Compilatori
Funzioni e struttura
Analisi lessicale: token, pattern e lessemi
Analisi sintattica
Parsing e grammatiche libere

Linguaggi formali e compilazione

Corso di Laurea in Informatica

A.A. 2014/2015

Linguaggi formali e compilazione

Compilatori

- Funzioni e struttura
- Analisi lessicale: token, pattern e lessemi

Analisi sintattica

- Parsing e grammatiche libere

Che cosa è un compilatore

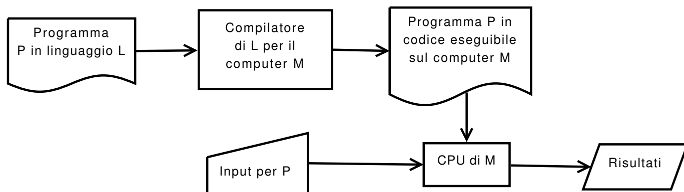
- Un compilatore è un *traduttore*.
- Questo vuol dire un *programma* che:
  - riceve in input la descrizione di un “oggetto” X scritto in un certo linguaggio L;
  - produce in output la descrizione di X in un altro linguaggio L'.
- La traduzione deve essere *corretta* nel senso che il *significato* (o *semantica*) dell’input deve essere preservato.

Traduttori in generale

- Alcuni semplici esempi:
  - un algoritmo che trasforma numeri dal sistema romano a quello posizionale decimale;
  - il programma `ps2pdf` disponibile in Linux;
  - un software che trasforma programmi scritti in C in programmi scritti in binario per l'architettura x86.
- Solo nell'ultimo caso si parla propriamente di *compilazione*.
- Che cosa significa, negli esempi sopra citati, che "il significato deve essere preservato" (e dunque che la traduzione è corretta)?

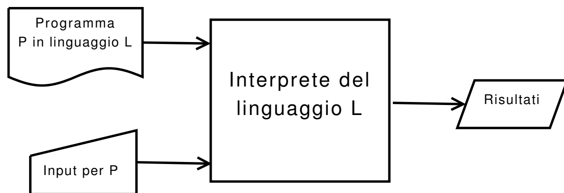
Compilazione ed esecuzione di programmi in linguaggio ad altro livello

- La compilazione è il più importante caso di traduzione in ambito informatico.
- Un *compilatore* per un linguaggio L e una macchina $\mathcal{M}$ è un traduttore che, data una stringa (programma) in linguaggio L, produce un programma "equivalente" nel linguaggio macchina di $\mathcal{M}$.
- Compilazione e successiva esecuzione di un programma



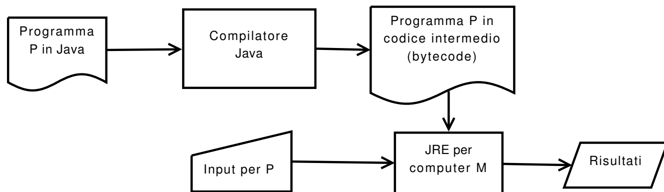
Uno schema alternativo: l'interpretazione pura

- Un'alternativa alla compilazione è l'*interpretazione* diretta dei programmi.
- Nel caso dell'interpretazione di L su $\mathcal{M}$, un programma in esecuzione su $\mathcal{M}$ prende direttamente in input frasi in linguaggio L e le "esegue", producendo in output il risultato dell'esecuzione.
- Lo schema per l'interpretazione pura



Esistono anche soluzioni “miste”

- Compilazione ed intepretazione in Java



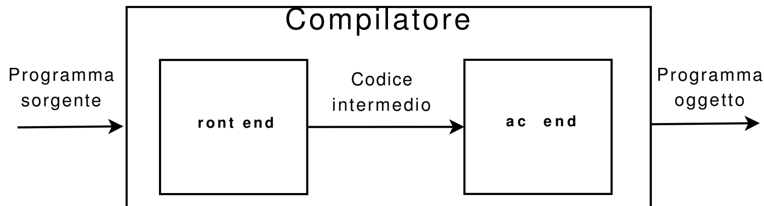
- In tutti i casi (anche nell’interpretazione “pura”), le tecniche di compilazione giocano un ruolo fondamentale.

