Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Attribute Grammars: A short tutorial Tree-Oriented Programming

S. Doaitse Swierstra (doaitse@cs.uu.nl), GPCE 2003
Arie Middelkoop (ariem@cs.uu.nl), LERNET 2008
Adapted for Brazil 2008/2009
Universiteit Utrecht

March 1, 2008

Implementing a Language

- Parsing is about syntax
- What about semantics?

Describing Semantics With a Grammar?

- Context-free grammars have limited expressiveness, and thus fail to describe:

Describing Semantics With a Grammar?

- Context-free grammars have limited expressiveness, and thus fail to describe:
  - Scope rules
  - Typing rules
  - Pretty printing
  - Code generation

Describing Semantics With a Grammar?

- Context-free grammars have limited expressiveness, and thus fail to describe:
  - Scope rules
  - Typing rules
  - Pretty printing
  - Code generation
- Are there extensions?

Parameterize Non-Terminal Symbols

Parameterize non-terminal symbols with values from some other domain: *attribute grammars* (Knuth)

$$
\begin{aligned}
E\langle read\ x\rangle &\rightarrow x \\
E\langle n + m\rangle &\rightarrow E\langle n\rangle \texttt{ "+" } E\langle m\rangle \\
E\langle n * m\rangle &\rightarrow E\langle n\rangle \texttt{ "*" } E\langle m\rangle
\end{aligned}
$$

An Attribute Grammar Consists Of:

- An underlying context free grammar

An Attribute Grammar Consists Of:

- An underlying context free grammar
- A description of which non-terminals have which attributes:
  - *Inherited* attributes, that are used or passing information *downwards* in the tree
  - *Synthesized* attributes that are used to pass information *upwards*

An Attribute Grammar Consists Of:

- An underlying context free grammar
- A description of which non-terminals have which attributes:
  - *Inherited* attributes, that are used or passing information *downwards* in the tree
  - *Synthesized* attributes that are used to pass information *upwards*
- For *each production* a description how to compute the:
  - Inherited attributes of the non-terminals in the *right hand side*
  - The synthesized attributes of the non-terminal at the *left hand side*
- In this way we describe *global* data flow over a tree, by defining *local* data-flow building blocks, corresponding to the productions of the grammar

Creating HTML From a Document

```
\section{Intro}
  \section{Section 1}
    \paragraph
      paragraph 1
    \end
    \paragraph
      paragraph 2
    \end
  \end
  \section{Section 2}
    \paragraph
      paragraph 1
    \end
    \paragraph
      paragraph 2
    \end
  \end \end
```

```
<h1>Intro</h1>
<h2>Section 1</h2>
<p>
  Paragraph 1
</p>
<p>
  Paragraph 2
</p>

<h2>Section 2</h2>
<p>
  Paragraph 1
</p>
<p>
  Paragraph 2
</p>
```

Introducing UUAG

- Special syntax for programming with attributes
- Domain specific language for specifying tree walks

Introducing UUAG

- Special syntax for programming with attributes
- Domain specific language for specifying tree walks

UUAG generates *semantic functions* which define the semantics of an *abstract syntax tree*.

Concrete and Abstract syntax

```
Docs  ::=  Doc*
Doc   ::=  "\section" "{" Text "}" Docs "\end"
         |  "\paragraph" Text "\end"
```

Concrete and Abstract syntax

```
Docs   ::=  Doc*
Doc    ::=  "\section" "{" Text "}" Docs "\end"
          |  "\paragraph" Text "\end"
```

Using a parser for the above concrete syntax, we produce a tree with the following *abstract syntax*:

DATA *Doc*	*Section*	*title* : *String*	*body* : *Docs*
	Paragraph	*text* : *String*	
DATA *Docs*	*Cons*	*hd* : *Doc*	*tl* : *Docs*
	Nil		

- *Docs* and *Doc* are non-terminals
- *Section* and *Paragraph* label different productions
- *title*, *body* and *string* are names for children

Synthesized attributes

Semantics: Our First Attribute!

- We introduce an attribute $html$ of type $String$ to return the generated html code in a synthesized attribute:

$$\textbf{ATTR}\ Doc\ Docs\ [\quad |\quad |\quad html : String\quad]$$

Semantics: Our First Attribute!

- We introduce an attribute $html$ of type $String$ to return the generated html code in a synthesized attribute:

$$\textbf{ATTR}\ Doc\ Docs\ [\quad |\quad |\quad html : String\quad]$$

- Nonterminal Doc has $html$ as attribute, so we now need to define for productions $Section$ and $Paragraph$ how to compute the value of this attribute. The same for productions $Cons$ and Nil of $Docs$.

Semantics: Our First Attribute!

