Linguaggi dinamici

Corso di Laurea in Informatica

A.A. 2019/2020

Linguaggi dinamici

Analisi sintattica

- Parsing e grammatiche libere
- Alberi di derivazione (parse tree) e ambiguità
- Abstract Syntax Tree

Linguaggi dinamici

Analisi sintattica

Parsing e grammatiche libere

Alberi di derivazione (parse tree) e ambiguità

Abstract Syntax Tree

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*.

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.

Un pezzetto della sintassi di Python 3.8


```
stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLI
small_stmt: (expr_stmt | del_stmt | pass_stmt |
             flow_stmt | import_stmt | global_stmt |
             nonlocal_stmt | assert_stmt)
expr_stmt: testlist_star_expr (annassign |
           augassign (yield_expr|testlist) |
            [('=' (yield_expr|testlist_star_expr))+
             [TYPE_COMMENT]] )
annassign: ':' test ['=' (yield_expr|
                          testlist_star_expr)]
```

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: $A \to xB$ oppure $A \to x$, dove $x \in T$ e $A, B \in \mathcal{N}$, la grammatica è detta *lineare destra*.
- Se invece le produzioni sono del tipo: $A \to Bx$ oppure $A \to x$, 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:

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

- 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 $\alpha \overset{*}{\Rightarrow}_G \beta$.
- Ad esempio, in G_5, $\mathtt{a}A \overset{*}{\Rightarrow}_G \mathtt{ab}$.

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

$$\begin{aligned} E &\rightarrow E+E \\ E &\rightarrow E*E \\ E &\rightarrow (E) \\ E &\rightarrow \textbf{number} \end{aligned}$$

intendiamo la grammatica $G_1 = (\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$ in cui:

- $\mathcal{N} = \{E\}$;
- $\mathcal{T} = \{\textbf{number}, +, *, (,)\}$;
- $\mathcal{S} = E$;

e le produzioni sono ovviamente quelle indicate.

Altri esempi di derivazione

Consideriamo la grammatica G_1 appena introdotta. Allora:

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

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 $X \to Y|Z$ al posto di $X \to Y$ e $X \to Z$.
- Usando anche questa convenzione, la grammatica G_5 può essere descritta nel seguente modo compatto:

$$\begin{aligned} S &\to \epsilon \mid A \\ A &\to \mathtt{a} \mid \mathtt{a}A \mid B \\ B &\to \mathtt{b}B \mid \mathtt{b} \end{aligned}$$

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

Frasi e linguaggio generato

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 $\mathcal{S} \overset{*}{\Rightarrow}_G \alpha$.
- 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 **number** $+$ (E) e **number** $+$ (**number**) sono forme di frase di G_1;
  - le stringhe `ab`B, `abb`B e `ab` sono forme di frase di G_5;
  - **number** $+$ (**number**) e `ab` sono anche frasi;
  - `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 `+`, `*`, `(`, `)` e `number`.
- Le stringhe più corte in $L(G_1)$ sono **number**, **number** $+$ **number**, **number** $*$ **number**, (**number**).

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

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

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

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

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

  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 $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$: $\mathtt{b}\mathtt{a}^*(\mathtt{b} + \mathtt{c})\mathtt{a}^*\mathtt{c}$

Linguaggi dinamici

Analisi sintattica

Equivalenza di grammatiche

- Per definire un linguaggio si possono usare grammatiche equivalenti (cioè che generano lo stesso insieme di stringhe terminali) anche molto diverse.
- Ad esempio, le seguenti grammatiche sono equivalenti:

$$\begin{array}{lcl} E & \rightarrow & E{+}E \mid E{*}E \\ E & \rightarrow & (E) \mid \texttt{number} \end{array} \qquad \begin{array}{lcl} E & \rightarrow & E{+}T \mid T \\ T & \rightarrow & T{*}F \mid F \\ F & \rightarrow & \texttt{number} \mid (E) \end{array}$$

- Quale delle due è “migliore”?

Quale grammatica usare?

