Compiler Construction

Lecture 14: Semantic Analysis III (Attribute Evaluation)

Summer Semester 2016

Thomas Noll
Software Modeling and Verification Group
RWTH Aachen University

https://moves.rwth-aachen.de/teaching/ss-16/cc/

Outline of Lecture 14

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 In_π and Out_π, 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 α is dependent on β 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 $A \in N$, $\quad IS(A) := \{is(A, t) \mid t \text{ syntax tree with root label A}\} \subseteq 2^{Inh \times Syn}$.

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 of Lecture 14

The Circularity Check I

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

Definition 14.1

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 each $i \in [r]$,

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

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

The Circularity Check I

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

Definition 14.1

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 each $i \in [r]$,

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

is defined by

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

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

Example 14.2

on the board

The Circularity Check II

Algorithm 14.3 (Circularity check for attribute grammars)

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

The Circularity Check II

Algorithm 14.3 (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:*

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

The Circularity Check II

Algorithm 14.3 (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:*

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

The Circularity Check II

Algorithm 14.3 (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:*

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:*

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

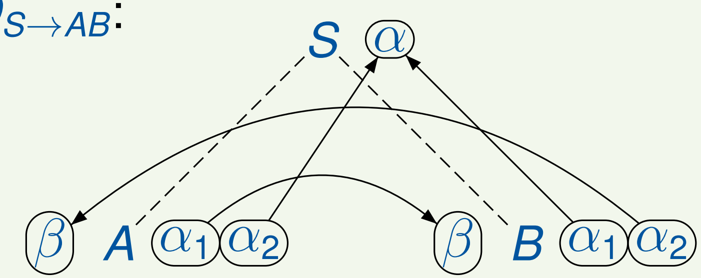
(where $p_i := \sum_{j=1}^{i} |w_{j-1}| + i$*)*

Output: *"yes" or "no"*

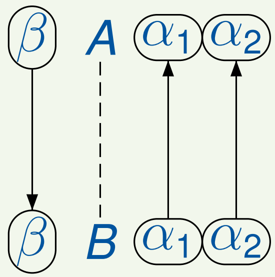
The Circularity Check III

Example 14.4

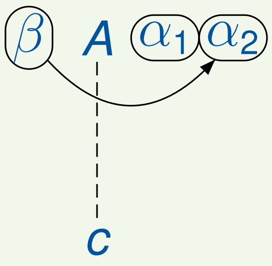
$D_{S \to AB}$:



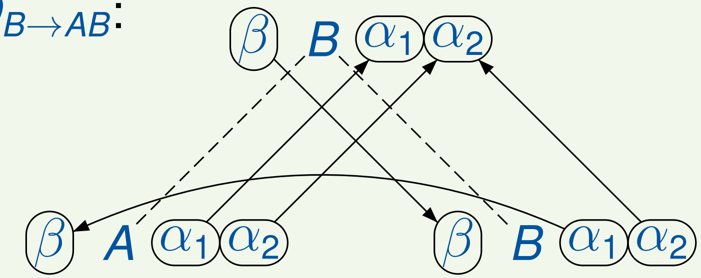
$D_{A \to B}$:



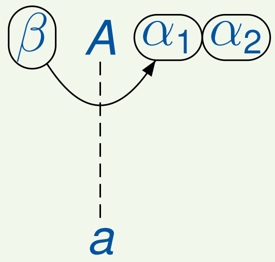
$D_{A \to c}$:



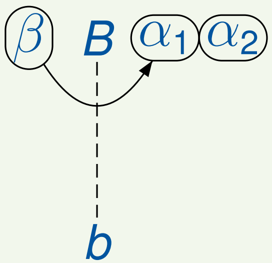
$D_{B \to AB}$:



$D_{A \to a}$:



$D_{B \to b}$:



Application of Algorithm 14.3: on the board

Outline of Lecture 14

Correctness and Complexity of Circularity Check

Theorem 14.5 (Correctness of circularity check)

An attribute grammar is circular iff Algorithm 14.3 yields the answer "yes"

Correctness and Complexity of Circularity Check

Theorem 14.5 (Correctness of circularity check)

An attribute grammar is circular iff Algorithm 14.3 yields the answer "yes"

Proof.

by induction on the syntax tree t with cyclic D_t □

Correctness and Complexity of Circularity Check

Theorem 14.5 (Correctness of circularity check)

An attribute grammar is circular iff Algorithm 14.3 yields the answer "yes"

Proof.

by induction on the syntax tree t with cyclic D_t □

Lemma 14.6

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

Correctness and Complexity of Circularity Check

Theorem 14.5 (Correctness of circularity check)

An attribute grammar is circular iff Algorithm 14.3 yields the answer "yes"

Proof.

by induction on the syntax tree t with cyclic D_t □

Lemma 14.6

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. ACM 28(4), 1981, pp. 715–720) □

