Grammatiche e automi

Analisi lessicale e sintattica

Analisi lessicale e sintattica

Le prime fasi del compilatore:

- divisione del codice in token
- costruzione dell'albero sintattico

Gli strumenti software (scanner-lexer, parser) si basano su

- teoria delle grammatiche e
- automi

Si usano risultati presentati a Fondamenti dell'informatica

Descrizione della sintassi del linguaggio di programmazione

- Quali sequenze di caratteri formano un programma, e quali no
- più nel dettaglio, qual è la struttura sintattica di un programma
  divisione del programma in componenti e sotto-componenti quali:
  dichiarazioni, blocchi, cicli, singole istruzioni.
- Descrizione, completamente formale,
  ottenuta tramite grammatiche

Grammatiche

Formalismo nato per descrivere linguaggi, anche naturali

- Panini, IV secolo a.C., definisce una grammatica formale per il Sanscrito,
  in Occidente nessuna teoria così sofisticata sino al XX secolo;
- Noam Chomsky, anni ’50, descrizione formale dei linguaggi naturali

Formalismo che mi permette di generare:

- tutte le frasi sintatticamente corrette della lingua italiana
- tutti i programmi sintatticamente corretti

Ma anche mettere in evidenza la struttura di una frase, un programma

Grammatica costituita da:

- un insieme di simboli terminali, elementi base,
  a seconda dei casi può essere:
  - l'insieme di parole della lingua italiana
  - l'insieme dei simboli base di un linguaggio di programmazione:
    simboli di operazione, identificatori, separatori
  - lettere di un alfabeto $\mathcal{A}$
- un insieme di simboli non terminali, categorie sintattiche,
  a seconda dei casi possono essere:
  - nomi, verbi, articoli,
    predicato verbale, complemento di termine,
    proposizione, subordinata, ecc.
  - identificatori, costanti, operazioni aritmetiche
    comando di assegnazione, espressione aritmetica,
    ciclo, dichiarazione di procedura, . . .

Grammatica costituita da:

- un insieme di regole di generazione,
  spiegano come sono composti i non-terminali,
  come posso espandere un non terminale esempi di regole (generative):
  - proposizione $\rightarrow$ soggetto predicato verbale
  - proposizione $\rightarrow$ soggetto predicato verbale complemento oggetto
  - programma $\rightarrow$ dichiarazioni programma principale
  - assegnamento $\rightarrow$ identificatore “=” espressione
- le grammatiche vengono divise in classi
  in base alla complessità delle regole ammesse
  - grammatiche regolari, libere da contesto, dipendenti dal contesto, . . .
- regole più sofisticate:
  - permettono di definire più linguaggi, linguaggi più complessi
  - ma determinare se una parola appartiene alla grammatica diventa più complesso

Formalmente

Un linguaggio su alfabeto $\mathcal{A}$, è un sottoinsieme di $\mathcal{A}^*$, (stringhe su $\mathcal{A}$)

Definizione: Grammatica libera (dal contesto)

Definita da:

- T: insieme di simboli terminali (alfabeto del linguaggio)
- NT: insieme di simboli non terminali (categorie sintattiche)
- R: insieme di regole di produzione
- S: simbolo iniziale $\in$ NT

Regole R (libere da contesto) nella forma:

$$V \rightarrow w$$

con V $\in$ NT e w $\in$ (T $\cup$ NT)*

Esempio: stringhe palindrome

sull'alfabeto formato dalle sole lettere a, b,

- terminali: a, b
- non-terminali: P
- regole:
  - $P \to \varepsilon$
  - $P \to a$
  - $P \to b$
  - $P \to aPa$
  - $P \to bPb$
- simbolo iniziale: P

Le regole sono corrette e complete

Esempio: espressioni aritmetiche

a partire dalle variabili a, b, c

- terminali: a, b, c, +, -, *, (,)
- non-terminali: E, T, F, Var (espressione, termine, fattore)
- regole:
  - $E \to T$
    $E \to E + T$
    $E \to E - T$
  - $T \to F$
    $T \to T * F$
  - $F \to (E)$
    $F \to Var$
  - $Var \to a$, $Var \to b$, $Var \to c$
- simbolo iniziale: E

Formulazione alternativa: BNF (Backus-Naur Form)

- Sviluppata per l'Algol60
- non terminali marcati da parentesi angolari: $\langle E \rangle$
  (non devo definirli esplicitamente)
- $\rightarrow$ sostituita da ::=
- regole con stesso simbolo a sinistra raggruppate

  $\langle E \rangle$ ::= $\langle T \rangle$ | $\langle E \rangle$ + $\langle T \rangle$ | $\langle E \rangle$ - $\langle T \rangle$
- si usa una diversa meta-sintassi

Esistono notazioni miste:

`E` $\rightarrow$ `T | E + T | E - T`

Derivazioni e alberi di derivazione

- Derivazione: come passare da simbolo iniziale a parola finale, una regola alla volta,
  - $E \to E + T \to T + T \to F + T \to Var + T \to a + T \to a + F \to$ $\to a + Var \to a + b$
  - $E \to E + T \to E + F \to E + Var \to E + b \to T + b \to F + b \to$ $\to Var + b \to a + b$
- Albero di derivazione:
  - rappresentazione univoca della derivazione,
  - mette in evidenza la struttura del termine.
- Formalmente gli alberi di derivazione sono alberi ordinati ossia:
  - grafo orientato,
  - aciclico, connesso
  - ogni nodo ha al più un arco entrante,
  - gli archi uscenti sono ordinati,

Albero di derivazione, definizione formale

L'albero di derivazione su una grammatica $\langle$ T, NT, R, S $\rangle$ soddisfa:

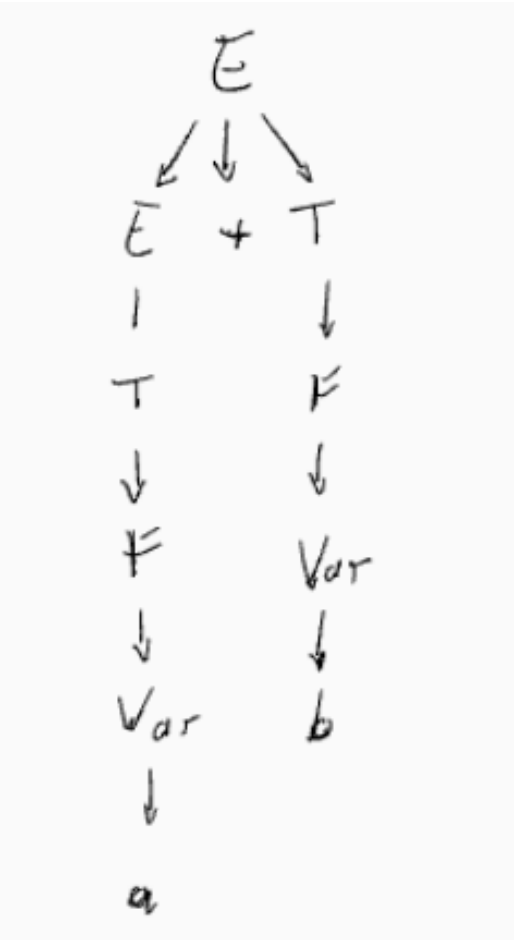
- radice etichettata con il simbolo iniziale S,
- foglie etichettate con simboli terminali in T,
- ogni nodo interno n
  - etichettato con un simbolo non terminale
  - la sequenza w delle etichette dei suoi figli, deve apparire in una regola E $\rightarrow$ w, in R

Alberi di derivazione fondamentali perché descrivono la struttura logica della stringa analizzata

I compilatori lavorano su alberi di derivazione

Esempio

- $E \rightarrow E + T \rightarrow T + T \rightarrow F + T \rightarrow Var + T \rightarrow a + T \rightarrow a + F \rightarrow$
  $\rightarrow a + Var \rightarrow a + b$



Grammatiche ambigue:

