Script generated by TTT

Title: Petter: Compilerbau (21.06.2018)

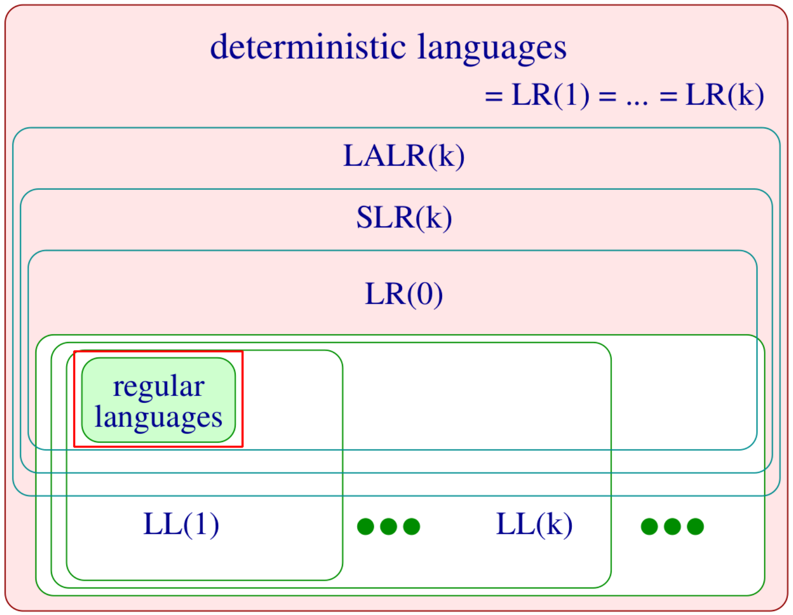
Date: Thu Jun 21 14:15:17 CEST 2018

Duration: 94:19 min

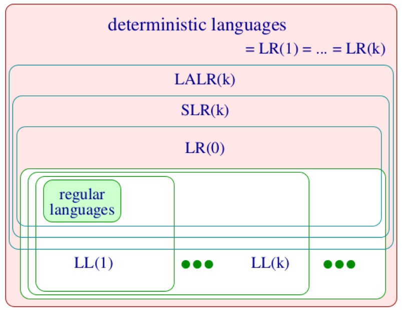
Pages: 47

Chapter 5:
Summary

Parsing Methods



Parsing Methods

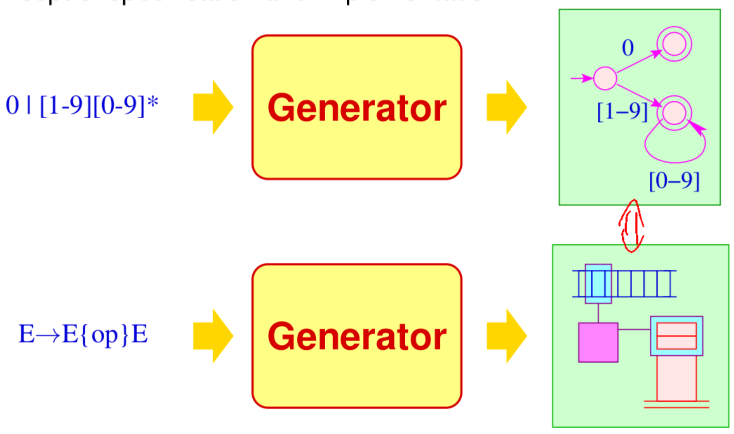


Discussion:

- All contextfree languages, that can be parsed with a deterministic pushdown automaton, can be characterized with an LR(1)-grammar.
- LR(0)-grammars describe all prefixfree deterministic contextfree languages
- The language-classes of LL(k)-grammars form a hierarchy within the deterministic contextfree languages.

