Linguaggi di Programmazione
Progetto Prolog e Common Lisp 2018-01-15 (E1P)

JSON Parsing

Marco Antoniotti e Gabriella Pasi

Versione 24 ottobre 2016

Consegna:
15 gennaio 2018, ore 23:59 GMT+1

Introduzione

Lo sviluppo di applicazioni web su Internet, ma non solo, richiede di scambiare dati fra applicazioni eterogenee, ad esempio tra un client web scritto in Javascript e un server, e viceversa. Uno standard per lo scambio di dati molto diffuso è lo standard *JavaScript Object Notation*, o JSON. Lo scopo di questo progetto è di realizzare due librerie, una in Prolog e l'altra in Common Lisp, che costruiscano delle strutture dati che rappresentino degli oggetti JSON a partire dalla loro rappresentazione come stringhe.

La sintassi delle stringhe JSON

Considereremo una versione ***semplificata*** della sintassi delle stringhe JSON:

```
JSON           ::= Object | Array
Object         ::= '{}' | '{' Members '}'
Members        ::= Pair | Pair ',' Members
Pair           ::= String ':' Value
Array          ::= '[]' | '[' Elements ']'
Elements       ::= Value | Value ',' Elements
Value          ::= JSON | Number | String
Number         ::= Digit+ | Digit+ '.' Digit+
Digit          ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
String         ::= '"' AnyCharSansDQ* '"' | ''' AnyCharSansSQ* '''
AnyCharSansDQ  ::= <qualunque carattere (ASCII) diverso da '"'>
AnyCharSansSQ  ::= <qualunque carattere (ASCII) diverso da '''>
```

Dalla grammatica data, un oggetto JSON può essere scomposto ricorsivamente nelle seguenti parti:

1. Object
2. Pair
3. Array
4. Value
5. String
6. Number

La specifica completa della notazione JSON è ottenibile presso l'URL http://json.org. Facciamo presente che non tutti gli esempi di stringhe JSON che potete trovare su Internet sono riconoscibili, data la specifica semplificata della quale è richiesta l'implementazione.

Esempi

L'oggetto vuoto:

```
{}
```

L'array vuoto:

```
[]
```

Un oggetto con due "items":

```
{
 "nome" : "Arthur",
 "cognome" : 'Dent'
}
```

Un oggetto complesso, contenente un sotto-oggetto, che a sua volta contiene un array di numeri (notare che, in generale, gli array non devono necessariamente avere tutti gli elementi dello stesso tipo)

```
{
 "modello" : "SuperBook 1234",
 "anno di produzione" : 2014,
 "processore" : {
                 "produttore" : "EsseTi",
                 "velocità di funzionamento (GHz)" : [1, 2, 4, 8]
                }
}
```

Un esempio tratto da Wikipedia (una possibile voce di menu)

```
{
 "type": "menu",
 "value": "File",
 "items": [
           {"value": "New", "action": "CreateNewDoc"},
           {"value": "Open", "action": "OpenDoc"},
           {"value": "Close", "action": "CloseDoc"}
          ]
}
```

Indicazioni e requisiti

Dovete costruire un parser per le stringhe JSON semplificate che abbiamo descritto. La stringa in input va analizzata ricorsivamente per comporre una struttura adeguata a memorizzarne le componenti. Si cerchi di costruire un parser guidato dalla struttura ricorsiva del testo in input. Ad esempio, un eventuale *array* (e la sua composizione interna in *elements*) va individuato dopo l'individuazione del *member* del quale fa parte, e il meccanismo di ricerca non deve ripartire dalla stringa iniziale ma bensì dal risultato della ricerca del *member* stesso.

In altre parole, approcci del tipo "*ora cerco la posizione del ':' e poi prendo la sottostringa…*", non sono il modo migliore di affrontare in problema. Anzi: quasi sicuramente porteranno ad un programma estremamente complicato, poco funzionante e quindi… insufficiente.

Valore `undefined`

Javascript e quindi JSON spesso ritornano un valore 'undefined' per varie operazioni. In tutti questi casi voi dovrete invece generare un errore chiamando la funzione **`error`** in Common Lisp o *fallendo* in Prolog.

Errori di sintassi

Se la sintassi che incontrate non è corretta dovete *fallire* in Prolog o segnalare un errore in Common Lisp chiamando la funzione **`error`**.

Realizzazione Prolog

La realizzazione in Prolog del parser richiede la definizione di due predicati: **`json_parse/2`** e **`json_get/3`**.