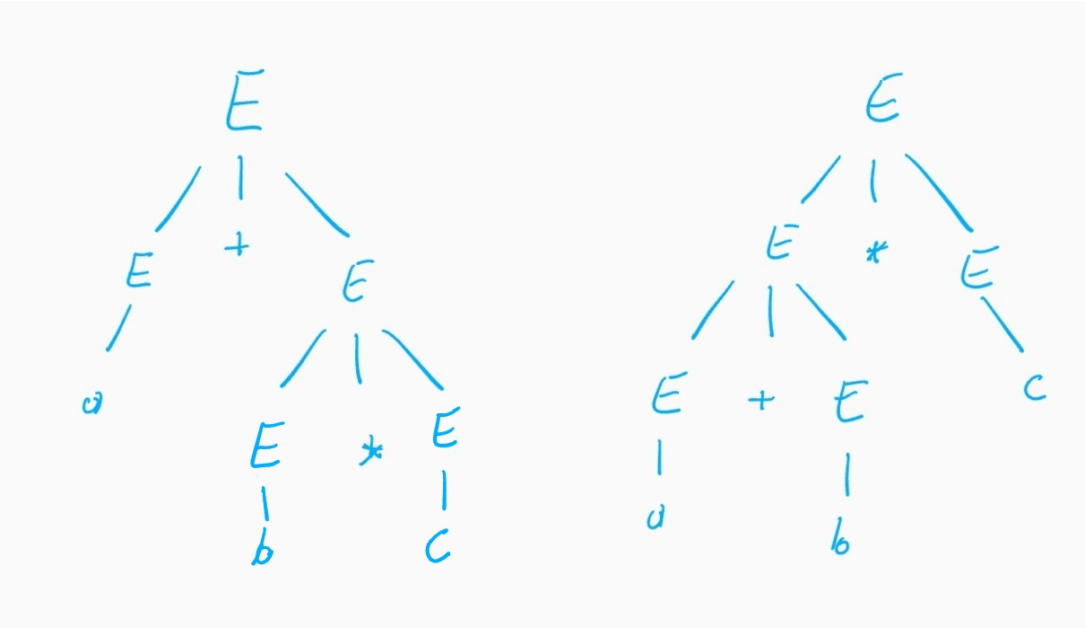
Una grammatica alternativa per le espressioni aritmetiche:

$$\texttt{E} \rightarrow \texttt{E + E} \mid \texttt{E - E} \mid \texttt{E * E} \mid \texttt{(E)} \mid \texttt{a} \mid \texttt{b} \mid \texttt{c}$$

Le due grammatiche generano lo stesso linguaggio ma la seconda è ambigua:

```
a + b * c
```

ha due alberi di derivazione nella seconda grammatica



Grammatiche ambigue

- Grammatiche per cui esiste una stringa con due alberi di derivazione differenti
- i due alberi di derivazione inducono
  - due interpretazioni diverse della stessa stringa
  - due meccanismi di valutazione differente
- Queste ambiguità vanno evitate (come in matematica)

Disambiguare una grammatica

Due soluzioni:

- rendere la grammatica non ambigua
  - tipicamente attraverso: nuovi non-terminali, non terminali
  - ottengo una grammatica che genera lo stesso linguaggio
    ma più complessa
- convivere con grammatiche ambigue
  - si forniscono informazioni aggiuntive su come risolvere le ambiguità
  - tipicamente, si specifica:
    - ordine di precedenza degli operatori,
    - per un singolo operatore, o
      per operatori nella stessa classe di equivalenza,
      se associativo a sinistra o a destra.

  soluzione usata anche nei parser

Altri esempi di ambiguità

la stessa grammatica

`E` $\rightarrow$ `E + E | E - E | E * E | ( E ) | a | b | c`

e le stringhe

```
a - b - c
a - b + c
```

bisogna stabilire se si associa a sinistra:

```
(a - b) - c
(a - b) + c
```

oppure si associa a destra:

```
a - (b - c)
a - (b + c)
```

Altri esempi di ambiguità

Grammatica

$$\langle \texttt{stat} \rangle \ ::= \ \texttt{IF} \ \langle \texttt{bool} \rangle \ \texttt{THEN} \ \langle \texttt{stat} \rangle \ \texttt{ELSE} \ \langle \texttt{stat} \rangle$$
$$\mid \ \texttt{IF} \ \langle \texttt{bool} \rangle \ \texttt{THEN} \ \langle \texttt{stat} \rangle \ \mid \ \ldots$$

ambigua su:

$$\texttt{IF} \ \langle \texttt{bool1} \rangle \ \texttt{THEN IF} \ \langle \texttt{bool2} \rangle \ \texttt{THEN} \ \langle \texttt{stat1} \rangle \ \texttt{ELSE} \ \langle \texttt{stat2} \rangle$$

due interpretazioni:

Altri esempi di ambiguità

Grammatica

⟨stat⟩ ::= IF ⟨bool⟩ THEN ⟨stat⟩ ELSE ⟨stat⟩
| IF ⟨bool⟩ THEN ⟨stat⟩ | ...

ambigua su:

IF ⟨bool1⟩ THEN IF ⟨bool2⟩ THEN ⟨stat1⟩ ELSE ⟨stat2⟩

due interpretazioni:

IF ⟨bool1⟩ THEN (IF ⟨bool2⟩ THEN ⟨stat1⟩ ELSE ⟨stat2⟩)

IF ⟨bool1⟩ THEN (IF ⟨bool2⟩ THEN ⟨stat1⟩) ELSE ⟨stat2⟩

In C-Java le parentesi della notazione evitano ambiguità

IF ⟨bool1⟩ { IF ⟨bool2⟩ {stat1} ELSE {stat2} }

IF ⟨bool1⟩ { IF ⟨bool2⟩ {stat1} } ELSE {stat2}

Abstract syntax tree

Gli alberi di derivazione contengono informazioni utili per interpretare, valutare, dare semantica alle stringhe.

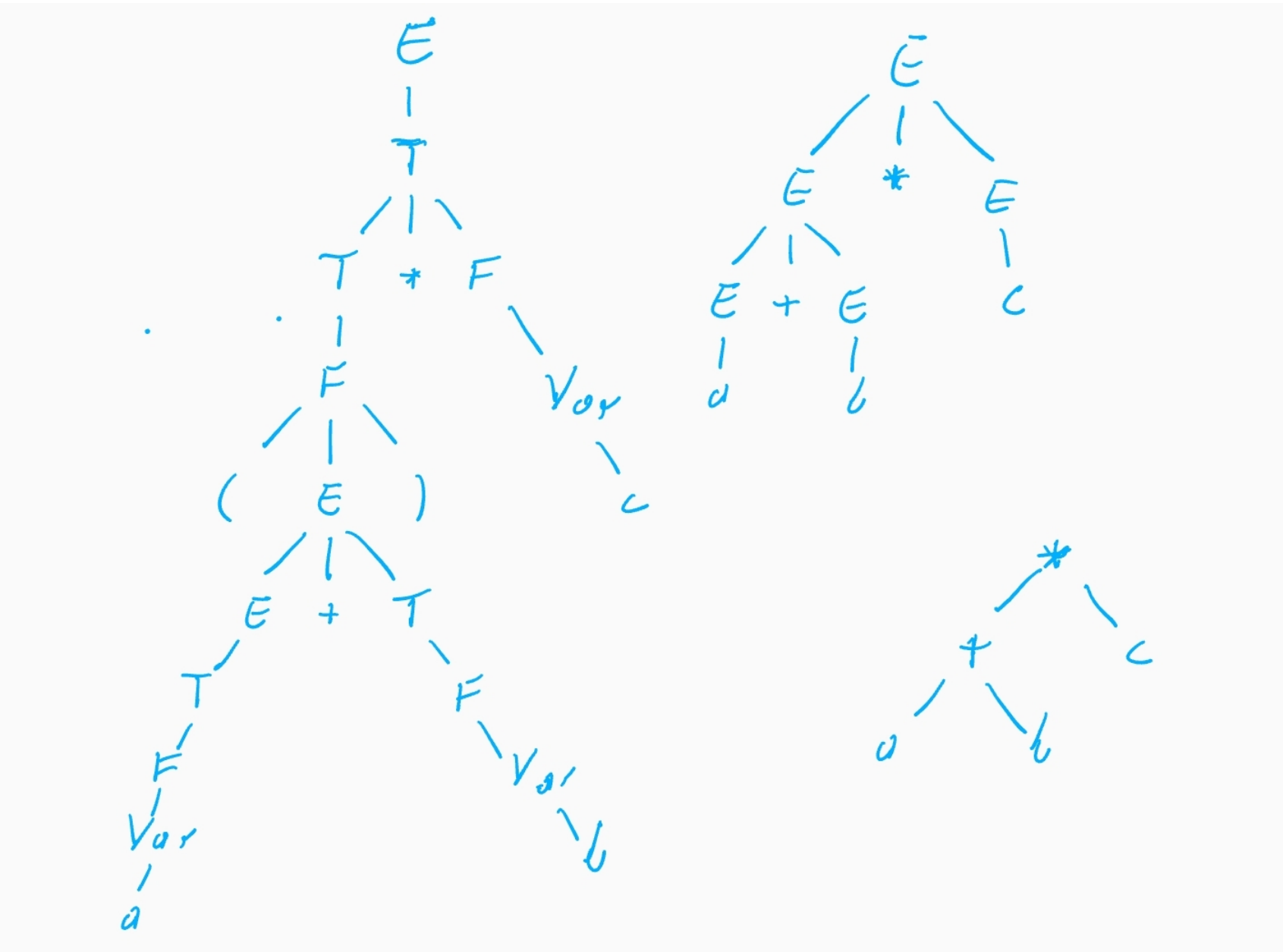
Ma :

