Linguaggi dinamici

Corso di Laurea in Informatica

A.A. 2019/2020

Brevi note sui linguaggi formali

Nozioni introduttive

- Alfabeti stringhe e linguaggi

Linguaggi regolari, espressioni ed automi

- Linguaggi regolari
- Espressioni regolari
- Espressioni regolari nelle applicazioni
- Semplici esercizi sulle e.r.

Analizzatori lessicali

- Il ruolo di un a.l. nel front-end
- Strumenti di sviluppo: il generatore di scanner Lex

Linguaggi dinamici

Qualche definizione preliminare

- Un *alfabeto* è un insieme finito di *simboli* (*caratteri*)
- Esempi di alfabeti importanti in Informatica:
  - I set di caratteri ASCII e UNICODE;
  - $\mathcal{B} = \{\texttt{0}, \texttt{1}\}$, l'alfabeto binario;
  - $\mathcal{D} = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$, l'alfabeto del DNA.
- Una *stringa* su un dato alfabeto è una sequenza di caratteri giustapposti
- Qualora ci possano essere ambiguità, per denotare un stringa useremo `questo font` oppure racchiuderemo la stringa fra apici
- Nella teoria, una stringa formata da un solo carattere è un oggetto diverso dal carattere stesso.
- Java segue questa impstazione, a differenza di Python
- Una stringa speciale è quella formata da zero caratteri, detta stringa vuota e indicata con ϵ

Operazioni sulle stringhe

- Concatenazione di due stringhe: $X = \texttt{Los}$, $Y = \texttt{Angeles} \Rightarrow Z = XY = \texttt{LosAngeles}$
  - In Python $Z = X + Y$
  - La concatenazione è un'operazione *associativa*
- Potenza di una stringa: $X = \texttt{Los} \Rightarrow Z = X^k = \underbrace{XX\ldots X}_{k} = \underbrace{LosLos\ldots Los}_{k}$
  - In Python: $Z = X * k$
  - Notare: $k = 0 \Rightarrow X^k = \epsilon$ (anche in Python)
- Riflessione di una stringa: $X = \texttt{Roma} \Rightarrow Z = X^R = \texttt{amoR}$
  - In Python. $Z = X[::-1]$
- Lunghezza di una stringa: $X = \texttt{Modena} \Rightarrow |X| = 6$
  - In Python: $\texttt{len}(X)$

Linguaggi formali

- Indicheremo genericamente con Σ l'alfabeto di riferimento
- Un *linguaggio* L su un alfabeto Σ è semplicemente un insieme di stringhe su Σ.
- Se le stringhe che compongono il linguaggio sono in numero finito, una possibilità per descrivere L consiste nella elencazione di tali stringhe.
- Esempio:

$$L_1 = \{00, 01, 10, 11\}$$

  è un linguaggio definito su $\mathcal{B}$ composto da 4 stringhe.
- Molto più interessanti sono i linguaggi composti da un numero infinito di stringhe, che comunque dobbiamo e vogliamo descrivere in modo finito.

Operazioni con i linguaggi

- Poiché i linguaggi sono insiemi, su di essi sono definite tutte le operazioni insiemistiche: unione, intersezione, differenza, ecc.
- Due linguaggi L_1 ed L_2 si possono poi *concatenare*, dando origine ad un nuovo linguaggio L così definito:

$$L = L_1 L_2 = \{X : \exists Y \in L_1, \exists Z \in L_2 \text{ t.c. } X = YZ\}$$

  In altre parole, L è costituito da tutte le stringhe che si possono ottenere concatenando una stringa di L_1 e una stringa di L_2.
- Più in generale, la seguente ricorrenza permette di definire, per ogni $m \geq 0$, la *potenza n-esima* di un linguaggio L:

$$\begin{aligned} L^0 &= \{\epsilon\} \\ L^n &= L^{n-1} L,\ n > 0. \end{aligned}$$

Chiusura riflessiva e transitiva

- La *chiusura riflessiva e transitiva* di un linguaggio L, indicata con L^*, è l'unione di tutte le potenze n-esime di L, per ogni valore di n positivo. In formule:

$$L^* = \bigcup_{n \geq 0} L^n$$

- Esempio: sia $L = \mathcal{B} = \{0, 1\}$. Abbiamo

$$\begin{aligned}
\mathcal{B}^0 &= \{\epsilon\} \\
\mathcal{B}^1 &= \{0, 1\} \\
\mathcal{B}^2 &= \{00, 01, 10, 11\} \\
\mathcal{B}^3 &= \{000, 001, 010, 011, 100, 101, 110, 111\} \\
&\ldots \\
\mathcal{B}^n &= \{\underbrace{0\ldots00}_{n}, \underbrace{0\ldots01}_{n}, \ldots, \underbrace{1\ldots11}_{n}\} \\
&\ldots
\end{aligned}$$

Chiusura riflessiva e transitiva (cont.)

