Linguaggi di Programmazione 2025-2026

Progetto Prolog e, Lisp o Julia gennaio 2026 E1P
Consegna 25 gennaio 2026

Compilazione d'espressioni regolari in automi non deterministici

Marco Antoniotti, Claudio Ferretti e Fabio Sartori

Introduzione

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

- $\texttt{<re}_1\texttt{><re}_2\texttt{>…<re}_k\texttt{>}$ (sequenza)
- $\texttt{<re}_1\texttt{>|<re}_2\texttt{>}$ (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 con `d`, e contengono tra questi due simboli un numero arbitrario di ripetizioni della sottosequenza `bc`.

Un'altra regex utile è:

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

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

Nota bene: nulla vieta di avere Σ `= {ciccio, 42, paperino, zio_di(Achille)}`.

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

Com'è noto, a ogni regex corrisponde un automa a stati finiti (non-deterministico o NFSA) 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 **Common Lisp** o **Julia**, un compilatore da *regexs* a NFSA. Le regexs sono espresse in un opportuno formato che verrà dettagliato in seguito. Infine, dovrete implementare anche 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 regexs così:

- `<re1><re2>…<rek>` diventa `c(<re1>, <re2>, …, <rek>)`
- `<re1>|<re2>|…|<rek>` diventa `a(<re1>, <re2>, …, <rek>)`
- `<re>*` diventa `z(<re>)`
- `<re>+` diventa `o(<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 **`nfsa_compile_regex /2`**. Il secondo predicato da realizzare è **`nfsa_recognize/2`**. Infine (o meglio, all'inizio) va realizzato il predicato **`is_regex/1`**, utile per debugging.

1. **`is_regex(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. **`nfsa_compile_regex(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 – nota bene: un qualunque termine Prolog, anche composto).
3. **`nfsa_recognize(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.
4. **`nfsa_delete_all`**, **`nfsa_delete(FA_id)`** sono veri quando dalla base di dati Prolog sono rimossi tutti gli automi definiti (caso **`nfsa_delete_all/0`**) o l'automa **`FA_id`** (caso **`nfsa_delete/1`**).

Esempi

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

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

[2] Qual è il funtore di `[a, b, c]`?

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

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

?- is_regex(c(a, b)).
true

?- nfsa_compile_regex(basic_nfsa_1, a).
true

?- nfsa_recognize(basic_nfsa_1, a).
false       % Perché?

?- nfsa_recognize(basic_nfsa_1, [a]).
true


?- nfsa_compile_regex(basic_nfsa_2, ab).
true


?- nfsa_recognize(basic_nfsa_2, [ab]).
true


?- nfsa_compile_regex(basic_nfsa_3, c(a, b)).
true

?- nfsa_recognize(basic_nfsa_3, [ab]).
false       % Perché?

?- nfsa_recognize(basic_nfsa_3, [a, b]).
true        % Perché?

?- nfsa_compile_regex(42, z(a(a, s, d, q))). % Complicato.
true

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

?- nfsa_compile_regex(automa_seq, c(a, s, d)). % Semplice.
true

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

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

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

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

?- nfsa_compile_regex(12, c(qwe, rty, a(uio, foo(bar)))).
            %% Cos'è un "simbolo" nell'alfabeto?
true

?- nfsa_recognize(12, [qwe, rty, foo(bar)]).
true

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

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

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

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

Suggerimenti

A lezione sono anche stati mostrati degli esempi su come rappresentare gli NFSA 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 **`nfsa_delta/4`**, **`nfsa_init/2`** e **`nfsa_final/2`** sono definiti con un **`FA_Id`** come primo argomento.

Il predicato **`nfsa_compile_regex`** 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
```

Common Lisp

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

- $\langle re_1\rangle\langle re_2\rangle\ldots\langle re_k\rangle$ diventa `(c` $\langle re_1\rangle$ $\langle re_2\rangle$ … $\langle re_k\rangle$`)`
- $\langle re_1\rangle|\langle re_2\rangle|\ldots|\langle re_k\rangle$ diventa `(a` $\langle re_1\rangle$ $\langle re_2\rangle$ … $\langle re_k\rangle$`)`
- `<re>*` diventa `(z <re>)`

- `<re>+` diventa `(o <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.

Nota bene: attenzione a distinguere the espressioni `(f …)` da quelle dove `f` è uno degli operatori `c`, `a`, `z`, oppure `o`.

Dovete implementare le seguenti funzioni Lisp:

1. **`(is-regex RE)`** ritorna vero quando `RE` è un'espressione regolare; falso (`NIL`) in caso contrario. Notate che un'espressione regolare puà essere una Sexp, nel qual caso il suo primo elemento deve essere diverso da `c`, `o`, `z`, oppure `o`.
2. **`(nfsa-compile-regex 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 regex `RE`, la funzione semplicemente ritorna `NIL`.
3. **`(nfsa-recognize FA Input)`** ritorna vero quando l'input per l'automa `FA` (ritornato da una precedente chiamata a `nfsa-compile-regex`) 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 `nfsa-compile-regex`, 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 **`nfsa-recognize`** e **`nfsa-compile-regex`**. Un tipico uso può essere il seguente:

```
cl-prompt> (nfsa-recognize (nfsa-compile-regex <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 (nfsa-compile-regex '(baz 42))
FOO

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

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

CL prompt> (is-regex '(c a b))
T

CL prompt> (defparameter basic-nfsa-1 (nfsa-compile-regex 'a))
BASIC-NFSA-1
```

```
CL prompt> basic-nfsa-1
#<This is an unreadable NFSA 424242>

CL prompt> (nfsa-recognize basic-nfsa-1 'a)
NIL          ; Perché?

