Compiler Construction

Lecture 12: Semantic Analysis I (Attribute Grammars)

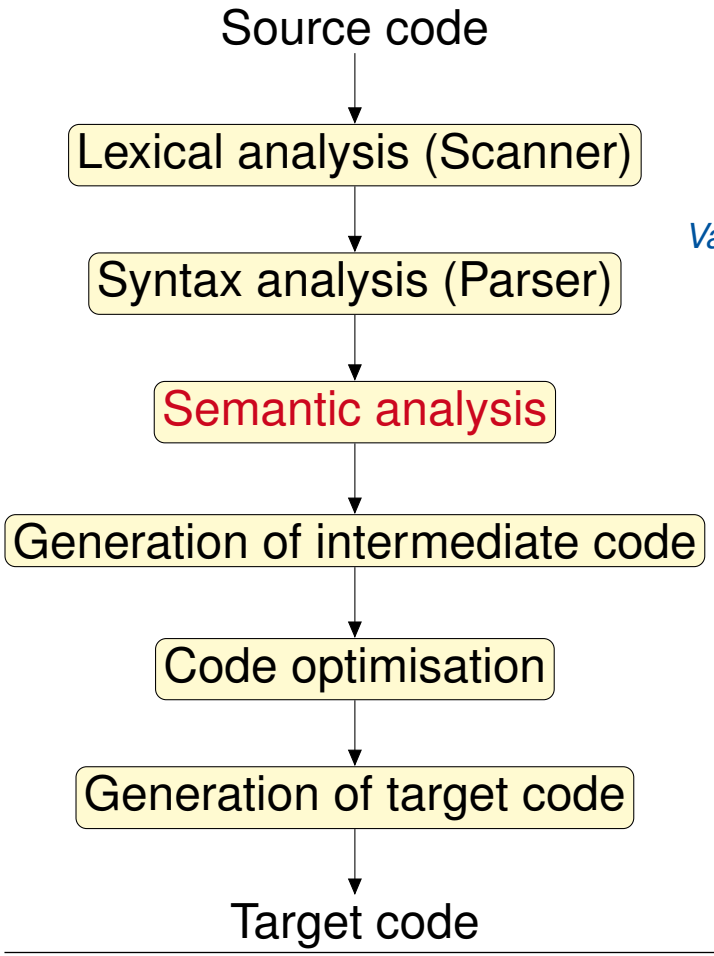
Winter Semester 2018/19

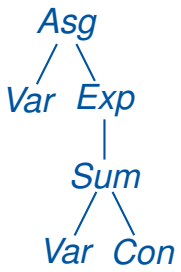
Thomas Noll
Software Modeling and Verification Group
RWTH Aachen University

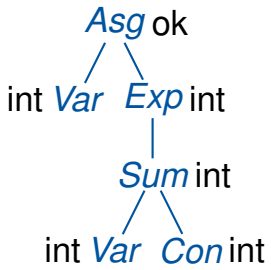
https://moves.rwth-aachen.de/teaching/ws-1819/cc/

Overview

Conceptual Structure of a Compiler







attribute grammars

Beyond Syntax

From Merriam-Webster's Online Dictionary

Semantic: of or relating to meaning in language

To generate (efficient) code, the compiler needs to answer many questions:

- Are there identifiers that are not declared? Declared but not used?
- Is `x` a scalar, an array, or a procedure? Of which type?
- Which declaration of `x` applies to which reference?
- Is `x` defined before it is used?
- Is the expression `3 * x + y` type consistent?
- Where should the value of `x` be stored (register/stack/heap)?
- Do `p` and `q` refer to the same memory location (aliasing)?
- ...

These cannot be expressed using context-free grammars!
(For example, $\{ww \mid w \in \Sigma^+\} \notin CFL_\Sigma$)

Static Semantics

Static semantics

Refers to properties of program constructs

- which are true for every occurrence of this construct in every program execution and
- can be decided at compile time (“static”)
- but are context-sensitive and thus not expressible using context-free grammars (“semantics”).

Example properties

Static: type or declaredness of an identifier, number of registers required to evaluate an expression, ...

Dynamic: value of an expression, size of procedure stack, ...

