**Petter: Compilerbau (21.06.2018)**

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

**Pages:** 47

---

**Chapter 5:**

**Summary**

---

**Parsing Methods**

- **Deterministic languages**
  - $\text{LR}(0) = \cdots = \text{LR}(k)$
  - LALR(k)
  - SLR(k)
  - LL(1), LL(k)

**Discussion:**

- All context-free languages, that can be parsed with a deterministic pushdown automaton, can be characterized with an $\text{LR}(1)$-grammar.
- LR(0)-grammars describe all prefix-free deterministic context-free languages.
- The language-classes of LL(k)-grammars form a hierarchy within the deterministic context-free languages.
Topic: Semantic Analysis

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

Chapter 1: Attribute Grammars

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 **empty** \([r]\) Attribute

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

![Syntax tree diagram]

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

### 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:** \(r \equiv t \mid a\) we define

\[
\text{empty}[r] = (a \equiv \epsilon).
\]

**otherwise:**

\[
\begin{align*}
\text{empty}[r_1 \mid r_2] &= \text{empty}[r_1] \lor \text{empty}[r_2] \\
\text{empty}[r_1 \cdot r_2] &= \text{empty}[r_1] \land \text{empty}[r_2] \\
\text{empty}[r_1'] &= t \\
\text{empty}[r_1?] &= t
\end{align*}
\]
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:

- We use consecutive indices to refer to neighbouring attributes

\[
\begin{align*}
\text{attribute}_i[0] & : \text{the attribute of the current root node} \\
\text{attribute}_i[i] & : \text{the attribute of the } i\text{-th child} \quad (i > 0)
\end{align*}
\]

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

Simultaneous Computation of Multiple Attributes

Computing empty, first, next from regular expressions:

\[
\begin{align*}
S \rightarrow E & : \\
\text{empty}[0] & := \text{empty}[1] \\
\text{first}[0] & := \text{first}[1] \\
\text{next}[1] & := \emptyset \\
E \rightarrow x & : \\
\text{empty}[0] & := (x \equiv e) \\
\text{first}[0] & := \{x \mid x \neq e\} \\
\end{align*}
\]

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

\[
\text{D}(E \rightarrow x) = \{\}
\]
Regular Expressions: Rules for Alternative

\[ E \rightarrow E | E \]

\[
\begin{align*}
  \text{empty}[0] & := \text{empty}[1] \lor \text{empty}[2] \\
  \text{first}[0] & := \text{first}[1] \cup \text{first}[2] \\
  \text{next}[1] & := \text{next}[0] \\
  \text{next}[2] & := \text{next}[0]
\end{align*}
\]

\[ D(E \rightarrow E) : \]

\[
\{ (\text{empty}[1], \text{empty}[0]), \\
    (\text{empty}[2], \text{empty}[0]), \\
    (\text{first}[1], \text{first}[0]), \\
    (\text{first}[2], \text{first}[0]), \\
    (\text{next}[0], \text{next}[2]), \\
    (\text{next}[0], \text{next}[1]) \}
\]

Regular Expressions: Rules for Concatenation

\[ E \rightarrow E \cdot E \]

\[
\begin{align*}
  \text{empty}[0] & := \text{empty}[1] \land \text{empty}[2] \\
  \text{first}[0] & := \text{first}[1] \cup \text{empty}[1] \lor \text{first}[2] : \emptyset \\
  \text{next}[1] & := \text{first}[2] \lor \text{empty}[2] \lor \text{next}[0] : \emptyset \\
  \text{next}[2] & := \text{next}[0]
\end{align*}
\]

\[ D(E \rightarrow E \cdot E) : \]

\[
\{ (\text{empty}[1], \text{empty}[0]), \\
    (\text{empty}[2], \text{empty}[0]), \\
    (\text{first}[1], \text{first}[0]), \\
    (\text{first}[2], \text{first}[0]), \\
    (\text{next}[0], \text{next}[2]), \\
    (\text{next}[0], \text{next}[1]) \}
\]

Regular Expressions: Kleene-Star ‘?’

\[ E \rightarrow E^* \]

\[
\begin{align*}
  \text{empty}[0] & := t \\
  \text{first}[0] & := \text{first}[1] \\
  \text{next}[1] & := \text{first}[1] \lor \text{next}[0]
\end{align*}
\]

\[ D(E \rightarrow E^*) : \]

\[
\{ (\text{first}[1], \text{first}[0]), \\
    (\text{first}[1], \text{next}[2]), \\
    (\text{next}[0], \text{next}[1]) \}
\]

\[ E \rightarrow E^? \]

\[
\begin{align*}
  \text{empty}[0] & := t \\
  \text{first}[0] & := \text{first}[1] \\
  \text{next}[1] & := \text{next}[0]
\end{align*}
\]

\[ D(E \rightarrow E^?) : \]

\[
\{ (\text{first}[1], \text{first}[0]), \\
    (\text{next}[0], \text{next}[1]) \}
\]

