Attribute grammars
A short tutorial

SD Swierstra
Dept of Computer Science
Utrecht University

Overview

- Historical Overview
- Example
- Discussion of aspect orientation
- Designing programs by drawing pictures

Not covered

- first class aspects
- delegation
- subtyping-like systems
- embedding attribute grammars in other languages
- extending compilers dynamically
- ...

Historical Overview

- Context-free grammars have limited expressiveness. Things we cannot express are:

Historical Overview

- Context-free grammars have limited expressiveness. Things we cannot express are:
  - scope rules
  - typing rules
  - pretty printing
  - code generation
  - incremental language editors (Synthesizer Generator, Reps/Teitelbaum)
  - ...

Historical Overview

- Context-free grammars have limited expressiveness. Things we cannot express are:
  - scope rules
  - typing rules
  - pretty printing
  - code generation
  - incremental language editors (Synthesizer Generator, Reps/Teitelbaum)
  - ...
- and so one started to look for extensions
- context-sensitive grammars are not very useful, so the idea came up to:

Parameterize Non-Terminal Symbols

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

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

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
- 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 efficiënt 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 efficiënt 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 efficiënt 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 efficiënt 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 to 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 efficiënt 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 to 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 efficiënt 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 to do something very complicated with *grammars*
  - and so they are still largely ignored
- We currently see attribute grammars as:
  - a way to do lazy functional programming in an imperative setting
  - an aspect oriented pogramming language
  - a domain-specific language for writing *catamorphisms (folds)*

An attribute grammar consists of:

- a underlying context free grammar

An attribute grammar consists of:

- a underlying context free grammar
- a description of which nonterminals 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:

- a underlying context free grammar
- a description of which nonterminals 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

Where do attribute grammar systems differ?

- either completely
  - *self-supporting*, with a special language for describing the semantic functions
  - or by leaning on some well-known *host language* (Pascal/C/Haskell/ML/Java)

Where do attribute grammar systems differ?

- either completely
  - *self-supporting*, with a special language for describing the semantic functions
  - or by leaning on some well-known *host language* (Pascal/C/Haskell/ML/Java)
- special syntax and conventions for describing common attribution patterns

Where do attribute grammar systems differ?

- either completely
  - *self-supporting*, with a special language for describing the semantic functions
  - or by leaning on some well-known *host language* (Pascal/C/Haskell/ML/Java)
- special syntax and conventions for describing common attribution patterns
- restrictions on the allowed dependencies between attributes, such as:
  - can be evaluated in a *sigle pass from left to right* over the abstract syntax tree (the Java visitor pattern),

Where do attribute grammar systems differ?

- either completely
  - *self-supporting*, with a special language for describing the semantic functions
  - or by leaning on some well-known *host language* (Pascal/C/Haskell/ML/Java)
- special syntax and conventions for describing common attribution patterns
- restrictions on the allowed dependencies between attributes, such as:
  - can be evaluated in a *sigle pass from left to right* over the abstract syntax tree (the Java visitor pattern),
    so it can be nicely combined with a recursive descent parser
  - can be evaluated in n (alternating) passes
  - for each non-terminal a fixed order in which attributes can be evaluated can be found (*ordered attribute grammars* (Kastens))

An Example: Creating HTML using UAG

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

An Example: Creating HTML using UAG

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

- Grammars closely resemble Haskell data types:

$$
\begin{array}{l}
\textbf{type}\ Docs = [Doc] \\
\textbf{data}\ Doc\ \ |Section\ title\ \ \ : String\ body : Docs \\
\qquad\qquad\ \ |Para\ \ \ \ string : String
\end{array}
$$

- $Docs$ and Doc are non-terminals (classes if you are an OO person)
- $Section$ and $Para$ label different alternatives/productions
- $title$, $body$ and $string$ are names for children

Our first attribute!

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

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

Our first attribute!

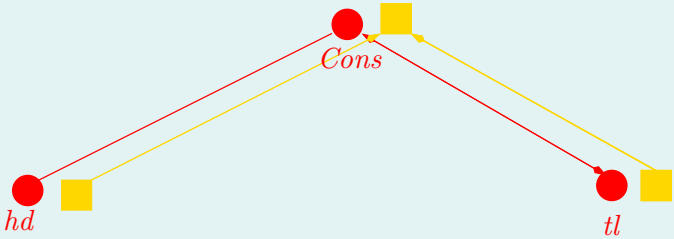
- We introduce an attribute *hmtl* of type *String* to return the generated html code in a synthesized attribute:

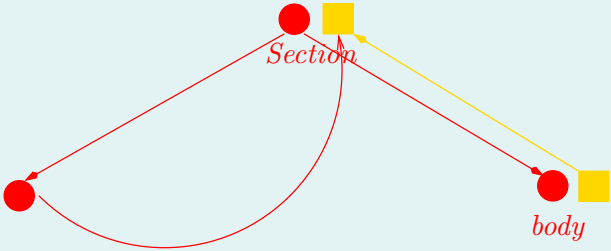
```
attr Doc Docs [||html : String]
```

- 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
  |Para lhs.html     = "<P>"  ++ @string ++ "</P>\n"
sem Docs
  |Cons lhs.html = @hd.html ++ @tl.html
  |Nil  lhs.html = ""
```

A Pictorial Representation





Example translation

The not very interesting input:

```
test = Section "Intro"
               [Section "Section"
                        [Para "paragraph",
                        ,Para "paragraph"
                        ]
               ,Section "Section"
                        [Para "paragraph",
                        ,Para "paragraph"
                        ]
               ]
```

...

gives the not very exciting html-code:

```
<P><B>Intro</B></P>
<P><B>Section</B></P>
<P>paragraph</P>
<P>paragraph</P>
<P><B>Section</B></P>
<P>paragraph</P>
<P>paragraph</P>'
```

...

gives the not very exciting html-code:

```
<P><B>Intro</B></P>
<P><B>Section</B></P>
<P>paragraph</P>
<P>paragraph</P>
<P><B>Section</B></P>
<P>paragraph</P>
<P>paragraph</P>'
```

Maybe we should use the different heading facilities available in HTML using the $<H_>$ tags.

Adding the Level aspect

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

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

Adding the Level aspect

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

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

- with the semantic rules:

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

Adding the Level aspect

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

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

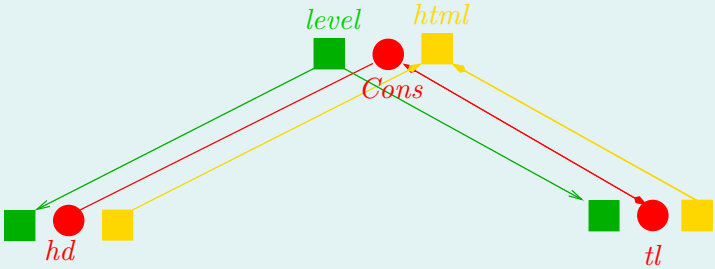
- with the semantic rules:

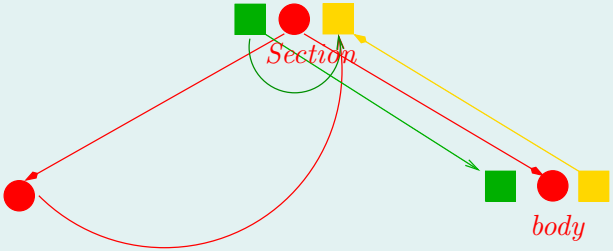
$$\begin{aligned}\textbf{sem}\ Doc|Section\ body.level &= @\textbf{lhs}.level + 1\\ \textbf{lhs}.html &:= mk_tag\ (\texttt{"H"} +\!\!+ show\ @\textbf{lhs}.level)\\ &\qquad \texttt{""}\\ &\qquad @title\\ &\quad +\!\!+ @body.html\end{aligned}$$

- where the function *mk_tag* is defined by:

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

A picture with level added





with as result:

Intro

Section

paragraph

paragraph

Section

paragraph

paragraph

Coming from the code:

```
<H1>Intro</H1>
<H2>Section</H2>
<P>paragraph</P>
<P>paragraph</P>
<H2>Section</H2>
<P>paragraph</P>
<P>paragraph</P>'
```

Note the following:

- we did not touch the original code
- we introduced an 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 an 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

Introducing Section Counters

Wouldn't it be even nicer if we had section numbering?

1 Intro

1.1 Section

paragraph

paragraph

1.2 Section

paragraph

paragraph

... from the following HTML

