Linguaggi di Programmazione 2019-2020
Progetto Lisp e Prolog gennaio 2020 E2P
Consegna 22 febbraio 2020

Compilazione d'espressioni regolari in automi non deterministici

Marco Antoniotti, Pietro Braione, Gabriella Pasi, Rafael Peñaloza e Giuseppe Vizzari

Introduzione

Le espressioni regolari – *regular expressions*, o, abbreviando *regexps* – sono tra gli strumenti più utilizzati (ed abusati) in Informatica. Un'espressione regolare rappresenta in maniera finita un linguaggio (regolare), ossia un insieme potenzialmente infinito di sequenze di "simboli"[1], o *stringhe*, dove i "simboli" sono tratti da un alfabeto che indicheremo con Σ.

Le regexp più semplici sono costituite da tutti i "simboli" Σ, da *sequenze* di "simboli" e/o regexps, *alternative* tra "simboli" e/o regexps, e la *ripetizione* di "simboli" e/o regexps (quest'ultima è anche detta "chiusura" di Kleene). Se `<re>`, `<re`$_1$`>`, `<re`$_2$`>` … sono regexp, in Perl (e prima di Perl in '`ed`' UNIX) allora `<re`$_1$`><re`$_2$`>`, `<re`$_1$`>|<re`$_2$`>` e `<re>*` sono anche regexps le espressioni:

- `<re`$_1$`><re`$_2$`>…<re`$_k$`>` (sequenza)
- `<re`$_1$`>|<re`$_2$`>` (alternativa, almeno una delle due)
- `<re>*` (chiusura di Kleene, ripetizione 0 o più volte)

Ad esempio, l'espressione regolare `x`, dove `x` è un "simbolo", rappresenta l'insieme {`x`} contenente il "simbolo" `x`, o meglio: la *sequenza* di "simboli" di lunghezza 1 composta dal solo "simbolo" `x`; l'espressione regolare `pq`, dove sia `p` che `q` sono "simboli", rappresenta l'insieme {`pq`}contenente solo la sequenza di simboli, di lunghezza 2, `pq` (*prima* `p`, dopo `q`); l'espressione regolare `a*`, dove `a` è un "simbolo", rappresenta l'insieme infinito contenente tutte le sequenze ottenute ripetendo il simbolo `a` un numero arbitrario di volte{ε, `a`, `aa`, `aaa`, …}, dove ε viene usato per rappresentare la "sequenza di simboli con lunghezza zero"; l'espressione regolare `a(bc)*d`, dove `a`, `b`, `c`, `d` sono "simboli", rappresenta l'insieme {`ad`, `abcd`, `abcbcd`, `abcbcbcd`…} di tutte le sequenze che iniziano con `a`, terminano

[1] Metteremo la parola "simbolo" tra virgolette per indicare che non intendiamo parlare dei simboli nei linguaggi Prolog e Lisp: i "simboli" che formano l'alfabeto Σ, in teoria, potrebbero essere qualsiasi tipo di oggetti.

con `d`, e contengono tra questi due simboli un numero arbitrario di ripetizioni della sottosequenza `bc`.

Un'altra regexp utile è:

- `<re>+` (ripetizione, 1 o più volte)

Notate che queste regexps possono essere definite utilizzando opportune combinazioni degli operatori di sequenza, alternativa e chiusura di Kleene.

Com'è noto, a ogni regexp corrisponde un automa a stati finiti (non-deterministico o NFA) in grado di determinare se una sequenza di "simboli" appartiene o no all'insieme definito dall'espressione regolare, in un tempo asintoticamente lineare rispetto alla lunghezza della stringa.

Indicazioni e requisiti

Scopo del progetto è implementare in Prolog e in LISP un compilatore da *regexps*, espresse in un opportuno formato che verrà dettagliato in seguito, a NFA, più altre operazioni che verranno anch'esse dettagliate in seguito.

