

Linguaggi e compilatori

Corso di Laurea in Informatica

Mauro Leoncini

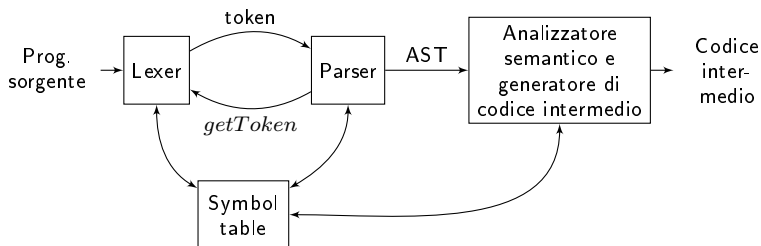
A.A. 2023/2024

1 Sintassi e grammatiche

- Linguaggi liberi
- Grammatiche formali
- Derivazioni
- Gerarchie di linguaggi
- La descrizione di linguaggi reali
- Alberi di derivazione (parse tree) e ambiguità
- Grammatiche di tipo 1 e di tipo 3

Ricordiamo ancora lo schema del front-end con il ruolo del parser

- Il compito del *parser* è di stabilire se una sequenza di token rappresenta un programma sintatticamente corretto e, se questo è il caso, di fornire una descrizione della sua struttura sintattica
- Come più volte osservato, il parser si interfaccia con il lexer, che agisce come suo sottoprogramma, e dall'altra con l'analizzatore semantico (con cui è di fatto "inter-allacciato")



Replichiamo la metodologia

- Un parallelo che illustra come abbiamo proceduto per la presentazione dell'analisi lessicale
- ... e come procederemo per descrivere la fase di *analisi sintattica*

Analisi lessicale

- Descrizione del lessico: linguaggi regolari **1**
- Descrizione formale: espressioni regolari 2. regex
- Strumenti automatici per il lexing: Lexer ^{3 espressione regolare -> AST (Abstract Syntax Tree)}
- Basi teorico/algoritmiche per Lexer: automi finiti

4. Automa non deterministico -> Automa deterministico

Analisi sintattica

- Descrizione della sintassi: linguaggi context-free
- Descrizione formale: grammatiche context-free
- Strumenti automatici per il parsing: Yacc/Bison
- Basi teorico/algoritmiche per Bison: algoritmi di parsing

Linguaggi e compilatori

1 Sintassi e grammatiche

- Linguaggi liberi
- Grammatiche formali
- Derivazioni
- Gerarchie di linguaggi
- La descrizione di linguaggi reali
- Alberi di derivazione (parse tree) e ambiguità
- Grammatiche di tipo 1 e di tipo 3

Limiti dei linguaggi regolari

● Riassunto : Linguaggi regolari OK (LESSICO)
Linguaggi regolari NO (SINTASSI)

Context free

- Consideriamo il seguente frammento scritto in C (o C++)
`while (a>0) {a=a-1}`
- Consideriamo ora il seguente “frammento”
`while (a>0 {a=a-1}`
- Naturalmente ci aspettiamo che il primo “compili” perfettamente e che, su input il secondo, il compilatore segnali invece una condizione di errore
- Di quale tipo di errore si tratta?
- L'errore consiste ovviamente nel fatto che la parentesi aperta che delimita la condizione logica non è stata chiusa
- Si tratta, in fondo, di una sola parentesi mancante

Limiti dei linguaggi regolari

- Il numero giusto di parentesi, non è esprimibile con un automa senza memoria => Non è un'espressione regolare.

- Consideriamo invece questo terzo frammento (che, per inciso, implementa il calcolo di un polinomio con il metodo di Horner)

$((a * x + b) * x + c) * x + d$

Calcola il polinomio
facendo meno
prodotti possibili

- In questo caso la frase è corretta e ciò dipende dal fatto che tutte e tre le parentesi aperte sono correttamente bilanciate
- Questo è un requisito sintattico e, se pensiamo alla generalizzazione dell'esempio (con espressioni arbitrarie), possiamo arrivare alla conclusione che esso non può essere fissato in un linguaggio regolare
- Tramite l'equivalenza con gli automi finiti, sappiamo infatti che, in generale, un linguaggio regolare non può includere frasi in cui una porzione di frase è condizionata da una precedente porzione di frase
- Non può essere regolare, ad esempio, proprio il linguaggio in cui tutte le frasi che includono n parentesi aperte includono anche n parentesi chiuse

Linguaggi liberi e linguaggi dipendenti dal contesto

- Per descrivere la sintassi di un linguaggio di programmazione abbiamo dunque bisogno di linguaggi “più ricchi” di struttura
- Oltre alla *capacità espressiva assoluta* (cioè il fatto che un linguaggio possa o non possa catturare determinate strutture linguistiche) in generale c'è anche una ragione di convenienza per usare strumenti più potenti
- Anche quando ciò fosse possibile, usare le espressioni regolari del descrivere aspetti sintattici del linguaggio sarebbe estremamente “faticoso” (e noioso)
- Esempio: supponiamo che un certo linguaggio consenta espressioni aritmetiche con i soli operatori binari $+$ e \times e in cui il numero di parentesi aperte (e non ancora chiuse) sia al più due
- Questo linguaggio è regolare: si provi a fornire un'espressione regolare per descriverlo (o almeno un AFD)

Linguaggi liberi e linguaggi dipendenti dal contesto

- La sintassi dei linguaggi di programmazione è descritta mediante i cosiddetti *linguaggi liberi da contesto* (context-free)
- La ragione di questa terminologia sarà immediatamente chiara quando parleremo delle grammatiche
- La struttura delle espressioni aritmetiche, come pure il corretto “annidamento” (*nesting*) di blocchi di programma o la corretta strutturazione di blocchi condizionali sono caratteristiche catturabili entro il perimetro dei linguaggi liberi
- Questo però non vuol dire che i linguaggi di programmazione siano linguaggi liberi!
- Altre caratteristiche non sono catturabili da linguaggi liberi; una delle più semplici consiste nel vincolo che una variabile sia definita (esplicitamente, come in C, o implicitamente, come in Python) prima del suo uso

Linguaggi liberi e linguaggi dipendenti dal contesto

- Altre caratteristiche “non libere” riguardano i vincoli imposti dal *type system* e il fatto che in una chiamata di funzione il numero (ed eventualmente il tipo) degli argomenti coincida con il numero (e il tipo) dei parametri formali nella definizione
- Questi aspetti sarebbero catturabili (in linea di principio) da una classe di linguaggi ancora più espressiva detta (dei linguaggi) *dipendenti da contesto* (*context-dependent*)
- Gli strumenti formali per una definizione context-dependent dei linguaggi di programmazione sarebbe però troppo complicata
- Nella pratica si segue dunque un approccio differente: la verifica che un programma soddisfi i vincoli non context-free viene posticipata in una fase detta di *analisi semantica*

Linguaggi e compilatori

1 Sintassi e grammatiche

- Linguaggi liberi
- **Grammatiche formali**
- Derivazioni
- Gerarchie di linguaggi
- La descrizione di linguaggi reali
- Alberi di derivazione (parse tree) e ambiguità
- Grammatiche di tipo 1 e di tipo 3

Cosa é una grammatica?