Inciso: linguaggi compilati e linguaggi interpretati

- In linea di principio, di uno stesso linguaggio L potrebbero esistere implementazioni compilate ed implementazioni interpretate.
- Nella pratica, i linguaggi “si specializzano” in una sola delle due alternative, anche (o forse soprattutto) perché compilazione ed interpretazione offrono vantaggi e svantaggi complementari, che sono poi riflessi nei linguaggi stessi.
- C, C++, Fortran e Pascal sono linguaggi compilati.
- I linguaggi dinamici (Perl, Python, PHP, ...) sono linguaggi interpretati.
- Il caso di Java.

Schema generale di un compilatore

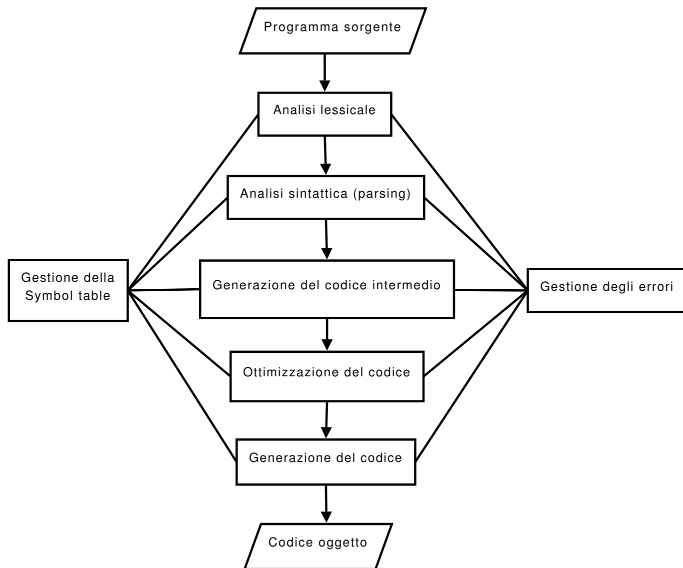
- Cominciamo a guardare “dentro la scatola” ...



Due componenti principali

- Il front-end rappresenta la fase di *analisi* dell'input.
- Il back-end è la fase di *sintesi* dell'output.
- Il codice intermedio è indipendente dall'architettura hardware (*i386*, *powerpc*, *sparc*, ...).
- Vantaggi di questa organizzazione:
  - modularità;
  - portabilità;
  - economicità.
- Noi ci occuperemo "esclusivamente" del front-end

Uno schema ancora più dettagliato

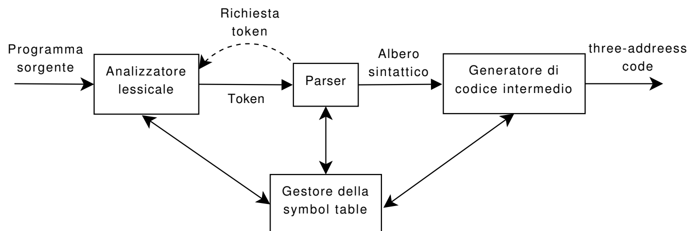


Il front-end più in dettaglio

- Il front-end di un compilatore (cioè la parte che termina con la creazione di una rappresentazione intermedia del programma) è costituita dai seguenti moduli:
  - analizzatore lessicale (scanner);
  - analizzatore sintattico (parser);
  - symbol table (e routine di gestione);
  - generatore di codice intermedio.
- Per ragioni di tempo non ci potremo occupare, in questo corso, della gestione degli errori.

Il modello di front-end

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



Analizzatore lessicale

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

```
X = 3.14;
```

potrebbe venire trasformata nella sequenza di token:

id assign number sep

Analizzatore lessicale (continua)

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

Parser

- Il parsing opera esclusivamente al livello di *sintassi* (cioè della corretta formazione delle frasi).
- Un *parser* (detto appunto anche *analizzatore sintattico*) è uno strumento che, in termini generali, consente di *dare struttura ad una descrizione lineare*.
- Per questa ragione il parsing è un processo molto comune, anche in altre ambiti dell'Informatica.
- Alcuni esempi:
  - passare dalla descrizione testuale di un grafo ad una sua rappresentazione in liste di adiacenze;
  - dato un insieme di pagine web, creare un grafo che ne descriva la struttura dei collegamenti;
  - dato un documento XML, creare una rappresentazione dell'informazione in esso contenuta (detta *information set*) da rendere disponibile all'applicazione client.

