JSON Parser and Validator

Caleb Howard

October 6, 2020

Abstract

This document covers the JSON Parser and Validator haskell program. It highlights the use cases of the program and explains key sections in the source code. EBNF diagrams are included to illustrate the parser syntax.

1 Introduction

The JSON Parser and Validator haskell program is responsible for parsing and comparing two JSON documents. The program reads a JSON data document and a JSON schema document. After parsing both documents into separate parse trees, both parses trees are used to verify that the schema document is a valid schema for the data document.
Before constructing the parser, EBNF diagrams were created to illustrate the lexical and context-free syntax of a JSON document. The Syntrax tool was used to generate the syntax (railroad) diagrams from .ebnf productions. For more information or to view these diagrams, refer to section 2, Syntax.

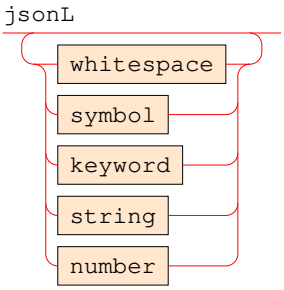
2 Syntax

EBNF and syntax diagrams for JSON Documents:

2.1 Lexical Syntax

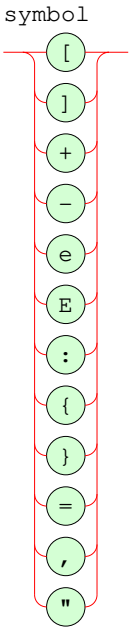
2.1.1 JsonL - All JSON document lexemes

```
jsonL ::= {whitespace | symbol | keyword | string | number};
level = "lexical".
```



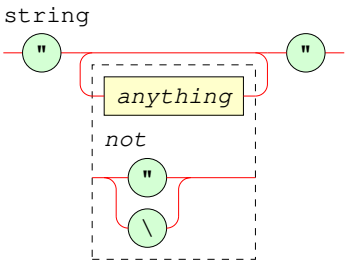
2.1.2 Symbol

```
symbol ::= "[" | "]" | "+" | "-" | "e" | "E" | ":" | "{" | "}" | "=" | "," | "\"";
level="lexical".
```



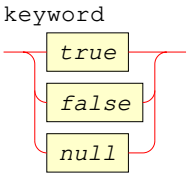
2.1.3 String

```
string ::= "\"" {<$anything$ ! ("\"" | "\\")>} "\"";
level="lexical".
```



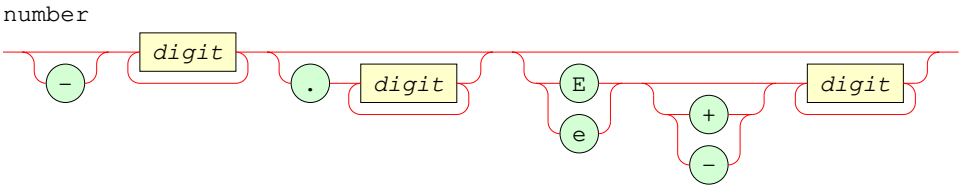
2.1.4 Keyword

```
keyword ::= $true$ | $false$ | $null$;
level = "lexical".
```



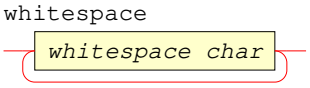
2.1.5 Number

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



2.1.6 Whitespace

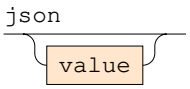
```
whitespace ::= {$whitespace char$}+;
level="lexical".
```