Il predicato **`json_parse/2`** è definibile come:

```
json_parse(JSONString, Object).
```

che risulta vero se `JSONString` (una stringa SWI Prolog o un atomo Prolog) può venire scorporata come stringa, numero, o nei termini composti:

```
Object = json_obj(Members)
```

```
Object = json_array(Elements)
```

e ricorsivamente:

```
Members = [] or
Members = [Pair | MoreMembers]

Pair = (Attribute, Value)

Attribute = <string SWI Prolog>

Number = <numero Prolog>

Value = <string SWI Prolog> | Number | Object

Elements = [] or
Elements = [Value | MoreElements]
```

Il predicato **`json_get/3`** è definibile come:

```
json_get(JSON_obj, Fields, Result).
```

che risulta vero quando `Result` è recuperabile seguendo la catena di campi presenti in `Fields` (una lista) a partire da `JSON_obj`. Un campo rappresentato da `N` (con `N` un numero maggiore o uguale a 0) corrisponde a un indice di un array JSON.

Come caso speciale dovete anche gestire il caso

```
json_get(JSON_obj, Field, Result).
```

Dove `Field` è una stringa SWI Prolog.

Esempi

```
?- json_parse('{"nome" : "Arthur", "cognome" : "Dent"}', O),
   json_get(O, ["nome"], R).
O = json_obj([("nome", "Arthur"), ("cognome", "Dent")])
R = "Arthur"

?- json_parse('{"nome": "Arthur", "cognome": "Dent"}', O),
   json_get(O, "nome", R).                    % Notare le differenza.
O = json_obj([("nome", "Arthur"), ("cognome", "Dent")])
R = "Arthur"

?- json_parse('{"nome"  : "Zaphod",
                "heads" : ["Head1", "Head2"]}',      % Attenzione al newline.
              Z),
   json_get(Z, ["heads", 1], R).
Z = json_obj([("name", "Zaphod"), (heads, json_array(["Head1", "Head2"]))])
R = "Head2"

?- json_parse('[]', X).
X = json_array([]).

?- json_parse('{}', X).
X = json_obj([]).

?- json_parse('[}', X).
false

?- json_parse('[1, 2, 3]', A), json_get(A, [3], E).
false
```

Notate che nel corso dell'elaborazione potrebbe essere necessario gestire le stringhe in termini di liste di "codici di caratteri", utilizzando i predicati di conversione `atom_chars`, `string_codes` e `atom_string`[1]. Tali liste non vengono però visualizzate in modo leggibile da parte di utenti umani, e.g., `"http"` è visualizzata come `[104, 116, 116, 112]`. Nella costruzione dei valori di tipo `String` è richiesta l'eventuale conversione da liste di questo genere a stringhe leggibili.

La costruzione di un predicato invertibile in grado di risolvere questo problema non è immediata, però, il vostro programma dovrebbe essere in grado di rispondere correttamente a query nelle quali i termini fossero parzialmente istanziati, come ad esempio:

```
?- json_parse('{"nome" : "Arthur", "cognome" : "Dent"}',
              json_obj([json_array(_) | _]).
No.

?- json_parse('{"nome" : "Arthur", "cognome" : "Dent"}',
              json_obj([(nome, N) | _]).
N = "Arthur"

?- json_parse('{"nome" : "Arthur", "cognome" : "Dent"}', JSObj),
   json_get(JSObj, [cognome], R),
R = "Dent"
```

Input/Output da e su file

La vostra libreria dovrà anche fornire due predicati per la lettura da file e la scrittura su file.

```
json_load(FileName, JSON).
json_write(JSON, FileName).
```

[1] Si assume che stiate usando SWIPL.

Il predicato `json_load/2` apre il file `FileName` e ha successo se riesce a costruire un oggetto JSON. Se `FileName` non esiste il predicato fallisce. Il suggerimento è di leggere l'intero file in una stringa e poi di richiamare `json_parse/2`.

Il predicato `json_write/2` scrive l'oggetto JSON sul file `FileName` in sintassi JSON. Se `FileName` non esiste, viene creato e se esiste viene sovrascritto. Naturalmente ci si aspetta che

```
?- json_write(json_obj([/* stuff */]), 'foo.json'),
   json_load('foo.json', JSON).
JSON = json_obj([/* stuff */])
```

Attenzione! Il contenuto del file `foo.json` scritto da `json_write/2` dovrà essere JSON standard. Ciò significa che gli attributi dovranno essere scritti come stringhe e non come atomi.

Realizzazione Common Lisp

La realizzazione Common Lisp deve fornire due funzioni. (1) una funzione **`json-parse`** che accetta in ingresso una stringa e produce una struttura simile a quella illustrata per la realizzazione Prolog. (2) una funzione **`json-get`** che accetta un oggetto JSON (rappresentato in Common Lisp, così come prodotto dalla funzione **`json_parse`**) e una serie di "campi", recupera l'oggetto corrispondente. Un campo rappresentato da `N` (con `N` un numero maggiore o uguale a 0) rappresenta un indice di un array JSON.

