Espressioni regolari e cenni sugli analizzatori lessicali

M.Moro - Dip. di Elettronica e Informatica - Univ. di Padova - Dicembre 1999

Nella trattazione che segue si utilizzeranno una terminologia ed alcune operazioni specifiche del contesto dei linguaggi formali. Un succinto richiamo alle definizioni principali è presente nelle prime due appendici terminali del documento.

1. Espressioni regolari

I linguaggi finiti (cioè con un numero finito di stringhe componenti), quando di cardinalità elevata, vengono con difficoltà definiti per enumerazione. Si preferisce fornire una definizione di tipo "generativo", che risulta in ogni caso indispensabile per definire linguaggi non finiti. Un primo modo è utilizzare una descrizione verbale, evidenziando proprietà che le stringhe del linguaggio devono soddisfare (vedi ad esempio L1 nella sezione A2). Esiste una classe particolare di linguaggi che possono essere descritti in forma generativa dalle "espressioni regolari". La definizione di espressione regolare è data in forma costruttiva (algoritmica).

Dato un alfabeto finito T si consideri l'alfabeto dei metasimboli M dato da:
$M = \{\bullet, *, |, \varnothing, \varepsilon, (,)\}$ con $T \cap M = \varnothing$
Si definisce ESPRESSIONE REGOLARE (e.r.) su $R = T \cup M$ una delle seguenti stringhe r:
1 - $r = \varnothing$ espressione vuota
2 - $r = \varepsilon$ espressione nulla
3 - $r = c \ \ \forall c \in T$ espressione elementare
oppure una stringa r di simboli di R ottenuta applicando (ripetutamente) le seguenti regole, che forniscono e.r. valide a partire da e.r. valide s e t:
4 - r = (s|t) alternativa
5 - r = (s•t)=(st) concatenazione
6 - r = (s*) chiusura (stella)
Si ottiene pertanto una espressione con parentesi da cui, interpretando * come operatore unario applicato alla sottoespressione che esso segue, e | e • come operatori binari infissi con priorità crescente (| < •), si possono omettere le parentesi ridondanti e rendere implicito l'operatore •.
Il linguaggio generato dall'espressione regolare si ottiene applicando regole corrispondenti:
1 - $L(\varnothing) = \varnothing$
2 - $L(\varepsilon) = \{\varepsilon\}$
3 - $L(c) = \{c\}$
4 - $L(s|t) = L(s) \cup L(t)$
5 - $L(s\bullet t) = L(s) \,||\, L(t) = L(s)L(t)$
6 - $L(s^*) = (L(s))^*$

Esempi (fra [] le regole di volta in volta applicate):
r = a(ab)*(c|d) = ((a((ab)*))(c|d))
r1 = a r2 = b r3 = c r4 = d [3]
r11 = (r1r2) [5] r22 = (r3 | r4) [4]
r111 = (r11*) [6]
r1111 = (r1r111) [5]
r = (r1111r22) [5]
$L(r) = L(a)(L(ab))^* (L(c) \cup (L(d)) =$ [distribuendo] =
$= L(a)(L(ab))^* L(c) \cup L(a)(L(ab))^* L(d) =$
$= \{a(ab)^n c\} \cup \{a(ab)^n d\}$ n>=0

Un linguaggio generato da una espressione regolare si dice LINGUAGGIO REGOLARE. Per le caratteristiche della definizione si ha che se si opera la concatenazione, l'unione, la stella, la croce e la potenza su linguaggi regolari si ottiene ancora un linguaggio regolare. Pertanto la classe dei linguaggi regolari è CHIUSA rispetto alle suddette operazioni. Risulta inoltre:

Associatività della concatenazione:	rst = (rs)t = r(st)
Associatività della alternativa:	r|s|t = (r|s)|t = r|(s|t)
Commutatività della alternativa:	r | s = s | r
Distributività:	(r | s) t = rt | st
Idempotenza:	(r*)* = r*

Per comodità di notazione, le espressioni regolari vengono arricchite con nuovi metasimboli che non aggiungono nulla alla potenza espressiva della definizione originaria. Le estensioni proposte sono quelle utilizzate dal programma LEX (standard in UNIX) e l'insieme esteso dei metasimboli M è qui sotto brevemente descritto:

" \	protezione dei metasimboli
[ ]	classi di simboli terminali
–	range di simboli
^	complemento rispetto a T e inizio linea
•	simbolo qualsiasi

? opzionale
* + chiusura
| alternativa
( ) raggruppamento
$ fine linea
/ contesto successivo
{ } ripetizione (potenza) e sostituzione
< > condizione di stato

Per una trattazione più ampia si veda il manuale in linea di Lex (in un sistema UNIX) oppure ad esempio :
http://www.via.ecp.fr/~eb/textes/minimanlexyacc-english.html
Un'altra utile estensione è quella di identificare sottoespressioni mediante un simbolo che può essere utilizzato una o più volte come parte di una successiva espressione (come se si facesse una assegnazione parziale). In alcuni casi viene anche esplicitato il fatto che la stringa generata termina con un carattere speciale di fine testo (EOT).
Per illustrare un esempio concreto di uso delle espressioni regolari per definire la componente lessicale di un linguaggio artificiale, si definisce il lessico di un ipotetico linguaggio di programmazione (Cl0) per il calcolo delle espressioni algebriche su numeri naturali e variabili simboliche. Il linguaggio consente di scrivere assegnazioni a variabili simboliche di espressioni con i consueti operatori dell'algebra intera, consente inoltre la chiamata a funzioni di libreria già definite.