Prolog

Rappresentare le espressioni regolari più semplici in Prolog è molto facile: senza disturbare il parser intrinseco del sistema, possiamo rappresentare le regexps così:

- $\langle re_1\rangle\langle re_2\rangle\ldots\langle re_k\rangle$ diventa $[\langle re_1\rangle, \langle re_2\rangle, \ldots, \langle re_k\rangle]$
- $\langle re_1\rangle|\langle re_2\rangle|\ldots|\langle re_k\rangle$ diventa $/(\langle re_1\rangle, \langle re_2\rangle, \ldots, \langle re_k\rangle)$
- `<re>*` diventa `*(<re>)`
- `<re>+` diventa `+(<re>)`

L'alfabeto dei "simboli" Σ è costituito da termini Prolog (più precisamente, da tutto ciò che soddisfa **`compound/1`** o **`atomic/1`**).

Il predicato principale da implementare è **`nfa_regexp_comp/2`**. Il secondo predicato da realizzare è **`nfa_rec/2`**. Infine (o meglio all'inizio) va realizzato il predicato **`is_regexp/1`**.

1. **`is_regexp(RE)`** è vero quando `RE` è un'espressione regolare. Numeri e atomi (in genere anche ciò che soddisfa **`atomic/1)`**, sono le espressioni regolari più semplici; i termini che soddisfano **`compound/1`** non devono avere come funtore uno dei funtori "riservati" di cui sopra[2].
2. **`nfa_regexp_comp(FA_Id, RE)`** è vero quando `RE` è compilabile in un automa, che viene inserito nella base dati del Prolog. `FA_Id` diventa un identificatore per l'automa (deve essere un termine Prolog senza variabili).
3. **`nfa_rec(FA_Id, Input)`** è vero quando l'input per l'automa identificato da `FA_Id` viene consumato completamente e l'automa si trova in uno stato finale. `Input` è una lista Prolog di simboli dell'alfabeto Σ sopra definito.

[2] Qual è il funtore di `[a, b, c]`?

4. **`nfa_clear`**, **`nfa_clear(FA_id)`** sono veri quando dalla base di dati Prolog sono rimossi tutti gli automi definiti (caso **`nfa_clear/0`**) o l'automa **`FA_id`** (caso **`nfa_clear/1`**).

Esempi

Negli esempi seguenti, si considera solo la prima risposta del sistema; a seconda dell'implementazione, in sistema potrebbe generare più soluzioni.

```
?- nfa_regexp_comp(foo, baz(42)).
false       % Gestire gli errori...


?- is_regexp(a).
true        % NOTA BENE! Un simbolo è anche un'espressione regolare!


?- is_regexp(ab).
true.       % NOTA BENE! 'ab' è un atomo Prolog!


?- is_regexp([a, b]).
true


?- nfa_regexp_comp(basic_nfa_1, a).
true


?- nfa_rec(basic_nfa_1, a).
false       % Perché?


?- nfa_rec(basic_nfa_1, [a]).
true


?- nfa_regexp_comp(basic_nfa_2, ab).
true


?- nfa_rec(basic_nfa_2, [ab]).
true


?- nfa_regexp_comp(basic_nfa_3, [a, b]).
true


?- nfa_rec(basic_nfa_3, [ab]).
false       % Perché?


?- nfa_rec(basic_nfa_3, [a, b]).
true        % Perché?


?- nfa_regexp_comp(42, *(/(a, s, d, q))). % Complicato.
true
```

```
?- nfa_rec(42, [s, a, s, s, d]).
true


?- nfa_regexp_comp(automa_seq, [a, s, d]). % Semplice.
true


?- nfa_rec(automa_seq, [asd]).
false       % Perché?


?- nfa_rec(automa_seq, [a, s, d]).
true


?- nfa_rec(automa_seq, [a, s, w]).
false


?- nfa_rec(automa_seq, [a, w, d]).
false


?- nfa_regexp_comp(12, [qwe, rty, uio]).
            %% Cos'è un "simbolo" nell'alfabeto?
true


?- nfa_rec(12, [qwe, rty, uio]).
true


?- nfa_rec(12, [qwer, tyui, o]).
false       % Perché?

?- nfa_rec(12, [qwe, foo, uio]).
false


?- nfa_rec(12, [qwe, rty, a]).
false
```

Suggerimenti

A lezione sono anche stati mostrati degli esempi su come rappresentare gli NFA in una base dati Prolog e su come scrivere un predicato che "riconosca" una sequenza di simboli come appartenente al linguaggio riconosciuto (o generato) da un automa. Potete rappresentare internamente l'automa come preferite.