- We introduce an attribute *html* of type *String* to return the generated html code in a synthesized attribute:

ATTR *Doc Docs* [| | *html* : *String*]

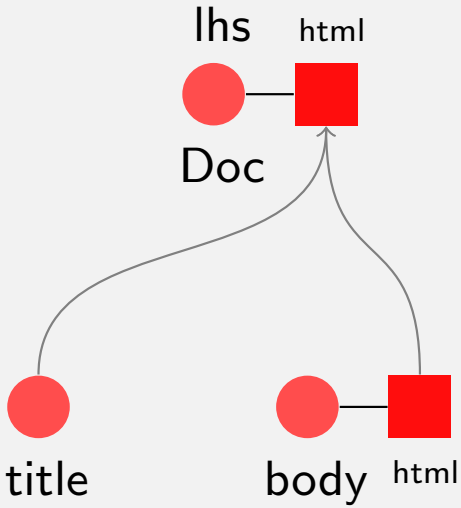
- Nonterminal Doc has *html* as attribute, so we now need to define for productions *Section* and *Paragraph* how to compute the value of this attribute. The same for productions *Cons* and *Nil* of *Docs*.
- Definitions for attributes are given in Haskell, with embedded references to attributes, in the form of `@<ntname>.<attrname>`:
- Assume for now that you can refer to:
  - the synthesized attributes defined on the children
  - values of child-terminals

A Picture for Section

$$\textbf{DATA}\ Doc \mid Section \quad title : \{String\} \quad body : Docs$$

$$\textbf{ATTR}\ Doc\ [\mid\mid html : \{String\}]$$

$$\textbf{SEM}\ Doc$$
$$\mid Section \quad \textbf{lhs}.html = \texttt{"<bf>"} +\!\!+ @title +\!\!+ \texttt{"</bf>\textbackslash n"} +\!\!+ @body.html$$



Universiteit Utrecht

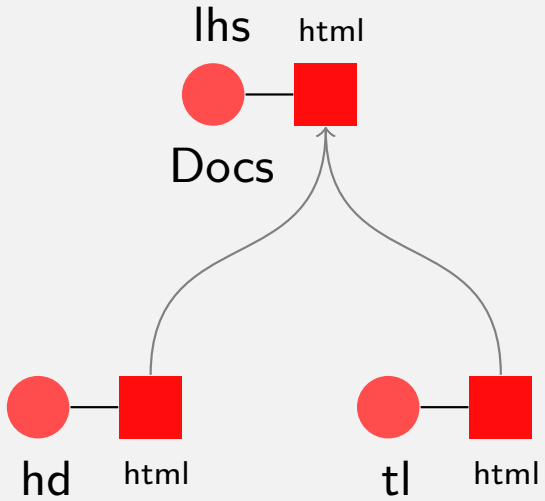
[Faculty of **Science**
Information and Computing Sciences]

A Picture for Cons

$$\textbf{DATA}\ Docs \mid Cons \quad hd : Doc \quad tl : Docs$$

$$\textbf{ATTR}\ Docs\ [||\ html : \{String\}]$$

$$\textbf{SEM}\ Docs$$
$$\quad \mid Cons \quad \textbf{lhs}.html = @hd.html +\!\!+ @tl.html$$



To Summarize: Our First Attribute!

- We introduce an attribute $html$ of type $String$ to return the generated html code in a synthesized attribute:

$$\textbf{ATTR}\ Doc\ Docs\ [\quad |\quad |\quad html : String\quad]$$

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

To Summarize: Our First Attribute!

- We introduce an attribute *html* of type *String* to return the generated html code in a synthesized attribute:

ATTR *Doc Docs* [| | *html* : *String*]

- Definitions for attributes are given in Haskell, with embedded references to attributes, in the form of `@<ntname>.<attrname>`:

```
SEM Doc
  | Section    lhs.html = "<bf>" ++ @title ++ "</bf>\n"
                          ++ @body.html
  | Paragraph  lhs.html = "<P>" ++ @text ++ "</P>"
SEM Docs
  | Cons       lhs.html = @hd.html ++ @tl.html
  | Nil        lhs.html = ""
```

Inherited attributes

Summary: Adding The Level Aspect

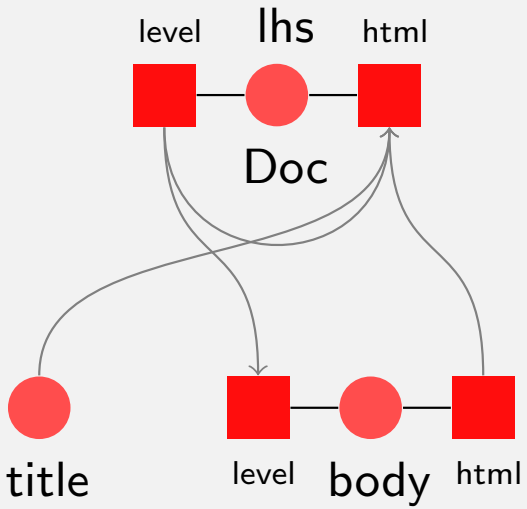
- Introduce an **inherited** attribute with name $level$, indicating the nesting level of the headings:

$$\textbf{ATTR}\ Doc\ Docs\ [\quad level : Int \quad |\quad |\quad]$$

- You can refer to the inherited attributes defined on the left-hand side
- You need to define the inherited attributes of the children