- Come le espressioni regolari, una grammatica è un formalismo per definire linguaggi
- Si tratta però di un formalismo più generale
- Se un linguaggio L é definibile mediante un'espressione regolare, allora esiste una grammatica che definisce L
- Il viceversa non è sempre vero
- La modalità con cui una grammatica definisce un linguaggio è di natura “generativa”
- In altre parole, esiste un modo con cui, usando la grammatica è possibile *generare* (o *derivare*) tutte e sole le frasi del linguaggio
- Prima di definire le grammatiche in modo rigoroso, vediamo (a scopo “provocativo”) un pezzo della grammatica del C

Un pezzo di grammatica del C

iteration_statement

Può essere → : while '(' expression ')' statement
 Oppure → | do statement while '(' expression ')' ';' ;
 | for '(' expression_statement expression_statement ')' statement
 | for '(' expression_statement expression_statement expression ')' statement ;

statement

: labeled_statement
 | compound_statement
 | expression_statement
 | selection_statement
 | iteration_statement
 | jump_statement ;

compound_statement

: '{' '}'
 | '{' statement_list '}'
 | '{' declaration_list '}'
 | '{' declaration_list statement_list '}' ;

Definizione formale di grammatica

- Diamo la definizione generale di grammatica formale
- Una *grammatica* G è una quadrupla di elementi:

$$G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S}),$$

S appartiene ad $N!!!$

dove

Esempio : iteration_statement

- \mathcal{N} è un insieme di simboli, detti (*simboli*) *non terminali*;
- \mathcal{T} è un insieme di simboli (*simboli*) *terminali*, $\mathcal{N} \cap \mathcal{T} = \Phi$;
- \mathcal{P} è l'insieme delle *produzioni*, cioè scritte della forma $X \rightarrow Y$, dove $X, Y \in (\mathcal{N} \cup \mathcal{T})^*$;
- $\mathcal{S} \in \mathcal{N}$ è il *simbolo iniziale* (o *assioma*).
 Tutti i punti dove troviamo X possiamo mettere Y
- L'insieme $\mathcal{V} = \mathcal{N} \cup \mathcal{T}$ viene anche detto *vocabolario* della grammatica.
- Per gli esempi più semplici (e non per le grammatiche reali) è consuetudine utilizzare lettere maiuscole per i non terminali e lettere minuscole per i terminali

Le produzioni

- Minuscole : terminali (a, x , ...)
- Maiuscole : non terminali (A, B , X , ...)

- La struttura delle produzioni è ciò che caratterizza propriamente il *tipo* di grammatica, cioè la sua capacità espressiva
- Se le produzioni sono del tipo: $A \rightarrow xB$ oppure $A \rightarrow x$, dove $x \in T$ e $A, B \in \mathcal{N}$, la grammatica è detta *lineare destra*
- Se invece le produzioni sono del tipo: $A \rightarrow Bx$ oppure $A \rightarrow x$, dove $x \in T$ e $A, B \in \mathcal{N}$, la grammatica è detta *lineare sinistra*
- Una *grammatica regolare* è una grammatica lineare (destra o sinistra).
- Il nome non è casuale. Infatti possibile dimostrare che le grammatiche regolari descrivono proprio linguaggi regolari che già conosciamo
- Le grammatiche regolari sono dunque un formalismo alternativo a espressioni regolari ed automi finiti per descrivere lo stesso insieme di linguaggi

Le produzioni libere

IMPORTANTE*

- Libera : Può scegliere qualsiasi NON TERMINALE

- Per la definizione della sintassi dei linguaggi di programmazione, hanno invece particolare importanza i cosiddetti *linguaggi liberi da contesto* (o più semplicemente *linguaggi liberi*).
- I linguaggi liberi sono generabili da grammatiche (dette anch'esse *libere*) in cui le produzioni hanno la seguente forma generale

$$A \rightarrow \alpha \rightarrow \text{Sequenza di terminali e non terminali}$$

dove $A \in \mathcal{N}$ e $\alpha \in \mathcal{V}^*$, cioè in cui la parte sinistra è un qualsiasi simbolo non terminale mentre la parte destra è una qualsiasi stringa di terminali o non terminali.

Grammatiche più espressive

IMPORTANTE*

• Non libera :

- Le cosiddette grammatiche dependenti dal contesto le produzioni hanno la seguente forma:

$$\beta A \gamma \rightarrow \beta \alpha \gamma,$$

è molto meglio nei linguaggi dividere la cosa, e fare una parte con la grammatica e l'altra con il CONTROLLO DELLA SEMANTICA.

dove $\beta, \gamma \in \mathcal{V}^*$, $\alpha \in \mathcal{V}^+$ e $A \in N$.

- Infine, in una grammatica ricorsiva le produzioni hanno la forma generale:

$$\alpha \rightarrow \beta$$

dove $\alpha, \beta \in \mathcal{V}^*$

- Per come sono state definite le produzioni, le grammatiche regolari sono casi particolari di quelle libere, che sono casi particolari di quelle dipendenti dal contesto che sono casi particolari di quelle ricorsive
- Nel seguito considereremo solo grammatiche regolari o libere

Linguaggi e compilatori

1 Sintassi e grammatiche

- Linguaggi liberi
- Grammatiche formali
- **Derivazioni**
- Gerarchie di linguaggi
- La descrizione di linguaggi reali
- Alberi di derivazione (parse tree) e ambiguità
- Grammatiche di tipo 1 e di tipo 3

Derivazioni

- Le produzioni possono essere viste come (e sono dette anche) *regole di riscrittura*
- Ad esempio, una produzione del tipo $A \rightarrow aA$ si può leggere nel seguente modo:

Il simbolo A può essere sostituito dal (riscritto come il) simbolo terminale a seguito ancora dal simbolo non terminale A

- Una *derivazione* può allora essere definita come il processo mediante il quale, a partire dall'assioma ed applicando una sequenza di produzioni, si ottiene una stringa formata da soli terminali
- Una derivazione è, in generale, composta da più applicazioni di produzioni e dunque da più sequenze intermedie
- I singoli passaggi, da una sequenza intermedia ad un'altra, sono indicati dal simbolo \Rightarrow

$$S \Rightarrow \alpha_1 \Rightarrow \dots \Rightarrow \alpha_i \Rightarrow \dots \alpha_k$$

Linguaggio generato da una grammatica

- Una grammatica descrive (genera) il linguaggio costituito dalle sequenze di simboli terminali derivabili a partire dall'assioma S
- Consideriamo la grammatica $G_{n,m} = (\mathcal{N}, \mathcal{T}, \mathcal{P}, S)$ così definita:
 - $\mathcal{N} = \{S, A, B\}$;
 - $\mathcal{T} = \{a, b\}$;
 - $S = S$;
 - \mathcal{P} contiene le seguenti produzioni:

$$\begin{array}{l}
 S \rightarrow \epsilon, \quad \underline{S \rightarrow A} \\
 A \rightarrow a, \quad A \rightarrow aA, \quad \underline{A \rightarrow B} \\
 B \rightarrow bB, \quad \underline{B \rightarrow b}
 \end{array}$$