- Ne consegue che $\mathcal{B}^*$, in quanto unione di tutte le potenze $\mathcal{B}^n$, è l'insieme di tutte le stringhe (di qualsiasi lunghezza) composte da 0 e 1.
- Questa costruzione vale per qualunque alfabeto.
- In generale, cioè, Σ^* è un modo per indicare l'insieme di tutte le possibili stringhe sull'alfabeto Σ
- Molto usata è anche la notazione L^+, detta *chiusura riflessiva* di L e così definita

$$L^* = \bigcup_{n \geq 1} L^n$$

- La differenza fra L^* ed L^+ consiste nel fatto che in L^+ non è necessariamente presente la stringa vuota. Essa appartiene a L^+ se e solo se appartiene ad L

Descrizione di un linguaggio infinito

- Possiamo definire (in modo non formale) il linguaggio usando l'Italiano o un'altra lingua "naturale"
- Ad esempio: "il linguaggio costituito da tutte le stringhe su $\mathcal{B}$ che terminano con il carattere 0".
- Possiamo altrimenti definire, con linguaggio matematico, la struttura delle stringhe, se è possibile individuarne una...:
- Nel caso del linguaggio definito sopra, che chiameremo L_2, possiamo scrivere:

$$L_2 = \{X \in \mathcal{B}^* | X = Y0, Y \in \mathcal{B}^*\}$$

- Altri esempi di linguaggi definiti in questo modo sono:
  - $L_3 = \{X \in \mathcal{B}^* : |X| \geq 3\}$;
  - $L_4 = \{X \in \mathcal{B}^* : \exists k \geq 0 \text{ t.c. } X = 01^k0\}$;
  - $L_5 = \{X \in \mathcal{B}^* : X = X^R\}$.

Descrizione di un linguaggio infinito (cont.)

- In ambito informatico ci aono due altri modi importanti per caratterizzare (e descrivere) un linguaggio
  - Caratterizzazione *algoritmica* (o *riconoscitiva*);
  - Caratterizzazione *generativa*
- Un algoritmo decisionale A ci permette di caratterizzare un linguaggio in quanto la sua "risposta" **True**/**False** ad ogni determinato input x può essere interpretata come indicativa dell'appartenenza o meno di x al linguaggio
- Più precisamente, se Σ è un alfabeto e A un algoritmo decisionale, possiamo definire il linguaggio associato a (o riconosciuto da) A nel modo seguente

$$\mathcal{L}_A = \{x \in \Sigma^* | A(x) = \textbf{True}\}$$

dove $A(x)$ indica l'output di A su input x

Descrizione di un linguaggio infinito (cont.)

- Vediamo un esempio concreto
- Un programma C++ legale è una stringa (tipicamente abbastanza lunga) sull'alfabeto ASCII
- Chiaramente il viceversa, in generale, non è vero: non tutte le stringhe ASCII sono programmi C++ legali
- Ora, su input una qualsiasi stringa ASCII, il compilatore C++ esegue una fra due possibili (macro)azioni: (1) produce il corrispondente codice macchina, (2) emette un messaggio di errore.
- Se facciamo corrispondere le due azioni rispettivamente ai valori **True** e **False**, possiamo a ragione affermare che il C++ è il linguaggio costituito dalle stringhe ASCII per cui il compilatore risponde **True** (cioè produce il codice macchina)

Descrizione di un linguaggio infinito (cont.)

- C'è naturalmente il problema dell'alfabeto di riferimento.
- Ad esempio, che cosa succede se l'input al compilatore non è una stringa ASCII?
- In tal caso il codice macchina non viene ovviamente prodotto. Possiamo quindi stabilire che il compilatore (e, in generale, l'algoritmo riconoscitore) risponde **False** oppure che emette un diverso messaggio di errore che segnali l'input mal-formato.
- La slide seguente mostra elementari funzioni Python per il riconoscimento dei linguaggi che abbiamo chiamato L_3, L_4 ed L_5 sull'alfabeto $\mathcal{B}$

Semplici riconoscitori in Python

```
def isbinary(x):
    '''
    Verifica che x sia una stringa binaria
    '''
    return all(c in '01' for c in x)

def L3(x):
    return  isbinary(x) and len(x)>=3

def L4(x):
    if isbinary(x):
        return len(x)>=2 and x[0]=='0' and \
               x[-1]=='0' and \
               x[1:-1].find('0')==-1

def L5(x):
   return isbinary(x) and x == x[::-1]
```

Descrizione di un linguaggio infinito (cont.)