- A volte è necessario complicare grammatica ed espressioni per definire la giusta interpretazione

  (a + b) * c

  - parentesi ( ) necessarie per definire la precedenza
  - diventano inutili una volta costruito l'albero sintattico dell'espressione.
- Gli alberi di derivazione possono essere ridondanti, contenere informazione inutile per l'interpretazione dell'espressione:

Esempio



Abstract syntax tree

- Abstract syntax tree: albero sintattico da cui sono stati eliminati
  - nodi ridondanti ai fini dell'analisi,
- rappresentazione più compatta, contiene solo le informazioni utili,
  - permette una computazione più pratica, efficiente
  - rappresentazione usata nei compilatori
- il significato di una stringa di caratteri è meglio evidenziato dal suo abstract tree che dall'espressione stessa

Sintassi astratta

- Sintassi che genera gli abstract syntax tree.
  Esempio, per le espressioni aritmetiche:

  E $\rightarrow$ E + E | E - E | E * E | Var

- sintassi minimale, mette in luce la parte importante della sintassi concreta
- sintassi intrinsecamente ambigua,
  - prevede dei meccanismi esterni alla grammatica per risolvere le ambiguità

Classi di grammatiche

Oltre alle grammatiche libere dal contesto esistono anche:

Grammatiche, gerarchia di Chomsky:

- a struttura di frase $v \to w$
- dipendenti da contesto $uAv \to uwv$
- libere da contesto $A \to w$
- regolari: lineari sinistre, lineari destre $A \to aB$ o $A \to a$

differenze:

- diverse grado di libertà nel definire le regole
- con grammatiche più generali:
  - una classe più ampia di linguaggi definibili
  - più complesso decidere se una parola appartiene al linguaggio, costruire albero di derivazione che genera quella parola

Classe di grammatiche usate negli analizzatori lessicali

Grammatiche regolari

- le classi di lessami (identificatori, letterali) hanno una struttura semplice
- grammatiche poco espressive sono sufficienti

Classe di grammatiche usate negli analizzatori sintattici

Grammatiche libere dal contesto

sono un compromesso ottimale tra espressività e complessità

- ampio insieme di linguaggi definibili
- nei casi d'uso, riconoscimento in tempo lineare sulla lunghezza della stringa
  - linguaggi riconoscibile mediante un automa a pila deterministico
  - un sottinsieme dei linguaggi liberi da contesto
- solo una complessità lineare accettabile per un compilatore

Vincoli sintattici contestuali:

Tuttavia con grammatiche libere dal contesto non posso eliminare dall'insieme di programmi riconosciuti (accettati)

programmi che non rispettano alcuni vincoli sintattici (contestuali) come:

- identificatori dichiarati prima dell'uso
- ugual numero di parametri attuali e formali
  - controllo non possibile perché il linguaggio $fa^n b^* fa^n b^* fa^n$ non è libero da contesto
- non modificare variabile di controllo ciclo "for"
- rispettati i tipi nelle assegnazioni, espressioni

Soluzione (dei compilatori):

- usare grammatiche libere come modo efficiente per costruire l'albero sintattico
- sull'albero sintattico si effettua un'ulteriore ricerca di errori l'analisi semantica

L' analisi semantica chiamata anche semantica statica

- controlli sul codice eseguibili a tempo di compilazione
- in contrapposizione alla semantica dinamica: controlli sul comportamento del programma eseguiti durante l'esecuzione

Semantica

- termine usato un po' impropriamente nelle espressioni:
  semantica statica, semantica dinamica
- la semantica associa a programma il suo significato,
  descrive il suo comportamento durante l'esecuzione
  concetto diverso da un semplice controllo degli errori
- semantica definita quasi sempre informalmente in linguaggio naturale
  un approccio formale è possibile:
  - semantica operazionale strutturata : programma descritto da un sistema di regole di riscrittura
    - un insieme di regole descrive il risultato della valutazione di un qualsiasi programma
  - semantica denotazionale: descrivo il comportamento del programma con
    strutture matematiche (funzioni)

Front end del compilatore

Similmente ai linguaggi naturali in cui:

- si descrivono l'insieme di parole valide, il dizionario, divise in categorie (articoli, nomi, verbi, . . .)
- regole sintattiche per costruire frasi da parole

nei linguaggi formali, nei compilatori:

- descrizione delle parti elementari lessemi , analisi lessicale
- descrizione della struttura generale, a partire da lessemi, analisi sintattica

Questa separazione rende più efficiente l'analisi dei programmi

Analisi lessicale (scanner, lexer)

Nella stringa di caratteri riconosco i lessemi, e per ogni lessema costituisco un token

token: (categoria sintattica, valore-attributo)

Esempio, data la stringa

```
x1 = a + b * 27;
```

viene generata la sequenza di token

```
[(identifier, 'x1'), (operator, =), (identifier, 'a'), (operat
(identifier, 'b'), (operator, *), (literal, 27), (separator, ;
```

La sequenza di token generati viene passata all'analizzatore sintattico (parser)

Analisi lessicale

Bisogna definire per ogni classe di lessemi:

- identificatori
- letterali
- parole chiave (ogni parola chiave ha una sua categoria sintattica)
- separatori
- . . .

la corrispondente sintassi
ossia quali stringhe di caratteri possono essere: un identificatore, un letterale . . .

Come esprimere la sintassi di una classe di lessemi: espressione regolare

Linguaggi e operazioni su linguaggi

Sia $\mathcal{A}$ un alfabeto, un insieme di simboli

Un linguaggio con alfabeto $\mathcal{A}$, è definito come

- un insieme di stringhe di elementi $\mathcal{A}$ (parole su $\mathcal{A}$)
  - {ab, abb, bbab} linguaggio sull'alfabeto {a, b}

Sui linguaggi posso definire le operazioni di:

- unione: $X \cup Y$
  - $\{ab, ba\} \cup \{a, b\} = \{a, b, ab, ba\}$
- concatenazione $X \cdot Y = \{ s\ t \mid s \in L , t \in M \}$ dove $s\ t$ indica la concatenazione della stringa s con la stringa t
  - $\{ab, ba\} \cdot \{a, b\} = \{aba, baa, abb, bab\}$
- chiusura di Kleene $X^* = \{ s_1\ s_2 \ldots\ s_n \mid \forall\ i\ .\ s_i \in X\}$
  - $\{ab, ba\}^* = \{\varepsilon, ab, ba, abab, abba, baab, baba, ababab, \ldots\}$

Espressioni regolari

A partire da queste tre operazioni (e da un insieme di costanti) costruisco

- l'insieme delle espressioni regolari
  chiama anche algebra di Kleene

Espressioni regolari permettono una rappresentazione sintetica dei linguaggi

Espressioni regolari, algebra di Kleene

Espressioni (algebriche), L, M, N, . . . costruite a partire da

- un insieme di costanti:
  - i simboli di un alfabeto $\mathcal{A}$,
  - dal simbolo ϵ rappresentante la stringa, parola, vuota.
- l'insieme delle operazioni sui linguaggi:
  - concatenazione: L M
  - unione: L | M
  - chiusura di Kleene: L*