La sintassi degli oggetti JSON in Common Lisp è:

***Object* = '(' json-obj *members* ')'**

***Object* = '(' json-array *elements* ')'**

e ricorsivamente:

*members = pair**

pair = '(' attribute value ')'

attribute = <atomo Common Lisp arbitrario> | <stringa Common Lisp>

number = <numero Common Lisp>

value = string | number | Object

*elements = value**

Esempio

```
CL-prompt> (defparameter x (json-parse "{\"nome\" : \"Arthur\",
                                         \"cognome\" : \"Dent\"}"))
X
;; Attenzione al newline!

CL-prompt> x
(json-obj ("nome" "Arthur") ("cognome" "Dent"))

CL-prompt> (json-get x "cognome")
"Dent"

CL-prompt> (json-get (json-parse
                       "{\"name\" : \"Zaphod\",
                         \"heads\" : [[\"Head1\"], [\"Head2\"]]}")
                     "heads" 1 0)
"Head2"

CL-prompt> (json-parse "[1, 2, 3]")
(json-array 1 2 3)

CL-prompt> (json-parse "{}")
(json-obj)

CL-prompt> (json-parse "[]")
(json-array)
```

```
CL-prompt> (json-parse "{]")
ERROR: syntax error

CL-prompt> (json-get (json-parse "[1, 2, 3]") 3) ; Arrays are 0-based.
ERROR: …
```

Input/Output da e su file

La vostra libreria dovrà anche fornire due funzioni per la lettura da file e la scrittura su file.

(json-load *filename*) ⇒ *JSON*
(json-write *JSON filename*) ⇒ *filename*

La funzione `json-load` apre il file *`filename`* ritorna un oggetto JSON (o genera un errore). Se *`filename`* non la funzione genera un errore. Il suggerimento è di leggere l'intero file in una stringa e poi di richiamare `json-parse`.

La funzione `json-write` scrive l'oggetto *`JSON`* sul file *`filename`* in sintassi JSON. Se *`filename`* non esiste, viene creato e se esiste viene sovrascritto. Naturalmente ci si aspetta che

```
CL-PROMPT> (json-load (json-write '(json-obj #| stuff |#) "foo.json"))
(json-obj #| stuff |#)
```

Da consegnare

LEGGERE MOLTO ATTENTATMENTE LE ISTRUZIONI!!!

Dovrete consegnare un file `.zip` (i files `.7z`, `.rar` o `.tar` etc, **non sono accettabili**!!!) dal nome

`MATRICOLA_Cognome_Nome_LP_E1P_JSON_2017.zip`

`Nome` e `Cognome` devono avere solo la prima lettera maiuscola, `Matricola` deve avere lo zero iniziale se presente. Cognomi e nomi multipli vanno inframmezzati con il carattere '_'; ad esempio: `Pravettoni_Brambilla_Gian_Giac_Pier_Carluca`.

Questo file compresso **deve contenere una sola directory con lo stesso nome**. Al suo interno ci deve essere una sottodirectory chiamata '`Prolog`' e una sottodirectory chiamata '`Lisp`'. Al loro interno queste directory devono contenere i files caricabili e interpretabili, più tutte le istruzioni che riterrete necessarie. Il file Prolog si deve chiamare '`json-parsing.pl`', e il file Lisp si deve chiamare '`json-parsing.lisp`'. Le due sottodirectory devono contenere un file chiamato `README.txt`. *In altre parole questa è la struttura della directory (folder, cartella) una volta spacchettata.*

```
MATRICOLA_Cognome_Nome_LP_E1P_JSON_2017
      Prolog
            json-parsing.pl
            README.txt

      Lisp
            json-parsing.lisp
            README.txt
```

Potete aggiungere altri files, ma il loro caricamento dovrà essere effettuato automaticamente al momento del caricamento ("loading") dei files sopracitati.

Come sempre, valgono le direttive standard (reperibili sulla piattaforma Moodle) circa la formazione dei gruppi.

Ogni file deve contenere all'inizio un commento con il nome e matricola di ogni membro del gruppo. Ogni persona deve consegnare un elaborato, anche quando ha lavorato in gruppo.

Il termine ultimo della consegna sulla piattaforma Moodle è il 15 gennaio 2018, ore 23:59 GMT+1

Valutazione

In aggiunta a quanto detto nella sezione “Indicazioni e requisiti” seguono ulteriori informazioni sulla procedura di valutazione.

Abbiamo a disposizione una serie di esempi standard che saranno usati per una valutazione oggettiva dei programmi. Se i files sorgente non potranno essere letti/caricati nell’ambiente Prolog (nb.: SWI-Prolog, ma non necessariamente in ambiente Windows, Linux, Mac), o nelll’ambiente Common Lisp (Lispworks, ma non necessariamente in ambiente Windows, Linux, Mac), il progetto non sarà ritenuto sufficiente.

Il mancato rispetto dei nomi indicati per funzioni e predicati, o anche delle strutture proposte e della semantica esemplificata nel testo del progetto, oltre a comportare ritardi e possibili fraintendimenti nella correzione, può comportare una diminuzione nel voto ottenuto.