```
<H1>1 Intro</H1>
<H2>1.1 Section</H2>
<P>paragraph</P>
<P>paragraph</P>
<H2>1.2 Section</H2>
<P>paragraph</P>
<P>paragraph</P>
```

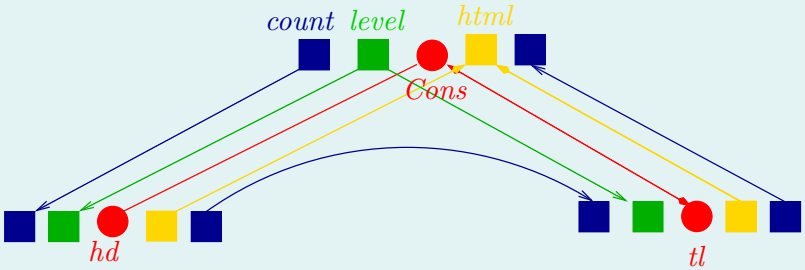
Adding the Section Counter Aspect

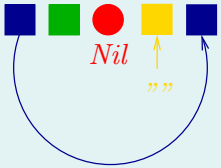
- introduce two inherited attributes:
  - the *context*, representing the counter state of the outer blocks
  - a *counter* for keeping track of the number of encountered siblings.

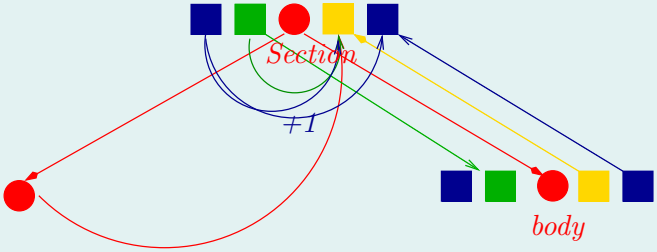
$$
\begin{array}{ll}
\textbf{attr}\ Doc\ Docs & [\, context : String \\
 & |\, count \quad : Int \\
 & | \\
 &]
\end{array}
$$

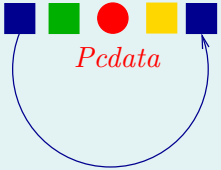
- since we do not now whether a *Doc* will update the counter we will have to pass it from *Docs* to *Doc*, and back up again. So *count* becomes a *threaded attribute*
- **loc** is a virtual non-terminal, with which we may associate local attributes

A picture with the count added









The semantic functions

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

I expected more rules. What happened?

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

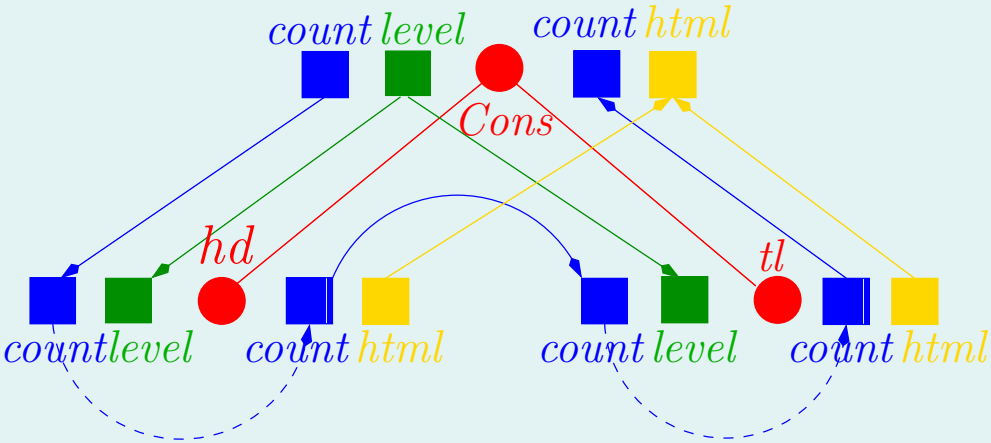
I expected more rules. What happened?

- 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:

$$
\begin{array}{lll}
\textbf{sem}\ Docs & & \\
|Nil & \textbf{lhs}.count & = @\textbf{lhs}.count \\
|Cons & hd\ .count & = @\textbf{lhs}.count \\
 & tl\ \ .count & = @hd.count \\
 & hd\ .context & = @\textbf{lhs}.context \\
 & tl\ \ .context & = @\textbf{lhs}.context \\
 & hd\ .level & = @\textbf{lhs}.level \\
 & tl\ \ .level & = @\textbf{lhs}.level
\end{array}
$$

Data Flow at the Cons Nodes

- we show some different aspects
- and use different colors for different aspects
- we show the aspects *count* and *level* and html



Adding extra alternatives

We may also add new alternatives, and as an example we will insert a table of contents

Adding extra alternatives

We may also add new alternatives, and as an example we will insert a table of contents

- an extra synthesized attribute *toclines* in which the table of contents is constructed

$$\textbf{attr}\ Doc\ Docs\ [||toclines\ USE\{+\!\!+\}\{\texttt{""}\} : String]$$