2.2 Context-Free Syntax

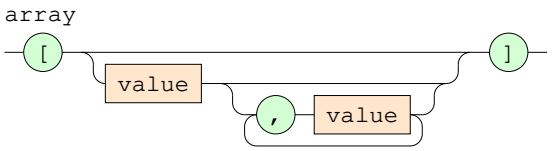
2.2.1 Json

```
json ::=  [value].
```



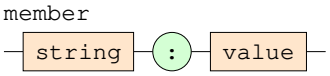
2.2.2 Array

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



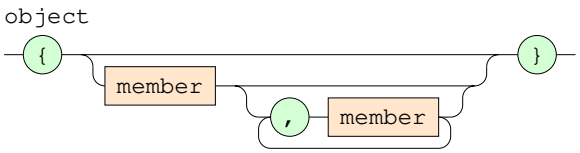
2.2.3 Member

```
member ::= string ":" value.
```



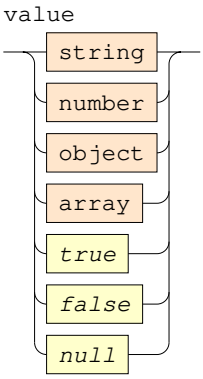
2.2.4 Object

```
object ::= "{" [member {"," member}] "}".
```



2.2.5 Value

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



3 Code

3.1 JsonTypes Module

The JsonTypes module holds Haskell data types for JSON data. These types are used by the the JsonParser and JsonValidator modules. The JsonMember type holds information for a JSON object's member, it consists of a String and JsonValue. The JsonValue data type is a recursive type, it holds data for items that can either be standalone in a JSON document, or can be a value in a "string":value JSON member.

```
module JsonTypes where

-- JsonMember
data JsonMember = JsonMember String JsonValue
    deriving (Show)

-- Added because the validator requires a lookup for type JsonMember.
lookupInMember :: String -> [JsonMember] -> Maybe JsonValue
lookupInMember n ms = lookup n ms'
  where
    ms' = map toTuple ms
    toTuple (JsonMember n v) = (n,v)

-- JsonValue
data JsonValue =
      JsonNull
    | JsonBool Bool
    | JsonNumber Double
    | JsonString String
    | JsonArray [JsonValue]
    | JsonObject [JsonMember]
    deriving (Show)
```

3.2 JsonParser Module

The JsonParser module handles all parsing of a given JSON document. It uses the ABR library to construct lexers and parsers which correlate to the syntax diagrams listed in section 2, Syntax. The parsers use the produced lexemes to create parse trees for two documents, or if any JSON syntax errors exist, a relevant error message will be displayed. The filenames of the two JSON documents are passed as command line arguments. The first command line argument is a reference to a JSON data document, the second is a reference to a JSON schema document, which is to be used by the JsonValidator module to validate the schema against the data document. Ultimately, this module will print whether the schema validation was successful or failed.

```
-- Json  Parser

module Main (main) where

import System.Environment
import System.IO
import Data.Char

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

import JsonValidator (validator)
import JsonTypes

-- Lexers ----------------------------------------------------------------

symbolL :: Lexer
symbolL = literalL '{' <|> literalL '}' <|> literalL '[' <|>
          literalL ']' <|> literalL ',' <|> literalL '=' <|>
