JSON Parser in Haskell

Assignment 2

Sam Grumley

Programming Languages
3802ICT

Griffith University
School of Information and Communication Technology
S5048240
5/10/2020

1 Introduction

This report both documents and demonstrates the implementation of a JSON file parser writen in Haskell. The EBNF and syntax diagrams have been constructed by the program Syntrax. The implementation is built around the third party module: ABR.

2 Instructions

Run the below commands in terminal to compile and parse your chosen JSON file

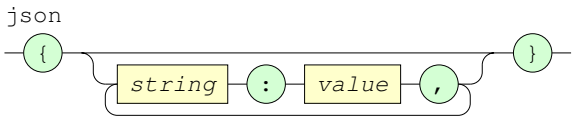
```
ghc json-parser.hs
json-parser input.json
```

3 JSON EBNF

EBNF and syntax diagrams:

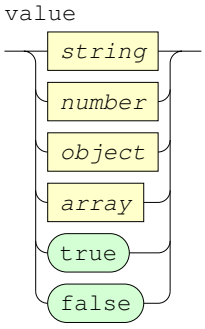
3.1 JSON Object

```
json ::= "{" { $string$ ":" $value$ ","} "}".
```



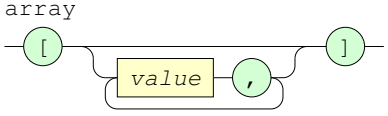
3.2 JSON Value

```
value ::= $string$ | $number$ | $object$ | $array$ | "true" | "false" .
```



3.3 JSON Array

```
array ::= "["  {$value$ ","}  "]".
```

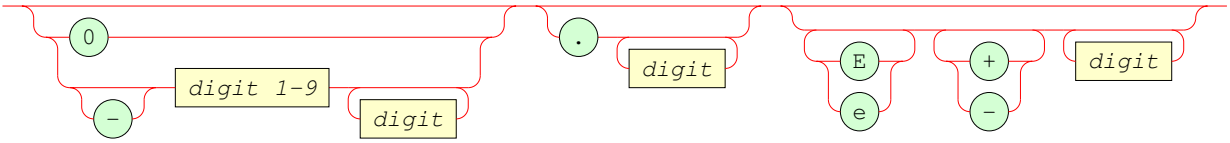


3.4 JSON Number

JNumber was not manually written as the ABR library handles number with the funcion floatL.

```
number ::= ["0" | ["-"] $digit 1-9$ {$digit$}]
["." {$digit$}]
[{"E" | "e"} {"+" | "-"} {$digit$}];
level="lexical".
```

number

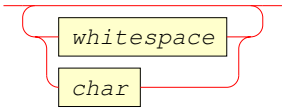


3.5 JSON String

JString was not manually written as the ABR library handles number with the funcion stringL.

```
string ::=  {$whitespace$ | $char$} ;
level="lexical".
```

string

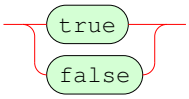


3.6 JSON Bool

JString was not manually written as the ABR library handles number with the funcion stringL.

```
bool ::= "true" | "false";
level="lexical" .
```

bool

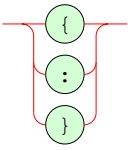


3.7 JSON Object Symbols

JString was not manually written as the ABR library handles number with the funcion stringL.

```
objectSymbols ::= "{" | ":" | "}";
level="lexical" .
```

objectSymbols

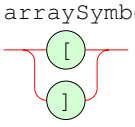


3.8 JSON Array Symbols

JString was not manually written as the ABR library handles number with the funcion stringL.

```
arraySymbols ::= "[" | "]";
level="lexical" .
```

arraySymbols



3.9 JSON Reapeatable Symbols

JString was not manually written as the ABR library handles number with the funcion stringL.

```
repeatableSymbols ::= ",";
level="lexical" .
```

4 Implementation

4.1 Data Structure

```
data
```

```
module Main (main) where
```

```
import System.Environment
```

```
import ABR.Util.Pos
import ABR.Parser
import ABR.Parser.Lexers
```

This type represents a JSON Object.

```
data JValue
    = JNumber Float
    | JString String
    | JArray[JValue]
    | JObject [(String, JValue)]
    | JBool Bool
    deriving(Show,Eq)


data JSON =  JValue JValue
               deriving Show
```

4.2 Lexers

`lexer` Lexer for boolean values

```
boolL :: Lexer
boolL = tokenL "true" %> "true"
    <|> tokenL "false" %> "false"
```

Lexer for array type symbols

```
arrayL :: Lexer
arrayL = literalL '[' %> "StartOfArray"
    <|> literalL ']' %> "EndOfArray"
```

Lexer for object type symbols

```
objectL :: Lexer
objectL = literalL '{' %> "StartOfObject"
    <|> literalL '}' %> "EndOfObject"
    <|> literalL ':' %> "PairDelimiter"
```

Lexer for more than one value symbols

```
repeatableL :: Lexer
repeatableL = literalL ',' %> "AnotherValue"
```

```
programL :: Lexer
programL = dropWhite $ nofail $ total $ listL
    [whitespaceL, floatL, repeatableL, stringL, arrayL, objectL, boolL]
```

4.3 Parser

`parser` Parse String

```
jstringP :: Parser String
jstringP = tagP "string"
    @> (\(_,s,_) -> s)
```

Parse Key : Value

```
keyValP :: Parser (String, JValue)
keyValP = jstringP            --tagP "string"
        <&> tagP "PairDelimiter"
        &> valueP
        @> (\a -> a)
```

Parse Any value

```
valueP :: Parser JValue
valueP =
       jstringP
       @> (\n -> JString n )
       <|> tagP "float"
       @> (\(_,n,_) -> JNumber (read n))
       <|> arrayP
       @> (\ n-> JArray (concat n))
       <|> objectP
       @> (\ n-> JObject (concat n))
       <|> tagP "true"
       @> (\(_,_,_) -> JBool True)
       <|> tagP "false"
       @> (\(_,_,_) -> JBool False)
```

Parse array

```
arrayP :: Parser [[JValue]]
arrayP =
    tagP "StartOfArray"
    <&> optional( valueP
        <&> many( tagP "AnotherValue"
        &> valueP
        )
        @> cons
    )
    <& nofail (tagP "EndOfArray" )
    @> (\((_, _, _), a ) -> a)
```

Parse JSON Object

```
objectP :: Parser [[(String, JValue)]]
objectP =
    tagP "StartOfObject"
    <&> optional( keyValP
    <&> many( tagP "AnotherValue"
        &> keyValP
        )
        @> cons
    )
    <& nofail (tagP "EndOfObject" )
    @> (\(_, a ) -> a)
```

```
programP :: Parser JSON
programP = nofail $ total (
    valueP @> JValue
   )
```