Linguaggi di Programmazione
Modulo di Laboratorio di Linguaggi di Programmazione
Progetto Lisp e Prolog Gennaio/Febbraio 2022 (E1P, E2P)

Parsing di stringhe URI

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.

La sintassi delle stringhe URI

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

```
URI     ::= URI1 | URI2

URI1    ::= scheme ':' [authorithy] ['/' [path] ['?' query] ['#' fragment]]

URI2    ::= 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         ::= <caratteri senza '#'>+

fragment      ::= <caratteri>+

<identificatore>     ::= <caratteri senza '/', '?', '#', '@', e ':'>+

<identificatore-host>  ::= <caratteri senza '.', '/', '?', '#', '@', e ':'>+

<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, e deve essere res come un numero. I **`NNN`** per l'**`indirizzo-IP`** devono essere compresi tra 0 e 255.

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

Si noti che la grammatica ammette URI contenenti il solo scheme (e il port con default 80).

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 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 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.

Sintassi speciali

Al fine di semplificare l'implementazione e la sintassi di alcuni casi, definiamo qui alcune sintassi speciali da prendere in considerazione. La sintassi è specificata per ogni schema desiderato.

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 la sintassi speciale è una variazione della produzione **`URI1`**, con il campo ***path*** avente una struttura diversa, che dovete controllare. Gli altri campi (*userinfo*, *host*, *port*, *query*, *fragment*), sono da riconoscere normalmente come nella produzione **`URI1`**.

```
path   ::=    <id44> ['(' <id8> ')']
id44   ::=    (<caratteri alfanumerici> | '.')+
id8    ::=    (<caratteri alfanumerici>)+
```

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 `URI-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:

- `uri-parse:` `string → uri-structure`
- `uri-scheme:` `uri-structure → string`
- `uri-userinfo:` `uri-structure → string`
- `uri-host:` `uri-structure → string`
- `uri-port:` `uri-structure → integer`
- `uri-path:` `uri-structure → string`
- `uri-query:` `uri-structure → string`
- `uri-fragment:` `uri-structure → string`
- `uri-display:` `uri-structure &optional stream → T`

Esempi

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

CL prompt> (uri-scheme disco)
"http"

CL prompt> (uri-host disco)
"disco.unimib.it"

CL prompt> (uri-query disco)
NIL

CL prompt> (uri-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 `uri-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 `SUBEQ` et similia (ovvero i predicati equivalenti Prolog) in modo inappropriato risulterà in riduzioni del voto.

Prolog

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

```
uri_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 `uri_display/1` e `uri_display/2` che stampano una URI in formato testuale.

Esempio

```
?- uri_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. Un altro predicato che vi può essere utile è `number_string/2`.

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:

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