          literalL ':'

-- For identifying keywords such as "true", "false", "null"
keywordL :: Lexer
keywordL =
       some (satisfyL isAlpha "")
   &%> "keyword"


jsonL :: Lexer
jsonL = dropWhite $ nofail $ total $ listL
    [whitespaceL, symbolL, keywordL, stringL, signedFloatL]

-- Parsers ---------------------------------------------------------------

arrayP :: Parser JsonValue
arrayP =
        literalP "'['" "["
     &> optional ((    valueP
                   <&> many (   literalP "','" ","
                             &> valueP)
                   @> (\(v, vs)-> (v:vs))))
    <&  literalP "']'" "]"
    @> (\(vs) -> case vs of
        [] -> JsonArray []
        (vs':_) -> JsonArray vs'
       )

memberP :: Parser JsonMember
memberP =
```

```
        tagP "string"
     <&> literalP "':'" ":"
      &> valueP
      @> (\((_,s,_), v) -> JsonMember s v)

objectP :: Parser JsonValue
objectP =
        literalP "'{'" "{"
     &> optional ((    memberP
                   <&> many (   literalP "','" ","
                             &> memberP)
                   @> (\(m, ms)-> (m:ms))))
    <&  literalP "'}'" "}"
    @> (\ms -> case ms of
        []     -> JsonObject []
        (ms':_) -> JsonObject ms'
       )



valueP :: Parser JsonValue
valueP =
        tagP "string"
        @> (\(_,s,_) -> JsonString s)
    <|> tagP "signedFloat"
        @> (\(_,n,_) -> JsonNumber (read n))
    <|> literalP "keyword" "true"
        @> (\_ -> JsonBool True)
    <|> literalP "keyword" "false"
        @> (\_ -> JsonBool False)
    <|> literalP "keyword" "null"
        @> (\_ -> JsonNull)
    <|> arrayP
    <|> objectP


jsonP :: Parser JsonValue
jsonP = nofail $ total $ valueP

main :: IO ()
main = do
    [json, schema] <- getArgs
    putStrLn "----- Json Data Document ------"
    j <- interpret json
    putStr "\n----- Json Schema Document ------\n"
    s <- interpret schema
    if validator j s
        then putStrLn "----- Verification Successful ------"
        else putStrLn "----- Verification Failed ------"


interpret :: FilePath -> IO JsonValue
interpret doc = do
    source <- readFile doc
    let cps = preLex source -- chars and positions
    putStrLn "----- Chars and Positions ------"
    print cps
    case jsonL cps of
        Error pos msg -> do
            error $ errMsg pos msg source
```

```
OK (tlps,_) -> do
    putStrLn "----- Lexemes ------"
    print tlps -- tags lexemes positions
    case jsonP tlps of
        Error pos msg -> do
            error $ errMsg pos msg source
        OK (json,_) -> do
            putStrLn "----- Parse Tree ------"
            print json
            return json
```