Attribute Grammars I

Goal: compute context-dependent but runtime-independent properties of a given program

Idea: enrich context-free grammar by semantic rules which annotate syntax tree with attribute values

$\Longrightarrow$ Semantic analysis = attribute evaluation

Result: attributed syntax tree

In greater detail:

- With every grammar symbol a set of attributes is associated.
- Two types of attributes are distinguished:
  Synthesised: bottom-up computation (from the leaves to the root)
  Inherited: top-down computation (from the root to the leaves)
- With every production a set of semantic rules is associated.

Attribute Grammars II

Advantage: attribute grammars provide a very flexible and broadly applicable mechanism for transporting information through the syntax tree (“syntax-directed translation”)

- Attribute values: symbol tables, data types, code, error flags, ...
- Application in Compiler Construction:
  - static semantics
  - program analysis for optimisation
  - code generation
  - error handling
- Automatic attribute evaluation by compiler generators
  (cf. `ANTLR`’s and `yacc`’s synthesised attributes)
- Originally designed by D. Knuth for defining the semantics of context-free languages
  (Math. Syst. Theory 2 (1968), pp. 127–145)

Example: Type Checking I

Example 12.1 (Attribute grammar for type checking)

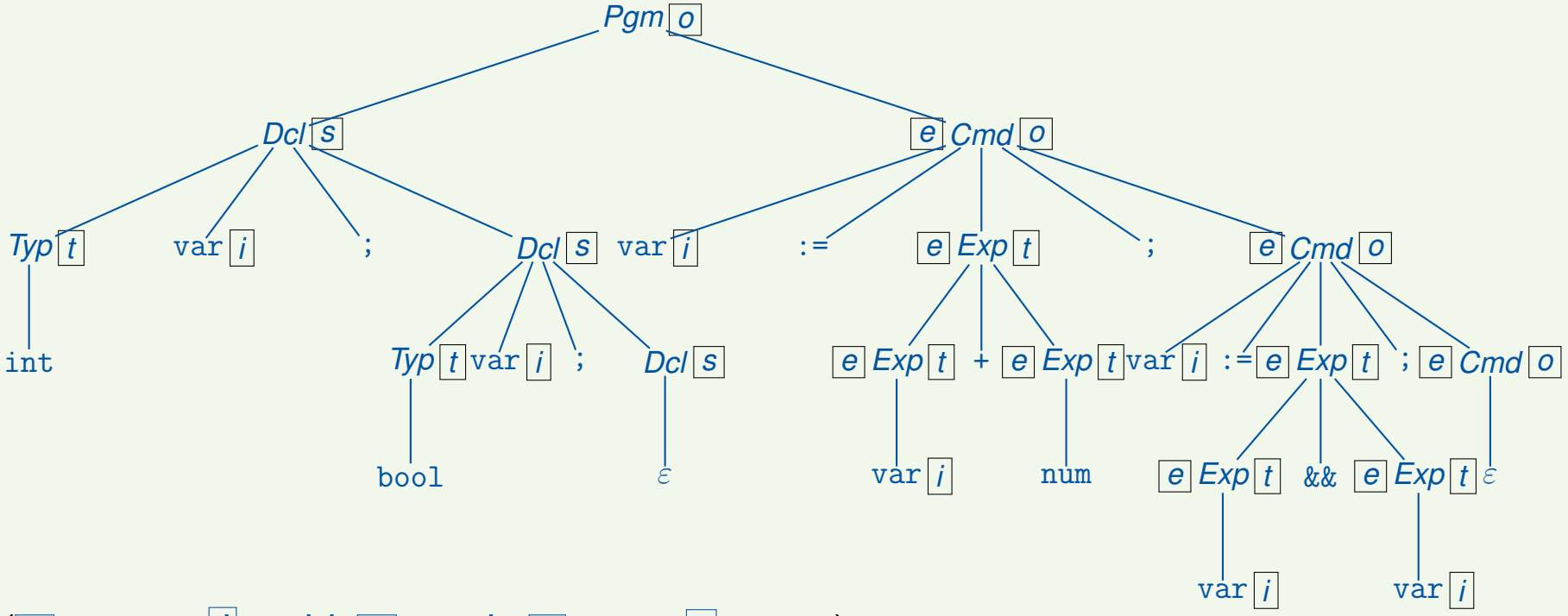
$Pgm \to Dcl\ Cmd$	$ok.0 = ok.2$	$env.2 = st.1$	
$Dcl \to \varepsilon$	$st.0 = [id \mapsto \text{err} \mid id \in Id]$		
$\mid\ Typ\ \texttt{var}\ ;\ Dcl$	$st.0 = st.4[id.2 \mapsto typ.1]$		
$Typ \to \texttt{int}$	$typ.0 = \text{int}$		
$\mid\ \texttt{bool}$	$typ.0 = \text{bool}$		
$Cmd \to \varepsilon$	$ok.0 = \text{true}$		
$\mid\ \texttt{var} := Exp\ ;\ Cmd$	$ok.0 = (env.0(id.1) = typ.3 \wedge ok.5)$	$env.3 = env.0$	$env.5 = env.0$
$Exp \to \texttt{num}$	$typ.0 = \text{int}$		
$\mid\ \texttt{var}$	$typ.0 = env.0(id.1)$		
$\mid\ Exp + Exp$	$typ.0 = (typ.1 = typ.3 = \text{int}\ ?\ \text{int} : \text{err})$	$env.1 = env.0$	$env.3 = env.0$
$\mid\ Exp < Exp$	$typ.0 = (typ.1 = typ.3 = \text{int}\ ?\ \text{bool} : \text{err})$	$env.1 = env.0$	$env.3 = env.0$
$\mid\ Exp\ \&\&\ Exp$	$typ.0 = (typ.1 = typ.3 = \text{bool}\ ?\ \text{bool} : \text{err})$	$env.1 = env.0$	$env.3 = env.0$

