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
Universiteit Utrecht

March 1, 2008

Overview

- Historical overview
- Example
- Concepts
- AG by example
- Discussion

Not covered

- First class aspects
- Delegation
- Subtyping-like systems
- Embedding attribute grammars in other languages
- Extending compilers dynamically

Implementing a language

- Parsing is about syntax
- What about semantics?

Historical Overview

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

Historical Overview

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

Historical Overview

- Context-free grammars have limited expressiveness, and thus fail to describe:
  - Scope rules
  - Typing rules
  - Pretty printing
  - Code generation
- Are there extensions?
- Context-sensitive grammars are not very useful, so the idea came up to...

Parameterize Non-Terminal Symbols

Parameterize non-terminal symbols:

- With strings forming part of their name: *2-level grammars* used for the description of Algol 68 (1973)
- With trees: affix grammars

$$
\begin{array}{lll}
S & \to L\langle N\rangle & \\
N & \to 1:N \mid 0 & \\
L\langle N\rangle & \to A\langle N\rangle \quad B\langle N\rangle \quad C\langle N\rangle & \\
A\langle 0\rangle & \to \epsilon \qquad\qquad B\langle 0\rangle \to \epsilon & \ldots \\
A\langle 1:N\rangle & \to a\ A\langle N\rangle \qquad B\langle 1:N\rangle \to b\ B\langle N\rangle & \ldots
\end{array}
$$

Parameterize Non-Terminal Symbols

Parameterize non-terminal symbols:

- With strings forming part of their name: *2-level grammars* used for the description of Algol 68 (1973)
- With trees: affix grammars

$$
\begin{array}{lll}
S & \to L\langle N\rangle & \\
N & \to 1:N \mid 0 & \\
L\langle N\rangle & \to A\langle N\rangle \quad B\langle N\rangle \quad C\langle N\rangle & \\
A\langle 0\rangle & \to \epsilon \qquad\qquad B\langle 0\rangle \quad \to \epsilon & \ldots \\
A\langle 1:N\rangle & \to a\ A\langle N\rangle \qquad B\langle 1:N\rangle \to b\ B\langle N\rangle & \ldots
\end{array}
$$

- With values from some other domain: *attribute grammars* (Knuth)

What Has Been Achieved?

- A lot of research on the efficient evaluation, both in space and time, and
  - So we could write compilers with it that were *almost* as efficient as hand-written compilers

What Has Been Achieved?

- A lot of research on the efficient evaluation, both in space and time, and
  - So we could write compilers with it that were *almost* as efficient as hand-written compilers
  - And so attribute grammars were not used by compiler writers

What Has Been Achieved?

- A lot of research on the efficient evaluation, both in space and time, and
  - So we could write compilers with it that were *almost* as efficient as hand-written compilers
  - And so attribute grammars were not used by compiler writers
  - And other people thought it was something for compiler writers only

What Has Been Achieved?

- A lot of research on the efficient evaluation, both in space and time, and
  - So we could write compilers with it that were *almost* as efficient as hand-written compilers
  - And so attribute grammars were not used by compiler writers
  - And other people thought it was something for compiler writers only
  - And had do do something very complicated with *grammars*

What Has Been Achieved?

- A lot of research on the efficient evaluation, both in space and time, and
  - So we could write compilers with it that were *almost* as efficient as hand-written compilers
  - And so attribute grammars were not used by compiler writers
  - And other people thought it was something for compiler writers only
  - And had do do something very complicated with *grammars*
  - And so they are still largely ignored

What Has Been Achieved?