- L'obiettivo è di costruire un riconoscitore (parser), quindi il criterio deve essere la maggior semplicità (se non proprio la fattibilità) del parsing
- I parser, infatti, devono essenzialmente riconoscere una stringa attraverso il processo (diretto o inverso) di derivazione dall'assioma iniziale
- Un criterio importante è che non ci siano "troppi" modi diversi per derivare una stessa stringa
- Questo criterio ha a che vedere col concetto di *ambiguità* di una grammatica

Parse tree (alberi di derivazione)

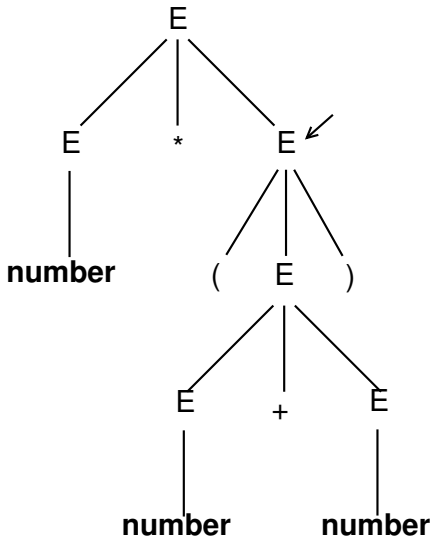
- Che cosa è un *parse tree* per una stringa del linguaggio?
- Si tratta di un albero che evidenza non solo la derivabilità della stringa dall'assioma iniziale, ma che impone pure una "struttura" alla derivazione e alla stringa stessa
- Tale struttura, come vedremo, è anche il primo passaggio del processo che conferisce un *significato operazionale* alla stringa

Parse tree (cont.)

- Formalmente, 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 $X_1, \ldots, X_k$ 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 $A \to X_1 X_2 \ldots X_k$;
  - 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 $A \to \epsilon$.

Parse tree (cont.)

- Ad esempio, consideriamo la stringa $3 * (5 + 7)$ e la grammatica G_1
- Un lexer per G_1 provvederà a trasformare la stringa in qualcosa del tipo:
  $\langle \textbf{number}, \textbf{3} \rangle * (\langle \textbf{number}, \textbf{5} \rangle + \langle \textbf{number}, \textbf{7} \rangle)$
- Dal punto di vista dell'analisi sintattica solo il token type (**number**) ha importanza
- Un esempio di parse tree (l'unico, in realtà per questa stringa rispetto a G_1) è dunque



Parse tree (cont.)

- Un parse tree non è in corrispondenza 1-1 con le derivazioni
- Vale infatti (banalmente) una relazioni di tipo 1-a-molti: un parse tree ha associate più derivazioni possibili
- Esempio relativo alla "solita" stringa (già tokenizzata) **number** $*$ (**number** $+$ **number**) e la grammatica G_1

$$\begin{aligned} E \quad \Rightarrow \quad & E*E \Rightarrow \textbf{number}*E \Rightarrow \textbf{number}*(E) \\ & \Rightarrow \textbf{number}*(E+E) \Rightarrow \textbf{number}*(\textbf{number}+E) \\ & \Rightarrow \textbf{number}*(\textbf{number}+\textbf{number}) \end{aligned}$$

e

$$\begin{aligned} E \quad \Rightarrow \quad & E*E \Rightarrow E*(E) \Rightarrow E*(E+E) \\ & E*(E+\textbf{number}) \Rightarrow E*(\textbf{number}+\textbf{number}) \\ & \Rightarrow \textbf{number}*(\textbf{number}+\textbf{number}) \end{aligned}$$

Alberi di derivazione (parse tree)

- Una stringa ammette derivazioni *ambigue* (in una data grammatica G) se può essere ottenuta con due derivazioni cui corrispondono parse tree diversi
- Se esiste una stringa con derivazioni ambigue allora la grammatica si dice *ambigua*
- Si noti che la stringa **number** $*$ (**number** $+$ **number**) non ammette derivazioni ambigue in G_1, ma nonostante ciò G_1 è una grammatica ambigua (si provi a dimostrarlo usando una stringa differente)