Sintassi espressioni regolari

Oltre alle operazioni base, nelle espressioni regolari sono presenti

- parentesi tonde, ( ), per determinare l'ordine di applicazione,

Convenzioni, regole per evitare di dover inserire troppe parentesi:

- un ordine di precedenza tra gli operatori
  in ordine decrescente:

  `*, · , |`

  `a|bc* = a |  (b (c*))`

- visto che concatenazione e unione sono associative
  non è necessario specificare se associano a sinistra o a destra

  `L(MN) = (LM)N  = LMN`

Da espressione regolari a linguaggi, esempi:

Ogni espressione regolare rappresenta un linguaggio

- a|b* = a|(b*) ->

Da espressione regolari a linguaggi, esempi:

Ogni espressione regolare rappresenta un linguaggio

- a|b* = a|(b*) -> {“a”, ε, “b”, “bb”, “bbb”, . . . }

Da espressione regolari a linguaggi, esempi:

Ogni espressione regolare rappresenta un linguaggio

- a|b* = a|(b*) -> {“a”, ε, “b”, “bb”, “bbb”, . . . }
- (a|b)* ->

Da espressione regolari a linguaggi, esempi:

Ogni espressione regolare rappresenta un linguaggio

- a|b* = a|(b*) -> {“a”, ε, “b”, “bb”, “bbb”, . . . }
- (a|b)* -> { ε, “a”, “b”, “aa”, “ab”, “ba”, “bb”, “aaa”, . . . }

Da espressione regolari a linguaggi, esempi:

Ogni espressione regolare rappresenta un linguaggio

- a|b* = a|(b*) -> {“a”, ε, “b”, “bb”, “bbb”, . . . }
- (a|b)* -> { ε, “a”, “b”, “aa”, “ab”, “ba”, “bb”, “aaa”, . . . }
- a|bc* = a|(b(c*)) ->

Da espressione regolari a linguaggi, esempi:

Ogni espressione regolare rappresenta un linguaggio

- a|b* = a|(b*) -> {“a”, ε, “b”, “bb”, “bbb”, . . . }
- (a|b)* -> { ε, “a”, “b”, “aa”, “ab”, “ba”, “bb”, “aaa”, . . . }
- a|bc* = a|(b(c*)) -> {“a”, “b”, “bc”, “bcc”, “bccc”, . . . }

Da espressione regolari a linguaggi, esempi:

Ogni espressione regolare rappresenta un linguaggio

- a|b* = a|(b*) -> {“a”, ε, “b”, “bb”, “bbb”, . . . }
- (a|b)* -> { ε, “a”, “b”, “aa”, “ab”, “ba”, “bb”, “aaa”, . . . }
- a|bc* = a|(b(c*)) -> {“a”, “b”, “bc”, “bcc”, “bccc”, . . . }
- ab*(c|ε) ->

Da espressione regolari a linguaggi, esempi:

Ogni espressione regolare rappresenta un linguaggio

- a|b* = a|(b*) -> {“a”, ε, “b”, “bb”, “bbb”, . . . }
- (a|b)* -> { ε, “a”, “b”, “aa”, “ab”, “ba”, “bb”, “aaa”, . . . }
- a|bc* = a|(b(c*)) -> {“a”, “b”, “bc”, “bcc”, “bccc”, . . . }
- ab*(c|ε) -> {“a”, “ac”, “ab”, “abc”, “abb”, “abbc”, . . . }

Da espressione regolari a linguaggi, esempi:

Ogni espressione regolare rappresenta un linguaggio

- a|b* = a|(b*) -> {“a”, ε, “b”, “bb”, “bbb”, . . . }
- (a|b)* -> { ε, “a”, “b”, “aa”, “ab”, “ba”, “bb”, “aaa”, . . . }
- a|bc* = a|(b(c*)) -> {“a”, “b”, “bc”, “bcc”, “bccc”, . . . }
- ab*(c|ε) -> {“a”, “ac”, “ab”, “abc”, “abb”, “abbc”, . . . }
- (0|(1(01*0)*1))* =

Da espressione regolari a linguaggi, esempi:

Ogni espressione regolare rappresenta un linguaggio

- a|b* = a|(b*) -> {“a”, ε, “b”, “bb”, “bbb”, . . . }
- (a|b)* -> { ε, “a”, “b”, “aa”, “ab”, “ba”, “bb”, “aaa”, . . . }
- a|bc* = a|(b(c*)) -> {“a”, “b”, “bc”, “bcc”, “bccc”, . . . }
- ab*(c|ε) -> {“a”, “ac”, “ab”, “abc”, “abb”, “abbc”, . . . }
- (0|(1(01*0)*1))* = { ε, “0”, “00”, “11”, “000”, “011”, “110”, “0000”, “0011”, “0110”, “1001”, “1100”, “1111”, “00000”, . . . }

  tutti i numeri binari multipli di 3.

Da espressioni regolari a linguaggi

Definizione formale:

- $\mathcal{L}(\ \varepsilon\ ) = \{\ \varepsilon\ \}$
- $\mathcal{L}(\text{a}) = \{\ \text{“a”}\ \}$
- $\mathcal{L}\ (\text{L} \mid \text{M}) = \mathcal{L}(\text{L}) \cup \mathcal{L}\ (\text{M})$
- $\mathcal{L}\ (\text{L M}) = \mathcal{L}(\text{L}) \cdot \mathcal{L}\ (\text{M})$
- $\mathcal{L}\ (\text{L}^*) = (\mathcal{L}(\text{L}))^*$

Uso espressioni regolari

Espressioni regolari usate:

- negli analizzatori sintaticci
- in applicativi per manipolare stringhe,
  - text-editor
  - comandi di sistemi operativi
  - ...
- in funzioni di libreria, per manipolare stringhe, di molti linguaggi di programmazione

In questi ambiti, vengono introdotte operazioni extra,
oltre alle 3 canoniche,
in maniera non uniforme

Estensioni delle espressioni regolari

Per poter scrivere espressioni regolari più compatte, sono introdotte operazioni extra:

- chiusura positiva: $L^+ = L\ L^*$
- zero o un istanza: $L? = \varepsilon \mid L$
- n concatenazioni di parole in L: $L\{n\} = L\ L \ldots L$
- uno tra: [acdz] = a | c | d | z
- range: [a-z] = a | b | c | . . . | z
- complemento: [^a-z] - tutti i caratteri meno le lettere minuscole.

Espressioni più compatte ma stessa classe di linguaggi definibili

Esistono molte altre estensioni

Definizione tramite equazioni

Usata in alcune applicativi, librerie.

Permette una scrittura più chiara:

- `digit  := [0-9]`
- `sign  := [\-\+]?`
- `simple := sign{digit}+`
- `fract := (sign{digit})? . {digit}+`
- `exp := ({simple} | {fract}) e  {simple}`
- `num := {simple} | {fract} | {exp}`

al posto di

- `num :=  [0-9]^+^ | [0-9]^+^.[0-9]^+^ | .[0-9]^+ | [0-9]+.[0-9]+ e ( ε  | \+ | \- ) [0-9]+ | .[0-9]+ e ( ε | \+ | \- ) [0-9]+`

Metacaratteri

Nelle espressioni regolari devo distinguere tra:

caratteri simboli che rappresentano se stessi

metacaratteri simboli che rappresentano altro:

- operazioni: `|` `*` `+`
- parentesi: `(` `)`
- insiemi di caratteri: `[` `]`
- ...
- il simbolo di escape: `\` mi permette di:
  - usare un metacarattere come carattere:
    - `\*` rappresenta il carattere `*` e non la chiusura di Kleene
  - usare un carattere per rappresentare altro:
    - `\n` rappresenta il newline

Risultati teorici

Teorema di equivalenza

Linguaggi regolari posso essere descritti in molti diversi:

- espressioni regolari
- grammatiche regolari (sinistre - destre)
- automi finiti non deterministici, NFA non determistic finite automata.
- automi finiti deterministici (macchine a stati finiti) DFA determistic finite automata.