Parser (continua)

- La struttura viene conferita alla descrizione lineare in input in accordo a regole formali.
- Nel caso dei linguaggi di programmazione tali formalismi sono generalmente *grammatiche*.
- L'output viene a sua volta specificato mediante un formalismo in grado di esprimere le *proprietà di struttura*.
- Generalmente tale formalismo è un *albero* (con varie proprietà).
- In tutti i casi, il parsing include un'analisi della *correttezza formale* (well-formedness) delle stringhe da analizzare.

Generazione del codice intermedio

- A partire dalla struttura prodotta dal parser e (come vedremo) dalle informazioni raccolte nella symbol table, questa fase produce un programma scritto in un linguaggio intermedio
- Il codice intermedio è indipendente dalla macchina e quindi è teoricamente portabile.
- Esistono diverse soluzioni per rappresentare tale codice e, fra queste, il cosiddetto *three-address code* (codice a tre indirizzi).

Breve descrizione delle altre fasi

Ottimizzazione del codice In questa fase operano algoritmi che *manipolano* il programma scritto in codice intermedio in modo da ottenere un programma equivalente che (in genere) utilizza una minore quantità di spazio e/o che "gira" più velocemente.

Generazione del codice In questa fase il programma in codice intermedio (eventualmente ottimizzato) viene trasformato in codice macchina. Qui vengono prese le decisioni sull'allocazione dei dati in memoria, il codice di accesso e l'utilizzo dei registri.

Gestione della tabella dei simboli È un'attività che accompagna tutte le fasi della compilazione. La *tabella dei simboli* è essenzialmente un dizionario che registra tutti i nomi usati nel programma e le informazioni ad esso associate (esempio, un identificatore e il suo

Breve descrizione delle fasi

Gestione degli errori Durante una qualunque delle fasi si possono verificare degli errori, che devono essere individuati e segnalati con opportuni messaggi diagnostici. Gli esempi più facilmente comprensibili di errore riguardano le fasi di analisi lessicale (ad esempio, un identificatore che contiene un carattere non ammesso) e di analisi sintattica (ad esempio un'espressione aritmetica mal formata). In generale se vengono rilevati errori non viene prodotto il codice oggetto.

Le fasi attraverso un semplice esempio


```
position = initial + rate * 60
```

Lexical Analyzer

⟨id, 1⟩ ⟨=⟩ ⟨id, 2⟩ ⟨+⟩ ⟨id, 3⟩ ⟨∗⟩ ⟨60⟩

Syntax Analyzer

= (⟨id, 1⟩, + (⟨id, 2⟩, ∗ (⟨id, 3⟩, 60)))

Semantic Analyzer

= (⟨id, 1⟩, + (⟨id, 2⟩, ∗ (⟨id, 3⟩, inttofloat (60))))

Intermediate Code Generator

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Code Optimizer

```
t1 = id3 * 60.0
id1 = id2 + t1
```