Derivazioni canoniche

- Se una grammatica non è ambigua, ogni stringa del linguaggio da essa generato ha associato un solo parse tree
- Il fatto (ineliminabile) che comunque esistono più derivazioni corrispondenti allo stesso albero ha importanza secondaria
- Se infatti l'albero è uno, possiamo comunque limitarci a considerare solo derivazioni in qualche modo "normalizzate" o, come correttamente si dice, *canoniche*
- In una derivazione canonica, si impone che il non-terminale da riscrivere sia univocamente determinato come quello più a destra ovvero quello più a sinistra
- Corripondentemente si parla di derivazioni canoniche destre o sinistre

Parse tree e significato

- Come già sottolineato, i parse tree impongono una struttura sintattica gerarchica alle frasi
- Non è difficile intuire che questo è un passo decisivo per attribuire un “senso” ad una frase
- In questo contesto, e in modo evidente per linguaggi dinamici come il PERL, il senso è precisamente ciò che la frase impone di fare all’interprete
- Ad esempio, se osserviamo nuovamente il parse tree per $\langle \textbf{number}, \textbf{3} \rangle * (\langle \textbf{number}, \textbf{5} \rangle + \textbf{number}, \textbf{7} \rangle)$, la struttura “suggerisce” che l’interprete dovrà prima eseguire la somma di 5 e 7 e solo dopo sommare il risultato al valore 3
- Se i parse tree fossero più d’uno, non sarebbe quindi possibile associare un significato univoco ad una frase del linguaggio, come vedremo meglio nelle prossime slide

Problemi legati all'ambiguità

- Sempre in relazione alla grammatica G_1, consideriamo ora la frase: $\langle \mathbf{number}, \mathbf{3} \rangle + \langle \mathbf{number}, \mathbf{5} \rangle * \mathbf{number}, \mathbf{7} \rangle$
- Essa può essere ottenuta con due differenti derivazioni canoniche destre, e precisamente:

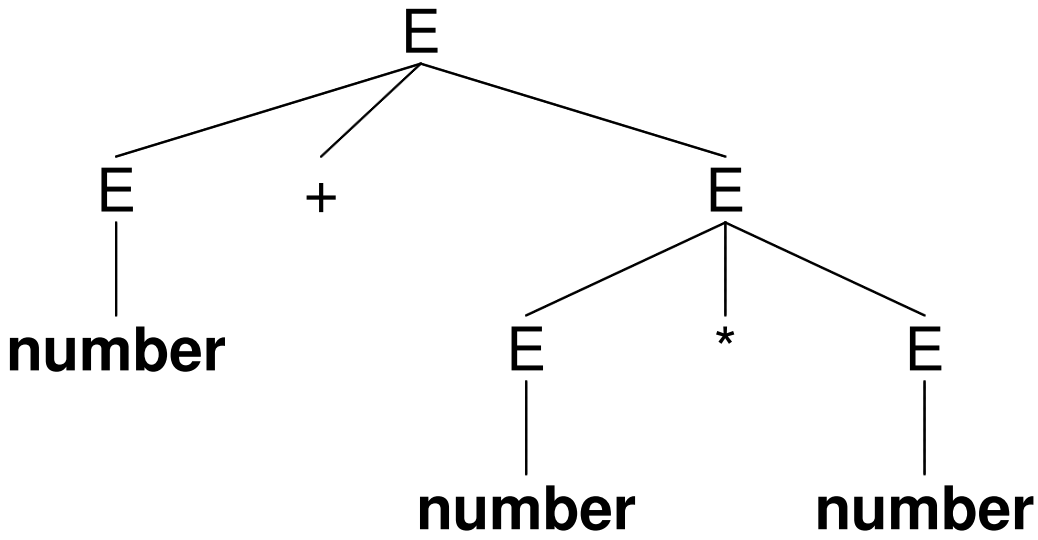
$$
\begin{aligned}
E &\Rightarrow E+E \\
&\Rightarrow E+E*E \\
&\Rightarrow E+E*\mathbf{number} \\
&\Rightarrow E+\mathbf{number}*\mathbf{number} \\
&\Rightarrow \mathbf{number}+\mathbf{number}*\mathbf{number}
\end{aligned}
$$

e

$$
\begin{aligned}
E &\Rightarrow E*E \\
&\Rightarrow E*\mathbf{number} \\
&\Rightarrow E+E*\mathbf{number} \\
&\Rightarrow E+\mathbf{number}*\mathbf{number} \\
&\Rightarrow \mathbf{number}+\mathbf{number}*\mathbf{number}
\end{aligned}
$$

Problemi legati all'ambiguità (cont.)