- A lot of research on the efficient evaluation, both in space and time, and
  - So we could write compilers with it that were *almost* as efficient as hand-written compilers
  - And so attribute grammars were not used by compiler writers
  - And other people thought it was something for compiler writers only
  - And had do do something very complicated with *grammars*
  - And so they are still largely ignored
- We may see attribute grammars however as:
  - A way to do lazy functional programming in an imperative setting
  - An aspect oriented programming language
  - A domain-specific language for writing *catamorphisms*

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

Introducing UUAG

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

This example:

- Attribute values do not influence the parsing process
- Semantic functions for the parser are generated from the attribute 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>
```

Concrete syntax

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

$$
\begin{aligned}
&pDocs :: Parser\ Token\ T\_Docs \\
&pDocs = pFoldr\_gr\ (sem\_Docs\_Cons, sem\_Docs\_Nil)\ pDoc \\
&pDoc :: Parser\ Token\ T\_Doc \\
&pDoc = \\
&\quad sem\_Doc\_Section\ \langle\$\ pKey\ \texttt{"\textbackslash Section"} \\
&\quad\ \langle*\rangle\ pPacked\ (pKey\ \texttt{"\{"})\ (pKey\ \texttt{"\}"})\ pText \\
&\quad\ \langle*\rangle\ pDocs\ \langle*\ pKey\ \texttt{"\textbackslash end"} \\
&\ \langle|\rangle \\
&\quad sem\_Doc\_Paragraph \\
&\quad\ \langle\$\ pKey\ \texttt{"\textbackslash Paragraph"}\ \langle*\rangle\ pText\ \langle*\ pKey\ \texttt{"\textbackslash end"}
\end{aligned}
$$

Abstract syntax

In our example we will use the Utrecht Attribute Grammar System, which borrows heavily from Haskell.

Abstract syntax

In our example we will use the Utrecht Attribute Grammar System, which borrows heavily from Haskell.

- Grammars closely resemble Haskell data types:

$$\begin{array}{lll} \textbf{TYPE}\ \mathit{Docs} = [\mathit{Doc}] & & \\ \textbf{DATA}\ \mathit{Doc} & | \ \mathit{Section} & \mathit{title} : \mathit{String}\ \mathit{body} : \mathit{Docs} \\ & | \ \mathit{Paragraph} & \mathit{text} : \mathit{String} \end{array}$$

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

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]

Our First Attribute!

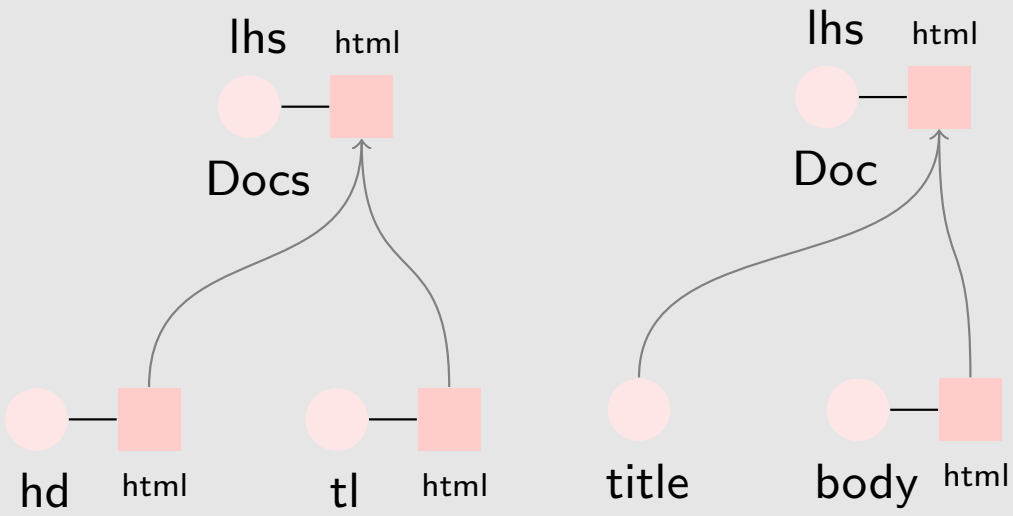
- We introduce an attribute *html* of type *String* to return the generated html code in a synthesized attribute:

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

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

$$\begin{array}{ll}
\textbf{SEM}\ \mathit{Doc} & \\
\quad |\ \mathit{Section} & \textbf{lhs}.\mathit{html} = \texttt{"<bf>"} +\!\!+ @\mathit{title} +\!\!+ \texttt{"</bf>\textbackslash n"} \\
 & \qquad +\!\!+ @\mathit{body}.\mathit{html} \\
\quad |\ \mathit{Paragraph} & \textbf{lhs}.\mathit{html} = \texttt{"<P>"} +\!\!+ @\mathit{text} +\!\!+ \texttt{"</P>"} \\
\textbf{SEM}\ \mathit{Docs} & \\
\quad |\ \mathit{Cons} & \textbf{lhs}.\mathit{html} = @\mathit{hd}.\mathit{html} +\!\!+ @\mathit{tl}.\mathit{html} \\
\quad |\ \mathit{Nil} & \textbf{lhs}.\mathit{html} = \texttt{""}
\end{array}$$

A Picture



Adding The Level Aspect

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

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

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:

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

Adding The Level Aspect

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

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

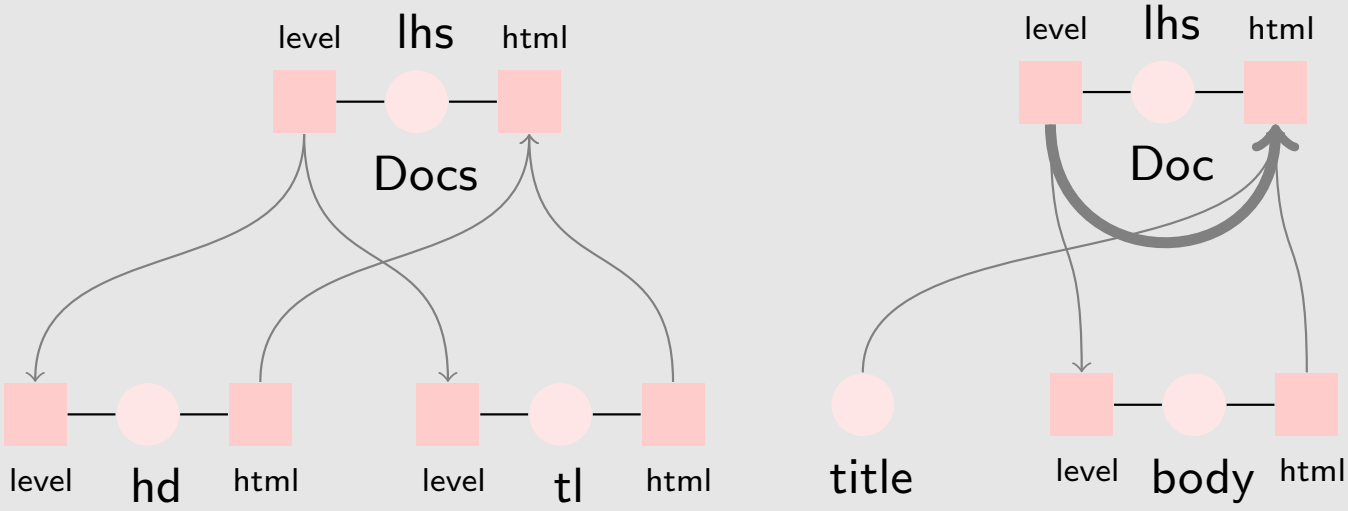
- With the semantic rules:

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

- Where the function mk_tag is defined by:

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

A Picture With level Added



Example