- La seconda tecnica importante in ambito informatico per descrivere un linguaggio è quella *generativa*.
- Con questa tecnica si danno “regole” mediante le quali è possibile generare tutte e sole le stringhe del linguaggio che si vuole specificare.
- I più importanti formalismi generativi in ambito informatico sono le *espressioni regolari* e le *grammatiche context-free*
- Le corrispondenti famiglie di linguaggi portano lo stesso nome: linguaggi *regolari* e linguaggi *context-free*

Linguaggi dinamici

Perché sono importanti

- Sono “pervasivi” in ambito informatico.
- Tipici pattern di ricerca all’interno di documenti definiscono linguaggi regolari.
- Rivestono poi un ruolo cruciale all’interno dei linguaggi di programmazione.
- Infatti, anche se un tipico linguaggio di programmazione non è un linguaggio regolare, sono tuttavia regolari i seguenti “sotto-linguaggi”:
  - l’insieme degli identificatori di funzione e di variabile;
  - l’insieme di tutte le costanti numeriche (integer o float).
- Sono inoltre regolari tutti i linguaggi *finiti*, cioè costituiti da un numero finito di stringhe.

Definizione di linguaggio regolare

- Dato un alfabeto Σ cominciamo col definire *unitario su* Σ ogni linguaggio costituito da un singolo carattere di Σ
- Ad esempio, se $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, i linguaggi unitari su Σ sono: $\{\mathtt{a}\}$, $\{\mathtt{b}\}$ e $\{\mathtt{c}\}$
- Un linguaggio L su un alfabeto $\Sigma = \{\mathtt{a}_1, \ldots, \mathtt{a}_n\}$ si dice *regolare* se può essere espresso usando un numero finito di operazioni di concatenazione, unione e chiusura riflessiva a partire dai suoi linguaggi unitari $\{\mathtt{a}_1\}, \ldots, \{\mathtt{a}_n\}$
- Più precisamente:
  - $\{\mathtt{a}_1\}, \ldots, \{\mathtt{a}_n\}$ sono linguaggi regolari
  - se R_1 ed R_2 sono linguaggi regolari, allora $R_1 \cup R_2$ e $R_1 R_2$ sono linguaggi regolari
  - se R è un linguaggio regolare allora R^* è un linguaggio regolare

Esempi di linguaggi regolari

- Sia Σ l'alfabeto ASCII e sia $X = X_1 X_2 \ldots X_n$ una generica stringa di Σ^*. Il linguaggio $\{X\}$ è regolare in quanto esprimibile come concatenazione dei linguaggi unitari $\{X_1\}, \{X_2\}, \ldots, \{X_n\}$
- Ad esempio $\{\texttt{C++}\}$ è concatenazione dei linguaggi unitari $\{\texttt{C}\}$, $\{+\}$ e $\{+\}$, mentre $\{\texttt{Python}\}$ è concatenazione dei linguaggi unitari $\{\texttt{P}\}$, $\{\texttt{y}\}$, $\{\texttt{t}\}$, $\{\texttt{h}\}$, $\{\texttt{o}\}$ e $\{\texttt{n}\}$
- Il linguaggio $\{X, Y, Z\}$, dove X, Y e Z sono stringhe generiche sull'alfabeto ASCII è regolare perché esprimibile come unione dei linguaggi regolari $\{X\}, \{Y\}$, e $\{Z\}$

Esempi di linguaggi regolari (continua)

- Il linguaggio $\{\texttt{C++}, \texttt{Python}\}$ è regolare perché unione di due linguaggi che sappiamo essere regolari
- Generalizzando gli esempi precedenti si dimostra facilmente come ogni linguaggio finito sia esprimibile come unione di concatenazioni di linguaggi unitari

Altri esempi

- $\{\mathtt{ab}, \mathtt{c}\}^2$ è (un linguaggio) regolare perché:

$$\{\mathtt{ab}, \mathtt{c}\}^2 = (\{\mathtt{a}\}\{\mathtt{b}\} \cup \{\mathtt{c}\})(\{\mathtt{a}\}\{\mathtt{b}\} \cup \{\mathtt{c}\})$$

- $L_6 = \{\mathtt{a}^n | n \geq 0\}$ è regolare perché $L_6 = \{\mathtt{a}\}^*$
- Anche il linguaggio $L_7 = \{\mathtt{a}^n\mathtt{b}^m | n, m \geq 0\}$ è regolare poiché $L_7 = \{\mathtt{a}\}^*\{\mathtt{b}\}^*$, cioè è la concatenazione di due linguaggi regolari
- Il linguaggio $\{\mathtt{a}\}^+ = \{\mathtt{a}^n | n \geq 1\}$ è regolare perché $\{\mathtt{a}\}^+ = \{\mathtt{a}\}\{\mathtt{a}\}^*$
- $\left(\{\mathtt{ab}, \mathtt{c}\}^2\right)^R$ è regolare poiché $\left(\{\mathtt{ab}, \mathtt{c}\}^2\right)^R = \{\mathtt{ba}, \mathtt{c}\}^2$
- In generale L^R è regolare se (e solo se) L è regolare.
- Il linguaggio $L_8 = \{\mathtt{a}^n\mathtt{b}^n | n \geq 0\}$ non è regolare
- Il linguaggio $L_9 = \{\mathtt{a}^n | n \text{ primo}\}$ non è regolare