T è costituito dal set di caratteri ASCII; le sottoespressioni utili sono le seguenti:

```
/* classi di caratteri */
alpha      [@_A-Za-z]
digit      [0-9]
delim      [ \t\r\n\f\v]
oper        [;=()*+\-/,^]
anybutnewl [^\n]
anyprintbutapex [^']
```

I tipi di elementi lessicali (*token*) previsti sono i seguenti:

Keyword	`in out end`
Operatori	`; = ( ) * + - / , ^`
Costante naturale	es. `123456`
Costante stringa	es. `'Questa e'' una stringa'` (l'apice all'interno della stringa è rappresentato da una sequenza di due apici)
Identificatore	es. `pippo pip34 pip_plu`
Separatori	spazio, *tab*, *newline*
Commento	es. `! commento` <fine riga>

Per ogni classe di *token* viene definita una espressione regolare che genera tali elementi; il linguaggio sorgente viene pensato come chiusura stella del linguaggio unione di quelli generati dalle singole espressioni, cioè:
$L = LU^* = \{L(r1) \cup L(r2) \cup ... L(rn)\}^*$
con r1, r2,...rn espressioni regolari di n classi di *token*. Si noti che in generale già L(ri) può essere non finito, anche se nella pratica i singoli *token* del testo sorgente hanno lunghezza limitata.

Per il linguaggio Cl0 vengono identificate le seguenti classi:

```
separatori        r1 = {delim}+
commento          r2 = "!"{anybutnewl}*"\n"
operatore         r3 = {oper}
numero            r4 = {digit}+
identificatore    r5 = {alpha}({alpha}|{digit})*
stringa           r6 = "'"({anyprintbutapex}*"''"?)*"'"
```

Si tenga presente che, in particolare quando il numero di parole chiave è considerevole, si preferisce evitare di definire una e.r. per ciascuna di esse, riconoscere a livello lessicale una parola chiave come un normale identificatore e successivamente, appoggiandosi alla tabella dei simboli, verificare che si tratta di un simbolo predefinito.

2. Analisi lessicale

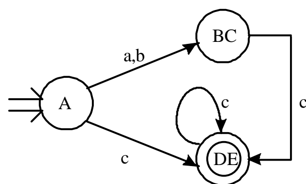
L'analisi lessicale ha come scopo la identificazione delle sottostringhe "elementari" che formano la stringa d'ingresso. Ciascuna sottostringa soddisfa ad una regola di un insieme di regole che costituiscono il "lessico": le regole sono dette perciò "regole lessicali" e ciascuna di esse può essere espressa, come visto, in in forma di una espressione regolare. Quando prefissi di lunghezza diversa della stessa sottostringa soddisfano tutti regole del lessico, si opterà per quello più lungo; quando invece la stessa sottostringa verificasse regole diverse, si farà riferimento ad una predefinita priorità tra le regole.
L'analizzatore è costituito di due parti: lo "scanner" che provvede alla suddivisione controllando, simbolo per simbolo, la corrispondenza con le regole lessicali, e l'"analizzatore" propriamente detto che riceve dallo scanner le singole sottostringhe (lessemi) e una codifica di tipo ricavata in base alla regola lessicale che ciascun lessema soddisfa. L'analizzatore provvede a

funzioni ausiliarie quali la eliminazione dei lessemi ridondanti (tipicamente sequenze di separatori e commenti), l'inserimento del lessema in una tabella detta "tabella dei simboli" e la costruzione di un record (token) che sintetizza le informazioni associate al lessema corrente (tipicamente un riferimento alla tabella dei simboli, un codice di tipo ed eventualmente altre informazioni quali il numero di riga in cui compare). Generalmente l'analizzatore lessicale si configura come una funzione che ad ogni chiamata legge i simboli del testo sorgente fino a riconoscere il prossimo lessema e restituisce il token relativo (se utile).
La tabella dei simboli è organizzata in modo da contenere tutte le informazioni utili associate a lessemi distinti: allo scopo si utilizza una delle tecniche per la gestione efficiente di dizionari (hashing, alberi binari ecc.). L'analizzatore inserisce le informazioni di sua pertinenza mentre, attraverso il token, altre componenti superiori (ad esempio un analizzatore semantico) possono aggiungere altre informazioni (attributi). I token predefiniti (operatori, keyword) possono essere preventivamente inseriti nella tabella dei simboli.

