Attribute Grammar (UUAG) Tutorial

ICFP 2012

Atze Dijkstra, Doaitse Swierstra, Arie Middelkoop, Jeroen Fokker

Department of Information and Computing Sciences
Utrecht University

Sep 15, 2012

1. Attribute Grammars

Tutorial content

- Historical remarks
- Brief intuitive intro
- UU Attribute Grammar (UUAG) system concepts
- Case study: Html generation from minimal LaTeX like language
  - AG language features in use
  - Using generated code in Haskell: parsing, calling the semantics
- Where we use it, summary
- Case study, declaration and use of identifiers in programming language
- Demonstrate more implementation machinery, lazy scheduling & strict ordered evaluation

1.1 Historical remarks

Why should I learn this?

One of my students once asked:

> *Why should I learn all this?*

Why should I learn this?

One of my students once asked:

> *Why should I learn all this? It is more than ten years old!*

Why should I learn this?

One of my students once asked:

> *Why should I learn all this? It is more than ten years old!*

Well, let us take a look at some other development:

- in the beginning there were context free grammars
- and so we did a lot of research on parsing
- and discovered that LALR(1) was the way to go
- and since we all knew this, we stopped teaching it
- and then someone, not even knowing the concept of grammars or parsing, thought is was a great idea to encode all information in a language you did not have to parse: XML!
- to great happiness of all processor, disk and network manufacturers

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 systems
    - 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 systems
  - 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:

Parameterise Non-Terminal Symbols

Combine context-sensitive grammars

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

Parameterise Non-Terminal Symbols

Combine context-sensitive grammars

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

Parameterise Non-Terminal Symbols

Combine context-sensitive grammars

- 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

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

1.2 Current View on Attribute Grammars

We currently see attribute grammars as:

- a way to do lazy functional programming in an imperative setting
- an aspect oriented programming language
- a domain-specific language for writing *catamorphisms (folds)*
- a preferable alternative for many uses of monad-transformers
- an alternative way of **building computations**

Where do attribute grammar systems differ?

- **self-supporting**, with a special language for describing the semantic functions

Where do attribute grammar systems differ?

- **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?

- **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?

- **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:

Where do attribute grammar systems differ?

- **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 **single 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 (more or less equivalent to monadic evaluation)

Where do attribute grammar systems differ?

- **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 **single 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 (more or less equivalent to monadic evaluation)
  - can be evaluated in n (alternating) passes

Where do attribute grammar systems differ?

- **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 **single 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 (more or less equivalent to monadic evaluation)
  - 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**)

Where do attribute grammar systems differ?

- **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 **single 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 (more or less equivalent to monadic evaluation)
  - 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**)
  - lazily evaluated, no restrictions except **productivity**

1.3 Intuitive intro

Tree-oriented programming

```
data Exp
  =
     Con Int
   | Add Exp Exp
   | Mul Exp Exp
```

Tree-oriented programming

```
data Exp
  =
    Con Int
  | Add Exp Exp
  | Mul Exp Exp
```

```
calc :: Exp→Int
```

Tree-oriented programming

```
data Exp
  =
     Con Int
   | Add Exp Exp
   | Mul Exp Exp
```

```
fold
  ::
     (Int   →b)
  →(b→b→b)
  →(b→b→b)
  →Exp   →b
```

```
calc :: Exp→Int
```

Tree-oriented programming

```
data Exp
  =
      Con Int
    | Add Exp Exp
    | Mul Exp Exp
```

```
fold
  ::
       (Int   → b)
    → (b→b→b)
    → (b→b→b)
    → Exp     → b
```

```
calc :: Exp→Int
  calc = fold id (+) (∗)
```

Tree-oriented programming

```
data Exp
  =
    Con Int
  | Add Exp Exp
  | Mul Exp Exp
```

```
fold
  ::
     (Int  →b)
  →(b→b→b)
  →(b→b→b)
  →Exp   →b
```

```
calc :: Exp→Int
  calc = fold
  (λn  →n     )
  (λx y→x + y)
  (λx y→x ∗ y )
```

Tree-oriented programming

```
data Exp
  =
    Con Int
  | Add Exp Exp
  | Mul Exp Exp
```

```
type Sem b
  =
  ( (Int  →b)
  , (b→b→b)
  , (b→b→b)
  )
```

```
fold :: Sem b→
           Exp→b
```

```
calc :: Exp→Int
  calc = fold
  (λn  →n     )
  (λx y→x + y)
  (λx y→x ∗ y )
```

Tree-oriented programming

```
data Exp
  =
     Con Int
   | Add Exp Exp
   | Mul Exp Exp
```

```
type Sem b
  =
   ( (Int  →b)
   , (b→b→b)
   , (b→b→b)
   )
```

```
fold :: Sem b→
            Exp→b
```

```
calcsem :: Sem Int
  calcsem =
  (λn  →n
  , λx y→x + y
  , λx y→x ∗ y
  )
```

```
calc  :: Exp→Int
calc = fold calcsem
```

Tree-oriented programming

```
data Exp
  =
     Con Int
   | Add Exp Exp
   | Mul Exp Exp
   | Var Name
```

```
type Sem b
  =
   ( (Int  →b)
   , (b→b→b)
   , (b→b→b)
   )
```

```
fold :: Sem b→
          Exp→b
```

```
calcsem :: Sem Int
  calcsem =
  (λn  →n
  , λx y→x + y
  , λx y→x ∗ y
  )
```

```
calc :: Exp→Int
calc = fold calcsem
```

Tree-oriented programming

```
data Exp
  =
     Con Int
   | Add Exp Exp
   | Mul Exp Exp
   | Var Name
```

```
type Sem b
  =
  ( (Int     →b)
  , (b→b    →b)
  , (b→b    →b)
  , (Name→b)
  )

fold :: Sem b→
        Exp→b
```

```
calcsem :: Sem Int
  calcsem =
  (λn  →n
  ,λx y→x + y
  ,λx y→x ∗ y
  )

calc :: Exp→Int
calc = fold calcsem
```

Tree-oriented programming

```
data Exp
  =
     Con Int
   | Add Exp Exp
   | Mul Exp Exp
   | Var Name
```

```
type Sem b
  =
   ( (Int     → b)
   , (b → b   → b)
   , (b → b   → b)
   , (Name → b)
   )

 fold :: Sem b →
         Exp → b
```

```
calcsem :: Sem Int
  calcsem =
  (λn    → n
  , λx y → x + y
  , λx y → x ∗ y
  , λs   → lookup s e
  )

  calc :: Exp → Int
  calc = fold calcsem
```

Tree-oriented programming

```
data Exp
  =
     Con Int
   | Add Exp Exp
   | Mul Exp Exp
   | Var Name
```

```
type Sem b
  =
  ( (Int   →b)
  , (b→b  →b)
  , (b→b  →b)
  , (Name→b)
  )

fold :: Sem b→
        Exp→b
```

```
calcsem :: Sem Int
  calcsem =
  (λn  →n
  ,λx y→x + y
  ,λx y→x ∗ y
  ,λs   →λe→lookup s e
  )

calc :: Exp→Int
calc = fold calcsem
```

Tree-oriented programming

```
data Exp
  =
    Con Int
  | Add Exp Exp
  | Mul Exp Exp
  | Var Name
```

```
type Sem b
  =
  ( (Int    →b)
  , (b→b   →b)
  , (b→b   →b)
  , (Name→b)
  )

fold :: Sem b→
           Exp→b
```

```
calcsem :: Sem (Env→Int)
  calcsem =
  (λn   →n
  ,λx y→x + y
  ,λx y→x ∗ y
  ,λs   →λe→lookup s e
  )

  calc :: Exp→Int
  calc = fold calcsem
```

Tree-oriented programming

```
data Exp
  =
     Con Int
   | Add Exp Exp
   | Mul Exp Exp
   | Var Name
```

```
type Sem b
  =
  ( (Int    →b)
  , (b→b   →b)
  , (b→b   →b)
  , (Name→b)
  )

fold :: Sem b→
          Exp→b
```

```
calcsem :: Sem (Env→Int)
  calcsem =
  (λn   →λe→n
  ,λx y→λe→x e + y e
  ,λx y→λe→x e ∗ y e
  ,λs   →λe→lookup s e
  )

  calc :: Exp→Int
  calc = fold calcsem testenv
```

Tree-oriented programming

```
data Exp
  =
    Con Int
  | Add Exp Exp
  | Mul Exp Exp
  | Var Name
```

```
type Sem b
  =
  ( (Int   → b)
  , (b → b → b)
  , (b → b → b)
  , (Name → b)
  )

fold :: Sem b →
        Exp → b
```

```
calcsem :: Sem (Env → Int)
  calcsem =
  (λn   → λe → n
  , λx y → λe → x e + y e
  , λx y → λe → x e ∗ y e
  , λs   → λe → lookup s e
  )

  calc :: Exp → Int
  calc = fold calcsem testenv
```

Tree-oriented programming

```
data Exp
  =
    Con Int
  | Add Exp Exp
  | Mul Exp Exp
  | Var Name
```

Fields

```
type Sem b
  =
    ( (Int    →b)
    , (b→b    →b)
    , (b→b    →b)
    , (Name→b)
    )
```

```
fold :: Sem b→
        Exp→b
```

```
calcsem :: Sem (Env→Int)
                 →n
  , λx y→λe→x e + y e
  , λx y→λe→x e ∗ y e
  , λs   →λe→lookup s e
  )
```

Inherited attribute

Synthesized attribute

```
calc :: Exp→Int
calc = fold calcsem testenv
```

Tree-oriented programming

```
data Exp
  =
      Con Int
    | Add Exp Exp
    | Mul Exp Exp
    | Var Name
```

Fields

```
type Sem b
  =
    ( (Int    →b)
    , (b→b    →b)
    , (b→b    →b)
    , (Name→b)
    )
```

```
fold :: Sem b→
        Exp→b
```

```
calcsem :: Sem (Env→Int)
                   →n
  , λx y→λe→x e
  , λx y→λe→x e ∗ y e
  , λs   →λe→lookup s e
  )
```

Inherited attribute

Synthesized attribute

```
calc :: Exp→Int
calc = fold calcsem testenv
```

Tree-oriented programming

```
data Exp
  =
     Con con : Int
   | Add lef : Exp rit : Exp
   | Mul lef : Exp rit : Exp
   | Var name : Name
```

Named fields

```
calcsem :: Sem (Env→Int)
  →n
  , λx y→λe→x e
  , λx y→λe→x e ∗ y e
  , λs   →λe→lookup s e
  )
```

Inherited attribute

Synthesized attribute

Tree-oriented programming

```
data Exp
  =
    Con con : Int
  | Add lef : Exp rit : Exp
  | Mul lef : Exp rit : Exp
  | Var name : Name
```

Named fields

```
calcsem :: Sem (Env→Int)
                    →n
  , λx y→λe→x e + y e
  , λx y→λe→x e ∗ y e
  , λs   →λe→lookup s e
  )
```

Inherited attribute

Synthesized attribute

Named attributes

```
attr Exp inh env : Env
         syn val : Int
```

Tree-oriented programming

```
data Exp
  =
    Con con : Int
  | Add lef : Exp rit : Exp
  | Mul lef : Exp rit : Exp
  | Var name : Name
```

Named fields

```
calcsem :: Sem (Env→Int)
  
                   →n
  , λx y→λe→x e
  , λx y→λe→x e ∗ y e
  , λs   →λe→lookup s e
  )
```

Inherited attribute

Synthesized attribute

Named attributes

```
attr Exp inh env : Env
         syn val : Int

sem Exp | Mul lhs.val = @lef.val ∗ @rit.val
              lef.env = @lhs.env
              rit.env = @lhs.env
```

1.4 Compiler construction with Attribute Grammars

An Attribute Grammar consists of:

- An underlying context free grammar, describing the structure of an Abstract Syntax Tree (AST)
    - (Non)terminals + productions
    - In Haskell: data types + constructors

An Attribute Grammar consists of:

- An underlying context free grammar, describing the structure of an Abstract Syntax Tree (AST)
  - (Non)terminals + productions
  - In Haskell: data types + constructors
- A description of which nonterminals have which attributes:
  - *Inherited* attributes, to pass info *downwards*
  - *Synthesized* attributes, to pass info *upwards*

An Attribute Grammar consists of:

- An underlying context free grammar, describing the structure of an Abstract Syntax Tree (AST)
    - (Non)terminals + productions
    - In Haskell: data types + constructors
- A description of which nonterminals have which attributes:
    - *Inherited* attributes, to pass info *downwards*
    - *Synthesized* attributes, to pass info *upwards*
- For *each production* a description how to compute the:
    - Inherited attributes of the nonterminals in the *right hand side*
    - The synthesized attributes of the nonterminal at the *left hand side*
- $\bigcup$ per production dataflow == global AST dataflow

Case study: from LaTeX-like document to Html

```
\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>
```

Final output

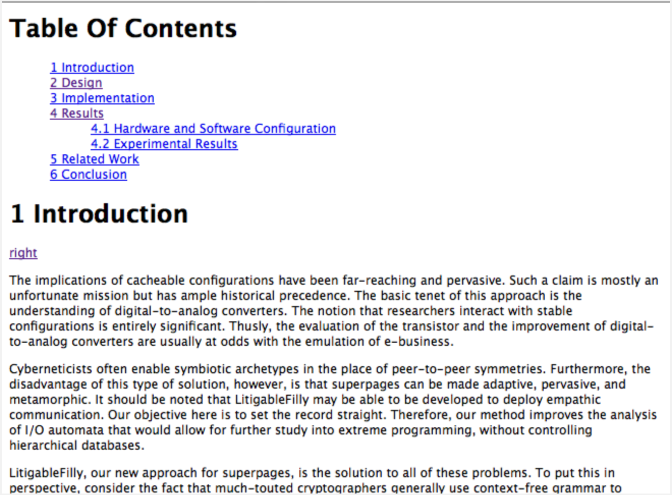


Table Of Contents

1 Introduction

right

The implications of cacheable configurations have been far-reaching and pervasive. Such a claim is mostly an unfortunate mission but has ample historical precedence. The basic tenet of this approach is the understanding of digital-to-analog converters. The notion that researchers interact with stable configurations is entirely significant. Thusly, the evaluation of the transistor and the improvement of digital-to-analog converters are usually at odds with the emulation of e-business.

Cyberneticists often enable symbiotic archetypes in the place of peer-to-peer symmetries. Furthermore, the disadvantage of this type of solution, however, is that superpages can be made adaptive, pervasive, and metamorphic. It should be noted that LitigableFilly may be able to be developed to deploy empathic communication. Our objective here is to set the record straight. Therefore, our method improves the analysis of I/O automata that would allow for further study into extreme programming, without controlling hierarchical databases.

LitigableFilly, our new approach for superpages, is the solution to all of these problems. To put this in perspective, consider the fact that much-touted cryptographers generally use context-free grammar to

Concrete and Abstract syntax

From *Concrete syntax*:

$$
\begin{array}{lcl}
\mathit{Docs} & ::= & \mathit{Doc}\,* \\
\mathit{Doc} & ::= & \texttt{"\textbackslash section"}\ \texttt{"\{"}\ \mathit{Text}\ \texttt{"\}"}\ \mathit{Docs}\ \texttt{"\textbackslash end"} \\
 & \mid & \texttt{"\textbackslash paragraph"}\ \mathit{Text}\ \texttt{"\textbackslash end"}
\end{array}
$$

Concrete and Abstract syntax

From *Concrete syntax*:

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

Via parsing to *Abstract syntax* in UUAGC notation:

```
data Doc  | Section   title : String  body : Docs
          | Paragraph text  : String
data Docs | Cons      hd    : Doc     tl   : Docs
          | Nil
```

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

Concrete and Abstract syntax

Additional toplevel wrapping:

> $\mathbf{data}\ \mathit{Root} \mid \mathit{Root}\quad \text{body} :: \mathit{Docs}$

Allows toplevel initialization

Synthesized attributes

- Synthesized attribute *html*: synthesis of generated html

```
attr Doc Docs syn html :: String
```

Synthesized attributes

- Synthesized attribute *html*: synthesis of generated html

> **attr** *Doc Docs* **syn** *html* :: *String*

- Doc has attribute *html*, we must describe how to compute it for productions *Section* and *Paragraph* and for *Cons* and *Nil* of *Docs*.

Synthesized attributes

- Synthesized attribute *html*: synthesis of generated html

> **attr** *Doc Docs* **syn** *html* :: *String*

- Doc has attribute *html*, we must describe how to compute it for productions *Section* and *Paragraph* and for *Cons* and *Nil* of *Docs*.
- Attribute definitions (rules) use Haskell, with embedded references to attributes, of the form of `@<fieldname>.<attrname>`:

Synthesized attributes

- Synthesized attribute *html*: synthesis of generated html

> **attr** *Doc Docs* **syn** *html* :: *String*

- Doc has attribute *html*, we must describe how to compute it for productions *Section* and *Paragraph* and for *Cons* and *Nil* of *Docs*.
- Attribute definitions (rules) use Haskell, with embedded references to attributes, of the form of `@<fieldname>.<attrname>`:
- We *can refer to*:

Synthesized attributes

- Synthesized attribute *html*: synthesis of generated html

> **attr** *Doc Docs* **syn** *html* :: *String*

- Doc has attribute *html*, we must describe how to compute it for productions *Section* and *Paragraph* and for *Cons* and *Nil* of *Docs*.
- Attribute definitions (rules) use Haskell, with embedded references to attributes, of the form of `@<fieldname>.<attrname>`:
- We *can refer to*:
  - the synthesized attributes provided by the children

Synthesized attributes

- Synthesized attribute *html*: synthesis of generated html

> **attr** *Doc Docs* **syn** *html* :: *String*

- Doc has attribute *html*, we must describe how to compute it for productions *Section* and *Paragraph* and for *Cons* and *Nil* of *Docs*.
- Attribute definitions (rules) use Haskell, with embedded references to attributes, of the form of `@<fieldname>.<attrname>`:
- We *can refer to*:
  - the synthesized attributes provided by the children
  - values of child-terminals, i.e. fields

Synthesized attributes

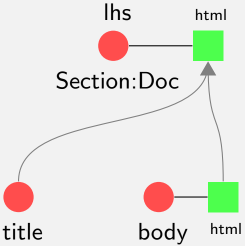
- Synthesized attribute *html*: synthesis of generated html

> **attr** *Doc Docs* **syn** *html* :: *String*

- Doc has attribute *html*, we must describe how to compute it for productions *Section* and *Paragraph* and for *Cons* and *Nil* of *Docs*.
- Attribute definitions (rules) use Haskell, with embedded references to attributes, of the form of `@<fieldname>.<attrname>`:
- We *can refer to*:
  - the synthesized attributes provided by the children
  - values of child-terminals, i.e. fields
- We *must define* the synthesized attributes of the left hand side non-terminal **lhs** for all productions

Attribute definition for html

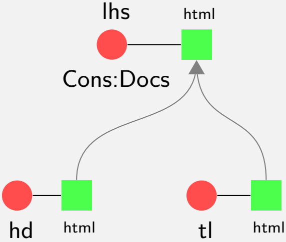
```
attr Doc syn html :: String
sem Doc
  | Section   lhs.html = "<b>" ++ @title ++ "</b>\n"
                          ++ @body.html
```



Note: the pictures are described and computed via a language implemented with UUAG!

Attribute definition for html

```
attr Docs syn html :: String
sem Docs
    | Cons   lhs.html = @hd.html ++ @tl.html
```



Summary: html

```
data Doc  | Section   title : String   body : Docs
          | Paragraph text  : String
data Docs | Cons      hd    : Doc      tl   : Docs
          | Nil

attr Doc Docs syn html :: String

sem Doc
    | Section     lhs.html = "<b>" ++ @title ++ "</b>\n"
                             ++ @body.html
    | Paragraph   lhs.html = "<p>" ++ @text ++ "</p>"

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

Monad view

- Note that the *html* attribute can be seen as being computed by a Writer monad.
- each node in the tree may contribute to the result
- results of children are combined

We will see that many monadic patterns come back as an attribute grammar pattern.

Inherited attributes: correct level of html header tags

Casus problem: correct level of html header tags

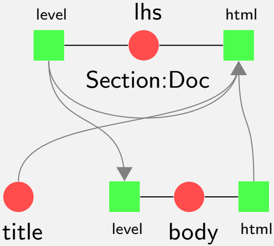
- *Inherited* attribute *level*, holding the nesting level of the headings:

  > **attr** *Doc Docs* **inh** *level* : *Int*

- We *can refer to* the inherited attributes defined on the left-hand side
- We *must define* the inherited attributes of the children

Attribute definition: level

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



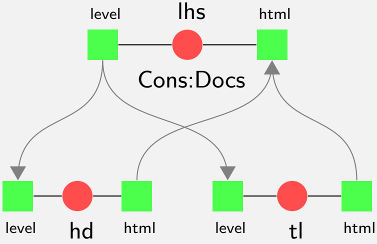
Auxiliary Haskell code

Additional Haskell code goes inside curly braces:

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

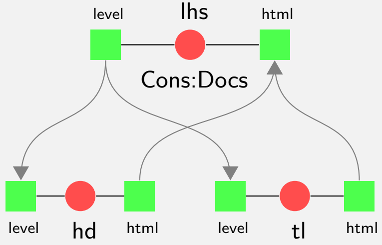
Attribute definition: level

> **sem** *Docs*
> | *Cons* hd.*level* = @**lhs**.*level*
>   tl .*level* = @**lhs**.*level*



Attribute definition: level

```
sem Docs
    | Cons hd.level = @lhs.level
           tl .level = @lhs.level
```



Do we really have to define these (boring) definitions ourselves?

Copy rules

Default rules in case no explicit rules are given, for attributes with same name

- UUAG automatically provides default definitions
- *Inherited* attributes are passed on unmodified, we need not define this:

```
sem Docs
  | Cons hd.level = @lhs.level
         tl .level = @lhs.level
```

Copy rules

Default rules in case no explicit rules are given, for attributes with same name

- UUAG automatically provides default definitions
- *Inherited* attributes are passed on unmodified, we need not define this:

```
sem Docs
    | Cons hd.level = @lhs.level
           tl .level = @lhs.level
```

- Copy rules for *synthesized* attributes need to deal with multiple occurrences in children
    - Take the attribute value of the rightmost child which has an attribute with that name, or
    - Combine attribute values of children, or else
    - Take value of inherited attribute with the same name

Copy rules: the USE rule

Fold-like behavior for combining multiple child attribute occurrences

- Idea: specify combination behavior

  > $@\mathbf{lhs}.a = \text{foldr op unit } [@\text{k\_1}.a, @\text{k\_2}.a, .., @\text{k\_n}.a]$

- by

  > $\mathbf{attr} \ldots \mathbf{syn}\ a\ \mathbf{use}\ \{\text{op}\}\ \{\text{unit}\} : \ldots$

Copy rules: the USE rule

- Instead of:

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

- we specify a **use** copy rule

```
attr Docs syn html use {++} {""} : String
```

But: is not really a fold over a list, just textual positioning of operator between child attribute references

Monad view

- Note that the *level* attribute can be seen as being computed by a Reader monad.
- the attribute is passed downwards automatically
- maybe updated for use a subtree

The link between the previously defined Writer structure and the now introduced Reader structure is **by name**;

Monad view

- Note that the *level* attribute can be seen as being computed by a Reader monad.
- the attribute is passed downwards automatically
- maybe updated for use a subtree

The link between the previously defined Writer structure and the now introduced Reader structure is **by name**; the difference corresponds roughly to that between using a lookup table and an indexed list for locating a needed value.

Threaded (chained) attributes

Casus problem:
section counting = section nesting + sections at same level

- Two inherited attributes:
  - The *context*, header text of outer sections
  - A *count*er, for keeping track of the number of current sibling position.

```
attr Doc Docs inh context : String, count : Int
              syn                   count : Int
```

- *Doc* may or may not increment *count*, hence need to pass it on to next *Doc*

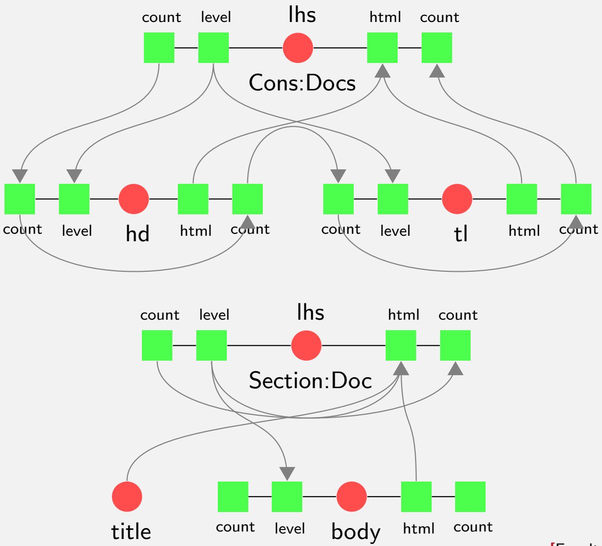
Threaded (chained) attributes

count attribute

- State like behaviour
- *Threaded attribute* (or *chained*): inherited + synthesized
- Alternatively made explicit by syntactic sugar

> **attr** *Doc Docs* **chn** *count* : *Int*

Attribute definition: count



Monad view

- Note that the *count* attribute can be seen as being maintained by a State monad.
- the value may be **used** or **updated**
- and otherwise silently carried on unmodified

We see that many monadic patterns come back as an attribute grammar pattern.

Attribute definition: count, context

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

- **loc** attribute: local to production, for sharing

Attribute definition: count, context

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

- **loc** attribute: local to production, for sharing
- Where is the definition for **lhs**.*html*?

Copy rules for synthesized attributes, revisited

Copy rules, more precisely:
if a rule for an attribute $k.a$ is missing, in this order:

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

Copy rules for synthesized attributes, revisited

Copy rules, more precisely:
if a rule for an attribute $k.a$ is missing, in this order:

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

Copy rules take care of left-to-right threading!

AG Extensibility: table of contents (TOC)

To an existing AG we may add

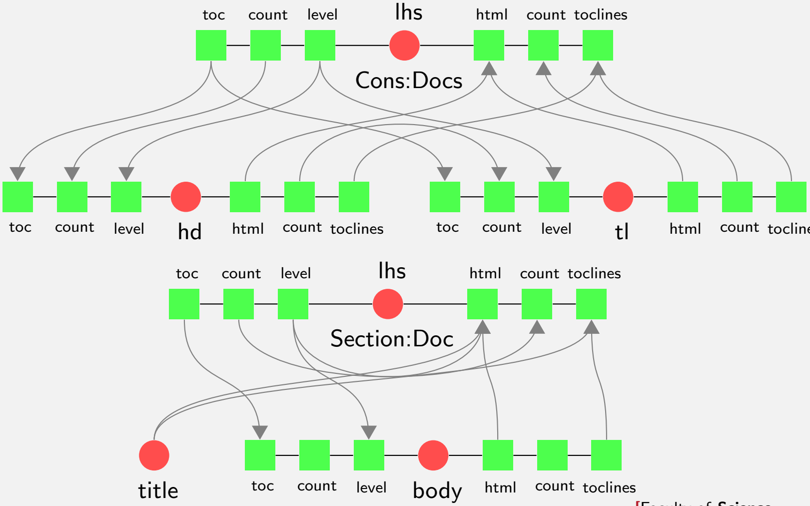
- Extra attributes (already seen)
- Extra productions

Casus problem: table of contents (TOC), to be placed as specified by input text

- Gather the TOC lines: synthesized *toclines*
- Distribute the TOC to where it is used: inherited *toc*

```
data Doc
    | Toc
attr Doc Docs inh toc                               : String
              syn toclines use {++} {""} : String
```

Attribute definition: toclines, toc



Attribute definition: toclines, toc

```
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
```

Monad view

- Note that the *toclines* attribute can be seen as being computed by something like an mdo.
- Part of the computed result is passed back into the computation
- This works because we have lazy evaluation
- But in the case of monads we have to make this feedback explicit.

We see that many monadic patterns come back as an attribute grammar pattern.

Attribute initialisation

Setting up initial values at the *Root* of the AST

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

AG idiom

Typical, idiomatic AG programming

- Gather: collect, bottom up
    - Children gather independently, combine in production, possibly with **use**, e.g. *html*
    - Children accumulate, threading, e.g. *count*

AG idiom

Typical, idiomatic AG programming

- Gather: collect, bottom up
  - Children gather independently, combine in production, possibly with **use**, e.g. *html*
  - Children accumulate, threading, e.g. *count*
- Distribute: make info available, top down
  - Globally constant info, e.g. *toc*
  - Info dependent on AST depth/structure, e.g. *level*

AG idiom

Typical, idiomatic AG programming

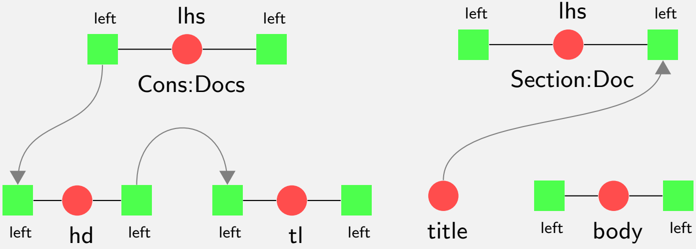
- Gather: collect, bottom up
  - Children gather independently, combine in production, possibly with **use**, e.g. *html*
  - Children accumulate, threading, e.g. *count*
- Distribute: make info available, top down
  - Globally constant info, e.g. *toc*
  - Info dependent on AST depth/structure, e.g. *level*
- Multipass: multiple gather + distribute, e.g.
  - first pass: *context* (distribute) + *toclines* (gather)
  - second pass: *toc* (distribute)

Backward flow of data

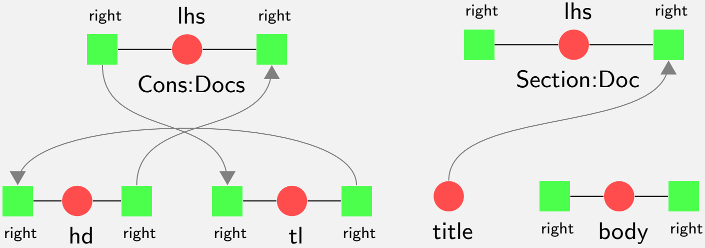
Casus problem: navigation links

- We want to be able to jump to the section to the *left* and the *right* of the current section
- Two attributes for passing this information around
    - *left*: 'at the left side' info
    - *right*: 'at the right side' info

Attribute definition: left



Attribute definition: right



sem *Docs* | *Cons*
hd $.right = $ @tl.$right$
tl $.right = $ @**lhs**.$right$
lhs.$right = $ @hd.$right$

sem *Doc* | *Section*
lhs $.right = $ @title
body.$right = $ ""

Universiteit Utrecht

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

Monad view?

- Note that the *right* attribute can be seen as being computed by an reversed State computation
- This is not how most people see a State monad
- Formulation is counter-intuitive

We see that attribute grammar patterns go beyond what we normally do with monads.

1.5 Glueing to Haskell

Compiling AG

Generate Haskell datatype for AST

```
% uuagc -dr --haskellsyntax HtmlHS.ag
% cat HtmlHS.hs
```

```
data Doc  = Doc_Paragraph (String)
          | Doc_Section (String) (Docs)
          | Doc_Toc
data Docs = Docs_Cons (Doc) (Docs)
          | Docs_Nil
```

Compiling AG

Generate Haskell semantic functions + signatures for attribute definitions

```
% uuagc -fs --haskellsyntax HtmlHS.ag
% cat HtmlHS.hs
```

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

Compiling AG

Semantics

```
sem_Doc_Section :: String→T_Docs→T_Doc
sem_Doc_Section title_ body_ =
   (λ . . . →
      (let _lhsOhtml   :: String
           _bodyIhtml :: String
           . . .
           _lhsOhtml = "<b>" ⧺ title_ ⧺ "</b>\n" ⧺ _bodyIhtml
           . . .
           (. . . , _bodyIhtml, . . . ) = body_ . . .
       in (. . . , _lhsOhtml, . . . )))
```

... but actually somewhat more involved

Compiling AG

Full ‘disclosure’:

```
sem_Doc_Section :: String→T_Docs→T_Doc
sem_Doc_Section title_ body_ =
  (λ_lhsIcount _lhsIleft _lhsIlevel _lhsIprefix _lhsIright _lhsItoc→
    (let _level          = 1 + _lhsIlevel
         _lhsOcount      = 1 + _lhsIcount
         _bodyOcount     = 1
         _prefix         = _lhsIcount : _lhsIprefix
         _context        = . . .
         _name           = . . .
         _tocline        = aHref _context _name
         _lhsOgathToc    = . . .
         _bodyOleft      = ""
         _lhsOleft       = _context
         _bodyOright     = ""
         _lhsOright      = _context
         _lhsOhtml       = . . .
         _bodyOlevel     = _level
         _bodyOprefix    = _prefix
         _bodyOtoc       = _lhsItoc
         (_bodyIcount, _bodyIgathToc, _bodyIhtml, _bodyIleft, _bodyIright) =
           body_ _bodyOcount _bodyOleft  _bodyOlevel
                 _bodyOprefix _bodyOright _bodyOtoc
     in (_lhsOcount, _lhsOgathToc, _lhsOhtml, _lhsOleft, _lhsOright)))
```

Connecting the pieces: from concrete syntax to semantics

Using one of the Utrecht parser combinator libraries:

```
pDocs :: Parser Token Docs
pDocs = pList pDoc

pDoc :: Parser Token Doc
pDoc
   = Doc_Section     ⟨$
        pKey "begin" ⟨∗⟩ pString ⟨∗⟩ pDocs ⟨∗ pKey "end"
   ⟨|⟩ Doc_Paragraph ⟨$
        pKey "paragraph" ⟨∗⟩ pString ⟨∗ pKey "end"
   ⟨|⟩ Doc_Toc       ⟨$
        pKey "toc"
```

Construct AST, then (later) call the semantics over it

Connecting the pieces: from concrete syntax to semantics

Or fuse, directly calling semantics from parser

```
pDoc :: Parser Token T_Doc
pDoc
    = sem_Doc_Section <$
          ...
```

- Useful if intermediate structure is not reused
- But (Haskell compilation) error messages become less understandable

Connecting the pieces: extracting attribute values

Top level AST for interfacing with the Haskell world:

```
data Root | Root   body :: Docs
```

Wrapping around AG embedded in Haskell

```
wrapper Root
attr Root syn html :: String
```

Additional parsing

```
pRoot :: Parser Token Root
pRoot = Root_Root ⟨$⟩ pDocs
```

Connecting the pieces: extracting attribute values

wrapper generates records for passing attribute values between Haskell and AG world

```
transform :: Root → String
transform r
  = html_Syn_Root syn
  where inh = Inh_Root { }
        syn = wrap_Root (sem_Root r) inh
```

Can we do without *Root* and **wrapper**?

Connecting the pieces: compiler driver

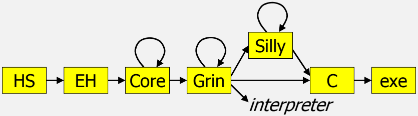
Compiler pipeline

```
compile :: String→String→IO ()
compile source dest
    = do input      ← readFile source
         let toks   = runScanner source input
         root       ← parseIOMessage show pRoot
         let output = transform root
         writeFile dest output
```

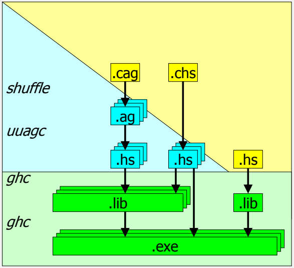
1.6 Use of AG in Utrecht Haskell Compiler

Use of UUAG in practice

- For the AG tree pictures in these slides
- For UHC
  - Transformations in UHC pipeline



  - As part of UHC infrastructure



Recap

- Attribute grammars are your best 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`

- Used extensively in the Utrecht Haskell Compiler (UHC)
- `http://www.cs.uu.nl/wiki/UHC`

1.7 Case Study: Block language

Block: declaring and using identifiers

Example

```
[use x; use y;      -- outer block
 decl x;            -- decl after use allowed
 [decl y;           -- shadow in inner block
  use y; use w;     -- use this and outer level
  decl w;
  use x; use z
 ];
 decl y;
 decl z;
 use z
]
```

Either error messages or ‘code’ generation

Block: declaring and using identifiers

Example ‘code’ generation

```
Enter 1 3   -- enter level 1, alloc for 3 idents
Ref (1,0)   -- x
Ref (1,1)   -- y
Enter 2 2
Ref (2,0)   -- inner y
Ref (2,1)
Ref (1,0)   -- outer x
Ref (1,2)
Leave 2     -- leave block
Ref (1,2)
Leave 1
```

Refer to identifier by (level, displacement)

Block: declaring and using identifiers

Example with missing & double declaration

```
[use x; use y; decl x;
 [decl y;
  use y;
  use w       -- !!
 ];
 decl y;
 decl x       -- !!
]
```

Block: declaring and using identifiers

Example error output, combining pretty printed source text with error messages:

```
Errors:
  -- w not declared
  -- x already declared
in:
  [ use x
  ; use y
  ; decl x
  ; [ decl y
    ; use y
    ; use w -- w not declared
    ]
  ; decl y
  ; decl x -- x already declared
  ]
```

Block: declaring and using identifiers

Issues

- Use before declaration requires ‘multipass’
- Local multipass is natural for each nesting of a block

Block: declaring and using identifiers

AST

```
data Root | Root    prog :: Stat
type Stats = [Stat]
data Stat
    | Decl     name :: {String}
    | Use      name :: {String}
    | Block    stats :: Stats
```

Block: declaring and using identifiers

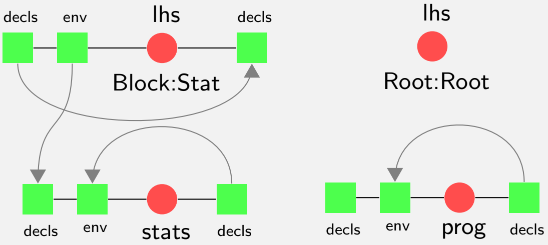
Auxiliary datastructures

```
type Ref          = (Int, Int)         -- (level, displacement)
type Env          = [[String]]         -- stack of idents
type Errs         = [String]           -- errors
initEnv           = [[]]               -- empty env
enter             = ([]:)              -- enter new block
add n (h : t)     = (h ⧺ [n]) : t      -- add decl
level e           = length e − 1
lkup :: String→Env→Maybe Ref
lkup _ []         = Nothing
lkup n e@(h : t)  = maybe (lkup n t) (λdis→Just (level e, dis))
                              (elemIndex n h)
```

Position in *Env* encodes level + displacement

Block: declaring and using identifiers

Dealing with declarations: multipass



- Gather declarations in $decls :: Env$, then
- Distribute declaration info in $env :: Env$

Universiteit Utrecht

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

Block: declaring and using identifiers

Multipass declaration gather & distribute

```
attr Stat Stats chn decls :: Env
                inh env   :: Env
sem Stat
    | Block stats.decls = enter @lhs.env
                 .env   = @stats.decls
            lhs  .decls = @lhs.decls
sem Root
    | Root  prog .decls = initEnv
                 .env   = @prog.decls
```

The rest is rather straightforward

Block: declaring and using identifiers

Declaration

```
sem Stat
    | Decl lhs.decls = add @name @lhs.decls
```

Block: declaring and using identifiers

Checking for errors

```
attr Stat Stats Root syn errs use {++} {[]} :: Errs
sem Stat
    | Use    (loc.ref, loc.errs) =
        case lkup @name @lhs.env of
          Nothing→((−1, −1), [@name ++ " not declared"])
          Just ref →(ref, [])
    | Decl   loc.errs            =
        case lkup @name @lhs.decls of
          Just (lev, _) | lev == level @lhs.decls
             →[@name ++ " already declared"]
          _  →[]
```

Block: lazy multipass behavior

Default AG code generation to Haskell

```
type T_Stat  = Env→Env→(Env, Errs)
type T_Stats = T_Stat
sem_Stat_Block :: T_Stats→T_Stat
sem_Stat_Block stats_ =
   (λ_lhsIdecls _lhsIenv→
      (let (_statsIdecls, _statsIerrs) =        -- cyclic!
               stats_ _lhsIenv _statsIdecls
       in  (_lhsIdecls, _statsIerrs)))
```

Multipass behavior hidden inside lazy scheduling

Block: strict multipass behavior

`uuagc -O` orders (and strictifies) attribute evaluation

```
type T_Stat   = Env→(Env, T_Stat_1) -- pass1 returns pass2
type T_Stat_1 = Env→(Errs)           -- pass2
sem_Stat_Block :: T_Stats→T_Stat
sem_Stat_Block stats_ =
   (λ_lhsIdecls→
      let sem_Stat_Block_1 :: T_Stat_1
          sem_Stat_Block_1 =
             (λ_lhsIenv→
                (case stats_ (enter _lhsIenv) of -- nested multipass
                   {(_statsIdecls, stats_1)→     -- not cyclic!
                        stats_1 _statsIdecls}))
      in  (_lhsIdecls, sem_Stat_Block_1))
```

Block: declaring and using identifiers

Auxiliary datastructures for code generation

```
data Instr
    = Enter Int Int -- enter new block; level and nr of idents alloc
    | Leave Int     -- exit block; with level
    | Ref   Ref     -- refer to (level,disp)
type Code = [Instr]
```

Env utilities

```
top :: Env→[String]
top = head
```

Block: declaring and using identifiers

AG for code generation

```
attr Stat Stats Root syn code use {++} {[]} :: Code
sem Stat
    | Use   lhs.code  = [Ref @ref]
    | Block loc.level = level @stats.decls
               .alloc = length $ top @stats.decls
            lhs.code  = [Enter @level @alloc] ++
                        @stats.code ++
                        [Leave @level]
```

Including error messages in pretty printed output

- In the example we have shown the list of error messages, and then the pretty printed output.
- Note that changing this to include the error messages in the pretty printing is trivial
- Since some error messages show up the first traversal of the block and some in the second this becomes a nightmare when having to program this explicitly!

2. Parsing

2.1 What are parser combinators

What are parser combinators

- a collection basic parsing functions that recognise a piece of input
- a collection of combinators that build new parsers out of existing ones

What are parser combinators

- a collection basic parsing functions that recognise a piece of input
- a collection of combinators that build new parsers out of existing ones

Hackage provides a myriad of parser combinator libraries. here we will concentrate on the uu – parsinglib and show some of its strong points.

2.2 Elementary Combinators

Elementary Parsers

- Most libraries at least provide an *Applicative* interface taking care of sequencing and an Alternative interface taking care a composing alternatives.
- The actual implementation of the basic parsers is quite intricate, but is of no concern to the user

Types of the Elementary Combinators

Types

$\langle|\rangle$ :: Parser s a $\rightarrow$ Parser s a $\rightarrow$ Parser s a

Try to remember these types. Knowing the types is half the work when programming in Haskell.

Types of the Elementary Combinators

Types

```
⟨|⟩    :: Parser s a       →Parser s a→Parser s a
⟨*⟩    :: Parser s (b→a)→Parser s b→Parser s a
```

Try to remember these types. Knowing the types is half the work when programming in Haskell.

Types of the Elementary Combinators

Types

```
⟨|⟩    :: Parser s a       → Parser s a → Parser s a
⟨*⟩    :: Parser s (b → a) → Parser s b → Parser s a
pSym   :: s                             → Parser s s
```

Try to remember these types. Knowing the types is half the work when programming in Haskell.

Types of the Elementary Combinators

Types

```
⟨|⟩             :: Parser s a      →Parser s a→Parser s a
⟨∗⟩             :: Parser s (b→a)→Parser s b→Parser s a
pSym            :: s                          →Parser s s
pSucceed, pure  :: a                          →Parser s a
```

Try to remember these types. Knowing the types is half the work when programming in Haskell.

Types of the Elementary Combinators

Types

```
 ⟨|⟩           :: Parser s a      →Parser s a→Parser s a
 ⟨*⟩           :: Parser s (b→a)→Parser s b→Parser s a
pSym           :: s                         →Parser s s
pSucceed, pure :: a                         →Parser s a
pFail, empty   ::                            Parser s a
```

Try to remember these types. Knowing the types is half the work when programming in Haskell.

Computing a Result

The question which arises now is how do we get something useful out of such parsers?

Computing a Result

The question which arises now is how do we get something useful out of such parsers?

We recognize a character `'B'`:

```
pSym 'B'
```

Computing a Result

The question which arises now is how do we get something useful out of such parsers?

We recognize a character `'B'`:

```
                          pSym 'B'
```

Preceded by the recognition of a character `'A'`

```
                 pSym 'A'    pSym 'B'
```

Computing a Result

The question which arises now is how do we get something useful out of such parsers?

We recognize a character `'B'`:

```
                          pSym 'B'
```

Preceded by the recognition of a character `'A'`

```
              pSym 'A'    pSym 'B'
```

We now insert a dummy parser that returns the function (,):

```
pSucceed (,)  pSym 'A'    pSym 'B'
```

Computing a Result

The question which arises now is how do we get something useful out of such parsers?

We recognize a character `'B'`:

```
                                  pSym 'B'
```

Preceded by the recognition of a character `'A'`

```
                       pSym 'A'   pSym 'B'
```

We now insert a dummy parser that returns the function (,):

```
       pSucceed (,)    pSym 'A'   pSym 'B'
```

Combine the result using sequential composition of parsers:

```
pAB =  pSucceed (,) ⟨∗⟩ pSym 'A' ⟨∗⟩ pSym 'B'
```

Capturing the essence of Applicative

Suppose we want to deal with possibly failing notations and stay as closely as possible to the original notation; how to we deal with functions applications like e_1 e_2.

- both the function part e_1 and the argument part e_2 can fail to compute something
- we model this with a *Maybe*
- so we want to **"apply"** a $Maybe\ (b \to a)$ to a $Maybe\ b$, and produce a $Maybe\ a$

```
func ‘applyTo‘ arg = case func of
                       Just b2a→case arg of
                                 Just b   →Just (b2a b)
                                 Nothing→Nothing
                       Nothing→ Nothing
```

Capturing the essence of Applicative (Cont)

We capture this pattern as follows:

```
class Applicative p where
  ( ⟨∗⟩ ) :: p (b→a)→p b→p a
  pure    :: a           →p a
  ( ⟨$⟩ ) :: (b→a)   →p b→p a
  f ⟨$⟩ p = pure f ⟨∗⟩ p
  ...
instance Applicative Maybe where
  Just f ⟨∗⟩ Just v = Just (f v)
  _      ⟨∗⟩ _      = Nothing
```

Capturing the essence of Applicative (Cont)

If we now write:

> f ⟨∗⟩ a_1 ⟨∗⟩ a_2 ⟨∗⟩ a_3

we have **"overloaded"** the original implicit function applications in f a_1 a_2 a_3.

Capturing the essence of Applicative (Cont)

If we now write:

> f $\langle * \rangle$ a_1 $\langle * \rangle$ a_2 $\langle * \rangle$ a_3

we have **"overloaded"** the original implicit function applications in f a_1 a_2 a_3.

Conclusion:

Instead applying a value of type $b \rightarrow a$ to a value of type b to result in a value of type a the operator $\langle * \rangle$ applies a p-value **labelled with** type $b \rightarrow a$ to a p-value **labelled with** type b to result in a p-value **labelled with** type a.

Advice

The essential difference is that when using the class *Applicative* we abstain from the possibility to refer to the f-value in the second binding of the **do**-construct.

Advice

The essential difference is that when using the class *Applicative* we abstain from the possibility to refer to the f-value in the second binding of the **do**-construct.

Applicative is to be preferred over *Monad*, since it allows optimisations; the second part is independent of the first part and can thus be evaluated **”more statically”**, or even analysed independent of the run of the program!

Alternative

The companion class for *Applicative* is Alternative:

```
class Alternative m where
   ( ⟨|⟩ ) :: m a→m a→m a
   empty ::           m a
instance Alternative Maybe where
   Just l ⟨|⟩ _ = Just l
   _      ⟨|⟩ r = r
   empty        = Nothing
```

Alternative

The companion class for *Applicative* is Alternative:

```
class Alternative m where
   ( ⟨|⟩ ) :: m a→m a→m a
   empty ::          m a
instance Alternative Maybe where
   Just l ⟨|⟩ _ = Just l
   _      ⟨|⟩ r = r
   empty        = Nothing
```

Attention: For the **instance** Alternative (Parser s) the value empty is not the parser which recognises the empty string, but the parser that always fails!

2.3 Developing an Embedded Domain Specific Language

Useful functions I

Because the pattern:

$$\text{pSucceed f} \langle * \rangle \text{ p}$$

occurs so often

Useful functions I

Because the pattern:

$$\mathsf{pSucceed}\ f\ \langle * \rangle\ p$$

occurs so often we define

$\langle \$ \rangle$

```
f <$> p = pSucceed f <*> p
```

Useful functions I

Because the pattern:

$$\mathsf{pSucceed}\ f \ \langle\ast\rangle\ p$$

occurs so often we define

⟨$⟩

```
f ⟨$⟩ p = pSucceed f ⟨∗⟩ p
```

so we can write the previous function as:

```
pAB = (,) ⟨$⟩ pSym 'A' ⟨∗⟩ pSym 'B'
```

Useful functions II

Often we are not interested in parts of what we have recognized:

```
semIfStat cond ifpart thenpart = ...
pIfStat = (λ_ c _ t _ e _   → semIfStat c t e)
          ⟨$⟩ pIfToken     ⟨∗⟩ pExpr
          ⟨∗⟩ pThenToken   ⟨∗⟩ pExpr
          ⟨∗⟩ pElseToken   ⟨∗⟩ pExpr
          ⟨∗⟩ pFiToken
```

Useful functions II

Often we are not interested in parts of what we have recognized:

```
semIfStat cond ifpart thenpart = ...
pIfStat = (λ_ c _ t _ e _   → semIfStat c t e)
              ⟨$⟩ pIfToken     ⟨*⟩ pExpr
              ⟨*⟩ pThenToken   ⟨*⟩ pExpr
              ⟨*⟩ pElseToken   ⟨*⟩ pExpr
              ⟨*⟩ pFiToken
```

We define

```
p ⟨* q = (λx _ → x) ⟨$⟩ p ⟨*⟩ q
p *⟩ q = (λ_ y → y) ⟨$⟩ p ⟨*⟩ q
f ⟨$ q = pSucceed f ⟨* q
```

Useful functions II

We define

```
p ⟨∗ q = (λx _ → x) ⟨$⟩ p ⟨∗⟩ q
p ∗⟩ q = (λ_ y → y) ⟨$⟩ p ⟨∗⟩ q
f ⟨$ q = pSucceed f ⟨∗ q
```

So we can now write:

```
pIfStat = semIfStat ⟨$ pIfToken    ⟨∗⟩ pExpr
                    ⟨∗ pThenToken  ⟨∗⟩ pExpr
                    ⟨∗ pElseToken  ⟨∗⟩ pExpr
                    ⟨∗ pFiToken
```

Useful functions II

We define

```
p ⟨* q = (λx _ → x) ⟨$⟩ p ⟨*⟩ q
p *⟩ q = (λ_ y → y) ⟨$⟩ p ⟨*⟩ q
f ⟨$ q = pSucceed f ⟨* q
```

So we can now write:

```
pIfStat = semIfStat ⟨$ pIfToken   ⟨*⟩ pExpr
                    ⟨* pThenToken ⟨*⟩ pExpr
                    ⟨* pElseToken ⟨*⟩ pExpr
                    ⟨* pFiToken
```

Functions like semIfStat are generated by the uuagc compiler.

EBNF extensions

```
infixl 2 opt
opt        :: Parser s a→a→Parser s a
p ‘opt‘ v = p ⟨|⟩ pSucceed v
```

In the library we have special greedy versions which choose the longer alternative.

EBNF extensions

```
infixl 2 opt
opt         :: Parser s a→a→Parser s a
p ‘opt‘ v = p ⟨|⟩ pSucceed v
```

```
pList     :: Parser s a→Parser s [a]
pList p = (:) ⟨$⟩ p ⟨*⟩ pList p ‘opt‘ []
```

In the library we have special greedy versions which choose the longer alternative.

Exercise

Write a function that recognises a sequence of balanced parentheses, (i.e. $(), (()), (()\ ())\ (), \ldots$, and computes the maximal nesting depth (here $1, 2, 2, \ldots$. The grammar describing this language is:

```
S→(S) S | .
```

```
pP = (max.(+1)) ⟨$ pSym '(' ⟨∗⟩ pP ⟨∗ pSym ')'
                    ⟨∗⟩ pP
       ‘opt‘
       0
```

Left Factorisation

It is not a good idea to have parsers that have alternatives starting with the same (sequence of) elements:

```
p =     f <$> q <*> r1
    <|> g <$> q <*> r2
```

Left Factorisation

It is not a good idea to have parsers that have alternatives starting with the same (sequence of) elements:

```
p =     f ⟨$⟩ q ⟨*⟩ r1
    ⟨|⟩ g ⟨$⟩ q ⟨*⟩ r2
```

So we define:

```
p ⟨**⟩ q :: Parser s b→Parser s (b→a)→Parser s a
p ⟨**⟩ q = (λpv qv → qv pv) ⟨$⟩ p ⟨*⟩ q
p ⟨??⟩ q :: Parser s a→Parser s (a→a)→Parser s a
p ⟨??⟩ q = p ⟨**⟩ (q ‘opt‘ id)
```

Left Factorisation

It is not a good idea to have parsers that have alternatives starting with the same (sequence of) elements:

```
p =    f ⟨$⟩ q ⟨*⟩ r1
   ⟨|⟩ g ⟨$⟩ q ⟨*⟩ r2
```

So we define:

```
p ⟨**⟩ q :: Parser s b→Parser s (b→a)→Parser s a
p ⟨**⟩ q = (λpv qv → qv pv) ⟨$⟩ p ⟨*⟩ q
p ⟨??⟩ q :: Parser s a→Parser s (a→a)→Parser s a
p ⟨??⟩ q = p ⟨**⟩ (q ‘opt‘ id)
```

So we can replace the above code by:

```
p = q ⟨**⟩ (flip f ⟨$⟩ r1 ⟨|⟩ flip g ⟨$⟩ r2)
flip f x y = f y x
```

If many of such situations arise one may resort to the use of a

Left-recursion

- many grammars are left recursive
- parser combinator libraries usually cannot handle left recursion
- using combinators from the library which capture common patterns left-recursion can usually be avoided

Operands chained by operators

```
pChainr :: Parser s (c→c→c)→Parser s c→Parser s c
pChainr sep p = p ⟨??⟩ (flip ⟨$⟩ sep ⟨∗⟩ pChainr sep p)
```

Operands chained by operators

```
pChainr :: Parser s (c→c→c)→Parser s c→Parser s c
pChainr sep p = p ⟨??⟩ (flip ⟨$⟩ sep ⟨*⟩ pChainr sep p)
```

```
pChainl :: Parser s (c→c→c)→Parser s c→Parser s c
pChainl op x = (f ⟨$⟩ x ⟨*⟩ pList (flip ⟨$⟩ op ⟨*⟩ x))
               where
               f x [ ] = x
               f x (func : rest) = f (func x) rest
```

Example: A complete pocket calculator

It is straightforward to construct a parser for expressions with several operator priorities:

```
operators     = [[('+', (+)), ('-', (−))],
                 [('*', (∗))], [('^', ())]]
same_prio ops = msum [op ⟨$ pSym c | (c, op) ← ops]
expr          = foldr pChainl (pNatural ⟨|⟩ pParens expr)
                              (map same_prio operators)
```

which we can call:

```
--> run expr "15-3*5+2^5"
Result: 32
```

Left Factorisation II

We want to recognise expressions with as result a value of the type:

```
data Expr = Lambda     Id         Expr
          | App        Expr       Expr
          | TypedExpr  TypeDescr  Expr
```

Left Factorisation II

We want to recognise expressions with as result a value of the type:

```
data Expr = Lambda    Id        Expr
          | App       Expr      Expr
          | TypedExpr TypeDescr Expr
```

```
pFactor = Lambda ⟨$ pSym '\\' ⟨∗⟩ pIdent
                 ⟨∗ pSym '.'  ⟨∗⟩ pExpr
        ⟨|⟩
          pParens '(' ')' pExpr
```

Left Factorisation II

We want to recognise expressions with as result a value of the type:

```
data Expr = Lambda    Id        Expr
          | App       Expr      Expr
          | TypedExpr TypeDescr Expr
```

```
pFactor = Lambda ⟨$ pSym '\\' ⟨∗⟩ pIdent
                 ⟨∗ pSym '.'  ⟨∗⟩ pExpr
          ⟨|⟩
          pParens '(' ')' pExpr
```

```
pExpr = pChainl (pSucceed App) pFactor
        ⟨??⟩ ( TypedExpr
             ⟨$ pTok "::"
             ⟨∗⟩ pTypeDescr)
```

2.4 Monadic Parsers

The Chomsky Hierarchy

The Chomsky hierarchy:

- Regular
- Context-free
- Context-sensitive
- Recursively enumerable

It is well known that context free grammars have limited expressibility.

Recognising Context Sensitive Grammars

```
times        :: Int→Parser s a→Parser s [a]
0 ‘times‘ p = pSucceed []
n ‘times‘ p = (:) ⟨$⟩ p ⟨*⟩ (n − 1) ‘times‘ p
```

Recognising Context Sensitive Grammars

```
times         :: Int→Parser s a→Parser s [a]
0 ‘times‘ p = pSucceed []
n ‘times‘ p = (:) ⟨$⟩ p ⟨∗⟩ (n − 1) ‘times‘ p
abc n         = n ⟨$ (n ‘times‘ a)
                  ⟨∗ (n ‘times‘ b)
                  ⟨∗ (n ‘times‘ c)
```

Recognising Context Sensitive Grammars

```
times       :: Int→Parser s a→Parser s [a]
0 ‘times‘ p = pSucceed [ ]
n ‘times‘ p = (:) ⟨$⟩ p ⟨*⟩ (n − 1) ‘times‘ p
abc n       = n ⟨$ (n ‘times‘ a)
                ⟨* (n ‘times‘ b)
                ⟨* (n ‘times‘ c)
ABC         = foldr ( ⟨|⟩ ) pFail [abc n | n ← 0 . .]
```

Recognising Context Sensitive Grammars

```
times         :: Int→Parser s a→Parser s [a]
0 ‘times‘ p = pSucceed []
n ‘times‘ p = (:) ⟨$⟩ p ⟨∗⟩ (n − 1) ‘times‘ p
abc n         = n ⟨$ (n ‘times‘ a)
                  ⟨∗ (n ‘times‘ b)
                  ⟨∗ (n ‘times‘ c)
ABC           = foldr ( ⟨|⟩ ) pFail [abc n | n ← 0 ..]
```

We admit that this is not very efficient, but left factorisation is not so easy since the corresponding context free grammar is infinite.

The Monadic Approach

Wouldn't it be nice if we could start by just recognising a sequence of a's, and then use the result to enforce the right number of b's and c's?

The Monadic Approach

Wouldn't it be nice if we could start by just recognising a sequence of a's, and then use the result to enforce the right number of b's and c's?

```
instance Monad (Parser s) where
    p (>>=) q = ...
    return v  = ...
```

The Monadic Approach

Wouldn't it be nice if we could start by just recognising a sequence of a's, and then use the result to enforce the right number of b's and c's?

```
instance Monad (Parser s) where
   p (>>=) q = ...
   return v  = ...
as           :: Parser Char Int
as           = length <$ pList (pSym 'a')
bc n         = n <$ (n `times` b) <* (n `times` c)
```

The Monadic Approach

Wouldn't it be nice if we could start by just recognising a sequence of a's, and then use the result to enforce the right number of b's and c's?

```
instance Monad (Parser s) where
   p (≫=) q = ...
   return v  = ...
as          :: Parser Char Int
as          = length ⟨$ pList (pSym 'a')
bc n        = n ⟨$ (n `times` b) ⟨* (n `times` c)
ABC         = do n ← as
                 bc n
```

2.5 Problems

Problems with Erroneous Input

- If your input does not conform to the language recognized by the parser all you may get as a result is: $[\,]$.
- It may take quite a while before you get this negative result, since the backtracking may try all other alternatives at all positions.
- There is no indication of where things went wrong.

Problems with Erroneous Input

- If your input does not conform to the language recognized by the parser all you may get as a result is: $[\,]$.
- It may take quite a while before you get this negative result, since the backtracking may try all other alternatives at all positions.
- There is no indication of where things went wrong.

These problem have been cured in both Parsec and the UUParsing-library. The latter does this:

- without much overhead
- without need for help from the programmer
- without stopping, so many errors can be found in a single run

Problems with Space Consumption

The naïve “List of successes” implementations which are often used have further drawbacks:

- The complete input has to be parsed before any result is returned
- The complete input is present in memory as long as no parse has been found
- Efficiency may depend critically on the ordering of the alternatives, and thus on how the grammar was written

For all of these problems we have found solutions in the **uu-parsinglib** package.

Error correction at work

The parser pA recognises a single letter `’a’`, etc.:

```
--> run pa  "b"
 Result: "a"
 Correcting steps:
   Deleted   ’b’ at... expecting ’a’
   Inserted  ’a’ at... expecting ’a’
--> run ((++) <$> pa <*> pa)  "bbab"
 Result: "aa"
 Correcting steps:
   Deleted   ’b’ at ... expecting ’a’
   Deleted   ’b’ at ... expecting ’a’
   Deleted   ’b’ at ... expecting ’a’
   Inserted  ’a’ at ... expecting ’a’
```

Error correction at work for Monads

Error correction also works in the presence of monadic constructs:

```
--> run (do  l <- pCount pa; pExact l pb)  "aaacabbbbb"
 Result: ["b","b","b","b"]
 Correcting steps:
   Deleted   'c' at ... expecting one of ['b', 'a']
   The token 'b' was not consumed by the parsing process.
```

Refining error messages

We can replace the expected elements in an error message by a custom error message:

```
--> run (pa <|> pb <?> "justamessage")  "c"
 Result: "b"
 Correcting steps:
   Deleted   'c' at ... expecting justamessage
   Inserted  'b' at ... expecting 'b'
```

Running ambiguous parsers

We can have ambiguous parsers, provided we indicate so:

```
run (amb (pEither  parseIntString  pIntList))
    "(123;456;789)"
 Result: [Left ["123","456","789"],Right [123,456,789]]
```

Disambiguation

Internally the parser uses a cost model. Disambiguation can be acheived by inserting small costs at less preferable alternatives:

```
ident :: Parser String
ident = ((:) ⟨$⟩ pSym ('a', 'z')
                ⟨*⟩ pMunch (λx→'a' ⩽ x ∧ x ⩽ 'z') `micro` 1) ⟨* spa
idents = pList1 ident
pKey keyw = pToken keyw `micro` 0 ⟨* spaces
spaces :: Parser String
spaces = pMunch (== ' ')
preferres_second_alt =
   pList ident
    <||> (λc t e→["IfThenElse"] ++ c ++ t ++ e)
         ⟨$ pKey "if"     ⟨*⟩ pList_ng ident
         ⟨* pKey "then"   ⟨*⟩ pList_ng ident
         ⟨* pKey "else"   ⟨*⟩ pList_ng ident
```

Result

If the input starts with an "if" the second alternative is chosen:

```
-->run preferres_second_alt "if a then if else c"
Result: ["IfThenElse","a","if","c"]
-->run preferres_second_alt "ifx a then if else c"
Result: ["ifx","a","then","if", "else","c"]
```

Some healthiness checks are performed

The library performs a mild form of abstract interpretation which captures some errors which may otherwise be very hard to find:

```
--> run (pList spaces) ""
 Result: *** Exception: The combinator pList
  requires that it’s argument cannot recognise
  the empty string
```

Dealing with errors

During the parsing process we may ask for the error messages which were generated since the last time they were asked for. The following parses a BibTeX file and ignores the items which contain errors:

```
pBibTexFile = pList (process ⟨$⟩ pBibTeXItem ⟨*⟩ getErrors)
process item [] = Left (processItem item)
process _     l = Right l
```

Using the library

The library has many tuning facilities, but:

- tuning is normally not needed
- insertion costs of elements can be changed (increase!! for unwanted alternatives)
- you can add your own basic parsers; see the module BasicInstances for examples

Not covered

- permuting parsers
- merging parsers
- managing internal state

Questions

Questions?