Script **generated by TTT**

Title: Petter: Compilerbau (13.06.2016)

Date: Mon Jun 13 14:24:25 CEST 2016

Duration: 79:01 min

Pages: 48

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

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

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

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

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

- a 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

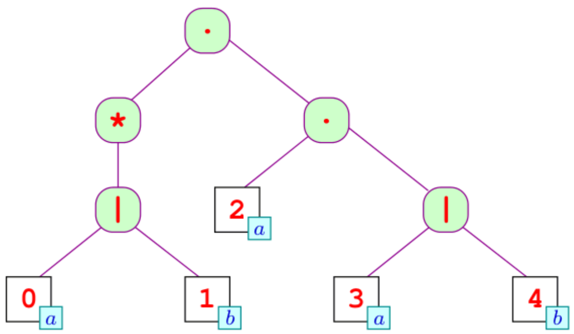
An attribute grammar is a CFG extended by

- a 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*

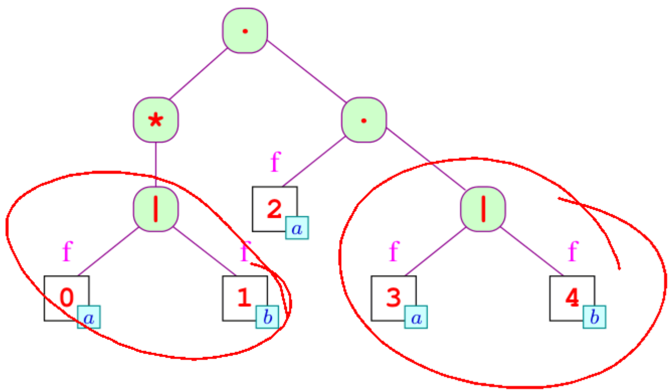
Example: Computation of the empty$[r]$ Attribute

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



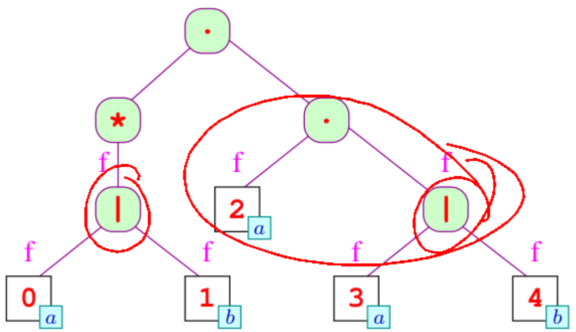
Example: Computation of the empty$[r]$ Attribute

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



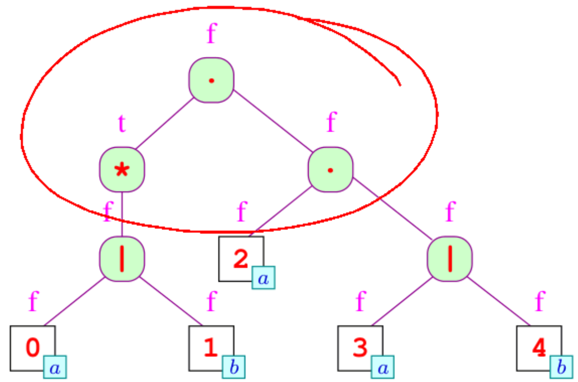
Example: Computation of the empty$[r]$ Attribute

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



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

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



$\rightsquigarrow$ equations for empty$[r]$ are computed from bottom to top (aka bottom-up)

Implementation Strategy

- attach an attribute empty to every node of the syntax tree
- compute the attributes in a *depth-first* post-order 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
- The local dependencies between the attributes are dependent on the *type* of the node

Implementation Strategy

- attach an attribute empty to every node of the syntax tree
- compute the attributes in a *depth-first* post-order 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
- The local dependencies between the attributes are dependent on the *type* of the node

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:

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

otherwise:

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

Specification of General Attribute Systems

General Attribute Systems

In general, for establishing attribute systems we need a flexible way to *refer to parents and children*:

$\rightsquigarrow$ We use consecutive indices to refer to neighbouring attributes

$\text{attribute}_k[0]$:	the attribute of the current root node
$\text{attribute}_k[i]$:	the attribute of the i-th child $\quad (i > 0)$