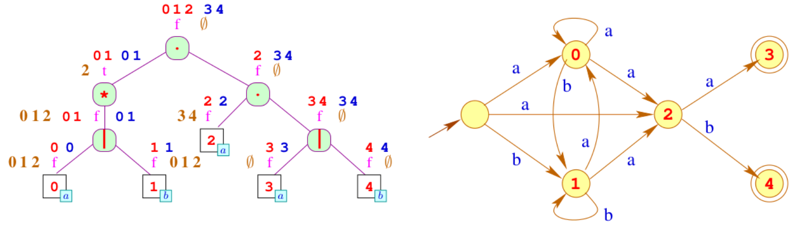
Lexical and Syntactical Analysis:

Concept of specification and implementation:

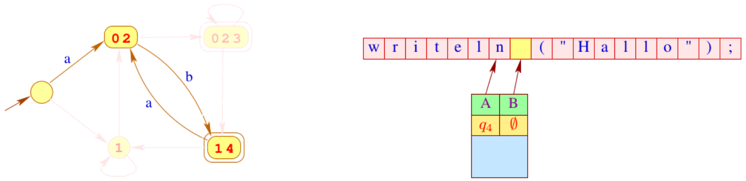


Lexical and Syntactical Analysis:

From Regular Expressions to Finite Automata



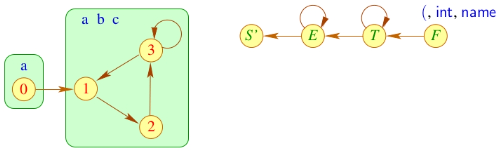
From Finite Automata to Scanners



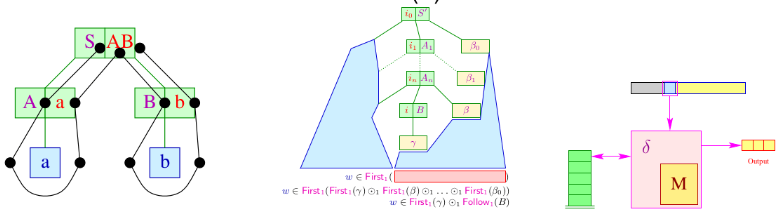
Lexical and Syntactical Analysis:

Computation of lookahead sets:

$F_\epsilon(S') \supseteq F_\epsilon(E)$	$F_\epsilon(E) \supseteq F_\epsilon(E)$
$F_\epsilon(E) \supseteq F_\epsilon(T)$	$F_\epsilon(T) \supseteq F_\epsilon(T)$
$F_\epsilon(T) \supseteq F_\epsilon(F)$	$F_\epsilon(F) \supseteq \{(, \text{name}, \text{int}\}$

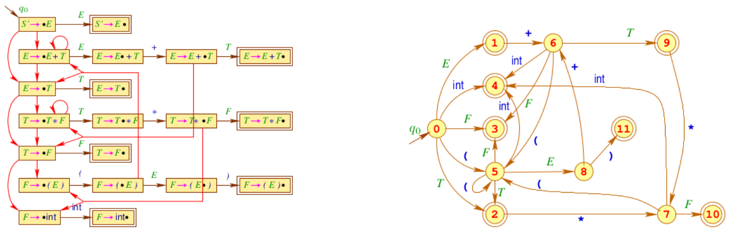


From Item-Pushdown Automata to LL(1)-Parsers:

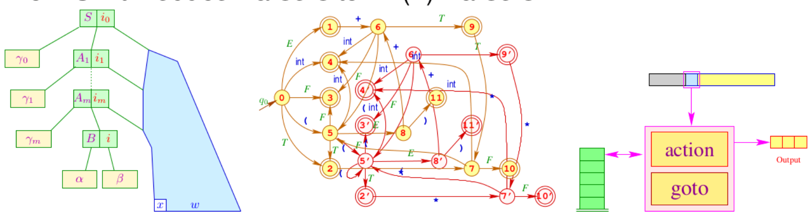


Lexical and Syntactical Analysis:

From characteristic to canonical Automata:



From Shift-Reduce-Parsers to LR(1)-Parsers:



Topic:

Semantic Analysis

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

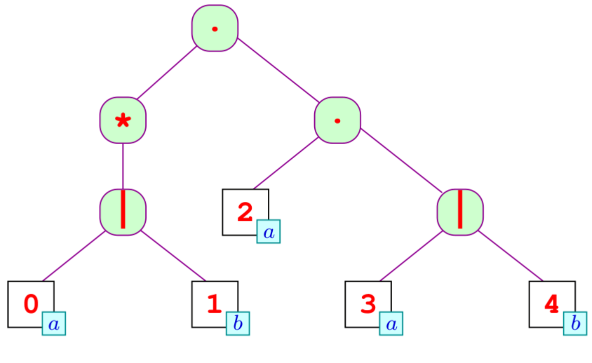
Definition attribute grammar

An attribute grammar is a CFG extended by

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

Example: Computation of the $\text{empty}[r]$ Attribute

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* 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 $\rightarrow$ root)
- inherited if its value is always propagated downwards in the tree (in the direction root $\rightarrow$ 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: $r \equiv \boxed{i \mid x}$ we define $\text{empty}[r] = (x \equiv \epsilon)$.

otherwise:

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

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*:

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

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

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

- 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 syntax tree
- in the example, the parent node is always depending on children only
  $\leadsto$ a depth-first post-order traversal is possible
- in general, variable dependencies can be much *more complex*

Simultaneous Computation of Multiple Attributes

Computing empty, first, next from regular expressions:

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

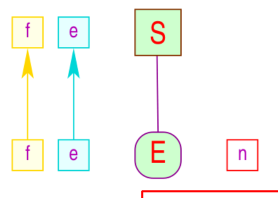
$$\begin{array}{lcl} \text{empty}[0] & := & \text{empty}[1] \\ \text{first}[0] & := & \text{first}[1] \\ \text{next}[1] & := & \emptyset \end{array}$$

$\boxed{E \to x}$:

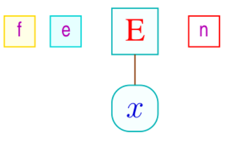
$$\begin{array}{lcl} \text{empty}[0] & := & (x \equiv \epsilon) \\ \text{first}[0] & := & \{x \mid x \neq \epsilon\} \end{array}$$

// (no equation for next)

$D(S \to E)$:



$D(E \to x)$:



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

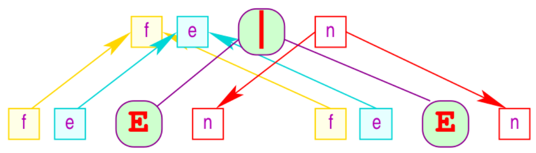
$D(E \to x) = \{ \quad \}$

Regular Expressions: Rules for Alternative

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

$$\boxed{\begin{aligned} \mathsf{empty}[0] &:= \mathsf{empty}[1] \vee \mathsf{empty}[2] \\ \mathsf{first}[0] &:= \mathsf{first}[1] \cup \mathsf{first}[2] \\ \mathsf{next}[1] &:= \mathsf{next}[0] \\ \mathsf{next}[2] &:= \mathsf{next}[0] \end{aligned}}$$

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



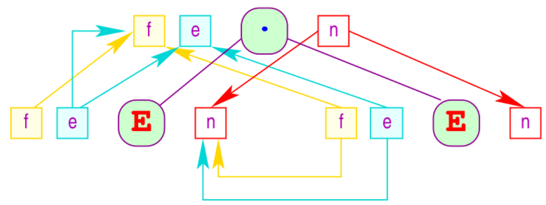
$$D(E \to E|E) = \{ \boxed{\begin{aligned} &(empty[1], empty[0]), \\ &(empty[2], empty[0]), \\ &(first[1], first[0]), \\ &(first[2], first[0]), \\ &(next[0], next[2]), \\ &(next[0], next[1]) \} \end{aligned}}$$

Regular Expressions: Rules for Concatenation

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

$$\begin{aligned} \mathsf{empty}[0] &:= \mathsf{empty}[1] \wedge \mathsf{empty}[2] \\ \mathsf{first}[0] &:= \mathsf{first}[1] \cup (\mathsf{empty}[1] \,?\, \mathsf{first}[2] : \emptyset) \\ \mathsf{next}[1] &:= \mathsf{first}[2] \cup (\mathsf{empty}[2] \,?\, \mathsf{next}[0] : \emptyset) \\ \mathsf{next}[2] &:= \mathsf{next}[0] \end{aligned}$$

