Script **generated by TTT**

Title: Simon: Compilerbau (26.05.2014)

Date: Mon May 26 14:18:10 CEST 2014

Duration: 87:37 min

Pages: 55

Topic:

Semantic Analysis

Semantic Analysis

Scanner and parser accept programs with correct syntax.

- not all programs that are syntacticallly correct make *sense*

Semantic Analysis

Scanner and parser accept programs with correct syntax.

- not all programs that are syntacticallly correct make *sense*
- the compiler may be able to *recognize* some of these
  - these programs are rejected and reported as erroneous
  - the language definition defines what erroneous means

Semantic Analysis

Scanner and parser accept programs with correct syntax.

- not all programs that are syntacticallly correct make *sense*
- the compiler may be able to *recognize* some of these
  - these programs are rejected and reported as erroneous
  - the language definition defines what erroneous means
- semantic analyses are necessary that, for instance:
  - check that identifiers are known and where they are defined
  - check the type-correct use of variables

Semantic Analysis

Scanner and parser accept programs with correct syntax.

- not all programs that are syntacticallly correct make *sense*
- the compiler may be able to *recognize* some of these
  - these programs are rejected and reported as erroneous
  - the language definition defines what erroneous means
- semantic analyses are necessary that, for instance:
  - check that identifiers are known and where they are defined
  - check the type-correct use of variables
- semantic analyses are also useful to
  - find possibilities to "optimize" the program
  - warn about possibly incorrect programs

c && (a == b)

if (a == b) {

Semantic Analysis

Scanner and parser accept programs with correct syntax.

- not all programs that are syntacticallly correct make *sense*
- the compiler may be able to *recognize* some of these
  - these programs are rejected and reported as erroneous
  - the language definition defines what erroneous means
- semantic analyses are necessary that, for instance:
  - check that identifiers are known and where they are defined
  - check the type-correct use of variables
- semantic analyses are also useful to
  - find possibilities to "optimize" the program
  - warn about possibly incorrect programs

$\leadsto$ a semantic analysis annotates the syntax tree with attributes

E → E+E

Chapter 1:
Attribute Grammars

Attribute Grammars

- many computations of the semantic analysis as well as the code generation operate on the syntax tree
- what is computed at a given node only depends on the *type* of that node (which is usually a non-terminal)
- we call this a *local* computation:
  - only accesses already computed information from neighbouring nodes
  - computes new information for the current node and other neighbouring nodes

$E \to E + E$

Definition attribute grammar

An attribute grammar is a CFG extended by

- an set of attributes for each non-terminal and terminal
- local attribute equations

Attribute Grammars

- many computations of the semantic analysis as well as the code generation operate on the syntax tree
- what is computed at a given node only depends on the *type* of that node (which is usually a non-terminal)
- we call this a *local* computation:
  - only accesses already computed information from neighbouring nodes
  - computes new information for the current node and other neighbouring nodes

Definition attribute grammar

An attribute grammar is a CFG extended by

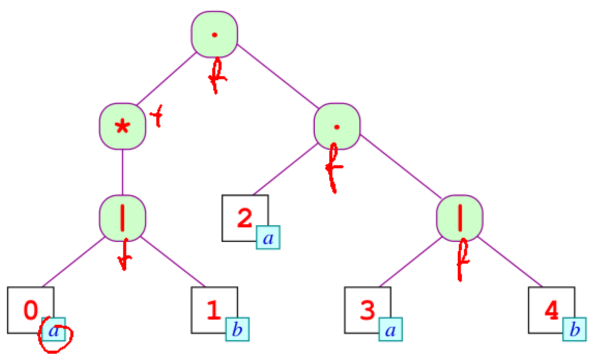
- an set of attributes for each non-terminal and terminal
- local attribute equations

- in order to be able to evaluate the attribute equations, all attributes mentioned in that equation have to be evaluated already

  $\leadsto$ the nodes of the syntax tree need to be visited in a certain *sequence*

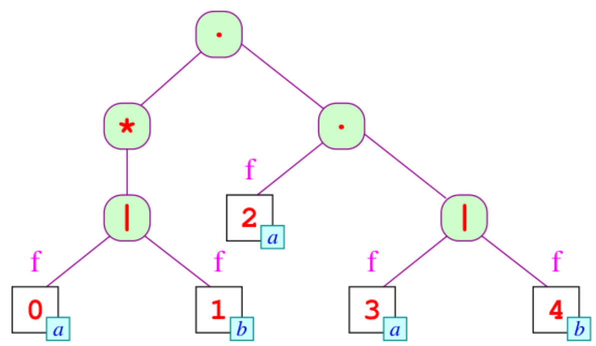
Example: Computation of the empty$[r]$ Property

Consider the syntax tree of the regular expression (a|b)*a(a|b):



Example: Computation of the empty$[r]$ Property

Consider the syntax tree of the regular expression (a|b)*a(a|b):