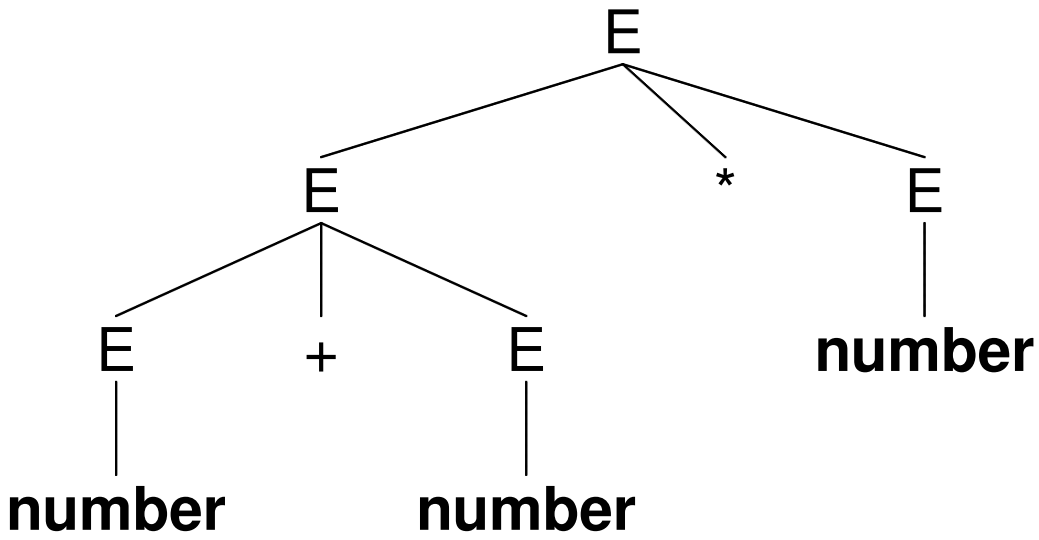
- Alla prima derivazione corrisponde il seguente parse tree:



- Come si vede l'albero "suggerisce" l'interpretazione corretta in base alla precedenza degli operatori, cioè prima la moltiplicazione e poi l'addizione

Problemi legati all'ambiguità (cont.)

- Alla seconda derivazione corrisponde il parse tree:



e questo suggerisce invece un senso che non è quello corretto, perché viola le regole di precedenza.

- In questo caso, infatti, l'interprete viene portato ad eseguire prima l'addizione e poi la moltiplicazione

Problemi legati all'ambiguità (cont.)

- (Ri)consideriamo il frammento di grammatica:

$$\begin{aligned} S &\rightarrow I \mid \ldots \\ I &\rightarrow \textbf{if } B \textbf{ then } S \mid \textbf{if } B \textbf{ then } S \textbf{ else } S \\ B &\rightarrow \ldots \end{aligned}$$

