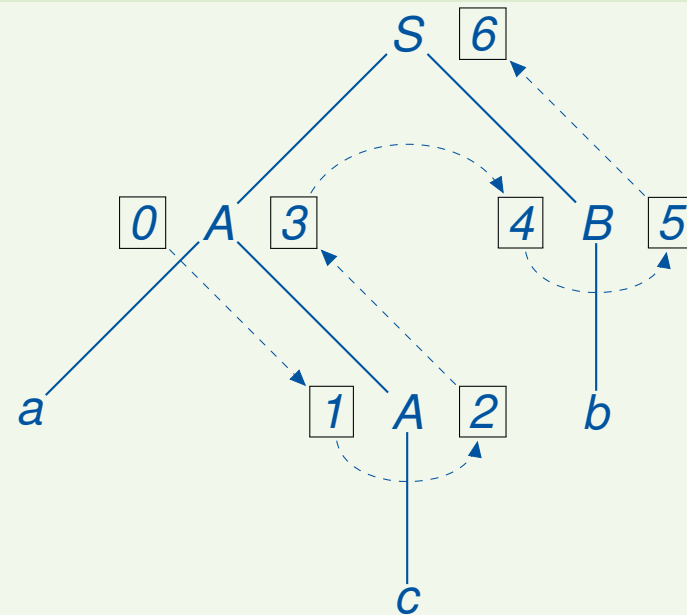
$$\begin{aligned} S &\rightarrow AB & i.1 &= 0 \\ & & i.2 &= s.1 + 1 \\ & & s.0 &= s.2 + 1 \\ A &\rightarrow aA & i.2 &= i.0 + 1 \\ & & s.0 &= s.2 + 1 \\ A &\rightarrow c & s.0 &= i.0 + 1 \\ B &\rightarrow b & s.0 &= i.0 + 1 \end{aligned}$$

L-Attributed Grammars

L-Attributed Grammars II

Example 14.10

L-attributed grammar:

$$\begin{aligned} S &\rightarrow AB & i.1 &= 0 \\ & & i.2 &= s.1 + 1 \\ & & s.0 &= s.2 + 1 \\ A &\rightarrow aA & i.2 &= i.0 + 1 \\ & & s.0 &= s.2 + 1 \\ A &\rightarrow c & s.0 &= i.0 + 1 \\ B &\rightarrow b & s.0 &= i.0 + 1 \end{aligned}$$


Evaluation of L-Attributed Grammars

Observation 1: the syntax tree of an L-attributed grammar can be attributed by a **depth-first, left-to-right tree traversal** with **two visits to each node**

1. **top-down**: evaluation of **inherited** attributes
2. **bottom-up**: evaluation of **synthesized** attributes

Evaluation of L-Attributed Grammars

Observation 1: the syntax tree of an L-attributed grammar can be attributed by a **depth-first, left-to-right tree traversal** with **two visits to each node**

1. **top-down**: evaluation of **inherited** attributes
2. **bottom-up**: evaluation of **synthesized** attributes

Observation 2: visit sequence fits nicely with **parsing**

1. **top-down**: expansion steps
2. **bottom-up**: reduction steps

L-Attributed Grammars

Evaluation of L-Attributed Grammars

Observation 1: the syntax tree of an L-attributed grammar can be attributed by a **depth-first, left-to-right tree traversal** with **two visits to each node**

1. **top-down**: evaluation of **inherited** attributes
2. **bottom-up**: evaluation of **synthesized** attributes

Observation 2: visit sequence fits nicely with **parsing**

1. **top-down**: expansion steps
2. **bottom-up**: reduction steps

Idea: extend LL parsing to support reduction steps, and integrate attribute evaluation

- use **recursive-descent parser** and
- add variables and operations for **attribute evaluation**

Recursive-Descent Parsing I

- Ingredients:
- variable `token` for current token
 - function `next()` for invoking the scanner
 - procedure `print(i)` for displaying the leftmost analysis (or errors)

Recursive-Descent Parsing I

- Ingredients:**
- variable `token` for current token
 - function `next()` for invoking the scanner
 - procedure `print(i)` for displaying the leftmost analysis (or errors)

Method: to every $A \in N$ we assign a procedure

`A()`

which

- tests `token` with regard to the lookahead sets of the A -productions,
- prints the corresponding rule number and
- evaluates the corresponding right-hand side as follows:
 - for $a \in \Sigma$: check `token`; call `next()`
 - for $A \in N$: call `A`

Recursive-Descent Parsing **and Attribute Evaluation I**

- Ingredients:**
- variable `token` for current token
 - function `next()` for invoking the scanner
 - procedure `print(i)` for displaying the leftmost analysis (or errors)

Method: to every $A \in N$ we assign a procedure

$A(\text{in: inh}(A), \text{out: syn}(A))$

which

- declares local variables for synthesized attributes on right-hand sides,
- tests `token` with regard to the lookahead sets of the A -productions,
- prints the corresponding rule number and
- evaluates the corresponding right-hand side as follows:
 - for $a \in \Sigma$: check `token`; call `next()`
 - for $A \in N$: call A with appropriate parameters

Recursive-Descent Parsing II

Example 14.10 (cf. Example 14.10)

```
proc main();
  token := next(); S()
proc S();
  if token in {'a','c'} then    (* S → AB *)
    print(1); A(); B()
  else print(error); stop fi
proc A();
  if token = 'a' then    (* A → aA *)
    print(2); token := next(); A()
  elsif token = 'c' then (* A → c *)
    print(3); token := next()
  else print(error); stop fi
proc B();
  if token = 'b' then    (* B → b *)
    print(4); token := next()
  else print(error); stop fi
```

Recursive-Descent Parsing and Attribute Evaluation II

Example 14.11 (cf. Example 14.10)

```
proc main(); var s;
  token := next(); S(s); print(s)
proc S(out s0); var s1,s2;
  if token in {'a','c'} then    (* S → AB: i.1 = 0, i.2 = s.1 + 1, s.0 = s.2 + 1 *)
    print(1); A(0,s1); B(s1+1,s2); s0 := s2+1
  else print(error); stop fi
proc A(in i0,out s0); var s2;
  if token = 'a' then    (* A → aA: i.2 = i.0 + 1, s.0 = s.2 + 1 *)
    print(2); token := next(); A(i0+1,s2); s0 := s2+1
  elsif token = 'c' then  (* A → c: s.0 = i.0 + 1 *)
    print(3); token := next(); s0 := i0+1
  else print(error); stop fi
proc B(in i0,out s0);
  if token = 'b' then    (* B → b: s.0 = i.0 + 1 *)
    print(4); token := next(); s0 := i0+1
  else print(error); stop fi
```