- Nel linguaggio che essa genera è inclusa la stringa ab perché:

$$S \Rightarrow A \Rightarrow aA \Rightarrow aB \Rightarrow ab.$$

Osservazioni metalinguistiche

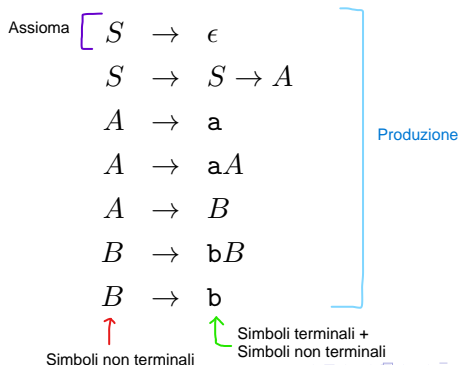
- Nelle derivazioni si usa il simbolo \Rightarrow (e non il simbolo \rightarrow impiegato nella definizione delle produzioni) allo scopo di non generare ambiguità
- Ad esempio, nella derivazione del lucido precedente ad un certo punto dalla stringa intermedia aA si passa alla stringa successiva aB
- Questo passaggio è denotato dalla scrittura $aA \Rightarrow aB$
- Il passaggio è possibile perché nella grammatica è presente la produzione $A \rightarrow B$ mentre non esiste una produzione $aA \rightarrow aB$
- La stringa di terminali generata da una derivazione è detta *frase* del linguaggio
- Le stringhe che in precedenza abbiamo indicato come “passaggi intermedi” sono dette *forme di frase*
- Si noti che una forma di frase deve includere almeno un simbolo non terminale che deve ancora essere riscritto

Osservazioni metalinguistiche

- Se da una forma di frase α è possibile ottenere una frase (o altra forma di frase) β mediante l'applicazione di $k \geq 0$ produzioni della grammatica G si scrive $\alpha \xRightarrow{k}_G \beta$
- Se si vuole evidenziare semplicemente che β è derivabile da α (con un numero arbitrario di produzioni) si scrive $\alpha \xRightarrow{*}_G \beta$
- Se $\alpha \neq \beta$, e dunque è necessaria l'applicazione di almeno una produzione, si scrive $\alpha \xRightarrow{+}_G \beta$
- Naturalmente, per snellire la notazione, quando non possano sorgere ambiguità, si omette il riferimento alla grammatica

Descrizione succinta di una grammatica

- Una grammatica può essere espressa in modo più succinto elencando le sole produzioni
- Si considerino, ad esempio, le regole di riscrittura della grammatica $G_{n,m}$ già introdotta



Descrizione succinta di una grammatica

- I quattro elementi che definiscono la grammatica possono essere facilmente dedotti qualora si convenga che le prime produzioni in elenco siano quelle relative all'assioma
- Infatti, i simboli non terminali sono tutti e soli i simboli che compaiono a sinistra nelle produzioni:
 - soli, perché un simbolo terminale non può essere riscritto
 - tutti, perché se un terminale non apparisse a sinistra vorrebbe dire che non esiste regola che lo riscrive e allora o quel simbolo non servirebbe (se non apparisse mai neppure a destra) oppure renderebbe impossibile applicare la produzione che lo dovesse utilizzare
- Da ciò deduciamo dunque che $\mathcal{N} = \{S, A, B\}$

Descrizione succinta di una grammatica

- L'assioma è il primo non terminale utilizzato (secondo la convenzione), e dunque $S = S'$
- I simboli terminali sono i simboli che compaiono a destra nelle produzioni e che non compaiono mai anche a sinistra, e dunque $\mathcal{T} = \{a, b\}$
- Infine le produzioni sono ovviamente quelle date (il solo elemento fornito esplicitamente)

Descrizione succinta e metasimboli

- Possiamo “economizzare” ancora sulla descrizione di una grammatica fondendo eventuali produzioni che hanno la stessa parte sinistra
- È consuetudine, infatti, usare la scrittura $X \rightarrow Y|Z$ al posto di $X \rightarrow Y$ e $X \rightarrow Z$
- Usando anche questa convenzione, la grammatica $G_{n,m}$ può essere descritta nel seguente modo, ancora più compatto:

$$\begin{array}{l}
 S \rightarrow \epsilon \mid A \\
 A \rightarrow a \mid aA \mid B \\
 B \rightarrow bB \mid b
 \end{array}$$

Meta-simbolo

MEANS

- Si noti come nella descrizione ora sono presenti due *metasimboli*, e precisamente \rightarrow e \mid .

Altri esempi di derivazioni

- Espressioni aritmetiche con +, * e () sempre bilanciate.

- Consideriamo la seguente grammatica G_{expr_1}

$$E \rightarrow E + E \mid E \times E \mid (E) \mid \text{number}$$

- $E + E \Rightarrow_{G_{expr_1}} \text{number} + E$ mediante l'applicazione della produzione $E \rightarrow \text{number}$ alla prima occorrenza di E .
- $E + E \xRightarrow{*}_{G_{expr_1}} \text{number} + \text{number}$ tramite l'applicazione della produzione $E \rightarrow \text{number}$ ad entrambe le occorrenze di E .
- $E \xRightarrow{*}_{G_{expr_1}} \text{number} + (E)$ in quanto
 $E \Rightarrow_{G_{expr_1}} E + E \Rightarrow_{G_{expr_1}} E + (E) \Rightarrow_{G_{expr_1}} \text{number} + (E)$.
- $E \xRightarrow{*}_{G_{expr_1}} \text{number} + (\text{number})$, in quanto $E \Rightarrow_{G_{expr_1}} E + E \Rightarrow_{G_{expr_1}} E + (E) \Rightarrow_{G_{expr_1}} \text{number} + (E) \Rightarrow_{G_{expr_1}} \text{number} + (\text{number})$.
- Una derivazione alternativa per $\text{number} + (\text{number})$ è
 $E \Rightarrow_{G_{expr_1}} E + E \Rightarrow_{G_{expr_1}} \text{number} + E \Rightarrow_{G_{expr_1}} \text{number} + (E) \Rightarrow_{G_{expr_1}} \text{number} + (\text{number})$.

Linguaggio generato dalle grammatiche $G_{n,m}$ e G_{expr_1}

● Example :

$n + n * (n + n)$

$E \Rightarrow E + E \Rightarrow E + \underline{E} * \underline{E} \Rightarrow E + E * (\underline{E}) \Rightarrow E + E * (E + E) \Rightarrow n + n * (n + n)$

- La grammatica $G_{n,m}$ genera il linguaggio (che abbiamo già più volte incontrato) $L_{n,m} = \{a^n b^m | n, m \geq 0\}$.
- La grammatica G_{expr_1} genera il linguaggio delle espressioni aritmetiche composte da $+$, \times , $($, $)$ e number.
- Le stringhe più corte in $L(G_{expr_1})$ sono
number, number + number, number * number, (number).

