



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)

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



# 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

<code>\section{Intro}</code>	<code>&lt;h1&gt;Intro&lt;/h1&gt;</code>
<code>  \section{Section 1}</code>	<code>&lt;h2&gt;Section 1&lt;/h2&gt;</code>
<code>    \paragraph</code>	<code>&lt;p&gt;</code>
<code>      paragraph 1</code>	<code>  Paragraph 1</code>
<code>    \end</code>	<code>&lt;/p&gt;</code>
<code>    \paragraph</code>	<code>&lt;p&gt;</code>
<code>      paragraph 2</code>	<code>  Paragraph 2</code>
<code>    \end</code>	<code>&lt;/p&gt;</code>
<code>  \end</code>	
<code>\section{Section 2}</code>	<code>&lt;h2&gt;Section 2&lt;/h2&gt;</code>
<code>  \paragraph</code>	<code>&lt;p&gt;</code>
<code>    paragraph 1</code>	<code>  Paragraph 1</code>
<code>  \end</code>	<code>&lt;/p&gt;</code>
<code>  \paragraph</code>	<code>&lt;p&gt;</code>
<code>    paragraph 2</code>	<code>  Paragraph 2</code>
<code>  \end</code>	<code>&lt;/p&gt;</code>
<code>\end \end</code>	



# 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*:

<b>DATA</b> <i>Doc</i>		<i>Section</i> <i>title</i> : <i>String</i> <i>body</i> : <i>Docs</i>
		<i>Paragraph</i> <i>text</i> : <i>String</i>

<b>DATA</b> <i>Docs</i>		<i>Cons</i> <i>hd</i> : <i>Doc</i> <i>tl</i> : <i>Docs</i>
		<i>Nil</i>

- ▶ *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**:

<b>ATTR</b> <i>Doc Docs</i> [     <i>html : String</i> ]
--





# 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*.



# 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



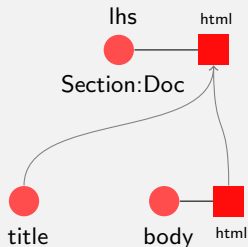
# A Picture for Section

**DATA** *Doc* | *Section* *title* : { *String* } *body* : *Docs*

**ATTR** *Doc* [|| *html* : { *String* }]

**SEM** *Doc*

| *Section* *lhs.html* = "<bf>" ++ @*title* ++ "</bf>\n"  
++ @*body.html*



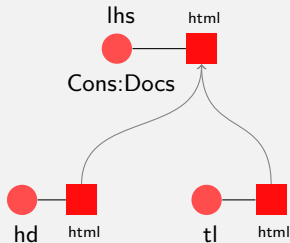
# A Picture for Cons

**DATA** Docs | *Cons*  $hd : Doc$   $tl : Docs$

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

**SEM** Docs

| *Cons*  $\text{lhs.html} = @hd.html \mathbin{++} @tl.html$



# To Summarize: Our First Attribute!

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

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



## 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 = ""
```



Universiteit Utrecht

[Faculty of Science  
Information and Computing Sciences]



13

- |  |
|--|
| <b>ATTR</b> <i>Doc Docs</i> [     <i>html : String</i> ] |
|--|

- SEM Doc

*Paragraph* `lhs.html = "<P>" ++ @text ++ "</P>"`

**Cons**  $\text{lhs.html} = @hd.html \mathbin{++} @tl.html$

```
| Nil      lhs.html = ""
```

# Inherited attributes



# Summary: Adding The Level Aspect

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

```
ATTR Doc Docs [ level : Int | | ]
```

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



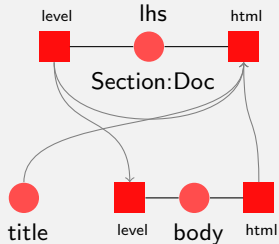


# A Picture For Section

**SEM** *Doc* | *Section*

$body.level = @lhs.level + 1$

$lhs.html = mk\_tag("H" ++ show @lhs.level)$   
           $"" @title$   
           $++ @body.html$

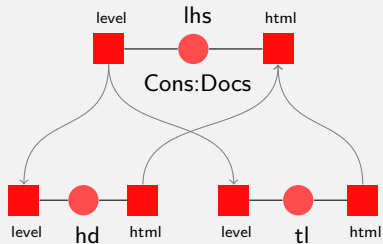


# A Picture For Cons

SEM Docs | *Cons*

$hd.level = @lhs.level$

$tl.level = @lhs.level$

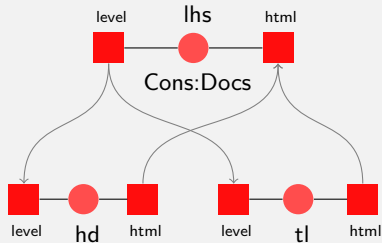


# A Picture For Cons

SEM Docs | *Cons*

$hd.level = @lhs.level$

$tl.level = @lhs.level$



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:

```
ATTR Doc Docs [ level : Int | | ]
```



# Summary: Adding The Level Aspect

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

```
ATTR Doc Docs [ level : Int | | ]
```

- ▶ With the semantic rules:

```
SEM Doc | Section  
    body.level = @lhs.level + 1  
    lhs.html   = mk_tag ("H" ++ show @lhs.level)  
                "" @title  
                ++ @body.html
```



# Summary: Adding The Level Aspect

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

```
ATTR Doc Docs [ level : Int | | ]
```

- ▶ With the semantic rules:

```
SEM Doc | Section  
    body.level = @lhs.level + 1  
    lhs.html   = mk_tag ("H" ++ show @lhs.level)  
                "" @title  
                ++ @body.html
```

- ▶ Where the function *mk\_tag* is defined by:

```
mk_tag tag attrs elem = "<" ++ tag ++ attrs ++ ">" ++ elem  
                        ++ "</" ++ tag ++ ">"
```



# 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:

**SEM Docs**

| *Cons*  $hd.level = @lhs.level$   
 $tl.level = @lhs.level$

- ▶ 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 **@loc.** $a$
- ▶ Use **@c.** $a$  for the rightmost child  $c$  to the left of  $k$ , which has a synthesized attribute named  $a$
- ▶ Use **@lhs.** $a$



# Special copy rule: the USE rule

Remember:

**SEM** *Docs*

| *Cons*  $\text{lhs.html} = @hd.html \uparrow @tl.html$

| *Nil*  $\text{lhs.html} = ""$

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



# Special copy rule: the USE rule

Remember:

**SEM** *Docs*

| *Cons*    $\text{lhs.html} = @hd.html \text{ ++ } @tl.html$

| *Nil*    $\text{lhs.html} = ""$

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*  $[|| \text{html USE } \{++\} \{""\} : \{String\}]$



# Chained attributes



# Adding The Section Counter Aspect

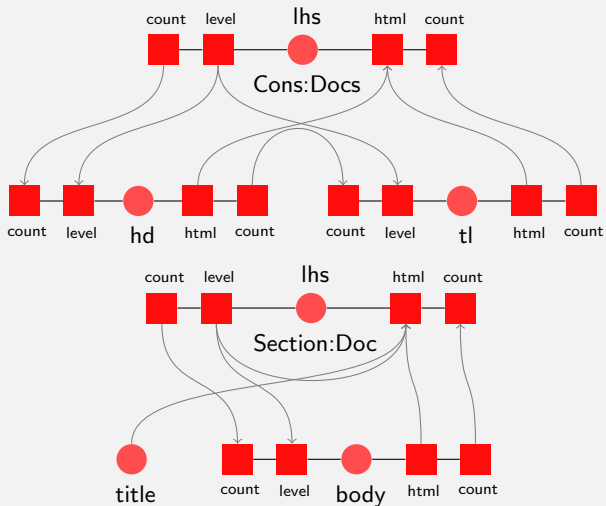
- ▶ Introduce two inherited attributes:
  - ▶ The *context*, representing the outer blocks
  - ▶ A *counter* for keeping track of the number of encountered siblings.