Teorema di minimalità

Esiste l´automa deterministico minimo (minor numero di stati)

Utilità pratica

L'analizzatore lessicale si ottiene

- a partire dalle espressioni regolari (specifica)
- costruendo i DFA minimi corrispondenti
- simulando la loro esecuzione contemporanea all'interno del lexer

Nuovo formalismo DFA, esempio

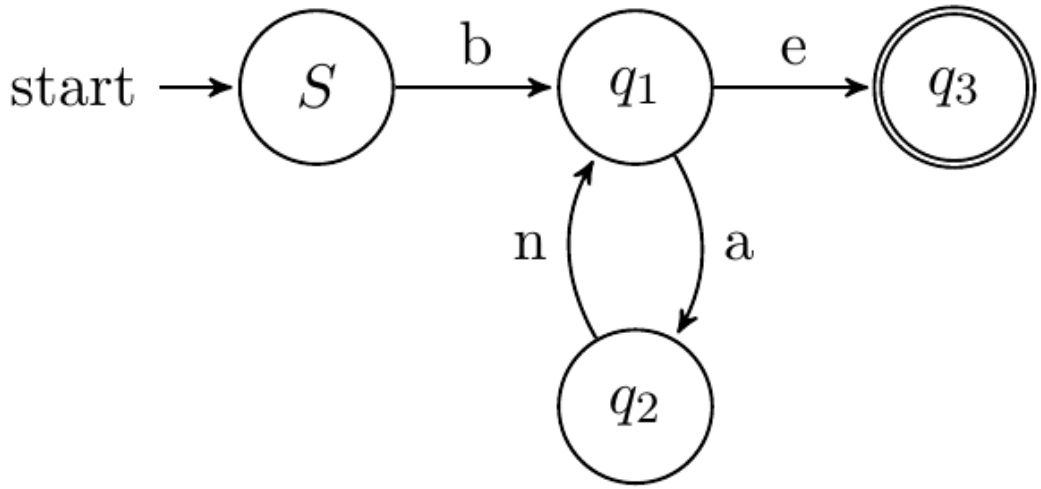


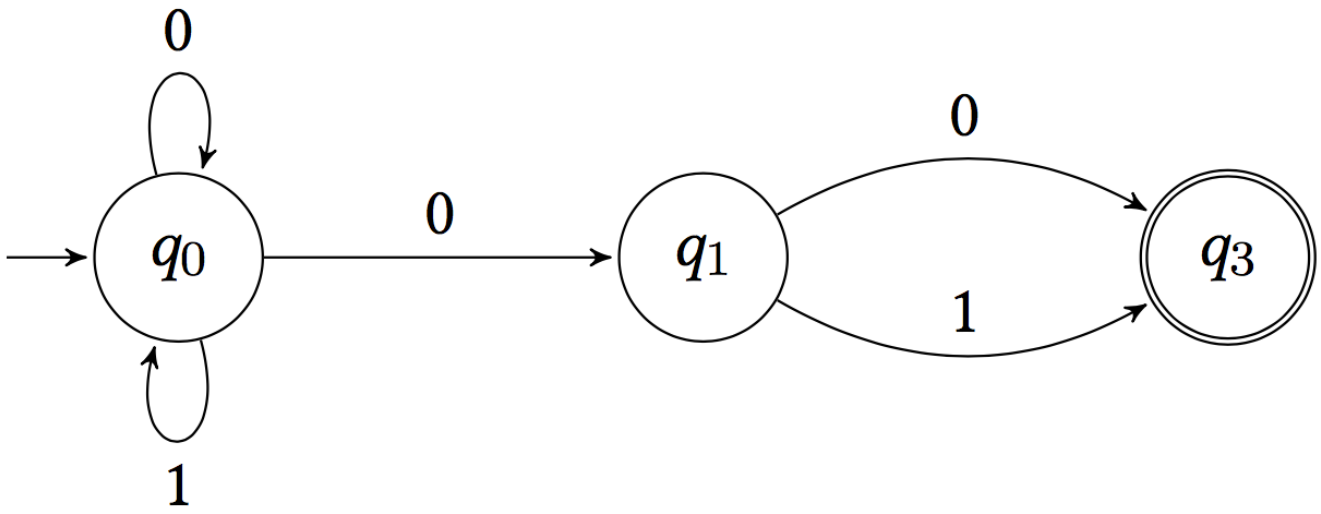
Figure 1: Finite State Automaton, accepting the pattern $b(an)+e$

Riconoscitore per b(an)*e

Se anche q_2 stato finale il linguaggio generato diventa

- b(an)*(e|a)
- b(an)*e | b(an)*a

Alternativa NFA, esempio



Riconoscitore per (0|1)*0(0|1)

Nei NFA (non-deterministic finite automata) più possibili alternative

- più archi in uscita con la stessa etichetta,
- stringa accettate se esiste una sequenza di transazioni che la riconosce.

Applicazioni dei risultati teorici

Per costruire un riconoscitore per un'espressione regolare

- Dall'espressione regolare costruisco:
  - NFA equivalente, da questi il
  - DFA equivalente, da questi il
  - l'automa minimo, (DFA minino),

  tutte costruzioni effettive.
- Dall'automa minimo costruisco un programma per decidere se una parola appartiene a un'espressione regolare.
- Programma simula il comportamento dell'automa minimo contiene una tabella che descrive le transazioni dell'automa minimo, e ne simula il comportamento.

Scanner, lexer

Lo scanner deve risolvere un problema più complesso del semplice riconoscimento di una singola espressione regolare.

- Dati
  - un insieme di espressioni regolari,
    classi di lessemi (es. identificatori, numeri, operazioni, . . .),
  - una stringa di ingresso
- lo scanner deve dividere la stringa d'ingresso in lessemi, ciascuno riconosciuto da un'espressione regolare.

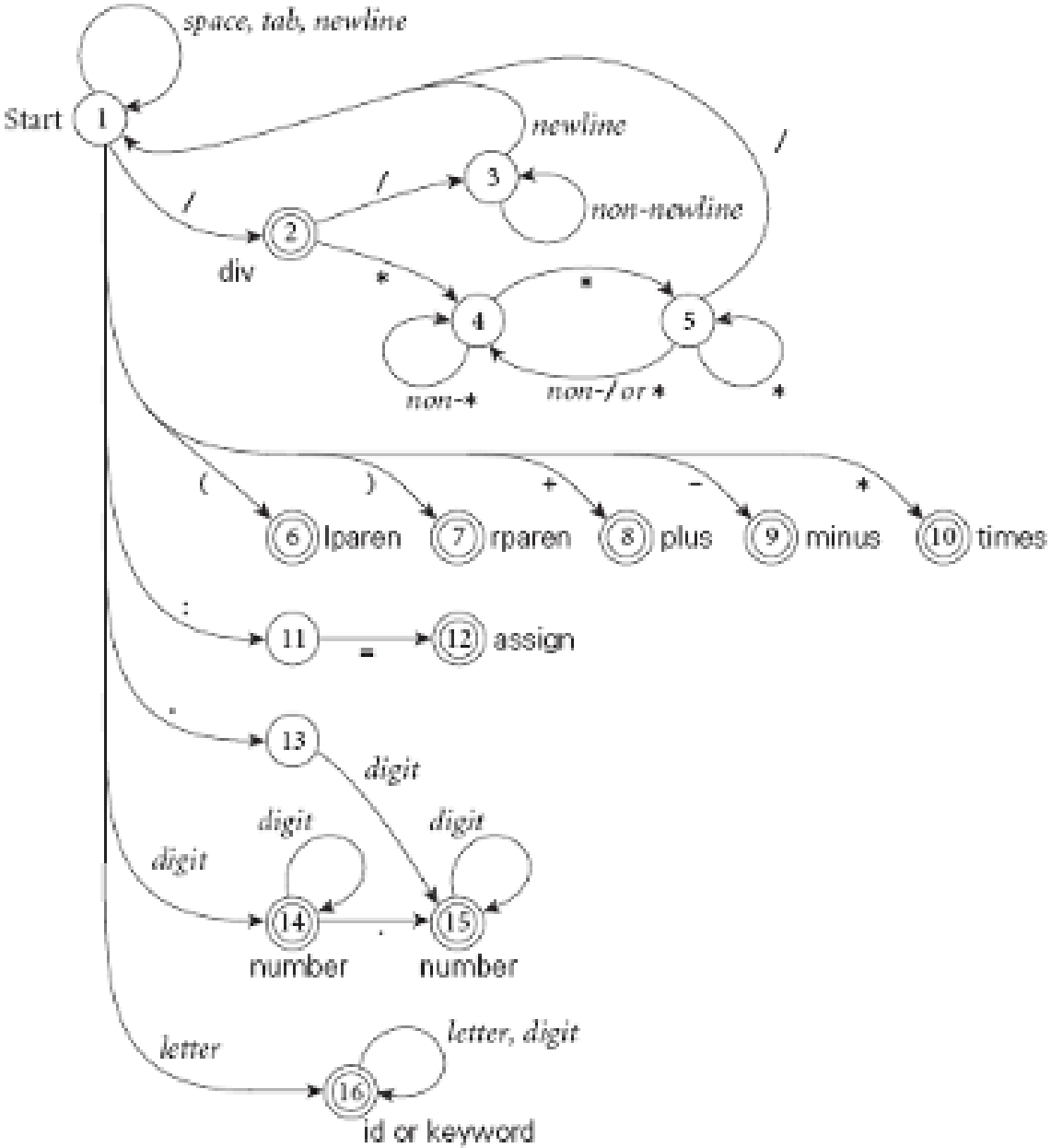
Problemi:

- quando termina un lessema, soluzione standard: la sequenza più lunga che appartiene a qualche espressione regolare,
  - la stringa '3.14e+sa' divisa in '3.14' 'e' '+' 'sa'
  - la stringa '3.14e+5a' divisa in '3.14e+5' 'a'
    per deciderlo posso essere necessari più simboli di lookahead;
- cosa fare se un lessema appartiene a più classi.

Come costruire uno scanner

- Costruisco un automa per ogni espressione regolare,
- sulla stringa di ingresso,
  - simulo il funzionamento parallelo degli automi,
    lo posso fare con un singolo automa
  - riconosco un lessema quando nessun automa può continuare.

Automa per lessemi di espressioni aritmetiche



Generatori di scanner (analizzatori lessicali)

La costruzione degli scanner può essere automatizzata

Classe di programmi che:

- dato
  - un insieme di espressioni regolari
  - delle corrispondenti azioni da compiere (codice da eseguire)
- genera un programma che:
  - data una stringa,
  - riconosce i lessemi sulla stringa
  - su ogni lessema riconosciuto esegue l'azione corrispondente tipicamente costruire un token, ma può fare altro

(F)LEX

Classico, diffuso generatore di scanner in C, per Unix (Linux)

Prende in input un file di testo con struttura

```
definizioni   (opzionale)
%%
regole
%%
funzioni ausiliarie  (opzionale)
```

la parte regole, la più importante, serie di regole nella forma

```
espressione-regolare    azione
```

- espressioni-regolare sono quelle di unix (grep, emacs), ricca sintassi
- azione istruzione C, istruzioni multiple appaiono tra { } eseguita quando viene riconosciuta la corrispondente espressione (esistono strumenti equivalenti per gli altri ling. progr.)

Esempio

```
%%
aa      printf("2a")
bb+     printf("many-b")
c       printf("cc")
```

genera un programma che:

- modifica coppie di “a”
- modifica sequenze di “b”, lunghe più di due caratteri
- raddoppia le “c”

i caratteri non riconosciuti da nessuna espressione regolare vengono stampati in uscita (restano inalterati)

Regole

Nel codice delle regole posso usare le variabili:

- `yytext`: stringa (array) contenente il lessema riconosciuto, puntatore al primo carattere
- `yyleng`: lunghezza del lessema
- `yyval`: usata per passare parametri al parser

Definizioni

`definizioni`

contiene la definizione di alcune espressioni regolari

nella forma

```
nome    espressione-regolare
```

esempio

```
letter [a-zA-Z]
digit  [0-9]
number {digit}+
```

notare le parentesi graffe in {digit}+

i nomi definiti possono essere usati nelle regole

Sintassi delle espressioni regolari

- metacaratteri: `* | ( ) + ? [ ] - ^ . $ { } / \ " % < >`
- `{ide}` : identificatore di espressione regolare fa inserito tra { }
- `e{n,m}`: con n, m naturali: da n a m ripetizioni di e,
  anche `e{n,}`, `e{,n}`, `e{n}`
- `[^abd]`: tutti i caratteri esclusi a b d
- `\n`: newline, `\s`: spazio generico, `\t`: tab
- `\*`: il carattere `*`, `\` trasforma un metacarattere in un carattere standard e viceversa
- `"a+b"`: la sequenza di caratteri `a+b` (+ non è più un metacarattere)
- `.`: tutti i caratteri meno il newline
- `^`: inizio riga
- `$`: fine riga

Funzioni ausiliarie

- Nella parte “funzioni ausiliarie” si può definire codice C da usare nelle regole
- Il codice C da inserire in testa al programma generato,
  - viene inserito nella parte ‘definizioni’
  - tra le parentesi `%{ }%`

Esempio

```
%{
  int val = 0;
%}
separatore [ \t\n]

%%
0    {val = 2*val;}
1    {val = 2*val+1;}
{separatore}    {printf("%d",val); val=0;}
```

sostituisce sequenze rappresentanti numeri binari con il loro valore, scritto in decimale

Uso standard

```
cifra              [0-9]
lettera            [a-zA-Z]
identificatore     {lettera}({cifra}|{lettera})*
%%
{identificatore}      printf("(IDE,%s)", yytext);
```

sostituisce il lessema con un token.

Funzionamento

- Si considerano tutte le espressioni regolari e si seleziona quella con match più lungo, la parte lookahead conta nella misura
- a parità di lunghezza, conta l'ordine delle regole
- vengono impostate yytext, yyleng e eseguita l'azione
- nessun matching: regola di default: copio carattere input in output

Codice generato

- anche insiemi semplici di espressioni regolari generano centinaia di righe di codice C
- il codice contiene all'interno una tabella descrittore di una DFA minimale
- il programma simula le operazioni del DFA

Esempio: cifrario di Cesare

```
%%
[a-z]    { char ch = yytext[0];
           ch += 3;
           if (ch > 'z') ch -= ('z' - 'a' + 1);
           printf ("%c", ch);
         }

[A-Z]    { char ch = yytext[0];
           ch += 3;
           if (ch > 'Z') ch -= ('Z' - 'A' + 1);
           printf ("%c", ch);
         }
%%
```

Esempio, conta caratteri

```
%{
int charcount=0, linecount=0;
%}
%%
.   charcount++;
\n {linecount++; charcount++;}
%%
void yyerror(const char *str)
   { fprintf(stderr,"errore: %s\n",str);}
int  yywrap() {return 1;}  /* funzioni ausiliarie */
void main() {
  yylex();

  printf("There were %d characters in %d lines\n",
  charcount,linecount);
}
```

Funzioni base

Devono essere definite le funzioni:

- `yyerror(const char *str)`: viene chiamata in condizioni di errore, tipicamente stampa un messaggio di errore usando la stringa argomento.
- `yywrap()`: viene chiamata a fine file di input, tipicamente restituisce `0` o `1`.
- `main()`: con opportune opzioni, possono essere create versioni di default.

Uso

```
> flex sorgente.l
```

genera un programma C `lex.yy.c`, compilabile con il comando

```
> gcc lex.yy.c -ll
```

in `lex.yy.c` viene creata una funzione `yylex()`

- chiamata dal programma “parser”
- legge un lessema ed esegue l’azione corrispondente

opzione ‘-ll’ necessaria per creare un programma stand-alone

- collegare alcune librerie
- con le definizioni `main`, `yywrap` `yyerror`
- non necessaria se inserisco nel file lex le relative definizioni

Utilizzabile per automatizzare del text editing.

