3802ICT Programming Languages, Trimester 2, 2020
Assignment 2

School of Information and Communication Technology
Griffith University

September 16, 2020

Marks	25
Due	11:59 pm, Sunday 4 October, 2020.

1 Problems

Problem 1: JSON EBNF (6 marks)

JSON is a popular format for interchange of structured data. The syntax for JSON is specified here: https://www.json.org/json-en.html. That page specifies the syntax using versions of syntax diagrams (on the left) and BNF (on the right).

Your task is to rewrite a *subset* of this specification in the version of EBNF defined as input to the `syntrax` tool, and to use `syntrax` to verify your EBNF by generating new syntax diagrams.

To simplify your task, you may[1]:

- simplify strings by omitting all escape sequences;
- omit the keyword `null`;
- copy and adapt any of the sample EBNF from the lecture notes; and
- put each EBNF production in a separate text file that can be included in your report.

Additional requirements:

- You must categorize your EBNF productions as either belonging to the lexical or context free grammar levels of concern in your report, and by use of metadata on the lexical productions, `level="lexical"`, so that the tracks are drawn red.
- Your context free grammar productions should not mention whitespace.

Problem 2: Haskell Type for JSON Data (5 marks)

Design and implement Haskell data types for JSON data. These will be the types output by your parsers in problem 3.

Problem 3: JSON Lexer and Parser (7 marks)

Implement a lexers and parsers that follow your EBNF specification for JSON.

Please note that there are many implementations of JSON parsers in Haskell and other languages available on the web. You may use any of them for hints and inspiration, but your implementation *must* use module `ABR.Parser` and related modules as the basis for the combinator parsing strategy. Do not use any other preexisting parsing related packages, modules, or libraries.

Implement a test program that parses a file containing some JSON and prints the parse tree using a (possibly derived) `Show` instance. Demonstrate this program in your report.

[1] By "may" I mean recommended but not required.

Problem 4: JSON Validation (7 marks)

Successfully parsed JSON is just that, but only that. The important question from the point of view of an application that wants to ingest this data is whether it is structured according to the needs of the application. A JSON schema is a document in JSON format that specifies the structure required of particular JSON data.

Write a Haskell program that reads two files:

1. a JSON data document with data to be validated,
2. a JSON schema document with a schema (defined below) against which the data should be validated.

Demonstrate this program in your report.

Simplified JSON schemas

This defines a simplified notation for JSON schemas with examples.[2]

A JSON data document is a document containing JSON data to be validated against a *JSON schema document*.

A JSON schema document is a document containing JSON data that specifies the format of a valid *JSON data document*.

The top level element in a JSON schema document is a *schema object* that will match the whole of a valid JSON data document.

For example this JSON schema document

```
{"type": "bool"}
```

matches this JSON data document.

```
true
```

A schema object is an object that defines what will be matched by whole or part of a JSON data document.

The simplest schema object has no members.[3]

```
{}
```

and matches any JSON value without checking.

A schema object may have a `type` member and perhaps other members depending on the `type` member. The predefined values for the `type` member are:

type	*matching values*	*other members permitted*
`bool`	`false` or `true`	no
`int`	signed whole numbers	no
`float`	signed numbers, whole or floating point	no
`string`	double-quote delimited strings	no
`array`	arrays	only `elements`
`object`	objects	yes

New values for the `type` member can be defined using a `schemas` member in `object` schemas.

For example this JSON schema object

```
{"type": "int"}
```

matches this JSON value.

[2] To see a more complete specification of JSON schemas, see https://json-schema.org.

[3] Both the data and the schemas are in JSON format. I have chosen to refer to the properties in the schemas as "members" and the properties in the data objects as "properties". Only arrays have "elements". I hope this helps.

```
42
```

The elements member for array type schemas specifies a schema to match *all* of the elements of an array. If the `elements` member is not present, any types of elements are permitted and they are not checked.

For example this JSON schema object

```
{
  "type": "array",
  "element": {"type": "int"}
}
```

matches this JSON value.

```
[1, 2, 4]
```

Most other members for object type schemas specify the *required* properties of the object values. The member name is the property name, and the member value is a schema that must match for that property's value. All of the property names should be unique within each object.

For example this JSON schema object

```
{
  "type": "object",
  "firstName": {"type": "string"},
  "lastName": {"type": "string"},
  "birthYear": {"type": "int"}
}
```

matches this JSON value.

```
{
  "firstName": "Shirley",
  "lastName": "Temple",
  "birthYear": 1928
}
```

The schemas member for object type schemas allows the definitions of new types, local to the current object type schema. Its value is an object, where the member names are the names of the new types, and their values are schemas.

For example this JSON schema object

```
{
  "type": "object",
  "name": {"type": "name"},
  "billingAddr": "address",
  "deliveryAddr": "address",
  schemas: {
    "name": {
      "type": "object",
      "firstName": {"type": "string"},
      "lastName": {"type": "string"}
    },
    "address": {
      "type": "object",
      "street": {"type": "string"},
      "suburb": {"type": "string"},
      "state": {"type": "string"},
```

```
      "postcode": {"type": "int"}
    }
  }
}
```

matches this JSON value.

```
{
  "name": {
    "firstName": "Scott",
    "lastName": "Morrison"
  },
  "billingAddr": {
    "street": "5 Adelaide Ave",
    "suburb": "Deakin",
    "state": "ACT",
    "postcode": 2600
  },
  "deliveryAddr": {
    "street": "5 Adelaide Ave",
    "suburb": "Deakin",
    "state": "ACT",
    "postcode": 2600
  }
}
```

2 Submission

- Put all of your EBNF in plain text files with names ending in `.ebnf`.
- Submit any generated `.eps` and `.pdf` files too.
- Put all the files for problem 1 in their own folder.
- Make sure your Haskell scripts are a plain text files, with a names ending in `.hs` or `.lhs`.
- Put all of the Haskell modules and data files for in another folder, containing subfolders for ABR library modules etc that are required to compile your programs.
- Submit your report file as a PDF. Your report should document the results you got for each problem. Preparing your report with LaTeX will make the job much easier, but that is not required.
- Put all this in a zip file and submit on Learning@Griffith.

3 Extensions

Extensions must only be applied for by the official online system, with appropriate documentation.

4 Help

- Ask questions in labs.
- Ask questions in the Assignment 2 chat on Teams. Post no code. Anything in English is OK. Watch what other people are asking for clues.
- I will provide a LaTeX report template as LaTeX sources, and some more example JSON data and schema files.

5 Feedback

- Marks to the marks centre on Learning@Griffith.
- Via the Rubric on Learning@Griffith.