Implementation Strategy

- attach an attribute empty to every node of the syntax tree
- compute the attributes in a *depth-first* traversal:
  - at a leaf, we can compute the value of empty without considering other nodes
  - the attribute of an inner node only depends on the attribute of its children
- the empty attribute is a *synthetic* attribute
- it may be computed by a ~~pre-~~ or post-order traversal

Implementation Strategy

- attach an attribute empty to every node of the syntax tree
- compute the attributes in a *depth-first* traversal:
  - at a leaf, we can compute the value of empty without considering other nodes
  - the attribute of an inner node only depends on the attribute of its children
- the empty attribute is a *synthetic* attribute
- it may be computed by a pre- or post-order traversal

in general:

Definition

An attribute is called

- synthetic if its value is always propagated upwards in the tree (in the direction leaf $\to$ root)
- inherited if its value is always propagated downwards in the tree (in the direction root $\to$ leaf)

Attribute Equations for empty

In order to compute an attribute *locally*, we need to specify attribute equations for each node.
These equations depend on the *type* of the node:

$r \mapsto x$

for leafs: $r \equiv \boxed{i \mid x}$ we define $\text{empty}[r] = (x \equiv \epsilon)$.

otherwise:

$r \to r \mid r$

$$\begin{aligned}
\text{empty}[r_1 \mid r_2] &= \text{empty}[r_1] \vee \text{empty}[r_2] \\
\text{empty}[r_1 \cdot r_2] &= \text{empty}[r_1] \wedge \text{empty}[r_2] \\
\text{empty}[r_1^*] &= t \\
\text{empty}[r_1?] &= t
\end{aligned}$$

Specification of General Attribute Systems

The empty attribute is *synthetic*, hence, the equations computing it can be given using *structural induction*.

Specification of General Attribute Systems

The empty attribute is *synthetic*, hence, the equations computing it can be given using *structural induction*.

In general, attribute equations combine information for children and parents.

- $\leadsto$ need a more flexible way to specify attribute equations that allows mentioning of parents and children
- use consecutive indices to refer to neighbouring attributes

E ↦ E + E

empty$[0]$:	the attribute of the current node	
empty$[i]$:	the attribute of the i-th child	$(i > 0)$

... in the example:

	$\boxed{x}$	:	empty$[0]$	:=	$(x \equiv \epsilon)$
r → r | r	$\boxed{\mid}$	:	empty$[0]$	:=	empty$[1] \vee$ empty$[2]$
	$\boxed{\cdot}$	:	empty$[0]$	:=	empty$[1] \wedge$ empty$[2]$
r → r*	$\boxed{*}$	:	empty$[0]$	:=	t
	$\boxed{?}$	:	empty$[0]$	:=	t

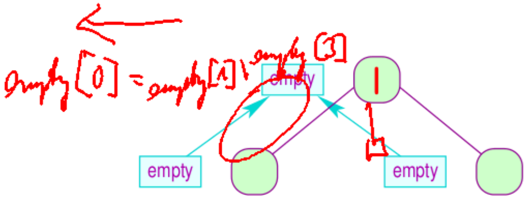
Observations

- the *local* attribute equations need to be evaluated using a *global* algorithm that knows about the dependencies of the equations
- in order to construct this algorithm, we need
  1. a sequence in which the nodes of the tree are visited
  2. a sequence within each node in which the equations are evaluated
- this *evaluation strategy* has to be compatible with the *dependencies* between attributes

Observations

- the *local* attribute equations need to be evaluated using a *global* algorithm that knows about the dependencies of the equations
- in order to construct this algorithm, we need
  1. a sequence in which the nodes of the tree are visited
  2. a sequence within each node in which the equations are evaluated
- this *evaluation strategy* has to be compatible with the *dependencies* between attributes

We illustrate dependencies between attributes using directed graph edges:



$\leadsto$ arrow points in the direction of information flow

Observations

- in order to infer an evaluation strategy, it is not enough to consider the *local* attribute dependencies at each node
- the evaluation strategy must also depend on the *global* dependencies, that is, on the information flow between nodes
- the global dependencies thus change with each new abstract syntax tree
- in the example, the information flows always from the children to the parent node
  $\leadsto$ a depth-first post-order traversal is possible
- in general, variable dependencies can be much more complicated

Simultaneous Computation of Multiple Attributes

Compute empty, first, next of regular expression:

1 $\boxed{S \to E:}$:

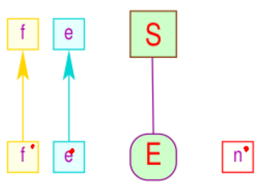
$$\begin{array}{lll} \text{empty}[0] & := & \text{empty}[1] \\ \text{first}[0] & := & \text{first}[1] \\ \text{next}[1] & := & \emptyset \end{array}$$

2 $\boxed{E \to x}$:

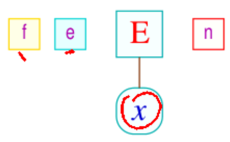
$$\begin{array}{lll} \text{empty}[0] & := & (x \equiv \epsilon) \\ \text{first}[0] & := & \{x \mid x \neq \epsilon\} \end{array}$$

// (no equation for next)

D_1 :



D_2 :

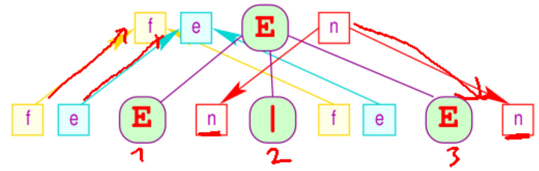


Regular Expressions: Rules for Alternative

3 $\boxed{E \to E|E}$:

$$\begin{array}{lll} \text{empty}[0] & := & \text{empty}[1] \vee \text{empty}[2] \\ \text{first}[0] & := & \text{first}[1] \cup \text{first}[2] \\ \text{next}[1] & := & \text{next}[0] \\ \text{next}[2] & := & \text{next}[0] \end{array}$$

D_3 :

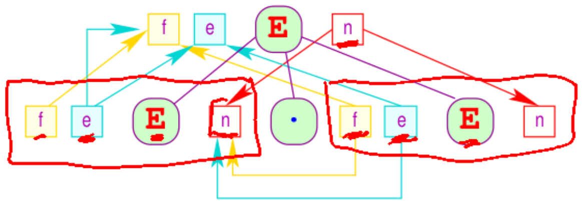


Regular Expressions: Rules for Concatenation

$\boxed{E \to E \cdot E}$:

$$\begin{array}{lll} \text{empty}[0] & := & \text{empty}[1] \wedge \text{empty}[2] \\ \text{first}[0] & := & \text{first}[1] \cup (\text{empty}[1]\ ?\ \text{first}[2] : \emptyset) \\ \text{next}[1] & := & \text{first}[2] \cup (\text{empty}[2]\ ?\ \text{next}[0] : \emptyset) \\ \text{next}[2] & := & \text{next}[0] \end{array}$$

D_4 :



Regular Expressions: Kleene-Star and '?'

$\boxed{E \to E*}$:

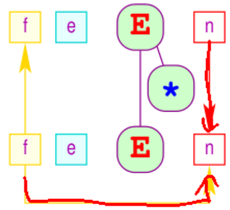
$$\begin{array}{lll} \text{empty}[0] & := & t \\ \text{first}[0] & := & \text{first}[1] \\ \text{next}[1] & := & \text{first}[1] \cup \text{next}[0] \end{array}$$

first[1] ∪ (empty[1] ? next[0] : ∅)

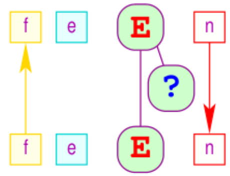
$\boxed{E \to E?}$:

$$\begin{array}{lll} \text{empty}[0] & := & t \\ \text{first}[0] & := & \text{first}[1] \\ \text{next}[1] & := & \text{next}[0] \end{array}$$

D_5 :



D_6 :



Challenges for General Attribute Systems

- assume that the grammar Gr has no useless productions
- let $\mathcal{T}$ denote all derivable syntax trees of Gr
- an evaluation strategy can only exist if for *any* abstract syntax tree $t \in \mathcal{T}$, the dependencies between attributes are acyclic

Challenges for General Attribute Systems

- assume that the grammar Gr has no useless productions
- let $\mathcal{T}$ denote all derivable syntax trees of Gr
- an evaluation strategy can only exist if for *any* abstract syntax tree $t \in \mathcal{T}$, the dependencies between attributes are acyclic

Consider the 6 productions of the regular expression grammar:

D_1

1	$S \to E$	4	$E \to E \cdot E$
2	$E \to x$	5	$E \to E*$
3	$E \to E \mid E$	6	$E \to E?$

Challenges for General Attribute Systems

- assume that the grammar Gr has no useless productions
- let $\mathcal{T}$ denote all derivable syntax trees of Gr
- an evaluation strategy can only exist if for *any* abstract syntax tree $t \in \mathcal{T}$, the dependencies between attributes are acyclic

Consider the 6 productions of the regular expression grammar:

D_i

1	$S \to E$	4	$E \to E \cdot E$
2	$E \to x$	5	$E \to E*$
3	$E \to E \mid E$	6	$E \to E?$

Idea: Compute a directed graph D'_i for each production i.

