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/

Software Modeling and Verification Chair
RWTH AACHEN UNIVERSITY

Recap: The Circularity Check

Circularity of Attribute Grammars

Goal: unique solvability of equation system
$\Longrightarrow$ avoid cyclic dependencies

Definition (Circularity)

An attribute grammar $\mathfrak{A} = \langle G, E, V \rangle \in AG$ 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.

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 $\pi = A \to w \in P$, then
$$is[\pi] \in IS(A)$$

ii. if $\pi = A \to w_0 A_1 w_1 \ldots A_r w_r \in P$ and $is_i \in IS(A_i)$ for every $i \in [r]$, then
$$is[\pi; is_1, \ldots, is_r] \in IS(A)$$

2. test whether $\mathfrak{A}$ is circular by checking if there exist $\pi = A \to w_0 A_1 w_1 \ldots A_r w_r \in P$ and $is_i \in IS(A_i)$ for every $i \in [r]$ such that the following relation is cyclic:
$$\to_\pi \cup \bigcup_{i=1}^{r} \{(\beta.p_i, \alpha.p_i) \mid (\beta, \alpha) \in is_i\}$$
(where $p_i := \sum_{j=1}^{i} |w_{j-1}| + i$)

Output: "yes" or "no"

Attribute Evaluation Methods

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

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

Methods:
1. Topological sorting of D_t (for arbitrary noncircular AGs):
   1. start with variables which depend at most on Syn_Σ
   2. 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 I

Algorithm 14.1 (Evaluation by topological sorting)

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

Procedure:
1. let $Var := Var_t \setminus Syn_\Sigma$ (* attributes to be evaluated *)
2. while $Var \neq \emptyset$ do
   i. let $x \in Var$ such that $\{y \in Var \mid y \to_t x\} = \emptyset$
   ii. let $x = f(x_1, \ldots, x_n) \in E_t$
   iii. let $v(x) := f(v(x_1), \ldots, v(x_n))$
   iv. let $Var := Var \setminus \{x\}$

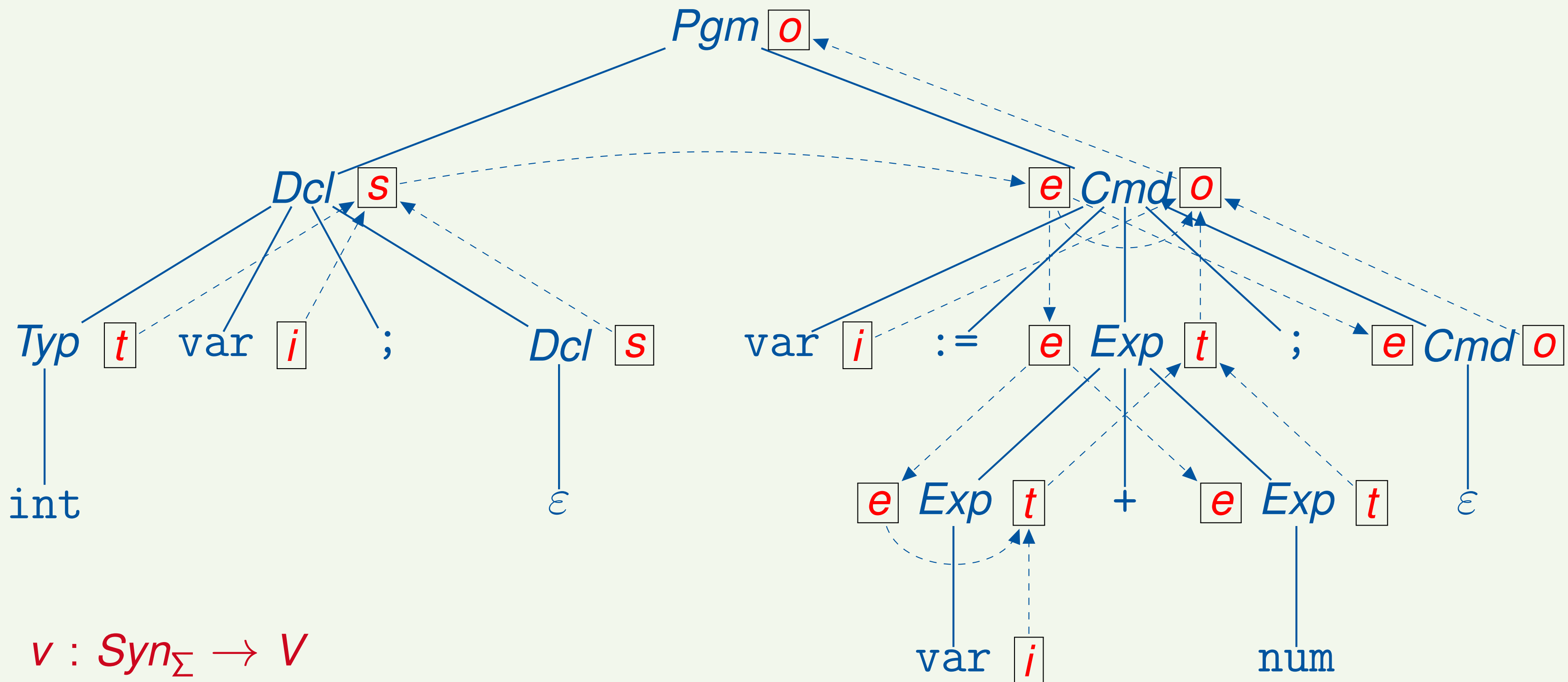
Output: solution $v : Var_t \to V$

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

Attribute Evaluation by Topological Sorting II

Example 14.2 (cf. Example 12.16)