Linguaggio generato da una grammatica

- Per dimostrare formalmente che una data grammatica G genera un dato linguaggio L (cioè per provare che $L = L(G)$) si procede di regola come segue
- Dapprima si formula una congettura sulla forma delle frasi di $L(G)$, provando alcune semplici derivazioni
- Dopodiché si dimostra separatamente che:
 - se α è generata dal linguaggio, allora α ha la particolare forma congetturata
 - se α è una stringa con quella particolare forma, allora esiste una derivazione in grado di generarla
- Il procedimento può essere (relativamente) complesso anche per grammatiche molto semplici

Un esempio

- Dimostriamo formalmente che la grammatica $G_{n,m}$ genera il linguaggio $L_{n,m} = a^*b^*$.
- Ricordiamo che $G_{n,m}$ è:

$$S \rightarrow \epsilon \mid A$$

$$A \rightarrow a \mid aA \mid B$$

$$B \rightarrow bB \mid b$$

- $G_{n,m}$ genera “solo” stringhe del tipo $a^n b^m$.
 - $S \Rightarrow \epsilon$
 - $S \Rightarrow A \xRightarrow{k} a^k A \Rightarrow a^{k+1}, \quad k \geq 0$
 - $S \Rightarrow A \xRightarrow{k} a^k A \Rightarrow a^k B \xRightarrow{h} a^k b^h B \Rightarrow a^k b^{h+1},$
 $h, k \geq 0$
- $G_{n,m}$ genera “tutte” le stringhe del tipo $a^n b^m$ con $n, m \geq 0$.
 - Segue dall'arbitrarietà di k e h sopra.

Linguaggi e compilatori

1 Sintassi e grammatiche

- Linguaggi liberi
- Grammatiche formali
- Derivazioni
- Gerarchie di linguaggi
- La descrizione di linguaggi reali
- Alberi di derivazione (parse tree) e ambiguità
- Grammatiche di tipo 1 e di tipo 3

Gerarchia di Chomsky

- La seguente tabella descrive la gerarchia di grammatiche, ad ognuna delle quali viene associato l'automa riconoscitore e la classe di linguaggi corrispondente.
- La progressione è dalla grammatica meno espressiva (tipo 3) a quella più espressiva (tipo 0).

Tipo	Grammatica	Automa	Linguaggio
3	Regolare	Automa finito	Regolare
2	Libera	Automa a pila	Libero
1	Dipendente dal contesto	Automa limitato linearmente	Dipendente dal contesto
0	Ricorsiva	Macchina di Turing	Ricorsivamente enumerabile

Tipi dei linguaggi

- Si può dimostrare che se un linguaggio è generabile da una grammatica lineare (tipo 3) allora è regolare.
- Se invece un linguaggio L è generato da una grammatica G di tipo i ($i = 0, \dots, 2$), allora L è “al più” di tipo i .
- Infatti L potrebbe essere generabile anche da una grammatica più semplice (di tipo $i + 1$).
- Il linguaggio $L_{n,m}$ è regolare perché generato da una grammatica lineare, come abbiamo appena dimostrato.
- Il linguaggio $L(G_{expr_1})$ è invece al più libero perché generabile da una grammatica libera.

Tipi dei linguaggi

Qui c'era un errore

- Posto $\Sigma = \{a, b, c\}$, il linguaggio su Σ^* costituito dalle stringhe in cui ogni a è seguita da una b è al più libero perché generato da

$$S \rightarrow \epsilon \mid R$$

$$R \rightarrow b \mid c \mid ab \mid RR$$

- In realtà L è regolare, in quanto definibile anche mediante l'espressione regolare $(b|c|ab)^*$.
- Il linguaggio L_{pal} su $\{a, b\}$ costituito da tutte le stringhe α palindromo (cioè tali che $\alpha^R = \alpha$) è libero in quanto generabile dalla grammatica

$$S \rightarrow aSa, S \rightarrow bSb, S \rightarrow a, S \rightarrow b, S \rightarrow \epsilon$$

Si può poi dimostrare che L_{11} non è regolare.

Le espressioni regolari come linguaggio libero

- Come sappiamo, le espressioni regolari sono un formalismo per definire linguaggi.
- Ad esempio, l'espressione regolare $0(0|1)^*$, sull'alfabeto \mathcal{B} , definisce il linguaggio delle stringhe binarie che iniziano con 0.
- Per essere “manipolabili” automaticamente (ad esempio, per passare da un'espressione regolare al corrispondente automa nondeterministico), le espressioni regolari devono essere riconosciute come *ben formate*.
- Ad esempio, l'espressione $0(0|^*1$ è ben formata? Potremmo procedere alla costruzione dell'automa?

Le espressioni regolari come linguaggio libero

- La grammatica G_{re} così definita

$$E \rightarrow T \mid T \mid E$$

$$T \rightarrow F \mid FT$$

$$F \rightarrow (E) \mid (E)^* \mid A \mid A^*,$$

$$A \rightarrow 0 \mid 1$$

genera tutte le stringhe sull'alfabeto $0, 1, (,), |, *, \epsilon$ che sono espressioni regolari ben formate sull'alfabeto \mathcal{B} .

Esempio

- La stringa $0(0|1)^*$ appartiene a $L(G_{re})$ per la derivazione:

E	$\Rightarrow_{G_{re}}$	T	
	$\Rightarrow_{G_{re}}$	FT	Usando $T \rightarrow FT$
	$\Rightarrow_{G_{re}}$	FF	Usando $T \rightarrow F$
	$\Rightarrow_{G_{re}}$	AF	Usando $F \rightarrow A$
	$\Rightarrow_{G_{re}}$	$A(E)^*$	Usando $F \rightarrow (E)^*$
	$\Rightarrow_{G_{re}}$	$0(E)^*$	Usando $A \rightarrow 0$
	$\Rightarrow_{G_{re}}$	$0(T E)^*$	Usando $E \rightarrow T E$
	$\Rightarrow_{G_{re}}$	$0(T T)^*$	Usando $E \rightarrow T$
	$\Rightarrow_{G_{re}}$	$0(F T)^*$	Usando $T \rightarrow F$
	$\Rightarrow_{G_{re}}$	$0(F F)^*$	Usando $T \rightarrow F$
	$\Rightarrow_{G_{re}}$	$0(A F)^*$	Usando $F \rightarrow A$
	$\Rightarrow_{G_{re}}$	$0(A A)^*$	Usando $F \rightarrow A$
	$\Rightarrow_{G_{re}}$	$0(0 A)^*$	Usando $A \rightarrow 0$
	$\Rightarrow_{G_{re}}$	$0(0 1)^*$	Usando $A \rightarrow 1$

Linguaggi e compilatori

1 Sintassi e grammatiche

- Linguaggi liberi
- Grammatiche formali
- Derivazioni
- Gerarchie di linguaggi
- **La descrizione di linguaggi reali**
- Alberi di derivazione (parse tree) e ambiguità
- Grammatiche di tipo 1 e di tipo 3

Backus-Naur Form (BNF)

- Nella descrizione della sintassi dei linguaggi di programmazione, i simboli non terminali sono anche detti *variabili sintattiche* e vengono a volte rappresentati mediante un identificatore descrittivo racchiuso fra parentesi angolate.
- Esempio

$$\begin{aligned} \langle \text{comando if} \rangle \rightarrow & \text{if } \langle \text{espressione booleana} \rangle \{ \\ & \quad \langle \text{lista di comandi} \rangle \\ & \} \mid \\ & \text{if } \langle \text{espressione booleana} \rangle \{ \\ & \quad \langle \text{lista di comandi} \rangle \\ & \} \text{ else } \{ \\ & \quad \langle \text{lista di comandi} \rangle \\ & \} \end{aligned}$$

