Penerapan Regular Expression dalam parsing JSON

Reinhard Benjamin Linardi, 13515011
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung 40132, Indonesia
13515011@std.stei.itb.ac.id

***Abstrak* — Persoalan *string matching* merupakan persoalan yang sangat umum di bidang informatika. Persoalan ini melibatkan string pola (*pattern*) yang akan dicari pada teks (*text*). Salah satu cara untuk melakukan *string matching* adalah menggunakan *regular expression* (sering disingkat menjadi *regex* atau *regexp*). Berbeda dari algoritma *string matching* pada umumnya, pada *pattern regex*, simbol-simbol tertentu tidak mengandung arti sebenarnya. Simbol-simbol ini justru memberikan pengaturan pada *pattern* sehingga *string matching* menjadi lebih fleksibel. Fleksibilitas ini dapat dimanfaatkan untuk parsing JSON, suatu *data interchange format* standar untuk pengiriman data melalui internet.**

Kata kunci* — *regular expression, JSON, parsing, string matching

I. PENDAHULUAN

Dewasa ini, hampir semua orang di sekitar kita mengenal internet. Teknologi ini sudah menjadi kebutuhan kita sehari-hari. Banyak sekali hal yang bisa dilakukan dengan memanfaatkan internet, contohnya membuka *search engine*, dan mengecek media sosial. Akses internet tidak hanya melalui *browser*, aplikasi lain pun dapat menggunakan internet untuk berkomunikasi, misalnya aplikasi *chatting*.

Dalam komunikasi melalui internet, ada dua pihak, yaitu *sender* (pengirim) dan *receiver* (penerima). *Sender* maupun *receiver* dapat berupa perangkat apapun, baik komputer, *server*, *smartphone*, *tablet*, atau perangkat lain yang memiliki konektivitas ke internet. Karena internet menghubungkan semua perangkat secara global, maka semua perangkat harus dapat diidentifikasi secara unik untuk membedakan satu pernagkat dengan perangkat lainnya. Angka identifikasi unik inilah yang disebut sebagai *IP address*.

Saat *sender* ingin mengirimkan data kepada *receiver*, data terlebih dahulu di-*breakdown* menjadi data-data yang lebih kecil. Setiap data yang lebih kecil ini dikirimkan dalam bentuk paket data (*data packets*). Paket data ini tidak hanya berisi data itu sendiri, tetapi juga berisi *IP header*, yaitu semua informasi yang dibutuhkan untuk mengirimkan paket data, seperti *source address, destination address*, *checksum*, protokol yang digunakan, dll. Akibatnya, ukuran paket data menjadi lebih besar dari ukuran data itu sendiri. Maka, data yang dikirim sebaiknya *lightweight* (ringan) untuk memperoleh ukuran paket data sekecil-kecilnya.

Selain ukuran data, salah satu aspek yang perlu diperhatikan adalah format dari data, yaitu bagaimana data direpresentasikan. Jika masing-masing pengembang perangkat lunak (*developer*) menggunakan format yang berbeda-beda, sebelum melakukan pengiriman data, diperlukan kesepakatan bagaimana format data yang digunakan. Bayangkan jika Anda sebagai *developer* membuat suatu aplikasi berbasis web yang dapat diakses pada *browser*. Apakah kita harus membahas format data yang digunakan dengan setiap *developer browser*? Akan jauh lebih mudah jika para *developer* menentukan suatu standar *format* yang berlaku secara universal. Pada tahun 2001, Douglas Crockford, menciptakan JSON, yang sekarang menjadi salah satu *data interchange format* standar.

Meskipun masalah format data sudah dapat diselesaikan, kita harus ingat, bahwa internet adalah tempat paling tidak aman dan rentan terhadap *cybercrime*. Banyak *cybercriminal* berusaha mengambil maupun memanipulasi data-data yang dikirimkan melalui internet. Salah satu prinsip dari *security* adalah "*never trust any input*". Kita tidak dapat mengetahui apa yang terjadi pada data selama perjalanan. Maka, diperlukan *parsing input*, untuk mencegah terjadinya *undefined behavior* maupun *cyber attack* (seperti *SQL injection*). Dalam melakukan *parsing*, kita dapat menggunakan *library*, maupun membuat *parser* sendiri. Jika kita membuat *parser* sendiri, kita dapat memanfaatkan *regex* untuk *string matching* yang fleksibel, karena *syntax* sendiri pada umumnya fleksibel, seperti *ignore multiple whitespaces*, dll.