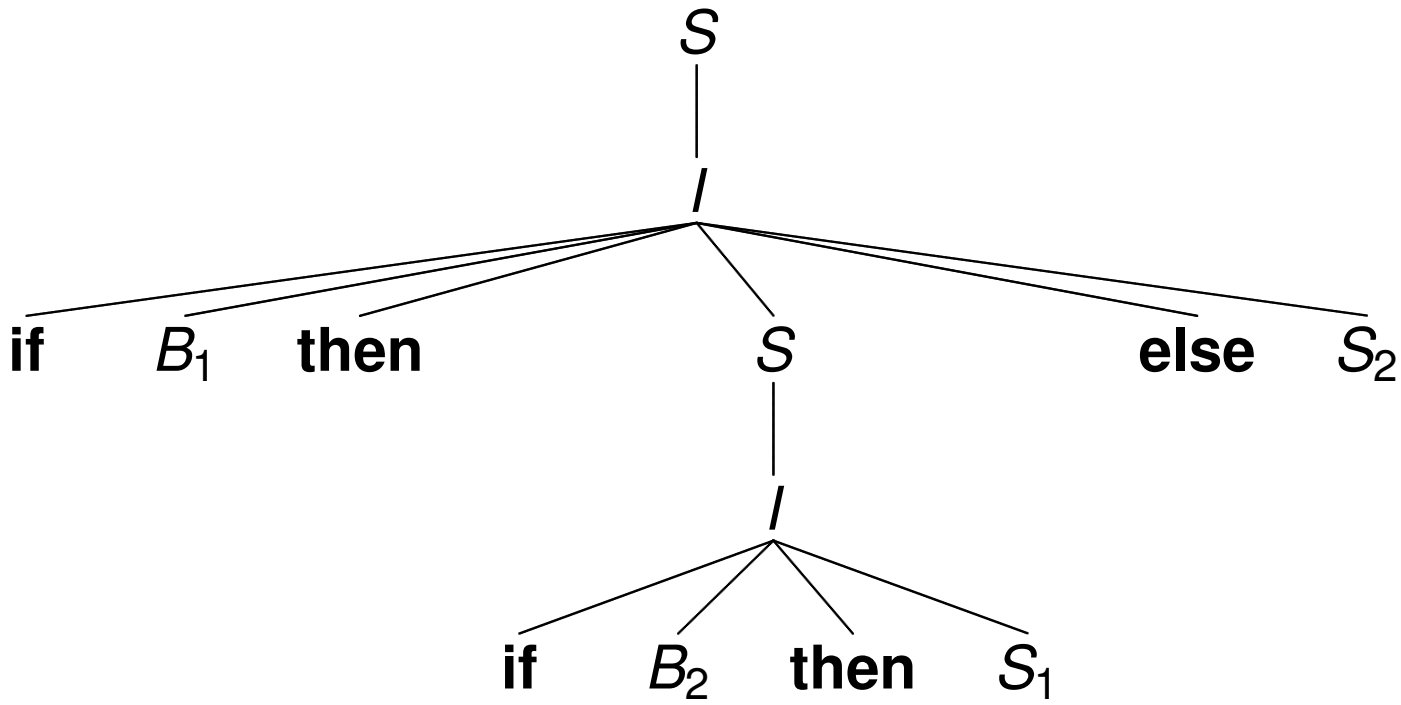
- Consideriamo ora la forma di frase:

$$\textbf{if } B_1 \textbf{ then if } B_2 \textbf{ then } S_1 \textbf{else } S_2$$

  dove i pedici ai simboli non terminali servono solo per potervi fare riferimento.
- Sulla base della produzione per I, la forma di frase ammette due derivazioni canoniche destre, e dunque due parse tree, differenti

Problemi legati all'ambiguità (cont.)