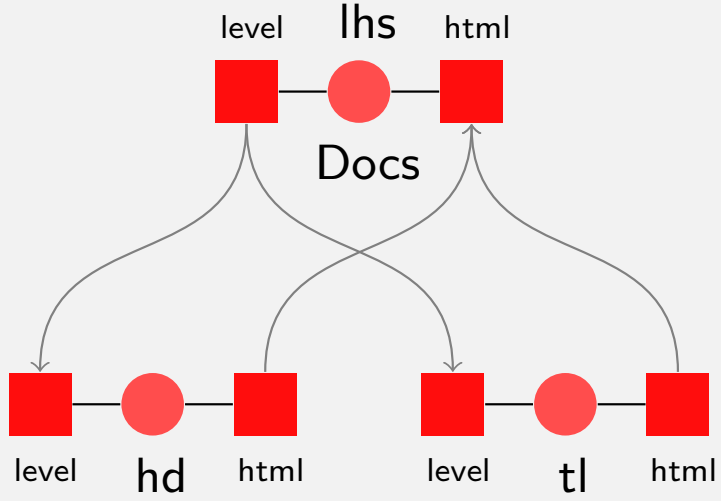
A Picture For Section

$$
\begin{aligned}
&\textbf{SEM}\ \mathit{Doc} \mid \mathit{Section} \\
&\qquad \mathit{body}.\mathit{level} = @\mathbf{lhs}.\mathit{level} + 1 \\
&\qquad \mathbf{lhs}.\mathit{html} \ = \mathit{mk\_tag}\ (\texttt{"H"} +\!\!+ \mathit{show}\ @\mathbf{lhs}.\mathit{level}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \texttt{""}\ @\mathit{title} \\
&\qquad\qquad\qquad\qquad +\!\!+ @\mathit{body}.\mathit{html}
\end{aligned}
$$



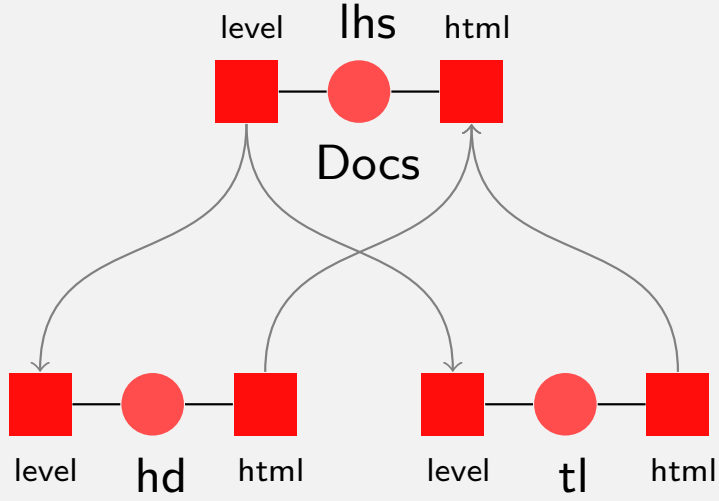
A Picture For Cons

$$\textbf{SEM}\ \mathit{Docs} \mid \mathit{Cons}$$
$$\mathit{hd}.\mathit{level} = @\textbf{lhs}.\mathit{level}$$
$$\mathit{tl}.\mathit{level} = @\textbf{lhs}.\mathit{level}$$



A Picture For Cons

$$\begin{aligned}&\textbf{SEM}\ \mathit{Docs} \mid \mathit{Cons}\\ &\qquad hd.level = @\textbf{lhs}.level\\ &\qquad tl.level = @\textbf{lhs}.level\end{aligned}$$



Actually, these two rules are not needed...

Summary: Adding The Level Aspect

- Introduce an **inherited** attribute with name $level$, indicating the nesting level of the headings:

$$\textbf{ATTR}\ Doc\ Docs\ [\quad level : Int \quad |\quad |\quad]$$

Summary: Adding The Level Aspect

- Introduce an **inherited** attribute with name $level$, indicating the nesting level of the headings:

$$\textbf{ATTR}\ \mathit{Doc}\ \mathit{Docs}\ [\quad \mathit{level} : \mathit{Int} \quad |\quad |\quad]$$