Specification of General Attribute Systems

General Attribute Systems

In general, for establishing attribute systems we need a flexible way to *refer to parents and children*:

$\rightsquigarrow$ We use consecutive indices to refer to neighbouring attributes

$\text{attribute}_k[0]$:	the attribute of the current root node
$\text{attribute}_k[i]$:	the attribute of the i-th child $\quad (i > 0)$

... in the example:

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

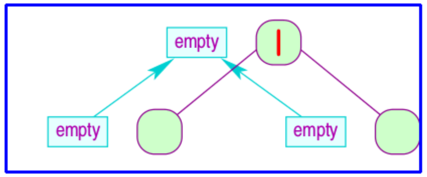
Observations

- 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 visualize the attribute dependencies $D(p)$ of a production p in a *Local Dependency Graph*:



Let $p = N_0 \mapsto N_1 | N_2$ in

$$D(p) = \{ \; (empty[1], empty[0]), \; (empty[2], empty[0]) \}$$

⇝ arrows point in the direction of information flow

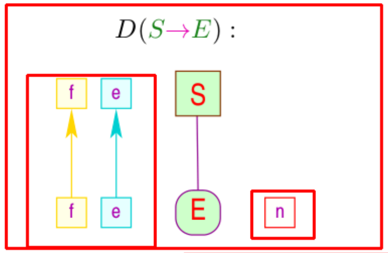
Simultaneous Computation of Multiple Attributes

Computing empty, first, next from regular expressions:

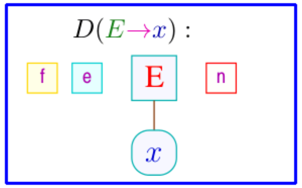
$S \to E$:
- empty[0] := empty[1]
- first[0] := first[1]
- next[1] := $\emptyset$

$E \to x$:
- empty[0] := $(x \equiv \epsilon)$
- first[0] := $\{x \mid x \neq \epsilon\}$
- // (no equation for next)

$D(S \to E)$:



$D(E \to x)$:



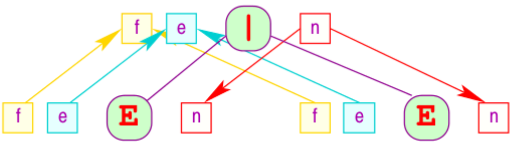
$$D(S \to E) = \{ (empty[1], empty[0]), (first[1], first[0]) \}$$

$$D(E \to x) = \{ \; \}$$

Regular Expressions: Rules for Alternative

$E \to E | E$:
- empty[0] := empty[1] $\vee$ empty[2]
- first[0] := first[1] $\cup$ first[2]
- next[1] := next[0]
- next[2] := next[0]

$D(E \to E | E)$:

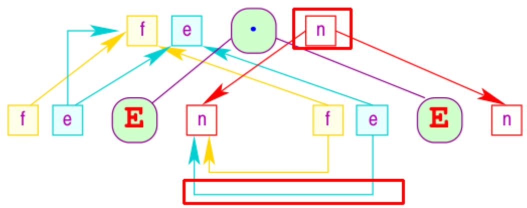


$$D(E \to E|E) = \{ (empty[1], empty[0]), (empty[2], empty[0]), (first[1], first[0]), (first[2], first[0]), (next[0], next[2]), (next[0], next[1]) \}$$

Regular Expressions: Rules for Concatenation

$E \to E \cdot E$:
- empty[0] := empty[1] $\wedge$ empty[2]
- first[0] := first[1] $\cup$ (empty[1] ? first[2] : $\emptyset$)
- next[1] := first[2] $\cup$ (empty[2] ? next[0] : $\emptyset$)
- next[2] := next[0]

$D(E \to E \cdot E)$:



$$D(E \to E \cdot E) = \{ (empty[1], empty[0]), (empty[2], empty[0]), (empty[2], next[1]), (first[1], first[0]), (first[2], first[0]), (first[2], next[1]), (next[0], next[2]), (next[0], next[1]) \}$$

Regular Expressions: Kleene-Star and '?'