Code Generator

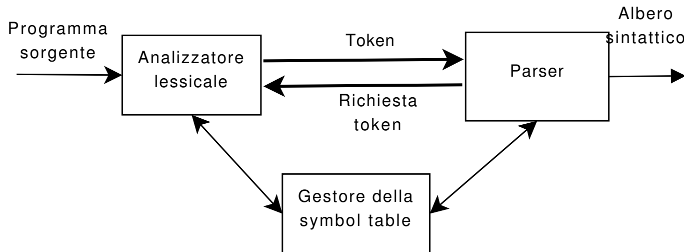
```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

1	position	···
2	initial	···
3	rate	···

SYMBOL TABLE

Interazione parser-lexical analyzer

- Lo scanner opera come “routine” del parser.



- Quando il parser deve leggere il prossimo simbolo di input esegue una chiamata allo scanner.
- Lo scanner legge la “prossima” porzione di input fino a quando non riconosce un *token*, che viene restituito al parser.
- Ciò che viene propriamente restituito è un *token name*, una delle due componenti che costituiscono il token vero e proprio.

Token

- Un token è un oggetto astratto caratterizzato da due attributi, un *token name* (obbligatorio) e un *valore* opzionale.
- Ecco alcuni esempi di sequenze di caratteri riconosciuti come token da un compilatore C++:
  - "`0.314E+1`", cui corrisponde la coppia (token) in cui il nome è **number** e il valore è il numero razionale 3.14, opportunamente rappresentato;
  - "`x1`", cui corrisponde un token in cui il nome è **id** e il cui valore è, a sua volta, un insieme di informazioni elementari (la stringa di caratteri che forma l'identificatore, il suo tipo, il punto del programma dove è stato definito);
  - "else", cui corrisponde il solo token name **else**.

Token name

- Il nome del token serve principalmente al parser, per stabilire correttezza sintattica delle frasi (e costruire l'albero sintattico).
- Il valore dei token è invece, da questo punto di vista, non rilevante (almeno in prima istanza).
- Ad esempio, le frasi "`x<10`" e "`pippo!=35`" sono del tutto equivalenti e riconducibili alla frase generica in cui il valore di un identificatore è confrontato con un numero.
- Una tale frase può quindi essere astrattamente descritta come "**id comparison number**", dove abbiamo utilizzato il nome **comparison** per il token che descrive gli operatori relazionali.

Token value

- Il valore di un token gioca il suo ruolo principale nella traduzione del codice.
- Con riferimento agli esempi del lucido precedente:
  - di un identificatore è importante sapere (tra gli altri) il tipo, perché da questo dipende (in particolare) la quantità di memoria allocare;
  - di un operatore di confronto è necessario sapere esattamente di quale confronto si tratta, per poter predisporre il test e l'istruzione (macchina) di salto opportuna;
  - di una costante numerica è necessario sapere il valore per effettuare l'inizializzazione corretta.

Token (continua)

- È naturalmente il progettista del linguaggio che stabilisce quali debbano essere considerati token.
- Ad un token name possono corrispondere (in generale) molte stringhe alfanumeriche nel codice sorgente (si pensi ai token che descrivono numeri o identificatori).
- Il progettista deve quindi stabilire esattamente anche la *mappatura* fra stringhe e token name (cioè, quali stringhe corrispondono a quali token).
- Tale mappatura viene descritta utilizzando opportuni formalismi, il più importante dei quali è costituito dalle *espressioni regolari*.

Un minimo di terminologia

- Le sequenze di simboli che possono legalmente comparire in un programma e che corrispondono ai token sono dette *lessemi*.
- Un *pattern* è invece una descrizione (formale o informale) delle stringhe (lessemi) che devono essere riconosciute (dall'analizzatore lessicale) come istanze di un determinato token name.
- Le espressioni regolari sono un formalismo per specificare tali pattern.

Pattern, lessemi e token name (continua)

- La seguente tabella illustra, mediante alcuni esempi, i concetti che stiamo analizzando.

Token name	Pattern	Esempio di lessema
id	$letter(letter \mid digit)^*$	`pippo1`
number	$[+ \mid -]nzd\ digit^*[.]digit^*$	`-3.14`
comparison	$< \mid > \mid <= \mid >= \mid = \mid !=$	`<`
literal	[:*alpha*:]	“`Pi greco`”
if	`if`	`if`
while	`while`	`while`

Si noti che, per ragioni di spazio, il pattern che descrive il token **number** non prevede la notazione scientifica.

Scopo del parsing

- L'obiettivo della fase di parsing è innanzitutto di stabilire se una sequenza di token rappresenta una "frase" corretta del linguaggio e, nel caso, descriverne la struttura.
- Sulla base di che cosa possiamo stabilire, ad esempio, che

  ```
  while (a>0) {a=a-1}
  ```

  è una frase corretta in C/C++, mentre

  ```
  while (a>0) a=a-1}
  ```

  è una frase sintatticamente errata?
- La risposta (anche se solo parziale) è che una frase è corretta se e solo se è conforme alla *sintassi* del linguaggio.
- Il formalismo che si è imposto per la descrizione della sintassi dei linguaggi di programmazione è quello delle *grammatiche libere* (da contesto), in inglese *context-free grammar*.

Un pezzo di grammatica del C