II. DASAR TEORI

A. Finite Automata

Finite Automata atau *finite-state machine* (*FSM*), merupakan jenis *automaton* yang paling sederhana. *Automata* sendiri merupakan suatu istilah yang muncul dari *automata theory*, sebuah bidang pada *theoretical computer science* yang mempelajari logika komputasi dari mesin abstrak sederhana, yang disebut *automata*. Sebuah model yang merepresentasikan mesin yang melakukan komputasi pada *input* melalui sejumlah keadaan (*state*) disebut *automaton*. Pada setiap *state*, terdapat fungsi transisi yang akan menentukan *state* berikutnya berdasarkan *state* saat ini. Kata *automaton* sendiri sangat erat dengan kata *automation*, yang berarti proses otomatis yang melakukan proses produksi tertentu. *Automata* yang paling *general* dan *powerful* saat ini adalah *Turing machine*.

Terdapat 4 jenis *automaton* secara umum, yaitu *finite-state machine*, *pushdown automata*, *linear-bounded automata*, dan *Turing machine*. *Finite automata* adalah *automaton* dimana jumlah *state* yang ada pada himpunan *Q* terbatas (*finite*). *Finite automata* merupakan mesin abstrak yang terdiri dari 5-*tuple* :

1. *A finite set of states*, *denoted as Q.*

2. *A finite set of input symbols*, *denoted as* Σ.

3. *A transition function*, *that takes as arguments a state and an input symbol and returns a state*, *denoted as* δ.

4. *A start state*, *one of the states in Q.*

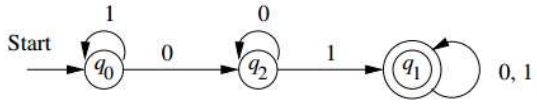
5. *A set of final or accepting states F. F is a subset of Q.*

B. Deterministic Finite Automata (*DFA*)

Deterministic Finite Automata (*DFA*) merupakan *finite automata* yang bersifat deterministik. Arti deterministik adalah, untuk suatu *input* tertentu, akan selalu menghasilkan *output* tertentu. Pada *DFA*, kelima *tuple* di atas dapat dituliskan dalam "*five-tuple*" *notation* :

$$A = (Q, \Sigma, \delta, q_0, F)$$

dimana *A* adalah nama *DFA*, *Q* adalah himpunan *states*, Σ adalah himpunan *input symbols*, δ adalah fungsi transisi, q_0 adalah *state* awal, dan *F* adalah himpunan *state* akhir. *DFA* juga dapat digambarkan sebagai *transition diagram*. Pada *transition diagram*, *circle* merepresentasikan *state*, *double circle* merepresentasikan *final state*, q_0 sampai q_2 merupakan nama *state*, *arrow* dari α ke β dengan *label* γ merepresentasikan fungsi transisi δ(α, γ) yang menerima parameter *state* α dan *input symbol* γ lalu mengembalikan *state* β sebagai hasilnya.

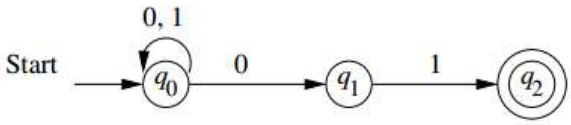


Gambar 1 - *Deterministic Finite Automata* (*DFA*)
Sumber : Introduction To Automata Theory Languages, and Computation, 3rd edition

Gambar di atas merupakan *transition diagram DFA* yang menerima semua *string* yang memiliki *substring* 01. *State* awal adalah q_0. Selama *input symbol* bukan 0, *state* akan tetap berada di q_0. Jika *input symbol* 0 sudah ditemukan, maka *state* akan berpindah ke q_2. Setelah *input symbol* 0 ditemukan, selama kita belum menemukan *input symbol* 1, maka *state* tetap berada di q_2. Jika *input symbol* 1 sudah ditemukan, maka *state* akan berpindah ke q_1. Saat berada di q_1, untuk *input symbol* apapun, kita akan tetap berada di q_1. Jika kita analisis, q_2 adalah *state* dimana *substring* 0 sudah ditemukan, sedangkan q_1 adalah *state* dimana *substring* 01 sudah ditemukan. *String* akan diterima jika berakhir di q_1 (*final state*).