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



$v : Syn_\Sigma \to V$

$Typ \to \texttt{int} : typ.0 = \mathrm{int}$

$Dcl \to \varepsilon : st.0 = [id \mapsto \mathrm{err} \mid id \in Id]$

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

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, $\alpha \in \mathrm{syn}(A)$, and $\beta \in \mathrm{inh}(A)$ such that $\beta.k \to_t^+ \alpha.k$, then α is dependent on β below A in t (notation: $\beta \overset{A}{\hookrightarrow} \alpha$).
- For every $A \in N$, $IS'(A) := \{(\beta, \alpha) \mid \beta \overset{A}{\hookrightarrow} \alpha \text{ in some } t \text{ with root label A}\} \subseteq Inh \times Syn$
- $\mathfrak{A}$ is strongly noncircular if for each $\pi = A \to w_0 A_1 w_1 \ldots A_r w_r \in P$ the relation

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

  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

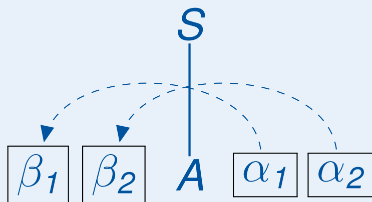
Lemma 14.5

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

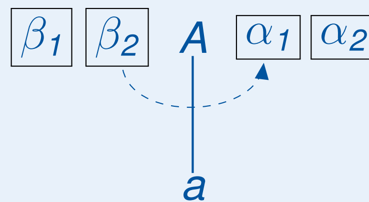
Proof.

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

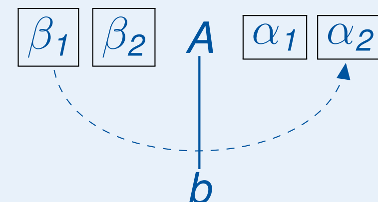
$D_{S \to A}$:



$D_{A \to a}$:



$D_{A \to b}$:



□

Attribute Evaluation by Recursive Functions I

Restriction: only for strongly noncircular attribute grammars

Principle: for every $A \in N$ and $\alpha \in \mathrm{syn}(A)$, 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 ($\{\beta \in Inh \mid (\beta, \alpha \in IS'(A))\}$)
- implementation of $\langle \alpha, A \rangle$: derived from semantic rule $\alpha.0 = f(\gamma_1.i_1, \ldots, \gamma_n.i_n) \in E_\pi$, where $\pi = A \to X_1 \ldots X_r$ is the production applied at node k
  - $\gamma_j \in Inh$, $i_j = 0$: use parameter γ of $\langle \alpha, A \rangle$
  - $\gamma_j \in Syn$, $1 \leq i_j \leq r$:
    - $X_{i_j} \in N$: call $\langle \gamma_j, X_{i_j} \rangle$ with parameters determined by rules in E_π for $\mathrm{inh}(X_{i_j})$
    - $X_{i_j} \in \Sigma$: use $v(\gamma_j.child_{i_j}(k))$ (where $child_i(k)$ = ith child of k)

Evaluation: given a syntax tree t with root k_0 and $v : Syn_\Sigma \to V$, evaluate $\langle \alpha, S \rangle(k_0)$ for every $\alpha \in \mathrm{syn}(S)$