- The *USE* construct collects all the *toclines* from the children that provide one

Passing the Table of Contents to all the Nodes

We add an extra inherited attribute toc, containing the table of contents:

$$\textbf{attr}\ Doc\ Docs\ [toc : String|||]$$

Passing the Table of Contents to all the Nodes

We add an extra inherited attribute toc, containing the table of contents:

$$\textbf{attr}\ Doc\ Docs\ [toc : String | |]$$

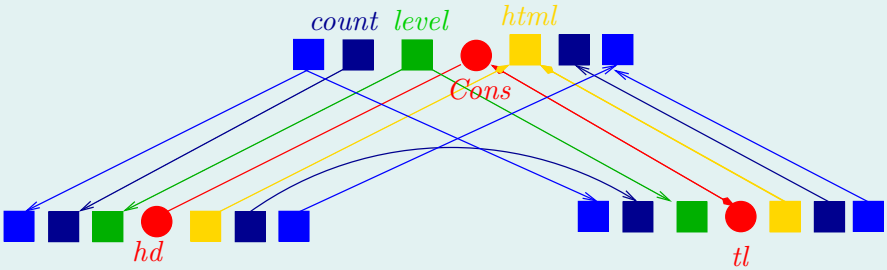
And a $Root$ non-terminal, for feeding back the Toc into the tree

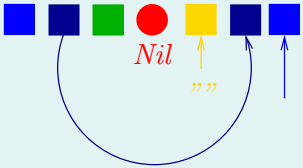
$$\textbf{data}\ Root | Root\ doc : Doc$$
$$\textbf{attr}\ Root\ [\ | html : String]$$
$$\textbf{sem}\ Root | Root\ doc.toc = @doc.toclines$$

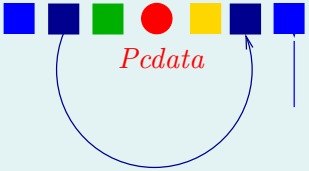
Locating the table of Contents

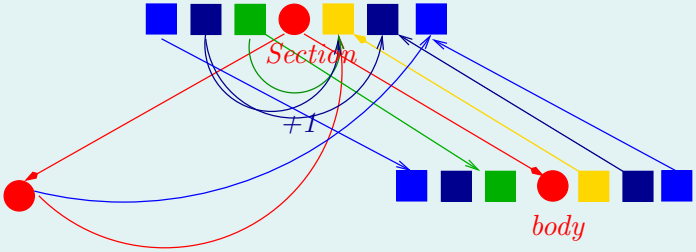
```
data Doc|Toc
sem Doc| Toc  lhs.html     = @lhs.toc
sem Root|Root doc.toc      = @doc.toclines
              doc.level    = 1
              doc.context  = ""
              doc.count    = 1
```

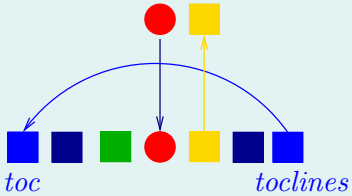
A picture with the toc and toclines added











...

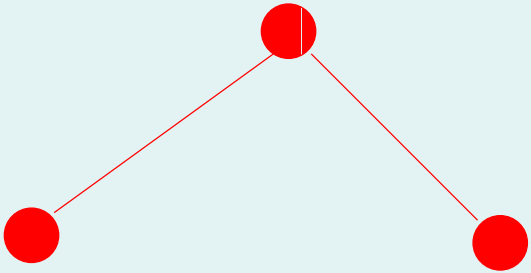
```
sem Doc
  |Section
    lhs.toclines = mk_tag "LI" ""
                     (mk_tag ("A")
                             (" HREF=\#" ++ @prefix)
                             (@prefix ++ " " ++ @title))
                 ++ mk_tag "UL" "" @body.toclines
    lhs.html :=    mk_tag "A" (" NAME=" ++ @prefix) ""
                 ++ @loc.html
```

Backward flow of data

- we want to be able to jump to the previous and next section of the current section
- we introduce two new attributes for passing this information around

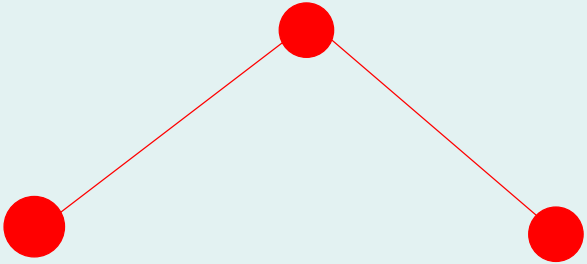
The non-terminals

Cons



Nil

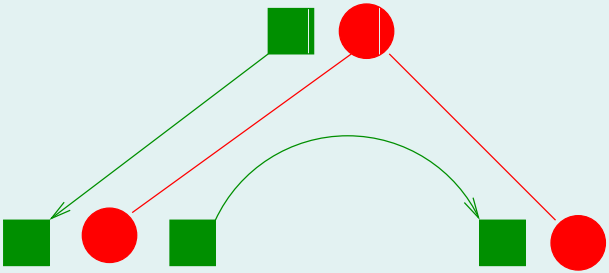
Section



Pcdata

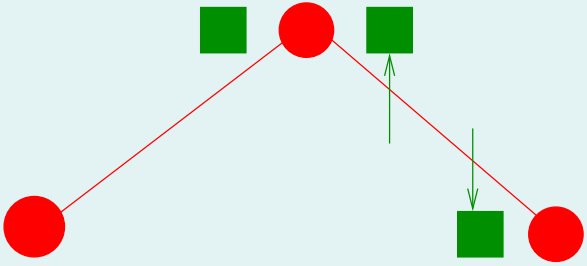
Left

Cons

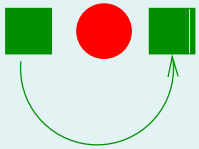


Nil

Section

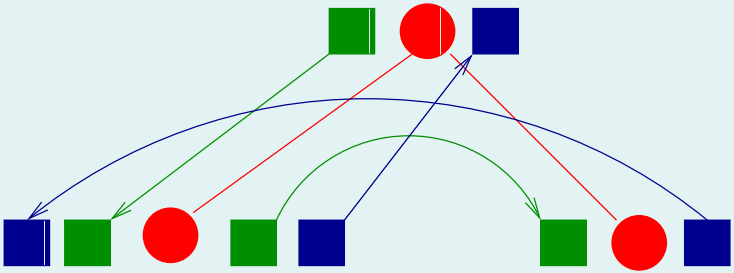


Pcdata

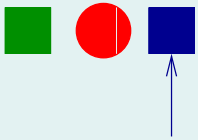


Right

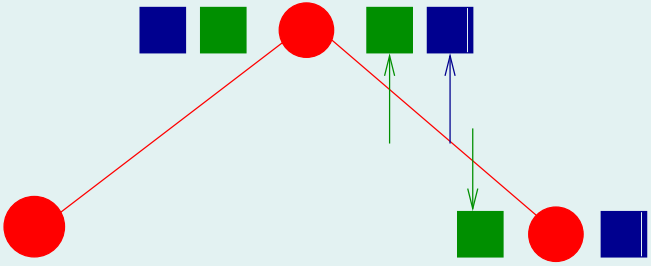
Cons



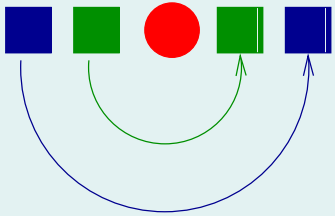
Nil



Section



Pcdata



