Script: Unparsing and parsing in Haskell

Headline

Lecture Parsing and unparsing as part of Course:Lambdas in Koblenz

Description

Various data structures, e.g., term- or tree-based representations, can be rendered as text by means of unparsing. (Unparsing is very similar to what's calledpretty printing.) The other way around, the structure underlying some text can be recovered by means of parsing so that the result can be processed as a tree or another data structure. In both cases, we assume some syntax for the underlying a textual representation. This syntax can be also be made explicit in the form of a grammar. We put suitable Haskellcombinator libraries to work for the implementation of unparsers and parsers.

Concepts

- Unparsing
  - Unparser
  - Pretty printing
  - Pretty printer
- Syntax
  - Concrete syntax
  - Abstract syntax
  - Grammar
    - Context-free grammar
- Parsing
  - Parsing problem
  - Syntax tree
  - Parse tree
  - Acceptor
  - Parser

Technologies

- Combinator libraries
  - Technology:HughesPJ
  - Technology:Parsec

Features

- Feature:Unparsing
- Feature:Parsing

Contributions

- Contribution:hughesPJ
- Contribution:haskellAcceptor
- Contribution:haskellParsec

Metadata

- Course:Lambdas in Koblenz
- Script:Functors_and_friends

Concept: Unparsing

Headline

Translate syntax trees into text or another representation

Illustration

See the related notion of pretty printing for an illustration.

Relationships

- Unparsing is performed by an unparser.
- Unparsing is the opposite of parsing.
- Unparsing is similar to pretty printing.

Metadata

- http://en.wikipedia.org/wiki/Unparser
- Vocabulary:Software language engineering
- Concept

Concept: Parsing

Headline

Analysis of text and construction of parse trees

Relationships

- Parsing is performed by a parser.
- Parsing is the opposite of unparsing.
- Parsing solves the parsing problem.

Illustration

See Contribution:haskellParsec for an extensive illustration.

Here is a simple illustration based on the assumed I/O behavior of parsing Java code.

Input of parsing: text

Consider the following textual representation of a Java statement, subject to Java's concrete syntax.

```
x = 42
```

Output of parsing: tree

Here is the corresponding parse tree, assuming some XML-based representation for abstract syntax:

```
<assign>
  <lhs>
    <id>x</id>
  </lhs>
  <rhs>
    <constant>42</constant>
  </rhs>
</assign>
```

Metadata

- http://en.wikipedia.org/wiki/Parsing
- Vocabulary:Software language engineering
- Vocabulary:Programming
- Unparsing

Course: Lambdas in Koblenz

Headline

Introduction to functional programming at the University of Koblenz-Landau

Schedule

- Lecture First steps
- Lecture Basic software engineering
- Lecture Searching and sorting
- Lecture Data modeling in Haskell
- Lecture Functional data structures
- Lecture Higher-order functions
- Lecture Type-class polymorphism
- Lecture Functors and friends
- Lecture Unparsing and parsing
- Lecture Monads
- Lecture Generic functions

Metadata

- http://softlang.wikidot.com/course:fp

Concept: Pretty printer

Headline

A software component that performs pretty printing

Illustration

See Technology:HughesPJ for (the illustration of) a combinator library for pretty printing.

See Contribution:hughesPJ for a contribution with a Language:Haskell-based pretty printer.

Relationships

- A pretty printer performs pretty printing.
- See also the very related notion of unparser.

Metadata

- Vocabulary:Software language engineering
- Language technology

Concept: Parsing problem

Headline

The membership problem for the language generated by a grammar

Details

The problem can described like this: Given a grammar and a string, is the string an element of the language generated by the grammar? There exist corresponding algorithms for certain classes of grammars, e.g., for context-free grammars, in which case the parsing problem turns out to be an algorithmic problem, thus. While the parsing problem, in a narrow sense, focuses indeed on the membership aspect, it may be understood more broadly to cover the richer algorithmic problem of what a parser is supposed to that, which include recovery of the syntactical structure of the input in the form of a syntax tree.

Metadata

- http://en.wikipedia.org/wiki/Parsing
- Algorithmic problem
- Vocabulary:Software language engineering

Concept: Concrete syntax

Headline

Syntax aimed at reading and writing as opposed to a focus on processing

Illustration

Consider the following sequence of Java statements:

```
x = 42;
System.out.println(x);
```

This textual notation is easy to read and write by a human. This is also a consequences of the use of certain features such as the use of spaces and linebreaks or special characters for language concepts. However, such a textual notation may require special preprocessing (i.e., parsiing), before it can be manipulated comveniently by programs. This is why we may also need an abstract syntax.

Metadata

- Syntax
- Abstract syntax
- https://en.wikipedia.org/wiki/Abstract_syntax

Concept: Unparser

Headline

A software component that performs unparsing

Illustration

See Contribution:hughesPJ for a contribution with a Language:Haskell-based unparser.

Relationships

- An unparser performs unparsing.
- An unparser is the opposite of a parser.
- See also the very related notion of pretty printer.

Metadata

- http://en.wikipedia.org/wiki/Unparser
- Vocabulary:Software language engineering
- Language technology

Concept: Abstract syntax

Headline

Syntax aimed at processing as opposed to reading and writing

Illustration

The Java assignment "x = 42" would be rendered in some possible XML-based abstract syntax like this:

```
<assign>
  <lhs>
    <id>x</id>
  </lhs>
  <rhs>
    <constant>42</constant>
  </rhs>
</assign>
```

For a bit more involved example, consider first this Java statement sequence inconcrete syntax:

```
x = 42;
System.out.println(x);
```

In XML-based abstract syntax:

```
<sequence>
  <assign>
    <lhs>
      <id>x</id>
    </lhs>
    <rhs>
      <constant>42</constant>
    </rhs>
  </assign>
  <methodcall>
    <receiver>
      <static>
        <id>System</id>
        <id>out</id>
      </static>
    </receiver>
    <methodname>
      <id>println</id>
    </methodname>
    <arguments>
      <id>x</id>
    </arguments>
  </methodcall>
</sequence>
```

Metadata

- Syntax
- Concrete syntax
- https://en.wikipedia.org/wiki/Abstract_syntax

Concept: Acceptor

Headline

A program that accepts input according to some formal definition

Details

For instance, we may implement a context-free grammar for parsing, e.g., as a recursive descent parser without adding any semantic actions, though, and thereby obtain an acceptor for the language generated by the grammar. The acceptor behaves essentially as a predicate on given input: accept (true) or reject (false).

Illustration

See Technology:Parsec for (the illustration of) a combinator library for parsing (or "accepting").

See Contribution:haskellAcceptor for a contribution with a Language:Haskell-based acceptor.

Relationship

- An acceptor is "degenerated" parser. (No parse trees are synthesized.)

Metadata

- http://en.wikipedia.org/wiki/Recognizer#Acceptors_and_recognizers
- Vocabulary:Software language engineering
- Software technology

Concept: Parse tree

Headline

Another term for syntax tree

Illustration

See the illustration of syntax tree.

Concept: Syntax tree

Headline

A tree representing the grammatical structure of text

Description

We explain the concept by means of an illustrative example.

Illustration

See the illustrations of abstract and concrete syntax for simple intuitions.

A more profound illustration follows.

Consider the following context-free grammar; we label the productions for convenience:

```
[literal] expression ::= literal
[binary] expression ::= "(" expression op expression ")"
[plus] op ::= "+"
[times] op ::= "*"
```

We assume that literals are integers as in sequences of digits.

Now consider this input:

```
((4 * 10) + 2)
```

A parsing algorithm would basically accept the input string, as it is an element of the language generated by the grammar; see also the parsing problem. In addition, an actual parser would also represent the grammar-based structure of the input string by a parse tree. We represent the parse tree for the input string here as a prefix term such that we use production labels as function symbols:

```
binary(
  "(",
  binary(
    "(",
    literal("4"),
    times("*"),
    literal("10"),
    ")",
  ),
  plus("+"),
  literal("2"),
  ")",
)
```

That is, each non-leaf node in the tree corresponds to a production label and each leaf node is a

terminal. Further, each non-leaf node has as many branches as the underlying production has symbols in the right-hand side sequence. We assume that the branches are ordered in the same way as the underlying right-hand side and the subtrees are parse trees for the symbols in the right-hand side. A parse tree for a given nonterminal is rooted by a production with the nonterminal on the left-hand side.

The parse tree shown above is a concrete syntax tree in that it captures even the terminals of the derivation. We may also remove those symbols to arrive at an abstract syntax tree. Thus:

```
binary(
  binary(
    literal("4"),
    times,
    literal("10")
  ),
  plus,
  literal("2")
)
```

Metadata

- http://en.wikipedia.org/wiki/Syntax_tree
- http://en.wikipedia.org/wiki/Parse_tree
- Vocabulary:Software language engineering

Feature: Parsing

Headline

Parse an external format for companies

Description

Users of the system:Company may need to exchange data with other systems or edit data independently of the system. To this end, some XML- or JSON-based format or a concrete textual or visual syntax may need to be supported. The corresponding representation format may actually be imposed on the system by external factors. Consequently, the system:Company may need to consume such an external representation through parsing, as covered by the present feature, or it may need to produce such data through unparsing, as covered by the extra Feature:Unparsing.

An implementation of parsing is to be demonstrated for a sample company as follows. In the most basic case, the implementation has to illustrate at least 'acceptance' of the input. Another option is that parsing populates a data model for companies. Yet another option is to perform the computation for totaling salaries according to Feature:Total along with parsing.

Relationships

- Feature:Parsing is complemented by Feature:Unparsing.
- Feature:Parsing and Feature:Unparsing are related to serialization, as covered by the designated Feature:Serialization, but we speak of parsing specifically, when the system:Company needs to actually process (parse) the external format, thus going beyond the uniform use of a serialization framework.

Metadata

- Functional requirement
- Optional feature
- Parsing
- Feature:Textual syntax
- System:FSML

Feature: Unparsing

Headline

Format companies in an external format

Description

Users of the system:Company may want to export data to another system. Hence, the system:Company may need to support such export, e.g., on the grounds of XML, JSON, or a concrete syntax. See also Feature:Parsing. Forms of open serialization may be used to enable import/export.

Metadata

- Functional requirement
- Optional feature

Concept: Pretty printing

Headline

Formating data structures such as source code as text

Details

Arguably, pretty printing is a synonym for unparsing. The term *pretty printing* hints at the fact that the output is supposed to be *pretty*. Thus, some emphasis is on the formatting rules, whereas we may prefer to speak of *unparsing* when we focus on mapping a (parse) tree or another data structure to a corresponding string representation. However, in practice, pretty printing and parsing are used often interchangeably.

Illustration

Consider this parse tree of an if-statement, as represented as a Haskell term:

```
If
  (Eq (Var "x") (Var "y"))
  (Return (Var "i))
  Skip
```

A pretty printer may yield the following formatted text:

```
if (x==y)
  return i;
```

If we make less effort and leave out linebreaks and indentation, we may instead get text like this:

```
if (x==y) return i;
```

Metadata

- http://en.wikipedia.org/wiki/Prettyprint
- Vocabulary:Software language engineering
- Concept

Contribution: haskellAcceptor

Headline

Parsing (acceptance only) in Haskell with Parsec

Motivation

The implementation demonstrates parsing (acceptance) in Haskell with the Parsec library of parser combinators. A concrete textual syntax for companies is assumed. Acceptance is considered only. Thus, no abstract syntax is constructued. We set up basic parsers for quoted strings and floating-point numbers. Further, we compose parsers for companies, departments, and employees using appropriate parser combinators for sequences, alternatives, and optionality. By design, the acceptor is kept simple in terms of leveraged programming technique; in particular, monadic style and applicative functors are avoided to the extent possible.

Illustration

We would like to process a textual representation of companies; "..." indicates an elision:

```
company "Acme Corporation" {
  department "Research" {
    manager "Craig" {
      address "Redmond"
      salary 123456.0
    }
    employee "Erik" {
      address "Utrecht"
      salary 12345.0
    }
    employee "Ralf" {
      address "Koblenz"
      salary 1234.0
    }
  }
  department "Development" {
    ...
  }
}
```

Let's assume that the textual representation is defined by the followingcontext-free grammar:

```
company = "company" literal "{" department* "}"
department = "department" literal "{" manager subunit* "}"
subunit = nonmanager | department
manager = "manager" employee
nonmanager = "employee" employee
employee = literal "{" "address" literal "salary" float "}"
```

We can now apply a mapping from the grammar to a functional program in the following way:

- Each nonterminal becomes a function that is of Parsec's parser type.
- The function definition composes parsers following the production's structure.
- We may need to deal with lexical trivia such as spaces.
- We may want to check for the end-of-file to be sure to have looked at the complete input.

At this point, we are merely interested in the syntactic correctness of such inputs. Thus, the parser functions do not need to construct any proper parse trees. They merely return "()".

Here is the parser function for departments:

```
-- Accept a department
parseDepartment :: Acceptor
parseDepartment
  =  parseString "department"
  >> parseLiteral
  >> parseString "{"
  >> parseManager
  >> many parseSubUnit
  >> parseString "}"
```

The composition uses ">>" for sequential composition in the same way as the original production for departments uses juxtaposition for the sequential composition of various terminals and nonterminals. The type *Acceptor* is defined as a parser type where the type ofparse trees is "()":

```
-- The parser type for simple acceptors
type Acceptor = Parsec String () ()
```

We also need parsers for the basic units of input: literals (strings) and floats. Here is the parser for floats:

```
-- Accept a float
parseFloat :: Acceptor
parseFloat
  =  many1 digit
  >> char '.'
  >> many1 digit
  >> spaces
  >> return ()
```

That is, a float is defined to start with a non-empty sequence of digits, followed by ".", followed by another non-empty sequence of digits. In addition, any pending spaces are consumed as well. Finally, "()" is returned as the trivial parse tree of such an acceptor.

Relationships

- Contribution:haskellParsec advances this acceptor into a proper parser.
- Contribution:antlrAcceptor and others use the same textual representation.

Architecture

There are these modules:

- Main: acceptance test

- Company/Parser: the actual parser (acceptor)

The input is parsed from a file "sampleCompany.txt".

Usage

See https://github.com/101companies/101haskell/blob/master/README.md.

Metadata

- Language:Haskell
- Technology:GHC
- Technology:Cabal
- Technology:Parsec
- Feature:Hierarchical company
- Feature:Parsing
- Contributor:rlaemmel

Concept: Parser

Headline

A program performing parsing

Illustration

See Technology:Parsec for (the illustration of) a combinator library for parsing.

See Contribution:haskellParsec for a contribution with a Language:Haskell-based parser.

Relationships

- A parser performs parsing.
- A parser is the opposite of an unparser.
- A parser is strictly more "powerful" than an acceptor. (Parse trees are synthesized.)

Metadata

- http://en.wikipedia.org/wiki/Parsing
- Vocabulary:Software language engineering
- Language technology

Technology: Parsec

Headline

A parser combinator library in Haskell

Illustration

Parsec-based parsers are built from parser combinators. For instance, the following trivial expressions denote parser for a digit or a letter, respectively. Such parsers for character classes are readily provided by Parsec:

```
digit
```

```
letter
```

The following expression denotes a parser for a non-empty sequence of digits; the*many1* combinator corresponds essentially to "+" in EBNF notation for context-free grammars:

```
many1 digit
```

We will look at other combinators shortly, but let us first run the composed parsers. Parsec provides a *runP* function. For instance, we can attempt to parse a digit:

```
> runP digit () "" "1"
Right '1'
```

The input string for the parser *digit* is "1". The remaining arguments resolve some parameterization of Parsec which we skip here. The run returns the successfully parsed character in the right summand of an either type; the left operand is reserved for error handling. We see an unsuccessful parse, indeed, in the next example:

```
> runP digit () "" "x"
Left (line 1, column 1):
unexpected "x"
expecting digit
```

That is, we receive an error message with line and column information about the discrepancy between actual and expected input. Clearly, the input "x" cannot be parsed as a digit. Let us also run the parser for non-empty sequences on a few inputs:

```
> runP (many1 digit) () "" "42"
Right "42"
> runP (many1 digit) () "" "42x"
Right "42"
> runP (many1 digit) () "" "x42"
Left (line 1, column 1):
unexpected "x"
expecting digit
```

The first is successful because the input string "42" is exactly a sequence of digits. The second parse is also successful because the input string "42x" does at least have a sequence of digits as a prefix. The third parse fails because it does not start with a non-empty sequence of digits.

We can compose parser sequentially and by choice:

```
> runP (letter >> digit) () "" "a1"
Right "1"
> runP (many1 (letter <|> digit)) () "" "a1"
Right "a1"
```

The first parser parses a sequence of a letter and a digit. The second parser parses any non-empty sequence of letters or digits ("<|>"). Consider the parse tree returned for the first parsers. It is evident that the first component of the sequence does not contribute to the resulting parse tree. This is because the simple form of sequential composition (">>") indeed ignores the result of the first operand. We would need to leverage a more complex form of sequential composition (">>=") to explicitly capture the intermediate results for both operands and return their composition. Thus:

```
> runP (letter >>= \l -> digit >>= \d -> return [l,d]) () "" "a1"
Right "a1"
```

This form of sequential composition passes the result from the first operand to the second so that the latter can capture the result with a lambda. We also see how the sequential composition is finished off with a trivial parser with simply *returns* a value. We can also use the value-passing form of sequential composition to improve the earlier example of a parser for a digit sequence such that the parser returns an actual int rather than a list of characters:

```
> runP (many1 digit >>= \s -> return (read s :: Int)) () "" "42"
Right 42
```

That is, we compose *many1 digit* with a function which converts the parsed string to an int and returns it as the final result. The function *return* is also a parser combinator, which is used when a given value should be returned as opposed to invoking an actual parser on the input. (If you are familiar with monads, then you realize that Parsec leverages a monad with its operations *return*, ">>", and ">>=" for parsing, but if you are not aware of monads, then this should not be any problem.)

In the most general case, parsers are of thispolymorphic type:

```
data ParsecT s u m a
```

The type parameters serve these roles:

- *s*: the stream type for the input
- *u*: a type for user state, e.g., for a symbol table
- *m*: an extra monad to add effects to parsing
- *a*: the type of theparse tree

When actual parsing does not involve any underlying monad, then the identity monad is used:

```
type Parsec s u = ParsecT s u Identity
```

In simple applications of Parsec, the stream type is*String]] and no user state is used. This results in the following simplification; we also provide a simplified variation on* runP*:*

```
type Parsec' = Parsec String ()
```

Here is another sample parser. It models names as they are similarly defined in many language syntax. That is, names should start with a letter and proceed with any number of letters or digits:

```
name :: Parsec' String
name = letter
    >>= \l -> many (letter <|> digit)
    >>= return . (l:)
```

The interesting bit is how we (need to) compose the initial letter with the remaining sequence. That is, we need to "cons up" the first letter with the remaining sequence. For instance:

```
> runP' name "a42 b88"
Right "a42"
```

See Contribution:haskellParsec for an illustration of using Parsec.

Metadata

- Combinator library
- Haskell technology
- http://hackage.haskell.org/package/parsec
- http://www.haskell.org/haskellwiki/Parsec
- http://research.microsoft.com/en-us/um/people/daan/download/parsec/parsec-letter.pdf
-
- Parsing

Concept: Context-free grammar

Headline

A kind of grammar used, for example, for syntax definition

Illustration

Context-free grammars consist of:

- a set of terminal ("strings" from which to compose inputs),
- a set of nonterminal (placeholders for syntactical categories in derivations),
- an (optional) designated startsymbol (a nonterminal from which to start derivations), and
- a set of productions (rules) for derivations.

In fact, a context-free grammar can be described just by the rules as these rules enumerate the terminals and nonterminals as well. (We may also assume that the left-hand side nonterminal of the first rule is simply the startsymbol). What's important is the structure of rules. Each rules consists of:

- a left-hand side which is a nonterminal, and
- a right-hand side which is some expression over terminals and nonterminals.

In the most basic form, said expressions are simply sequences over terminals and nonterminals. Alternatives for derivation are already expressible, as there could be multiple rules with the same left-hand side nonterminal. In practice, notational extensions are commonplace. For example, so-called EBNF notations may cater for these expression forms:

- *x**: Any number of repetitiions of *x* including 0 repetitions.
- *x?*: Any number of repetitiions of *x* excluding 0 repetitions.
- *x*|*y*: *x* or *y*.
- *x?*: *x* or the empty string.

Here is a context-free grammar for a possible concrete syntax for companies of the @system; for what it matters, we use Technology:ANTLR's EBNF-like notation:

Nonterminals (with explanation):

- **Company**: complete company structures
- **Department**: department sub-structures of companies
- **Employee**: employee sub-structures of departments
- **NonManager**: managers rather than non-managerial employees
- **QString**: double-quoted strings for names and addresses
- **Number**: floating point numbers for salaries

Terminals:

- "company"
- "department"
- "employee"
- "manager"
- "address"
- "salary"
- "{"
- "}"

Startsymbol: Company

Productions:

```
Company :
  'company' QString '{'
    Department*
  '}';

Department :
  'department' QString '{'
    'manager' Employee
    Department*
    NonManager*
  '}';

NonManager : 'employee' Employee;

Employee : QString '{'
    'address' QString
    'salary' Number
  '}';
```

As an exercise, let us define about the same syntax with a different grammar. In the original grammar, the lists of departments and employees were separated. We may also consider a mixed list of departments and employees. To this end, we assume an extra nonterminal "SubUnit" for a choice between department and employee. As a result, we would need these alternative productions:

```
Company :
  'company' QString '{'
    Department*
  '}';

Department :
  'department' QString '{'
    'manager' Employee
    SubUnit*
  '}';

SubUnit : NonManager | Department ;

NonManager : 'employee' Employee;

Employee : QString '{'
    'address' QString
    'salary' Number
```

'}';

A (context-free) grammar has a simple semantics. It defines a set of strings (the so-called language generated by the grammar) which are derivable by repeated rule application starting from the symbol such that nonterminals are replaced by matching right-hand sides until no nonterminals are left. This generative definition also gives rise to an algorithmic problem, the parsing problem, such that one can check whether a given string is actually in the language generated by a grammar and recover the underlying syntactical structure as parse tree.

Metadata

- http://en.wikipedia.org/wiki/Context-free_grammar
- Grammar
- Vocabulary:Software language engineering

Concept: Grammar

Headline

A set of formation rules for strings, trees, graphs, or other artifacts

Illustration

See the concept of context-free grammars for a more specific form of grammars and an associated illustration.

Metadata

- http://en.wikipedia.org/wiki/Formal_grammar
- http://en.wikipedia.org/wiki/Grammar
- Vocabulary:Software language engineering

Concept: Syntax

Headline

Rules defining a software language as a set of structured elements

Illustration

See the illustrations for more specific types of syntax, e.g.:

- Concrete syntax
- Abstract syntax

Metadata

- https://en.wikipedia.org/wiki/Syntax_(programming_languages)
- Vocabulary:Programming languages

Contribution: haskellParsec

Headline

Parsing in Haskell with Parsec

Motivation

The implementation demonstrates parsing in Haskell with the Parsec library of parser combinators. A concrete textual syntax for companies is assumed.Parse trees are constructed in accordance to an abstract syntax defined in terms ofalgebraic data types. We set up basic parsers for quoted strings and floating-point numbers. Further, we compose parsers for companies, departments, and employees using appropriate parser combinators for sequences, alternatives, and optionality. By design, the parser is kept simple in terms of leveraged programming technique; in particular, monadic style and applicative functors are avoided to the extent possible.

Illustration

See Contribution:haskellAcceptor for a basic illustrationof Parsec-based parsing. The present contribution is more complex in that it constructs proper parse trees.

```
parseDepartment :: Parser Department
parseDepartment = Department
  <$> (parseString "department"
  >>  parseLiteral)
  <*  parseString "{"
  <*> parseEmployee "manager"
  <*> many parseSubUnit
  <*  parseString "}"
```

To this end, we use a parser type that is still parametric in the type of parse trees. Thus:

```
-- Shorthand for the parser type
type Parser = Parsec String ()
```

Relationships

- Contribution:haskellAcceptor is merely an acceptor as opposed to the proper parser at hand.
- Contribution:haskellVariation sponsored the data model used in the present contribution.
- Contribution:antlrAcceptor and others use the same textual representation.

Architecture

There are these modules:

- Main: parser test
- Company/Parser: the actual parser
- Company/Data: the abstract syntax definition
- Company/Sample: a baseline for testing at the level of abstract syntax

The input is parsed from a file "sampleCompany.txt".

Usage

See https://github.com/101companies/101haskell/blob/master/README.md.

Metadata

- Language:Haskell
- Technology:GHC
- Technology:Cabal
- Technology:Parsec
- Feature:Hierarchical company
- Feature:Parsing
- Contributor:MedeaMelana
- Contributor:tschmorleiz
- Contributor:rlaemmel
- Theme:Haskell potpourri
- Theme:Haskell introduction
- Contribution:haskellAcceptor
- Contribution:haskellVariation
- Contribution:haskellAcceptor

Contribution: hughesPJ

Headline

Unparsing in Language:Haskell with a pretty printing combinator library

Characteristics

This contribution demonstrates unparsing, i.e., rendering a term-based representation as according to a some specific text-based concrete syntax. The unparser is described at a relatively high level of abstraction by means of an appropriate library in Language:Haskell.

Illustration

Input of unparsing is a term-based representation like this; "..." indicates an elision:

```
sampleCompany = Company
  "Acme Corporation"
  [ Department "Research"
      (Employee "Craig" "Redmond" 123456)
      []
      [ Employee "Erik" "Utrecht" 12345,
        Employee "Ralf" "Koblenz" 1234
      ],
    Department "Development"
      ...
  ]
```

Output of unparsing is a test-based representation like this:

```
company "Acme Corporation" {
  department "Research" {
    manager "Craig" {
      address "Redmond"
      salary 123456.0
    }
    employee "Erik" {
      address "Utrecht"
      salary 12345.0
    }
    employee "Ralf" {
      address "Koblenz"
      salary 1234.0
    }
  }
  department "Development" {
    ...
  }
}
```

Here is the function for unparsing:

```
unparse :: Company -> Doc
unparse (Company n ds) =
  bracy "company" n (vcat (map unparseD ds))
  where
    bracy :: String -> String -> Doc -> Doc
    bracy k n d =
        text k <+> doubleQuotes (text n) <+> text "{"
      $$ nest 2 d
      $$ text "}"
    unparseD :: Department -> Doc
    unparseD (Department n m ds es) =
      bracy "department" n (vcat (   [unparseE "manager" m]
                                  ++ map unparseD ds
                                  ++ map (unparseE "employee") es))
      where
        unparseE :: String -> Employee -> Doc
        unparseE k (Employee n a s) = bracy k n (a' $$ s')
          where
            a' = text "address" <+> doubleQuotes (text a)
            s' = text "salary" <+> text (show s)
```

It uses various combinators of Technology:HughesPJ all over the place. For instance, it uses *vcat* to vertically compose departments; it uses *nest* to achieve indentation for constituents of companies, departments, and employees. Overall, all the subexpressions render the company terms to a library-specific type of *Doc*, which is essentially an abstraction over text.

Relationships

- The data model is the same as the one of Contribution:haskellComposition.
- The textual output format is the same as the one parsed by Contribution:haskellParsec.

Metadata

- Language:Haskell
- Language:Haskell 98
- Technology:GHC
- Technology:HughesPJ
- Contribution:haskellComposition
- Feature:Hierarchical company
- Feature:Closed serialization
- Feature:Unparsing
- Contributor:rlaemmel
- Theme:Haskell potpourri

Technology: HughesPJ

Headline

A Language:Haskell library pretty printing

Illustration

The library's central type is *Doc* for documents. *Doc* is abstraction over text. The idea is that one maps abstract syntax (modeled by algebraic data types) to *doc*s with the help of combinators serving, for example, horizontal and vertical composition. A *doc* can then be "shown" literally as text.

Thus:

```
instance Show Doc
  where
    -- type is abstract; it can be shown
```

Documents can be contructed from literals by these combinators:

```
-- Map string to document
text :: String -> Doc

-- Map int to document
int :: Int -> Doc
```

Here is an illustration:

```
> text "hello"
hello
> int 42
42
```

Documents can be composed in some ways, e.g.:

```
-- The empty docment
empty :: Doc

-- Compose horizontally
(<>) :: Doc -> Doc -> Doc

-- Compose horizontally with extra space for separation
(<+>) :: Doc -> Doc -> Doc

-- Compose vertically
($$) :: Doc -> Doc -> Doc
```

Here is an illustration:

```
> empty
```

```
> text "4" <> text "2"
42
> text "before" <+> text "after"
before after
> text "above" $$ text "below"
above
below
```

The combinators also satisfy some reasonable laws. For example,*empty* is a unit of horizontal composition -- even the form with an extra space for separation.

```
> empty <+> int 42
42
```

Metadata

- Library
- Unparsing
- http://hackage.haskell.org/package/pretty