3. Equivalenza espressione regolare - DFA

Esiste un preciso criterio di equivalenza tra un'espressioni regolare e un automa a stati finiti deterministico, per il quale è possibile trovare un DFA M tale che: $L(M) \equiv L(r)$
L'espressione regolare associata ad un automa si può desumere spesso con facilità per ispezione sui cammini del grafo a partire dallo stato iniziale. L'esperienza porta inoltre a tentare direttamente il passaggio da espressione regolare, che è il modo più sintetico e immediato di descrivere un linguaggio regolare, ad un automa equivalente, soprattutto se l'espressione regolare non è molto complessa. In seguito ciò verrà fatto per il linguaggio Cl0. Alcuni generatori automatici (come Lex) creano con regole ben stabilite dalla espressione regolare un NFA (automa detto 'non deterministico') che viene poi trasformato, sempre applicando precise regole, in un DFA.
La realizzazione da programma di un riconoscitore per linguaggi regolari corrisponde a simulare le mosse del DFA equivalente. Si supporrà che la stringa di ingresso sia terminata da un simbolo speciale EOT che ne identifica la terminazione (non essendone nota a priori la lunghezza). L'utilizzazione di un automa deterministico elimina ogni ambiguità di scelta in ciascun passo. La durata del riconoscimento (cioè il numero di mosse necessarie) è proporzionale alla lunghezza della stringa d'ingresso.
Per la simulazione del DFA si possono utilizzare due tecniche: mediante uso di switch o goto (forma procedurale) e mediante tabelle di stato/azione (forma tabellare).
La forma con goto codifica in sostanza gli stati con altrettante posizioni del programma. Il salto dalla posizione corrente ad un'altra corrisponde ad una mossa purché consentita dalla definizione dell'automa nella configurazione corrente (cioè in dipendenza del simbolo d'ingresso). Quindi ogni mossa sarà condizionata da istruzioni di test sul simbolo d'ingresso: qualora le possibilità siano multiple (molti archi di uscita dallo stato corrente), si può utilizzare una istruzione di tipo "case of" (switch nel caso del C) che consente un test multiplo, in alternativa alla successione di if-then-else. In pratica l'automa diventa il flow-chart del programma che lo realizza e che pertanto risulta in generale non strutturato. La ispezione del codice consente eventuali ottimizzazioni (maggiore strutturazione, usi di cicli while, eliminazione di istruzioni ripetute ecc.).
La seconda forma prevede la codifica degli stati mediante interi e la inizializzazione di una tabella di stato futuro t che rappresenta la funzione $\delta(q, a)$. t si configura come una matrice nxm ove n è il numero degli stati e m la cardinalità dell'alfabeto d'ingresso. Il simulatore è di tipo generale e si limita semplicemente ad aggiornare una variabile di stato s con il valore s'=t[s,a], essendo a la codifica del simbolo d'ingresso. Gli stati finali potranno essere conservati in una lista a parte. Riservando una codifica di stato speciale per le transizioni non ammesse, si potrà interrompere la scansione per stringhe non accettate. Al termine della stringa si controllerà se lo stato corrente è finale o meno, rispettivamente decretando o dichiarando fallito il riconoscimento della stringa.

Ad esempio si consideri il linguaggio: $L(M) = (a|b)c+ \mid c+ = (a|b|\varepsilon)cc^*$ l'automa equivalente è il seguente:



Per quanto attiene la realizzazione da programma, la forma procedurale è data dal seguente segmento di codice in linguaggio C (getsym acquisisce la codifica del carattere-simbolo successivo, qui per semplicità assunta come ASCII):

```
A:
    switch(getsym()) {
      case 'a':
      case 'b':
        goto BC;
      case 'c':
        goto DE;
      default:
        error();
    }
```

```
BC:
    if (getsym() == 'c')
        goto DE;
    else
        error();

DE:
    if ((c=getsym()) == 'c')
        goto DE;
    else if (c == EOT)
        accepted();
    else
        error();
```

L’equivalente Java, anche con alcune semplici ottimizzazioni, è rappresentato dal seguente codice:

```
// A:
    if ( ((c=getsym()) == 'a') || (c == 'b'))
    {
// BC:
        if (getsym() != 'c')
            error();
    }
    else if (c != 'c')
        error();
// DE:
    while ((c = getsym()) == 'c');
    if (c == EOT)
        accepted();
    else
        error();
```

La forma tabellare per lo stesso automa prevede le seguenti definizioni:

codifiche degli stati: ACP -2 ERR -1 A 0 BC 1 DE 2

codifiche degli ingressi: EOT 0 a 1 b 2 c 3 altro 4

```
t[3][5] = {
/*        EOT   a   b   c   altro */
/* A  */ { -1,  1,  1,  2,  -1},
/* BC */ { -1, -1, -1,  2,  -1},
/* DE */ { -2, -1, -1,  2,  -1}
};
```

Si noti come si sia riservata una codifica di stato speciale (-1) per le transizioni non ammesse e che la condizione di accettazione sia stata tradotta con una transizione speciale con ingresso EOT, per il quale si è riservata una codifica, e rappresentata da un codice di stato pure speciale (-2). Così operando, il simulatore è molto semplice e indipendente dalla definizione dell'automa. Si noti inoltre che risulta conveniente riservare una codifica unica per tutti i simboli che, pur non appartenendo formalmente a T, possono comparire come ritorno della funzione getsym (è questo ad esempio il caso in cui T è un sottoinsieme proprio del set ASCII), per consentirne la gestione di errore.

```
    s = 0;  /* stato iniziale */
    while ((s = t[s][code(getsym())]) >=0);
    if (s == -1)
        error();
    else
        accepted();
```

La funzione code() effettua la trascodifica dei caratteri d'ingresso.