AG's as a D(omain) S(pecific) L(anguage)

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

AG's as a D(omain) S(pecific) L(anguage)

- attribute grammars are a domain specific language for describing *catamorphisms*
- with higher-order domains, i.e. we generate 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
  - special constructs like the USE clause that was used to combine attribute values presented by children
  - some systems allow to refer to *far away* attributes, resembling *aspect-oriented programming* as known from the OO world

What is generated by the AG System?

- data types

$$
\begin{aligned}
\textbf{data}\ Doc = \ & Doc_Para\ (String) \\
& |Doc_Section\ (String)\ (Docs) \\
& |Doc_Toc
\end{aligned}
$$

... and ...

- types

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

... and ...

- semantic functions

$$
\begin{array}{l}
_sem_Docs_Cons\ ({}_h d)\ ({}_t l) = \\
\quad \backslash\backslash\ _lhs_context \\
\quad\quad _lhs_count \\
\quad\quad _lhs_level \\
\quad\quad _lhs_toc \rightarrow \\
\quad\quad\quad \mathbf{let}\ (_hd_count, _hd_html, _hd_toclines) = \\
\quad\quad\quad\quad\quad ({}_h d\ (_lhs_context)\ (_lhs_count)\ (_lhs_level)\ (_lhs_toc)) \\
\quad\quad\quad\quad (_tl_count, _tl_html, _tl_toclines) = \\
\quad\quad\quad\quad\quad ({}_t l\ (_lhs_context)\ (_hd_count)\ (_lhs_level)\ (_lhs_toc)) \\
\quad\quad\quad \mathbf{in}\ \ (_tl_count, _hd_html +\!\!+ _tl_html, _hd_toclines +\!\!+ _tl_toclines)
\end{array}
$$

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

Other systems ...

- perform an abstract interpretation of the grammar
- computing dependencies between attributes

Other systems ...

- perform an abstract interpretation of the grammar
- computing dependencies between attributes
- schedule the attributes for computation per non-terminal

Other systems ...

- perform an abstract interpretation of the grammar
- computing dependencies between attributes
- schedule the attributes for computation per non-terminal
- and in this way achieve a data-driven evaluation

Other systems ...

- perform an abstract interpretation of the grammar
- computing dependencies between attributes