- Synthesised attributes: *id* (identifier name), *ok* (Boolean result), *st* (symbol table, mapping identifiers to types), *typ* (data type in $\{\text{bool}, \text{int}, \text{err}\}$)
- Inherited attributes: *env* (environment – same type as symbol table)

Example: Type Checking II

Example 12.2 (Attributed syntax tree)

For program `int x; bool y; x := x+1; y := x && y`:



($e = env$, $i = id$, $o = ok$, $s = st$, $t = typ$)

Formal Definition of Attribute Grammars I

Definition 12.3 (Attribute grammar)

Let $G = \langle N, \Sigma, P, S \rangle \in CFG_\Sigma$ with $X := N \uplus \Sigma$.

- Let $Att = Syn \uplus Inh$ be a set of (synthesised or inherited) attributes, and let $V = \bigcup_{\alpha \in Att} V^\alpha$ be a collection of value sets.
- Let $\mathrm{att} : X \to 2^{Att}$ be an attribute assignment, and let $\mathrm{syn}(Y) := \mathrm{att}(Y) \cap Syn$ and $\mathrm{inh}(Y) := \mathrm{att}(Y) \cap Inh$ for every $Y \in X$.
- Every production $\pi = Y_0 \to Y_1 \dots Y_r \in P$ determines the set
$$Var_\pi := \{\alpha.i \mid \alpha \in \mathrm{att}(Y_i), i \in \{0, \dots, r\}\}$$
of attribute variables of π with the subsets of internal and external variables:
$$Int_\pi := \{\alpha.i \mid (i = 0, \alpha \in \mathrm{syn}(Y_i)) \text{ or } (i \in [r], \alpha \in \mathrm{inh}(Y_i))\} \quad Ext_\pi := Var_\pi \setminus Int_\pi$$
- A semantic rule of π is an equation of the form
$$\alpha_0.i_0 = f(\alpha_1.i_1, \dots, \alpha_n.i_n)$$
where $n \in \mathbb{N}$, $\alpha_0.i_0 \in Int_\pi$, $\alpha_j.i_j \in Ext_\pi$, and $f : V^{\alpha_1} \times \dots \times V^{\alpha_n} \to V^{\alpha_0}$.
- For each $\pi \in P$, let E_π be a set with exactly one semantic rule for every internal variable of π, and let $E := (E_\pi \mid \pi \in P)$.

Then $\mathfrak{A} := \langle G, E, V \rangle$ is called an attribute grammar: $\mathfrak{A} \in AG$.