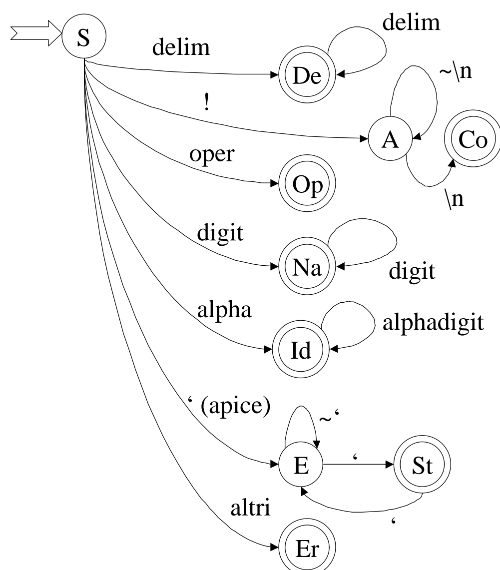
Anche nella versione tabellare si può operare una ottimizzazione osservando che alcuni simboli d'ingresso comportano le stesse transizioni partendo da ciascuno degli stati, cioè tali per cui le rispettive colonne della tabella t sono eguali. Nell'esempio ciò si verifica per a e b. Dando una unica codifica per i simboli "equivalenti" si ottiene una riduzione della tabella stessa.

codifiche degli ingressi: EOT 0 a,b 1 c 2 altro 3

```
t[3][4] = {
/*        EOT a,b   c   altro */
/* A  */ { -1,  1,  2,  -1},
/* BC */ { -1, -1,  2,  -1},
```

```
/* DE */ { -2, -1,  2,  -1}
};
```

Per il linguaggio Cl0 il DFA equivalente è quello della figura che segue. Si noti che ogni stato finale corrisponde ad un particolare tipo di *token*, la cui codifica viene restituita dall'analizzatore lessicale come risultato del riconoscimento. Si noti anche che, per tenere semplice il DFA, si preferisce riconoscere inizialmente come identificatore anche una parola chiave, salvo provvedere al suo corretto riconoscimento mediante una successiva ricerca in una tabella dei simboli precaricata con tutte le parole chiave.



A1. Definizioni base

Un ALFABETO T è un insieme finito di elementi distinti detti "simboli terminali" ovvero:
$T = \{c_1, c_2, .. c_k\}$
$|T| = k$ è la CARDINALITÀ dell'alfabeto
Dato un alfabeto T, si dice STRINGA su T di lunghezza n la n-upla ordinata: $s = s_1 s_2 .. s_n$ con $s_i \in T$. Si denoterà $n = |s|$. Per estensione viene definita STRINGA NULLA (VUOTA) la stringa ε priva di simboli e di lunghezza 0.
Due stringhe sono eguali se hanno la stessa lunghezza e se i simboli in posizione corrispondente sono eguali.
Si definisce l'insieme di tutte le possibili stringhe s su T aventi determinata lunghezza k la potenza T^k:
$T^k = \{s \mid |s|=k\ k>=0\}$
Pertanto: $T^0 = \{\varepsilon\}$ $T^1 = \{"c_1", "c_2", .. "c_k"\}$ c_i simboli terminali (d'ora in avanti si ometteranno gli apici quando il contesto chiarirà che si intendono stringhe monocarattere e non singoli caratteri)
Si definisce l'insieme (non finito) di tutte le possibili stringhe su T di lunghezza >=1:
$T^+ = \cup_i T^i$ con $1 <= i < \infty$
Se si aggiunge anche la stringa nulla, si ottiene l'insieme: $T^* = T^+ \cup \{\varepsilon\} = T^+ \cup T^0 = \cup_i T^i$ con $0 <= i < \infty$
Risulta in generale: $T^i \subset T^*$
Per sintesi si utilizzerà la notazione a^k come composta di k>=0 simboli eguali ad $a \in T$ ($a^0 \equiv \varepsilon$).
Date due stringhe su T s, s': $s = s_1 s_2 .. s_n$ $s' = s'_1 s'_2 .. s'_{n'}$
si definisce CONCATENAZIONE l'operazione di giustapposizione che fornisce la stringa:
$s'' = s \,||\, s' = s_1 .. s_n s'_1 .. s'_{n'}$ $|s''| = |s| + |s'| = n + n'$ $s = s \,||\, \varepsilon = \varepsilon \,||\, s$
Ove non ambiguo, la concatenazione s||t sarà indicata più semplicemente st.
La concatenazione puo` essere applicata più volte e gode della proprietà associativa ma non di quella commutativa, ovvero:
$z = stu = (st)u = s(tu)$ $xy \neq yx$ (in generale)
Se è z = stu, s, t e u sono SOTTOSTRINGHE di z, s è un PREFISSO di z, u è un SUFFISSO di z. Si dice PREFISSO DI LUNGHEZZA k di z con |z| >= k e si indica con z/k il prefisso s di z con |s|=k. Si dice riflessione di $s=s_1..s_n$ la stringa:
$s^R = s_n..s_1$
Risulta: $(s^R)^R = s$ $\varepsilon^R = \varepsilon$ $(st)^R = t^R s^R$
Si definisce ricorsivamente POTENZA k-ESIMA (k>=0) di una stringa s la stringa:
$s^k = \varepsilon$ per k=0
$= s^{(k-1)} s$ per k>0
Si definisce L linguaggio su alfabeto T un insieme di stringhe su T, cioè $L \subseteq T^*$
Esempi:
T1 = {A, B, ... X, Y, Z} lettere maiuscole |T1| = 26
T2 = {0, 1} cifre binarie |T2| = 2
T3 = {⊗, ⊕, ∅, ∩, ∪, ⊃, ⊇, ⊄, ⊂, ⊆, ∈} simboli grafici |T3| = 11
$s11 = AABZAT \in T1^*$ |s11| = 6