Linguaggi dinamici

Espressioni regolari

- Le *espressioni regolari* su un alfabeto Σ sono un formalismo (cioè a loro volta sono linguaggi) per definire linguaggi regolari
- Definiremo dapprima le espressioni regolari (e.r.) nella forma matematicamente più pulita
- In seguito presenteremo “abbreviazioni” linguistiche comunemente riconosciute da molti strumenti/ambienti (da `MS Word`® a `grep`)
- Negli esercizi useremo quasi esclusivamente le espressioni nella forma base

Espressioni regolari di base

- Le e.r. su un alfabeto Σ riflettono le costruzioni usate nella definizione dei linguaggi regolari su Σ

  Base
  - $\mathbf{\Phi}$ è un'espressione regolare che denota il linguaggio vuoto
  - per ogni $\mathtt{a} \in \Sigma$, $\mathbf{a}$ è un'e.r. che denota il linguaggio unitario $\{\mathtt{a}\}$

  Ricorsione Se $\mathcal{E}$ ed $\mathcal{F}$ sono e.r. che denotano, rispettivamente, i linguaggi E ed F, allora la scrittura:
  - $\mathcal{E}\mathcal{F}$ è un'e.r. che denota il linguaggio EF (*concatenazione*)
  - $\mathcal{E} + \mathcal{F}$ (o $\mathcal{E}|\mathcal{F}$) è un'e.r. che denota il linguaggio $E \cup F$ (*unione*)
  - $\mathcal{E}^*$ è un'e.r. che denota il linguaggio E^* (*chiusura riflessiva*)

Espressioni regolari

Un'ulteriore regola

Parentesi. Se $\mathcal{E}$ è un'e.r., la scrittura $(\mathcal{E})$ è un'e.r. *equivalente* alla prima, cioè che denota lo stesso insieme di stringhe

serve a forzare un ordine di composizione delle espressioni diverso da quello standard (in base al quale chiusura precede concatenazione che precede unione)

Esempi

- L'espressione regolare $\mathbf{0} + \mathbf{1}^*\mathbf{10}$ su $\mathcal{B}$ (interpretabile come $\mathbf{0} + ((\mathbf{1}^*)\mathbf{10})$, in base alle regole di precedenza) denota il linguaggio $R_1 = \{0, 10, 110, 1110, \ldots\}$
- Il linguaggio R_1 è chiaramente differente dal linguaggio R_2 su $\mathcal{B}$ definito dall'espressione regolare $(\mathbf{0} + \mathbf{1})^*\mathbf{10}$, che consiste di tutte le stringhe binarie che terminano con 10
- Posto $\Sigma = \{a, b, c\}$, l'espressione regolare $\mathbf{a}(\mathbf{b} + \mathbf{c})^*\mathbf{a}$ denota il linguaggio R_3 su Σ costituito dalle stringhe che iniziano e terminano con il carattere a e che non contengono altri caratteri a
- La scrittura $(\mathbf{1} + \mathbf{01})^*(\mathbf{0} + \mathbf{1} + \mathbf{01})$ denota il linguaggio delle stringhe su $\mathcal{B}$ di lunghezza almeno 1 che non contengono due caratteri 0 consecutivi

Linguaggi dinamici

Abbreviazioni di uso pratico

- Il simbolo ϵ si usa per indicare l'insieme $\{\epsilon\}$ (Attenzione! Non è l'insieme vuoto)
- La scrittura $[\mathcal{E}]$ si può utilizzare al posto della e.r. $\mathcal{E}+\epsilon$ e l'operatore [ ] prende il nome di *opzione*
- Se è definito un ordinamento fra i caratteri di Σ, allora si possono utilizzare convenzioni specifiche per denotare intervalli di caratteri. Ad esempio, la scrittura $[\texttt{a} - \texttt{f}]$ denota i caratteri compresi fra `a` ed `f` (opzione su un intervallo)
- Le scritture `[xyz]` e `[^xyz]` indicano rispettivamente un qualunque carattere appartenente o non appartenente all'insieme `{x,y,z}`

