Linguaggi di Programmazione
Modulo di Laboratorio di Linguaggi di Programmazione
Progetto Lisp e Prolog Gennaio 2025 (E1P)

Parsing di stringhe URI

Marco Antoniotti, Fabio Sartori

Consegna

18 gennaio 2025, 23:55, CET

Introduzione

Navigare in Internet, ma non solo, richiede ad un programma ed ai suoi programmatori l'abilità di manipolare delle stringhe che rappresentano degli "*Universal Resource Identifiers*" (URI). Lo scopo di questo progetto è di realizzare due librerie che costruiscono delle strutture che rappresentino internamente delle URI a partire dalla loro rappresentazione come stringhe.

Una libreria va implementata in Prolog; potete scegliere se implementare l'altra in Common Lisp o Julia.

La sintassi delle stringhe URI

Per questo progetto considereremo una sintassi ***semplificata*** di stringhe URI:

```
URI     ::= scheme ':' authorithy ['/' [path] ['?' query] ['#' fragment]]
        |   scheme ':' ['/'] [path] ['?' query] ['#' fragment]
        |   scheme `:' scheme-syntax

scheme          ::= <identificatore>

authorithy      ::= '//' [ userinfo '@' ] host [':' port]

userinfo        ::= <identificatore>

host            ::= <identificatore-host> ['.' <identificatore-host>]*
                | indirizzo-IP

port            ::= <digit>+

indirizzo-IP    ::= <NNN.NNN.NNN.NNN - con N un digit>

path            ::= <identificatore> ['/' <identificatore>]*

query           ::= <carattere>+

fragment        ::= <carattere>+

<identificatore>      ::= <carattere>+

<identificatore-host> ::= <lettera> <alfanum>*
```

```
<alfanum>      ::= <lettera> | <digit>

<carattere>    ::= <alfanum> | ‘_’ | ‘=’ | ‘+’ | ‘-’

<lettera>      ::= <lettere A-Z e a-z>

<digit>        ::= ‘0’ |‘1’ |‘2’ |‘3’ |‘4’ |‘5’ |‘6’ |‘7’ |‘8’ |‘9’

scheme-syntax  ::= <sintassi speciale – si veda sotto>
```

Si noti che molti degli elementi sono opzionali; il “port” ha 80 di default o quello tradizionale per lo schema. I **`NNN`** per l’**`indirizzo-IP`** devono essere compresi tra 0 e 255 e possono essere anche di una o due sole cifre.

Ripetiamo: la sintassi è semplificata, anche per evitare problemi di “escape” dei caratteri. L’intera specifica è contenuta nel seguente RFC (*Request For Comment*, gli ‘standard’ di Internet): http://tools.ietf.org/html/rfc3986

A partire dalla grammatica data, un’URI può essere scomposta quindi nelle seguenti componenti

1. Scheme
2. Userinfo
3. Host
4. Port
5. Path
6. Query
7. Fragment

Sebbene non sia richiesta l’implementazione di un interprete per la specifica completa, il documento sopraindicato è molto utile in quanto contiene una serie di esempi per valutare fino a che punto un programma è in grado di interpretare correttamente una URI. Ripetiamo che non tutti gli esempi di URI nel documento sono riconoscibili data la specifica semplificata della quale è richiesta l’implementazione.

Indicazioni e requisiti

La realizzazione dei progetti si basa in parte sulle conoscenze che avete acquisito nella parte di Linguaggi e Computabilità. Costruire un parser per le URI semplificate che abbiamo descritto richiede la costruzione di un automa a stati finiti.

Attenzione, sia in Lisp, che in Julia, che in Prolog la costruzione di automi a stati finiti è facilitata dal linguaggio. Non è detto che dobbiate costruire gli insiemi degli stati, la funzione di transizione e così via. E.g., in Lisp e Julia potreste scrivere una serie di funzioni mutuamente ricorsive che riconoscano la grammatica delle URI semplificate. In ogni caso, è richiesta una sequenzialità nella analisi e scomposizione della stringa in input, che va analizzata carattere per carattere per comporre una struttura adeguata a memorizzarne le componenti: ad esempio, l’eventuale *authority* (e la sua composizione interna in *userinfo*, *host* e *port*) va determinata dopo l’individuazione dello *scheme*, ed il meccanismo di ricerca non deve ripartire dalla stringa iniziale ma bensì dal risultato della ricerca dello *scheme* stesso.

In altre parole, approcci del tipo “ora cerco la posizione del ‘:’ e poi prendo la sottostringa …”, non sono il modo migliore di affrontare il problema.

Schemi e sintassi speciali

Gli schemi **`http`**, **`hppts`** e **`ftp`** sono "standard". Altri schemi sono da considerarsi "speciali". DI seguito trovate le sintassi speciali che dovete prendere in considerazione.