$s12 = CC \in T1^2$ $|s12| = 2$
$s13 = L \in T1^2$ $|s13| = 1$
$s21 = 0100100 \in T2^+$ $|s21| = 7$
$s22 = \varepsilon \in T2^*$ $|s22| = 0$
$s31 = \otimes\oplus\otimes \in T3^3$ $|s31| = 3$
$s32 = \varnothing\varnothing\varnothing\varnothing \notin T3^5$ $|s32| = 4$
$s14 = (s11 \parallel s12) = AABZATCC <> (s12 \parallel s11) = CCAABZAT$ $|s14| = |s11| + |s12| = |s12| + |s11| = 8$
s15 = s12 s11 s13 = CCAABZATL
s12 = CC è prefisso di s15
s11 = AABZAT è sottostringa si s15
s13 = L è suffisso di s15
s15/4 = CCAA
s23 = s21 s22 = 0100100 = s22 s21 = s21
$s24 = s21^R = 0010010$
$s25 = s21^3 = 010010001001000100100$
|s25| = |s21|*3 = 21
$s34 = s32^0 = \varepsilon$

L11 = {A, L, Z, AM, ACC} definito per enumerazione
$L12 = \{x \mid x \in T1^*$ e x non contiene due lettere eguali$\}$ definito mediante regola
L12 è finito poiché per la regola di definizione risulta:
max |x| = |T1| = 26
$L21 = \{x \mid x \in T2^*$ e $\forall k>0$ si hanno solo due stringhe $x1 \neq x2$
con $|x1| = |x2| = k$ e $x2 = x1^R\}$
L21 è non finito.

A2. Operazioni sui linguaggi

Dati due linguaggi: $L1 \subseteq T1^*$ $L2 \subseteq T2^*$
si definisce CONCATENAZIONE DEI DUE LINGUAGGI il linguaggio: $L3 = L1 \parallel L2 = L1\ L2 = \{xy \mid x \in L1 \text{ e } y \in L2\}$
cioè ottenuto dalla concatenazione di una stringa di L1 con una stringa di L2 in tutte le combinazioni possibili. Risulta come conseguenza che: $L1\ L2 \subseteq (T1 \cup T2)^*$ $L1\ \{\varepsilon\} = \{\varepsilon\}\ L1 = L1$
Si definisce LINGUAGGIO VUOTO ∅ per estensione l'insieme vuoto, cioè senza alcuna stringa, per il quale vale sempre:
$L1\ \varnothing = \varnothing\ L1 = L1$
Si noti che $|\varnothing| = 0 \neq |\{\varepsilon\}| = 1$.
Si definisce ricorsivamente POTENZA K-ESIMA di un linguaggio L il linguaggio:
$L^k = \{\varepsilon\}$ per k = 0
$= L^{(k-1)}\ L$ per k > 0
e si ottiene dalla applicazione ripetuta k volte della concatenazione di stringhe del linguaggio originario. Si definisce CHIUSURA TRANSITIVA NON RIFLESSIVA (croce) il linguaggio limite: $L^+ = \cup_i L^i$ $1 <= i < \infty$
e contiene tutte le stringhe che si possono ottenere o come stringhe del linguaggio originario o da concatenazioni di tali stringhe. Se si aggiunge anche la stringa nulla si ottiene la CHIUSURA TRANSITIVA RIFLESSIVA (stella):
$L^* = L^+ \cup \{\varepsilon\} = \cup_i L^i$ $0 <= i < \infty$
Se $L = T^1$ allora $L^* = T^*$ cioè l'insieme non finito che contiene tutte le stringhe formate da simboli di T (alfabeto del linguaggio): T* è detto LINGUAGGIO UNIVERSALE.
Poiché in generale i linguaggi sono insiemi di stringhe, si possono definire le operazioni ∪ (unione), | (intersezione) e - (differenza) che danno ancora linguaggi come risultato; confronti quali ⊆ (inclusione), ⊂ (inclusione stretta) e = (eguaglianza) che danno risultato logico; inoltre essendo $L \subseteq T^*$ è possibile definire il COMPLEMENTO di L come:
$\sim L = T^* - L = \{x \mid x \in T^* \text{ e } x \notin L\}$
Proprietà della chiusura:

$L \subseteq L^*$	le stringhe originarie sono presenti nella chiusura riflessiva
$x,y \in L^* ==> xy \in L^*$	è chiusa rispetto alla concatenazione delle sue stringhe
$(L^*)^* \equiv L^*$	è chiusa rispetto all'autoconcatenazione
$(L^*)^R \equiv (L^R)^*$	invertibilità degli operatori R e * (I^R è l'insieme delle riflessioni delle stringhe dell'insieme I)

Esempi:
$L1 = \{a^i \mid i>=0, \text{pari}\}$ $L2 = \{b^j a \mid j>=1, \text{dispari}\}$
$L1\ L2 = \{a^I b^j a \mid i>=0 \text{ pari}, j>=1 \text{ dispari}\} = \{eba, a^2ba, a^4ba, ..., eb^3a, a^2b^3a, a^4b^3a, ...\} = \{ba, aaba, aaaaba, ..., bbba, aabbba, ...\}$
$L1^2 = \{ee, ea^2, ea^4, ..., a^2e, a^2a^2, ...\} = L1 = L^*$
$L1^+ = L^*$ poiché L1 contiene già ε
L3 = {ab, ba}
$L3^* = \{\varepsilon\} \cup \{ab, ba\} \cup \{abab, abba, baab, baba\} \cup ...$
$L3^+ = \{ab, ba\} \cup ... \neq L3^* = L3^+ \cup \{\varepsilon\}$

L3 è finito, $L3^*$ no.

casi limite:
$\emptyset^0 = \{\varepsilon\}$
$\emptyset^* = \emptyset^0 \cup \emptyset^1 \cup ... = \emptyset^0 = \{\varepsilon\}$
$\{\varepsilon\}^* = \{\varepsilon\}$

TA = {A, B, ... Z}
TN = {0, 1, ... 9}
T = TA $\cup$ TN
$L4 = TA\ (T \cup \{\varepsilon\})^5$ è il linguaggio degli identificatori di lunghezza non superiore a 6
T = {a} $\sim L = T^* - L = \{a^i \mid i>=1 \text{ dispari}\}$

A3. Cenni su grammatiche e analisi sintattica

Una grammatica (non contestuale) G è una quadrupla (N, T, P, S) con:
- N è un insieme finito e non vuoto detto "Alfabeto non terminale" e costituito da simboli detti "Tipi sintattici" (o semplicemente "Simboli non terminali");
- T è un insieme finito non vuoto di simboli detto "Alfabeto terminale" e costituito da simboli detti "Simboli terminali". N e T sono disgiunti e si definisce come "Alfabeto totale" l'insieme: $V = N \cup T$.
- P (insieme finito e non vuoto delle *produzioni*) è una relazione binaria tra N e V*, cioè ogni produzione ha una "Parte sinistra" costituita da un non terminale e una "Parte destra" costituita da una stringa di terminali e non terminali.

Sinteticamente una produzione di G è indicata con: $A \rightarrow \alpha$
- S è detto l'assioma ed è un particolare simbolo non terminale.

La notazione base per le produzioni viene chiamata BNF (*Backus-Naur Form*). Sono state definite alcune notazioni equivalenti più sintetiche. Si può ad esempio rappresentare produzioni che hanno la medesima parte sinistra con un'unica produzione secondo la seguente regola:
$A \rightarrow \alpha \mid \beta \mid \delta \mid ... \mid \eta$ equivale : $A \rightarrow \alpha$ $A \rightarrow \beta$ $A \rightarrow \delta$ $A \rightarrow \eta$
A partire dall'assioma S è possibile effettuare 'sostituzioni' che trasformano una stringa di terminali e non terminali, che contenga almeno un non terminale, in una stringa derivata. Ad esempio, usando la produzione $A \rightarrow \alpha$, la stringa $\eta A \varphi$ può essere sostituita dalla stringa $\eta\alpha\varphi$ e si indicherà: $\eta A \varphi \Rightarrow \eta\alpha\varphi$
Si dirà che $\eta\alpha\varphi$ "deriva direttamente secondo G" da $\eta A \varphi$. Se la derivazione produce una stringa che è ancora derivabile, significa che la stringa derivata contiene ancora non terminali. È allora possibile applicare la potenza k della derivazione se è vero che:

$$\alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_k$$

in questo caso si dice che α_k deriva in k passi da α_0 e si indica sinteticamente con $\alpha_0 \Rightarrow k\ \alpha_k$.
Se β deriva da α, allora si scrive: $\alpha \Rightarrow^* \beta$
cioè esiste un r>=0 tale che $\alpha \Rightarrow r\ \beta$ e si dice che β deriva da α (secondo G) oppure è $\alpha = \beta$. Se $\alpha \neq \beta$ allora la relazione è vera per r>0 e si indica con: $\alpha \Rightarrow + \beta$ r viene chiamata "lunghezza della derivazione".
È ora possibile definire il linguaggio generato da G come le stringhe di simboli TERMINALI ottenute per derivazione da S, ovvero: $L(G) = \{t \in T^* \mid S \Rightarrow^* t\}$
G è alcune volte detta "Sintassi" di L e le stringhe del linguaggio sono dette "frasi" di L.
Esempio:
G1 = { N={E, A}, T={+, -, *, /, (,), id}, P, E}
P: (ogni produzione è numerata)

```
E → EAE |        [1]
     (E) |       [2]
     -E |        [3]
     id          [4]
A → + |          [5]
     - |         [6]
     * |         [7]
     /           [8]
```