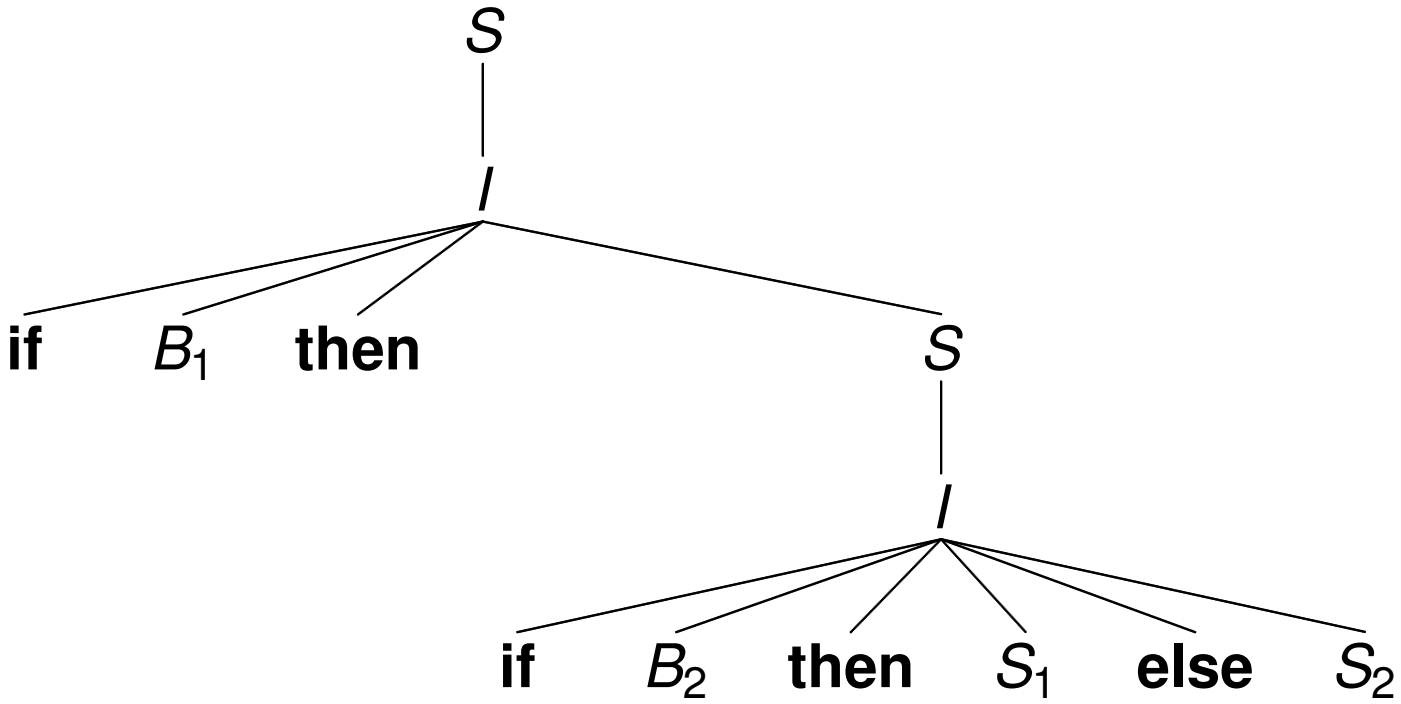
- Una prima derivazione è caratterizzata dal seguente parse tree (parziale):



- Questo albero non corrisponde però alla classica intepretazione della frase comune a tutti i linguaggi.
- Ad esempio, se il valore di B_1 è falso, non dovrebbe essere eseguito nessuno degli statement S_1 ed S_2
- Il parse tree suggerisce invece che, se B_1 è falso, allora viene eseguito S_2.

Problemi legati all'ambiguità (cont.)

- La seconda derivazione possibile è invece quella corretta:



Precedenza degli operatori

- Come sappiamo, la grammatica G_1:

$$E \rightarrow E+E \mid E*E \mid \textbf{number} \mid (E)$$

è ambigua

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

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

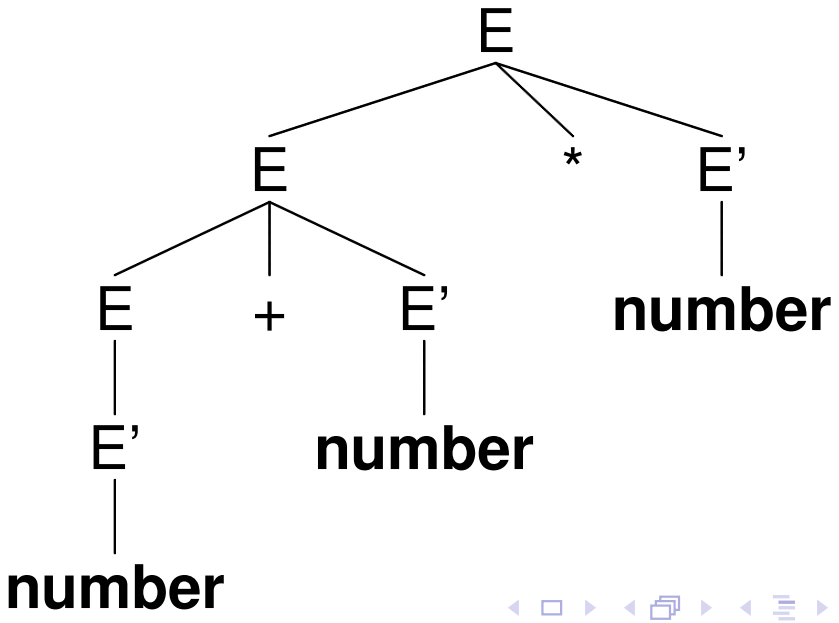
- La nuova grammatica non è più ambigua ma, come anche la stessa grammatica G_1 di partenza, presenta un problema legato alla precedenza degli operatori.