Formal Definition of Attribute Grammars II

Example 12.4 (cf. Example 12.1)

$\mathfrak{A} \in AG$ for type checking:

- Attributes: $Att = Syn \uplus Inh$ with $Syn = \{id, ok, st, typ\}$ and $Inh = \{env\}$
- Value sets: $V^{id} \subseteq \Omega^+$, $V^{ok} = \mathbb{B}$, $V^{st} = V^{env} = \left(V^{id} \to V^{typ}\right)$, $V^{typ} = \{\text{bool}, \text{int}, \text{err}\}$
- Attribute assignment:

$Y \in X$	Pgm	Dcl	Typ	Cmd	Exp	`var`	$a \in \Sigma \setminus \{\texttt{var}\}$
$\text{syn}(Y)$	$\{ok\}$	$\{st\}$	$\{typ\}$	$\{ok\}$	$\{typ\}$	$\{id\}$	$\emptyset$
$\text{inh}(Y)$	$\emptyset$	$\emptyset$	$\emptyset$	$\{env\}$	$\{env\}$	$\emptyset$	$\emptyset$

- Attribute variables:

$\pi \in P$	$Pgm \to Dcl\ Cmd$	$Cmd \to \texttt{var} := Exp; Cmd$	$Exp \to Exp + Exp$	...
Int_π	$\{ok.0, env.2\}$	$\{ok.0, env.3, env.5\}$	$\{typ.0, env.1, env.3\}$	...
Ext_π	$\{st.1, ok.2\}$	$\{env.0, id.1, typ.3, ok.5\}$	$\{env.0, typ.1, typ.3\}$	...

- Semantic rules: see Example 12.1 (e.g., $E_{Pgm \to Dcl\ Cmd} = \{ok.0 = ok.2, env.2 = st.1\}$)

Attribution of Syntax Trees I

Definition 12.5 (Attribution of syntax trees)

Let $\mathfrak{A} = \langle G, E, V \rangle \in AG$, and let t be a syntax tree of G with the set of nodes K.

- K determines the set of attribute variables of t:

$$Var_t := \{\alpha.k \mid k \in K \text{ labelled with } Y \in X, \alpha \in \mathrm{att}(Y)\}.$$

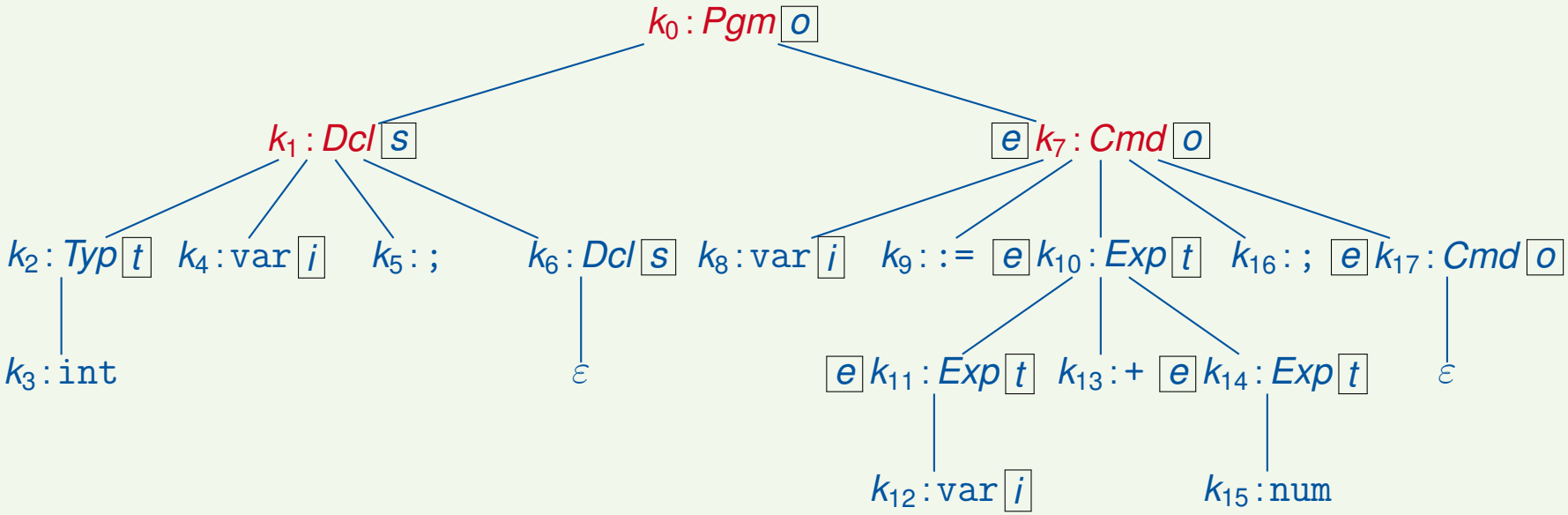
- Let $k_0 \in K$ be an (inner) node where production $\pi = Y_0 \to Y_1 \dots Y_r \in P$ is applied, and let $k_1, \dots, k_r \in K$ be the corresponding successor nodes. The attribute equation system E_{k_0} of k_0 is obtained from E_π by substituting every attribute index $i \in \{0, \dots, r\}$ by k_i.
- The attribute equation system of t is given by