La stringa: E A id + E è derivabile in 4 passi da S. Infatti risulta (sopra le derivazioni la produzione applicata, mentre il non terminale espanso è sottolineato):

```
  [1]      [5]      [1]       [4]
E ⇒ EAE ⇒ E+E ⇒ EAE+E ⇒ EA id +E
```

Si noti che le derivazioni possono essere applicate in un altro ordine:

```
  [1]      [1]        [4]          [5]
E ⇒ EAE ⇒ EAEAE ⇒ EA id AE ⇒ EA id +E
```

Il linguaggio generato da G1 è quello delle espressioni aritmetiche con parentesi, operatori infissi e operatore - prefisso. Ad esempio:

$E \Rightarrow EAE \Rightarrow EAEAE \Rightarrow$ id AEAE $\Rightarrow$ id * EAE $\Rightarrow$ id * (E) AE $\Rightarrow$ id * (EAE) AE $\Rightarrow$ id * (id AE) AE $\Rightarrow$ id * (id + E) AE $\Rightarrow$ id * (id + id) AE $\Rightarrow$ id * (id + id) / E $\Rightarrow$ id * (id + id) / id

Secondo le definizioni originarie, le grammatiche non generano ε (stringa nulla): nella pratica è conveniente poter avere ε nel linguaggio generato e pertanto si rilasserà il vincolo posto accettando per tali grammatiche produzioni del tipo A→ε poiché si può dimostrare che, a meno della stringa ε, le grammatiche senza ε-produzioni generano la stessa classe di linguaggi delle rispettive grammatiche estese con ε-produzioni.

Una grammatica generatrice può essere tradotta in un algoritmo che, scandendo le regole sintattiche, genera in modo automatico stringhe del linguaggio associato. Invece la possibilità di realizzare un algoritmo riconoscitore, cioè un programma in grado di verificare se un testo d'ingresso, visto come stringa dell'alfabeto d'ingresso, appartenga al linguaggio generato dalla grammatica, richiede che, nel caso di stringa corretta, ad ogni passo l'algoritmo scelga la produzione che dovrebbe essere applicata se quella stringa fosse generata e invece, nel caso di una stringa non appartenente al linguaggio (quindi per noi sintatticamente scorretta) l'algoritmo posso accorgersene il più presto possibile, segnalandolo. Si tenga presente che alcune grammatiche non consentono di ottenere un algoritmo riconoscitore deterministico, nel senso che con un certo ingresso è possibile avere più produzioni applicabili anche in caso di stringa corretta. Nella pratica si evita di usare questo tipo di grammatiche in modo da rendere deterministica la scelta della produzione da applicare ad ogni passo solo sulla base del simbolo d'ingresso.

Una derivazione può essere rappresentata in modo equivalente con un albero (albero *sintattico* o *parse tree*) i cui nodi intermedi sono etichettati con non terminali, le foglie con i terminali della stringa derivata, scritti nell'ordine da sinistra a destra. Ogni nodo intermedio ha come figli nodi etichettati con la parte destra della produzione che è stata utilizzata per sostituire il non terminale che etichetta quel nodo (padre). Ad esempio, per la derivazione sopra, l'albero sintattico è il seguente (in notazione a parentesi):

```
[E    [E
            [E [id]
            A [*]
            E     [( E
                        [E [id]
                        A [+]
                        E [id]
                  ] )
            ]
      A [/]
      E [id]]]
```

Per "analisi sintattica" si intende quella fase che, ricevendo uno stream di *token* fornito attraverso successive chiamate dell'analizzatore lessicale, *token* che vengono considerati, dal punto di vista della grammatica, simboli terminali d'ingresso, è in grado di determinare del testo d'ingresso la struttura, costruendo una immagine in memoria sottoforma di "alberi sintattici". La costruzione avviene sulla base della definizione grammaticale del linguaggio di cui il testo dovrebbe rappresentare una stringa. L'analisi deve pertanto segnalare se il testo si discosta da tale definizione (errori sintattici) e possibilmente ovviare agli effetti degli errori riscontrati. Esistono due tipi fondamentali di analisi: discendente, che cerca di costruire l'albero sintattico dalla radice cioè dall'alto; ascendente, che cerca di costruire l'albero sintattico dalle foglie, cioè dal basso.