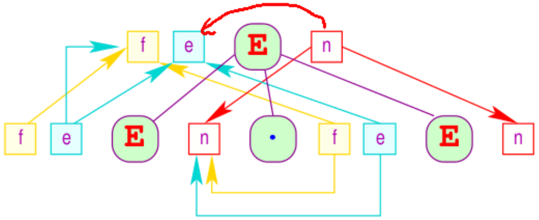
- the vertices of D'_i are its lhs attributes $a_1[0], \ldots a_n[0]$
- an edge $a_i[0] \to a_j[0]$ indicates $a_i[0]$ must be evaluated so that visiting the production can compute $a_j[0]$
- for productions whose rhs only contains terminals $D'_i = D_i$
- compute new edges for other productions based on the current edges of its rhs non-terminals (⇝ next slide)
- when no new edges can be added, the graphs D'_i denote the dependencies of all possible derivation trees
- no. of edges in each graph is finite ⇝ termination guaranteed

Computing Dependencies i: $N \to N_1 \ldots N_n \ldots$

Define $D'_i[G_1, \ldots G_n]$ to be the graph obtained from D'_i by adding an edge from $a_i[0]$ to $a_j[0]$ if there is a path from $a_i[0]$ to $a_j[0]$ in the dependency graph D_i where graphs G_i give the dependencies for the corresponding rhs-attributes. Abort when cycles exists.

Computing Dependencies

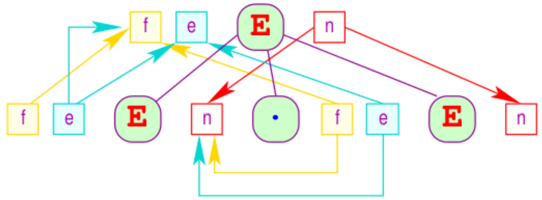
Define $D'_i[G_1, \dots G_n]$ to be the graph obtained from D'_i by adding an edge from $a_i[0]$ to $a_j[0]$ if there is a path from $a_i[0]$ to $a_j[0]$ in the dependency graph D_i where graphs G_i give the dependencies for the corresponding rhs-attributes. Abort when cycles exists.
Example: $D'_4[G_1, G_2, G_3]$: Dependency graph of D_4 :



Suppose graph G_2 is empty and graphs G_1 and G_3 are as follows:

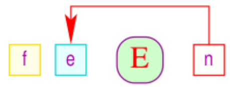
Computing Dependencies

Define $D'_i[G_1, \dots G_n]$ to be the graph obtained from D'_i by adding an edge from $a_i[0]$ to $a_j[0]$ if there is a path from $a_i[0]$ to $a_j[0]$ in the dependency graph D_i where graphs G_i give the dependencies for the corresponding rhs-attributes. Abort when cycles exists.
Example: $D'_4[G_1, G_2, G_3]$: Dependency graph of D_4 :



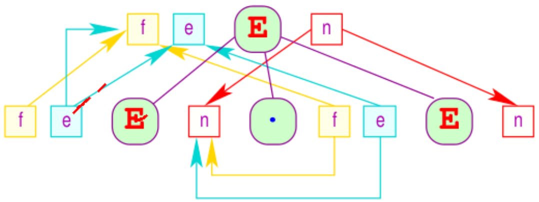
Suppose graph G_2 is empty and graphs G_1 and G_3 are as follows:

Edges added to D'_4:



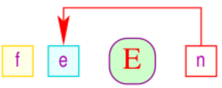
Computing Dependencies

Define $D'_i[G_1, \dots G_n]$ to be the graph obtained from D'_i by adding an edge from $a_i[0]$ to $a_j[0]$ if there is a path from $a_i[0]$ to $a_j[0]$ in the dependency graph D_i where graphs G_i give the dependencies for the corresponding rhs-attributes. Abort when cycles exists.
Example: $D'_4[G_1, G_2, G_3]$: Dependency graph of D_4 :



Suppose graph G_2 is empty and graphs G_1 and G_3 are as follows:

Edges added to D'_4:



Which edges are added for $D'_4[G_3, G_2, G_1]$?

Complexity of Computing Dependencies

p	rule	D'_p			p	rule	D'_p		
1	$S \to E$	$f[0]$	$e[0]$		4	$E \to E \cdot E$	$f[0]$	$e[0]$	$n[0]$
2	$E \to x$	$f[0]$	$e[0]$	$n[0]$	5	$E \to E*$	$f[0]$	$e[0]$	$n[0]$
3	$E \to E \mid E$	$f[0]$	$e[0]$	$n[0]$	6	$E \to E?$	$f[0]$	$e[0]$	$n[0]$

Add edges by repeatedly evaluating until stable:

$D'_1[G_1]$	for $G_1 \in \{D'_2, \dots D'_6\}$
$D'_3[G_1, G_2, G_3]$	for $G_1, G_3 \in \{D'_2, \dots D'_6\}, G_2$ empty
$D'_4[G_1, G_2, G_3]$	for $G_1, G_3 \in \{D'_2, \dots D'_6\}, G_2$ empty
$D'_5[G_1, G_2]$	for $G_1 \in \{D'_2, \dots D'_6\}, G_2$ empty
$D'_6[G_1, G_2]$	for $G_1 \in \{D'_2, \dots D'_6\}, G_2$ empty