```
iteration_statement
    : while '(' expression ')' statement
    | 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 '}'
    ;
```

Grammatiche formali

- Come le espressioni regolari, anche le *grammatiche formali* (da qui in avanti semplicemente *grammatiche*), sono uno strumento per la descrizione di linguaggi.
- Una grammatica è un formalismo *generativo* perché il linguaggio da essa definito coincide con l'insieme delle stringhe che possono essere "generate" usando determinate regole che fanno parte della grammatica stessa.
- Le grammatiche possono avere diversi *gradi di espressività*, e dunque definire linguaggi più o meno ricchi.
- Esiste però un forte compromesso fra espressività e possibilità di riconoscimento automatico, che vedremo ben rappresentato nel caso caso dei linguaggi di programmazione.

Definizione formale di grammatica

- Diamo ora la definizione generale di grammatica (formale).
- Una grammatica G è una quadrupla di elementi:

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

  dove
  - $\mathcal{N}$ è un insieme di simboli, detti *non terminali*;
  - $\mathcal{T}$ è un insieme di simboli *terminali*, $\mathcal{N} \cap \mathcal{T} = \Phi$;
  - $\mathcal{P}$ è l'insieme delle *produzioni*, cioè scritture della forma $X \to Y$, dove $X, Y \in (\mathcal{N} \cup \mathcal{T})^*$;
  - $\mathcal{S} \in \mathcal{N}$ è il *simbolo iniziale* (o *assioma*).
- Conviene anche definire l'insieme $\mathcal{V} = \mathcal{N} \cup \mathcal{T}$ come il *vocabolario* della grammatica.

Le produzioni

- La forma 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 grammatiche regolari descrivono proprio i linguaggi regolari che già conosciamo.

Le produzioni

- 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

$$A \rightarrow X$$

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

Derivazioni

- Il meccanismo in base al quale le grammatiche "generano" linguaggi è quello delle derivazioni.
- Una *derivazione* è il processo mediante il quale, a partire dall'assioma ed applicando una sequenza di produzioni, si ottiene una stringa di $\mathcal{T}^*$, cioè una stringa composta da soli terminali.
- Le produzioni rappresentano infatti vere e proprie *regole di riscrittura*.
- Ad esempio, una produzione del tipo

$$E \rightarrow E{+}E$$

si può leggere nel seguente modo: il simbolo E può essere "riscritto" come $E{+}E$.

Derivazioni (continua)

- L'idea è che una grammatica descriva (generi) il linguaggio costituito proprio dalle sequenze di simboli terminali derivabili a partire dall'assioma $\mathcal{S}$.
- Consideriamo, ad esempio, la grammatica $G_5 = (\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$ così definita:
  - $\mathcal{N} = \{S, A, B\}$;
  - $\mathcal{T} = \{\mathtt{a}, \mathtt{b}\}$;
  - $\mathcal{S} = S$;
  - $\mathcal{P}$ contiene le seguenti produzioni:

$$\begin{array}{lcl} S & \rightarrow & \epsilon, \quad S \rightarrow A \\ A & \rightarrow & \mathtt{a}, \quad A \rightarrow \mathtt{a}A, \quad A \rightarrow B \\ B & \rightarrow & \mathtt{b}B, \quad B \rightarrow \mathtt{b} \end{array}$$

- Nel linguaggio generato da G_5 è inclusa la stringa $\mathtt{ab}$ perché:

$$S \Rightarrow A \Rightarrow \mathtt{a}A \Rightarrow \mathtt{a}B \Rightarrow \mathtt{ab}.$$

Nota sulla notazione...

- La scrittura $\alpha \Rightarrow \beta$ (dove $\alpha, \beta \in \mathcal{V}^*$) indica che β può essere ottenuta direttamente da α mediante l'applicazione di una produzione della grammatica.
- Ad esempio, con riferimento alla derivazione del lucido precedente, $\mathtt{a}A \Rightarrow \mathtt{a}B$ perché nella grammatica G_5 è presente la produzione $A \rightarrow B$.
- Non sarebbe stato corretto scrivere $\mathtt{a}A \rightarrow \mathtt{a}B$ (perché non esiste una tale produzione).
