Expand: Towards an extensible Pandoc system

An application of extensible compiler construction in Haskell

Jacco Krijnen Doaitse Swierstra Marcos Viera

Department of Computer Science, Utrecht University
Utrecht, The Netherlands

Instituto de Computación, Universidad de la República
Montevideo, Uruguay

January 4, 2014

Universiteit Utrecht

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

Introduction

We can write documents using different markup languages:

- Latex (`.tex`)
- markdown (`.md`)
- Word (`.doc`)
- mediawiki
- ...

Usage might depend on syntactic preference, language features or purpose.

Introduction

All formats have their own way of describing document elements.

document element	HTML	Latex	Markdown
bold text	`<b>...</b>`	`\textbf{...}`	`**...**`
header	`<h2>...</h2>`	`\subsection{...}`	`##...`
paragraph	`<p>...</p>`	Blank lines	
url	`<a href="...">...</a>`	`\url{...}{...}`	`[...](...)`

Introduction

Problem: We have written a document in one language and would like to convert it to another, without doing all the formatting again.

Introduction (Pandoc)

Pandoc is open-source software that allows for automatic conversion between different types of markup formats.

- Supports many types of input and output formats
- Also provides its functionality as a Haskell library

Introduction (Pandoc)

Pandoc is open-source software that allows for automatic conversion between different types of markup formats.

- Supports many types of input and output formats
- Also provides its functionality as a Haskell library
- Architecture:

  Readers that can parse specific markup file formats

  Abstract Syntax Tree an intermediate data structure to describe a general markup document

  Writers that can unparse (or write) to a target markup file format

Introduction (Problem)

Unfortunately, Pandoc's readers, intermediate type and writers are not extensible.

We propose `expand`, a Haskell library, in which all of the three components can be described in a modular way, with the possibility to extend and reuse different components.

Contents

Extensibility

Implementing `expand`: Core Latex to HTML

- Abstract Syntax
- Grammar - Core Latex
- Semantics - HTML
- Composing the Tool

Extending components

- Numbered Headers
- Table of Contents

Conclusion

1. Extensibility

Extensibility

Suppose you want to extend the formatting of markdown.

Extensibility

Suppose you want to extend the formatting of markdown.

One should be able to

Extensibility

Suppose you want to extend the formatting of markdown.

One should be able to

1. import an existing reader, writer or the definition of the intermediate data type

Extensibility

Suppose you want to extend the formatting of markdown.

One should be able to

1. import an existing reader, writer or the definition of the intermediate data type
2. write extensions that build on these components

Extensibility

Suppose you want to extend the formatting of markdown.

One should be able to

1. import an existing reader, writer or the definition of the intermediate data type
2. write extensions that build on these components
3. combine the extensions with the existing components

Extensibility

Suppose you want to extend the formatting of markdown.

One should be able to

1. import an existing reader, writer or the definition of the intermediate data type
2. write extensions that build on these components
3. combine the extensions with the existing components

Furthermore, the Haskell type system should verify that the composition is

Extensibility

We base our solution on the following Haskell libraries

- `murder` for defining parser using grammar fragments
- `AspectAG` for describing semantics using attribute grammars rules.

(Viera, M.: First Class Syntax, Semantics and Their Composition. Ph.D. thesis)

2. Implementing `expand`: Core Latex to HTML

Example input & output


```
\section{\plain{Introduction}}
\begin{paragraph}
    \plain{Iam id ipsum absurdum, \textbf{\plain{maximum}}
           malum neglegi. Ut id aliis narrare gestiant? }
\end{paragraph}

\subsection{\plain{subintroduction}}
\begin{paragraph}
    \plain{Sic enim censent, oportunitatis esse beate vivere. A primo,
           ut opinor, animantium ortu petitur origo summi boni. }
\end{paragraph}
```

Example input & output


```
\section{\plain{Introduction}}
\begin{paragraph}
    \plain{Iam id ipsum absurdum, \textbf{\plain{maximum}}
           malum neglegi. Ut id aliis narrare gestiant? }
\end{paragraph}

\subsection{\plain{subintroduction}}
\begin{paragraph}
    \plain{Sic enim censent, oportunitatis esse beate vivere. A primo,
           ut opinor, animantium ortu petitur origo summi boni. }
\end{paragraph}
```

```
<h1>Introduction</h1>
<p>Iam id ipsum absurdum, <b>maximum</b>neglegi. Ut id
aliis narrare gestiant? </p>

<h2>subintroduction</h2>
<p>Sic enim censent, oportunitatis esse beate vivere. A
primo, ut opinor, animantium ortu petitur origo summi
boni. </p>
```