C. Nondeterministic Finite Automata (*NFA*)

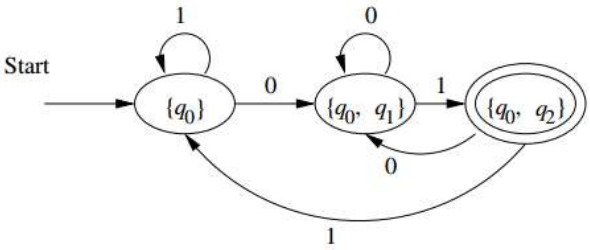
Nondeterministic Finite Automata (*NFA*) adalah *finite automata* yang bersifat non-deterministik, artinya untuk suatu *input* tertentu, *output*-nya bisa berbeda-beda. Perbedaan dari *DFA* dan *NFA* adalah pada fungsi transisinya (δ). Fungsi transisi pada *NFA* dapat mengembalikan himpunan *state*, berbeda dengan *DFA* yang mengembalikan satu *state* saja. Oleh karena itu, jika *NFA* digambarkan dengan *transition diagram*, pada suatu *state* tertentu, untuk *input symbol* tertentu, dapat tidak mempunyai transisi, atau punya transisi ke satu atau lebih *state*.



Gambar 2 - *Nondeterministic Finite Automata* (*NFA*)
Sumber : Introduction To Automata Theory Languages, and Computation, 3rd edition

Gambar di atas merupakan *transition diagram NFA* yang menerima semua string yang berakhir dengan angka 01. Dari *state* awal q_0, jika kita menerima *input symbol* 0, ada 2 pilihan, tetap berada di *state* q_0, atau berpindah ke *state* q_1. Pilihan inilah yang menyebabkan jalur yang dapat diambil menjadi banyak. Secara intuitif, untuk kasus di atas, jika *input* adalah suatu *string* dengan panjang *n*, maka jalur yang akan mencapai *final state* adalah mulai dari q_0, untuk *n* - 2 *input symbol* berikutnya, tetap berada di q_0, lalu untuk 2 *input symbol* terakhir, pindah dari *state* q_0 jika memungkinkan. Maka, untuk mencapai *final state*, sebenarnya hanya dibutuhkan satu jalur dengan pilihan yang tepat jika *input*-nya memang berakhir dengan 01. Pada *NFA*, jika *input* diterima oleh minimal satu jalur, maka *input* tersebut diterima oleh *NFA*.

NFA dapat diubah menjadi *DFA* dengan cara mengubah pilihan-pilihan tadi menjadi suatu *state* yang berisi himpunan *state* yang dapat dipilih. Pada contoh di atas, jika pada *state* q_0 kita menerima *input symbol* 0, *state* akan berpindah ke *state* $\{q_0, q_1\}$.



Gambar 3 – *DFA* yang ekivalen dengan **Gambar 2**
Sumber : Introduction To Automata Theory Languages, and Computation, 3rd edition

Dengan adanya konversi dari *NFA* ke *DFA*, maka sebenarnya *NFA* dan *DFA* menerima *language* yang sama, yaitu *regular language*. Pada *computer science*, sebuah *language* adalah *subset* dari semua *word* yang mungkin, dimana sebuah *word* adalah *concatenation of symbols*. Masing-masing *symbol* yang digunakan untuk membentuk *word* disebut *alphabet*. Contoh dari beberapa *word* yang dapat dibentuk oleh *alphabet* {0, 1} adalah 0, 1, 00, 01, 10, dan 11. Contoh sebuah *language* adalah himpunan semua *string* yang berakhir dengan 01. *Symbol* dan *string* memiliki pengertian secara matematis yang tidak akan dibahas pada makalah ini.

Sebuah *language* disebut *regular language* jika terdapat suatu *finite automata M*, dimana *language* dari *M* sama dengan *language* yang dimaksud. *M* dapat berupa *DFA*, atau *NFA* (berbentuk graf), maupun *regular expression* (berbentuk notasi seperti aljabar).

D. Regular Expression (Automata Theory)

Regular expression pada *automata theory* merupakan suatu notasi yang dapat mendefinisikan semua *language* yang merupakan *regular language*. *Regular expression* dapat dijabarkan secara rekursif, dengan $L(x)$ menandakan *language* dari *x*.

Basis :

1. ε (*empty string*) dan ∅ (*empty set*) merupakan *regular expression*, dengan $L(\varepsilon) = \{\varepsilon\}$, dan $L(\emptyset) = \emptyset$.