```
Section "Intro"
        [Section "Section 1"
                 [Paragraph "paragraph 1",
                 ,Paragraph "paragraph 2"
                 ]
        ,Section "Section 2"
                 [Paragraph "paragraph 3",
                 ,Paragraph "paragraph 4"
                 ]
        ]
```

Note The Following:

- We did not touch the original code
- We introduced a new inherited attribute with its definitions
- We only redefined the definition of $Doc.section.html$, hence the use of $:=$

Note The Following:

- We did not touch the original code
- We introduced a new inherited attribute with its definitions
- We only redefined the definition of $Doc.section.html$, hence the use of $:=$
- Maybe we also want to add yet another aspect: section counters

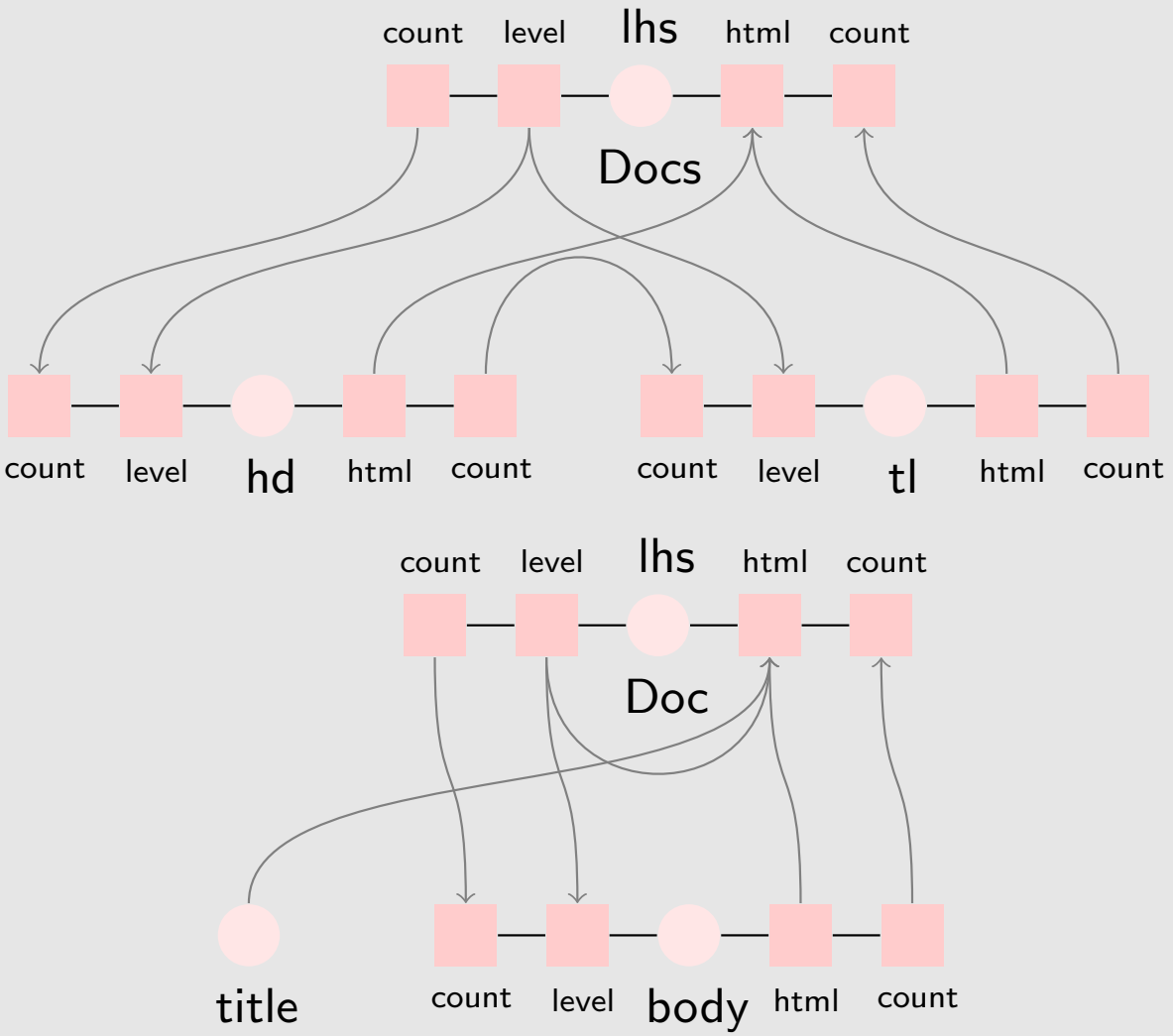
Adding The Section Counter Aspect

- 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** is a virtual non-terminal, with which we may associate local attributes

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
```