Precedenza degli operatori

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

$$
\begin{aligned}
E &\Rightarrow E * E' \Rightarrow E * \textbf{number} \\
&\Rightarrow E + E' * \textbf{number} \Rightarrow E + \textbf{number} * \textbf{number} \\
&\Rightarrow \textbf{number} + \textbf{number} * \textbf{number}
\end{aligned}
$$

alla quale corrisponde il parse tree "sbagliato":



Precedenza degli operatori (continua)

- 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 catogoria delle espressioni di base (nel nostro caso identificatori ed espressioni tra parentesi):

$$\begin{aligned} E &\rightarrow E+T \mid T \\ T &\rightarrow T*F \mid F \\ F &\rightarrow \textbf{number} \mid (E) \end{aligned}$$

Precedenza degli operatori (continua)

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

$$\begin{aligned} E &\rightarrow E+T \mid E-T \mid T \\ T &\rightarrow T*F \mid T/F \mid F \\ F &\rightarrow \textbf{number} \mid (E) \end{aligned}$$

- Domanda: perché scriviamo (ad esempio)

$$T \rightarrow T*F$$

e non, invece,

$$T \rightarrow F*T \quad ?$$

Precedenza degli operatori (continua)

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

$$
\begin{aligned}
E &\rightarrow E+T \mid E-T \mid T \\
T &\rightarrow T*P \mid T/P \mid P \\
P &\rightarrow F\hat{}P \mid F \\
F &\rightarrow \textbf{number} \mid (E)
\end{aligned}
$$

Esercizi

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

$$\begin{aligned} S \;\to\; & \mathtt{a} \mid \mathtt{a}S \mid S\mathtt{a} \mid \mathtt{ab}S \mid \mathtt{a}S\mathtt{b} \mid S\mathtt{ab} \mid \\ & \mathtt{ba}S \mid \mathtt{b}S\mathtt{a} \mid S\mathtt{ba} \end{aligned}$$

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

$$\begin{aligned} S &\to \mathtt{a}A \\ A &\to AB \mid B \\ B &\to \mathtt{b} \end{aligned}$$

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):

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

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*

Linguaggi dinamici

Analisi sintattica

Abstract syntax tree

- Un *abstract syntax tree* (*AST*) per un linguaggio L è un albero in cui:
  - i nodi interni rappresentano costrutti di L;
  - i figli di un nodo che rappresenta un costrutto $\mathcal{C}$ rappresentano a loro volta le "componenti significative" di $\mathcal{C}$;
  - le foglie sono "costrutti elementari" (non ulteriormente decomponibili) caratterizzati da un *valore lessicale* (tipicamente un numero o un puntatore alla symbol table).
- Le diapositive seguenti illustrano la nozione di abstract syntax tree.

Esempio

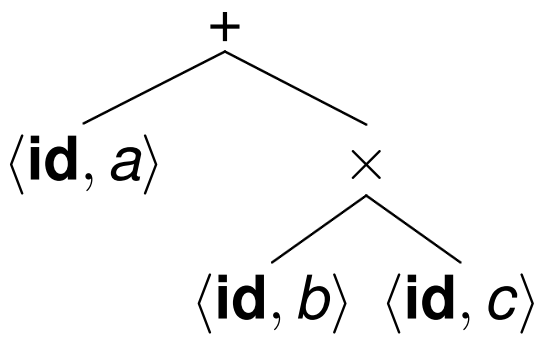
- Un abstract syntax tree per la “frase” (espressione aritmetica)