2. Jika *a* adalah *symbol*, maka *a* adalah *regular expression* dengan $L(a) = \{a\}$.

Rekurens :

1. Jika *E* dan *F* adalah *regular expression*, maka $E + F$ adalah *regular expression* yang merupakan gabungan (*union*) dari *E* dan *F*. Sehingga, $L(E + F) = L(E) \cup L(F)$.

2. Jika *E* dan *F* adalah *regular expression*, maka *EF* adalah *regular expression* yang merupakan *concatenation* dari *E* dan *F*. Sehingga, $L(EF) = L(E)\ L(F)$.

3. Jika *E* adalah *regular expression*, maka *E** adalah *regular expression* yang merupakan *closure* dari *E*. Sehingga $L(E^*) = L((E)^*)$.

4. Jika *E* adalah *regular expression*, maka (*E*) juga merupakan *regular expression*. Sehingga, $L((E)) = L(E)$.

Tanda *asterisk* (*) di atas menandakan *Kleene closure* dari sebuah *language L*. Definisi *Kleene closure* dari *L* (*L**) adalah "*the set of those strings that can be formed by taking any number of strings from L, possibly with repetitions, and concatenating all of them*" (Hopcroft, 87). Sebagai contoh, jika $L = \{0, 1\}$, maka $L^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$.

Pada *regular expression*, terdapat *operator precedence*. Tanda * mempunyai *precedence* paling tinggi, kemudian diikuti oleh *concatenation* dan terakhir *union*. Tanda kurung (*parentheses*) mempunyai arti seperti ekspresi matematika, jika kita ingin melakukan *grouping* tanpa mengikuti *precedence* sebenarnya. Tanda kurung bebas digunakan jika dibutuhkan, bahkan ketika arti dari *regular expression* tanpa tanda kurung sama dengan arti jika menggunakan tanda kurung.

DFA dapat diubah menjadi *regular expression*, sedangkan *regular expression* dapat diubah menjadi *NFA*. Untuk persoalan yang sederhana seperti contoh-contoh pada *DFA* dan *NFA*, *regular expression*-nya bisa langsung kita tentukan secara intuitif. Contohnya, *regular expression* yang menerima semua *string* yang berakhir dengan 01 adalah **(0+1)*01**. Bagian **(0+1)*** artinya *closure* dari 0 atau 1, yang mencakup *empty string*, dan semua *string* lain dengan panjang berapapun yang terdiri dari kombinasi 0 dan 1. Bagian **01** secara eksplisit adalah bagian akhir *string*. Jadi, *regular expression* kita akan menerima *string* dengan bagian akhir 01 dengan bagian depannya *string* apapun. Untuk kasus yang lebih kompleks, terdapat cara menyusun dan menyederhanakan *regular expression*, tetapi tidak akan dibahas pada makalah ini.

E. Regular Expression

Regular Expression (disebut juga *regex* atau *regexp*) adalah salah satu metode *string matching* yang cukup penting pada dunia informatika sekarang ini. *Regex* sangat banyak aplikasinya, terutama untuk validasi, seperti validasi nama, nomor telepon, *email*, dll. *Regex* sendiri berbeda dari algoritma *string matching* seperti KMP dan Boyer-Moore, karena *regex* memiliki fleksibilitas yaitu tidak selalu *exact match*, tergantung pada *pattern* dari *regex*. *Regex* sendiri bisa berbeda-beda implementasi, *syntax*, dan kemampuannya tergantung *regex engine* yang digunakan. Salah satu jenis *regex* yang paling umum dan paling *powerful* adalah PCRE (*Perl Compatible Regular Expression*).

Alasan dari *regex* ini dibedakan dengan *regular expression* dari *automata theory* bukan hanya karena *regex* secara *practical* memiliki syntax yang jauh lebih kompleks, tetapi pada umumnya *regex* yang ada di komputer kita tidak lagi termasuk dalam *regular language*, tetapi sudah lebih *powerful* dari itu. *Regex* sudah memiliki kemampuan kondisional, rekursi, penamaan, penyimpanan nilai, dll. Berikut ini adalah beberapa *syntax* dari *regex* :

Character	Matches
^	Beginning of line
$	End of line
.	Any character (except newline)
*	Zero or more occurrences of preceding expression
+	One or more occurrences of preceding expression
?	Optional (zero or one occurrence) of preceding expression
[]	One character from the characters inside the bracket
[^]	One character from the characters which is not inside the bracket
-	Range