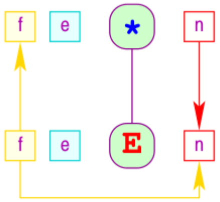
$E \to E*$:

empty$[0]$:= t
first$[0]$:= first$[1]$
next$[1]$:= first$[1]$ ∪ next$[0]$

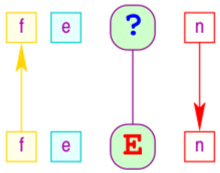
$E \to E?$:

empty$[0]$:= t
first$[0]$:= first$[1]$
next$[1]$:= next$[0]$

$D(E \to E*)$:



$D(E \to E?)$:



$D(E \to E*) = \{ (first[1], first[0]), (first[1], next[2]), (next[0], next[1]) \}$

$D(E \to E?) = \{ (first[1], first[0]), (next[0], next[1]) \}$

Challenges for General Attribute Systems

Static evaluation
Is there a static evaluation strategy, which is generally applicable?

- an evaluation strategy can only exist, if for *any* derivation tree the dependencies between attributes are acyclic
- it is *DEXPTIME*-complete to check for cyclic dependencies [Jazayeri, Odgen, Rounds, 1975]

Challenges for General Attribute Systems

Static evaluation
Is there a static evaluation strategy, which is generally applicable?

- an evaluation strategy can only exist, if for *any* derivation tree the dependencies between attributes are acyclic
- it is *DEXPTIME*-complete to check for cyclic dependencies [Jazayeri, Odgen, Rounds, 1975]

Ideas
1. Let the *User* specify the strategy
2. Determine the strategy dynamically
3. Automate *subclasses* only

Subclass: Strongly Acyclic Attribute Dependencies

Idea: For all nonterminals X compute a set $\mathcal{R}(X)$ of relations between its attributes, as an *overapproximation of the global dependencies* between root attributes of every production for X.

Describe $\mathcal{R}(X)$s as sets of relations, similar to $D(p)$ by

- setting up each production $X \mapsto X_1 \dots X_k$'s effect on the relations of $\mathcal{R}(X)$
- compute effect on all so far accumulated evaluations of each rhs X_i's $\mathcal{R}(X_i)$
- iterate until stable

Subclass: Strongly Acyclic Attribute Dependencies

The 3-ary operator $L[p,i]$ re-decorates relations from L

$$L[p,i] = \{(p.a[i], p.b[i]) \mid (a,b) \in L\}$$

π_0 projects only onto relations between root elements only

$$\pi_0(S) = \{(a,b) \mid (p.a[0], p.b[0]) \in S\}$$

Subclass: Strongly Acyclic Attribute Dependencies

The 3-ary operator $L[p,i]$ re-decorates relations from L

$$L[p,i] = \{(p.a[i], p.b[i]) \mid (a,b) \in L\}$$

π_0 projects only onto relations between root elements only

$$\pi_0(S) = \{(a,b) \mid (p.a[0], p.b[0]) \in S\}$$

root-projects the transitive closure of relations from the L_is and $D(p)$

$$[\![p]\!]^\sharp(L_1,\dots,L_k) = \pi_0((D(p) \cup L_1[p,1] \cup \dots \cup L_k[p,k])^+)$$

Subclass: Strongly Acyclic Attribute Dependencies

The 3-ary operator $L[p,i]$ re-decorates relations from L

$$L[p,i] = \{(p.a[i], p.b[i]) \mid (a,b) \in L\}$$

π_0 projects only onto relations between root elements only

$$\pi_0(S) = \{(a,b) \mid (p.a[0], p.b[0]) \in S\}$$

root-projects the transitive closure of relations from the L_is and $D(p)$

$$[\![p]\!]^\sharp(L_1,\dots,L_k) = \pi_0((D(p) \cup L_1[p,1] \cup \dots \cup L_k[p,k])^+)$$

Subclass: Strongly Acyclic Attribute Dependencies

The 3-ary operator $L[p,i]$ re-decorates relations from L

$$L[p,i] = \{(p.a[i], p.b[i]) \mid (a,b) \in L\}$$

π_0 projects only onto relations between root elements only

$$\pi_0(S) = \{(a,b) \mid (p.a[0], p.b[0]) \in S\}$$

root-projects the transitive closure of relations from the L_is and $D(p)$

$$[\![p]\!]^\sharp(L_1,\dots,L_k) = \pi_0((D(p) \cup L_1[p,1] \cup \dots \cup L_k[p,k])^+)$$