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 \textbf{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 \(\text{DEXPTIME}\)-complete to check for cyclic dependencies
  [Jazayeri, Odgen, Rounds, 1975]

**Ideas**
- Let the User specify the strategy
- Determine the strategy dynamically
- Automate subclases only

Subclass: Strongly Acyclic Attribute Dependencies

The 2-ary operator \(L[i]\) re-decorates relations from \(L\)

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

\(\pi_0\) 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

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 \(X \rightarrow X_1 \ldots X_k\)'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 \(L\)

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

\(\pi_0\) projects only onto relations between root elements only

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

\(\pi_0\) 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 \(L[i]\)s and \(D(p)\)

\[
[p]^\pi(L_1, \ldots, L_k) = \pi_0((D(p) \cup L_1[1] \cup \ldots \cup L_k[k])^*)
\]

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

\[
\mathcal{R}(X) = \bigcup \{[p]^\pi(\mathcal{R}(X_1), \ldots, \mathcal{R}(X_k)) \mid p : X \rightarrow X_1 \ldots X_k \} \mid X \in N
\]

\[
\mathcal{R}(X) \supseteq \emptyset \quad \land \quad \mathcal{R}(a) = \emptyset \quad \mid a \in T
\]

Strongly Acyclic Grammars

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

- characterizes the class of strongly acyclic Dependencies
- has a unique least solution \(\mathcal{R}^*(X)\) (as \([.]^\pi\) is monotonic)
**Subclass: Strongly Acyclic Attribute Dependencies**

**Strongly Acyclic Grammars**

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

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

**Example: Strong Acyclic Test**

Continue with \( R(S) = [S \rightarrow L]^+(R(L)) \):

- re-decorate and embed \( R(L)[1] \)
- transitive closure of all relations \( (D(S \rightarrow L) \cup \{(k[1],j[1])\} \cup \{(i[1],h[1])\})^+ \)
- apply \( \pi_0 \)
- \( R(S) = \emptyset \)

**Example: Strong Acyclic Test**

Given grammar \( S \rightarrow L, L \rightarrow a \mid b \). Dependency graphs \( D_p \):

**Strong Acyclic and Acyclic**

The grammar \( S \rightarrow L, L \rightarrow a \mid b \) 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 \( R(S) \):
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 $(a|b)^*a(a|b)$:

```
[ ] :  next[1] := next[0]
     next[2] := next[0]
```

```
     next[2] := \text{next}[0]
```

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
- 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

Strongly Acyclic Attribute Systems

attributes have to be evaluated for each production \( p \) according to

\[
D(p) \cup R^*(X_1)[1] \cup \ldots \cup R^*(X_k)[k]
\]

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

\( \text{requires at most } n \text{ passes for evaluating } n \text{ attributes} \)

\( \text{find a strategy to evaluate more attributes} \)

\( \text{~ an optimization problem} \)

Note: evaluating attribute set \( \{a[0], \ldots, z[0]\} \) for rule \( N \rightarrow \ldots N \ldots \) may evaluate a different attribute set of its children

\( \sim 2^k - 1 \) evaluation functions for \( N \) (with \( k \) as the number of attributes)

Example: 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 \( \text{pre} \) and \( \text{post} \)
- in \( \text{pre} \) we pass the value for the first leaf down (inherited attribute)
- in \( \text{post} \) we pass the value of the last leaf up (synthetic attribute)

<table>
<thead>
<tr>
<th>root</th>
<th>pre[0] (:=) 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>pre[1] := pre[0]</td>
</tr>
<tr>
<td></td>
<td>post[0] := post[1]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>node</th>
<th>pre[1] := pre[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>post[0] := post[2]</td>
</tr>
</tbody>
</table>

| leaf | post[0] := pre[0] + 1 |
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
- the attributes of \( S_1, S_2, \ldots S_{j-1} \)
- the inherited attributes of \( S_i \)

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

- attribute-evaluation works via pre-order / post-order callbacks
  
  ```java
  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
  
  ```java
  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

```java
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);
}
```

```java
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(OrEx re) { n.put(r, n.get(r)+1); }
    public void pr(AndEx re) { po(re); }
    public void po(BinEx re) { n.put(r, n.get(r)+1); }
    public void in(BinEx re) { n.put(r, n.get(r)); }
    public void po(BinEx re) { n.put(r, n.get(r)); }
}
```