# Introduzione

Le espressioni regolari – *regular expressions*, o, abbreviando *regexps* – sono tra gli strumenti più utilizzati 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>`, $\texttt{<re}_1\texttt{>}$, $\texttt{<re}_2\texttt{>}$ … sono regexp, in Perl (e prima di Perl in '`ed`' UNIX) allora anche le espressioni seguenti sono regexps:

- $\texttt{<re}_1\texttt{><re}_2\texttt{>...<re}_k\texttt{>}$ sequenza
- $\texttt{<re}_1\texttt{>|<re}_2\texttt{>}$ alternativa, almeno una delle due
- `<re>*` chiusura di Kleene, ripetizione 0 o più volte
- `<re>+` chiusura di Kleene, ripetizione 1 o più volte
- `¬<re>` negazione
- $[\sigma_1, \sigma_2, \ldots, \sigma_m]$ uno dei simboli $\sigma_i$ (appartenenti a **Σ**) elencati

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`. Infine, l'espressione regolare `¬xyz`, dove `x`, `y`, `z` sono "simboli", rappresenta l'insieme infinito contenente tutte le sequenze di simboli tratti dall'alfabeto (compresa ε), ad esclusione della sola stringa `xyz`.

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.

---

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

## Indicazioni e requisiti

Scopo del progetto è implementare in Prolog un compilatore da regexps, espresse in un opportuno formato che sarà 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 del sistema, possiamo rappresentare le regexps così:

- `<re1><re2>…<rek>` diventa `seq(<re1>,<re2>,…, <rek>)`
- `<re>*` diventa `star(<re>)`
- `<re>+` diventa `plus(<re>)`
- `¬<re>` diventa `bar(<re>)`
- `<re1>|<re2>|…|<rek>` diventa `alt(<re1>,<re2>,…,<rek>)`
- `[σ1, σ2, …, σm]` diventa `oneof(σ1, σ2, …, σm)`

Non preoccupatevi di usare operatori diversi da quelli nella lista qui sopra. La semantica degli operatori qui sopra è quella standard:

- `seq` indica la sequenza di regexps.
- `star` indica la chiusura di Kleene con 0 o più ripetizioni; `plus` quella con almeno una ripetizione.
- `bar` indica la negazione della regexp.
- `alt` indica l'unione delle regexps.
- `oneof` indica la scelta tra i simboli in questione

L'alfabeto dei "simboli" **Σ** è costituito dai numeri e atomi di Prolog (più precisamente, da tutto ci che soddisfa **`atomic/1)`**.

Il predicato principale da implementare è **`nfa_compile_regexp/2`**. Il secondo predicato da realizzare è **`nfa_recognize/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 ci che soddisfa **`atomic/1)`**, sono le espressioni regolari più semplici.
2. **`nfa_compile_regexp(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_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.

## *Esempi*

```
?- nfa_compile_regexp(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! Questo non è simbolo 'a' seguito da 'b'.


?- is_regexp(seq(a, b, 42, qwe)).
true.


?- nfa_compile_regexp(basic_nfa_1, a).
true.


?- nfa_recognize(basic_nfa_1, a).
false.   % Perchè?


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


?- nfa_compile_regexp(basic_nfa_2, ab).
true.


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


?- nfa_compile_regexp(basic_nfa_3, seq(a, b, c)).
true.


?- nfa_recognize(basic_nfa_3, [abc]).
false.    % Perchè?
```

```
?- nfa_recognize(basic_nfa_3, [a, b, c]).
true.


?- nfa_compile_regexp(42, star(alt(a, s, d, q))). % Complicato.
true ;
false.


?- nfa_recognize(42, [s, a, s, s, d]).
true ;
false.


?- nfa_compile_regexp(12, seq(qwe, rty, uio)). % Cos'è un "simbolo"?
true ;
false.


?- nfa_recognize(12, [qwe, rty, uio]).
true ;
false.


?- nfa_recognize(12, [qwer, tyui, o]).
false.   % Perchè?

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


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


?- nfa_compile_regexp(nonfa, plus(bar(alt(qwe, rty, uio)))).
true ;
false.


?- nfa_recognize(nonfa, [a, s, a]).
true.


?- nfa_recognize(nonfa, [a, 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.

Per la negazione `bar(<re>)` dovete semplicemente verificare che la stringa non sia riconosciuta dall'automa per `<re>`.

Per seguire l'esempio presentato a lezione, dovrete definire i predicati "delta", "initial", "final" etc. Sarà bene definire i predicati: **`nfa_delta/4`**,

**`nfa_initial/2`** e **`nfa_final/2`**; questi predicati sono definiti con un **`FA_Id`** come primo argomento.

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

Si suggerisce anche di definire dei predicati **`nfa_clear/0`**, **`nfa_clear_nfa/1`**, **`nfa_list/0`** e **`nfa_list/1`** che "puliscano" la base dati e che "listino" la struttura di un automa.

Data la specifica, dovrebbe essere chiaro che i predicati **`functor/3`**, **`arg/3`** e **`=..`** sono necessari per la stesura del codice.

Potrebbe essere utile avere la possibilità di 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
```