Complexity of Computing Dependencies

p	rule	D'_p			p	rule	D'_p		
1	$S \to E$	$f[0]$	$e[0]$		4	$E \to E{\cdot}E$	$f[0]$	$e[0]$	$n[0]$
2	$E \to x$	$f[0]$	$e[0]$	$n[0]$	5	$E \to E*$	$f[0]$	$e[0]$	$n[0]$
3	$E \to E \mid E$	$f[0]$	$e[0]$	$n[0]$	6	$E \to E?$	$f[0]$	$e[0]$	$n[0]$

Add edges by repeatedly evaluating until stable:

$D'_1[G_1]$	for $G_1 \in \{D'_2, \dots D'_6\}$
$D'_3[G_1, G_2, G_3]$	for $G_1, G_3 \in \{D'_2, \dots D'_6\}$, G_2 empty
$D'_4[G_1, G_2, G_3]$	for $G_1, G_3 \in \{D'_2, \dots D'_6\}$, G_2 empty
$D'_5[G_1, G_2]$	for $G_1 \in \{D'_2, \dots D'_6\}$, G_2 empty
$D'_6[G_1, G_2]$	for $G_1 \in \{D'_2, \dots D'_6\}$, G_2 empty

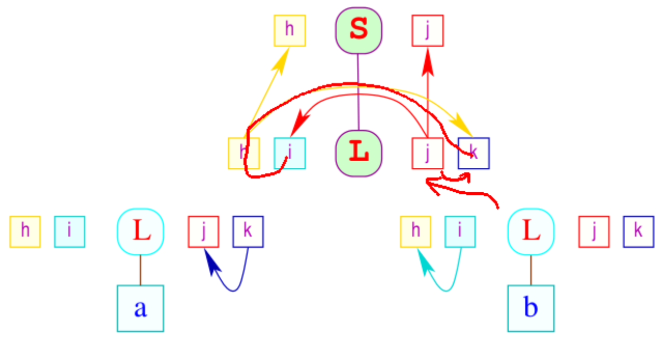
- for n attributes, there are n^2 possible edges
- worst case: only one edge is added in each evaluation of D'_i
- checking that no cyclic attribute dependencies can arise is DEXPTIME-complete [Jazayeri, Odgen, Rounds, 1975] 2^n

Example: Checking Circularity

Grammar:

p	rule	D'_p			
1	$S \to L$	$h[0]$	$j[0]$		
2	$L \to a$	$h[0]$	$i[0]$	$j[0]$	$k[0]$
3	$L \to b$	$h[0]$	$i[0]$	$j[0]$	$k[0]$

Dependency graphs D_p:



Apply until stable:

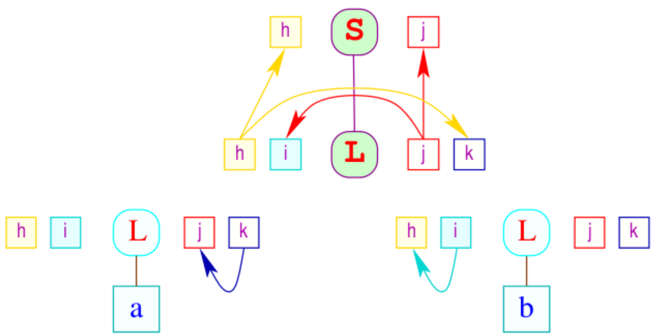
$D'_1[G_1]$ for $G_1 \in \{D'_2, D'_3\}$

Example: Checking Circularity

Grammar:

p	rule	D'_p			
1	$S \to L$	$h[0]$	$j[0]$		
2	$L \to a$	$h[0]$	$i[0]$	$j[0] \leftarrow$	$k[0]$
3	$L \to b$	$h[0] \leftarrow$	$i[0]$	$j[0]$	$k[0]$

Dependency graphs D_p:



Apply until stable:

$D'_1[G_1]$ for $G_1 \in \{D'_2, D'_3\}$

Strongly Acyclic Attribute Dependencies

Problem: with larger grammars, this algorithm is too expensive
Goal: find a *sufficient* condition for an attribute system to be acyclic

Strongly Acyclic Attribute Dependencies

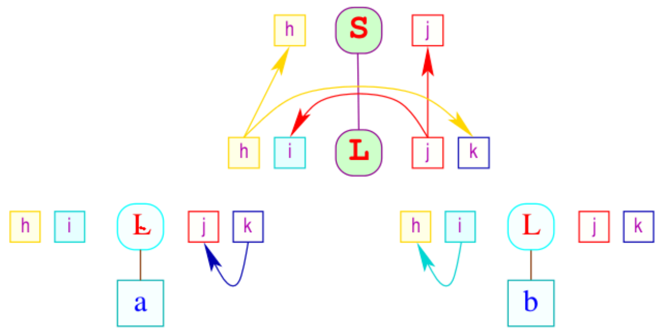
Problem: with larger grammars, this algorithm is too expensive
Goal: find a *sufficient* condition for an attribute system to be acyclic

Idea: Compute dependency graph D'_s for each *non-terminal* $s \in N$.