2.1 Abstract Syntax

```
module Declarations.CoreLatex where

data Document = Document { blocks :: BlockL } deriving Show
```

Figure: A datatype for the AS

```
module Declarations.CoreLatex where

data Document = Document { blocks :: BlockL } deriving Show

type BlockL = [Block]
data Block  = Header    { level_header  :: Int
                        , inlines_header :: InlineL }
            | Paragraph { inlines_par   :: InlineL }
    deriving (Show)
```

Figure: A datatype for the AS

```
module Declarations.CoreLatex where

data Document = Document {blocks :: BlockL} deriving Show

type BlockL = [Block]
data Block  = Header    {level_header   :: Int
                        ,inlines_header :: InlineL}
            | Paragraph {inlines_par    :: InlineL}
    deriving (Show)
type InlineL = [Inline]
data Inline = Plain     {str_plainInl   :: String}
            | Bold      {inlines_boldInl :: InlineL}
            | Italics   {inlines_italInl :: InlineL}
  deriving (Show)
```

Figure: A datatype for the AS

```
module Declarations.CoreLatex where

data Document = Document {blocks :: BlockL} deriving Show


type BlockL = [Block]
data Block  = Header     {level_header   :: Int
                         ,inlines_header :: InlineL}
            | Paragraph  {inlines_par    :: InlineL}
    deriving (Show)
type InlineL = [Inline]
data Inline  = Plain      {str_plainInl    :: String}
             | Bold       {inlines_boldInl :: InlineL}
             | Italics    {inlines_italInl :: InlineL}
  deriving (Show)
$(deriveAG ''Document)
$(deriveLang "Doc" [''Document, ''BlockL, ''Block,
                    ''InlineL, ''Inline])
```

Figure: A datatype for the AS

```
$ (deriveAG ‘Document)
```

Generates the necessary labels and types to be used in the attribute grammar fragments.

```
$ (deriveLang "Doc" [‘Document, ‘BlockL, ‘Block,
                                ‘InlineL, ‘Inline])
```

Generates a record with the appropriate semantic function types, to be used by the parser.

2.2 Grammar - Core Latex

Core Latex (EBNF)

document ::= *block**
block ::= *paragraph*
| *header*
paragraph ::= "\begin" "{" "paragraph" "}" *inline**
"\end" "{" "paragraph" "}"
header ::= "\section" "{" *inline** "}"
| "\subsection" "{" *inline** "}"
| "\subsubsection" "{" *inline** "}"
inline ::= "\plain" "{" *text* "}"
| "\textbf" "{" *inline** "}"
| "\textit" "{" *inline** "}"

Figure: The EBNF for our input language

Core Latex (Haskell)


```
module Grammars.CoreLatex where

import Declarations.CoreLatex

gLatex sem = proc () → do
  rec
      doc    ← addNT ≺ ⟦ (pDocument sem) blocks ⟧
    blocks   ← addNT ≺ pFoldr (pBlockL_Cons sem, pBlockL_Nil sem)
                              ⟦ block ⟧
    block    ← addNT ≺ ⟦ head ⟧ <|> ⟦ par ⟧

      ...
```

Each production is expressed using the idiom brackets[1] ⟦ and ⟧
(`iI` and `Ii` in Haskell code)

[1] http://www.haskell.org/haskellwiki/Idiom_brackets

Core Latex (Haskell)


```
  ...
  par  ← addNT ≺ ⟦ (pParagraph sem)
                   "\\begin" "{" "paragraph" "}"
                   inls
                   "\\end" "{" "paragraph" "}" ⟧
  head ← addNT ≺ let h (x, name) = ⟦ (pHeader sem x) "\\" name
                                                     "{" inls
                                                     "}" ⟧
                     headers      = [(1, "section")
                                    ,(2, "subsection")
                                    ,(3, "subsubsection")]
                 in  foldr1 (<|>) (map h headers)
  ...
```