```
ATTR Doc Docs [ context : String  
                | count : Int  
                |]
```

- ▶ 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*

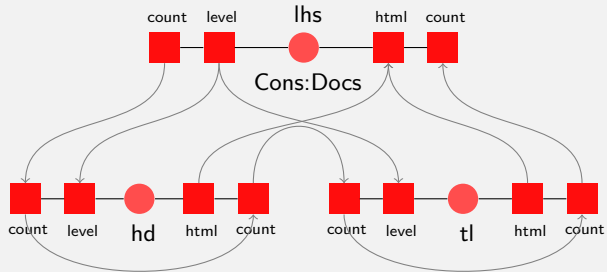
```
| Section body.count = 1  
    body.context = @loc.prefix  
    lhs.count    = @lhs.count + 1  
    lhs.html     = @loc.html  
    loc.prefix   = if    null @lhs.context  
                    then show @lhs.count  
                    else @lhs.context  
                      ++ "."  
                      ++ 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

```
DATA Root | Root doc : Doc
```

```
ATTR Root [| | html : String]
```

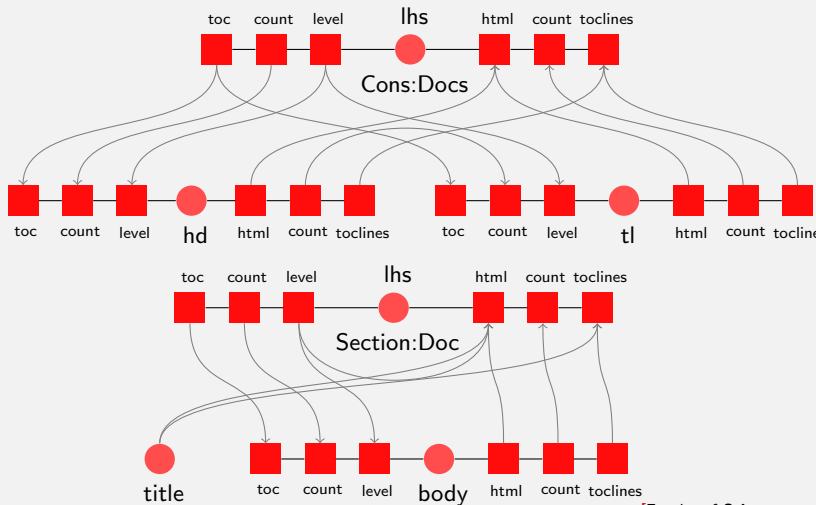
```
DATA Doc | Toc
```

```
ATTR Doc Docs [ toc : String  
                  |  
                  | toclines USE { ++ } { "" } : String]
```

- ▶ 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

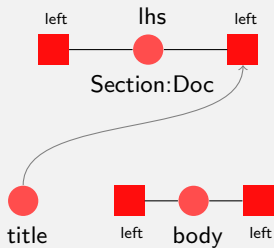
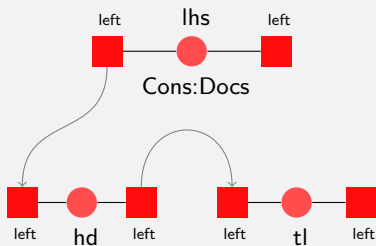
```
| Root doc.toc      = @doc.toclines
  doc.level         = 1
  doc.context       = ""
  doc.count         = 1
```

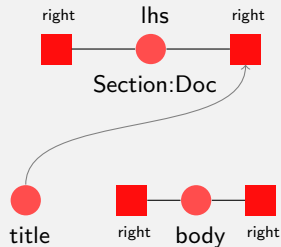
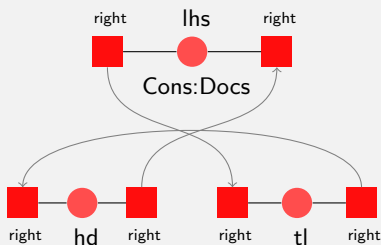


# Backward Flow Of Data

- ▶ 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







**SEM Docs**

| *Cons*

$hd.right = @tl.right$   
 $tl.right = @lhs.right$   
 $lhs.right = @hd.right$

**SEM Doc**

| *Section*

$lhs.right = @title$   
 $body.right = ""$





# What Is Generated?

## ► Data types

```
data Doc = Doc_Paragraph String  
         | Doc_Section String Docs  
         | Doc_Toc
```



# What Is Generated?

## ► Data types

```
data Doc = Doc_Paragraph String  
         | Doc_Section String Docs  
         | Doc_Toc
```

## ► Types

```
type T_Doc = String →  
          Int    →  
          Int    →  
          String →  
          (Int, String, String)
```



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>