Altre o diverse convenzioni nella BNF

- Elementi opzionali possono essere inclusi fra i meta simboli [e].
- Ad esempio, potremo usare la scrittura

$$\begin{aligned} \langle \text{comando if} \rangle \rightarrow & \text{if } \langle \text{espressione booleana} \rangle \{ \\ & \langle \text{lista di comandi} \rangle \\ & \} [\text{else } \{ \\ & \langle \text{lista di comandi} \rangle \\ & \}] \end{aligned}$$

- Elementi ripetitivi possono essere inclusi fra i metasimboli { e }.
- Ad esempio

$$\langle \text{list di comandi} \rangle \rightarrow \langle \text{comando} \rangle \{ ; \langle \text{comando} \rangle \}$$

Altre o diverse convenzioni nella BNF

- Più recentemente, nella BNF i simboli terminali vengono scritti in grassetto.
- In questo modo diventa possibile sopprimere l'uso delle parentesi angolate intorno alle variabili sintattiche, migliorando la leggibilità complessiva. Per ulteriore “evidenziazione” le variabili sintattiche possono essere scritte in corsivo.
- Ad esempio, potremo scrivere

$$\langle \text{comando if} \rangle \rightarrow \mathbf{if} \langle \text{espressione booleana} \rangle \{ \\ \qquad \qquad \qquad \langle \text{lista di comandi} \rangle \\ \qquad \qquad \qquad \} [\mathbf{else} \{ \\ \qquad \qquad \qquad \langle \text{lista di comandi} \rangle \\ \qquad \qquad \qquad \}]$$

Altre o diverse convenzioni nella BNF

- Nel caso in cui possano sorgere ambiguità, i simboli terminali vengono racchiusi fra doppi apici.
- Un esempio è costituita dal caso di simboli terminali coincidenti con qualche metasimbolo
- Esempio (tratto dalla sintassi del C):

comando composto \rightarrow " { " { *dichiarazione* } { *comando* } " } "

- Infine, nelle forme più recenti, si usano spesso i due punti al posto della freccia nelle produzioni

Esercizi

1. Fornire una grammatica libera per l'insieme delle stringhe costituite da parentesi correttamente bilanciate (ad esempio, $()(())$ e $((()))$ devono far parte del linguaggio, mentre $()))$ (non deve farne parte).
2. Fornire una grammatica libera per il linguaggio $L_{12} = \{a^n b^{2n} | n \geq 0\}$ sull'alfabeto a, b .
3. Si consideri la seguente grammatica G_I

$S \rightarrow aSbb \mid \text{epsilon}$

$S \rightarrow I \mid A$

$I \rightarrow \text{if } B \text{ then } S \mid \text{if } B \text{ then } S \text{ else } S$

$A \rightarrow a$

$B \rightarrow b$

e si fornisca una derivazione per la stringa

if b then if b then a else a

Esercizi

1. Si scriva una grammatica libera o lineare per generare il linguaggio \mathcal{L} composto da tutte le stringhe sull'alfabeto $\{a, b, c\}$ che non contengono due caratteri consecutivi uguali.

Si consideri la seguente grammatica libera G

$$A \rightarrow bBa \mid a \mid b$$

$$B \rightarrow bA \mid Aa$$

2. Si verifichi dapprima che in essa le derivazioni hanno lunghezza $2\ell + 1$, $\ell \geq 0$. Si dimostri quindi, per induzione su ℓ , che il linguaggio generato da G è:

$$L(G) = \{b^h a^k : h + k = 3\ell + 1, h \geq \ell, k \geq \ell\}$$

3. Si scriva una grammatica libera per generare il linguaggio \mathcal{L} definito dalla seguente espressione regolare sull'alfabeto $\{a, b, c\}$:

$$ba^*(b \overset{\text{or}}{+} c)a^*c$$

Linguaggi e compilatori

$S \rightarrow bAbAc \mid bAcAc$
 $A \rightarrow \text{Eps} \mid aA$

1 Sintassi e grammatiche

- Linguaggi liberi
- Grammatiche formali
- Derivazioni
- Gerarchie di linguaggi
- La descrizione di linguaggi reali
- Alberi di derivazione (parse tree) e ambiguità
- Grammatiche di tipo 1 e di tipo 3

Equivalenza di grammatiche

- In Informatica (e non solo, naturalmente) esistono sempre molti modi per risolvere un problema, alcuni più efficienti altri meno.
- Per definire un linguaggio, ad esempio, si possono usare grammatiche equivalenti (cioè che generano lo stesso insieme di stringhe terminali) anche molto diverse.
- Le seguenti grammatiche definiscono $L_{n,n} = a^*b^*$:

$$\begin{array}{l} S \rightarrow \epsilon \mid A \\ A \rightarrow a \mid aA \mid B \\ B \rightarrow bB \mid b \end{array}$$

$$\begin{array}{l} S \rightarrow AB \\ A \rightarrow \epsilon \mid aA \\ B \rightarrow \epsilon \mid bB \end{array}$$

● 2 linguaggi diversi per la stessa espressione

- Le seguenti grammatiche definiscono lo stesso semplice linguaggio per le espressioni aritmetiche:

$$\begin{array}{l} E \rightarrow E + E \mid E * E \\ E \rightarrow (E) \mid \text{number} \end{array}$$

Forza precedenza operatori

$$\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T \times F \mid F \\ F \rightarrow \text{number} \mid (E) \end{array}$$

Quale grammatica usare?

- Poiché l'obiettivo è di costruire un riconoscitore (parser), il criterio che deve guidare il progettista è proprio di rendere possibile, e agevole, il parsing
- Fondamentalmente un parser deve “trovare” una derivazione dall'assioma alla stringa in input
- In generale, se una stringa appartiene al linguaggio, essa può essere derivata in molti modi possibili
- Tale ridondanza non è un fatto positivo perché in generale ha ripercussioni sull'interpretazione da dare alla stringa (si ricordi che le “stringhe” più importanti per noi sono programmi in un qualche linguaggio di programmazione)
- Cominceremo a lavorare proprio su questo.

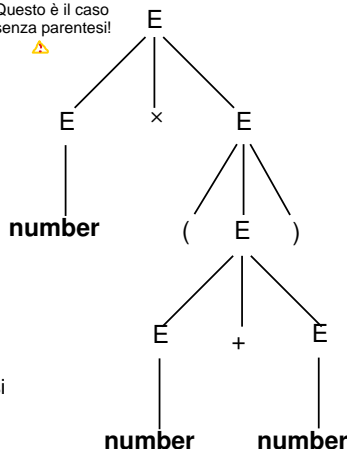
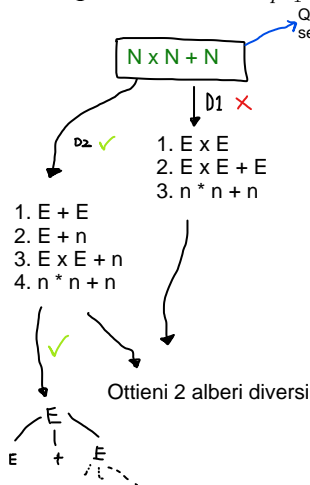
Alberi di derivazione (parse tree)