- With the semantic rules:

$$\begin{aligned}\textbf{SEM}\ &\mathit{Doc} \mid \mathit{Section}\\ &\quad \mathit{body}.\mathit{level} = @\textbf{lhs}.\mathit{level} + 1\\ &\quad \textbf{lhs}.\mathit{html} \ = \mathit{mk\_tag}\ (\texttt{"H"} +\!\!+ \mathit{show}\ @\textbf{lhs}.\mathit{level})\\ &\qquad\qquad\qquad\qquad \texttt{""}\ @\mathit{title}\\ &\qquad\qquad\qquad +\!\!+ @\mathit{body}.\mathit{html}\end{aligned}$$

Summary: Adding The Level Aspect

- Introduce an **inherited** attribute with name $level$, indicating the nesting level of the headings:

$$\textbf{ATTR}\ Doc\ Docs\ [\quad level : Int \quad |\quad |\quad]$$

- With the semantic rules:

$$\begin{aligned}\textbf{SEM}\ Doc \mid Section\\ \quad body.level &= @\textbf{lhs}.level + 1\\ \quad \textbf{lhs}.html &= mk\_tag\ (\texttt{"H"} +\!\!+ show\ @\textbf{lhs}.level)\\ &\qquad\qquad \texttt{""}\ @title\\ &\quad +\!\!+ @body.html\end{aligned}$$

- Where the function mk_tag is defined by:

$$\begin{aligned}mk\_tag\ tag\ attrs\ elem = \quad & \texttt{"<"} +\!\!+ tag +\!\!+ attrs +\!\!+ \texttt{">"} +\!\!+ elem\\ & +\!\!+ \texttt{"</"} +\!\!+ tag +\!\!+ \texttt{">"}\end{aligned}$$

Copy rules

Copy rules

- We do not need to give rules for $level$ for Docs?

Copy rules

- We do not need to give rules for $level$ for Docs?
- We generate copy rules in case attributes are passed on unmodified
- The *copy rules* that were automatically generated are:

$$
\begin{array}{l}
\textbf{SEM}\ Docs \\
\quad \mid Cons\ hd.level = @\textbf{lhs}.level \\
\qquad\quad\ \ tl.level\ \ = @\textbf{lhs}.level
\end{array}
$$

- The same process holds for the synthesized attributes, except that if there is more than one child with this synthesized attribute, then the right most child with this attribute is chosen.

Formally: Copy Rules

If a rule for an attribute $k.a$ is missing:

- Use $@\mathbf{loc}.a$
- Use $@c.a$ for the rightmost child c to the left of k, which has a synthesized attribute named a
- Use $@\mathbf{lhs}.a$

Special copy rule: the USE rule

Remember:

$$
\begin{array}{l}
\textbf{SEM}\ \mathit{Docs} \\
\quad \mid \mathit{Cons} \quad \textbf{lhs}.\mathit{html} = @\mathit{hd}.\mathit{html} +\!\!+ @\mathit{tl}.\mathit{html} \\
\quad \mid \mathit{Nil} \quad\ \ \textbf{lhs}.\mathit{html} = \text{""}
\end{array}
$$

These rules cannot be produced by the copy rules. Why not?

Special copy rule: the USE rule

Remember:

```
SEM Docs
  | Cons   lhs.html = @hd.html ++ @tl.html
  | Nil    lhs.html = ""
```

These rules cannot be produced by the copy rules. Why not? But this code can be produced by a special copy rule, for which we need to provide extra information:

```
ATTR Docs [|| html USE {++} {""} : {String}]
```