I expected more rules. What happened?

- We have not given rules for $count$ and $prefix$ for Docs?

I expected more rules. What happened?

- We have not given rules for *count* and *prefix* for Docs?
- Since we generate copy rules in case attributes are passed on unmodified
- Some *copy rules* that were automatically generated are:

```
SEM Docs
  | Nil   lhs.count   = @lhs.count
  | Cons  hd.count    = @lhs.count
          tl.count    = @hd.count
          hd.context  = @lhs.context
          tl.context  = @lhs.context
          hd.level    = @lhs.level
          tl.level    = @lhs.level
```

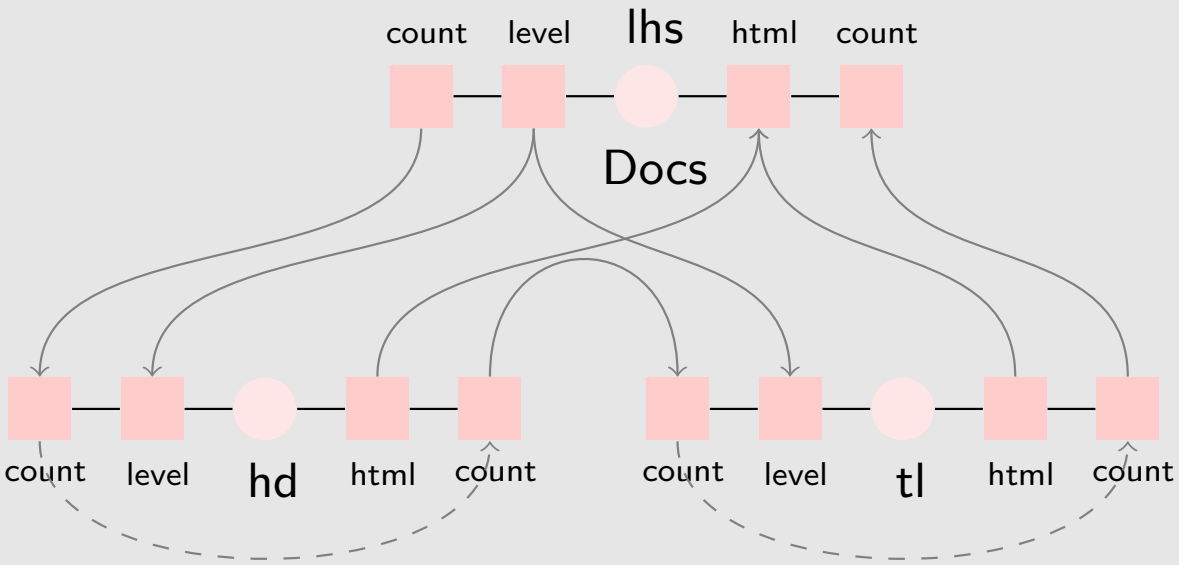
Copy rules

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 $@\mathbf{lhs}.a$

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

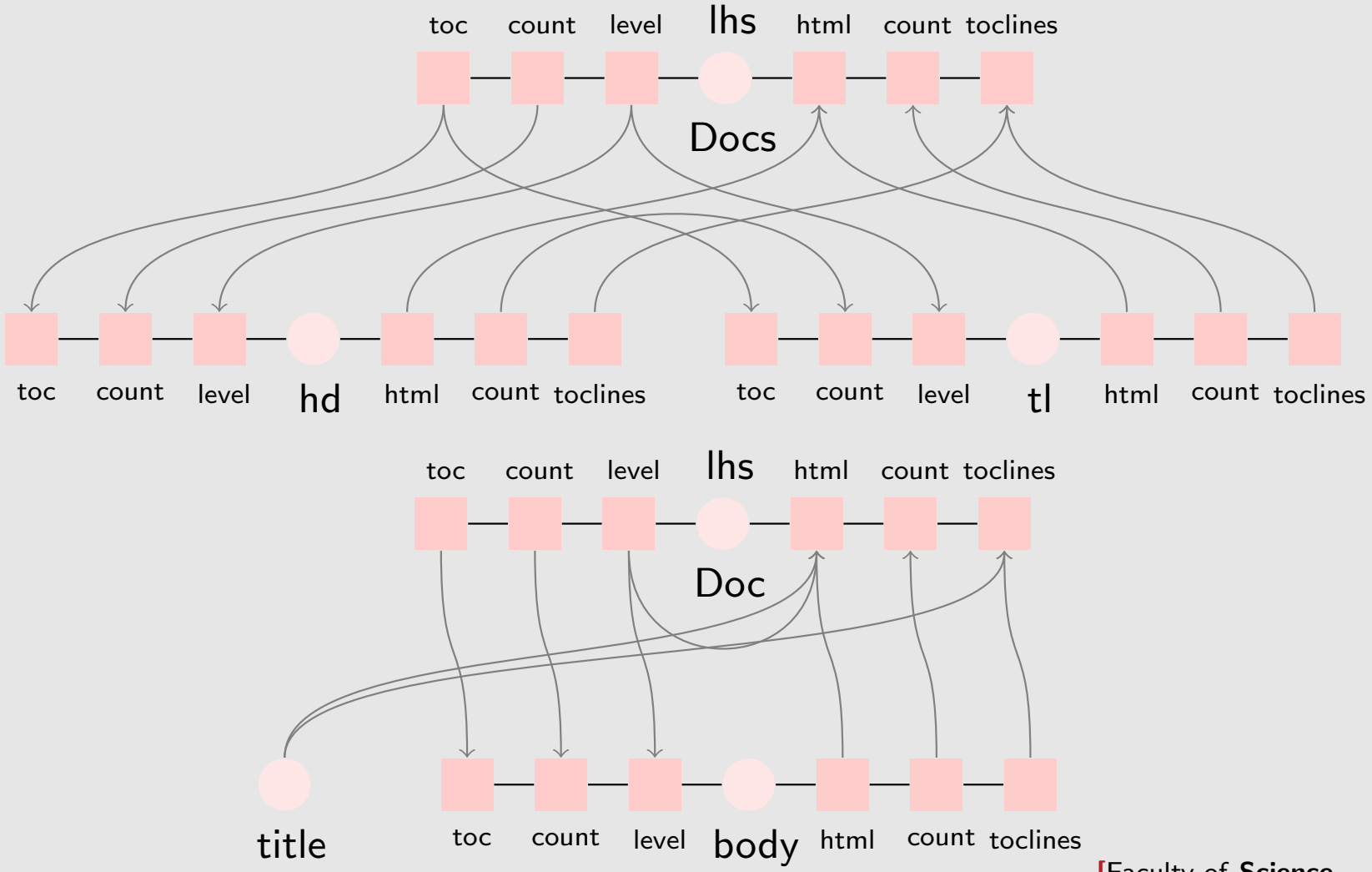
Adding Extra Productions

- 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



Universiteit Utrecht

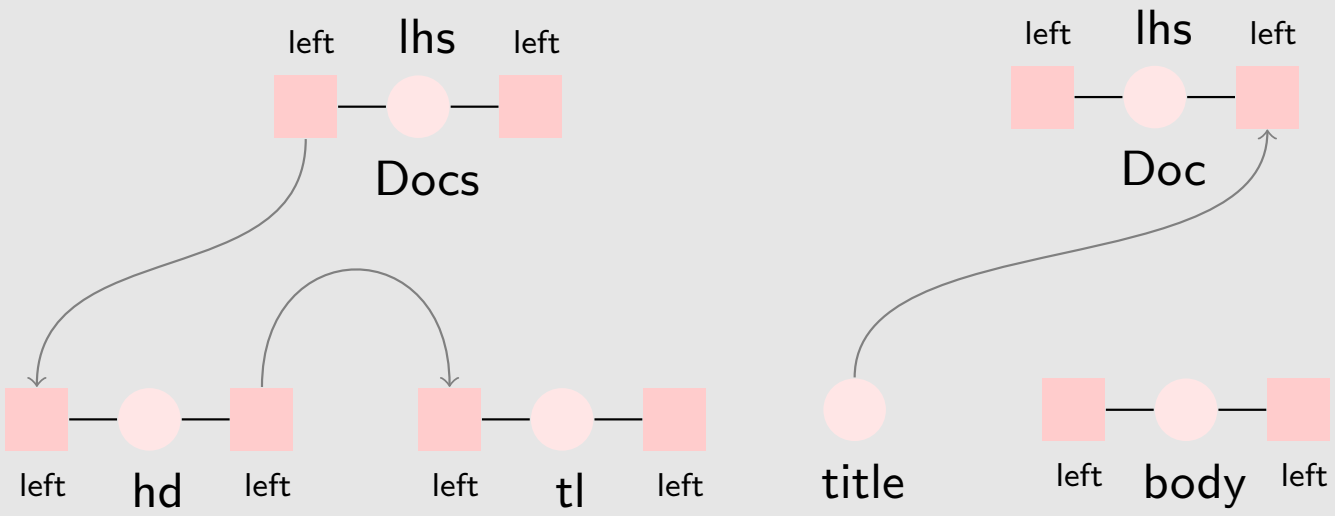
[Faculty of **Science**
Information and Computing Sciences]

```
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
```