$\mathcal{R}$ maps symbols to relations (global attributes dependencies)

$$\mathcal{R}(X) = \bigcup\{[\![p]\!]^\sharp(\mathcal{R}(X_1),\dots,\mathcal{R}(X_k)) \mid p : X \to X_1 \dots X_k\} \mid X \in N$$

$$\mathcal{R}(X) \supseteq \emptyset \quad \mid X \in N \qquad \wedge \qquad \mathcal{R}(a) = \emptyset \quad \mid a \in T$$

Subclass: Strongly Acyclic Attribute Dependencies

The 3-ary operator $L[p,i]$ re-decorates relations from L

$$L[p,i] = \{(p.a[i], p.b[i]) \mid (a,b) \in L\}$$

π_0 projects only onto relations between root elements only

$$\pi_0(S) = \{(a,b) \mid (p.a[0], p.b[0]) \in S\}$$

root-projects the transitive closure of relations from the L_is and $D(p)$

$$[\![p]\!]^\sharp(L_1,\ldots,L_k) = \pi_0((D(p) \cup L_1[p,1] \cup \ldots \cup L_k[p,k])^+)$$

$\mathcal{R}$ maps symbols to relations (global attributes dependencies)

$$\mathcal{R}(X) = \bigcup\{[\![p]\!]^\sharp(\mathcal{R}(X_1),\ldots,\mathcal{R}(X_k)) \mid p : X \to X_1 \ldots X_k\} \mid X \in N$$

$$\mathcal{R}(X) \supseteq \emptyset \quad \mid X \in N \qquad \wedge \qquad \mathcal{R}(a) = \emptyset \quad \mid a \in T$$

Strongly Acyclic Grammars

The system of inequalities $\mathcal{R}(X)$

- characterizes the class of strongly acyclic Dependencies
- has a unique least solution $\mathcal{R}^\star(X)$ (as $[\![.]\!]^\sharp$ is monotonic)

Subclass: Strongly Acyclic Attribute Dependencies

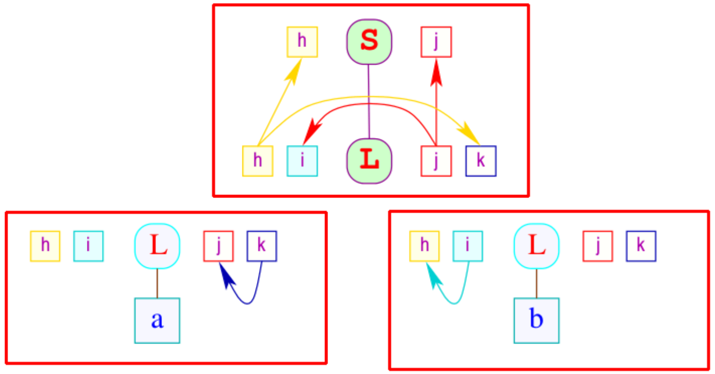
Strongly Acyclic Grammars

If all $D(p) \cup \mathcal{R}^\star(X_1)[p,1] \cup \ldots \cup \mathcal{R}^\star(X_k)[p,k]$ are acyclic for all $p \in G$, G is strongly acyclic.

Idea: we compute the least solution $\mathcal{R}^\star(X)$ of $\mathcal{R}(X)$ by a fixpoint computation, starting from $\mathcal{R}(X) = \emptyset$.