Chained attributes

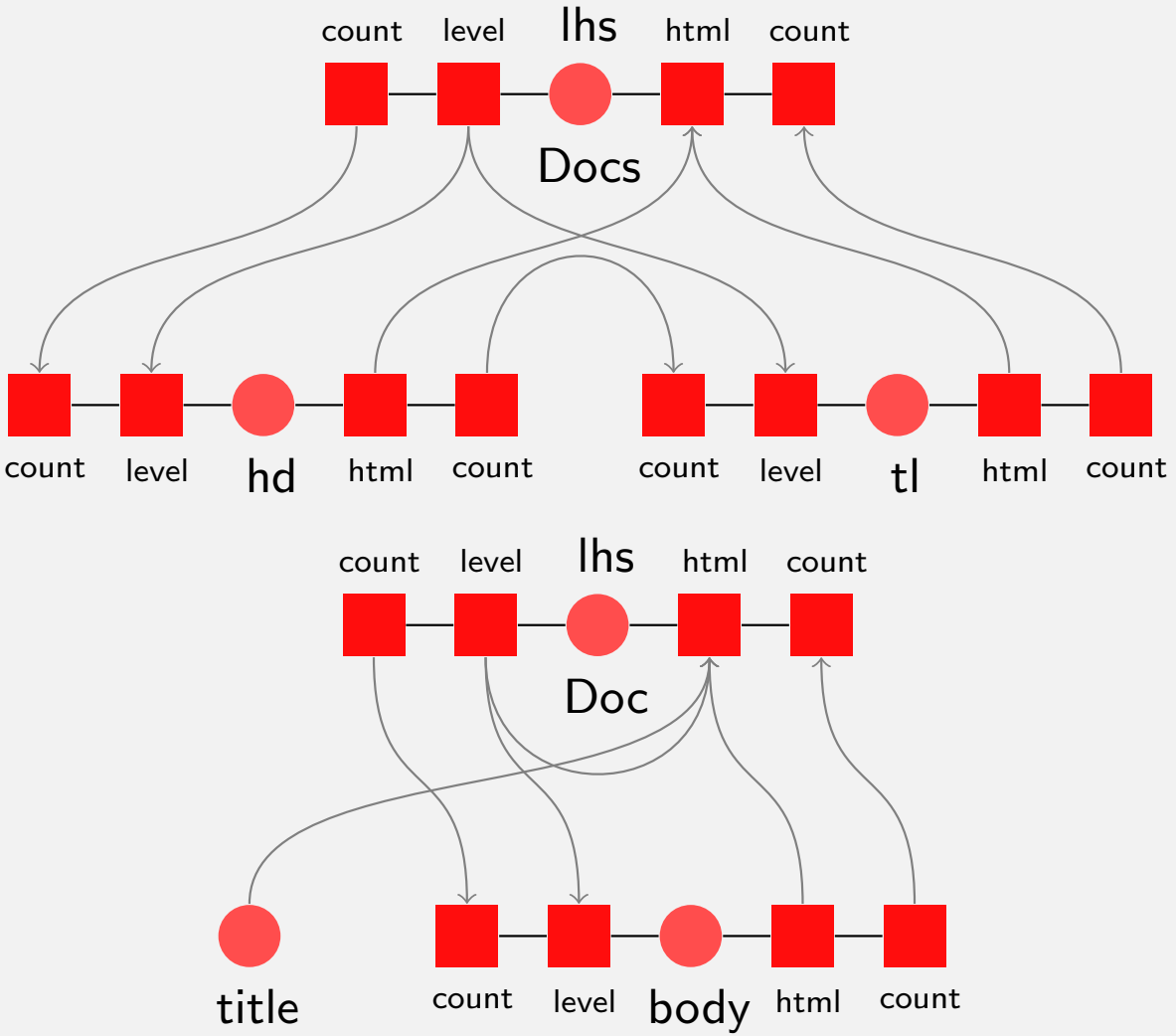
Adding The Section Counter Aspect

- Introduce two inherited attributes:
  - The $context$, representing the outer blocks
  - A $counter$ for keeping track of the number of encountered siblings.

$$\begin{array}{l}\textbf{ATTR}\ Doc\ Docs\ [\ \ context : String \\ \qquad\qquad\qquad\qquad\quad |\ \ count : Int \\ \qquad\qquad\qquad\qquad\quad |\,]\end{array}$$

- Since we do not now whether a Doc will update the counter we will have to pass it from $Docs$ to Doc, and back up again. So $count$ becomes a *threaded attribute*
- **loc** represents a local attribute, which is just a local definition