Character	Matches
\d	Any digit character [0-9]
\w	Any alphanumeric character [A-Za-z_0-9]
\s	Any whitespace character
\D	Any non-digit character
\W	Any non-alphanumeric character
\S	Any non-whitespace character
{m,n}	Between m and n occurrences (inclusive) of preceding expression
{m,}	m occurrences or more of preceding expression
|	Alternation (or)
(*expr*)	Capturing group
(?:*expr*)	Non-capturing group
\1	Backreference
(?<=)	Positive lookbehind (assertion)
(?<!=)	Negative lookbehind (assertion)
(?=)	Positive lookahead (assertion)
(?!=)	Negative lookahead (assertion)
\	Escape character

Sebagai contoh, *regex* untuk *match* semua *string* yang berakhir dengan 01 adalah **.*01$**. Tanda **.** akan *match* semua karakter sebanyak 0 atau lebih kali, lalu **01** akan *match* 01 secara literal (harafiah). Tanda **$** dimasukkan ke dalam *pattern* untuk memastikan bahwa 01 terdapat di akhir *string*.

Kemampuan *regex* tidak hanya bersumber dari tingkat fleksibilitas dari *pattern regex*, tetapi juga dari *regex engine* yang melakukan *matching regex pattern*. Pada *regex engine*, biasanya terdapat *flag* yang bisa di-*set* untuk memodifikasi pencarian, seperti *ignore case* (besar kecilnya huruf tidak diperhatikan), *global* (mencari semua *match* yang ada), dll.

F. JavaScript Object Notation (JSON)

JavaScript Object Notation (JSON) adalah salah satu *data interchange format* yang paling populer. JSON banyak digunakan untuk pengiriman data, maupun penyimpanan data yang relatif sederhana. JSON memiliki *syntax* yang diturunkan dari JavaScript, sehingga membuat JSON *fully compatible* dengan JavaScript. Seiring dengan berkembangnya teknologi seperti *dynamic web pages* (elemen HTML bersifat dinamis), AJAX (untuk *asynchronous data exchange*), jQuery (*framework* JavaScript), dan Node.js (*server-side* JavaScript), maka JavaScript sebagai *client-side language* semakin populer. Hal ini mempengaruhi orang-orang untuk menggunakan JSON sebagai *data interchange format*.

Syntax yang digunakan pada JSON cukup sederhana, dan relatif *simple*. JSON dibangun berdasarkan 2 struktur data yang *universal* :

1. *A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array*.

2. *An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence*.

Pada JSON, dikenal tipe data primitif, dan struktur data berdasarkan 2 struktur data di atas sebagai berikut.

1. *Object. An object is an unordered set of name/value pairs. An object begins with { (left brace) and ends with } (right brace). Each name is followed by : (colon) and the name/value pairs are separated by , (comma).*

2. *Array. An array is an ordered collection of values. An array begins with [(left bracket) and ends with] (right bracket). Values are separated by , (comma).*

3. *Value. A value can be a string in double quotes, or a number, or true or false or null, or an object or an array. These structures can be nested.*

4. *String. A string is a sequence of zero or more Unicode characters, wrapped in double quotes, using backslash escapes. A character is represented as a single character string. A string is very much like a C or Java string.*

5. *Number. A number is very much like a C or Java number, except that the octal and hexadecimal formats are not used.*

Whitespace boleh disisipkan di antara struktur data atau tipe data tersebut. Jika *character enconding* tidak didefinisikan, maka akan digunakan *default character encoding* untuk JSON, yaitu UTF-8. *File* JSON mempunyai *extension* **.json** . Proses mengubah suatu objek menjadi JSON *string* sering disebut juga *serialization*, sedangkan proses mengubah JSON *string* menjadi suatu objek sering disebut juga *deserialization*. Contoh JSON *string* :

```
{
   "book": [

      {
         "id":"01",
         "language": "Java",
         "edition": "third",
         "author": "Herbert Schildt"
      },

      {
         "id":"07",
         "language": "C++",
         "edition": "second",
         "author": "E.Balagurusamy"
      }
   ]
}
```

G. Parsing

Parsing merupakan proses membagi data dengan sekumpulan aturan menjadi data yang lebih kecil agar lebih mudah untuk disimpan serta dimanipulasi. *Parser* adalah program yang melakukan *parsing*. Tujuan dari *parsing* biasanya untuk melakukan validasi terhadap *input*, mengekstrak data dari *input*, lalu menyimpannya dalam bentuk yang sesuai. Salah satu aplikasi *parsing* yang paling luas adalah pengecekan *syntax* oleh *compiler*.