$$E_t := \bigcup \{E_k \mid k \text{ inner node of } t\}.$$

Attribution of Syntax Trees II

Example 12.6 (cf. Example 12.1)

Attributed syntax tree for `int x; x := x+1;`:



$$E_{Pgm \to Dcl\ Cmd}: \begin{array}{l} ok.0 = ok.2 \\ env.2 = st.1 \end{array} \xrightarrow{\text{subst}} E_{k_0}: \begin{array}{l} ok.k_0 = ok.k_7 \\ env.k_7 = st.k_1 \end{array}$$

Attribution of Syntax Trees III

Corollary 12.7

For each $\alpha.k \in Var_t$ except for the inherited attribute variables at the root and the synthesised attribute variables at the leaves of t, E_t contains exactly one equation with left-hand side $\alpha.k$.

Assumptions:

- The start symbol does not have inherited attributes: $\mathrm{inh}(S) = \emptyset$.
- Synthesised attributes of terminal symbols are provided by the scanner.

Solvability of Attribute Equation System I

Definition 12.8 (Solution of attribute equation system)

Let $\mathfrak{A} = \langle G, E, V \rangle \in AG$, and let t be a syntax tree of G. A solution of E_t is a mapping

$$v : Var_t \to V$$

such that, for every $\alpha_0.k_0 \in Var_t$ and $\alpha_0.k_0 = f(\alpha_1.k_1, \ldots, \alpha_n.k_n) \in E_t$,

$$v(\alpha_0.k_0) = f(v(\alpha_1.k_1), \ldots, v(\alpha_n.k_n)).$$

In general, the attribute equation system E_t of a given syntax tree t can have

- no solution,
- exactly one solution, or
- several solutions.

Solvability of Attribute Equation System II

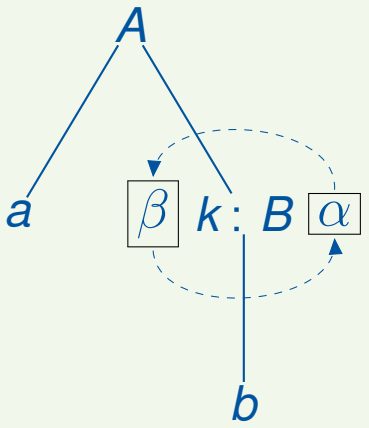
Example 12.9

- $A \to aB, B \to b \in P$
- $\alpha \in \mathrm{syn}(B)$, $\beta \in \mathrm{inh}(B)$
- $\beta.2 = f(\alpha.2) \in E_{A \to aB}$
- $\alpha.0 = \beta.0 \in E_{B \to b}$

$\Longrightarrow$ for $V^\alpha := V^\beta := \mathbb{N}$ and

- $f(x) := x + 1$: no solution
- $f(x) := 2x$: exactly one solution
  $(v(\alpha.k) = v(\beta.k) = 0)$
- $f(x) := x$: infinitely many solutions
  $(v(\alpha.k) = v(\beta.k) = y$ for any $y \in \mathbb{N})$

$\Longrightarrow$ cyclic dependency:



$$E_t: \ \beta.k = f(\alpha.k)$$
$$\alpha.k = \beta.k$$

Circularity of Attribute Grammars

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

Definition 12.10 (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.