- schedule the attributes for computation per non-terminal
- and in this way achieve a data-driven evaluation
- that may be somewhat cheaper
- and takes far less space

One of the old questions in Computer Science

- should matrices be stored
  - row-wise
  - or column-wise

One of the old questions in Computer Science

- should matrices be stored
  - row-wise
  - or column-wise
- in both cases there are algorithms that will have a bad paging behaviour
- and the best solution is to break the matrix up into small submatrices

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 new alternatives to data types
  - stepwise adding new 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 new alternatives to data types
  - stepwise adding new attributes
- aspects are largely independent
- but interactions can take place by just referring to other aspects
- the system will weave things together

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 new alternatives to data types
  - stepwise adding new attributes
- aspects are largely independent
- but interactions can take place by just referring to other aspects
- the system will weave things together
- one might compare a complicated system with a protein:
  - nucleid chains ⇔ copy rules (of which we have many)
  - protein folding ⇔ aspect interactions (of which we have a few)

Domain specific language implementation

- can in most cases be easily done by:
  - describing the extra syntax using operator declarations of the right priorities
  - describing the meaning of each operator by a semantic function that is generated from an attribute grammar
  - one synthesized attribute contains the denotational semantics of the construct

...

- in general this is not sufficient, and more properties have to be computed at the same time

...

- in general this is not sufficient, and more properties have to be computed at the same time
- e.g. for a parser we tuple three computations, that build on each other:
  - to compute whether a parse can recognise the empty string
  - a table mapping each look-ahead symbol to the corresponding parser
  - the actual function that looks up the next input token in the table, and calls the corresponding recognizer