Analisi sintattica (Parsing) - Analizzatore sintattico (Parser)

A partire da

- una grammatica libera da contesto
- una stringa di token

costruisco

- l'albero di derivazione della stringa, a partire dal simbolo iniziale

Automi a pila - la teoria

Le grammatiche libere possono essere riconosciute da automi a pila non deterministici

- Automi con un uso limitato di memoria:
  - insieme finito di stati
  - una pila, in cui inserire elementi finiti
  - passo di computazione, in base a:
    - stato
    - simbolo in testa alla pila
    - simbolo in input
  - si determina:
    - nuovo stato
    - sequenze di simboli da rimuovere e inserire in pila
    - se consumare o meno l'input

Parola accettata se, a fine scansione, si raggiunge una configurazione di accettazione:

- pila vuota
- stato finale

Deterministici – Non deterministici

Per gli automi a pila non vale l'equivalenza tra:

- deterministici (ad ogni passo una sola azione possibile)
- non-deterministici (più alternative possibili)

Per le grammatiche libere sono necessari, in generale, automi non-deterministici.

Complessità del riconoscimento

- Un automa a pila non deterministico, simulato tramite backtracking, porta a complessità esponenziali.
- Esistono due algoritmi [Earley, Cocke-Younger-Kasami] capaci di riconoscere qualsiasi linguaggio libero in tempo $O(n^3)$.
- Un automa a pila deterministico risolve il problema in tempo lineare.

In pratica:

- complessità $O(n^3)$ non accettabile, compilatore troppo lento
- ci si limita ai linguaggi riconoscibili da automi a pila deterministici
- classe sufficientemente ricca da contenere quasi tutti i linguaggi di programmazione (C++ è un eccezione `x * y ;`)

Due tipi di automi a pila: LL e LR, due metodi di riconoscimento

Automi LL(n)

Costruiscono l'albero di derivazione in modo top-down:

- a partire dal simbolo iniziale
- esaminando al più n simboli della stringa non consumata (lookahead)
- si determina la prossima regola (espansione) da applicare

Esempio di parsing, data la grammatica:

$$
\begin{aligned}
S &\rightarrow aAB \\
A &\rightarrow C \mid D \\
B &\rightarrow b \\
C &\rightarrow c \mid \epsilon \\
D &\rightarrow d
\end{aligned}
$$

la stringa adb viene riconosciuta con i seguenti passi:

Output	Pila	Input
Start	$S\$$	$adb\$$
$S \rightarrow aAB$	$aAB\$$	$adb\$$
	$AB\$$	$db\$$
$A \rightarrow D$	$DB\$$	$db\$$
$D \rightarrow d$	$dB\$$	$db\$$
	$B\$$	$b\$$
$B \rightarrow b$	$b\$$	$b\$$
	$\$$	$\$$

OK!

Seconda derivazione

$$
\begin{array}{lcl}
S & \to & aAB \\
A & \to & C \mid D \\
B & \to & b \\
C & \to & c \mid \epsilon \\
D & \to & d
\end{array}
$$

la stringa abb viene rifiutata con i seguenti passi:

Output	Pila	Input
Start	$S\$$	$abb\$$
$S \to aAB$	$aAB\$$	$abb\$$
	$AB\$$	$bb\$$
$A \to C$	$CB\$$	$bb\$$
$C \to \epsilon$	$B\$$	$bb\$$
$B \to b$	$b\$$	$bb\$$
	$\$$	$b\$$

Errore!

Più nel dettaglio

Il parsing è guidato da una tabella che, in base a:

- al simbolo in testa alla pila
- ai primi n simboli di input non ancora consumati (normalmente n = 1),

determina la prossima azione da svolgere, tra queste possibilità:

- applicare una regola di riscrittura, espandendo la pila
- consumare un simbolo in input e in testa alla pila (se coincidono)
- generare un segnale di errore (stringa rifiutata)
- accettare la stringa (quando input e pila sono vuoti)

Analizzatori LL

- La teoria verrà presentata nel corso di Linguaggi e Compilatori (laurea magistrale).
- È relativamente semplice capire la teoria e costruire automi (anche a mano, partendo da semplici grammatiche).
- La costruzione prevede:
  - dei passaggi di riformulazione di una grammatica per ottenere una equivalente (che determina lo stesso linguaggio)
  - dalla nuova grammatica, un algoritmo determina:
    - se è LL(1)
    - la tabella delle transizioni (descrizione dell'automa)

Meno generali dell'altra classe di automi LR(n), quelli effettivamente usati nei tool costruttori di parser.

Significato del nome LL(n)

- Esamina la stringa from Left to right.
- Costruisce la derivazione Leftmost.
- Usa n simboli di lookahead.

Una derivazione è sinistra (leftmost) se ad ogni passo espando sempre il non terminale più a sinistra.

- $S \rightarrow aAB \rightarrow aDB \rightarrow adB \rightarrow adb$

Una derivazione è destra (rightmost) se ad ogni passo espando sempre il non terminale più a destra.

- $S \rightarrow aAB \rightarrow aAb \rightarrow aDb \rightarrow adb$

Gli automi LL(n) generano sempre la derivazione sinistra

Analizzatori LR

Approccio bottom-up:

- a partire dalla stringa di input,
- applico una serie di contrazioni, (regole al contrario)
- fino a contrarre tutto l'input nel simbolo iniziale della grammatica.

Esempio - Grammatica non LL

$$
\begin{aligned}
E &\rightarrow T \mid T + E \mid T - E \\
T &\rightarrow A \mid A * T \\
A &\rightarrow \mathbf{a} \mid \mathbf{b} \mid (E)
\end{aligned}
$$

	Pila	Input	Azione	Output
1	$\$$	$\mathbf{a} + \mathbf{b} * \mathbf{b}\$$	shift	
2	$\$\mathbf{a}$	$+ \mathbf{b} * \mathbf{b}\$$	reduce	$A \rightarrow \mathbf{a}$
3	$\$A$	$+ \mathbf{b} * \mathbf{b}\$$	reduce	$T \rightarrow A$
4	$\$T$	$+ \mathbf{b} * \mathbf{b}\$$	shift	
5	$\$T +$	$* \mathbf{b}\$$	shift	
6	$\$T + \mathbf{b}$	$* \mathbf{b}\$$	reduce	$A \rightarrow \mathbf{b}$
7	$\$T + A$	$* \mathbf{b}\$$	shift	
8	$\$T + A *$	$\mathbf{b}\$$	shift	
9	$\$T + A * \mathbf{b}$	$\$$	reduce	$A \rightarrow \mathbf{b}$
10	$\$T + A * A$	$\$$	reduce	$T \rightarrow A$
11	$\$T + A * T$	$\$$	reduce	$T \rightarrow A * T$
12	$\$T + T$	$\$$	reduce	$E \rightarrow T$
13	$\$T + E$	$\$$	reduce	$E \rightarrow T + E$
14	$\$E$	$\$$	stop	OK!

Più nel dettaglio

- Ad ogni passo si sceglie tra un azione di:
  - shift inserisco un token in input nella pila
  - reduce riduco la testa della pila applicando una riduzione al contrario
- Nella pila introduco una coppia <simbolo della grammatica, stato>

  l'azione da compiere viene decisa guardando:
  - la componente stato in testa alla pila (non serve esaminarla oltre)
  - n simboli di input, per l'automa LR(n)

Automi LR

esiste un algoritmo che a partire da:

- una grammatica libera L

mi permette di:

- stabilire se L è LR
- costruire l'automa a pila relativo
  - insieme degli strati
  - tabella delle transazioni

  come deve comportarsi Í'automa ad ogni passo.

In realtà esistono tre possibili costruzioni.