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 II

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

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

yields

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

$Cmd \to \varepsilon :$ true

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

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

$Exp \to \texttt{num} :$ int

$Exp \to \texttt{var} :$ $env(v(id.child_1(k)))$

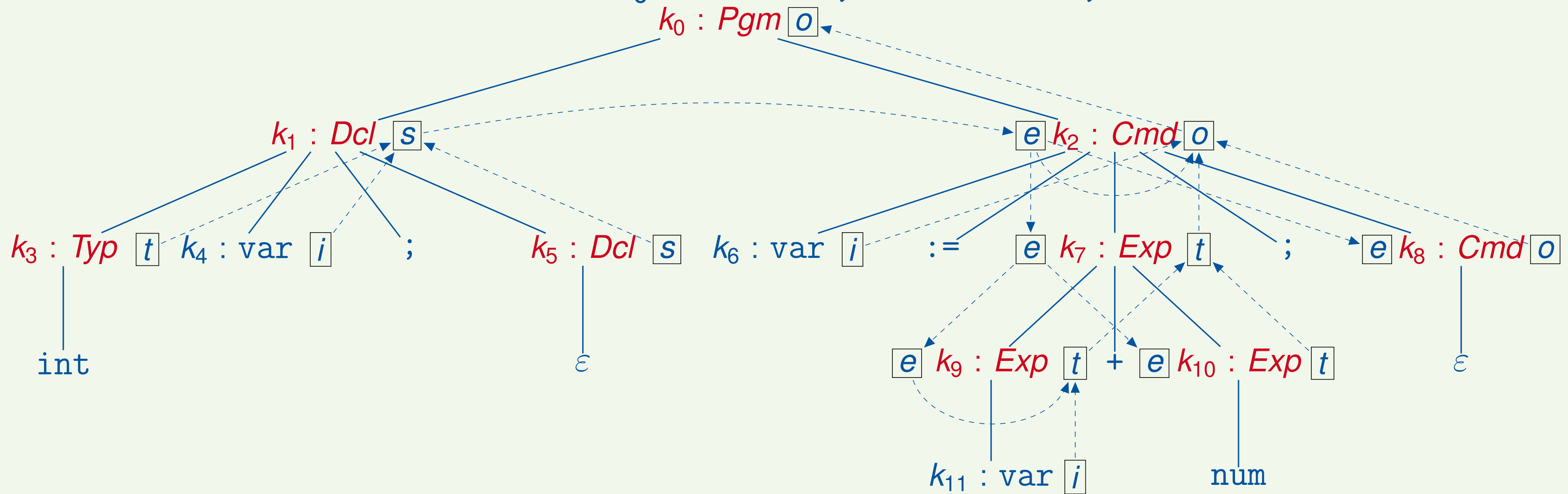
$Exp \to Exp + Exp :$ $(\langle typ, Exp\rangle(child_1(k), env) = \langle typ, Exp\rangle(child_3(k), env) = \text{int}\ ?\ \text{int} : \text{err})$

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

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:

$Pgm \to Dcl\ Cmd\ Dcl \to Typ$ `var` $; Dcl\ Typ \to$ `int` $Dcl \to \varepsilon\ Cmd \to$ `var` $:= Exp; Cmd\ Exp \to Exp + Exp$

$ok.0 = ok.2 \quad env.2 = st.1 st.0 = st.4[id.2 \mapsto typ.1] typ.0 = \text{int} st.0 = [id \mapsto \text{err} \mid id \in Id] ok.0 = (env$

Evaluation: $\langle ok, Pgm\rangle(k_0) = \langle ok, Cmd\rangle(k_2, \langle st, Dcl\rangle(k_1)) \langle ok, Cmd\rangle(k_2, \langle st, Dcl\rangle(k_5)[v(id.k_4) \mapsto \langle typ, Ty$

Software Modeling and Verification Chair
RWTH AACHEN UNIVERSITY

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 $\beta.i = f(\ldots, \alpha.j, \ldots) \in E_\pi$ 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

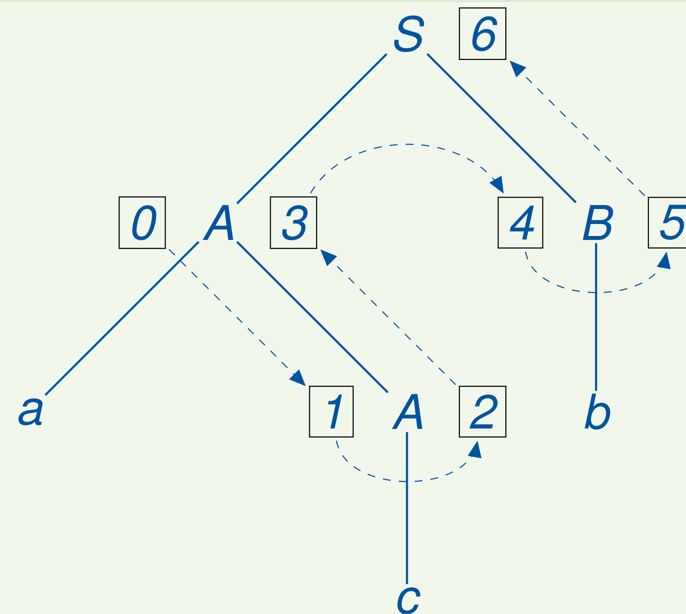
Every $\mathfrak{A} \in LAG$ is noncircular.

L-Attributed Grammars II

Example 14.10

L-attributed grammar:

$S \to AB$	$i.1 = 0$
	$i.2 = s.1 + 1$
	$s.0 = s.2 + 1$
$A \to \mathrm{a}A$	$i.2 = i.0 + 1$
	$s.0 = s.2 + 1$
$A \to \mathrm{c}$	$s.0 = i.0 + 1$
$B \to \mathrm{b}$	$s.0 = i.0 + 1$



Software Modeling and Verification Chair
RWTH AACHEN UNIVERSITY

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

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
```

L-Attributed Grammars

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
```