A picture With The count Added

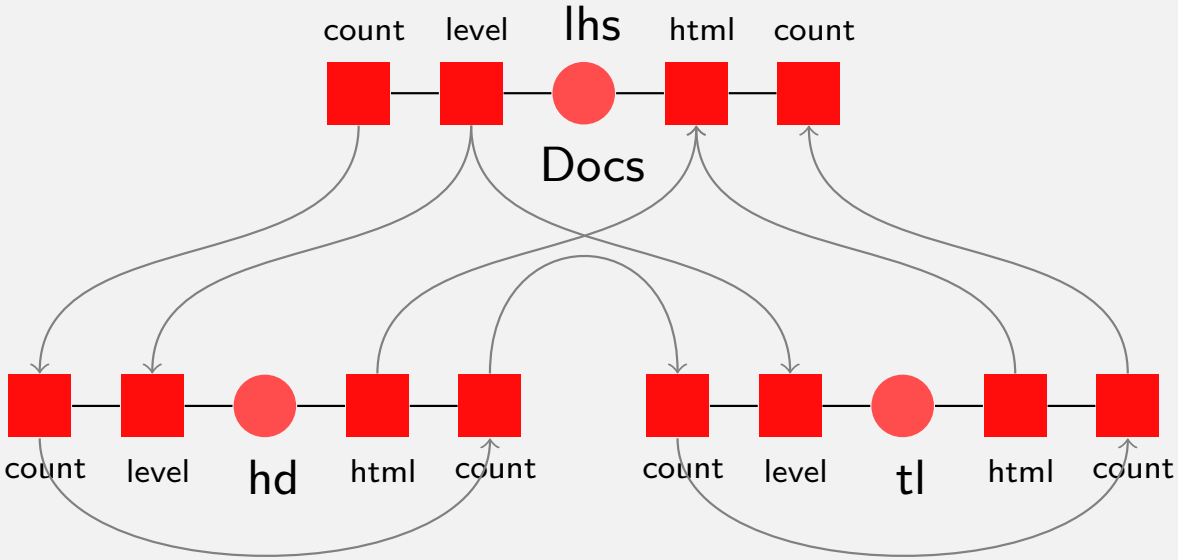


The Semantic Functions

```
SEM Doc
  | Section body.count   = 1
              body.context = @loc.prefix
              lhs.count    = @lhs.count + 1
              lhs.html     = @loc.html
              loc.prefix   = if   null @lhs.context
                             then show @lhs.count
                             else @lhs.context
                                  ++ "."
                                  ++ show @lhs.count
              loc.html     = mk_tag ("H" ++ show @lhs.level)
                                    ""
                                    (@loc.prefix ++ "  "
                                     ++ @title)
                            ++ @body.html
```

A Pictorial Representation

- We show some different aspects
- We show the aspects *count* and *level* and html



Adding Extra Productions

- We may also add extra productions, and as an example we will insert a table of contents

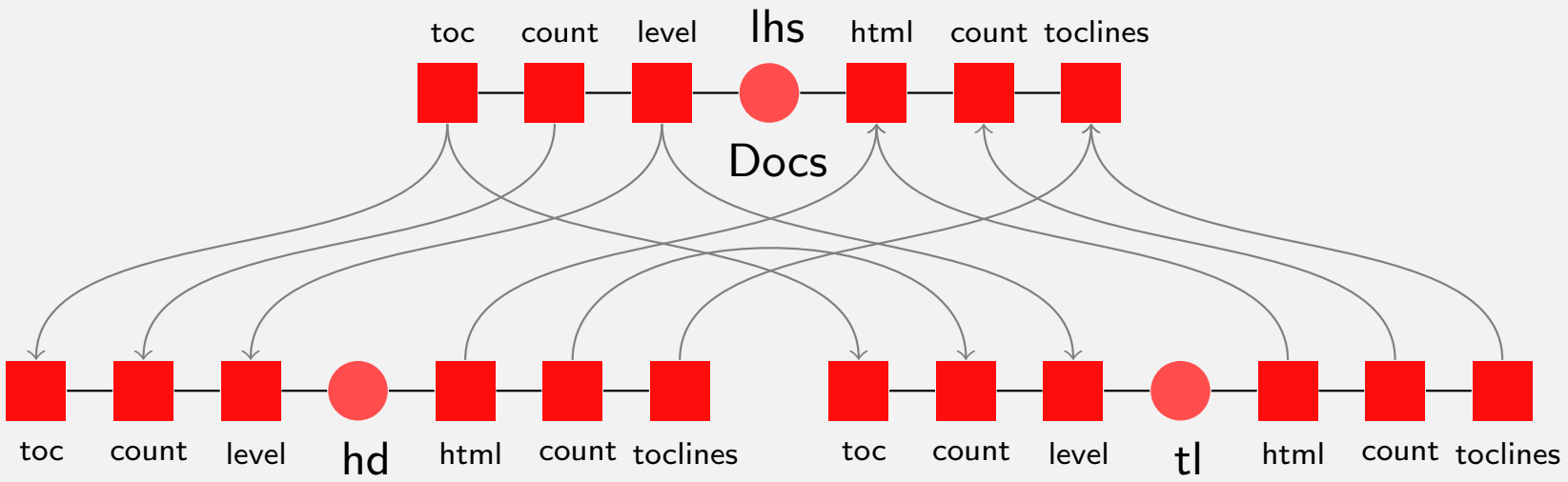
Adding Extra Productions

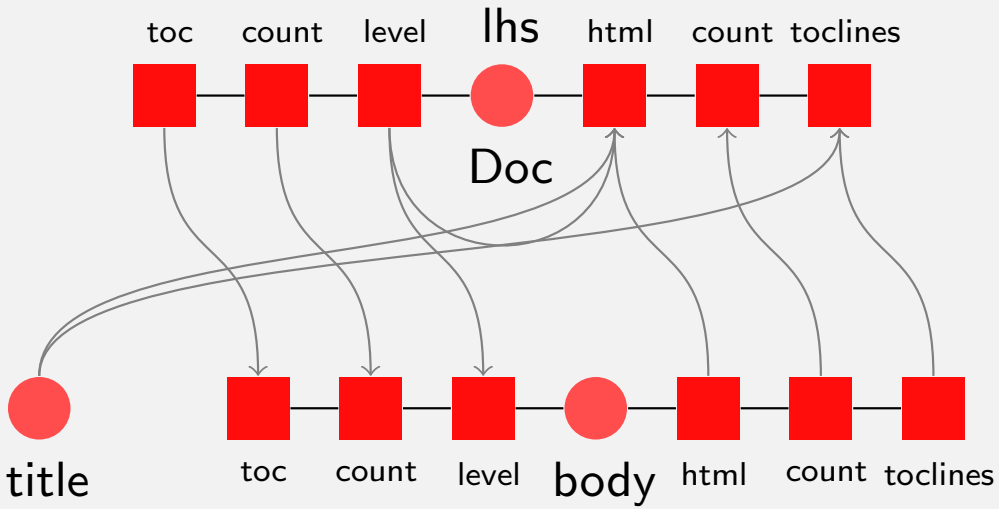
- We may also add extra productions, and as an example we will insert a table of contents
- An extra synthesized attribute $toclines$ in which the table of contents is constructed
- An extra inherited attribute toc, containing the table of contents