Figure: Our EBNF encoded as a series of grammar transformations

Core Latex (Haskell)


```
...
inls ← addNT ≺ pFoldr (pInlineL_Cons sem, pInlineL_Nil sem)
                      ||| inline |||
inl  ← addNT ≺ ||| (pBold sem)   "\\textbf" "{" inlineL "}" |||
           <|> ||| (pItalics sem) "\\textit" "{" inlineL "}" |||
           <|> ||| (pPlain sem)   "\\plain"  "{"
                                  (someExcept "\\&%$#_{}~^")
                                  "}" |||
exportNTs ≺ exportList document ( export cs_document doc
                                ∘ export cs_blockL    blocks
                                ∘ export cs_paragraph par
                                ∘ export cs_header    head
                                ∘ export cs_inline    inl
                                ∘ export cs_inlineL   inls)
```

Figure: Our EBNF encoded as a series of grammar transformations

2.3 Semantics - HTML

Rules

We want to compute a *String* from the abstract syntax tree: the HTML text. Therefore we introduce a synthesized attribute *html*.

```
module Semantics.HTML where
import Declarations.CoreLatex
$ (attLabels ["html"])
```

Using *AspectAG*, we can now define a rule per data constructor, describing how to compute this *String*.

```
document_html  = ...
blockLnil_html  = ...
blockLcons_html = ...
header_html    = syn html $
                   do level ← at ch_level_header
                      inls  ← at ch_inlines_header
                      return $ "<h" ++ show level ++ ">"
                               ++ inls # html
                               ++ "</h" ++ show level ++ ">"
                               ++ "\n"
bold_html      = syn html $
                   do inls ← at ch_inlines_boldInl
                      return $ "<b>"
                               ++ inls # html
                               ++ "</b>"
        ...
```

We can now generate the record of semantic functions

```
semHtml = mkDoc blockLcons_html
                blockLnil_html
                ...
                paragraph_html
                plain_html
```

to be used by the parsers:

```
module Grammars.CoreLatex where

gLatex sem = proc () → do
  rec
    doc    ← addNT ≺ ⟦ (pDocument sem) blocks ⟧
    blocks ← addNT ≺ pFoldr (pBlockL_Cons sem, pBlockL_Nil sem)
                            ⟦ block ⟧
    block  ← addNT ≺ ⟦ head ⟧ <|> ⟦ par ⟧
    par    ← addNT ≺ ⟦ (pParagraph sem)
                       "\\begin" "{" "paragraph" "}"
    ...
```

- We use a deforestated approach, i.e. the Tree never comes into existence

2.4 Composing the Tool

Composing the tool


```
module Converter where

import Grammars.CoreLatex (gLatex)
import Semantics.HTML (semHTML, html)
import Expand.Utils (buildConverter)

latex2html :: String → String
latex2html = buildConverter (gLatex semHtml) html
```