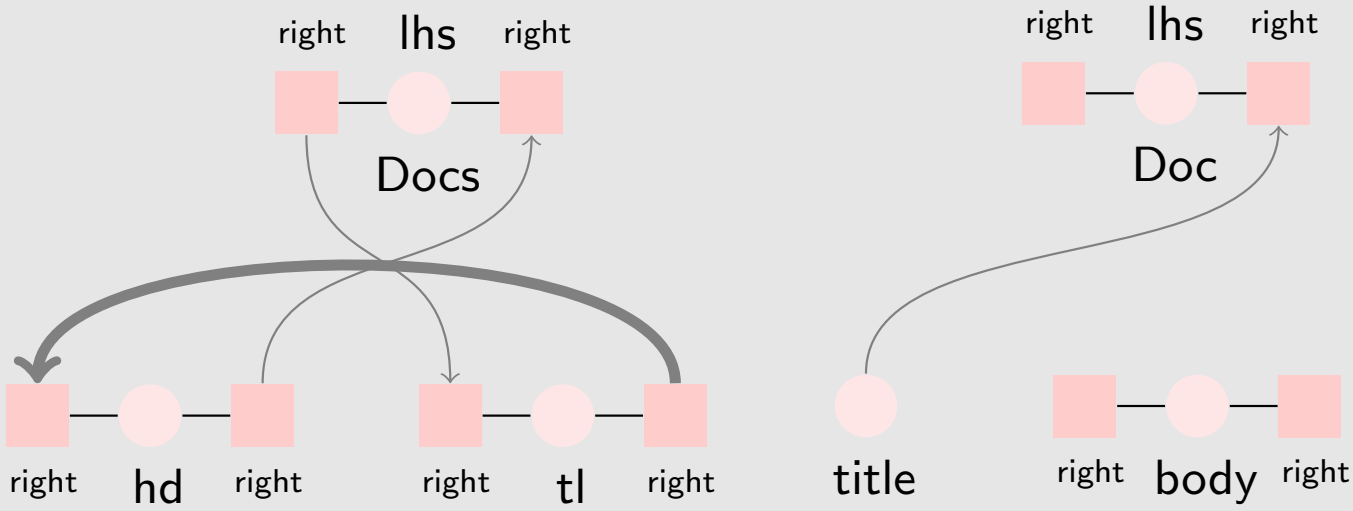
Backward Flow Of Data

- 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*
$\mathbf{lhs}.right = @title$
$body.right =$ ""

Current Situation

- Attribute grammars are a domain specific language for describing *catamorphisms*

Current Situation

- Attribute grammars are a domain specific language for describing *catamorphisms*
- With higher-order domains, i.e. we assign functions mapping *inherited* to *synthesized attributes*
- Different systems differ in what kind of specific patterns of attribution being supported, such as:
  - The copy rules we have seen
  - The USE clause that was used to cmbine attribute values presented by children
  - Some systems allow to refer to *far away* attributes

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

What is generated now?

Type signatures:

$$
\begin{aligned}
&\textbf{type } T\_Doc = Int \rightarrow \\
&\quad String \rightarrow \\
&\quad ([Int]) \rightarrow \\
&\quad (Int, PP\_Doc, String, T\_Doc\_1) \\
&\textbf{type } T\_Doc\_1 = Int \rightarrow \\
&\quad String \rightarrow \\
&\quad PP\_Doc \rightarrow \\
&\quad (PP\_Doc, String)
\end{aligned}
$$

... And ...

