Script **generated by TTT**

Title: Petter: Compiler Construction (18.06.2020)
-40: Visitors for L-Attributed Grammars

Date: Fri Jun 19 10:37:59 CEST 2020

Duration: 14:33 min

Pages: 7

Practical Applications

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

Practical Applications

- symbol tables, type checking/inference, and simple code generation can all be specified using L-attributed grammars
- most applications *annotate* syntax trees with additional information

Practical Applications

- symbol tables, type checking/inference, and simple code generation can all be specified using L-attributed grammars
- most applications *annotate* syntax trees with additional information
- the nodes in a syntax tree usually have different *types* that depend on the non-terminal that the node represents

Practical Applications

- symbol tables, type checking/inference, and simple code generation can all be specified using L-attributed grammars
- most applications *annotate* syntax trees with additional information
- the nodes in a syntax tree usually have different *types* that depend on the non-terminal that the node represents

⇝ the different types of non-terminals are characterized by the set of attributes with which they are decorated

Practical Applications

- symbol tables, type checking/inference, and simple code generation can all be specified using L-attributed grammars
- most applications *annotate* syntax trees with additional information
- the nodes in a syntax tree usually have different *types* that depend on the non-terminal that the node represents

⇝ the different types of non-terminals are characterized by the set of attributes with which they are decorated

Example: Def-Use Analysis

- *a statement* may have two attributes containing valid identifiers: one ingoing (inherited) set and one outgoing (synthesised) set
- *an expression* only has an ingoing set

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); }
  ... /* redirecting to default handler for bin exprs */
  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 Map<Regex,Integer> pre  = new HashMap<>();
  public Map<Regex,Integer> post = new HashMap<>();
  public LeafNum (Regex r) { pre .put(r,0); r.accept(this); }
  public void pre(Const r) { post.put(r,    pre .get(r)+1); }
  public void pr (BinEx r) { pre .put(r.l,  pre .get(r)); }
  public void in (BinEx r) { pre .put(r.r,  post.get(r.l)); }
  public void po (BinEx r) { post.put(r,    post.get(r.r)); }
}
```