Contoh dari *parsing* adalah mengambil menit dan jam dari *time* dengan *format* HH : MM . Jika kita tidak melakukan *parsing*, akan sulit untuk memanipulasi data tersebut, karena berbentuk *string*. Maka, dilakukan *parsing* dengan mengambil nilai jam dari HH dan nilai menit dari MM, lalu keduanya disimpan sebagai *integer*.

III. PEMBAHASAN

Untuk melakukan *parsing* terhadap JSON dengan menggunakan *regex*, kita perlu mendeskripsikan dengan rinci seluruh tipe data pada JSON dan menentukan *regex*-nya masing-masing. Jika kita sudah mendapatkan *regex* dari masing-masing tipe data, kita hanya perlu menyusun sebuah kode program yang menggunakan pengulangan dan percabangan untuk mendapatkan semua data dari JSON. Mari kita teliti satu persatu tipe data pada JSON.

Pertama, kita mulai dari *string*. *String* cukup *simple*, karena *string* mempunyai arti secara literal atau harafiah. *String* diapit petik dua pada awal dan akhirnya. Yang tidak memiliki arti literal dalam *string* adalah `\"` dan `\\`. Karena itu, setelah kita mendapatkan *string* yang dimaksud, kita harus me-*replace* semua `\"` menjadi `"` dan semua `\\` menjadi `\`. Maka, *regex* untuk *match* tipe *string* adalah :

```
^"(.*)"$
```

Semua yang ada di dalam kurung akan di-*capture*, kemudian hasilnya dapat dikembalikan melalui *backreference*. Sebelum disimpan, akan dilakukan *replacement* terhadap `\"` dan `\\`.

Kedua, kita akan melihat tipe data *number*. Tipe data *number* bisa berupa bilangan negatif maupun positif. Maka, tanda *minus* (-) bersifat *optional*. Pada *number*, *leading zeroes* tidak diperbolehkan, maka, jika angka yang berada di paling depan adalah 0, tidak boleh ada angka dibelakangnya lagi. Sedangkan, jika angka yang berada di paling depan bukan 0, boleh diikuti 0 atau lebih angka.

Number tidak selalu berupa *integer*, bisa merupakan *floating point*. Pada kasus *floating point*, maka akan ada tanda titik yang diikuti 1 atau lebih angka. Karena tanda titik dan angka di belakang koma tidak bisa dipisahkan, maka kita akan melakukan *grouping*. Mereka juga bersifat *optional*, karena *number* bisa berupa *integer*.

Hal terakhir yang mungkin menjadi tambahan pada *number* adalah *scientific notation*. Pada *scientific notation*, angka dapat berupa *integer* maupun *floating point*, baik positif maupun negatif, dengan syarat hanya 1 digit angka di depan koma (tidak boleh angka 0), lalu pada bagian belakangnya ditambah dengan e atau E (eksponen), diikuti dengan tanda + atau -, lalu satu atau lebih angka yang menjadi eksponennya. Contohnya, kecepatan cahaya, 300.000.000 m/s dapat ditulis menjadi 3e+8, yang artinya 3 x 10^8. Perlu diperhatikan, bahwa *scientific notation* juga bersifat *optional*. Maka, dapat kita rumuskan *regex* untuk tipe data *number* :

```
^(-?(?:(?:(?:0|[1-9]\d*)(?:\.\d+)?)|
 (?:(?:0|[1-9])(?:\.\d+)?(?:(?:e|E)
        (?:\+|-)\d+)?)))$
```

Ketiga, tipe data *value*. *Value* sendiri dapat bernilai *true*, *false*, *null*, sebuah *string*, sebuah *number*, sebuah *object*, atau sebuah *object*. Untuk itu, kita hanya perlu menguji apakah nilainya *true*, *false*, atau *null*. Jika ya, nilainya bisa langsung dimasukkan, jika tidak, maka *string* tersebut akan kita uji terhadap *regex* lainnya untuk mengetahui tipe data sesungguhnya. *Regex* untuk tipe *value* :

```
^(true|false|null|.*)$
```

Jika untuk nilai *true*, *false*, *null*, ingin dilakukan *matching* secara *case insensitive*, maka kita dapat mengatur *flag ignore case* pada *regex engine*.