Schema `mailto`

In questo caso solo gli slot 'Userinfo' e 'Host' della struttura risultante dovranno essere riempiti.

```
scheme-syntax ::=    userinfo ['@' host]
```

Schema `news`

Questo caso (in una sua interpretazione lievemente semplificata) differisce dal primo tipo di URI nella sintassi indicata per la mancanza di `'//'` e dall'impossibilità di indicare una serie di parti opzionali. In pratica solo la parte 'Host' della struttura va riempita.

```
scheme-syntax ::=    host
```

Schemi `tel` e `fax`

In questo caso si richiede di poter specificare un numero di telefono nella forma:

```
scheme-syntax ::=    userinfo
```

Per semplicità non è richiesto alcun controllo sulla consistenza dell'identificatore associato a **`userinfo`**, a parte il rispetto delle relative regole sintattiche.

Schema `zos`

Lo schema **`zos`** descrive i nomi di data-sets su mainframes IBM. In questo caso il campo ***path*** ha una struttura diversa, che dovete controllare. Gli altri campi (*userinfo*, *host*, *port*, *query*, *fragment*), sono da riconoscere normalmente.

```
path   ::=    <id44> ['(' <id8> ')']
id44   ::=    <lettera> (<alfanum> | '.')*
id8    ::=    <lettera> <alfanum>*
```

dove la lunghezza di **`id44`** è al massimo 44 e quella di **`id8`** è al massimo 8. Inoltre, **`id44`** e **`id8`** devono iniziare con un carattere alfabetico; **`id44`** non può terminare con un `'.'`.

Common Lisp

In Common Lisp dovrete implementare una funzione `URILIB-PARSE` che riceve in ingresso una stringa e che ritorna una "struttura" con almeno i 7 campi di cui sopra. La scelta su come rappresentare questa struttura è libera (le persone più audaci possono provare ad utilizzare `DEFSTRUCT` o addirittura `DEFCLASS`), mentre è richiesta la realizzazione di opportune funzioni per l'accesso alle varie componenti di questo genere di struttura. In particolare, è richiesta la realizzazione delle seguenti funzioni:

- `urilib-parse:` `string → urilib-structure`
- `urilib-scheme:` `urilib-structure → string`
- `urilib-userinfo:` `urilib-structure → (or null string)`
- `urilib-host:` `urilib-structure → (or null string)`
- `urilib-port:` `urilib-structure → (or null string)`
- `urilib-path:` `urilib-structure → (or null string)`
- `urilib-query:` `urilib-structure → (or null string)`
- `urilib-fragment:` `urilib-structure → (or null string)`

- `urilib-display:` `urilib-structure &optional stream → T`

Esempio

```
CL prompt> (defparameter disco (urilib-parse ”http://disco.unimib.it”))
DISCO

CL prompt> (urilib-scheme disco)
”http”

CL prompt> (urilib-host disco)
”disco.unimib.it”

CL prompt> (urilib-query disco)
NIL

CL prompt> (urilib-display disco)
Scheme:      HTTP
Userinfo:    NIL
Host:        ”disco.unimib.it”
Port:        80
Path:        NIL
Query:       NIL
Fragment:    NIL

;;; The above is an example.  Note the NIL where you do not
;;; have a value.

T
```

Note

Notate che nel corso dell'elaborazione potrebbe essere necessario gestire le stringhe in termini di liste di caratteri, utilizzando la funzione di conversione `coerce`. Tali liste non vengono però visualizzate in modo leggibile da parte utenti umani, e.g., `”http”` potrebbe essere visualizzata come `(#\h #\t #\t #\p)`. Nella costruzione della struttura `urilib-structure` è richiesta l'eventuale conversione da liste di questo genere a stringhe leggibili. In alternativa, i più sofisticati di voi potranno usare la macro `WITH-INPUT-FROM-STRING` e la funzione `UNREAD-CHAR`.

*Notate che vi è **vietato** usare alcune librerie che si trovano in rete: in particolare non potete usare CL-PPCRE, ma non solo questa. Ovviamente secondo la nota qui sopra, usare `POSITION`, `FIND` e `SUBSEQ` et similia in modo inappropriato risulterà in riduzioni del voto.*

Julia

In Julia dovrete implementare una funzione `urilib_parse` che riceve in ingresso una stringa e che ritorna una "struttura" con almeno i 7 campi di cui sopra. La scelta su come rappresentare questa struttura è libera (ma si presume adopererete una `struct`), mentre è richiesta la realizzazione di opportune funzioni per l'accesso alle varie componenti di questo genere di struttura. In particolare, è richiesta la realizzazione delle seguenti funzioni:

- `urilib_parse:` `String → URILib_structure`
- `urilib_scheme:` `URILib_structure → String`
- `urilib_userinfo:` `URILib_structure → Union{Nothing, String}`
- `urilib_host:` `URILib_structure → Union{Nothing, String}`

