# Compiler Construction

## Lecture 15: Semantic Analysis III (Attribute Evaluation)

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

RWTH Aachen University

noll@cs.rwth-aachen.de

http://www-i2.informatik.rwth-aachen.de/i2/cc12/

Summer Semester 2012

# Outline

1. Repetition: Circularity of Attribute Grammars
2. The Circularity Check
3. Correctness and Complexity of the Circularity Check
4. Attribute Evaluation
5. Attribute Evaluation by Topological Sorting
6. L-Attributed Grammars

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

# 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 $\pi = A_0 \to w_0 A_1 w_1 \ldots A_r w_r \in P$ 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 $\mathrm{syn}(A_i)$ depend on attributes in $\mathrm{inh}(A_i)$.

## Example

on the board

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

# Attribute Dependency Graphs and Circularity II

## Definition (Attribute dependence)

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

- If $t$ is a syntax tree with root label $A \in N$ and root node $k$, $\alpha \in \mathrm{syn}(A)$, and $\beta \in \mathrm{inh}(A)$ such that $\beta.k \to_t^+ \alpha.k$, then $\alpha$ is dependent on $\beta$ below $A$ in $t$ (notation: $\beta \overset{A}{\hookrightarrow} \alpha$).
- For every syntax tree $t$ with root label $A \in N$,
$$is(A, t) := \{(\beta, \alpha) \in \mathrm{inh}(A) \times \mathrm{syn}(A) \mid \beta \overset{A}{\hookrightarrow} \alpha \text{ in } t\}.$$
- For every $A \in N$,
$$IS(A) := \{is(A, t) \mid t \text{ syntax tree with root label A}\} \subseteq 2^{Inh \times Syn}.$$

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

## Example

on the board

# Outline

# The Circularity Check I

In the circularity check, the dependency systems $IS(A)$ are iteratively computed. The following notation is employed:

**Definition 15.1**

Given $\pi = A \to w_0 A_1 w_1 \ldots A_r w_r \in P$ and $is_i \subseteq \mathrm{inh}(A_i) \times \mathrm{syn}(A_i)$ for every $i \in [r]$, let

$$is[\pi; is_1, \ldots, is_r] \subseteq \mathrm{inh}(A) \times \mathrm{syn}(A)$$

be given by

$$is[\pi; is_1, \ldots, is_r] :=$$
$$\left\{(\beta, \alpha) \mid (\beta.0, \alpha.0) \in \left(\to_\pi \cup \bigcup_{i=1}^{r} \{(\beta'.p_i, \alpha'.p_i) \mid (\beta', \alpha') \in is_i\}\right)^+\right\}$$

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

**Example 15.2**

on the board

# The Circularity Check II

**Algorithm 15.3 (Circularity check for attribute grammars)**

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

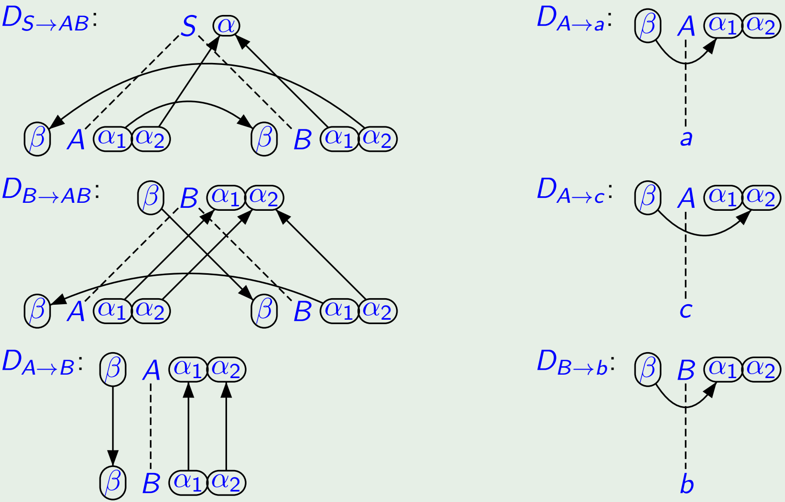
Procedure:

1. *for every* $A \in N$, *iteratively construct* $IS(A)$ *as follows:*
   1. *if* $\pi = A \to w \in P$, *then* $is[\pi] \in IS(A)$
   2. *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"*

# The Circularity Check III

## Example 15.4




Application of Algorithm 15.3: on the board

# Outline

# Correctness and Complexity of Circularity Check

**Theorem 15.5 (Correctness of circularity check)**

*An attribute grammar is circular iff Algorithm 15.3 yields the answer "yes".*

**Proof.**

by induction on the syntax tree $t$ with cyclic $D_t$ □

**Lemma 15.6**

*The time complexity of the circularity check is exponential in the size of the attribute grammar (= maximal length of right-hand sides of productions).*

**Proof.**

by reduction of the word problem of alternating Turing machines (see M. Jazayeri: *A Simpler Construction for Showing the Intrinsically Exponential Complexity of the Circularity Problem for Attribute Grammars*, Comm. of the ACM 28(4), 1981, pp. 715–720) □

# Outline

# Attribute Evaluation Methods

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

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

Methods:
1. Topological sorting of $D_t$ (later):
   1. start with variables which depend at most on $Syn_\Sigma$
   2. proceed by successive substitution
2. Strongly noncircular AGs: recursive functions (details omitted)
   1. for every $A \in N$ and $\alpha \in \mathrm{syn}(A)$, define evaluation function $g_{A,\alpha}$ with the following parameters:
      - the node of $t$ where $\alpha$ has to be evaluated and
      - all inherited attributes of $A$ on which $\alpha$ (potentially) depends
   2. for every $\alpha \in \mathrm{syn}(S)$, evaluate $g_{S,\alpha}(k_0)$ where $k_0$ denotes the root of $t$
3. L-attributed grammars: integration with top-down parsing (later)
4. S-attributed grammars (i.e., $Inh = \emptyset$): `yacc`

# Outline

# Attribute Evaluation by Topological Sorting

## Algorithm 15.7 (Evaluation by topological sorting)

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

Procedure:

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

Output: *solution* $v : Var_t \to V$

**Remark:** noncircularity guarantees that in step 2.1 at least one such $x$ is available

## Example 15.8

see Examples 13.1 and 13.2 (Knuth's binary numbers)

# Outline

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

**Corollary 15.2**

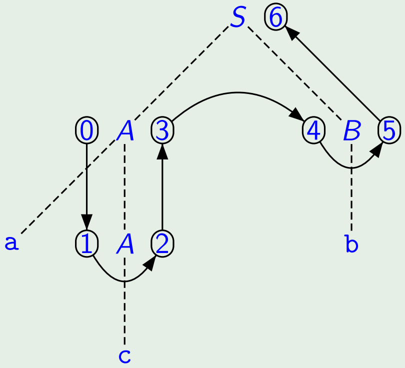
*Every $\mathfrak{A} \in LAG$ is noncircular.*

# L-Attributed Grammars II

## Example 15.3

L-attributed grammar:

$$
\begin{array}{llll}
S \to AB & i.1 & = & 0 \\
 & i.2 & = & s.1 + 1 \\
 & s.0 & = & s.2 + 1 \\
A \to \mathtt{a}A & i.2 & = & i.0 + 1 \\
 & s.0 & = & s.2 + 1 \\
A \to \mathtt{c} & s.0 & = & i.0 + 1 \\
B \to \mathtt{b} & s.0 & = & i.0 + 1
\end{array}
$$



# 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 $\Longrightarrow$

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

# Recursive-Descent Parsing and 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

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

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 15.4 (cf. Example 15.3)

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

## Example 15.5 (cf. Example 15.3)

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