

# Utrecht Attribute Grammar System

## Syntax Macros and Attribute Redefinitions

Arthur Baars and Doaitse Swierstra

Institute of Information and Computing Science  
Utrecht University

e-mail: {arthurb,doaitse}@cs.uu.nl

Januari 9, 2004

# Overview

- ▶ UUAG System by example
- ▶ Syntax Macros and Attribute Redefinitions

# 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
  - incremental language editors (Synthesizer Generator, Reps/Teitelbaum)
  - ...

# Historical Overview

- ▶ Context-free grammars have limited expressiveness, and thus fail to describe:
  - 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)
- ▶ with trees; affix grammars
- ▶ with values from some other domain: *attribute grammars* (Knuth)

# An attribute grammar consists of:

- ▶ an underlying context free grammar

# An attribute grammar consists of:

- ▶ an 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:

- ▶ an underlying context free grammar
- ▶ a description of which nonterminals have which attributes:
  - *inherited* attributes, that are used for 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*

# Attribute Grammars

An Attribute Grammar describes *global* data flow over a tree, by defining *local* data-flow building blocks, that correspond to the productions of the grammar

## Advantages of the Utrecht Attribute Grammar System

- ▶ Compositional
- ▶ Step-wise refinement
  - add attributes
  - add productions
  - override attribute definitions
- ▶ Haskell expressions for semantic functions
- ▶ Automatic generation of trivial rules

# An Example: Creating HTML using UAG

- ▶ Grammars closely resemble Haskell data types:

```
type Docs = [Doc]
data Doc | Section title : String body : Docs
        | PCDATA string : String
```

- ▶ *Docs* and *Doc* are non-terminals
- ▶ *Section* and *PCDATA* label different alternatives/productions
- ▶ *title*, *body* and *string* are names for children

# Example document

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

# Simple printing of example

**Intro**

**Section**

paragraph

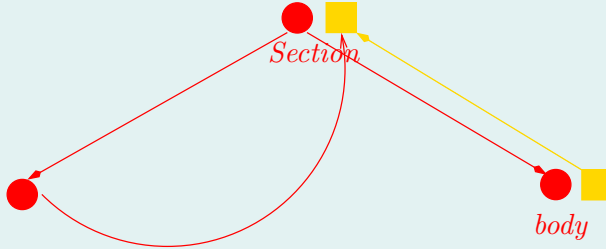
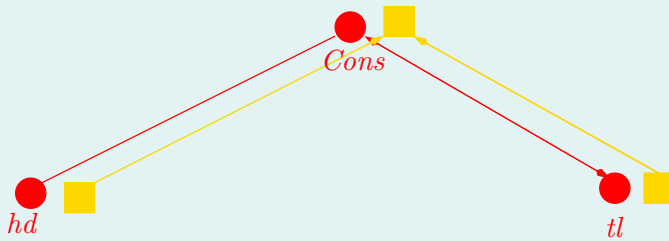
paragraph

**Section**

paragraph

paragraph

# Building blocks



# 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]
```

# 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 = "<strong>" + title + "</strong>\n"  
                  + @body.html
```

```
|Pcdata lhs.html = "<p>" + string + "</p>"
```

```
sem Docs
```

```
|Cons lhs.html = @hd.html + @tl.html
```

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

## **Intro**

## **Section**

paragraph

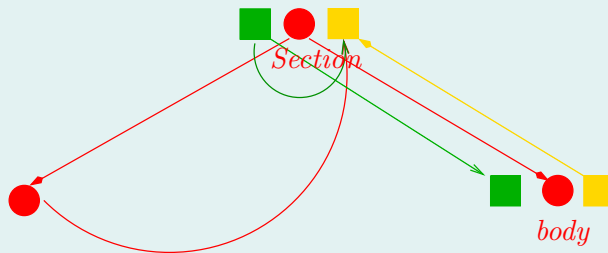
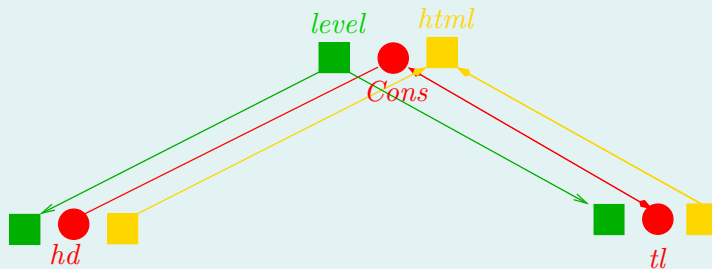
paragraph

## **Section**

paragraph

paragraph

# Building blocks with level added



# Adding the Level aspect

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

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

# 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
```

# 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 elem = "<" ++ tag ++ ">" ++ elem  
                ++ "</" ++ tag ++ ">"
```

## 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

## 1 Intro

### 1.1 Section

paragraph

paragraph

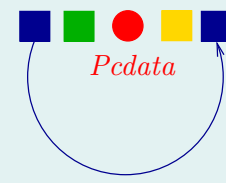
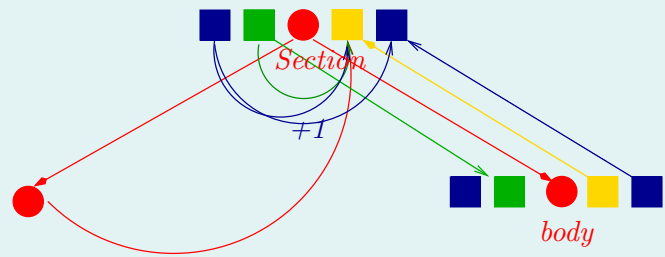
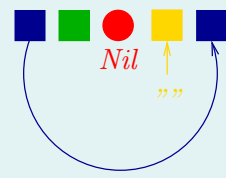
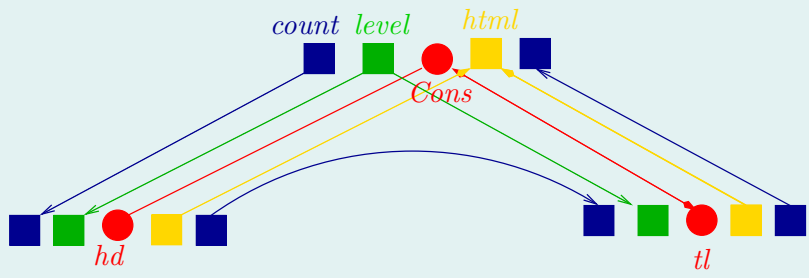
### 1.2 Section

paragraph

paragraph



# Productions with the count added



# 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*

# The semantic functions

**sem** *Doc*

```
| Section body.count = 1
   body.context = @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)
                 (@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
```

# Attribute Grammars

An Attribute Grammar describes *global* data flow over a tree, by defining *local* data-flow building blocks, corresponding to the productions of the grammar

## Advantages of the Utrecht Attribute Grammar System

- ▶ Compositional
- ▶ Step-wise refinement
  - add attributes
  - add productions
  - override attribute definitions
- ▶ Haskell expressions for semantic functions
- ▶ Automatic generation of trivial rules
- ▶ You can write cool programs using UAG(e.g. Helium)

Download:

- ▶ <http://www.cs.uu.nl/groups/ST/>

# Syntax Macros and Attribute Redefinitions

...from now on, a main goal in designing a language should be to plan for growth. The language must start small, and the language must grow as the set of users grows.

[Guy Steele]

- ▶ small core language
- ▶ possibility for growth

# Growing a language with UAG

- ▶ Attribute Grammar
  - defines language and its semantics

growth:

- ▶ Syntax macros
  - extend concrete syntax
- ▶ Attribute redefinitions
  - adapt semantics

# Example: Attribute Grammar

```
DATA Expression
```

```
| Case expr:Expression  
      branches:CaseArms  
| Var name:String  
| Apply fun:Expression arg:Expression  
| ...
```

```
TYPE CaseArms = [CaseArm]
```

```
DATA CaseArm
```

```
| CaseArm pattern:Expression expr:Expression
```

```
ATTR Expression CaseArms CaseArm [ | | pretty:PP_Doc ]
```

```
SEM Expression
```

```
| Case lhs.pretty = "case" >#< @expr.pretty >#< "of"  
                    >-< indent 2 @branches.pretty
```

```
...
```

# Example: if-then-else

Translation scheme from Haskell Report:

```
if  $exp_1$  then  $exp_2$  else  $exp_3$ 
  ⇒
case  $exp_1$  of
  True  →  $exp_2$ 
  False →  $exp_3$ 
```

Translation scheme as a syntax macro:

nonterminals:

Expr :: Expression

rules:

```
Expr ::= "if" exp1=Expr "then" exp2=Expr "else" exp3=Expr
      => Case exp1
          (CaseArms_Cons (CaseArm (Var "True" ) exp2 )
           (CaseArms_Cons (CaseArm (Var "False") exp3 )
            CaseArms_Nil))
```

# Example: if-then-else

Translation scheme as a syntax macro using concrete syntax:

nonterminals:

Expr :: Expression

rules:

```
Expr ::= "if" exp1=Expr "then" exp2=Expr "else" exp3=Expr
=> case [| exp1 |] of
    True   -> [| exp2 |]
    False  -> [| exp3 |]
```

The symbols [|, and |] are used to switch between concrete syntax and abstract syntax.