3.3 JsonValidator Module

The JsonValidator module handles the verification of a provided JSON schema document against a provided JSON data document. The module uses the parse trees created in the JsonParser. Firstly, the parse tree of the JSON schema is compiled into a different tree structure. The compilation involves converting all the {string : value} objects into JsonSchema types. The compilation stage also loads any custom schema types using the JsonSchemaObject data constructor. To further understand how to the JsonSchemaObject data constructor works, refer to the code comments surrounding its definition.
The compilation is done to make the pattern matching in the validate function more readable and neat.

For the most part, the validate function uses only pattern matching. However, when we encounter a JsonSchemaObject JsonObject pattern match, we must also confirm the JsonObject has a 1 to 1 matching for its members and the customTypes defined in the schema. Refer to the code comments in the code piece below for a step by step explanation of how this part of validation was implemented.

```
module JsonValidator (validator) where
import Data.List (partition)
import JsonTypes

data JsonSchemaType =
      SchemaNull
    | SchemaBool
    | SchemaInt
    | SchemaFloat
    | SchemaString
    | SchemaArray
    | SchemaObject
    | SchemaCustom String
    deriving (Show, Eq)

type MemberSchemas = [(String, JsonSchema)] -- String is the member (property) name, schema is the t
type CustomSchemas = [(String, JsonSchema)] -- String is the custom type name, schema is the schema

-- A schema defining a custom type could look like:
-- JsonSchemaObject
--   [("name", JsonSchemaValue (SchemaCustom "customNameType"))]
--   [("customNameType", JsonSchemaValue SchemaString)]

data JsonSchema =
    JsonAll
  | JsonSchemaArray
  | JsonSchemaValue JsonSchemaType
  | JsonSchemaElementArray JsonSchema
  | JsonSchemaObject MemberSchemas CustomSchemas
  deriving (Show, Eq)


compileSchema :: JsonValue -> JsonSchema
-- Empty
compileSchema (JsonObject []) =
    JsonAll
-- Null
compileSchema (JsonObject [JsonMember "\"type\"" (JsonString "\"null\"")])   =
    JsonSchemaValue SchemaNull
-- String
compileSchema (JsonObject [JsonMember "\"type\"" (JsonString "\"string\"")]) =
    JsonSchemaValue SchemaString
-- Int
compileSchema (JsonObject [JsonMember "\"type\"" (JsonString "\"int\"")])    =
    JsonSchemaValue SchemaInt
-- Float
```

```
compileSchema (JsonObject [JsonMember "\"type\"" (JsonString "\"float\"")])  =
    JsonSchemaValue SchemaFloat
-- Bool
compileSchema (JsonObject [JsonMember "\"type\"" (JsonString "\"bool\"")])   =
    JsonSchemaValue SchemaBool
-- Array with element type specification
compileSchema (JsonObject [JsonMember "\"type\"" (JsonString "\"array\""), JsonMember "\"elements\""
    JsonSchemaElementArray (compileSchema e)
-- Array
compileSchema (JsonObject [JsonMember "\"type\"" (JsonString "\"array\"")]) =
    JsonSchemaArray
-- Object
compileSchema (JsonObject ((JsonMember "\"type\"" (JsonString "\"object\"")):props) ) =
    let
      (props', schemas) = partition (\(JsonMember name _) -> name /= "\"schemas\"") props
      schemas' =
        case schemas of
          ((JsonMember _ (JsonObject customSchemaTypes)) : _) -> map compileMember customSchemaTypes
          [] -> []
          _ -> error "Invalid schemas definition value"
    in
      JsonSchemaObject (map compileMember props') schemas'
    where
      compileMember (JsonMember name memberSchema) = (name, compileSchema memberSchema)
-- Custom
compileSchema (JsonObject [JsonMember "\"type\"" (JsonString custom)]) =
    JsonSchemaValue (SchemaCustom custom)

-- Invalid schema
compileSchema s = error $ "failed to compile - " ++ (show s) ++ "\n*** Invalid json schema."

isInt :: RealFrac a => a -> Bool
isInt x = x == fromInteger (round x)

validate :: JsonSchema -> JsonValue -> Bool
validate  JsonAll                       _                = True
validate (JsonSchemaValue SchemaNull) JsonNull         = True
validate (JsonSchemaValue SchemaBool) (JsonBool _)     = True
validate (JsonSchemaValue SchemaInt) (JsonNumber x)    = isInt $ x
validate (JsonSchemaValue SchemaFloat) (JsonNumber x)  = not $ isInt $ x
validate (JsonSchemaValue SchemaString) (JsonString _) = True
validate (JsonSchemaElementArray elementSchema) (JsonArray elements) =
    all (== True) . map (validate elementSchema) $ elements
validate (JsonSchemaArray) (JsonArray _) = True
validate (JsonSchemaObject memberSchemas customTypes) (JsonObject members) =
    -- An object is validated if all of its members are validated
    -- To make it 1:1 (disallow missing)
    all (== True) . map validateMemberSchema $ memberSchemas
    where
      -- To validate a member schema for the presence of the members
      validateMemberSchema (name, memberSchema) =
        -- look up the member within the member list
        case lookupInMember name members of
          -- If it's there, validate its value against the schema
          Just value -> validateMemberWith memberSchema value customTypes
          -- Missing members not allowed
          Nothing -> False

      -- Special validate function that only works for members
      -- And makes use of local custom types.
```

```
      -- If the member is defined as a custom type,
      validateMemberWith (JsonSchemaValue (SchemaCustom customTypeName)) v customTypes =
        -- try to find the custom type definition
        case lookup customTypeName customTypes of
          -- if it's there, validate the member against the custom type schema
          Just customSchema -> validate customSchema v
          -- If the custom type for the member is unknown, fail validation
          Nothing -> False

      -- If the member is not defined as a custom type, just validate it as any other value
      validateMemberWith memberSchema v _ = validate memberSchema v

validator :: JsonValue -> JsonValue -> Bool
validator dataDoc schemaDoc = validate schema dataDoc
  where schema = compileSchema schemaDoc
```