- Un parse tree per una grammatica libera $G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$ è un albero radicato ed etichettato che soddisfa le seguenti proprietà:
 - i nodi interni sono etichettati con simboli di \mathcal{N} e, in particolare, la radice è etichettata con l'assioma;
 - le foglie sono etichettate con simboli di \mathcal{T} o con il simbolo ϵ ;
 - se $A \in \mathcal{N}$ etichetta un nodo interno e X_1, \dots, X_k sono le etichette dei figli di (il nodo con etichetta) A , con $X_i \in \mathcal{V}$, allora deve esistere in \mathcal{P} la produzione $A \rightarrow X_1 X_2 \dots X_k$;
 - Se $A \in \mathcal{N}$ etichetta un nodo interno il cui unico figlio è un nodo con etichetta ϵ , allora deve esistere in \mathcal{P} la produzione $A \rightarrow \epsilon$.
- La slide successiva presenta un primo esempio di parse tree

● La relazione padre \rightarrow figlio è la relazione pt.sx \rightarrow pt.dx vocabolario

Parse tree (Esempio)

- Parse tree della frase $\text{number} \times (\text{number} + \text{number})$ per la grammatica G_{expr_1}



DEF

Parsing :
Passare da una
espressione
regolare ad un
albero, quindi con
una struttura

Parse tree

- Ad ogni parse tree corrisponde una e una sola frase, che si ottiene giustapponendo le etichette sulle foglie considerate da sinistra verso destra
- Il viceversa in generale non vale: ad una frase possono corrispondere più parse tree
- Questo non è il caso della frase **number** \times (**number** + **number**) alla quale corrisponde il solo parse tree presentato nella slide precedente
- Come semplice esercizio, si mostrino invece due differenti parse tree della frase **number** \times **number** + **number** per la grammatica G_{expr_1} (la “soluzione” si trova comunque più avanti)
- In realtà però ciò che conta sono i parse tree in relazione alle derivazioni delle frasi piuttosto che alle frasi
- Anche per una frase cui è associabile un solo parse tree (come quella vista in precedenza) possono infatti corrispondere più derivazioni

Parse tree

- Per la frase **number** \times (**number** + **number**) esistono almeno le seguenti due derivazioni

$$\begin{aligned} E &\Rightarrow E \times E \Rightarrow E \times (E) \Rightarrow E \times (E + E) \\ &\Rightarrow E \times (E + \text{number}) \Rightarrow E \times (\text{number} + \text{number}) \\ &\Rightarrow \text{number} \times (\text{number} + \text{number}) \end{aligned}$$

e

$$\begin{aligned} E &\Rightarrow E \times E \Rightarrow \text{number} \times E \Rightarrow \text{number} \times (E) \\ &\Rightarrow \text{number} \times (E + E) \Rightarrow \text{number} \times (\text{number} + E) \\ &\Rightarrow \text{number} \times (\text{number} + \text{number}) \end{aligned}$$

Indeterminatezza eliminabile

- L'indeterminatezza illustrata nella slide precedente è ineliminabile
- Infatti il parse tree lascia sempre indeterminato l'ordine di applicazione di tutte quelle riscritture che non coinvolgano, nell'albero stesso, nodi interni su uno stesso cammino radice-foglia.
- Solo le grammatiche regolari (lineari destre o sinistre) ne sono esenti perché, sulla base della struttura delle produzioni, ogni forma di frase include un solo non terminale e dunque l'ordine di riscrittura è vincolato
- Questa però è una indeterminatezza eliminabile fissando un ordine (che può essere arbitrario, ma che naturalmente deve "agevolare" il processo di parsing) con cui procedere alla riscrittura dei non terminali quando ve ne sia più d'uno
- Questo ci porta a considerare derivazioni con ordine di riscrittura prefissato, dette più precisamente *canoniche*

Derivazioni canoniche

- Se in una derivazione, in presenza di una forma di frase con più di un simbolo non terminale, viene riscritto sempre il non terminale più a destra (risp. sinistra) allora la derivazione è detta *canonica destra* (risp. *canonica sinistra*)
- Le due derivazioni presentate in precedenza per la frase **number** \times (**number** + **number**) sono esempi rispettivamente di derivazioni canoniche destre e sinistre
- Se consideriamo solo derivazioni canoniche (destre o sinistre, ma non entrambe) allora vale la proprietà che ad ogni parse tree corrisponde una e una sola derivazione
- È però sempre possibile che di una frase esistano derivazioni canoniche differenti e, dunque, per quanto appena osservato, parse tree differenti

Ambiguità

- Questa eventualità, ovvero il fatto che esistano più derivazioni canoniche per una frase, e quindi parse tree diversi, è un fenomeno che viene definito con il termine di *ambiguità* della grammatica
- Si definisce cioè *ambigua* una grammatica in cui almeno una frase ammette differenti parse tree.
- L'ambiguità è un ostacolo alla realizzazione di un parser e, prima ancora, alla attribuzione di un significato alle frasi, poiché alberi diversi impongono strutture sintattiche diverse
- Vediamo subito due esempi importanti che riguardano la seguente grammatica G_I che descrive una parte di un ipotetico linguaggio imperativo

$$S \rightarrow I \mid A, \quad A \rightarrow a, \quad B \rightarrow b$$
$$I \rightarrow \text{if } B \text{ then } S \mid \text{if } B \text{ then } S \text{ else } S$$

Esempio 1

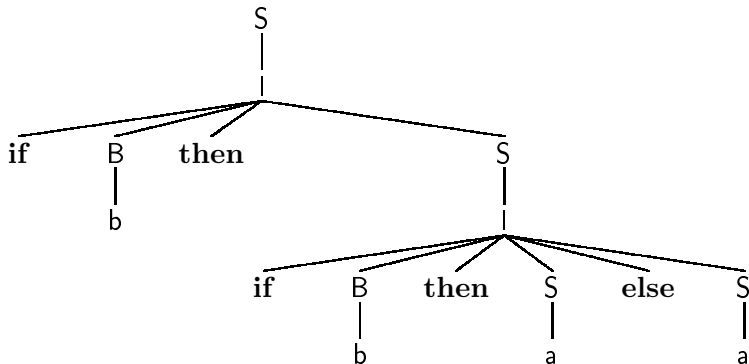
ALBERO1



- Un parse tree per la frase

if b then if b then a else a

nella grammatica G_I è:



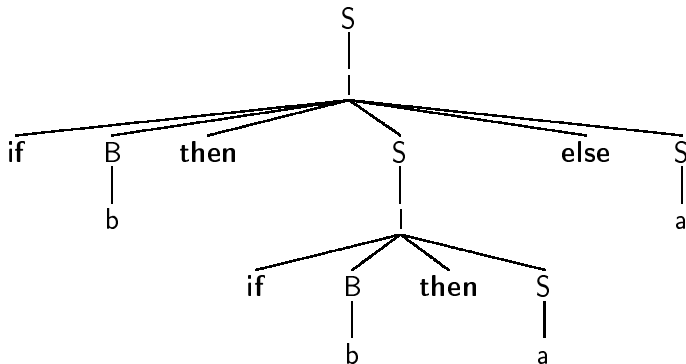
Esempio 1

ALBERO2 ✗

- La grammatica G_I è ambigua perché alla stessa frase

if b then if b then a else a

(che riportiamo per comodità) corrisponde anche un diverso parse tree, e precisamente

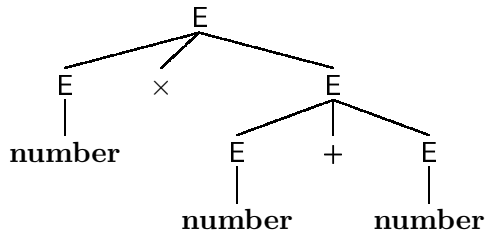


Esempio 2

- Alla frase

number \times number + number

corrisponde in G_{exp_1} un primo parse tree illustrato di seguito:



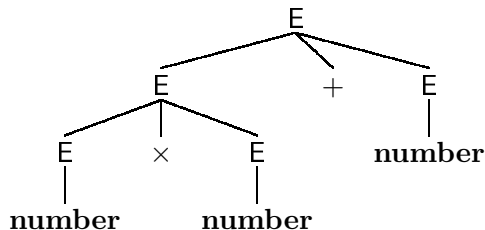
- Si noti come esso suggerisca un'*interpretazione* dell'espressione, e precisamente **number \times (number + number)**, che non coincide con quella che universalmente si dà alla frase in Matematica.