Abbreviazioni di uso pratico

- Poiché $L^+ = LL^*$, l'operatore di chiusura (non riflessiva) è ammesso nelle e.r. dove si intende che $\mathcal{E}^+ = \mathcal{E}\mathcal{E}^*$ (in tal caso l'operatore di unione o alternativa viene sostituito da $|$)
- Poiché $L^n = \overbrace{LL\ldots L}^{n}$, l'operatore di elevamento a potenza è ammesso nelle e.r. e si intende che $\mathcal{E}^n = \overbrace{\mathcal{E}\mathcal{E}\ldots\mathcal{E}}^{n}$
- La scrittura $[\mathcal{E}]_i^j$ si può utilizzare al posto della e.r. $\mathcal{E}^i + \mathcal{E}^{i+1} + \ldots + \mathcal{E}^j$

Definizione di numeri e di identificatori

- Per alcuni insiemi di caratteri di particolare importanza (cifre, lettere, caratteri alfanumerici, caratteri di spaziatura, ...) si possono usare espressioni specifiche, come ad esempio (prendendo a prestito la notazione dalle espressioni riconosciute dal comando `grep` di Linux):
  [: *digit* :], [: *alpha* :], [: *alnum* :], [: *space* :], ...
- L'espressione regolare $[1-9][:\textit{digit}:]^*$ denota l'insieme delle stringhe che rappresentano (nella consueta rappresentazione in base 10) i numeri interi positivi
- L'espressione regolare $[:\textit{alpha}:]([:\textit{alpha}:]|[:\textit{digit}:])^*$ denota l'insieme degli identificatori legali in alcuni linguaggi di programmazione (soprattutto fra i più vecchi)

Applicazioni diffuse che usano le espressioni regolari

- Innanzitutto gli editori di testo, più o meno sofisticati (`MS Word`®, `Libre Office Writer`, `Emacs`, ...)
- Molte applicazioni che manipolano file eseguibili da linea di comando in ambiente Unix/Linux (ad esempio, `grep`, `find` e `sed`)
- Utility per la costruzione di analizzatori lessicali (come `Lex`)
- In tutti questi casi, la sintassi per le espressioni regolari è molto più ampia, e (pur non aumentando il potere espressivo) rende la definizione dei pattern molto più semplice

Espressioni regolari in applicazioni di Linux

- . ha un match con qualsiasi singolo carattere, ad eccezione di ‘\n’
- * ha un match con 0 o più copie della precedente espressione
- + ha un match con 1 o più copie della precedente espressione
- ? ha un match con 0 o 1 copia della precedente espressione
- $\{n\}$ e $\{n, m\}$, dove n ed m sono numeri, hanno un match con la precedente espressione rispettivamente n volte (prima forma) oppure fra n ed m volte (seconda forma)
- [ ] ha un match con qualunque carattere incluso tra le parentesi; se il primo carattere è ^, allora c’è un match con qualunque carattere non incluso fra le parentesi

Espressioni regolari in applicazioni di Linux (2)

- Un simbolo - entro le parentesi quadre serve per indicare un intervallo di caratteri, come nel caso di $[0-9]$
- ^ come primo carattere di una e.r. ha un match con l'inizio di una linea
- $ come ultimo carattere di una e.r. ha un match con la fine di una linea
- \ è il classico carattere di escape
- | è il simbolo di alternativa
- ( ),o \(\), servono per il raggruppamento di e.r. e per il loro eventuale riferimento

Esempi di uso di grep

- `grep -e 404 algogroup.log`
  stampa tutte le richieste che contengono la sottostringa 404 (quindi, presumibilmente, cha hanno originato status code 404 (file not found)
- `grep -e "algogroup.unimo\(re\)\?.it" algogroup.log`
  stampa tutte le richieste il cui *referer* è una pagina del sito *algogroup.unimore.it* (viene accettato anche l'alias `algogroup.unimo.it`)
- `grep -e "http:[^;\)\"]*" -o algogroup.log | sort | uniq -c | sort -n -r`
  stampa i differenti referer, ordinati per numerosità decrescente

Linguaggi dinamici

Esercizi sulle e.r.

- Scrivere un'e.r. per il seguente linguaggio sull'alfabeto $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$:

$$E_1 = \{\mathtt{a}^n \mathtt{b}^m \mathtt{c}^k | m = 0 \Rightarrow k = 3\}$$

- Scrivere un'e.r. per il linguaggio E_2, sull'alfabeto $\{\mathtt{a}, \mathtt{b}\}$, delle stringhe contenenti al più due $\mathtt{a}$
- Scrivere un'e.r. per il linguaggio E_3, sull'alfabeto $\{\mathtt{a}, \mathtt{b}\}$, delle stringhe contenenti un numero dispari di $\mathtt{b}$
- Scrivere un'e.r. per il linguaggio E_4, sull'alfabeto $\{\mathtt{a}, \mathtt{b}\}$, definito ricorsivamente nel modo seguente:
  1. $\epsilon \in E_4$;
  2. Se $x \in E_4$ allora anche $\mathtt{aba}x \in E_4$ e $x\mathtt{aa} \in E_4$

  Inoltre, solo stringhe ottenibili in questo modo appartengono a E_4.