# Example: if-then-else

The following:

```
test  $x = \text{if } x \text{ then 'a' else 'A'}$ 
```

is translated into:

```
test  $x = \text{case } x \text{ of}$   
  True  $\rightarrow \text{'a'}$   
  False  $\rightarrow \text{'A'}$ 
```

## Example: if-then-else

However, for the following erroneous program

```
test x = if x then 'a' else "A"
```

this error message is given:

```
Couldn't match 'Char' against 'String'  
  Expected type: Char  
  Inferred type: String  
In a case alternative: False -> "A"  
In the case expression:  
  case x of  
    True -> 'a'  
    False -> "A"
```

Confusing for a programmer! Messages are given about translated expression.

# Attribute redefinition

nonterminals:

```
Expr ::= Expression
```

rules:

```
Expr ::= "if" exp1=Expr "then" exp2=Expr "else" exp3=Expr
=> case [| exp1 |] of
  True   -> [| exp2 |]
  False  -> [| exp3 |]
{
  lhs.pretty = text "if" >#< @exp1.pretty
               >-< text "then" >#< @exp2.pretty
               >-< text "else" >#< @exp3.pretty
}
```

The redefinition only redefines the pretty printing aspect, all other aspects are left unchanged.

# List comprehension: Haskell report

Haskell report: List comprehensions satisfy these identities, which may be used as a translation into the kernel:

$$\begin{aligned} [e] &= [e] \\ [e|b, Q] &= \mathbf{if} \ b \ \mathbf{then} \ [e|Q] \ \mathbf{else} \ [] \\ [e|p \leftarrow l, Q] &= \mathbf{let} \ ok \ x = \mathbf{case} \ x \ \mathbf{of} \\ &\quad \begin{array}{l} p \rightarrow [e|Q] \\ \_ \rightarrow [] \end{array} \\ &\quad \mathbf{in} \ \mathit{concatMap} \ ok \ l \\ [e|\mathbf{let} \ \mathit{decls}, Q] &= \mathbf{let} \ \mathit{decls} \ \mathbf{in} \ [e|Q] \end{aligned}$$

where  $e$  ranges over expressions,  $p$  over patterns,  $l$  over list-valued expressions,  $b$  over boolean expressions,  $\mathit{decls}$  over declaration lists,  $q$  over qualifiers, and  $Q$  over sequences of qualifiers.  $ok$  and  $x$  are fresh variables. The function `concatMap`, and boolean value `True`, are defined in the `Prelude`.

Note: the expression  $e$  is pushed to the end of the list of qualifiers.

# List comprehension, as a syntax macro

nonterminals:

```
Expr      :: Expression
Pattern   :: Expression
Decls     :: Declarations
Qualifiers :: Expression -> Expression
```

rules:

```
Expr      ::= "[" e=Expr "|" qs=Qualifiers "]"
           => [| qs e |]
```

```
Qualifiers ::=
           => [| \e::Expr . [e] |]
```

```
Qualifiers ::= b=Expr "," qs=Qualifiers
           => [| \e::Expression . [| if [|b|]
                                     then [|qs e|]
                                     else []
                                     |]
           |]
```

# List comprehension, as a syntax macro

```
Qualifiers ::= p=Pattern "<-" l=Expr "," qs=Qualifiers
             ok=Fresh x=Fresh
=> [| \e::Expression .
      [| let [|ok|] [|x|]
           = case [|x|] of
               [|p|] -> [|qs e|]
               _     -> []
           in concatMap [|ok|] [|l|]
      |]
    |]
```

```
Qualifiers ::= "let" decls=Decls "," qs=Qualifiers
=> [| \e::Expression . let [|decls|]
                          in [|qs e|]
    |]
```

# List comprehension(4)

The following expression:

```
let as = [1, 2]
in [a | a ← as, even a]
```

is now translated into:

```
let as = [1, 2]
in let _1 = λ_2 → case _2 of
    a → if even a
        then [a]
        else []
    _ → []
in concatMap _1 as
```

# Syntax Macros and Attribute redefinitions

- ▶ Attribute Grammar
  - defines language and semantics
  - types, constructors, and attributes
- ▶ Syntax Macros
  - map new syntax onto the core language
- ▶ Attribute redefinitions
  - adapt semantic rules

# Syntax Macros and Attribute redefinitions

- ▶ Extendible parsers
  - combinator parsers
- ▶ Interpretation of macros and redefinitions
  - meta information about types, constructors, and attributes is generated from attribute grammar
  - dynamic typing
- ▶ Extendible attribute grammars
  - techniques from first-class attribute grammars(Oege de Moor, Kevin Backhouse, Doaitse Swierstra)

# Questions?