



# Compiler Construction

## Lecture 14: Semantic Analysis IV (Attribute Evaluation)

Summer Semester 2017

Thomas Noll

Software Modeling and Verification Group

RWTH Aachen University

<https://moves.rwth-aachen.de/teaching/ss-17/cc/>

# 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  $Var_\pi$  into  $In_\pi$  and  $Out_\pi$ , cyclic dependencies cannot occur at production level.

## Recap: The Circularity Check

---

### Attribute Dependency Graphs and Circularity I

**Observation:** a cycle in the dependency graph  $D_t$  of a given syntax tree  $t$  is caused by the occurrence of a “cover” production  $\pi = A_0 \rightarrow w_0 A_1 w_1 \dots A_r w_r \in P$  in a node  $k_0$  of  $t$  such that

- the dependencies in  $E_{k_0}$  yield the “upper end” of the cycle and
- for at least one  $i \in [r]$ , some attributes in  $\text{syn}(A_i)$  depend on attributes in  $\text{inh}(A_i)$ .

To identify such “critical” situations we need to determine for each  $i \in [r]$  the possible ways in which attributes in  $\text{syn}(A_i)$  can depend on attributes in  $\text{inh}(A_i)$ .

# Recap: The Circularity Check

## Attribute Dependency Graphs and Circularity II

### Definition (Attribute dependence)

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

- 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  $\alpha$  is dependent on  $\beta$  below  $A$  in  $t$  (notation:  $\beta \xrightarrow{A} \alpha$ ).
- For every syntax tree  $t$  with root label  $A \in N$ ,

$$is(A, t) := \{(\beta, \alpha) \in \text{inh}(A) \times \text{syn}(A) \mid \beta \xrightarrow{A} \alpha \text{ in } t\}.$$

- For every  $A \in N$ ,

$$IS(A) := \{is(A, t) \mid t \text{ syntax tree with root label } A\} \subseteq 2^{\text{Inh} \times \text{Syn}}.$$

**Remark:** it is important that  $IS(A)$  is a **system** of attribute dependence sets, not a **union** (otherwise: **strong noncircularity** – later).

# 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

---

## 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  $D_t$ :

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

2. **Strongly noncircular** AGs: using **recursive functions**

3. **L-attributed** grammars: integration with top-down parsing

4. **S-attributed grammars** (i.e.,  $Inh = \emptyset$ ): yacc/ANTLR

# 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  $v : \text{Var}_t \rightarrow V$

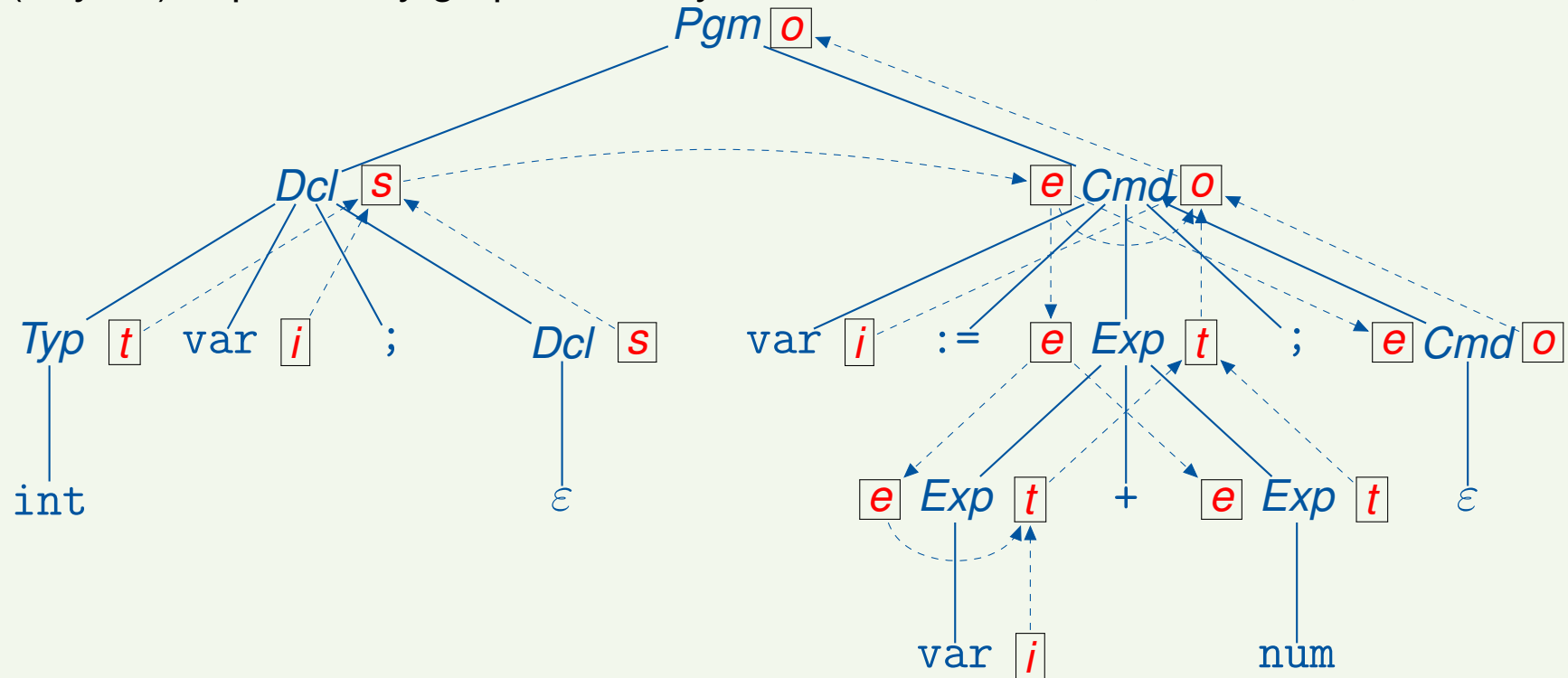
**Remark:** noncircularity guarantees that in step 2.i at least one such  $x$  is available