Esercizi sulle e.r.

- Scrivere un'e.r. per il linguaggio E_5, sull'alfabeto $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, costituito dalle stringhe in cui ogni occorrenza di $\mathtt{b}$ è seguita da almeno un'occorrenza di $\mathtt{c}$
- Descrivere nel modo più semplice possibile, in Italiano, il linguaggio corrispondente alla seguente espressione regolare: $((\mathbf{a}|\mathbf{b})^3)^*(\mathbf{a}|\mathbf{b})$
- Si dica qual è la stringa più corta che non appartiene al linguaggio descritto dall'espressione regolare $\mathbf{a}^*(\mathbf{ab})^*\mathbf{b}^*$

Esercizi sulle e.r.

- Si scriva un'espressione regolare per definire il linguaggio delle stringhe sull'alfabeto $\{0, 1\}$ che non contengono tre 1 di fila
- Si consideri l'espressione regolare

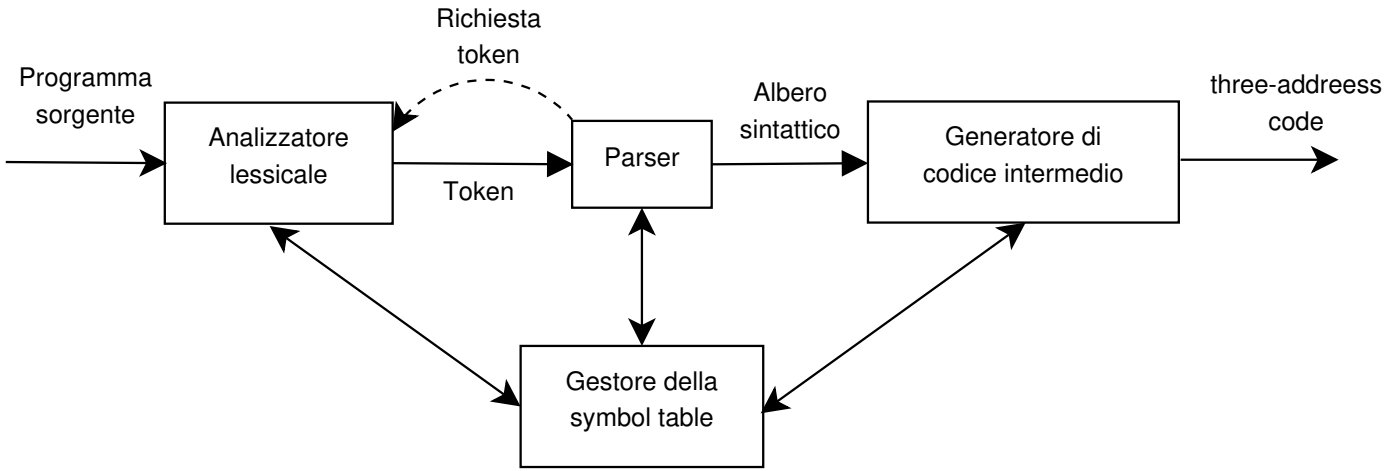
$$\mathtt{b}^*\mathtt{aa}\,(\mathtt{ba} \mid \mathtt{b})^*\,\mathtt{b}$$

  e si descriva "a parole" il linguaggio da essa rappresentato
- Si scriva un'espressione regolare per il linguaggio su $\{0, 1\}$ costituito dalle stringhe che iniziano con 00 oppure terminano con 01

Linguaggi dinamici

Il front-end più in dettaglio

- La seguente figura illustra un tipico schema di organizzazione del front-end di un compilatore (tratto da Aho, Lam, Sethi, Ullman (2007)).



Analizzatore lessicale

- L'*analizzatore lessicale*, detto anche *scanner*, è l'unico modulo che legge il file di testo che costituisce l'input per il compilatore.
- Il suo ruolo è di raggruppare i caratteri in input in *token*, ovvero "oggetti" significativi per la successiva analisi sintattica.
- Esempi di token sono i numeri, gli identificatori e le parole chiave di un linguaggio di programmazione.
- Ad esempio, la sequenza di caratteri:

```
X = 3.14;
```

potrebbe venire trasformata nella sequenza di token:

id assign number sep

Analizzatore lessicale (cont.)