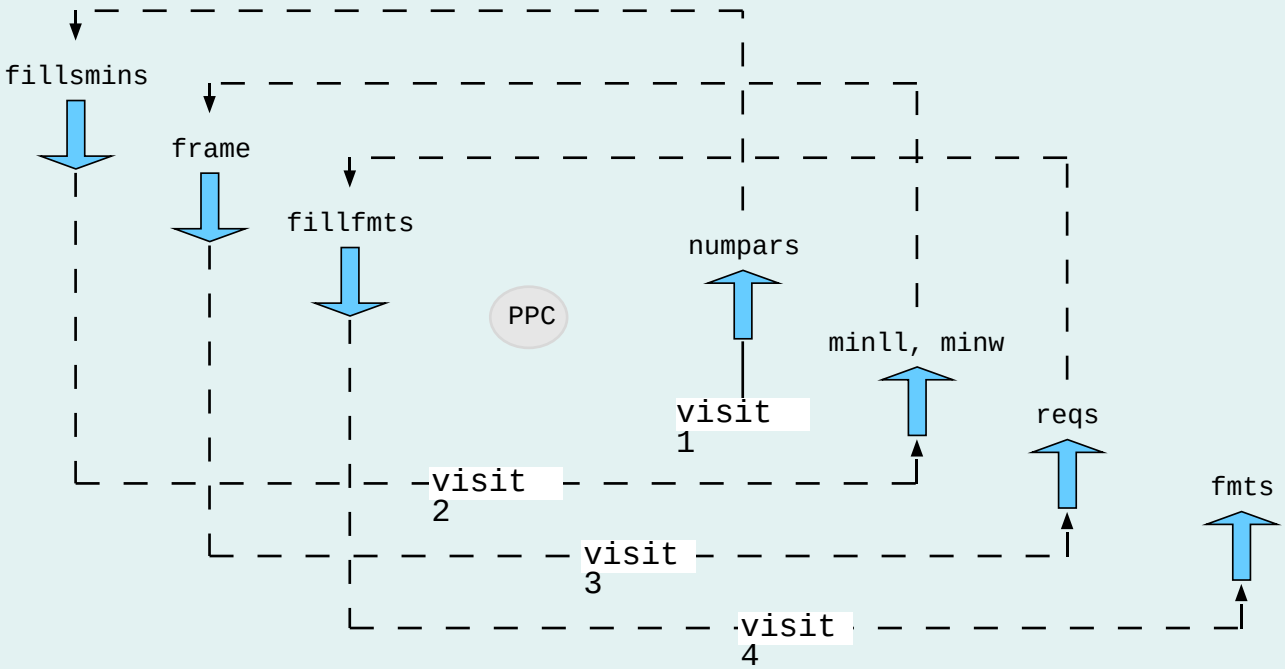
...

- in general this is not sufficient, and more properties have to be computed at the same time
- e.g. for a parser we tuple three computations, that build on each other:
  - to compute whether a parse can recognise the empty string
  - a table mapping each look-ahead symbol to the corresponding parser
  - the actual function that looks up the next input token in the table, and calls the corresponding recognizer
- these are just extra attributes that are discarded when no longer needed, i.e. when the function describing the meaning is selected

...

- in general this is not sufficient, and more properties have to be computed at the same time
- e.g. for a parser we tuple three computations, that build on each other:
  - to compute whether a parse can recognise the empty string
  - a table mapping each look-ahead symbol to the corresponding parser
  - the actual function that looks up the next input token in the table, and calls the corresponding recognizer
- these are just extra attributes that are discarded when no longer needed, i.e. when the function describing the meaning is selected
- some of the functions we constructed in this way take 8 arguments and deliver 8 results

From our pretty-printing combinators



Why use Attribute Grammars

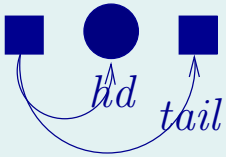
- attribute grammars are your friend if you want to implement a language
- attributes may even depend on themselves if you are building on a lazy language
- even thinking in terms of attribute grammars you may construct interesting programs

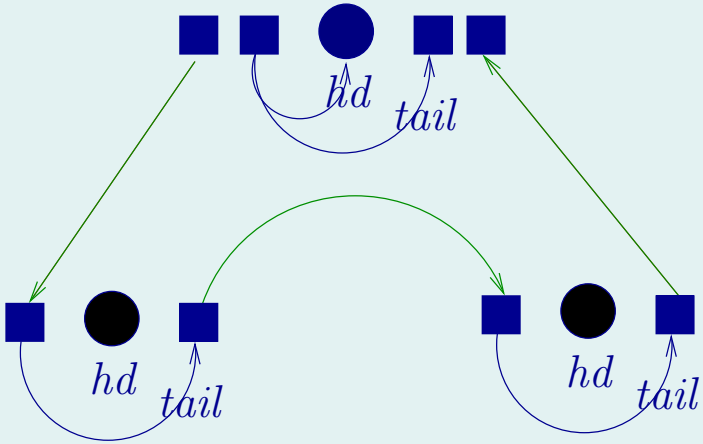
Finale: Breadth-first labelling of a Tree

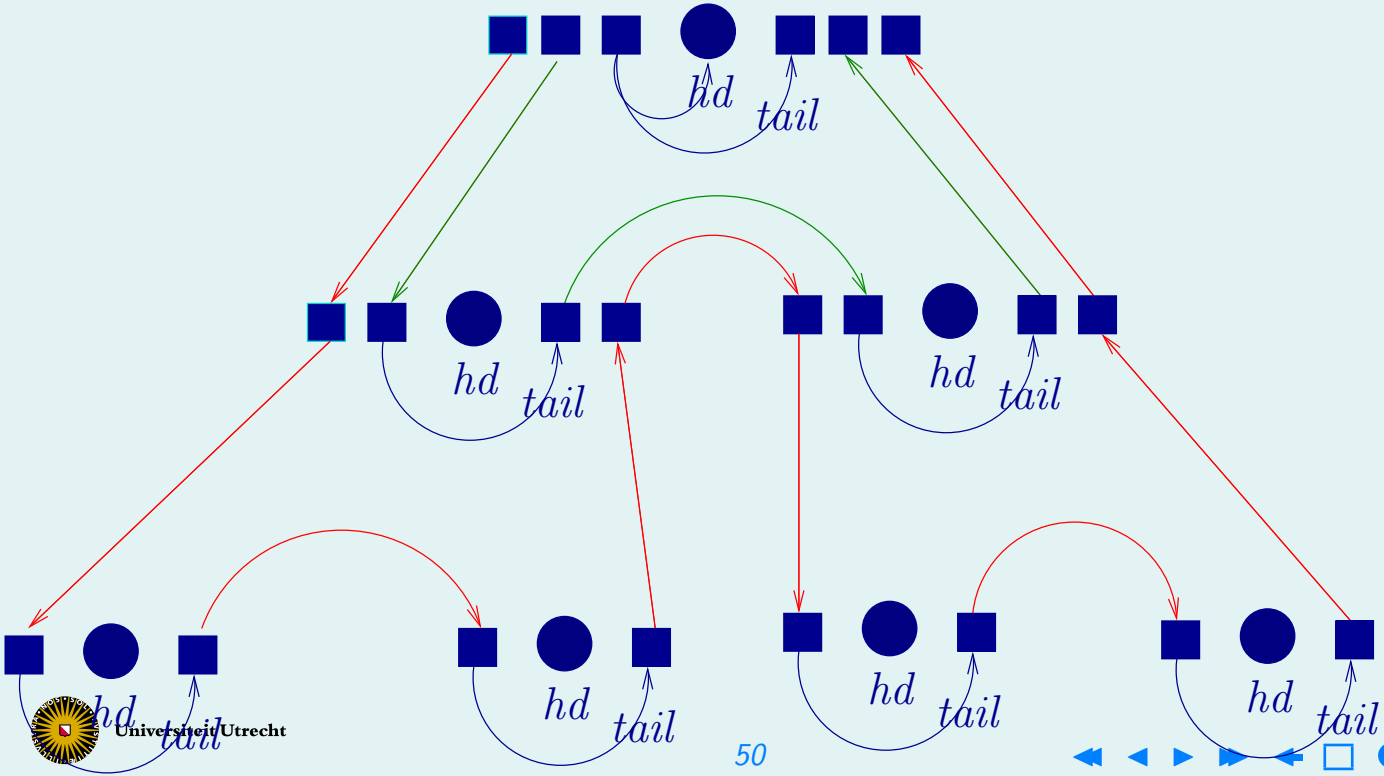
- given a tree we want to

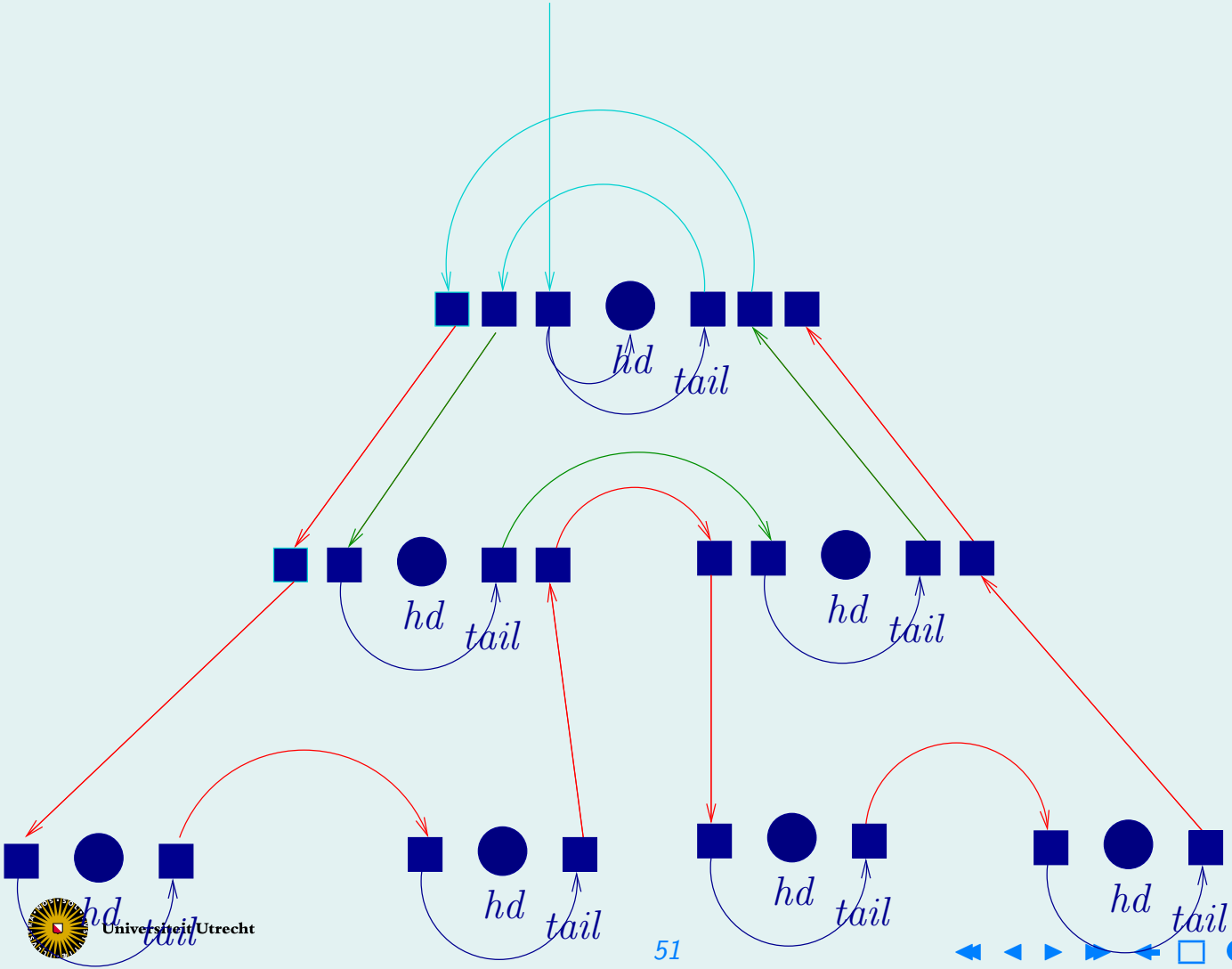
Finale: Breadth-first labelling of a Tree

- given a tree we want to
- construct a new tree that is labelled in a breadth-first manner
- we design programs by drawing pictures


hd
tail


hd
tail
hd
tail
hd
tail


hd
tail
hd
tail
hd
tail
hd
tail
hd
tail
hd
tail
hd
tail


hd
tail
hd
tail
hd
tail
hd
tail
hd
tail
hd
tail
hd
tail

Read the program from the graph!!