$$
\begin{array}{l}
\textbf{DATA}\ Root \quad | \ Root\ doc : Doc \\
\textbf{ATTR}\ Root\ [\,||\ html : String\,] \\
\textbf{DATA}\ Doc \quad | \ Toc \\
\textbf{ATTR}\ Doc\ Docs\ [\ toc : String \\
\qquad\qquad\qquad\qquad\ \ | \\
\qquad\qquad\qquad\qquad\ \ | \ toclines\ USE\ \{++\}\ \{\texttt{""}\} : String\,]
\end{array}
$$

- The USE clause defines default semantic computation

A picture with the toc and toclines added





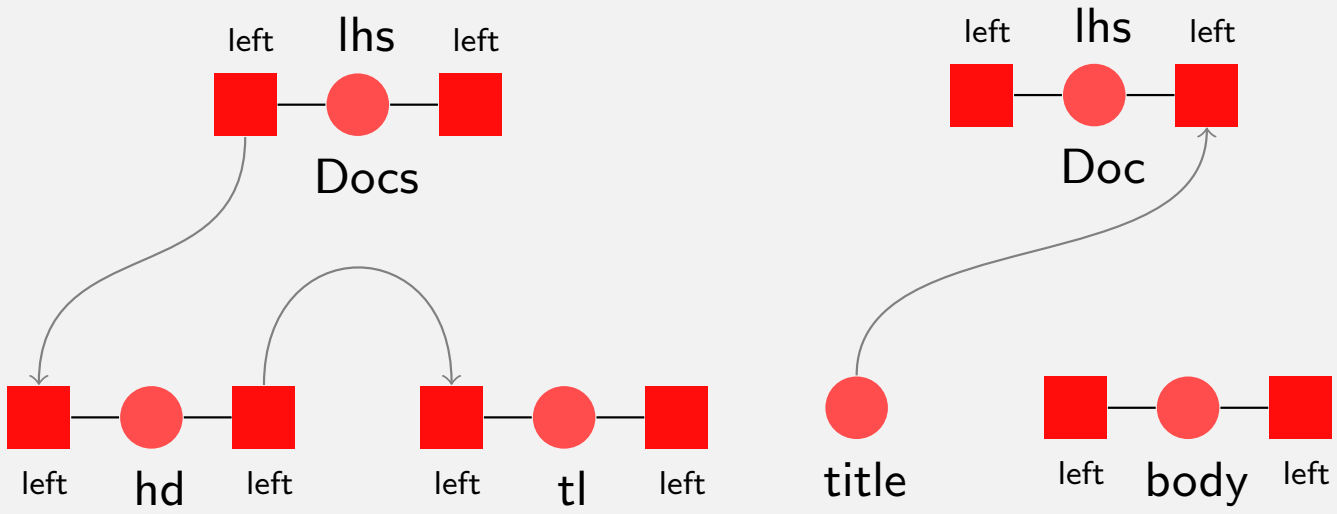
...

```
SEM Doc
  | Section
      lhs .toclines = mk_tag "LI" ""
                      (mk_tag ("A")
                      (" HREF=#" ++ @loc.prefix)
                      (@loc.prefix ++ "  "
                       ++ @title))
                       ++ mk_tag "UL" ""@body.toclines
      lhs .html     := mk_tag "A" (" NAME="
                       ++ @loc.prefix) ""
                       ++ @loc.html
  | Toc lhs .html    = @lhs.toc
SEM Root
  | Root doc.toc     = @doc.toclines
         doc.level   = 1
         doc.context = ""
         doc.count   = 1
```