I predicati **`nfa_delta/4`**, **`nfa_initial/2`** e **`nfa_final/2`** sono definiti con un **`FA_Id`** come primo argomento.

Il predicato **`nfa_regexp_comp`** e i suoi predicati ancillari usano – ça va sans dire – il predicato **`assert/1`** o sue varianti.

Potrebbe essere utile poter generare identificatori univoci per gli stati dei vari automi. A tal proposito, si suggerisce l'utilizzo del predicato **`gensym/2`** che permette di costruire nuovi atomi caratterizzata da una prima parte costante seguita da un numero auto-incrementante secondo il seguente esempio:

```
?- gensym(foo, X).
X = foo1
?- gensym(foo, Y).
Y = foo2
```

Lisp

La rappresentazione in Lisp è del tutto analoga a quella in Prolog; rappresentate le regexps con delle liste così formate:

- `<re1><re2>…<rek>` diventa `([] <re1> <re2> … <rek>)`
- `<re1>|<re2>|…|<rek>` diventa `(/ <re1> <re2> … <rek>)`
- `<re>*` diventa `(* <re>)`
- `<re>+` diventa `(+ <re>)`

L'alfabeto dei "simboli" Σ è costituito S-exps Lisp. Quindi dovete pensare a quale predicato d'uguaglianza dovete usare per riconoscere al meglio gli elementi dell'input.

Dovete implementare le seguenti funzioni Lisp:

1. **`(is-regexp RE)`** ritorna vero quando `RE` è un'espressione regolare; falso (`NIL`) in caso contrario. Notate che un'espressione regolare può essere una Sexp, nel qual caso il suo primo elemento deve essere diverso da `s`, `/`, `*`, oppure `+`.
2. **`(nfa-regexp-comp RE)`** ritorna l'automa ottenuto dalla compilazione di `RE`, se è un'espressione regolare, altrimenti ritorna `NIL`. Attenzione, la funzione non deve generare errori. Se non può compilare la regexp `RE`, la funzione semplicemente ritorna `NIL`.
3. **`(nfa-rec FA Input)`** ritorna vero quando l'input per l'automa `FA` (ritornato da una precedente chiamata a `nfa-regexp-comp`) viene consumato completamente e l'automa si trova in uno stato finale. `Input` è una lista Lisp di simboli dell'alfabeto Σ sopra definito. Se `FA` non ha la corretta struttura di un automa come ritornato da `nfa-regexp-comp`, la funzione dovrà segnalare un errore. Altrimenti la funzione ritorna `T` se riesce a riconoscere l'`Input` o `NIL` se non ce la fa.

Attenzione a come sono utilizzate le funzioni **`nfa-rec`** e **`nfa-regexp-comp`**. Un tipico uso può essere il seguente:

```
cl-prompt> (nfa-rec (nfa-regexp-comp <some-re>) <some-input>)
T ; Or NIL, or a call to error.
```

