3802ICT Programming Languages - Assignment 2

Jesse Schneider

September 27, 2020

Abstract

This report is targeted at investigating EBNF and parsing for the JavaScript Object Notation (JSON) data-interchange format. It includes EBNF definitions, a Haskell JSON Data Type, a JSON Lexer and Parser written in Haskell and Validation of the parser.

1 Task 1: JSON EBNF

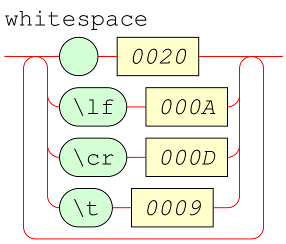
For this report, we have 2 different sections of EBNF defined: Lexical syntax and Context-free syntax. Our Lexical EBNF is used to define Lexical tokens that will be in the parsed content. The Context-free rules will define how we combine the Lexical tokens to define rules, in this instance defining how JSON will be interpreted.

1.1 Lexical Syntax Rules

Here is the Lexical EBNF and Railroad Diagrams drawn from those rules, to display the different Lexical Tokens within JSON:

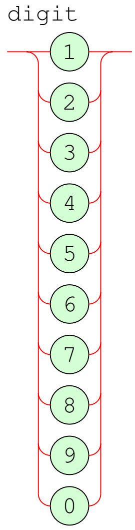
Whitespace - Spaces, Line Feeds, Carriage Returns, Tabs

```
whitespace ::= { " " $0020$| "\lf" $000A$ | "\cr" $000D$ | "\t" $0009$ }+ ;

level="lexical".
```



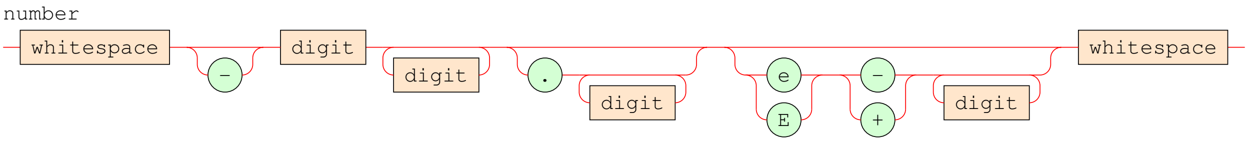
Digits - All digits from 0 - 9

```
digit ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "0";

level="lexical".
```



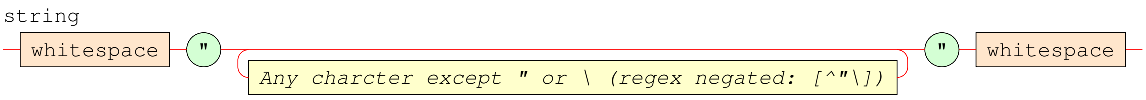
Numbers - positive and negative Integer, Decimal, Exponential

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

level="lexical".
```



Strings - A collection of any characters grouped together

```
string ::= whitespace "\""
    { $Any charcter except " or \\ (regex negated: [^"\])$ }
    "\"" whitespace;

level="lexical".
```

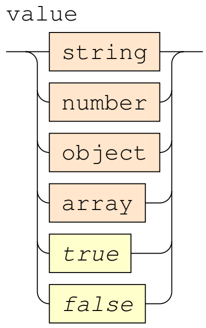


1.2 Context-Free Syntax Rules

Here is the Context-Free EBNF and Railroad Diagrams drawn from those rules, to demonstrate how the Lexical Tokens can be combined within JSON:

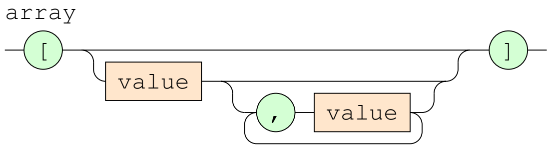
Values - Numbers, Strings, Arrays, Objects, True, False

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



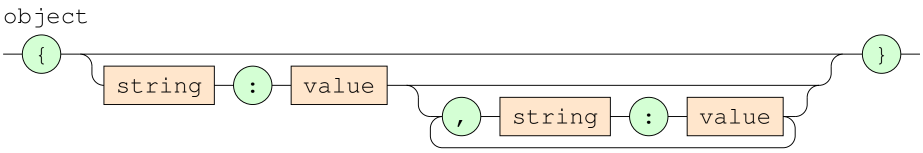
Arrays - A collection of any Values

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



Objects - A (key:value) type data structure to store any type of Value

```
object ::= "{"  [ string ":" value { "," string ":" value } ] "}".
```



2 Task 2: Haskell JSON Data Type

An Algebraic Haskell Data Type has been designed to store JSON as seen here:
Note: Array and Object store repeated JSON objects, so as to contain any other Type of JSON Type.