3. Extending components

3.1 Numbered Headers

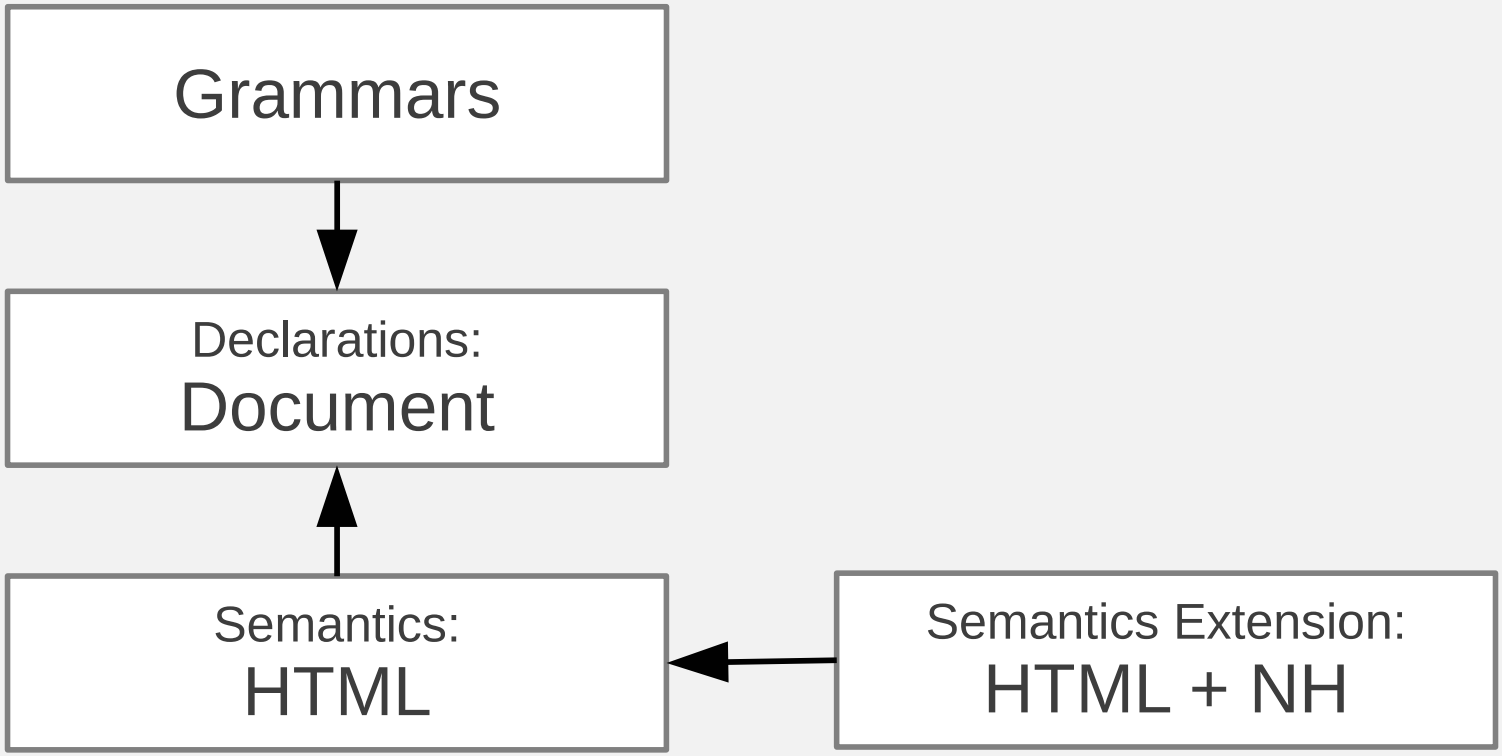
Numbered Headers

§3.1

We want to extend the HTML generation in a way that headers are numbered.

Numbered Headers

We want to extend the HTML generation in a way that headers are numbered.



Numbered Headers

We introduce a chained attribute *cHeaderNum*, which is threaded through the tree in a *State Monad* like fashion. We keep a local copy of the value in every header by introducing the *headerNum* attribute (so that it is available for future extension).

```
module Semantics.General.NumberedHeaders where

import Declarations.CoreLatex

$ (attLabels ["cHeaderNum", "headerNum"])
```

Numbered Headers

We introduce a chained attribute *cHeaderNum*, which is threaded through the tree in a *State Monad* like fashion. We keep a local copy of the value in every header by introducing the *headerNum* attribute (so that it is available for future extension).

```
module Semantics.General.NumberedHeaders where

import Declarations.CoreLatex

$ (attLabels ["cHeaderNum", "headerNum"])
```

We will model the header number as a $[Int]$, and introduce two utility functions.

```
updateHeaderNum :: Int → [Int] → [Int]
updateHeaderNum level par = zipWith (+) par' (zeros ++ [1])
  where par'  = par ++ repeat 0
        zeros = replicate (level − 1) 0

formatHeaderNum :: [Int] → String
formatHeaderNum = intercalate "." ∘ map show
```

Numbered Headers


```
cHeaderNum_NTs = nt_BlockL .*. nt_Block .*. hNil

default_cHeaderNum     = chain cHeaderNum cHeaderNum_NTs

document_cHeaderNum = inh cHeaderNum cHeaderNum_NTs $
  do return (ch_blocks .=. ([] :: [Int]) .*. emptyRecord)

header_headerNum       = loc headerNum $
  do lhs     ← at lhs
     level   ← at ch_level_header
     return $ updateHeaderNum level (lhs # cHeaderNum)

header_cHeaderNum      = syn cHeaderNum $
  do loc     ← at loc
     return $ loc # headerNum
```

Note that the attribute rules are independent of the target language.

Numbered Headers

We still have to change the HTML output. We use the *synmodM* function to define a modification of a rule.

```
module Semantics.HTML.NumberedHeaders where

import Declarations.HTML
import Semantics.HTML

header_html' = synmodM html $
  do level ← at ch_level_header
     inls  ← at ch_inlines_header
     loc← at loc
     let num = loc # headerNum
     return $ "<h" ++ show level ++ ">"
              ++formatNH num ++ "  "
              ++ inls # html
              ++ "</h" ++ show level ++ ">" ++ "\n"
```

Numbered Headers

We can now construct a new semantic record for *html* generation by combining both the *html* and the *cHeaderNum* aspects:

```
semHtml' = mkDoc
  (default_cHeaderNum     `ext` blockLcons_html)
  (default_cHeaderNum     `ext` blockLnil_html)
  bold_html
  (document_cHeaderNum `ext` document_html)
  (header_headerNum       `ext` header_cHeaderNum
                          `ext` header_html'
                          `ext` header_html)
  inlineLcons_html
  inlineLnil_html
  italics_html
  (default_cHeaderNum     `ext` paragraph_html)
  plain_html
```

Numbered Headers

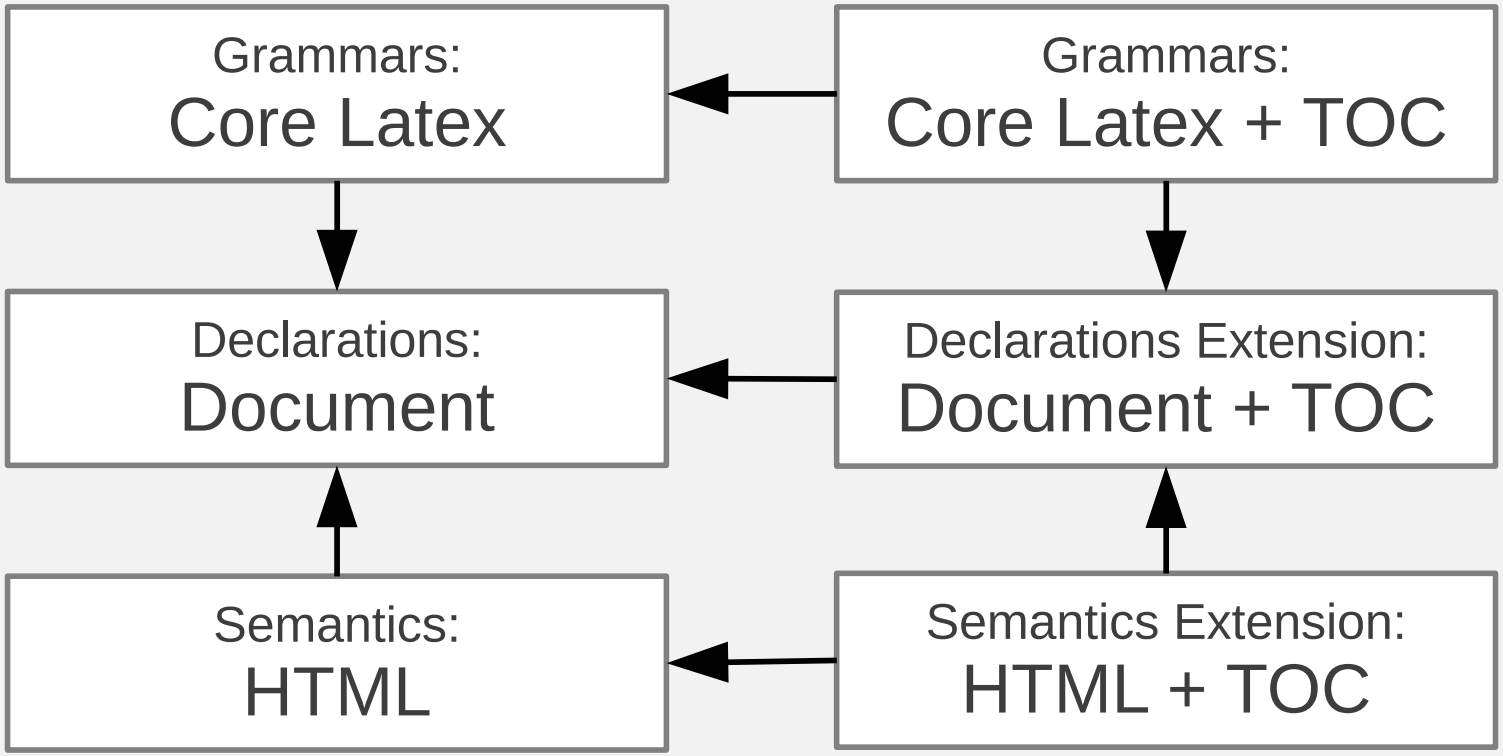
Our alternative converter now becomes:

$$latex2html' :: String \to String$$
$$latex2html' = buildConverter\ (gLatex\ semHtml')\ html$$

3.2 Table of Contents

Table of Contents

A table of contents requires extensions in all three components.



The abstract syntax needs support for a TOC node.

```
module Declarations.CoreLatex.Toc

import Declarations.CoreLatex

data EXT_Block = Toc

$(extendAG '' EXT_Block [])
$(deriveLang "DocToc" ['' EXT_Block])
```

The function *deriveLang* will also produce a new record type containing the semantic function of the *Toc* production.

Table of Contents

Our latex language needs an additional construct:

$$
\begin{array}{lll}
\textit{document} & ::= & \textit{block} * \\
\textit{block} & ::= & \textit{paragraph} \\
 & | & \textit{header} \\
 & & \\
\textit{paragraph} & ::= & \texttt{"\textbackslash begin"}\ \ \texttt{"\{"}\ \texttt{"paragraph"}\ \texttt{"\}"}\ \textit{inline} * \\
 & & \texttt{"\textbackslash end"}\ \texttt{"\{"}\ \texttt{"paragraph"}\ \texttt{"\}"} \\
\textit{header} & ::= & \texttt{"\textbackslash section"}\ \ \ \ \ \ \texttt{"\{"}\ \textit{inline} * \ \texttt{"\}"} \\
 & | & \texttt{"\textbackslash subsection"}\ \ \ \texttt{"\{"}\ \textit{inline} * \ \texttt{"\}"} \\
 & | & \texttt{"\textbackslash subsubsection"}\ \texttt{"\{"}\ \textit{inline} * \ \texttt{"\}"} \\
\textit{inline} & ::= & \texttt{"\textbackslash plain"}\ \ \texttt{"\{"}\ \textit{text}\ \ \ \texttt{"\}"} \\
 & | & \texttt{"\textbackslash textbf"}\ \texttt{"\{"}\ \textit{inline} * \ \texttt{"\}"} \\
 & | & \texttt{"\textbackslash textit"}\ \texttt{"\{"}\ \textit{inline} * \ \texttt{"\}"}
\end{array}
$$

Table of Contents

Our latex language needs an additional construct:

$$
\begin{array}{lll}
\textit{document} & ::= & \textit{block} * \\
\textit{block} & ::= & \textit{paragraph} \\
 & | & \textit{header} \\
 & | & \texttt{"\textbackslash tableofcontents"} \\
\textit{paragraph} & ::= & \texttt{"\textbackslash begin"}\ \ \texttt{"\{"}\ \texttt{"paragraph"}\ \texttt{"\}"}\ \textit{inline} * \\
 & & \texttt{"\textbackslash end"}\ \texttt{"\{"}\ \texttt{"paragraph"}\ \texttt{"\}"} \\
\textit{header} & ::= & \texttt{"\textbackslash section"}\ \ \ \ \ \ \texttt{"\{"}\ \textit{inline} *\ \texttt{"\}"} \\
 & | & \texttt{"\textbackslash subsection"}\ \ \ \texttt{"\{"}\ \textit{inline} *\ \texttt{"\}"} \\
 & | & \texttt{"\textbackslash subsubsection"}\ \texttt{"\{"}\ \textit{inline} *\ \texttt{"\}"} \\
\textit{inline} & ::= & \texttt{"\textbackslash plain"}\ \ \texttt{"\{"}\ \textit{text}\ \ \ \texttt{"\}"} \\
 & | & \texttt{"\textbackslash textbf"}\ \texttt{"\{"}\ \textit{inline} *\ \texttt{"\}"} \\
 & | & \texttt{"\textbackslash textit"}\ \texttt{"\{"}\ \textit{inline} *\ \texttt{"\}"}
\end{array}
$$