Varie costruzioni per automi LR

Le costruzioni differiscono per:

- complessità della costruzione
- numero degli stati dell'automa pila generato
  complessità dell'algoritmo
- ampiezza dei linguaggi riconoscibili

In ordine crescente per complessità e ampiezza di linguaggi riconosciuti:

- SLR(n)
- LALR(n)
- LR(n)

n parametro, indica il numero di caratteri lookahead, crescendo n si ambia l'insieme di linguaggi riconosciuti

Analizzatori LALR

Compromesso ideale tra numero di stati e varietà dei linguaggi riconosciuti

Costruzione piuttosto complessa: da spiegare e da implementare

Esempio di applicazione di risultati teorici:

- Donald Knuth: 1965, parser LR, (troppi stati per i computer dell'epoca)
- Frank DeRemer: SLR and LALR,
  (pratici perché con meno stati)

LALR usato dai programmi generatori di parser:
Yacc, Bison, Happy

Yacc (Yet Another Compiler-Compiler)

Generatore di parser tra i più diffusi:

Riceve in input una descrizione astratta del parser:

- descrizione di una grammatica libera
- un insieme di regole da applicare ad ogni riduzione

Restituisce in uscita:

- programma C che implementa il parser
  - l'input del programma sono token generati da uno scanner (f)lex
  - simula l'automa a pila LALR
  - calcola ricorsivamente un valore da associare a ogni simbolo inserito nella pila:
    - albero di derivazione
    - altri valori

Programmi equivalenti per costruire parser in altri linguaggi:
ML, Ada, Pascal, Java, Python, Ruby, Go, Haskell, Erlang

Struttura codice Yacc:

```
%{ prologo  %}
definizioni
%%
regole
%%
funzioni ausiliarie
```

Una produzione della forma

`nonterm` $\rightarrow$ `corpo_1 | · · · | corpo_k`

diventa in Yacc con le regole:

```
nonterm : corpo_1   {azione semantica_1 }
          ...
          | corpo_k {azione semantica_k }
        ;
```

Azione semantica

```
exp : num '*' fact  { Ccode }
```

`Ccode` Codice C che tipicamente

- a partire dai valori calcolati in precedenza per `num` e `fact`
- calcola il valore da associare ad `exp`

Meccanismo compatibile con il riconoscimento bottom-up

- descrivo cosa fare quando applico una riduzione
  - la pila contiene anche i valori associati ai simboli della parte destra
  - determino il valore da associare al nuovo non-terminale

Esempio Valutazione Espressioni Aritmetiche

Costruisco un programma che valuta

- una serie di espressioni aritmetiche
- divise su più righe di input

espressioni composte da:

- costanti numeriche: numeri positivi e negativi
- le quattro operazioni
- parentesi

valgono le usuali regole di precedenza tra operazioni

Prologo e definizioni

```
/* PROLOGO */
%{
#include "lex.yy.c"

void yyerror(const char *str){
    fprintf(stderr,"errore: %s\n",str);}
int yywrap() {return 1;}
int main() { yyparse();}
%}
/* DEFINIZIONI */
%token NUM

%left '-' '+'
%left '*' '/'
%left NEG     /* meno unario */
```

Esempio - Regole

```
%% /* REGOLE E AZIONI SEMANTICHE */
          /* si inizia con il simbolo iniziale */
input:    /* empty */
        | input line
;
line :  '\n'
        | exp '\n' { printf("The value is %d \n", $1); }
;
exp : NUM     { $$=$1;     }
    | exp '+' exp     { $$=$1+$3;     }
    | exp '-' exp     { $$=$1-$3;     }
    | exp '*' exp     { $$=$1*$3;     }
    | exp '/' exp     { $$=$1/$3;     }
    | '-' exp %prec NEG { $$ =-$2;     }
    | '(' exp ')'     { $$ = $2;     }
;
```

Esempio - Codice LEX associato

```
%{
#include <stdio.h>
#include "y.tab.h"
%}

%%
[ \t]    ; // ignore all whitespace
[0-9]+       {yylval = atoi(yytext); return NUM;}
\n               {return *yytext;}
"+"              {return *yytext;}
[\-\*\/\(\)]     {return *yytext;}
```

Definizione dei token

- l'insieme dei possibili token definiti nel file Yacc con la dichiarazione `%token NUM`
- singoli caratteri possono essere token
  - non necessario definirli
  - codificati con in loro codice ASCII
  - gli altri token codificati con intero `>257`
- token diventano i terminali delle grammatica libera in Yacc
- Yacc crea un tabella `y.tab.h`
  - contiene la lista dei token e la loro codifica come interi
  - lex la dichiarazione
    #include "y.tab.h"
    accede ai dati in questa tabella

Funzioni ausiliare

Nel file Yacc è necessario definire le funzioni

- `yyerror` procedura invocata in caso di errori nella sintassi dell'input
  - in input una stringa da usare nel messaggio di errore
- `yywrap` chiamata al termine del file di input
  - di default restituisce l'intero 0 o 1
  - può essere usata per gestire più file di input,
- `main` per creare un programma stand-alone

La compilazione YACC crea una funzione C

- `yyparser` che implementa il parser LALR

Integrazione tra lex e Yacc (flex e Bison)

- lex non crea un programma stand alone ma una funzione `yylex()` chiamata all'interno di `yyparser`
- `yylex()` restituisce un token ad ogni chiamata e, indirettamente, un valore
  - il token è il valore restituito esplicitamente dalla funzione:
    - intero, codifica la classe del lessema
  - i token diventano i terminali della grammatica usata in Yacc
  - il valore attraverso la variabile `yylval`, globale e condivisa

Differenze tra YACC e LALR

YACC produce codice C che implementa un automa LALR ma:

- non esiste uno stretto controllo che la grammatica sia LALR
  - grammatiche non LALR vengono accettate ma:
    - si costruisce un automa a pila dove, per alcuni casi, più scelte sono possibili (automa non-deterministico)
    - YACC genera codice che ne sceglie una, eseguendo solo quella
    - si possono fare scelte sbagliate
    - si possono rifiutare parole valide
- grammatiche ambigue possono essere usate:
  - attraverso la definizione di priorità si possono eliminare ambiguità
  - per automi non LALR: attraverso le priorità si indicano le scelte da fare

Differenze tra YACC e LALR

- il codice C, non solo riconosce la stringa ma la “valuta¨
  - valutazione bottom-up - come il riconoscimento,
  - risultati parziali inserite nella pila, `$$`, `$1`, `$2`, . . . fanno riferimento alla pila
  - la parte “azioni” delle regole specificano che valore associare una parola
    a partire dai valori delle sotto-parole.
  - attraverso le azioni posso costruire l’albero di derivazione ma anche altro

Secondo esempio: sintassi di comandi ad un termostato.

File LEX

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
%%
[0-9]+              { yylval=atoi(yytext); return NUMERO; }
riscaldamento       return TOKRISCALDAMENTO;
acceso|spento       { yylval=strcmp(yytext,"spento");
                      return STATO;}
obiettivo           return TOKOBIETTIVO;
temperatura         return TOKTEMP;
[ \t\n]+            /* ignora spazi bianchi e fine linea */;
%%
```

File Yacc Prologo e dichiarazioni

```
%{
#include <stdio.h>
#include <string.h>

void yyerror(const char *str)
{ fprintf(stderr,"errore: %s\n",str);}
int yywrap() {return 1;}
int main() { yyparse();}
%}

%token NUMERO TOKRISCALDAMENTO STATO TOKOBIETTIVO TOKTEMP

%%
```

File YACC: regole

```
comandi: /* vuoto */
        | comandi comando
        ;

comando: interruttore_riscaldamento
        |  imposta_obiettivo
        ;
interruttore_riscaldamento: TOKRISCALDAMENTO STATO
        { if($2)     printf("\t Riscaldamento acceso \n");
          else       printf("\t Riscaldamento spento \n"); }
        ;
imposta_obiettivo:  TOKOBIETTIVO TOKTEMP NUMERO
        { printf("\t Temperatura impostata a %d \n",$3); }
        ;
```

Modifica tipo YYSTYPE

Nel file C, le variabili `yylval`, `$$`, `$1`, `...` hanno tipo `YYSTYPE` - per default è un intero - posso modificare `YYSTYPE` con la dichiarazione

```
%union {
nodeType        *expr;
int             value;
}
```

definisce un tipo `union` associato alla variabile LEX (`yyval`)
e alle variabili YACC (`$$`, `$1`, `...`)

con le seguenti dichiarazioni, specifico a quali tipi dei componenti sono associati diversi terminali e non terminali.

```
%token <value> INTEGER, CHARCON;   /* terminali */
%type <expr>     expression;       /* non terminali */
```

dichiarazioni da inserire nella parte definizioni (prologo) del file YACC.

Creazione del codice

```
lex file.l
yacc -d file.y
cc lex.yy.c y.tab.c -o fileEseguibile
```

in alternativa:

```
flex file.l
bison -d file.y
gcc lex.yy.c y.tab.c -o fileEseguibile
```

in alternativa, inserisco:

```
#include "lex.yy.c"
```

nel prologo yacc, e uso il comando

```
cc y.tab.c -o fileEseguibile
```

- L'opzione `-d` forza la creazione del file `y.tab.h`.