```
module Json where
```

JSON Data Types

Key Value pair data type to show what is inside a JSON Object:

```
data KeyValue = KeyValue (String, Json) deriving Show
```

JSON Data Type:

```
data Json = String String
            | Num Float
            | Object [KeyValue]
            | Array [Json]
            | Bool Bool deriving (Show)
```

3 Task 3: Json Lexers + Parsers

Now that we have a basic idea of what we need for our Lexers and Parsers, its a lot easier to implement them.

```
module JsonParser where
import ABR.Util.Pos
import ABR.Parser
import ABR.Parser.Lexers
import Data.Char
import Json
```

Input Data Type for user input:

```
data Input = Json Json deriving Show
```

Lexers

Boolean value lexers:

```
trueL :: Lexer
trueL = tokenL "true" %> "true"

falseL :: Lexer
falseL = tokenL "false" %> "false"
```

Symbol Lexer to find all symbols in JSON:

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

This is a list of Lexers, all the ones we need to use to get JSON Lexemes:

```
inputL :: Lexer
inputL = dropWhite $ nofail $ total $ listL
    [whitespaceL, floatL, stringL, literalL 'q', symbolL, trueL, falseL]
```

Parsers

Our input Parser, parsing our Json Lexemes at the highest level:

```
inputP :: Parser Input
inputP = nofail $ total (
     jsonP @> Json
    )
```

JSON value Parser, a Parser than can read identify values within JSON:

```
jsonP :: Parser Json
jsonP =
        tagP "string"
        @> (\(_, x, _) -> String x)
    <|> tagP "float"
        @> (\(_, x, _) -> Num (read x))
    <|> tagP "true"
        @> (\(_, x, _) ->  Bool True )
    <|> tagP "false"
        @> (\(_, x, _) -> Bool False)
    <|> arrayP
        @> (\x -> Array x)
    <|> objectP
        @> (\j -> Object j)
```

JSON Array Parser:

```
arrayP :: Parser [Json]
arrayP =
    literalP "'['" "["
    <&> optional (
        jsonP
        <&> many (
                    literalP "','" ","
                    &> nofail' "value expected" jsonP
            )
        @> cons
    )
    <& nofail (literalP "']'" "]")
    @> (\((_,_,_), ars) -> concat ars)
```

JSON Object Parser:

```
objectP :: Parser [KeyValue]
objectP =
    literalP "'{'" "{"
    <&> optional (
            keyValueP
        <&> many (
                literalP "','" ","
                &> nofail' "value expected" keyValueP
            )
        @> cons
    )
    <& nofail (literalP "'}'" "}")
    @> (\((_,_,_), ars) -> concat ars)
```

Object Key Value Pair Parser:

```
keyValueP :: Parser KeyValue
keyValueP =
    tagP "string"
    <&> nofail (literalP "':'" ":")
    &> nofail' "value expected" jsonP
    @> (\((_,l,_),v) -> KeyValue (l, v))
```

Parsing Test Program

```
module Main (main) where
import System.IO
import ABR.Util.Pos
import ABR.Parser

import JsonParser as JS
```

Here is our main function to:
- read Input
- prelex the Input into [(Character, Position)]
- Lex the prelex pairs into lexemes
- Parse the output Lexemes
- Display the output or any errors

```
main :: IO ()
main = do
   json <- readFile "object.json"
   let error :: Pos -> Msg -> IO ()
       error (_,col) msg = do
          putStrLn $ "Error: " ++ msg
          putStrLn json
          let col' = if col < 0
                 then length json
                 else col
          putStrLn $ replicate col' ' '
             ++ "^"
          main
       cps = preLex json
   case inputL cps of
      Error pos msg -> error pos msg
      OK (tlps,_) -> do
         case inputP tlps of
            Error pos msg -> error pos msg
            OK (input,_) -> do
               case input of
                  JS.Json j -> do
                     putStrLn $ "ParseTree: " ++ show j
```

Parsing Example

Here is our input test JSON:

```
{
    "Name": "John",
    "Age": 36,
    "Cars": [
        {"type": "Mustang","age": 3},
        {"type": "Ferrari","age": 1}
    ]
}
```

After execution, this is what our Parse Tree looks like:

```
ParseTree: Object [
    KeyValue ("\"Name\"",String "\"John\""),
    KeyValue ("\"Age\"",Num 36.0),
    KeyValue ("\"Cars\"",
        Array [Object [
            KeyValue ("\"type\"",String "\"Mustang\""),
            KeyValue ("\"age\"",Num 3.0)],
                Object [
                    KeyValue ("\"type\"",String "\"Ferrari\""),
                        KeyValue ("\"age\"",Num 1.0)]])]
```