Keempat, tipe data *array*. *Array* pada JSON adalah kumpulan *value*. Masing-masing elemen *array* dipisahkan oleh tanda koma. *Regex* untuk mendapatkan isi *array* cukup sederhana :

```
^\s*\[\s*(.*)\s*\]\s*$
```

Isi *array* tersebut terdiri dari *n* buah *value* dan *n* - 1 buah tanda koma. Jika kita ingin mendapatkan nilai masing-masing *value*, maka kita dapat memisahkan elemen pertama terlebih dahulu, karena elemen pertama tidak diikuti tanda koma di depannya. *Regex* yang digunakan :

```
^([^,])+(.*)$
```

Elemen pertama sudah berhasil kita pisahkan. Sebelum kita melanjutkan ke elemen berikutnya, kita cek apakah elemen pertama adalah *empty string*. Jika ya, kita harus mengecek bahwa elemen sisanya juga *empty string*, yang menandakan memang *array* ini kosong, sehingga kita tidak perlu lanjut ke elemen berikutnya. Jika elemen pertama *empty string* tetapi elemen kedua tidak, maka terjadi kesalahan *syntax*, yaitu tidak adanya elemen pertama. Jika elemen pertama bukan *empty string*, maka *array* ini tidak kosong dan perlu dilakukan pengecekan terhadap elemen berikutnya.

Dengan *regex* di atas, kita mendapatkan elemen kedua sampai terakhir, tetapi semuanya masih menjadi satu *string*. Untuk memisahkan semuanya, kita dapat melakukan pengulangan dengan menggunakan *regex* berikut :

```
^,\s*([^,]+)(.*)$
```

Setiap kali kita selesai melakukan *match*, kita akan melakukan mengganti *string* yang akan di-*match* dengan *capturing group* terakhir, yang akan mengambil sisa *string* dari *match* kita sebelumnya. Saat *capturing group* terakhir sama dengan *empty string*, maka proses pemisahan *value* sudah selesai. Jika *capturing group* terakhir *match* dengan regex di bawah ini :

```
^,\s*(?:,.*)?$
```

maka *syntax* JSON pada *array* ini tidak *valid*, karena ada tanda koma yang tidak diikuti oleh *value*.

Pemisahan elemen dengan koma tidak ada masalah, selama *value* elemen itu sendiri tidak mengandung koma. Bagaimana jika *value* adalah *array* juga, sehingga mengandung koma? Kita tidak memerlukan cara baru untuk *parsing array* tersebut, tetapi cukup melakukan beberapa proses tambahan berikut ini.

Lakukan pengecekan masing-masing *value*, apakah sudah *valid* atau belum dengan *regex-regex* yang sudah ditulis di atas. Untuk *value* yang belum *valid*, cek apakah penyebab tidak *valid*-nya *value* tersebut. Jika *value* tersebut *match* dengan salah satu *regex* di bawah :

```
^\s*\[\s*(.+)$
^(.+)\s*\]\s*$
^\s*\{\s*(.+)$
^(.+)\s*\}\s*$
```

kemungkinan *value* tidak *valid* karena *value* tersebut merupakan *array* atau *object* dalam *array* kita. Oleh karena itu, pertama, kita harus mengecek apakah *capturing group* dari *regex* di atas merupakan sebuah *value* yang valid. Jika ya, kita hanya perlu mencari pasangan kurung yang tepat dan *capturing group*-nya *valid* untuk memastikan itu *array* maupun *object* di dalam *array* kita, dengan syarat semua *value* di dalamnya juga *valid*. Hal ini mungkin membutuhkan pengecekan ulang, karena isi dari sebuah *array* berbeda dari isi sebuah *object*.

Jika tidak, kemungkinan *value* tidak *valid* karena *value* mengandung koma, jadi coba gabungkan dengan *value* di sebelah kanannya hingga *value* gabungan tersebut *valid*. Jika tidak berhasil menemukan *value* gabungan yang *valid*, maka *syntax* pada JSON tersebut salah. Pastikan pengecekan *value* yang tidak *valid* dilakukan dari kiri ke kanan, sesuai urutan serialisasi JSON itu sendiri.