- for all productions $N \to t_1 \ldots t_n$ with t_i terminals:
  $D'_N = D_{i_1} \cup \ldots \cup D_{i_k}$ where $i_1, \ldots i_k$ are the productions indices
- compute $D'_N[G_1, \ldots G_n]$ for each production $N \to s_1 \ldots s_n$ where G_i is the graph between $a_{i_1}[0], \ldots a_{i_k}[0]$ of s_i
- re-evaluate each rule until none of the graphs change anymore
- if a cycle is detected during the computation of D'_N, report "may have cycle"

Example: Strong Acyclic Test

Consider again the grammar $S \to L$, $L \to a \mid b$. Dependency graphs D_p:



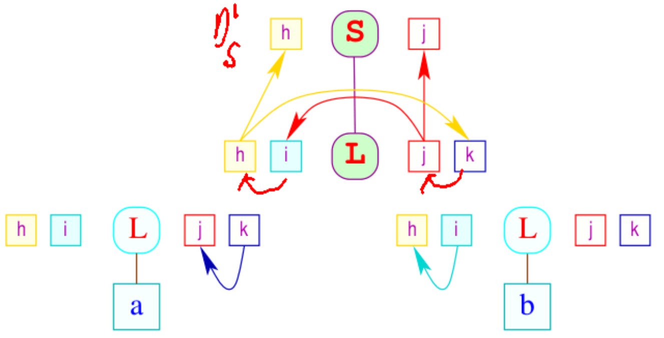
- for all productions $N \to t_1 \ldots t_n$ with t_i terminals:
  $D'_N = D_{i_1} \cup \ldots \cup D_{i_k}$ where $i_1, \ldots i_k$ are the productions indices
- compute $D'_N[G_1, \ldots G_n]$ for each production $N \to s_1 \ldots s_n$ where G_i is the graph between $a_{i_1}[0], \ldots a_{i_k}[0]$ of s_i

p	rule
1	$S \to L$
2	$L \to a$
3	$L \to b$

N	D'_N			
S	$h[0]$		$j[0]$	
L	$h[0] \leftarrow$	$i[0]$	$j[0] \leftarrow$	$k[0]$

Example: Strong Acyclic Test

Consider again the grammar $S \to L$, $L \to a \mid b$. Dependency graphs D_p:



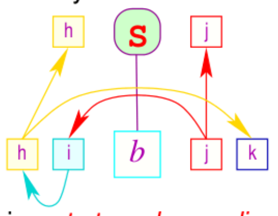
- for all productions $N \to t_1 \ldots t_n$ with t_i terminals:
  $D'_N = D_{i_1} \cup \ldots \cup D_{i_k}$ where $i_1, \ldots i_k$ are the productions indices
- compute $D'_N[G_1, \ldots G_n]$ for each production $N \to s_1 \ldots s_n$ where G_i is the graph between $a_{i_1}[0], \ldots a_{i_k}[0]$ of s_i

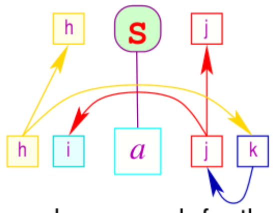
p	rule
1	$S \to L$
2	$L \to a$
3	$L \to b$

N	D'_N			
S	$h[0]$		$j[0]$	
L	$h[0] \leftarrow$	$i[0]$	$j[0] \leftarrow$	$k[0]$

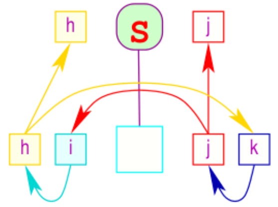
Strong Acyclic and Acyclic

The grammar $S \to L$, $L \to a \mid b$ has only two derivation trees which are both acyclic:





It is *not strongly acyclic* since the dependence graph for the non-terminal L has a cycle when computing D'_S:



From Dependencies to Evaluation Strategies

Possible strategies:

From Dependencies to Evaluation Strategies

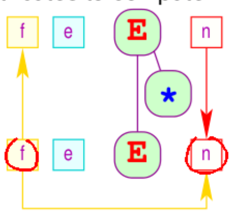
Possible strategies:

1. let the user define the evaluation order

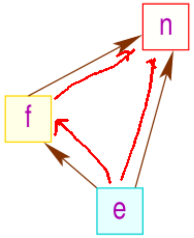
From Dependencies to Evaluation Strategies

Possible strategies:

1. let the user define the evaluation order
2. automatic strategy based on the dependencies:
   - use local dependencies to determine which attributes to compute
     - suppose we require $s[1]$
     - computing $s[1]$ requires $f[1]$
     - $f[1]$ depends on an attribute in the child, so descend