- `urilib_port:` `URILib_structure → Union{Nothing, String}`
- `urilib_path:` `URILib_structure → Union{Nothing, String}`
- `urilib_query:` `URILib_structure → Union{Nothing, String}`
- `urilib_fragment:` `URILib_structure → Union{Nothing, String}`
- `urilib_display:` `URILib_structure, stream → Boolean`

Esempio

```
julia> disco = urilib_parse("http://disco.unimib.it")
… your URI representation …

julia> urilib_scheme(disco)
"http"

julia> urilib_host(disco)
"disco.unimib.it"

julia> urilib_query(disco)
nothing

julia> urilib_display(disco)
Scheme:      "http"
Userinfo:    nothing
Host:        "disco.unimib.it"
Port:        80
Path:        nothing
Query:       nothing
Fragment:    nothing

### The above is an example.  Note the 'nothing' where you do not
### have a value.

true
```

Prolog

Diversamente da quanto richiesto per l'implementazione in Lisp, la realizzazione in Prolog richiede da definizione del predicato `urilib_parse/2`.

```
urilib_parse(URIString, URI).
```

che risulta vero se `URIString` può venire scorporata nel termine composto

```
URI = uri(Scheme, Userinfo, Host, Port, Path, Query, Fragment).
```

Dovete anche implementare i predicati `urilib_display/1` e `urilib_display/2` che stampano una URI in formato testuale.

Esempio

```
?- urilib_parse("http://disco.unimib.it", URI).
URI = uri(http, [], 'disco.unimib.it', 80, [], [], [])
```

Notate che nel corso dell'elaborazione potrebbe essere necessario gestire le stringhe in termini di liste di caratteri, utilizzando i predicati di conversione `string_codes/2` e

`atom_string/2`[1]. Tali liste non vengono però visualizzate in modo leggibile da parte utenti umani, e.g., `”http”` potrebbe essere visualizzata come `[104, 116, 116, 112]`. Nella costruzione del termine composto `uri` è richiesta l'eventuale conversione da liste di questo genere a stringhe leggibili, nella fattispecie ad atomi Prolog.

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:

```
?- urilib_parse(”http://disco.unimib.it”,
                uri(https, _, _, _, _, _, _)).
No
?- urilib_parse(”http://disco.unimib.it”,
                uri(_ ,_ , Host, _, _, _, _)).
Host = ’disco.unimib.it’
```

ATTENZIONE!

Non fate copia-incolla di codice da questo documento, o da altre fonti. Spesso vengono inseriti dei caratteri UNICODE nel file di testo che creano dei problemi agli scripts di valutazione.

ATTENZIONE (2)!

PRIMA DI CONSEGNARE, CONTROLLATE ACCURATAMENTE CHE TUTTO SIA NEL FORMATO E CON LA STRUTTURA DI CARTELLE RICHIESTI.

RIPETIAMO! RILEGGETE BENE TUTTO IL TESTO (e le istruzioni di consegna).

Da consegnare

Dovrete consegnare un file `.zip` (i files `.rar` o `.rar` non sono accettabili!) dal nome

`Cognome_Multiplo_Nome_Secondo_Nome_Matricola_LP_E1P_2025.zip`

`Nome` e `Cognome` devono avere solo la prima lettera maiuscola, `Matricola` deve avere lo zero iniziale se presente.

Questo file *deve contenere una sola directory* con lo stesso nome. Al suo interno ci devono essere due sottodirectory chiamate ‘`Lisp`’ e ‘`Prolog`’. Al loro interno queste directories devono contenere i files caricabili ed interpretabili, più tutte le istruzioni che riterrete

[1] Si assume che stiate usando SWIPL.

necessarie. Il file Lisp si deve chiamare '`urilib-parse.lisp`'; il file Prolog si deve chiamare '`urilib-parse.pl`'; il file `Julia` si deve chiamare '`urilib_parse.jl`'Tutte le directories devono contenere un file chiamato `README.txt`. In altre parole, questa è la struttura della directory (folder, cartella) una volta spacchettata (se decidete di implementare in Julia, avrete la cartella `Julia` al posto di quella `Lisp` con il file `urilib-parse.jl`.

```
Cognome_Multiplo_Nome_Secondo_Nome_Matricola_LP_E1P_2025
      Lisp
            urilib-parse.lisp
            README.txt
      Prolog
            urilib-parse.pl
            README.txt
```

Potete aggiungere anche altri file, 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 componente del gruppo. Ogni persona deve consegnare un elaborato, anche quando ha lavorato in gruppo.

Il termine ultimo della consegna sulla piattaforma Moodle è sabato 18 gennaio 2025, ore 23:59 Zulu Time.

Valutazione

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

Disponiamo di una serie di esempi standard che verranno usati per una valutazione oggettiva dei programmi. Se i files sorgenti non potranno essere letti/caricati negli ambienti Lisp, Julia e Prolog (nb.: non necessariamente Lispworks e SWI-Prolog, e non necessariamente in ambiente Windows), 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 un decremento nel voto ottenuto.