- Se α deriva β mediante l'applicazione di 0 o più produzioni si scrive $\alpha \stackrel{*}{\Rightarrow}_G \beta$.
- Ad esempio, in G_5, $\mathtt{a}A \stackrel{*}{\Rightarrow}_G \mathtt{ab}$.

Descrizione succinta di una grammatica

- Una grammatica può essere espressa in modo più succinto elencando le sole produzioni, qualora si convenga che le prime produzioni in elenco siano quelle relative all'assioma.
- Ad esempio, scrivendo

$$\begin{array}{lcl} E & \rightarrow & E+E \\ E & \rightarrow & E*E \\ E & \rightarrow & (E) \\ E & \rightarrow & \texttt{id} \end{array}$$

  intendiamo la grammatica $G_1 = (\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$ in cui:
  - $\mathcal{N} = \{E\}$;
  - $\mathcal{T} = \{\texttt{id}, +, *, (,)\}$;
  - $\mathcal{S} = E$;

  e le produzioni sono ovviamente quelle indicate.

Altri esempi di derivazione

Compilatori
Funzioni e struttura
Analisi lessicale: token, pattern e lessemi
Analisi sintattica
Parsing e grammatiche libere

Consideriamo la grammatica G_1 appena introdotta. Allora:

- $E{+}E \Rightarrow_{G_1} \mathtt{id}{+}E$ tramite l'applicazione della produzione $E \rightarrow \mathtt{id}$ alla prima occorrenza di E.
- $E{+}E \stackrel{*}{\Rightarrow}_{G_1} \mathtt{id} + \mathtt{id}$ tramite l'applicazione della produzione $E \rightarrow \mathtt{id}$ ad entrambe le occorrenze di E.
- $E \stackrel{*}{\Rightarrow}_{G_1} \mathtt{id} + (E)$ in quanto $E \Rightarrow_{G_1} E{+}E \Rightarrow_{G_1} E{+}(E) \Rightarrow_{G_1} \mathtt{id} + (E)$.
- $E \stackrel{*}{\Rightarrow}_{G_1} \mathtt{id} + (\mathtt{id})$, in quanto $E \Rightarrow_{G_1} E{+}E \Rightarrow_{G_1} E{+}(E) \Rightarrow_{G_1} \mathtt{id} + (E) \Rightarrow_{G_1} \mathtt{id} + (\mathtt{id})$.
- Una derivazione alternativa per $\mathtt{id} + (\mathtt{id})$ è $E \Rightarrow_{G_1} E{+}E \Rightarrow_{G_1} \mathtt{id}{+}E \Rightarrow_{G_1} \mathtt{id} + (E) \Rightarrow_{G_1} \mathtt{id} + (\mathtt{id})$.

Descrizione succinta e metasimboli

- Possiamo economizzare ancora sulla descrizione di una grammatica “fondendo” 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_5 può essere descritta nel seguente modo compatto:

$$
\begin{array}{lcl}
S & \rightarrow & \epsilon \mid A \\
A & \rightarrow & \mathtt{a} \mid \mathtt{a}A \mid B \\
B & \rightarrow & \mathtt{b}B \mid \mathtt{b}
\end{array}
$$

- Si noti come nella descrizione di una grammatica si utilizzino simboli che non sono terminali ne’ non terminali, come ad esempio $\rightarrow$ e $\mid$.
- Tali simboli prendono il nome di *metasimboli*.

Frasi e linguaggio generato

Compilatori
Funzioni e struttura
Analisi lessicale: token, pattern e lessemi
Analisi sintattica
Parsing e grammatiche libere

Sia $G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$ una grammatica.

- Si chiama *forma di frase* di G una qualunque stringa α di $\mathcal{V}^*$ tale che $\mathcal{S} \stackrel{*}{\Rightarrow}_G \alpha$.
- Se $\alpha \in \mathcal{T}^*$ allora si dice che α è anche una *frase* di G.
- Dagli esempi precedenti possiamo conlcudere (ad esempio) che:
  - le stringhe $\texttt{id} + (E)$ e $\texttt{id} + (\texttt{id})$ sono forme di frase di G_1;
  - le stringhe $\texttt{ab}B$, $\texttt{abb}B$ e $\texttt{ab}$ sono forme di frase di G_5;
  - $\texttt{id} + (\texttt{id})$ e $\texttt{ab}$ sono anche frasi;
  - $\texttt{ba}A$ non è una forma di frase di G_5.
- Il linguaggio generato da G, spesso indicato con $L(G)$, è l'insieme delle frasi di G.

Esempi

- La grammatica G_5 genera il linguaggio $L_5 = \{\mathtt{a}^n\mathtt{b}^m | n, m \geq 0\}$.
- La grammatica G_1 genera il linguaggio delle espressioni aritmetiche composte da +, $\ast$, (,) e id.