Esempi

Negli esempi seguenti, si considera solo la prima risposta del sistema; a seconda dell'implementazione, in sistema potrebbe generare più soluzioni.

```
CL prompt> (defparameter foo (nfa-regexp-comp '(baz 42))
FOO


CL prompt> foo
NIL


CL prompt> (is-regexp 'a)
T           ; NOTA BENE! Un simbolo e' anche un'espressione regolare!


CL prompt> (is-regexp 'ab)
T           ; NOTA BENE! 'ab' e' un simbolo Lisp!


CL prompt> (is-regexp '([] a b))
T


CL prompt> (defparameter basic-nfa-1 (nfa-regexp-comp 'a))
BASIC-NFA-1


CL prompt> basic-nfa-1
#<This is an unreadable NFA 424242>


CL prompt> (nfa-rec basic-nfa-1 'a)
NIL         ; Perché?


CL prompt> (nfa-rec basic-nfa-1 '(a))
T


CL prompt> (nfa-rec 42 '(1 2 3 4 (5) 5 5))
Error: 42 is not a Finite State Automata.
…


CL prompt> (defparameter basic-nfa-2
             (nfa-regexp-comp '([] 0 (+ 1) 0)))
BASIC-NFA-2


CL prompt> (nfa-rec basic-nfa-2 '(0 1 0))
T


CL prompt> (nfa-rec basic-nfa-2 '(0 1 1 1 1 1 0))
T
```

```
CL prompt> (nfa-rec basic-nfa-2 '(0 0))
NIL


CL prompt> (defparameter basic-nfa-3
             (nfa-regexp-comp '([] a b)))
T


CL prompt> (nfa-rec basic-nfa-3 '(ab))
NIL          % Perché?


CL prompt> (nfa-rec basic-nfa-3 '(a b))
T


CL prompt> (defparameter nfa42
             (nfa-regexp-comp '(* (/ a s d q)))) ; Complicato.
NFA42


CL prompt> (nfa-rec NFA42 '(s a s s d))
T


CL prompt> (defparameter automa-seq
             (nfa-regexp-comp '([] a s d)) ; Semplice.
AUTOMA-SEQ


CL prompt> (nfa-rec automa-seq '(asd))
NIL          ; Perché?


CL prompt> (nfa-rec automa-seq '(a s d))
T


CL prompt> (nfa-rec automa-seq '(a s w))
NIL


CL prompt> (nfa-rec automa-seq '(a w d))
NIL


CL prompt> (defparameter nfa123
             (nfa-regexp-comp '([] qwe (rty uio) (* asd)))
             ;; Cos'è un "simbolo" nell'alfabeto?
NFA123


CL prompt> (nfa-rec nfa123 '(qwe (rty uio)])
T


CL prompt> (nfa-rec nfa123 '(qwe (rty uio) asd asd asd])
T
```

```
CL prompt> (nfa-rec nfa123 '(qwe foo uio))
NIL
```

Da consegnare

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

```
Cognome_Nome_Matricola_LP_202002.zip
```

`Nome` e `Cognome` devono avere solo la prima lettera maiuscola, `Matricola` deve avere lo zero iniziale se presente. Cognomi e nomi multipli dovranno essere scritti sempre con il carattere "underscore" ('_'). Ad esempio, "Gian Giacomo Pier Carl Luca De Mascetti Vien Dal Mare" che ha matricola 424242 diventerà:

```
De_Mascetti_Vien_Dal_Mare_Gian_Giacomo_Pier_Carl_Luca_424242_LP_202002
```

Questo file deve contenere *una sola directory* con lo stesso nome del file stesso. Al suo interno si devono trovare due sottodirectory chiamate rispettivamente 'Lisp' e 'Prolog'. Al loro interno ciascuna sottodirectory deve contenere i rispettivi files, caricabili e interpretabili in automatico, più tutte le istruzioni che ritenete necessarie. Il file Prolog deve chiamarsi '`nfa.pl`' ed il file Lisp deve chiamarsi '`nfa.lisp`'. Entrambe le directory devono contenere un file di testo chiamato `README.txt`. In altre parole, questa è la struttura della directory (folder, cartella) una volta spacchettata.

```
Cognome_Nome_Matricola_LP_202002
    Lisp
        nfa.lisp
        README.txt
    Prolog
        nfa.pl
        README.txt
```

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

Le prime righe dei files '`nfa.lisp`' e '`nfa.pl`' dovranno contenere i nomi e le matricole delle persone che hanno svolto il progetto in gruppo.

Il termine ultimo della consegna sulla piattaforma Moodle è venerdì 22 febbraio 2020, ore 23:59 GMT+1.

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 e Prolog (assumiamo che stiate usando Lispworks e SWI-Prolog, ma non necessariamente in ambiente Linux), 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.

Riferimenti

[HMU06] J. E. Hopcroft, R. Motwani, J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd Edition, Addison Wesley, 2006

[Sip05] M. Sipser, *Introduction to the Theory of Computation*, 2nd Edition, Course Technology, 2005