- Ci sono però altri compiti che deve svolgere l'analizzatore lessicale:
  - riconoscere e "filtrare" commenti, spazi e altri caratteri di separazione;
  - associare gli eventuali errori trovati da altri moduli del compilatore (in particolare dal parser) alle posizioni (righe di codice) dove tali errori si sono verificati allo scopo di emettere appropriati messaggi diagnostici;
  - procedere all'eventuale espansione delle macro (se il compilatore le prevede).

Linguaggi dinamici

Che cosa è Lex

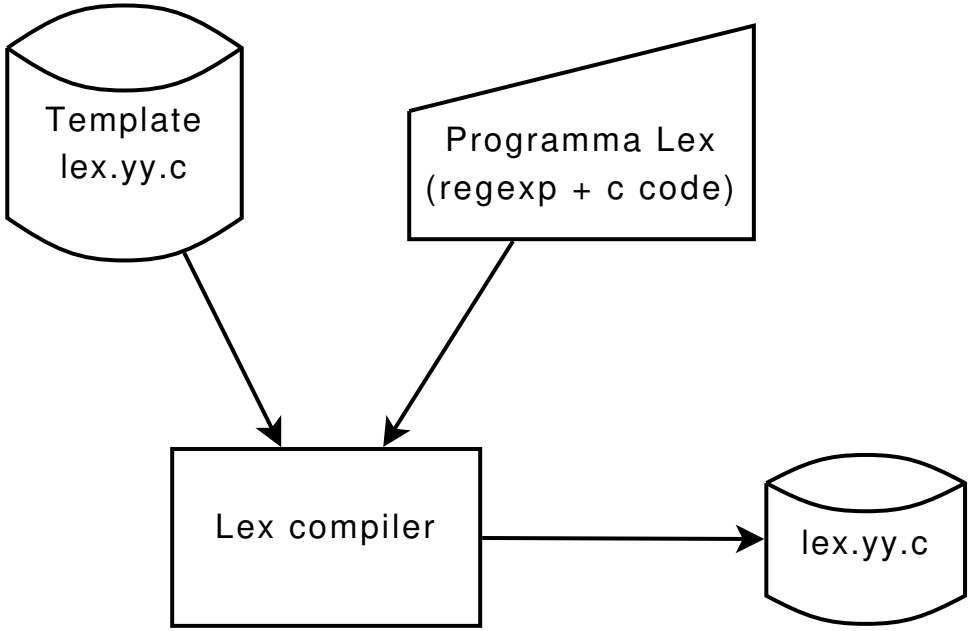
- *Lex* è un *generatore di scanner*.
- Si tratta cioè di un software in grado di generare automaticamente un altro programma che riconosce stringhe di un linguaggio regolare.
- Non solo, il software generato da Lex ha “capacità di scanning”, cioè di acquisire le stringhe da analizzare in sequenza (da file o standard input) e di passare l’output ad un altro programma, tipicamente un parser.
- Lex può quindi essere uno strumento prezioso nella realizzazione di un compilatore.

Come funziona Lex

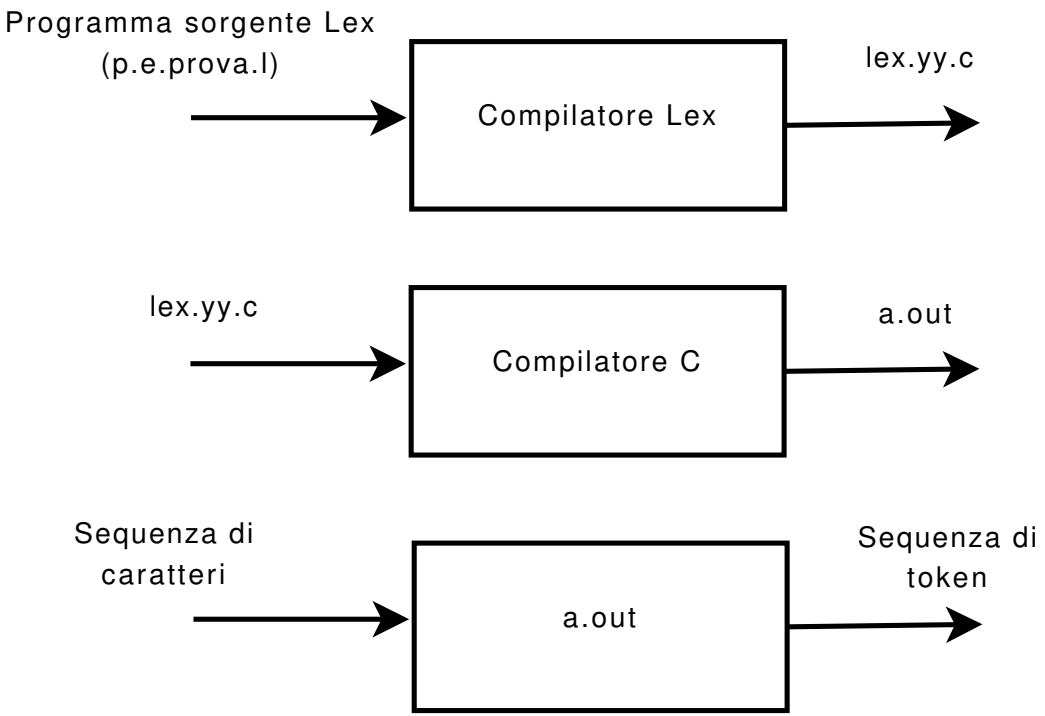
- L'input per un programma Lex è costituito "essenzialmente" da un insieme di espressioni regolari/pattern da riconoscere.
- Inoltre, ad ogni espressione regolare $\mathcal{E}$, l'utente associa un'*azione*, espressa sotto forma di codice C.
- Lex "trasforma" $\mathcal{E}$ nella descrizione di un "automa" che riconosce il linguaggio descritto da $\mathcal{E}$.
- Lex inoltre associa, ad ogni stato terminale dell'automa che riconosce $\mathcal{E}$, il corrispondente software fornito dall'utente.
- Nel programma generato da Lex, tale software andrà in esecuzione quando viene riconosciuto un lessema di $\mathcal{E}$.