# Attribute Evaluation by Topological Sorting

## Attribute Evaluation by Topological Sorting II

### Example 14.2 (cf. Example 13.6)

(Acyclic) dependency graph of the syntax tree for `int x; x := x+1;`





# 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  $\alpha$  is **dependent on  $\beta$  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 directly computable in polynomial time

# Attribute Evaluation by Recursive Functions

## Strong Noncircularity II

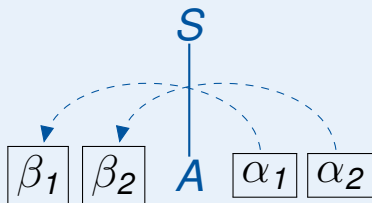
### Lemma 14.5

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

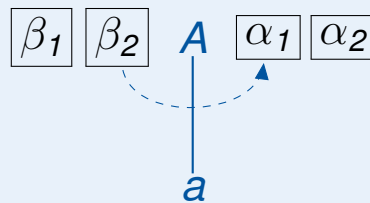
### Proof.

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  $IS'(A) = \{(\beta_2, \alpha_1), (\beta_1, \alpha_2)\}$ ):

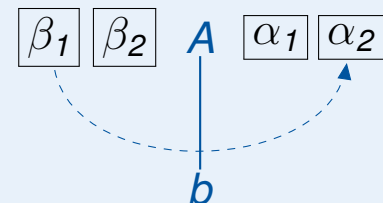
$D_{S \rightarrow A}$ :



$D_{A \rightarrow a}$ :



$D_{A \rightarrow b}$ :



# Attribute Evaluation by Recursive Functions

---

## Attribute Evaluation by Recursive Functions I

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

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

- parameters of  $g_{A,\alpha}$ :
  - the **node**  $k$  of  $t$  where  $\alpha$  has to be evaluated (which is labelled by  $A$ ) and
  - all **inherited attributes** of  $A$  on which  $\alpha$  (potentially) depends (i.e.,  $\{\beta \in \text{Inh} \mid (\beta, \alpha \in \text{IS}'(A))\}$ )
- implementation of  $g_{A,\alpha}$ : derived from semantic rule  $\alpha.0 = f(\gamma_1.i_1, \dots, \gamma_n.i_n) \in E_\pi$ , where  $\pi = A \rightarrow X_1 \dots X_k$  is the production applied at node  $k$ 
  - $\gamma_j \in \text{Inh}, i_j = 0$ : use parameter  $\gamma$  of  $g_{A,\alpha}$
  - $\gamma_j \in \text{Syn}, 1 \leq i_j \leq k$ : call  $g_{X_j, \gamma_j}$  with parameters determined by rules in  $E_\pi$  for  $\text{inh}(X_j)$

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

**Result:** evaluates synthesized attribute variables at root of  $t$  and all attribute variables on which they actually depend (according to  $E_t$ )

- **eager** evaluation: possible dependencies (according to  $\text{IS}'$  sets)
- **lazy** evaluation: actual dependencies (as induced by semantic rules)

# Attribute Evaluation by Recursive Functions

## Attribute Evaluation by Recursive Functions II

### Example 14.6 (Attribute grammar for type checking; cf. Example 11.2)

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

yields

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

$Cmd \rightarrow \varepsilon : \quad true$

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

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

$Exp \rightarrow num : \quad int$

$Exp \rightarrow \text{var} : \quad env(v(id.child_1(k)))$

$Exp \rightarrow \text{Exp + Exp} : \quad \text{if } g_{Exp,typ}(child_1(k), env) = g_{Exp,typ}(child_3(k), env) = int \text{ then } int \text{ else } err$

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

# 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 14.7 (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 outer variable and
- every inner variable can depend on any inherited attribute of the left-hand side.

### Corollary 14.8

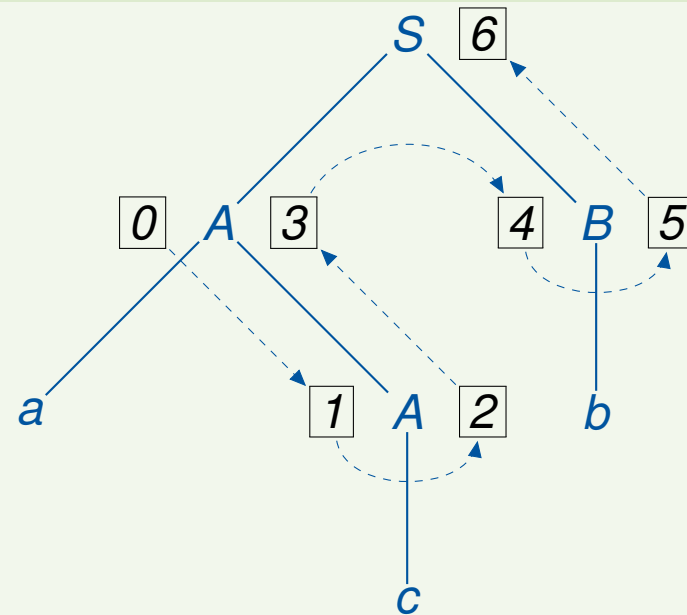
*Every  $\mathfrak{A} \in LAG$  is noncircular.*

# L-Attributed Grammars

## L-Attributed Grammars II

### Example 14.9

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

---

## 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 **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 14.9 (cf. Example 14.9)

```
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 14.10 (cf. Example 14.9)

```
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
```