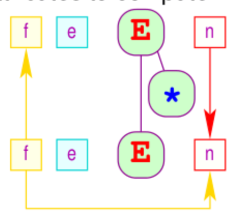
   - compute attributes in passes
     - compute a dependency graph between attributes (no go if cyclic)
     - traverse AST once for each attribute; here three times, once for e, f, g
     - compute one attribute in each pass



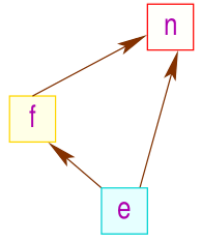
From Dependencies to Evaluation Strategies

Possible strategies:

1. let the user define the evaluation order
2. automatic strategy based on the dependencies:
   - use local dependencies to determine which attributes to compute
     - suppose we require $s[1]$
     - computing $s[1]$ requires $f[1]$
     - $f[1]$ depends on an attribute in the child, so descend



   - compute attributes in passes
     - compute a dependency graph between attributes (no go if cyclic)
     - traverse AST once for each attribute; here three times, once for e, f, g
     - compute one attribute in each pass



3. consider a fixed strategy and only allow an attribute system that can be evaluated using this strategy

Linear Order from Dependency Partial Order

Possible *automatic* strategies:

Linear Order from Dependency Partial Order

Possible *automatic* strategies:

1. demand-driven evaluation
   - start with the evaluation of any required attribute
   - if the equation for this attribute relies on as-of-yet unevaluated attributes, compute these recursively
   - $\rightsquigarrow$ visits the nodes of the syntax tree on demand
   - (following a dependency on the parent requires a pointer to the parent)
2. evaluation in passes
   - *minimize* the number of *visits* to each node
   - organize the evaluation of the tree in passes
   - for each pass, pre-compute a strategy to visit the nodes together with a local strategy for evaluation within each node type

Linear Order from Dependency Partial Order

Possible *automatic* strategies:

1. demand-driven evaluation
   - start with the evaluation of any required attribute
   - if the equation for this attribute relies on as-of-yet unevaluated attributes, compute these recursively
   - $\rightsquigarrow$ visits the nodes of the syntax tree on demand
   - (following a dependency on the parent requires a pointer to the parent)
2. evaluation in passes
   - *minimize* the number of *visits* to each node
   - organize the evaluation of the tree in passes
   - for each pass, pre-compute a strategy to visit the nodes together with a local strategy for evaluation within each node type

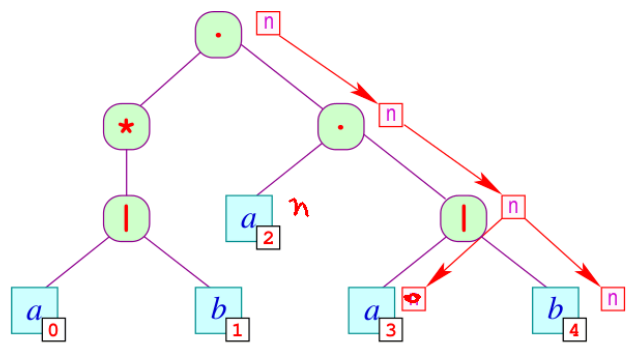
consider example for *demand-driven* evaluation

Example for Demand-Driven Evaluation

Compute next at the leaves of $a(a|b)$ in the expression $((a|b)^*a(a|b))$:

$\boxed{|}$: $\text{next}[1] := \text{next}[0]$
$\text{next}[2] := \text{next}[0]$

$\boxed{\cdot}$: $\text{next}[1] := \text{first}[2] \cup (\text{empty}[2]\,?\,\text{next}[0] : \emptyset)$
$\text{next}[2] := \text{next}[0]$

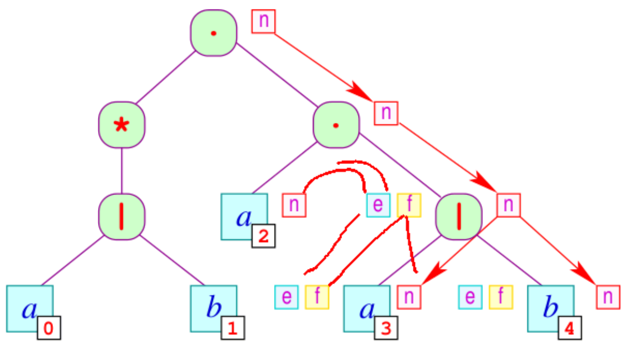


Example for Demand-Driven Evaluation

Compute next at the leaves of $a(a|b)$ in the expression $((a|b)^*a(a|b))$:

$$\boxed{|} \quad : \quad \begin{array}{lcl} \text{next}[1] & := & \text{next}[0] \\ \text{next}[2] & := & \text{next}[0] \end{array}$$

$$\boxed{\cdot} \quad : \quad \begin{array}{lcl} \text{next}[1] & := & \text{first}[2] \cup (\text{empty}[2]\ ?\ \text{next}[0] : \emptyset) \\ \text{next}[2] & := & \text{next}[0] \end{array}$$