Table of Contents

Since the abstract syntax tree now supports a table of contents, we can extend the grammar for the latex-like language:

```
module Grammars.CoreLatex.Toc

import Grammars.CoreLatex
import Declarations.CoreLatex
import Declarations.CoreLatex.Toc

gLatexToc sem = proc imported →
  do
    let block = getNT cs_block imported
    addProds ≺ (block, ⟦ (pToc sem) "\\tableofcontents" ⟧)
    exportNTs ≺ imported
```

Table of Contents

We model the table of contents as a [([Int], String)]

```
module Semantics.General.Toc where
import Declarations.CoreLatex
$ (attLabels ["sToc","toc"])
sToc_NTs = nt_Document .*. nt_Block .*. nt_BlockL .*. HNil
default_sToc = use sToc sToc_NTs (++) []
header_sToc = syn sToc $ do loc <- at loc
                            inls <- at ch_inlines_header
                            return [(loc # headerNum,
                                     inls # sInlStr)]
```

Table of Contents

Using the inherited attribute *toc* we can now format it using an appropriate *html* rule.

```
module Semantics.HTML.Toc where

import Semantics.General.Toc

toc_html = syn html $ do lhs ← at lhs
                         return $ formatToc (lhs # toc)

formatToc :: [([Int], String)] → String
formatToc = foldr f ""
  where f (x, section) table = "<a href=#" ++ show x ++ ">"
                             ++ (formatNH x) ++ "  " ++ section
                             ++ "</a><br />\n" ++ table
```

The *html* formatting of a header is changed so that it contains an `id` attribute (omitted)

We now have all the building blocks to create the new conversion tool:

```
latex2html'' :: String → String
latex2html'' = buildConverter   (  gLatex      semHtml''
                                +>> gLatexToc  semHtmlToc) html
```

Example input


```
\tableofcontents
\section{\plain{Introduction}}
\begin{paragraph}
    \plain{Iam id ipsum absurdum, \textbf{\plain{maximum}}
           malum neglegi. Ut id aliis narrare gestiant? }
\end{paragraph}

\subsection{\plain{subintroduction}}
\begin{paragraph} ...
```

Example input


```
\tableofcontents
\section{\plain{Introduction}}
\begin{paragraph}
    \plain{Iam id ipsum absurdum, \textbf{\plain{maximum}}
           malum neglegi. Ut id aliis narrare gestiant? }
\end{paragraph}

\subsection{\plain{subintroduction}}
\begin{paragraph} ...
```

```
<a href=#[1]>1 Introduction</a><br />
<a href=#[1,1]>1.1 subintroduction</a><br />
<a href=#[2]>2 Conclusion</a><br />

<h1 id="[1]">1 Introduction</h1>
<p>Iam id ipsum absurdum, <b>maximum</b>neglegi. Ut id
aliis narrare gestiant? </p>

<h2 id="[1,1]">1.1 subintroduction</h2>
<p> ...
```

4. Conclusion

Conclusion

- We have shown how to build an extensible document formatting system using the `murder` and `AspectAG` libraries.

Conclusion

- We have shown how to build an extensible document formatting system using the `murder` and `AspectAG` libraries.
- The Haskell type system validates the composition of all the extensions
  - Grammar fragments state in their type which non terminals they require and expose
  - All usages of non-terminals are guaranteed to point to valid productions
  - AG rules state in their type which attributes should be present and which attribute they define

Conclusion

- We have shown how to build an extensible document formatting system using the `murder` and `AspectAG` libraries.
- The Haskell type system validates the composition of all the extensions
  - Grammar fragments state in their type which non terminals they require and expose
  - All usages of non-terminals are guaranteed to point to valid productions
  - AG rules state in their type which attributes should be present and which attribute they define
- By the modular nature of the components we can reuse aspects, like the Table of Contents.

All code can be found at `http://hackage.haskell.org/package/expand`.

```
header_sToc
  :: (HasField (Proxy (Ch_inlines_header, [Inline])) chi r,
      HasField (Proxy Att_headerNum) l t,
      HasField (Proxy Att_sInlStr) r t1,
      HExtend (Att (Proxy Att_sToc) [(t, t1)]) sp sp') ⇒
      Rule l ho chi par l1 ho1 ic sp l1 ho1 ic sp'

header_sToc = syn sToc $
  do loc ← at loc
     inls ← at ch_inlines_header
     return [(loc # headerNum, inls # sInlStr)]
```