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



$$D(E \to E \cdot E) = \{ \begin{aligned} &(empty[1], empty[0]), \\ &\cancel{(empty[2], empty[0]),} \\ &\boxed{(empty[2], next[1]),} \\ &\cancel{(empty[1], first[0]),} \\ &(first[1], first[0]), \\ &(first[2], first[0]), \\ &(first[2], next[1]), \\ &(next[0], next[2]), \\ &(next[0], next[1]) \} \end{aligned}$$

Regular Expressions: Kleene-Star and '?'

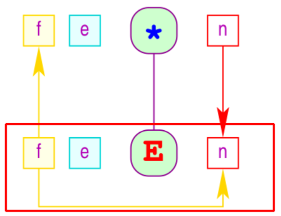
$\boxed{E \to E*}$:

$$\begin{aligned} \mathsf{empty}[0] &:= t \\ \mathsf{first}[0] &:= \mathsf{first}[1] \\ \mathsf{next}[1] &:= \mathsf{first}[1] \cup \mathsf{next}[0] \end{aligned}$$

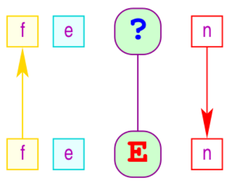
$\boxed{E \to E?}$:

$$\begin{aligned} \mathsf{empty}[0] &:= t \\ \mathsf{first}[0] &:= \mathsf{first}[1] \\ \mathsf{next}[1] &:= \mathsf{next}[0] \end{aligned}$$

$D(E \to E*)$:



$D(E \to E?)$:



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

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

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 $X \mapsto X_1 \ldots X_k$'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 2-ary operator $L[i]$ re-decorates relations from L

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

π_0 projects only onto relations between root elements only

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

Subclass: Strongly Acyclic Attribute Dependencies

The 2-ary operator $L[i]$ re-decorates relations from L

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

π_0 projects only onto relations between root elements only

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

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

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

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

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

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

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

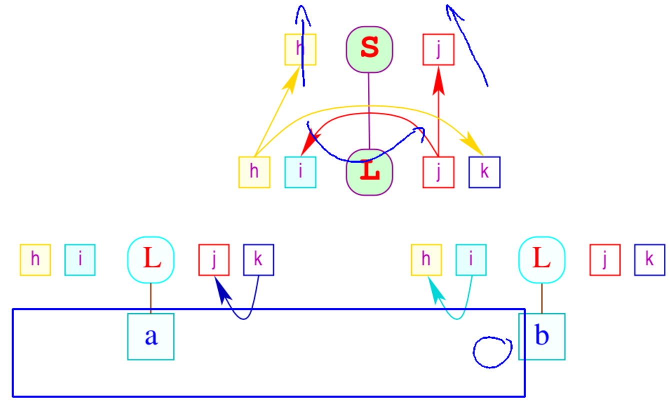
Strongly Acyclic Grammars

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

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

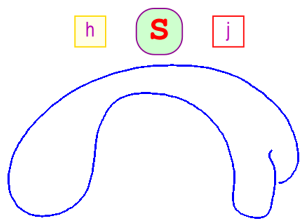
Example: Strong Acyclic Test

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



Example: Strong Acyclic Test

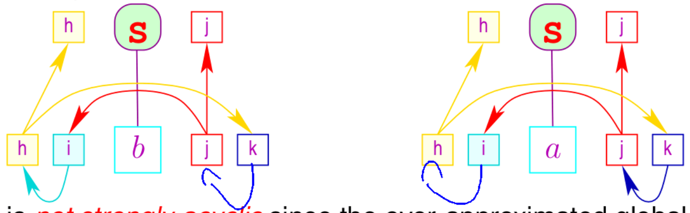
Continue with $\mathcal{R}(S) = [\![S \to L]\!]^\sharp(\mathcal{R}(L))$:



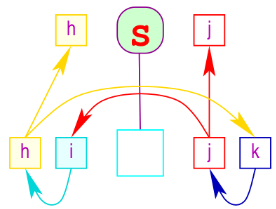
1. re-decorate and embed $\mathcal{R}(L)[1]$
2. transitive closure of all relations $(D(S \to L) \cup \{(k[1], j[1])\} \cup \{(i[1], h[1])\})^+$
3. apply π_0
4. $\mathcal{R}(S) = \{\}$

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 over-approximated global dependence graph for the non-terminal L contributes to a cycle when computing $\mathcal{R}(S)$:



From Dependencies to Evaluation Strategies

Possible strategies:

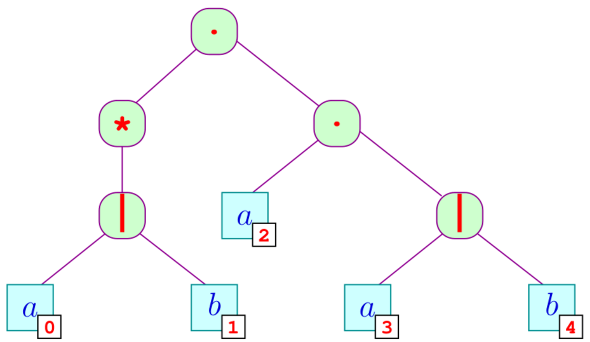
Linear Order from Dependency Partial Order

Possible *automatic* strategies:

Example: Demand-Driven Evaluation

Compute next at leaves a_2, a_3 and b_4 in the expression $(a|b)^*a(a|b)$:

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



Demand-Driven Evaluation

Observations

- each node must contain a pointer to its parent
- *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

$\rightsquigarrow$ the algorithm is not local

Evaluation in Passes

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

Evaluation in Passes

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

Strongly Acyclic Attribute Systems'

attributes have to be evaluated for each production p according to

$$D(p) \cup \mathcal{R}^\star(X_1)[1] \cup \ldots \cup \mathcal{R}^\star(X_k)[k]$$

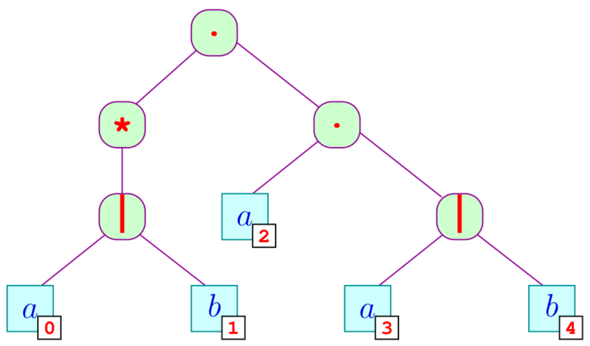
Implementation

- determine a sequence of child visitations such that the most number of attributes are possible to evaluate
- in each pass at least one new attribute is evaluated
  - requires at most n passes for evaluating n attributes
  - find a strategy to evaluate more attributes
    $\leadsto$ optimization problem

Note: evaluating attribute set $\{a[0], \ldots, z[0]\}$ for rule $N \to \ldots N \ldots$ may evaluate a different attribute set of its children
$\leadsto 2^k - 1$ evaluation functions for N (with k as the number of attributes)

Implementing State

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



Example: Implementing Numbering of Leafs

Idea:

- use helper attributes pre and post
- in pre we pass the value for the first leaf down (inherited attribute)
- in post we pass the value of the last leaf up (synthetic attribute)

root:

$$\begin{array}{lll} \text{pre}[0] & := & 0 \\ \text{pre}[1] & := & \text{pre}[0] \\ \text{post}[0] & := & \text{post}[1] \end{array}$$

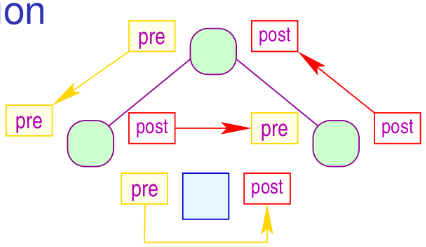
node:

$$\begin{array}{lll} \text{pre}[1] & := & \text{pre}[0] \\ \text{pre}[2] & := & \text{post}[1] \\ \text{post}[0] & := & \text{post}[2] \end{array}$$

leaf:

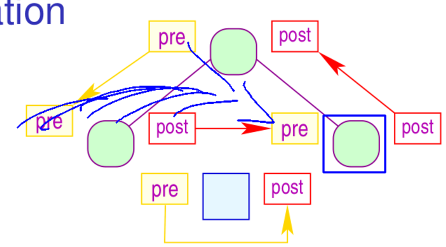
$$\text{post}[0] := \text{pre}[0] + 1$$

L-Attribution



- the attribute system is apparently strongly acyclic

L-Attribution



- the attribute system is apparently strongly acyclic
- each node computes
  - the inherited attributes before descending into a child node (corresponding to a pre-order traversal)
  - the synthetic attributes after returning from a child node (corresponding to post-order traversal)

Definition L-Attributed Grammars

An attribute system is L-attributed, if for all productions $S \rightarrow S_1 \ldots S_n$ every inherited attribute of S_j where $1 \leq j \leq n$ only depends on

1. the attributes of $S_1, S_2, \ldots S_{j-1}$ and
2. the inherited attributes of S.

L-Attribution

Background:

- the attributes of an L-attributed grammar can be evaluated during parsing
- important if no syntax tree is required or if error messages should be emitted while parsing
- example: pocket calculator

Practical Applications

- symbol tables, type checking/inference, and simple code generation can all be specified using L-attributed grammars

Implementation of Attribute Systems via a *Visitor*

- class with a method for every non-terminal in the grammar

```
public abstract class Regex {
  public abstract void accept(Visitor v);
}
```

- attribute-evaluation works via *pre-order / post-order callbacks*

```
public interface Visitor {
  default void pre(OrEx re)   {}
  default void pre(AndEx re)  {}
  ...
  default void post(OrEx re)  {}
  default void post(AndEx re){}
}
```

- we pre-define a depth-first traversal of the syntax tree

```
public class OrEx extends Regex {
  Regex l,r;
  public void accept(Visitor v) {
     v.pre(this);l.accept(v);v.inter(this);
     r.accept(v); v.post(this);
} }
```


Example: Leaf Numbering

```
public abstract class AbstractVisitor
        implements Visitor {
  public void pre(OrEx re)   { pr(re); }
  public void pre(AndEx re)  { pr(re); }
  ...
  public void post(OrEx re)  { po(re); }
  public void post(AndEx re){ po(re); }
  abstract void po(BinEx re);
  abstract void in(BinEx re);
  abstract void pr(BinEx re);
}
public class LeafNum extends AbstractVisitor {
  public LeafNum(Regex r) { n.put(r,0);r.accept(this);}
  public Map<Regex,Integer> n = new HashMap<>();
  public void pr(Const r) { n.put(r,  n.get(r)+1); }
  public void pr(BinEx r) { n.put(r.l,n.get(r));   }
  public void in(BinEx r) { n.put(r.r,n.get(r.l)); }
  public void po(BinEx r) { n.put(r,n.get(r.r));   }
}
```


Example: Leaf Numbering

```
public abstract class AbstractVisitor
        implements Visitor {
  public void pre(OrEx re)   { pr(re); }
  public void pre(AndEx re)  { pr(re); }
  ...
  public void post(OrEx re)  { po(re); }
  public void post(AndEx re){ po(re); }
  abstract void po(BinEx re);
  abstract void in(BinEx re);
  abstract void pr(BinEx re);
}
public class LeafNum extends AbstractVisitor {
  public LeafNum(Regex r) { n.put(r,0);r.accept(this);}
  public Map<Regex,Integer> n = new HashMap<>();
  public void pr(Const r) { n.put(r,  n.get(r)+1); }
  public void pr(BinEx r) { n.put(r.l,n.get(r));   }
  public void in(BinEx r) { n.put(r.r,n.get(r.l)); }
  public void po(BinEx r) { n.put(r,n.get(r.r));   }
}
```