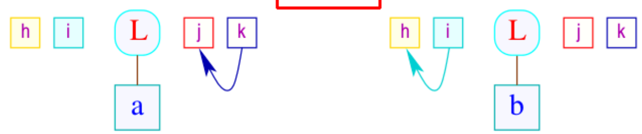
Example: Strong Acyclic Test

Given grammar $S \to L,\ L \to a \mid b$. Dependency graphs D_p:



Example: Strong Acyclic Test

Start with computing $\mathcal{R}(L) = [\![L \to a]\!]^\sharp() \sqcup [\![L \to b]\!]^\sharp()$:



- terminal symbols do not contribute dependencies

Subclass: Strongly Acyclic Attribute Dependencies

The 3-ary operator $L[p,i]$ re-decorates relations from L

$$L[p,i] = \{(p.a[i], p.b[i]) \mid (a,b) \in L\}$$

π_0 projects only onto relations between root elements only

$$\pi_0(S) = \{(a,b) \mid (p.a[0], p.b[0]) \in S\}$$

root-projects the transitive closure of relations from the L_is and $D(p)$

$$[\![p]\!]^\sharp(L_1, \ldots, L_k) = \pi_0((D(p) \cup L_1[p,1] \cup \ldots \cup L_k[p,k])^+)$$

$\mathcal{R}$ maps symbols to relations (global attributes dependencies)

$$\mathcal{R}(X) = \bigcup\{[\![p]\!]^\sharp(\mathcal{R}(X_1), \ldots, \mathcal{R}(X_k)) \mid p : X \to X_1 \ldots X_k\} \mid X \in N$$

$$\mathcal{R}(X) \supseteq \emptyset \quad \mid X \in N \qquad \wedge \qquad \mathcal{R}(a) = \emptyset \quad \mid a \in T$$

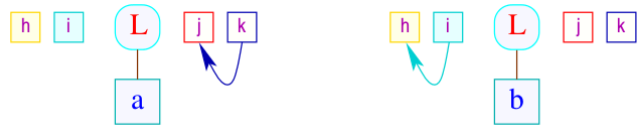
Strongly Acyclic Grammars

The system of inequalities $\mathcal{R}(X)$

- characterizes the class of strongly acyclic Dependencies
- has a unique least solution $\mathcal{R}^\star(X)$ (as $[\![.]\!]^\sharp$ is monotonic)

Example: Strong Acyclic Test

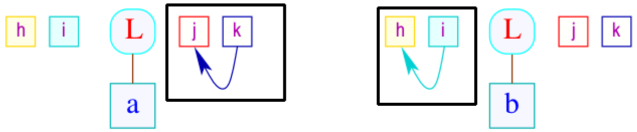
Start with computing $\mathcal{R}(L) = [\![L \to a]\!]^\sharp() \sqcup [\![L \to b]\!]^\sharp()$:



1. terminal symbols do not contribute dependencies

Example: Strong Acyclic Test

Start with computing $\mathcal{R}(L) = [\![L \to a]\!]^\sharp() \sqcup [\![L \to b]\!]^\sharp()$:



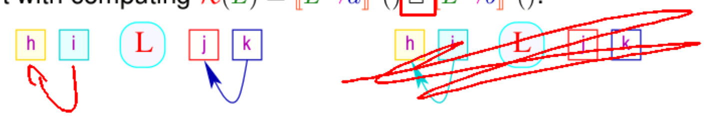
1. terminal symbols do not contribute dependencies

check for cycles!

2. transitive closure of all relations in $(D(L \to a))^+$ and $(D(L \to b))^+$

Example: Strong Acyclic Test

Start with computing $\mathcal{R}(L) = [\![L \to a]\!]^\sharp() \sqcup [\![L \to b]\!]^\sharp()$:



1. terminal symbols do not contribute dependencies
2. transitive closure of all relations in $(D(L \to a))^+$ and $(D(L \to b))^+$
3. apply π_0

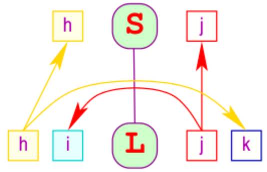
Example: Strong Acyclic Test

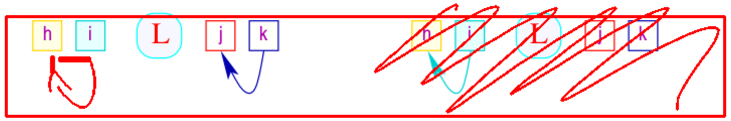
Start with computing $\mathcal{R}(L) = [\![L \to a]\!]^\sharp() \sqcup [\![L \to b]\!]^\sharp()$:

1. terminal symbols do not contribute dependencies
2. transitive closure of all relations in $(D(L \to a))^+$ and $(D(L \to b))^+$
3. apply π_0
4. $\mathcal{R}(L) = \{(k, j), (i, h)\}$