Semantic functions:

```
sem_Docs_Cons :: T_Doc → T_Docs → T_Docs
sem_Docs_Cons !hd_ !tl_ =
  (λ(!_lhsIcount) (!_lhsIleft) (!_lhsIprefix) →
    (case (_lhsIprefix) of
      {!_tlOprefix →
       ...
         (case ((hd_ _hdOcount _hdOleft _hdOprefix)) of
           {(!_hdIcount, !_hdIgathToc, !_hdIleft, !hd_1) →
            ...
              (case ((tl_ _tlOcount _tlOleft _tlOprefix)) of
                {(!_tlIgathToc, !tl_1) →
                 ...
                   {(!sem_Docs_1) →
                   (_lhsOgathToc, sem_Docs_1)})
```

... And ...

Semantic functions:

```
sem_Docs_Cons_1 :: String → T_Docs_1 →
   T_Doc_1 → T_Docs_1
sem_Docs_Cons_1 ! _hdIleft ! tl_1 ! hd_1 =
   (λ(!_lhsIlevel) (!_lhsIright) (!_lhsItoc) →
      (case (_lhsItoc) of
         {!_tlOtoc →
         (case (_lhsIlevel) of
          ...
            (case ((tl_1 _tlOlevel _tlOright _tlOtoc)) of
               {(!_tlIcount, !_tlIhtml, !_tlIleft, !_tlIright) →
               (case (_tlIcount) of
                ...
                  (case (_hdIhtml >-< _tlIhtml) of
                   ...
                     (_lhsOcount, _lhsOhtml, _lhsOleft, _lhsOright)})
```

This Is Easy Because..

- We use Haskell as the target language
- And Haskell has lazy/demand driven evaluation

This Is Easy Because..

- We use Haskell as the target language
- And Haskell has lazy/demand driven evaluation
- And so we do not have to schedule the computations ourselves

This Is Easy Because..

- We use Haskell as the target language
- And Haskell has lazy/demand driven evaluation
- And so we do not have to schedule the computations ourselves
- Furthermore we borrow:
  - The type system from Haskell
  - The language for defining the semantic functions

Why Are Attribute Grammars Nice?

If we look at functional languages we see that:

Why Are Attribute Grammars Nice?

If we look at functional languages we see that:

- It is easy to define a new function that computes a property of a data type: define an alternative for each alternative of the data type.

Why Are Attribute Grammars Nice?

If we look at functional languages we see that:

- It is easy to define a new function that computes a property of a data type: define an alternative for each alternative of the data type.
- It is difficult to add a new alternative to a data type, since we have to update all functions so it deals with this extra alternative

Why Are Attribute Grammars Nice?

If we look at functional languages we see that:

- It is easy to define a new function that computes a property of a data type: define an alternative for each alternative of the data type.
- It is difficult to add a new alternative to a data type, since we have to update all functions so it deals with this extra alternative

If we look at object oriented languages we see that:

- It is easy to define a subclass: simply provide a method contributing its part for each property we are interested in.

Why Are Attribute Grammars Nice?

If we look at functional languages we see that:

- It is easy to define a new function that computes a property of a data type: define an alternative for each alternative of the data type.
- It is difficult to add a new alternative to a data type, since we have to update all functions so it deals with this extra alternative

If we look at object oriented languages we see that:

- It is easy to define a subclass: simply provide a method contributing its part for each property we are interested in.
- It is difficult to add a property to a data type, since we have to update all subclasses with a new method

We Do Not Have To choose...

- Attribute grammars do not force you to think along either axis
- You may "grow" a system by:

We Do Not Have To choose...

- Attribute grammars do not force you to think along either axis
- You may "grow" a system by:
  - Stepwise adding extra productions to data types
  - Stepwise adding extra attributes

We Do Not Have To choose...

- Attribute grammars do not force you to think along either axis
- You may "grow" a system by:
  - Stepwise adding extra productions to data types
  - Stepwise adding extra attributes
- Aspects are largely independent
- But interactions can take place by just referring to other aspects
- The system will weave things together

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 you my construct interesting programs

- `http://www.cs.uu.nl/wiki/HUT/WebHome`
- or check your CD