- Le stringhe più corte in $L(G_1)$ sono $\mathtt{id}, \mathtt{id} + \mathtt{id}, \mathtt{id} \ast \mathtt{id}, (\mathtt{id})$.

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 solitamente per induzione.
- Dapprima si formula una congettura sulla forma delle frasi di $L(G)$, provando alcune semplici derivazioni.
- Dopodiché si dimostra separatamente che:
  - se X è generata dal linguaggio, allora X ha la particolare forma congetturata;
  - se X 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_5 genera il linguaggio $L_5 = \mathtt{a}^*\mathtt{b}^*$.
- Ricordiamo che G_5 è:

$$\begin{array}{lcl} S & \rightarrow & \epsilon \mid A \\ A & \rightarrow & \mathtt{a} \mid \mathtt{a}A \mid B \\ B & \rightarrow & \mathtt{b}B \mid \mathtt{b} \end{array}$$

- G_5 genera "solo" stringhe del tipo $\mathtt{a}^n\mathtt{b}^m$.
  - $S \Rightarrow \epsilon$
  - $S \Rightarrow A \stackrel{k}{\Rightarrow} \mathtt{a}^k A \Rightarrow \mathtt{a}^{k+1}, \qquad k \geq 0$
  - $S \Rightarrow A \stackrel{k}{\Rightarrow} \mathtt{a}^k A \Rightarrow \mathtt{a}^k B \stackrel{h}{\Rightarrow} \mathtt{a}^k \mathtt{b}^h B \Rightarrow \mathtt{a}^k \mathtt{b}^{h+1}$, $h, k \geq 0$
- G_5 genera "tutte" le stringhe del tipo $\mathtt{a}^n\mathtt{b}^m$.
  - Segue dall'arbitrarietà di k e h sopra.

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 ($i = 0, .., 2$), 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_5 è regolare perché generato da una grammatica lineare, come abbiamo appena dimostrato.
- Il linguaggio $L(G_1)$ è invece al più libero perché generabile da una grammatica libera.

Tipi dei linguaggi

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

$$\begin{array}{rcl} S & \rightarrow & \epsilon \mid R \\ R & \rightarrow & \mathtt{b} \mid \mathtt{c} \mid \mathtt{ab} \mid RR \end{array}$$

- In realtà L è regolare, in quanto definibile anche mediante l'espressione regolare $(\mathbf{b} + \mathbf{c} + \mathbf{ab})^*$.
- Il linguaggio L_{11} su $\{\mathtt{a}, \mathtt{b}\}$ costituito da tutte le stringhe α palindrome (cioè tali che $\alpha^R = \alpha$) è libero in quanto generabile dalla grammatica

$$S \rightarrow \mathtt{a}S\mathtt{a}, S \rightarrow \mathtt{b}S\mathtt{b}, S \rightarrow \mathtt{a}, S \rightarrow \mathtt{b}, S \rightarrow \epsilon$$

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

Un esempio più complesso: le e.r. come linguaggio libero

- Come sappiamo, le espressioni regolari sono un formalismo per definire linguaggi.
- Ad esempio, l'espressione regolare $\mathtt{0}(\mathtt{0}+\mathtt{1})^*$, 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 $\mathtt{0}(\mathtt{0}+^*\mathtt{1}$ è ben formata? Potremmo procedere alla costruzione dell'automa?

Un esempio più complesso: le e.r. come linguaggio libero

- La grammatica G_2 così definita

$$
\begin{array}{lcl}
E & \rightarrow & T \mid T+E \\
T & \rightarrow & F \mid FT \\
F & \rightarrow & (E) \mid (E)^{*} \mid A \mid A^{*}, \\
A & \rightarrow & \mathtt{0} \mid \mathtt{1}
\end{array}
$$

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

Esempio (continua)

- La stringa (epressione regolare) `0(0+1)`* appartiene a $L(G_2)$ in quanto esiste la derivazione:

$$
\begin{array}{lll}
E & \Rightarrow_{G_2} T & \\
& \Rightarrow_{G_2} FT & \text{Usando } T \rightarrow FT \\
& \Rightarrow_{G_2} FF & \text{Usando } T \rightarrow F \\
& \Rightarrow_{G_2} AF & \text{Usando } F \rightarrow A \\
& \Rightarrow_{G_2} A(E)^* & \text{Usando } F \rightarrow (E)^* \\
& \Rightarrow_{G_2} \mathtt{0}(E)^* & \text{Usando } A \rightarrow \mathtt{0} \\
& \Rightarrow_{G_2} \mathtt{0}(T+E)^* & \text{Usando } E \rightarrow T+E \\
& \Rightarrow_{G_2} \mathtt{0}(T+T)^* & \text{Usando } E \rightarrow T \\
& \Rightarrow_{G_2} \mathtt{0}(F+T)^* & \text{Usando } T \rightarrow F \\