Outline of Lecture 14

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 $Syn_\Sigma := \{\alpha.k \mid k \text{ labelled by } a \in \Sigma, \alpha \in \mathrm{syn}(a)\} \subseteq Var_t$

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 $v : Var_t \to V$

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 (later):

  i. start with variables which depend at most on Syn_Σ
  ii. proceed by successive substitution

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 (later):
   i. start with variables which depend at most on Syn_Σ
   ii. proceed by successive substitution
2. Strongly noncircular AGs: recursive functions (details omitted)
   i. 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 α has to be evaluated and
      - all inherited attributes of A on which α (potentially) depends
   ii. for every $\alpha \in \mathrm{syn}(S)$, evaluate $g_{S,\alpha}(k_0)$ where k_0 denotes the root of t

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 (later):
   i. start with variables which depend at most on Syn_Σ
   ii. proceed by successive substitution
2. Strongly noncircular AGs: recursive functions (details omitted)
   i. 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 α has to be evaluated and
      - all inherited attributes of A on which α (potentially) depends
   ii. 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)

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 (later):
   i. start with variables which depend at most on Syn_Σ
   ii. proceed by successive substitution
2. Strongly noncircular AGs: recursive functions (details omitted)
   i. 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 α has to be evaluated and
      - all inherited attributes of A on which α (potentially) depends
   ii. 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 of Lecture 14

Attribute Evaluation by Topological Sorting

Algorithm 14.7 (Evaluation by topological sorting)

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

Attribute Evaluation by Topological Sorting

Algorithm 14.7 (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* $Var := Var \setminus \{x\}$

Attribute Evaluation by Topological Sorting

Algorithm 14.7 (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* $v : Var_t \to V$

Attribute Evaluation by Topological Sorting

Algorithm 14.7 (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

Algorithm 14.7 (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

Example 14.8

see Examples 12.1 and 12.2 (Knuth's binary numbers)

Outline of Lecture 14

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 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 14.1 (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.

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 14.1 (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 14.2

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

L-Attributed Grammars II

Example 14.3

L-attributed grammar:

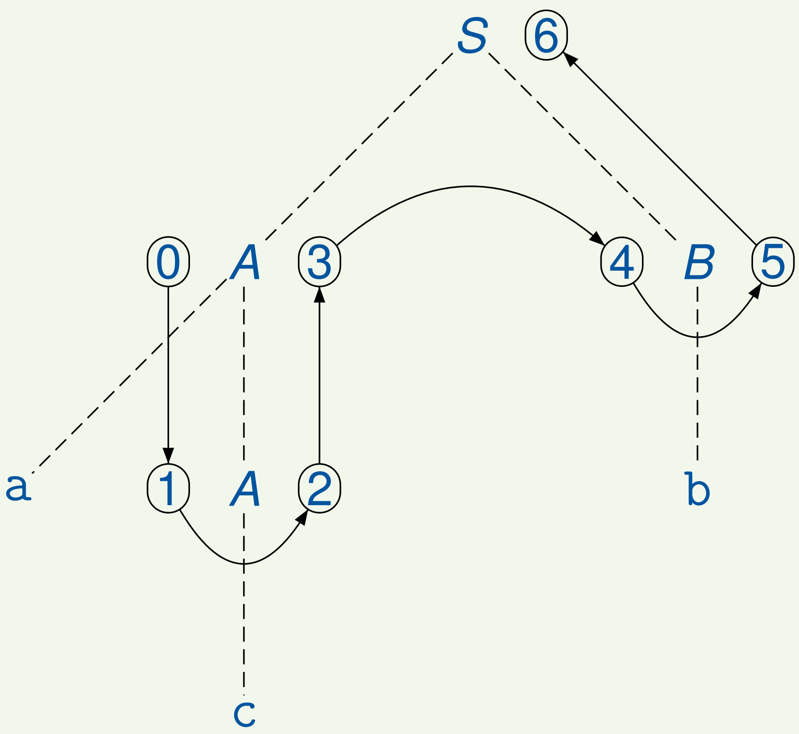
$$
\begin{array}{lll}
S \rightarrow AB & i.1 & = 0 \\
 & i.2 & = s.1 + 1 \\
 & s.0 & = s.2 + 1 \\
A \rightarrow \mathrm{a}A & i.2 & = i.0 + 1 \\
 & s.0 & = s.2 + 1 \\
A \rightarrow \mathrm{c} & s.0 & = i.0 + 1 \\
B \rightarrow \mathrm{b} & s.0 & = i.0 + 1
\end{array}
$$

L-Attributed Grammars II

Example 14.3

L-attributed grammar:

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



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

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
$\Longrightarrow$

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

L-Attributed Grammars

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)

Software Modeling and Verification Chair
RWTH AACHEN UNIVERSITY

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

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

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 14.4 (cf. Example 14.3)

```
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 14.5 (cf. Example 14.3)

```
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
```