Esempio 2

- Il secondo parse tree che corrisponde, in G_{exp_1} , alla frase

number \times number $+$ number

è:



- Questa volta l'albero suggerisce l'interpretazione "corretta" **(number \times number) $+$ number**.

Ambiguità

- Come accennato sopra, se una grammatica è ambigua il parsing (e poi l'interpretazione) diviene problematico
- È quindi importante che la grammatica non sia ambigua e, nel caso lo sia, sapere come disambiguare le “frasi ambigue”.
- Per certe classi di grammatiche l'eventuale ambiguità può essere rilevata (e rimossa) mediante un algoritmo
- Si tratta però di un approccio che, quand'anche possibile, risulta macchinoso
- In certi casi si preferisce utilizzare una grammatica ambigua e implementare una regola di disambiguazione direttamente nel parser

Esempio

- Sappiamo che la grammatica G_I è ambigua perché ammette due differenti interpretazioni per la frase **if b then if b then a else a**
- Una grammatica non ambigua che “forza” l'interpretazione classica è:

$$S \rightarrow O \mid C$$

$$C \rightarrow \text{if } B \text{ then } C \text{ else } C \mid A$$

$$O \rightarrow \text{if } B \text{ then } S \mid \text{if } B \text{ then } C \text{ else } O$$

$$A \rightarrow a$$

$$B \rightarrow b$$

- L'unica derivazione possibile è:

Genererà un albero corretto

$$S \Rightarrow O \Rightarrow \text{if } B \text{ then } S$$

$$\Rightarrow \text{if } B \text{ then } C \Rightarrow \text{if } B \text{ then if } B \text{ then } C \text{ else } C$$

$$\stackrel{*}{\Rightarrow} \text{if } b \text{ then if } b \text{ then } a \text{ else } a$$

Altri aspetti problematici

- Le seguenti (eventuali) caratteristiche di una grammatica possono ostacolare seriamente il processo di parsing:
 - possibilità di avere cicli (derivazioni del tipo $A \xRightarrow{+} A$) o comunque di *left recursion* (derivazioni del tipo $A \xRightarrow{+} A\alpha$, con $\alpha \neq \epsilon$);
 - presenza di prefissi comuni a due o più produzioni relative allo stesso non terminale, ad esempio $A \rightarrow \alpha\beta_1$ e $A \rightarrow \alpha\beta_2$.
- Ad esempio, la grammatica

$$S \rightarrow Sa \mid b$$

presenta una ricorsione immediata a sinistra in quanto $S \rightarrow Sa$.

- Per risolvere questo problema basta che quando trova un non terminale, deve essere sempre definito dopo (not sure)

Esempi

- La grammatica

$$\begin{aligned} S &\rightarrow A \mid b \\ A &\rightarrow Aa \mid Sa \end{aligned}$$

presenta una ricorsione a sinistra in quanto $S \xRightarrow{+} Sa$.

- La grammatica G_I (oltre ad essere ambigua) presenta un prefisso comune alla due produzioni che riscrivono il simbolo I .
- Nella grammatica

$$\begin{aligned} S &\rightarrow SA \\ A &\rightarrow a \mid \epsilon \end{aligned}$$

c'è una produzione $A \rightarrow \epsilon$ e questa provoca il ciclo $S \xRightarrow{+} S$.

Precedenza degli operatori

- Come sappiamo, la grammatica G_1 :

$$E \rightarrow E + E \mid E \times E \mid \mathbf{number} \mid (E)$$

è ambigua

- In questo caso per eliminare l'ambiguità è sufficiente introdurre un nuovo simbolo non terminale E'

$$\begin{aligned} E &\rightarrow E + E' \mid E \times E' \mid E' \\ E' &\rightarrow \mathbf{number} \mid (E) \end{aligned}$$

- Quindi esiste una sola derivazione, anche se non è detto che sia con le giuste precedenze.

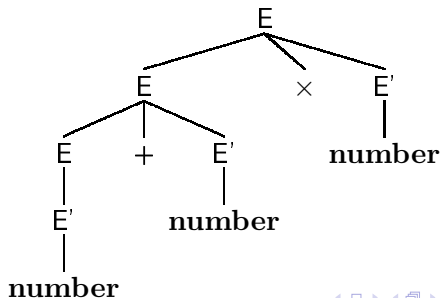
- La nuova grammatica non è più ambigua ma, come anche la stessa grammatica G_{exp_1} di partenza, presenta un problema legato alla precedenza degli operatori.

Precedenza degli operatori

- Ad esempio, per la frase **number + number × number** l'unica derivazione canonica destra possibile è:

$$\begin{aligned}
 E &\Rightarrow E \times E' \Rightarrow E \times \text{number} \\
 &\Rightarrow E + E' \times \text{number} \Rightarrow E + \text{number} \times \text{number} \\
 &\Rightarrow \text{number} + \text{number} \times \text{number}
 \end{aligned}$$

alla quale corrisponde l'albero di derivazione "erroneo":




Precedenza degli operatori

- Le usuali precedenze di operatore possono essere forzate introducendo differenti simboli non terminali per i diversi livelli di precedenza.
- Ad esempio, nel caso di somma e prodotto possiamo introdurre due livelli di precedenza, rappresentati dai non terminali E e T , oltre ad un simbolo (useremo F) per la categoria delle espressioni di base (nel nostro caso identificatori ed espressioni tra parentesi):

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T \times F \mid F$$

$$F \rightarrow \text{number} \mid (E)$$



Garantisce che in assenza di parentesi il $*$ viene fatto con precedenza

Precedenza degli operatori (continua)

- La grammatica precedente (con le ovvie estensioni) è in grado di trattare senza ambiguità anche il caso degli operatori di divisione e sottrazione, inseriti nelle giuste categorie sintattiche:

$$E \rightarrow E + T \mid E - T \mid T$$

$$T \rightarrow T \times F \mid T / F \mid F$$

$$F \rightarrow \mathbf{number} \mid (E)$$

- Domanda: cosa cambierebbe se scrivessimo

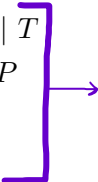
$$T \rightarrow F \times T$$

invece di

$$T \rightarrow T \times F \quad ?$$

Precedenza degli operatori

- Volendo inserire anche l'operatore di esponenziazione (che ha precedenza maggiore), è necessario prevedere un'ulteriore simbolo non terminale e ricordarsi che l'operatore di esponenziazione (qui useremo il simbolo $^$) è associativo a destra:

Precedenza liv1	$E \rightarrow E + T \mid E - T \mid T$		Parsa in modo corretto +, *, -, /, ^
Precedenza liv2	$T \rightarrow T \times P \mid T / P \mid P$		
Precedenza liv3	$P \rightarrow F^P \mid F$		
	$F \rightarrow \text{number} \mid (E)$		

Esercizi

- Si consideri il linguaggio $L_{a>b}$ delle stringhe su a, b che contengono più a che b . Si dica, giustificando la risposta, se la seguente grammatica libera genera $L_{a>b} \longrightarrow$ è vero perc

$$S \rightarrow a \mid aS \mid Sa \mid abS \mid aSb \mid Sab \mid baS \mid bSa \mid Sba$$

- Si fornisca una grammatica libera per il linguaggio su B contenente tutte e sole le stringhe con un diverso numero di 0 e 1.
- Si dica qual è il linguaggio generato dalla grammatica

$$\begin{aligned} S &\rightarrow aA \\ A &\rightarrow AB \mid B \\ B &\rightarrow b \end{aligned}$$

Ex :
1100
100
1110000

Per incominciare
una grammatica che
riconosce uno zero in
più degli 1

Linguaggi e compilatori

1 Sintassi e grammatiche

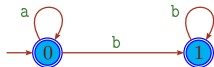
- Linguaggi liberi
- Grammatiche formali
- Derivazioni
- Gerarchie di linguaggi
- La descrizione di linguaggi reali
- Alberi di derivazione (parse tree) e ambiguità
- Grammatiche di tipo 1 e di tipo 3

Automi finiti e grammatiche lineari destre

- Dato un automa a stati finiti M che riconosce il linguaggio L è immediato definire una grammatica lineare destra G_M che genera lo stesso linguaggio, cioè tale che $L(G_M) = L$.
- La costruzione è molto semplice:
 - per ogni stato q dell'automata la grammatica conterrà un simbolo non terminale A_q ;
 - l'assioma iniziale è A_{q_0} ;
 - per ogni transizione $\delta(q, a) = q'$, la grammatica conterrà la produzione $A_q \rightarrow aA_{q'}$;
 - se q' è uno stato finale, la grammatica conterrà la produzione $A_{q'} \rightarrow \epsilon$.

Esempio

- La grammatica corrispondente all'automa $M_{n,m}$:



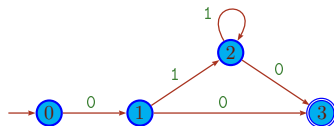
è

$$A_{q_0} \rightarrow aA_{q_0} \mid bA_{q_1} \mid \epsilon$$

$$A_{q_1} \rightarrow bA_{q_1} \mid \epsilon$$

Esempio

- La grammatica corrispondente all'automa M_{ss} :



è

$$A_{q_0} \rightarrow 0A_{q_1}$$

$$A_{q_1} \rightarrow 1A_{q_2} \mid 0A_{q_3}$$

$$A_{q_2} \rightarrow 1A_{q_2} \mid 0A_{q_3}$$

$$A_{q_3} \rightarrow \epsilon$$

Esempio

- La grammatica corrispondente all'automa M_{parity} :



è

$$A_{q_0} \rightarrow 0A_{q_0} \mid 1A_{q_1} \mid \epsilon$$

$$A_{q_1} \rightarrow 1A_{q_0} \mid 0A_{q_1}$$

Automi finiti e grammatiche lineari destre

- La costruzione appena vista è completamente reversibile.
- Data una grammatica lineare destra G , possiamo facilmente costruire un automa non deterministico M che riconosce il linguaggio generato da G .
 - per ogni non terminale A della grammatica definiamo uno stato q_A dell'automa;
 - per ogni produzione $A \rightarrow aB$ inseriamo la transizione $\delta(q_A, a) = q_B$ e per ogni produzione $A \rightarrow B$ inseriamo la transizione $\delta(q_A, \epsilon) = q_B$
 - lo stato iniziale è q_S ;
 - si definisce uno stato finale q_f , non corrispondente ad alcun simbolo non terminale;
 - per ogni produzione $A \rightarrow a$ si pone $\delta(q_A, a) = q_f$ e, analogamente, per ogni produzione $A \rightarrow \epsilon$ si pone $\delta(q_A, \epsilon) = q_f$.

Esempio

- L'automa corrispondente alla grammatica:

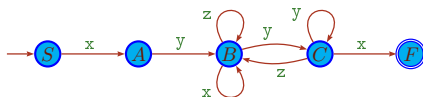
$$S \rightarrow xA$$

$$A \rightarrow yB$$

$$B \rightarrow xB \mid zB \mid yC$$

$$C \rightarrow x \mid zB \mid yC$$

è



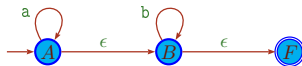
Esempio

- L'automa corrispondente alla grammatica:

$$A \rightarrow aA \mid B$$

$$B \rightarrow bB \mid \epsilon$$

è



Grammatiche dipendenti dal contesto

- Ricordiamo la forma generale delle produzioni di una grammatica di tipo 1 (dipendente dal contesto):

$$\beta A \gamma \rightarrow \beta \alpha \gamma$$

dove $\beta, \gamma \in \mathcal{V}^*$, $\alpha \in \mathcal{V}^+$ e $A \in \mathcal{N}$.

- La forma delle produzioni “spiega” il nome di queste grammatiche.
- Infatti, il simbolo A può essere riscritto come α solo se appare nel *contesto* rappresentato dalle stringhe β e γ .

Linguaggi dipendenti dal contesto

- Sono così detti i linguaggi generabili da grammatiche dipendenti dal contesto.
- Poiché la forma generale delle produzioni di una grammatica context-dependent sono più generali di quelle di una grammatica context-free, l'insieme dei linguaggi dipendenti dal contesto “contiene” l'insieme dei linguaggi liberi.
- Viceversa, ci sono linguaggi dipendenti dal contesto che non sono liberi.
- Un esempio è il linguaggio

$$L_{n,n,n} = \{a^n b^n c^n | n \geq 0\}$$

Costrutti non liberi nei linguaggi di programmazione

- Alcuni aspetti della sintassi dei linguaggi di programmazione non sono catturabili da produzioni di grammatiche libere.
- Ad esempio, il fatto che una variabile debba essere dichiarata prima dell'uso non è esprimibile mediante una grammatica libera.
- Tale caratteristica è catturata dal cosiddetto *linguaggio delle repliche* (che non è libero):

$$L_R = \{\alpha\alpha \mid \alpha \in \mathcal{T}^*\}.$$

Costrutti non liberi nei linguaggi di programmazione

- Un altro aspetto non esprimibile con grammatiche libere è che il numero e il tipo degli argomenti di una funzione coincida ordinatamente con il numero e il tipo dei parametri formali.
- Anziché utilizzare grammatiche più potenti (che renderebbero difficoltoso, quando non impossibile, il parsing), l'approccio consiste nell'ignorare il problema a livello sintattico.
- La verifica di correttezza viene completata invece durante l'*analisi semantica*