Kelima, tipe data *object*. Tipe data ini tidak jauh berbeda dari *array*, hanya *value* diganti *key-value pair*, dengan tipe data *key* adalah *string*, dan tipe data *value* adalah *value*. *Regex* untuk mendapatkan isi dari *object* pun tidak jauh berbeda dari *array*, hanya dengan mengganti kurung siku (*square brackets*) menjadi kurung kurawal (*curly brackets*) :

```
^\s*\{\s*(.*)\s*\}\s*$
```

Sama seperti *array*, kita akan memisahkan elemen pertama dari elemen sisanya. *Regex* yang digunakan juga sama, sehingga tidak perlu dituliskan lagi disini. Perhatikan bahwa kasus-kasusnya pun sama, sehingga kita dapat menentukan apakah *object* tersebut kosong, *syntax* nya tidak *valid*, atau harus mengecek elemen selanjutnya.

Perbedaannya, untuk *object*, setiap elemen dapat dicek *valid* atau tidaknya dengan cara mendapatkan terlebih dahulu *key* dan *value*-nya. Untuk mengantisipasi adanya tanda titik dua (:) pada *key* maupun *value*, kita harus melakukan hal yang sama seperti pada tanda koma, yaitu pisahkan yang pertama, kemudian lakukan pengecekan, seperti pengecekan di atas. Jika dibutuhkan, lakukan pengecekan kembali pada *substring* setelah tanda titik dua. Setelah selesai, lakukan pengecekan apakah *key* dan *value valid*. Jika *value* tidak *valid*, maka lakukan strategi yang sudah disebutkan untuk menemukan penyebab *value* tidak valid. Jika *value* pada elemen tersebut sudah *valid*, ulangi langkah-langkah tersebut untuk setiap elemen dari *object*. Apabila membutuhkan *regex*, untuk pengecekan setiap elemen *object* dapat digunakan *regex* yang sama seperti *regex-regex* pada *array*, sedangkan untuk pengecekan *key-value pair*, cukup mengganti tanda koma pada *regex-regex* pada *array* menjadi titik dua.

IV. KESIMPULAN

Regex merupakan salah satu cara untuk melakukan *string matching* secara fleksibel. Oleh karena itu, *regex* dapat dimanfaatkan untuk *parsing input*, tidak hanya sebatas *input* yang sederhana seperti nama, nomor telepon, atau *email*, tetapi juga *format* seperti JSON. Pada makalah ini, *format* JSON yang digunakan masih sederhana, sehingga mungkin banyak kasus yang tidak di-*handle* oleh *regex* dengan baik. Untuk *format* JSON yang lebih kompleks, *regex* dapat dikombinasikan dengan algoritma lainnya agar mencapai hasil yang lebih optimum.

V. ACKNOWLEDGEMENT

Pertama-tama, penulis mengucapkan syukur kepada Tuhan Yang Maha Esa, oleh karena penyertaan-Nya penulis dapat menyelesaikan makalah ini dengan baik. Penulis juga ingin mengucapkan terima kasih untuk Bapak Dr. Ir. Rinaldi Munir, M.T., Ibu Dr. Masayu Leylia Khodra S.T., M.T., dan Ibu Dr. Nur Ulfa Maulidevi, S.T., M.Sc. selaku dosen mata kuliah Strategi Algoritma yang telah memberikan ilmu dan menginspirasi penulis dalam penulisan makalah ini.

REFERENSI

[1] Munir, Rinaldi. 2009. Diktat Kuliah IF 2211 Strategi Algoritma. Bandung: Penerbit Teknik Informatika Institut Teknologi Bandung.

[2] Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2006). Introduction To Automata Theory Languages, and Computation. Boston : Pearson.

[3] https://www.tutorialspoint.com/ipv4/ipv4_packet_structure.htm diakses pada tanggal 15/5/2017 pukul 17:37

[4] https://json-csv.com/json diakses pada tanggal 15/5/2017 pukul 18:41

[5] https://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/automata-theory/basics.html diakses pada tanggal 15/5/2017 pukul 20:02

[6] http://stackoverflow.com/questions/6718202/what-is-a-regular-language diakses pada tanggal 17/5/2017 pukul 12:02

[7] http://www.cse.msu.edu/~torng/360Book/RegLang/ diakses pada tanggal 17/5/2017 pukul 12:33

[8] http://www.json.org/ diakses pada tanggal 17/5/2017 pukul 21:16

[9] https://www.tutorialspoint.com/json/json_overview.htm diakses pada tanggal 17/5/2017 pukul 21:59

[10] http://whatis.techtarget.com/definition/parse diakses pada tanggal 17/5/2017 pukul 22:07

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 18 Mei 2017

Reinhard Benjamin Linardi
13515011