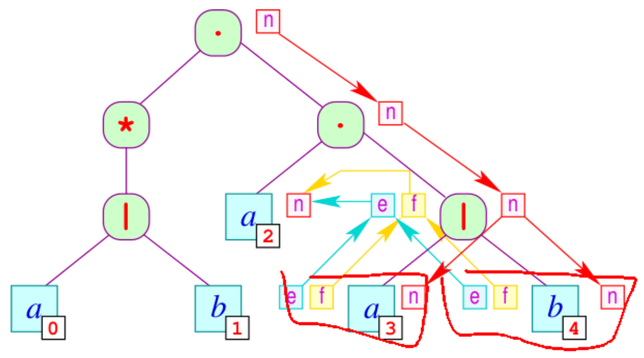


Example for Demand-Driven Evaluation

Compute next at the leaves of $a(a|b)$ in the expression $((a|b)^*a(a|b))$:

$$\boxed{|} \quad : \quad \begin{array}{lcl} \text{next}[1] & := & \text{next}[0] \\ \text{next}[2] & := & \text{next}[0] \end{array}$$

$$\boxed{\cdot} \quad : \quad \begin{array}{lcl} \text{next}[1] & := & \text{first}[2] \cup (\text{empty}[2]\ ?\ \text{next}[0] : \emptyset) \\ \text{next}[2] & := & \text{next}[0] \end{array}$$



Demand-Driven Evaluation

Observations

- *only required* attributes are evaluated
- the evaluation sequence depends – in general – on the actual syntax tree
- the algorithm must track which attributes it has already evaluated
- the algorithm may visit nodes more often than necessary
- each node must contain a pointer to its parent
- the algorithm is not local

Demand-Driven Evaluation

Observations

- *only required* attributes are evaluated
- the evaluation sequence depends – in general – on the actual syntax tree
- the algorithm must track which attributes it has already evaluated
- the algorithm may visit nodes more often than necessary
- each node must contain a pointer to its parent
- the algorithm is not local

approach only beneficial in principle:

- evaluation strategy is dynamic: difficult to debug
- computation of all attributes is often cheaper
- usually all attributes in all nodes are required

Demand-Driven Evaluation

Observations

- *only required* attributes are evaluated
- the evaluation sequence depends – in general – on the actual syntax tree
- the algorithm must track which attributes it has already evaluated
- the algorithm may visit nodes more often than necessary
- each node must contain a pointer to its parent
- the algorithm is not local

approach only beneficial in principle:

- evaluation strategy is dynamic: difficult to debug
- computation of all attributes is often cheaper
- usually all attributes in all nodes are required

$\leadsto$ perform evaluation in *passes*

Evaluation in Passes

Idea: traverse the syntax tree several times; each time, evaluate all those equations $a[i_a] = f(b[i_b], \dots z[i_z])$ whose arguments $b[i_b], \dots z[i_z]$ are known

For a *strongly acyclic attribute system:*

- the local dependencies in D_i of the ith production $N \to s_1 \dots s_n$ together the dependencies D_i' for each s_i define a sequence in which attributes can be evaluated
- determine a sequence in which the children are visited so that as many attributes as possible are evaluated
- in each pass at least one new attribute is evaluated
- requires at most n passes for evaluating n attributes
- since a traversal strategy exists for evaluating one attribute, it might be possible to find a strategy to evaluate more attributes $\leadsto$ optimization problem
- note: evaluating attribute set $\{a_{i_1}[0] \dots a_{i_m}[0]\}$ for rule $N \to \dots N \dots$ may evaluate a different attribute set of its children $\leadsto$ up to $2^k - 1$ evaluation functions for N

Evaluation in Passes

Idea: traverse the syntax tree several times; each time, evaluate all those equations $a[i_a] = f(b[i_b], \dots z[i_z])$ whose arguments $b[i_b], \dots z[i_z]$ are known

For a *strongly acyclic attribute system:*

- the local dependencies in D_i of the ith production $N \to s_1 \dots s_n$ together the dependencies D_i' for each s_i define a sequence in which attributes can be evaluated
- determine a sequence in which the children are visited so that as many attributes as possible are evaluated
- in each pass at least one new attribute is evaluated
- requires at most n passes for evaluating n attributes
- since a traversal strategy exists for evaluating one attribute, it might be possible to find a strategy to evaluate more attributes $\leadsto$ optimization problem
- note: evaluating attribute set $\{a_{i_1}[0] \dots a_{i_m}[0]\}$ for rule $N \to \dots N \dots$ may evaluate a different attribute set of its children $\leadsto$ up to $2^k - 1$ evaluation functions for N

. . . in the example:

- empty and first can be computed together
- next must be computed in a separate pass

Implementing State

Problem: In many cases some sort of state is required.
Example: numbering the leafs of a syntax tree