& \Rightarrow_{G_2} \mathtt{0}(F+F)^* & \text{Usando } T \rightarrow F \\
& \Rightarrow_{G_2} \mathtt{0}(A+F)^* & \text{Usando } F \rightarrow A \\
& \Rightarrow_{G_2} \mathtt{0}(A+A)^* & \text{Usando } F \rightarrow A \\
& \Rightarrow_{G_2} \mathtt{0}(\mathtt{0}+A)^* & \text{Usando } A \rightarrow \mathtt{0} \\
& \Rightarrow_{G_2} \mathtt{0}(\mathtt{0}+\mathtt{1})^* & \text{Usando } A \rightarrow \mathtt{1}
\end{array}
$$

Backus-Naur Form (BNF)

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

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

Altri esempi

- Usando la BNF la grammatica G_1 verrebbe descritta dalle seguenti produzioni

$$\begin{aligned}
\langle \text{espressione} \rangle &\rightarrow \langle \text{espressione} \rangle + \langle \text{espressione} \rangle \mid \\
\langle \text{espressione} \rangle &\rightarrow \langle \text{espressione} \rangle * \langle \text{espressione} \rangle \mid \\
\langle \text{espressione} \rangle &\rightarrow (\langle \text{espressione} \rangle) \mid \text{id}
\end{aligned}$$

- Una grammatica per le chiamate di procedura in Java

$$\begin{aligned}
\langle \text{chiamata} \rangle &\rightarrow \text{id}(\langle \text{parametri-opzionali} \rangle) \\
\langle \text{parametri-opzionali} \rangle &\rightarrow \langle \text{lista-di-parametri} \rangle \mid \epsilon \\
\langle \text{lista-di-parametri} \rangle &\rightarrow \langle \text{lista-di-parametri} \rangle, \langle \text{parametro} \rangle \mid \\
&\quad\ \langle \text{parametro} \rangle
\end{aligned}$$

Altre convenzioni

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

$$
\begin{array}{lll}
\langle \text{comando if} \rangle & \rightarrow & \text{if} \, \langle \text{espressione booleana} \rangle \, \text{then} \\
& & \qquad \langle \text{lista di comandi} \rangle \\
& & [\, \text{else} \\
& & \qquad \langle \text{lista di comandi} \rangle \,] \\
& & \text{endif}
\end{array}
$$

- 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 convenzioni (continua)

- 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. Le variabili sintattiche continuano ad essere scritte in corsivo.
- Ad esempio, potremo scrivere

$$\begin{array}{lll} \textit{comando if} & \rightarrow & \textbf{if}\ \textit{espressione booleana}\ \textbf{then} \\ & & \quad \textit{lista di comandi} \\ & & [\,\textbf{else} \\ & & \quad \textit{lista di comandi}\,] \\ & & \textbf{endif} \end{array}$$

Altre convenzioni (continua)

- 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 → "{" { *dichiarazione* } { *comando* } "}"

Qualche esercizio

- 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).
- Fornire una grammatica libera per il linguaggio $L_{12} = \{\mathtt{a}^n\mathtt{b}^{2n} | n \geq 0\}$ sull'alfabeto $\{\mathtt{a}, \mathtt{b}\}$.
- Si consideri la seguente grammatica G_I

$$\begin{array}{rcl} S & \rightarrow & I \mid A \\ I & \rightarrow & \textbf{if } B \textbf{ then } S \mid \textbf{if } B \textbf{ then } S \textbf{ else } S \\ A & \rightarrow & \mathtt{a} \\ B & \rightarrow & \mathtt{b} \end{array}$$

e si fornisca una derivazione per la stringa

if b **then** **if** b **then** a **else** a

Qualche esercizio

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

$$\begin{array}{lcl} A & \rightarrow & \mathtt{b}\, B\, \mathtt{a} \mid \mathtt{a} \mid \mathtt{b} \\ B & \rightarrow & \mathtt{b}\, A \mid A\, \mathtt{a} \end{array}$$

  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:

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

- Si scriva una grammatica libera per generare il linguaggio $\mathcal{L}$ definito dalla seguente espressione regolare sull'alfabeto $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$: $\mathtt{ba}^*(\mathtt{b} + \mathtt{c})\mathtt{a}^*\mathtt{c}$