Purtroppo esistono grammatiche ambigue per le quali la medesima stringa terminale ammette più di una derivazione. Ad esempio la stringa id+id*id ammette per G1 le due derivazioni:

$E_0 \Rightarrow E_1AE_2 \Rightarrow E_3AE_4AE_2 \Rightarrow$ id $AE_4AE_2 \Rightarrow$ id + $E_4AE_2 \Rightarrow$ id + id $AE_2 \Rightarrow$ id + id * $E_2 \Rightarrow$ id + id * id

$E_0 \Rightarrow E_1AE_2 \Rightarrow E_1AE_3AE_4 \Rightarrow$ id $AE_3AE_4 \Rightarrow$ id + $E_3AE_4 \Rightarrow$ id + id $AE_4 \Rightarrow$ id + id * $E_4 \Rightarrow$ id + id * id

(i pedici sono stati aggiunti per specificare meglio la derivazione ad ogni passo) da cui è possibile ricavare due alberi distinti, il secondo dei quali rispetta l'aggregazione che viene indotta dalla normale priorità degli operatori, il primo no. Già questo semplice esempio fa vedere che è bene evitare l'uso di grammatiche ambigue al fine del riconoscimento, in modo da ricavare, data una certa stringa d'ingresso, un solo albero sintattico. Una grammatica base non ambigua per espressioni algebriche è la seguente:

G2= { N={E, T, F}, T={+, -, *, /, (,), id}, P, E}

E → E+T | E-T | T

T → T*F | T/F | F

F → (E) | -F | id

Essa produce come la precedente espressioni con parentesi (il non terminale A di G1 è stato direttamente sostituito nelle parti destre con gli operatori) e con la stessa stringa d'ingresso si ottiene un'unico albero sintattico. Per esercizio si ricavi l'albero per la già vista stringa id+id*id.

Solo alcune grammatiche consentono di effettuare un riconoscimento discendente tale per cui, in presenza di più parti destre con lo stesso non terminale come parte sinistra da sostituire, si riesce a scegliere la produzione da applicare sulla sola base del simbolo corrente in lettura. Queste grammatiche sono dette LL1 (*Leftmost - lookahead(1)*, cioè con scansione della stringa d'ingresso da sinistra e con prospezione (*lookahead*) di un solo simbolo d'ingresso): ad esempio la grammatica G2, malgrado sia non ambigua, non è LL1. Una grammatica LL1 ad essa equivalente è la seguente:

G3= { N={E, EE, T, TT, F}, T={+, -, *, /, (,), id}, P, E}

E → T EE

EE → + T EE | - T EE | ε

T → F TT

TT → * F TT | / F TT | ε

F → (E) | -F | id

Il riconoscitore, nel momento in cui deve sostituire ad esempio EE, utilizzerà la produzione EE → + T EE se il simbolo d'ingresso è '+', EE → - T EE se il simbolo d'ingresso è '-',EE → ε altrimenti.

Un modo per realizzare un riconoscitore discendente a partire dalla grammatica LL1 è quello di definire una funzione (metodo) per ogni non terminale della grammatica, funzione che si incarica di applicare le parti destre delle produzioni che hanno, come parte sinistra, il non terminale associato. La scelta della parte destra da applicare avverrà come detto sulla base di regole derivate dalla grammatica (qui nell'esempio solo intuibili) e sulla base del simbolo d'ingresso. Una volta scelta la produzione, ad ogni simbolo della parte destra, che viene scandita dalla funzione, corrisponde un'azione. Se il simbolo è terminale, se ne controlla l'eguaglianza con il simbolo in lettura: se verificata, si avanza la 'testina' di lettura al simbolo successivo e si procede con la parte destra, altrimenti si decreta un errore sintattico. Se il simbolo è non terminale, viene effettuata una chiamata alla funzione ad esso associata, al ritorno dalla quale si procede con la scansione della parte destra. Se il simbolo è ε (stringa nulla) semplicemente si ritorna al chiamante. L'analisi va a buon fine se, dopo aver inizialmente letto il primo simbolo di ingresso e attivata la funzione associata all'assioma, da quest'ultima chiamata si ritorna dopo aver letto l'intera stringa d'ingresso (ovvero il simbolo corrente è EOT). Osservando che la grammatica può presentare ricorsioni nei non terminali, da ciò discende in generale la ricorsività della funzioni associate (ad esempio per G2 un percorso ammesso nell'albero sintattico è E — T — F — E con ricorsione indiretta su E).

Per esemplificare la costruzione dell'analizzatore secondo queste regole, i non terminali E ed EE in G3 produrrebbero le seguenti due funzioni (ricorsive):

```
void E()
{
    T();
    EE();
}

void EE()
{
    if (curtoken() != PLUS && curtoken() != MINUS)
        // terza produzione, non fa null
        return;
    // prima o seconda produzione, sono simili
    gettoken(); // salta il '+' o il '-' e passa al successivo
    T();
    EE();
}
```

Una forma equivalente e più sintetica della BNF è detta EBNF (*extended* BNF) e utilizza metasimboli simili a quelli della espressioni regolari, pensate definite su V = T∪N. Ad esempio la grammatica G2 può essere resa in questa notazione estesa nel modo seguente:

G4 = ({E,T,F}, {i,+,-,*,/,(,)}, P, E)

E → T ((+|-)T)*

T → F ((*|/)F)*

F → (-)* ("(" E ")" | id)

(sono state messa tra apici le parantesi che sono simboli terminali e non metasimboli della notazione estesa). La EBNF porta di solito a realizzare la funzione di riconoscimento in modo più efficiente. Ad esempio per la E:

```
void E()
{
    T();
    while(curtoken() == PLUS || curtoken() == MINUS)
    {
        gettoken();
        T();
    }
}
```

Si possono definire per questa notazione i concetti di derivazione e albero sintattico in analogia con la definizione BNF. Gli alberi sintattici che ne risultano possono avere ora una grado illimitato.