CL prompt> (nfsa-recognize basic-nfsa-1 '(a))
T

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

CL prompt> (defparameter basic-nfsa-2
             (nfsa-compile-regex '(c 0 (o 1) 0)))
BASIC-NFSA-2

CL prompt> (nfsa-recognize basic-nfsa-2 '(0 1 0))
T

CL prompt> (nfsa-recognize basic-nfsa-2 '(0 1 1 1 1 1 0))
T

CL prompt> (nfsa-recognize basic-nfsa-2 '(0 0))
NIL

CL prompt> (defparameter basic-nfsa-3
             (nfsa-compile-regex '(c a b)))
T

CL prompt> (nfsa-recognize basic-nfsa-3 '(ab))
NIL          % Perché?

CL prompt> (nfsa-recognize basic-nfsa-3 '(a b))
T


CL prompt> (defparameter nfsa42
             (nfsa-compile-regex '(z (a a s d q)))) ; Complicato.
NFSA42

CL prompt> (nfsa-recognize NFSA42 '(s a s s d))
T

CL prompt> (defparameter automa-seq
             (nfsa-compile-regex '(s a s d)) ; Semplice.
AUTOMA-SEQ

CL prompt> (nfsa-recognize automa-seq '(asd))
NIL          ; Perché?

CL prompt> (nfsa-recognize automa-seq '(a s d))
T
```

```
CL prompt> (nfsa-recognize automa-seq '(a s w))
NIL

CL prompt> (nfsa-recognize automa-seq '(a w d))
NIL

CL prompt> (defparameter nfsa123
             (nfsa-compile-regex '(s qwe (rty uio) (z asd)))
             ;; Cos'è un "simbolo" nell'alfabeto?
NFSA123

CL prompt> (nfsa-recognize nfsa123 '(qwe (rty uio)])
T

CL prompt> (nfsa-recognize nfsa123 '(qwe (rty uio) asd asd asd])
T

CL prompt> (nfsa-recognize nfsa123 '(qwe foo uio))
NIL
```

Julia

In Julia dovete implementare la rappresentazione delle varie espressioni regolari utilizzando le caratteristiche di *meta-programmazione* (*meta-programming*) del linguaggio. Il link dove trovare informazioni è: https://docs.julialang.org/en/v1/manual/metaprogramming/. Le funzioni da implementare sono le stesse di Common Lisp: **`nfsa_recognize`** and **`nfsa_compile_regex`**, con un comportamento del tutto simile.

La rappresentazione in Julia è del tutto analoga a quella in Prolog e Common Lisp; ma dovete rappresentare le regexs con delle espressioni (vedasi il link sopra) così formate:

- `<re`$_1$`><re`$_2$`>…<re`$_k$`>` diventa: `:(c(<re`$_1$`>, <re`$_2$`>, … <re`$_k$`>))`
- `<re`$_1$`>|<re`$_2$`>|…|<re`$_k$`>` diventa: `:(a(<re`$_1$`>, <re`$_2$`>, … <re`$_k$`>))`
- `<re>*` diventa: `:(z(<re>))`
- `<re>+` diventa: `:(o(<re>))`

Notate il '`:`' prima della parentesi. Potete usare la funzione standard `Meta.parse` per trasformare stringhe in espressioni.

Note

1. La sintassi delle espressioni regolari sembra standardizzata, ma non lo è: ne esistono molte varianti. Le più diffuse sono quelle POSIX, `egrep` ("*Extended Regular Expression*") e, ovviamente, Emacs. La sintassi e semantica che vi sono richieste sono minimali.
2. Le versioni di SWI Prolog, Lispworks Personal Edition e Julia da usare sono le ultime disponibili sui rispettivi siti.
3. **AI**.
   Non possiamo chiedervi di non usare le AI a cui avete accesso. Ma ricordate che dovete capire profondamente il codice che consegnate. In altre parole, dovete padroneggiare l'ambiente ed i linguaggi per essere sicuri che ciò che vi chiediamo sia quello che l'AI du jour produce.

Da consegnare

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

```
Cognome_Nome_Matricola_NFSA_LP_202601.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_NFSA_LP_202601
```

Questo file deve contenere *una sola directory* con lo stesso nome del file stesso. Al suo interno si devono trovare **un file** chiamato "`Gruppo.txt`" **e due sottodirectory** chiamate rispettivamente '`Lisp`' (o '`Julia`') 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 '`nfsa.pl`' ed il file Lisp deve chiamarsi '`nfsa.lisp`'; quello Julia '`nfsa.jl`'. 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 (si assume con Lisp).

```
Cognome_Nome_Matricola_NFSA_LP_202601
      Gruppo.txt
      Lisp
            nfsa.lisp
            README.txt
      Prolog
            nfsa.pl
            README.txt
```

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

Il file "`Gruppo.txt`" deve contenere, in ordine alfabetico, il nome dei componenti del gruppo, uno per linea con il formato

```
Cognome<tab>Nome<tab>Matricola"
```

Fate molta attenzione ai caratteri di tabulazione.

Le prime righe dei files '`nfsa.lisp`', '`nfsa.jl`' e '`nfsa.pl`' dovranno contenere i nomi e le matricole delle persone che hanno svolto il progetto in gruppo; in ordine alfabetico e con il formato da usarsi per il file `Gruppo.txt`.

Il termine ultimo della consegna sulla piattaforma Moodle è sabato 24 gennaio 2026, ore 23:59 GMT+1.

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.

Inoltre, **NON** usate *packages* in Common Lisp e *moduli* in Julia e Prolog! Il vostro codice deve rimanere nell'ambiente (*namespace*) "utente".

LEGGETE BENE TUTTE LE ISTRUZIONI!!!

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, SWI-Prolog e Julia, 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