```
data Tree a = Leaf
            | Bin a Tree Tree
label t elements
  = let (result, levels) = label' t (elements : levels)
        label' (Bin _ l r) ((f : ff) : rest)
                 = let (ltree, rest')  = label' l rest
                       (rtree, rest'') = label' r rest'
                   in  (Bin f ltree rtree, ff : rest'')
    in result
```

- note that there is only one list that is threaded through the tree
- a list attribute represents a list of attributes: one for each level!

Desirable properties of AG's

We want a system to support the following useful properties:

- F(irst)-C(lassness): define your own copy rules and other systematic attributions
- G(rammar) C(ompleteness): every attribute is defined
- The semantic rules are T(ype) C(orrect)

Desirable properties of AG's

We want a system to support the following useful properties:

- F(irst)-C(lassness): define your own copy rules and other systematic attributions
- G(rammar) C(ompleteness): every attribute is defined
- The semantic rules are T(ype) C(orrect)

	FC	GC	TC
UAG as demonstrated	-	+	+/-
First-class AG (de Moor, Swierstra, Backhouse)	+	-	+
Extendible Records	+-	+	+

Desirable properties of AG's

We want a system to support the following useful properties:

- F(irst)-C(lassness): define your own copy rules and other systematic attributions
- G(rammar) C(ompleteness): every attribute is defined
- The semantic rules are T(ype) C(orrect)

	FC	GC	TC
UAG as demonstrated	-	+	+/-
First-class AG (de Moor, Swierstra, Backhouse)	+	-	+
Extendible Records	+-	+	+

In the last approach we use extendible records in Haskell to encode the Grammar Correctness through the Haskell type system. It can be done, but error messages are incomprehensible.