Example: Strong Acyclic Test

Continue with $\mathcal{R}(S) = [\![S \to L]\!]^\sharp(\mathcal{R}(L))$:

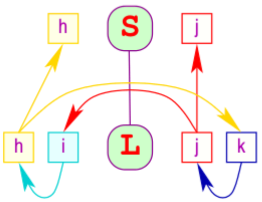




1. re-decorate $\mathcal{R}(L)$ via $L[S \to L, 1]$

Example: Strong Acyclic Test

Continue with $\mathcal{R}(S) = [\![S \to L]\!]^\sharp(\mathcal{R}(L))$:

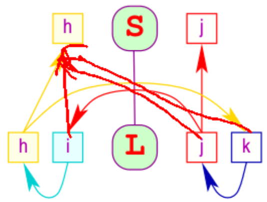


1. re-decorate $\mathcal{R}(L)$ via $L[S \to L, 1]$
2. transitive closure of all relations
$(D(S \to L) \cup \{(p.k[1], p.j[1])\} \cup \{(p.i[1], p.h[1])\})^+$

check for cycles!

Example: Strong Acyclic Test

Continue with $\mathcal{R}(S) = [\![S \to L]\!]^\sharp(\mathcal{R}(L))$:



1. re-decorate $\mathcal{R}(L)$ via $L[S \to L, 1]$
2. transitive closure of all relations
$(D(S \to L) \cup \{(p.k[1], p.j[1])\} \cup \{(p.i[1], p.h[1])\})^+$

check for cycles!

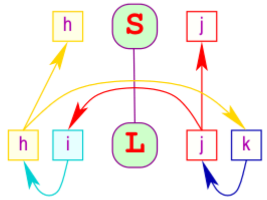
Example: Strong Acyclic Test

Start with computing $\mathcal{R}(L) = [\![L\to a]\!]^\sharp() \sqcup [\![L\to b]\!]^\sharp()$:

1. terminal symbols do not contribute dependencies
2. transitive closure of all relations in $(D(L\to a))^+$ and $(D(L\to b))^+$
3. apply π_0
4. $\mathcal{R}(L) = \{(k,j),(i,h)\}$

Example: Strong Acyclic Test

Continue with $\mathcal{R}(S) = [\![S\to L]\!]^\sharp(\mathcal{R}(L))$:



1. re-decorate $\mathcal{R}(L)$ via $L[S\to L,1]$
2. transitive closure of all relations $(D(S\to L) \cup \{(p.k[1], p.j[1])\} \cup \{(p.i[1], p.h[1])\})^+$

check for cycles!

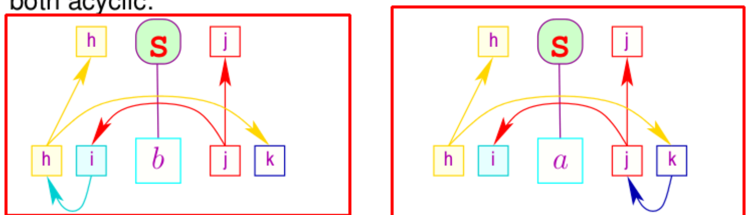
Example: Strong Acyclic Test

Continue with $\mathcal{R}(S) = [\![S\to L]\!]^\sharp(\mathcal{R}(L))$:

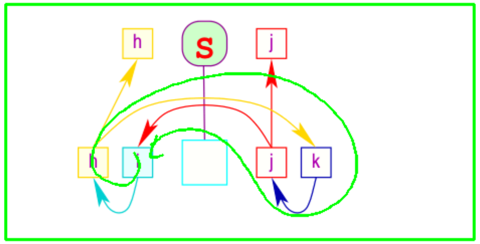
1. re-decorate $\mathcal{R}(L)$ via $L[S\to L,1]$
2. transitive closure of all relations $(D(S\to L) \cup \{(p.k[1], p.j[1])\} \cup \{(p.i[1], p.h[1])\})^+$
3. apply π_0

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 contribute to a cycle when computing $\mathcal{R}(S)$:



From Dependencies to Evaluation Strategies

Possible strategies: