PARSING USING PARSEC: A PRACTICAL EXAMPLE

WIM VANDERBAUWHEDE

The aim of this tutorial it to explain step by step how to build a simple parser using the Parsec library.

The source code can be found on GitHub

Parsing the output of derived `Show`. The polymorphic function `show` returns a string representation of any data type that is an instance of the type class `Show`. The easiest way to make a data type an instance of type class is using the `deriving` clause.

```
show :: Show a => a -> String
data D = D ... deriving (Show)
d :: D
d = D ...
str :: String
str = show d -- string representation of the instance of the data type
```

In this tutorial we will show how to create a parser that will parse the output of a derived `show` and return it in XML format.

```
parseShow :: String -> String
xml = parseShow $ show res
```

Example data type. First we create a data type `PersonRecord`

```
data PersonRecord  = MkPersonRecord {
    name :: String,
    address :: Address,
    id :: Integer,
    labels :: [Label]
} deriving (Show)
```

The types `Address` and `Label` are defined as follows:

```
data Address = MkAddress {
    line1 :: String,
    number :: Integer,
    street :: String,
    town :: String,
```

```
    postcode :: String
} deriving (Show)

data Label = Green | Red | Blue | Yellow deriving (Show)
```

We derive `Show` using the `deriving` clause. The compiler will automatically create the show function for this data type. Our parser will parse the output of this automatically derived show function.

Then we create some instances of `PersonRecord`:

```
rec1 = MkPersonRecord
    "Wim Vanderbauwhede"
    (MkAddress "School of Computing Science" 17 "Lilybank Gdns" "Glasgow" "G12 8QQ")
    557188
    [Green, Red]

rec2 = MkPersonRecord
    "Jeremy Singer"
    (MkAddress "School of Computing Science" 17 "Lilybank Gdns" "Glasgow" "G12 8QQ")
    42
    [Blue, Yellow]
```

We can test this very easily:

```
main = putStrLn $ show [rec1,rec2]
```

This program produces the following output:

```
[wim@workai HaskellParsecTutorial]$ runhaskell test_ShowParser_1.hs
[MkPersonRecord {name = "Wim Vanderbauwhede", address = MkAddress {line1 = "School o
```

The derived Show format can be summarized as follows:

- Lists: [... comma-separated items ...]
- Records: { ... comma-separated key-value pairs ...}
- Strings: "..."
- Algebraic data types: variant type name

Building the parser. We create a module `ShowParser` which exports a single function `parseShow`:

```
module ShowParser ( parseShow ) where
```

Some boilerplate:

```
import Text.ParserCombinators.Parsec
import qualified Text.ParserCombinators.Parsec.Token as P
import Text.ParserCombinators.Parsec.Language
```

The `Parsec.Token` module provides a number of basic parsers. Each of these takes as argument a *lexer*, generated by `makeTokenParser` using a language definition. Here we use `emptyDef` from the `Language` module.

It is convenient to create a shorter name for the predefined parsers you want to use, e.g.

```
parens = P.parens lexer
-- and similar
```

The parser. The function `parseShow` takes the output from `show` (a `String`) and produces the corresponding XML (also a `String`). It is composed of the actual parser `showParser` and the function `run_parser` which applies the parser to a string.

```
parseShow :: String -> String
parseShow = run_parser showParser

showParser :: Parser String

run_parser :: Parser a -> String -> a
run_parser p str =  case parse p "" str of
    Left err -> error $ "parse error at " ++ (show err)
    Right val  -> val
```

The XML format. We define an XML format for a generic Haskell data structure. We use some helper functions to create XML tags with and without attributes.

Header:

```
<?xml version="1.0" encoding="utf-8"?>
```

```
xml_header =  "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n"
```

Tags:

```
<tag> ... </tag>
```

```
otag t = "<"++t++">"
ctag t = "</"++t++">"
tag t v = concat [otag t,v,ctag t]
```

Attributes:

```
<tag attr1="..." attr2="...">
```

```
    tagAttrs :: String -> [(String,String)] -> String -> String
    tagAttrs t attrs v =
        concat [
            otag (unwords $ [t]++(map (\(k,v) -> concat [k,"=\"",v,"\""]) attrs))
            ,v
            ,ctag t
            ]
```

We also use some functions to join strings together. From the Prelude we take:

```
    concat :: [[a]] -> [a] -- join lists
    unwords :: [String] -> String -- join words using spaces
```

We also define a function to join strings with newline characters:

```
    joinNL :: [String] -> String -- join lines using "\"
```

This is identical to `unlines` from the Prelude, just to illustrate the use of `intercalate` and the `Data.List` module.

Parsers for the derived `Show` *format.*

Lists.

```
[ ..., ..., ... ]
```

XML :

```
<list>
<list-elt>...</list-elt>
...
</list>
```

```
    list_parser = do
        ls <- brackets $ commaSep showParser
        return $ tag "list" $ joinNL $ map (tag "list-elt") ls
```

Tuples.

```
: ( ..., ..., ... )
```

XML:

```
<tuple>
<tuple-elt>...</tuple-elt>
...
</tuple>
```

```
tuple_parser = do
    ls <- parens $ commaSep showParser
    return $ tag "tuple" $ unwords $ map (tag "tuple-elt") ls
```

Record types.

```
Rec { k=v, ... }
```

XML:

```
<record>
<elt key="k">v</elt>
...
</record>
```

```
key-value pairs: k = v -- v can be anything
```

```
record_parser = do
    ti <- type_identifier
    ls <- braces $ commaSep kvparser
    return $ tagAttrs "record" [("name",ti)] (joinNL ls)

kvparser = do
    k <- identifier
    symbol "="
    t <- showParser
    return $ tagAttrs "elt" [("key",k)] t

type_identifier = do
    fst <- oneOf ['A' .. 'Z']
    rest <- many alphaNum
    whiteSpace
    return $ fst:rest
```

Algebraic data types.

```
e.g. Label
```

XML:

```
<adt>Label</adt>
```

```
adt_parser = do
    ti <- type_identifier
    return $ tag "adt" ti
```

Quoted strings and numbers.

```
quoted_string = do
    s <- stringLiteral
    return $ "\""++s++"\""

number = do
    n <- integer
    return $ show n
```

Complete parser. Combine all parsers using the choice combinator <|>.

```
    showParser :: Parser String
    showParser =
        list_parser <|> -- [ ... ]
        tuple_parser <|> -- ( ... )
        try record_parser <|> -- MkRec { ... }
        adt_parser <|> -- MkADT ...
        number <|>    -- signed integer
        quoted_string <?> "Parse error"
```

Parsec will try all choices in order of occurrence. Remember that `try` is used to avoid consuming the input.

Main program. Import the parser module

```
    import ShowParser (parseShow)
```

Use the parser

```
    rec_str =  show [rec1,rec2]
    main = putStrLn $ parseShow rec_str
```

Test it:

```
[wim@workai HaskellParsecTutorial]$ runhaskell test_ShowParser.hs
<?xml version="1.0" encoding="UTF-8"?>
<list><list-elt><record name="MkPersonRecord"><elt key="name">"Wim Vanderbauwhede"</elt>
<elt key="address"><record name="MkAddress"><elt key="line1">"School of Computing Scienc
<elt key="number">17</elt>
<elt key="street">"Lilybank Gdns"</elt>
<elt key="town">"Glasgow"</elt>
```

```
<elt key="postcode">"G12 8QQ"</elt></record></elt>
<elt key="id">557188</elt>
<elt key="labels"><list><list-elt><adt>Green</adt></list-elt>
<list-elt><adt>Red</adt></list-elt></list></elt></record></list-elt>
<list-elt><record name="MkPersonRecord"><elt key="name">"Jeremy Singer"</elt>
<elt key="address"><record name="MkAddress"><elt key="line1">"School of Computing Science"</el
<elt key="number">17</elt>
<elt key="street">"Lilybank Gdns"</elt>
<elt key="town">"Glasgow"</elt>
<elt key="postcode">"G12 8QQ"</elt></record></elt>
<elt key="id">42</elt>
<elt key="labels"><list><list-elt><adt>Blue</adt></list-elt>
<list-elt><adt>Yellow</adt></list-elt></list></elt></record></list-elt></list>
```

Summary.

- Parsec makes it easy to build powerful text parsers from building blocks using predefined parsers and parser combinators.
- The basic structure of a Parsec parser is quite generic and reusable
- The example shows how to parse structured text (output from Show) and generate an XML document containing the same information.