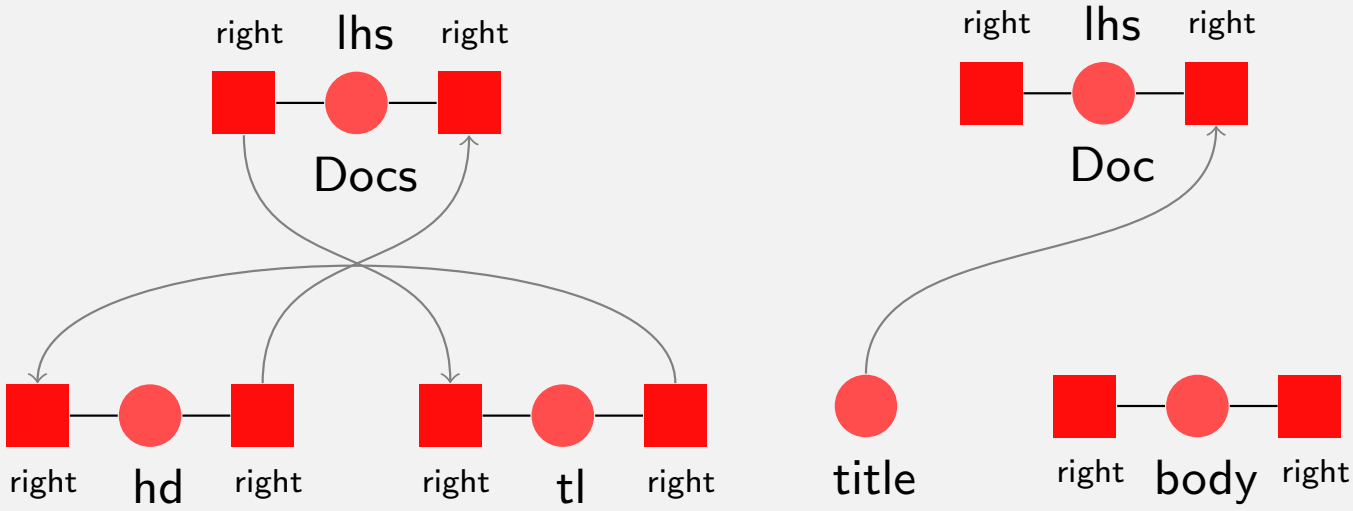
Backward Flow Of Data

- We want to be able to jump to the section to the $left$ and the $right$ of the current section
- We introduce two new attributes for passing this information around

Left


left
lhs
left
Docs
left
hd
left
left
tl
left
left
lhs
left
Doc
title
left
body
left

Right



SEM *Docs*
| *Cons*
$hd.right = @tl.right$
$tl.right = @\mathbf{lhs}.right$
$\mathbf{lhs}.right = @hd.right$

SEM *Doc*
| *Section*
$\mathbf{lhs}.right = @title$
$body.right =$ ""

What Is Generated?

- Data types

$$
\begin{aligned}
\textbf{data}\ Doc = {} & Doc\_Paragraph\ String \\
\mid {} & Doc\_Section\ String\ Docs \\
\mid {} & Doc\_Toc
\end{aligned}
$$

What Is Generated?

- Data types

$$\begin{aligned}\textbf{data}\ Doc = {} & Doc\_Paragraph\ String \\ \mid {} & Doc\_Section\ String\ Docs \\ \mid {} & Doc\_Toc\end{aligned}$$

- Types

$$\begin{aligned}\textbf{type}\ T\_Doc = {} & String \rightarrow \\ & Int \rightarrow \\ & Int \rightarrow \\ & String \rightarrow \\ & (Int, String, String)\end{aligned}$$

... And ...

Semantic functions:

```
sem_Docs_Cons (_hd) (_tl) =
  λ_lhs_context
    _lhs_count
    _lhs_level
    _lhs_toc →
      let (_hd_count, _hd_html, _hd_toclines) =
      (_hd (_lhs_context) (_lhs_count) (_lhs_level) (_lhs_toc))
          (_tl_count, _tl_html, _tl_toclines) =
      (_tl (_lhs_context) (_hd_count) (_lhs_level) (_lhs_toc))
      in  (_tl_count, _hd_html ++ _tl_html
          , _ hd_toclines ++ _tl_toclines)
```

Optimizations

- Perform an abstract interpretation of the grammar
- Computing dependencies between attributes

Optimizations

- Perform an abstract interpretation of the grammar
- Computing dependencies between attributes
- Schedule the attributes for computation per non-terminal (multiple visits)

Optimizations

- Perform an abstract interpretation of the grammar
- Computing dependencies between attributes
- Schedule the attributes for computation per non-terminal (multiple visits)
- And is this way achieve a data-driven evaluation

Optimizations

- Perform an abstract interpretation of the grammar
- Computing dependencies between attributes
- Schedule the attributes for computation per non-terminal (multiple visits)
- And is this way achieve a data-driven evaluation
- That may be somewhat cheaper
- And takes far less space

Conclusions

- Attribute grammars are your friend if you want to implement a language
- Attributes may even depend on themselves if you are building on a lazy language
- Even thinking in terms of attribute grammars may help you

- `http://www.cs.uu.nl/wiki/HUT/WebHome`