Attribute Dependency Graphs I

Goal: graphical representation of attribute dependencies

Definition 12.11 (Production dependency graph)

Let $\mathfrak{A} = \langle G, E, V \rangle \in AG$ with $G = \langle N, \Sigma, P, S \rangle$. Every production $\pi \in P$ determines the dependency graph $D_\pi := \langle Var_\pi, \rightarrow_\pi \rangle$ where the set of edges $\rightarrow_\pi \subseteq Var_\pi \times Var_\pi$ is given by

$$x \rightarrow_\pi y \quad \text{iff} \quad y = f(\ldots, x, \ldots) \in E_\pi.$$

Corollary 12.12

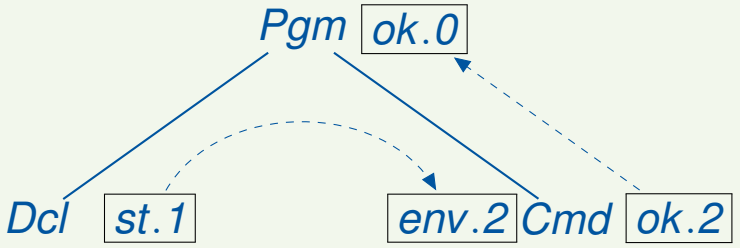
The dependency graph of a production is acyclic (since $\rightarrow_\pi \subseteq Ext_\pi \times Int_\pi$ and $Ext_\pi \cap Int_\pi = \emptyset$).

Attribute Dependency Graphs II

Example 12.13 (cf. Example 12.1)

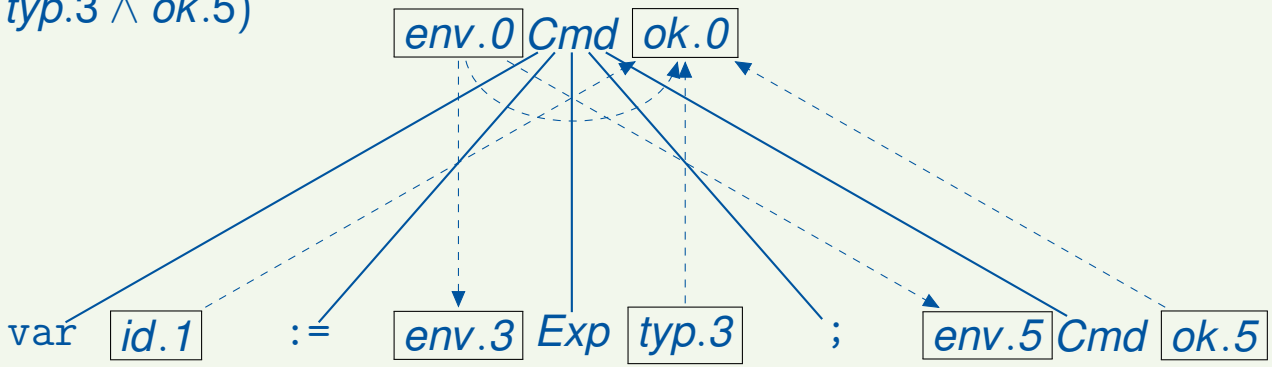
1. $Pgm \to Dcl\ Cmd$: $\Longrightarrow$ $D_{Pgm \to Dcl\ Cmd}$:

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



2. $Cmd \to \texttt{var} := Exp\,;\,Cmd$: $\Longrightarrow$ $D_{Cmd \to \texttt{var} := Exp\,;\,Cmd}$:

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



Attribute Dependency Graphs III

Just as the attribute equation system E_t of a syntax tree t is obtained from the semantic rules of the contributing productions, the dependency graph of t is obtained by "glueing together" the dependency graphs of the productions.

Definition 12.14 (Tree dependency graph)

Let $\mathfrak{A} = \langle G, E, V \rangle \in AG$, and let t be a syntax tree of G.

- The dependency graph of t is defined by $D_t := \langle Var_t, \rightarrow_t \rangle$ where the set of edges, $\rightarrow_t \subseteq Var_t \times Var_t$, is given by

$$x \rightarrow_t y \quad \text{iff} \quad y = f(\ldots, x, \ldots) \in E_t.$$

- D_t is called cyclic if there exists $x \in Var_t$ such that $x \rightarrow_t^+ x$.

Corollary 12.15

An attribute grammar $\mathfrak{A} = \langle G, E, V \rangle \in AG$ is circular iff there exists a syntax tree t of G such that D_t is cyclic.

Attribute Dependency Graphs IV

Example 12.16 (cf. Example 12.1)

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