Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Compiler Construction

WWW: `http://www.cs.uu.nl/wiki/Cco`

Edition 2010/2011

Agenda

Overview

Mini project A: BibTeX2HTML

Tools

Questions

1. Overview

Mini projects

§1

The lab work that is to be handed in consists of four mini projects.

Deadlines for these have been announced on the wiki. Each project takes about two weeks.

Library

Utility code for often occurring tasks in compiler construction will be made available through the wiki by means of a "Cabalised" Haskell library: `cco`.

This library comes with Haddock documentation and will be extended incrementally.

It relies on the `ansi-terminal` package, which is available from Hackage.

2. Mini project A: BibTeX2HTML

Mini project: BibTeX2HTML

BibTeX is a tool for generating bibliographies and including them in LaTeX-documents. Bibliographies are produced from bibliographic databases written in a domain-specific language.

The aim of this project is to implement a set of command-line tools that facilitate the rendering of BibTeX-databases in HTML.

BibTeX-database: example


```
@book{pierce02types,
  author    = "Pierce, Benjamin C.",
  title     = "Types and Programming Languages",
  publisher = "The MIT Press",
  address   = "Cambridge, Massachusetts",
  year      = 2002}

@inproceedings{loeh03dependency,
  author    = "L{\"o}h, Andres and Clarke, Dave and Jeuring,
                 Johan",
  title     = "Dependency-style {G}eneric {H}askell",
  editor    = "Runciman, Colin and Shivers, Olin",
  booktitle = "Proceedings of the Eighth ACM SIGPLAN
               International Conference on Functional
               Programming, ICFP 2003, Uppsala, Sweden,
               August 25--29, 2003",
  pages     = "141--152",
  publisher = "ACM Press",
  year      = 2003}
```

HTML-output: example


```
<html>
  <head><title>Bibliography</title></head>
  <body>
    <a href="loeh03dependency">[LCJ03]</a> |
    <a href="pierce02types">[P02]</a>
    <hr>
    <table border="0">
      <tr valign="top">
        <td><a name="loeh03dependency">[LCJ03]</a></td>
        <td>
          Andres L&ouml;h, Dave Clarke, and Johan
          Jeuring. Dependency-style Generic Haskell. In:
          Colin Runciman and Olin Shivers, editors,
          <em>Proceedings of the Eighth ACM SIGPLAN
          International Conference on Functional
          Programming, ICFP 2003, Uppsala, Sweden,
          August 25&ndash;29, 2003</em>, pages
          141&ndash;152. ACM Press, 2003.
        </td>
      </tr>
      <tr valign="top">
        <td><a name="pierce02types">[P02]</a></td>
        <td>
          Benjamin C. Pierce. <em>Types and Programming
          Languages</em>. The MIT Press, Cambridge,
          Massachusetts, 2002.
        </td>
      </tr>
    </table>
  </body>
</html>
```


BIBTEX-format

Detailed descriptions of the BIBTEX-format can be found on the web.

A full and faithful implementation of the format will need to deal with a lot of subtleties: variations in syntax, cross references, formatting of names, . . .

☞ You will probably not be able to implement all of these, but your implementation should support at least a reasonable subset of the format.

Validation

A BIBTEX-database consists of zero or more entries.
Each entry is of a specific type (`book`, `inproceedings`, . . .).

Each type comes with a number of required and optional fields (`author`, `publisher`, . . .).
Your implementation should check, for each entry, that all required fields are present and emit error messages if this check fails.
If fields are provided that are neither required nor optional for a specific entry type, warning messages should be issued and these fields should be ignored when HTML is generated.

Architecture

Your implementation should consist of (at least) three main components:

- A program **parse-bib** that consumes a BIBTEX-database and produces an ATerm for it.
- A program **bib2html** that consumes an ATerm for a BIBTEX-database, validates the database, and produces an ATerm for the HTML-rendering of the database.
- A program **pp-html** that consumes an ATerm for an HTML-document and produces a pretty printing of the actual HTML-code for the document.

☞ These need to be stand-alone programs, invocable and combinable from the command line.

Tasks

You will have to implement:

- A tree respresentation for BibTeX-databases.
- A parser for BibTeX.
- A parser for ATerms.
- A validator for BibTeX-trees.
- A tree representation of HTML-documents.
- A translation from BibTeX-trees to HTML-trees.
- A pretty printer for HTML.

☞ You will not need to support all of HTML.

3. Tools

Haskell Utrecht Tools

- Haskell Utrecht Tools Library:
  - Parameterisable scanner
  - Fast, error-correcting parser combinators
  - Pretty-printing combinators
- Utrecht University Attribute Grammar Compiler

Haskell Utrecht Tools Library

Package `uulib`. (Latest stable version: 0.9.5.)

Available from Hackage:
`http://hackage.haskell.org/cgi-bin/hackage-scripts/package/uulib`.

Installation:

```
% runhaskell Setup.hs configure --user --prefix=...
...

% runhaskell Setup.hs build
...

% runhaskell Setup.hs install
...
```

Parameterisable scanner

Performs some simple lexical analysis on an input text, producing a stream of tokens to be consumed by a separate parser.

Disposes of whitespace and Haskell-style comments.

Parameterisable scanner: interface


```
data Pos = Pos !Line !Column Filename

type Line     = Int
type Column   = Int
type Filename = String
```

```
scan :: [String] →     -- reserved identifiers
        [String] →     -- reserved operators
        [Char] →       -- special characters
        [Char] →       -- operator characters
        Pos →          -- initial source position
        String →       -- input
        [Token]
```

```
data Token = ···
```


Parser combinators

- Sophisticated implementation of the applicative interface ((<$>), (<∗>), (<|>), ...).
- Far more efficient than the backtracking parser combinators from the course on Grammars and parsing/Languages and compilers.
- Repairs syntax errors when encountered.
- Complicated types.
- A little thin on documentation.
- To be replaced by a new library in the near future.

☞ Interaction with CCO library (*Feedback* monad, *Component* arrow) requires some additional programming.

Parser combinators: interface

Type of parsers consuming symbols of type σ and producing values of type α:

```
type Parser σ α = ···
```

```
pSucceed :: α → Parser σ α
pFail    :: Parser σ α
pSym     :: σ → Parser σ σ
```

```
(<∗>)    :: Parser σ (α → β) → Parser σ α → Parser σ β
(<$>)    :: (α → β) → Parser σ α → Parser σ β
(<|>)    :: Parser σ α → Parser σ α → Parser σ α
```

Parser combinators: interface (cont'd)

$$opt :: Parser\ \sigma\ \alpha \to \alpha \to Parser\ \sigma\ \alpha$$

$$\begin{aligned} pList &:: Parser\ \sigma\ \alpha \to Parser\ \sigma\ [\alpha] \\ pList1 &:: Parser\ \sigma\ \alpha \to Parser\ \sigma\ [\alpha] \end{aligned}$$

$$\begin{aligned} pChainl &:: Parser\ \sigma\ (\alpha \to \alpha \to \alpha) \to \\ &\quad\ \ Parser\ \sigma\ \alpha \to Parser\ \sigma\ \alpha \\ pChainr &:: Parser\ \sigma\ (\alpha \to \alpha \to \alpha) \to \\ &\quad\ \ Parser\ \sigma\ \alpha \to Parser\ \sigma\ \alpha \end{aligned}$$

Parser combinators: parsing tokens

Parsing reserved identifiers and operators:

$$pKey :: String \to Parser\ Token\ String$$

Parsing special characters:

$$pSpec :: Char \to Parser\ Token\ String$$

Parsing identifiers and operators:

$$\begin{aligned} pVarid, pConid &:: Parser\ Token\ String \\ pVarsym, pConsym &:: Parser\ Token\ String \end{aligned}$$

☞ Indeed, the scanner is somewhat Haskell-centric.

Parser combinators: parsing tokens (cont'd)

Parsing literals:

$$\begin{aligned} pInteger &:: Parser\ Token\ String \\ pFraction &:: Parser\ Token\ String \\ pChar &:: Parser\ Token\ String \\ pString &:: Parser\ Token\ String \end{aligned}$$

Parser combinators: running a parser

$$\begin{aligned} parseIO :: &\ (Eq\ \sigma, Show\ \sigma, Symbol\ \sigma) \Rightarrow \\ &\ Parser\ \sigma\ \alpha \to \\ &\ [\sigma] \to \\ &\ IO\ \alpha \end{aligned}$$

☞ A more involved, lower-level interface is available that allows you, for example, to integrate the parser combinators with the CCO library.

4. Questions

Q: how do I import the parseATerm function?

The slides of one of the previous lectures mentioned the function $parseATerm$.

You have to write a component providing this functionality yourself as part of Mini Project A. ;-)

Q: how do I import <$>?

The libraries that we have considered provide two distinct versions of <$>.

The module *UU.Parsing* from the package `uulib` provides

$$(\texttt{<\$>}) :: (\alpha \to \beta) \to Parser\ \sigma\ \alpha \to Parser\ \sigma\ \beta$$
$$f\ \texttt{<\$>}\ p = pSucceed\ f\ \texttt{<*>}\ p$$

The module *Control.Applicative* from the `base` package provides

$$(\texttt{<\$>}) :: Functor\ \varphi \Rightarrow (\alpha \to \beta) \to \varphi\ \alpha \to \varphi\ \beta$$
$$f\ \texttt{<\$>}\ xs = fmap\ f\ xs$$

☞ The type *ArgumentParser* from *CCO.Tree.Parser* is an instance of *Functor*.

Q: how do I parse an ATerm for a nullary constructor?

$$\textbf{data}\ Direction = North \mid East \mid South \mid West$$

$$
\begin{array}{l}
\textbf{instance}\ Tree\ Direction\ \textbf{where} \\
\quad fromTree\ North = App\ \texttt{"North"}\ [\,] \\
\quad fromTree\ East\ \ = App\ \texttt{"East"}\ \ [\,] \\
\quad fromTree\ South = App\ \texttt{"South"}\ [\,] \\
\quad fromTree\ West\ \ = App\ \texttt{"West"}\ \ [\,] \\
\quad toTree = parseTree\ [\ app\ \texttt{"North"}\ (pure\ North) \\
\qquad\qquad\qquad\qquad\ ,\ app\ \texttt{"East"}\ \ (pure\ East\ ) \\
\qquad\qquad\qquad\qquad\ ,\ app\ \texttt{"South"}\ (pure\ South) \\
\qquad\qquad\qquad\qquad\ ,\ app\ \texttt{"West"}\ \ (pure\ West\ ) \\
\qquad\qquad\qquad\qquad\ ]
\end{array}
$$

☞ $pure :: Applicative\ \varphi \Rightarrow \alpha \to \varphi\ \alpha$ is exported by *Control.Applicative*.

☞ *ArgumentParser* is an instance of *Applicative*.

Functors

From the *Prelude*:

$$
\begin{array}{l}
\textbf{class } \mathit{Functor}\ \varphi\ \textbf{where} \\
\quad \mathit{fmap} :: (\alpha \to \beta) \to \varphi\ \alpha \to \varphi\ \beta
\end{array}
$$

Applicative functors

From *Control.Applicative*:

$$
\begin{array}{l}
\textbf{class } \mathit{Functor}\ \varphi \Rightarrow \mathit{Applicative}\ \varphi\ \textbf{where} \\
\quad \mathit{pure}\ \ :: \alpha \to \varphi\ \alpha \\
\quad (<\!\!*\!\!>) :: \varphi\ (\alpha \to \beta) \to \varphi\ \alpha \to \varphi\ \beta
\end{array}
$$

Monads

From the *Prelude*:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{class } \mathit{Monad}\ \mu\ \textbf{where}} \\
\quad \mathit{return} & :: & \alpha \to \mu\ \alpha \\
\quad (\gg\!=) & :: & \mu\ \alpha \to (\alpha \to \mu\ \beta) \to \mu\ \beta \\
\quad (\gg) & :: & \mu\ \alpha \to \qquad\quad \mu\ \beta\ \ \to \mu\ \beta \\
\quad \mathit{fail} & :: & \mathit{String} \to \mu\ \alpha
\end{array}
$$

Arrows

From *Control.Arrow*:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{class } \mathit{Arrow}\ \varphi\ \alpha\ \beta\ \textbf{where}} \\
\quad \mathit{arr} & :: & (\alpha \to \beta) & \to \varphi\ \alpha\ \beta \\
\quad \mathit{pure} & :: & (\alpha \to \beta) & \to \varphi\ \alpha\ \beta \\
\quad (\ggg) & :: & \varphi\ \alpha\ \beta \to \varphi\ \beta\ \gamma & \to \varphi\ \alpha\ \gamma \\
\quad \mathit{first} & :: & \varphi\ \alpha\ \beta & \to \varphi\ (\alpha, \gamma)\ (\beta, \gamma) \\
\quad \mathit{second} & :: & \varphi\ \alpha\ \beta & \to \varphi\ (\gamma, \alpha)\ (\gamma, \beta) \\
\quad (*\!*\!*) & :: & \varphi\ \alpha\ \beta \to \varphi\ \gamma\ \delta & \to \varphi\ (\alpha, \gamma)\ (\beta, \delta) \\
\quad (\&\&\&) & :: & \varphi\ \alpha\ \beta \to \varphi\ \alpha\ \gamma & \to \varphi\ \alpha\ (\beta, \gamma)
\end{array}
$$

Arrows (cont'd)

Or—as of GHC version 6.10.1:

$$
\begin{array}{l}
\textbf{class } \mathit{Category}\ \varphi \Rightarrow \mathit{Arrow}\ \varphi\ \textbf{where} \\
\quad \mathit{arr} \quad\ :: (\alpha \to \beta) \qquad\qquad \to \varphi\ \alpha\ \beta \\
\quad \mathit{first} \quad :: \varphi\ \alpha\ \beta \qquad\qquad\quad \to \varphi\ (\alpha,\gamma)\ (\beta,\gamma) \\
\quad \mathit{second} :: \varphi\ \alpha\ \beta \qquad\qquad\quad \to \varphi\ (\gamma,\alpha)\ (\gamma,\beta) \\
\quad (***) \quad :: \varphi\ \alpha\ \beta \to \varphi\ \gamma\ \delta \ \to \varphi\ (\alpha,\gamma)\ (\beta,\delta) \\
\quad (\&\&\&) :: \varphi\ \alpha\ \beta \to \varphi\ \alpha\ \gamma \to \varphi\ \alpha\ (\beta,\gamma)
\end{array}
$$

From $\mathit{Control.Category}$:

$$
\begin{array}{l}
\textbf{class } \mathit{Category}\ \varphi\ \textbf{where} \\
\quad \mathit{id} \ :: \varphi\ \alpha\ \alpha \\
\quad (\circ) :: \varphi\ \beta\ \gamma \to \varphi\ \alpha\ \beta \to \varphi\ \alpha\ \gamma \\
\\
(\ggg) :: \mathit{Category}\ \varphi \Rightarrow \varphi\ \alpha\ \beta \to \varphi\ \beta\ \gamma \to \varphi\ \alpha\ \gamma \\
(\ggg) = \mathit{flip}\ (\circ)
\end{array}
$$

Q: why do we need a parser for ATerms?

A term:

$$2 + 3 * 5$$

Its Haskell representation:

$$\mathit{Add}\ 2\ (\mathit{Mul}\ 3\ 5) :: \mathit{Tm}$$

The Haskell representation of the corresponding ATerm:

$$\mathit{App}\ \texttt{"Add"}\ [\mathit{Integer}\ 2, \mathit{App}\ \texttt{"Mul"}\ [\mathit{Integer}\ 3, \mathit{Integer}\ 5]] :: \mathit{ATerm}$$

Concrete syntax for the ATerm:

$$\mathit{Add}(2, \mathit{Mul}(3, 5))$$

The parser for ATerms mediates between the concrete syntax and the Haskell representation of an ATerm.

Q: why do I need to define my own tree types?

Although the ATerm format is mainly an exchange format, in principle you could program just against the unified interface that ATerms provide.

However, you miss out on quite a bit of safety then: little guarantees are provided by the implementation language on the actual shape of trees. In particular, if trees (not only those that are given as inputs to a compiler, but also those constructed by the compiler itself) are of an incorrect form, this can only be detected when the compiler is run (as opposed to when the compiler is compiled).

Moreover, programming against the unified ATerm-interface is typically more complicated than programming in terms of abstractions that are tailored to a specific domain.

Q: is there a bug in the parser for trees?

Will the following program compile?

```
import CCO.Tree              (ATerm (App), Tree (fromTree, toTree))
import CCO.Tree.Parser       (parseTree, app, arg)
import Control.Applicative   ((<$>))

data Unit = Unit

instance Tree Unit where
  fromTree Unit = App "Unit" []
  toTree = parseTree [app "Unit" (Unit <$> arg)]
```

No, it will not:

```
    Couldn't match expected type `a -> Unit'
           against inferred type `Unit'
    In the first argument of `(<$>)', namely `Unit'
    In the second argument of `app', namely `(Unit <$> arg)'
    In the expression:  app "Unit" (Unit <$> arg)
Failed, modules loaded:  none.
```