Come funziona Lex (2)

- Lex inserisce la descrizione dell'automa e il codice fornito dall'utente in uno scheletro di programma (*template*) per ottenere così il programma finale "completo", per default chiamato `yy.lex.c`.
- La parte più "complessa" dal punto di vista teorico consiste proprio nella trasformazione delle espressioni regolari in automi.
- Ma noi sappiamo già "che cosa ci sta sotto"!



Schema d'uso di Lex



Un primo programma Lex

- Programma Lex per riconoscere (un sottoinsieme de) i token del linguaggio Pascal

```
delim [ \t\n]
ws {delim}+
letter [A-Za-z]
digit [0-9]
id {letter}({letter}|{digit})*
number [+-]?{digit}+(\.{digit}+)?(E[+-]?{digit}+)?
%%
{ws} { }
if {printf("IF ");}
then {printf("THEN ");}
else {printf("ELSE ");}
{id} {printf("ID ");}
{number} {printf("NUMBER ");}
"<" {printf("RELOP ");}
"<=" {printf("RELOP ");}
"=" {printf("RELOP ");}
"<>" {printf("RELOP ");}
">" {printf("RELOP ");}
">=" {printf("RELOP ");}
":=" {printf("ASSIGNMENT ");}
%%
main()
{ yylex();
  printf("\n"); }
```

Esecuzione del programma

- Se mandiamo in esecuzione il programma C ottenuto dopo le due compilazioni (si rammenti lo schema d'uso) con il seguente input:

```
if var1<0.0 then
    sign := -1
    else sign := 1
```

otteniamo come output la sequenza di token

IF ID RELOP NUMBER THEN ID ASSIGNMENT NUMBER ELSE ID ASSIGNMENT NUMBER

in realtà poiché i token name vengono internamente rappresentati mediante interi, la sequenza le output potrebbe essere:

12 2 6 1 13 2 18 1 14 2 18 1

Struttura generale di un programma Lex

- Un generico programma Lex contiene tre sezioni, separate dal dalla sequenza %%

```
Dichiarazioni
%%
Regole di traduzione
%%
Funzioni ausiliarie
```

- La sezione “Dichiarazioni” può contenere definizione di costanti e/o variabili, oltre alle cosiddette *definizioni regolari*, cioè espressioni che consentono di “dare un nome” ad espressioni regolari.
- La sezione più importante è quella relativa alle regole di traduzione che contiene le descrizioni dei pattern da riconoscere e, corrispondentemente, le azioni che devono essere eseguite dallo scanner.

Struttura generale di un programma Lex (continua)

- L'ultima sezione può contenere funzioni aggiuntive (che vengono tipicamente invocate nella parte relativa alle regole di traduzione).
- Se lo scanner non è utilizzato come routine del parser o di altro programma, quest'ultima sezione contiene anche il main program.
- Se presente, il main conterrà ovviamente la chiamata allo scanner (funzione `yylex`).

Non solo riconoscimento di token

- Il seguente programma Lex conta caratteri, parole e linee presenti in un file (assimiglia dunque al programma `wc` di Unix/Linux).

```
%{
unsigned charCount = 0, wordCount = 0, lineCount = 0;
%}

word [^ \t\n]+
eol \n

%%

{word}      {wordCount++; charCount += yyleng; }
{eol}       {charCount++; lineCount++}
.           charCount++;

%%
main()
{ yylex();
  printf("%d %d %d\n", lineCount, wordCount, charCount);
```