$$a + b * c$$

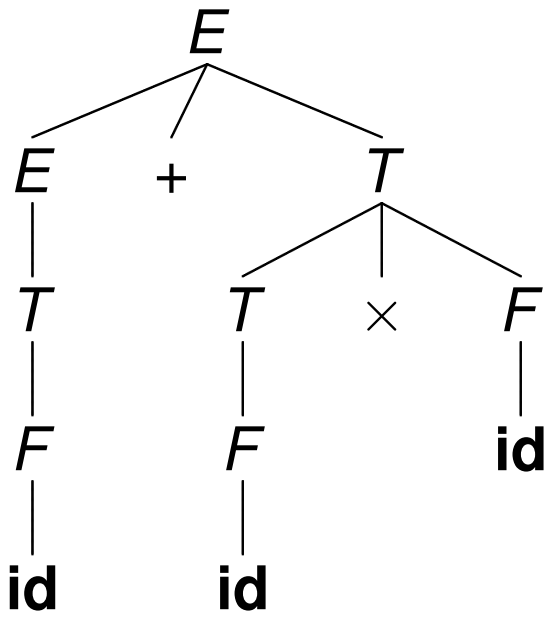
dopo l’opportuna “tokenizzazione”

$$\mathbf{id} + \mathbf{id} \times \mathbf{id}$$

è:

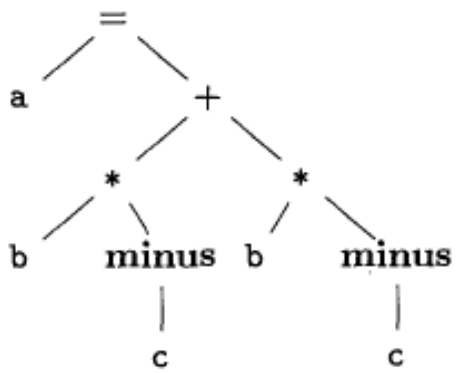


- Si tenga ben presenteil fatto che abstract syntax tree e parse tree sono oggetti diversi:
- Parse tree della precedente espressione (tokenizzata):



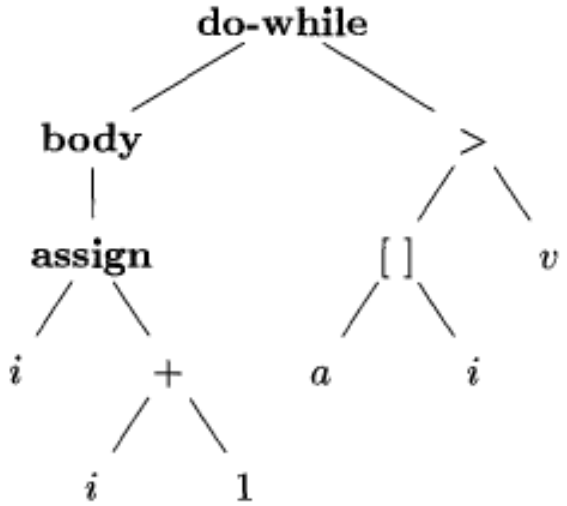
Esempio

- Un AST per l'assegnamento `a=b*(-c)+b*(-c)`:



- Un AST per il comando

do `i=i+1` **while** (`a[i]>v`)



Abstract syntax tree (2)

- L'utilità degli AST è riassumibile nelle seguenti affermazioni.
  - Partendo da un AST la generazione del three address code è un esercizio sufficientemente semplice (anche se l'intero processo è meno efficiente della generazione diretta di codice intermedio).
  - Nella realizzazione di semplici linguaggi intepretati (o comunque di applicazioni dove l'efficienza non sia il principale requisito) gli AST possono rappresentare il risultato ultimo della compilazione.
  - Risulta infatti molto semplice (in generale, e in rapporto alla complessità di realizzare un compilatore completo) implementare un software per interpretare gli AST.

Un semplice interprete per le espressioni aritmetiche

- La diapositiva seguente presenta lo pseudocodice per un semplice interprete di AST che rappresentano espressioni aritmetiche.
- Ogni nodo dell'albero tre campi:
  - un campo etichetta (`label`) che, se il nodo è interno, contiene un codice di operatore (come, ad esempio, `PLUS, TIMES, MINUS, UNARY_MINUS, ...`), se invece il nodo è una foglia contiene un puntatore alla symbol table;
  - un campo puntatore al primo operando (`left`);
  - un campo puntatore all'eventuale secondo operando.
- Lo pseudocodice usa una routine (`apply`) che restituisce il valore dell'applicazione di un operatore binario a due operandi passati come parametri.

Un semplice interprete per le espressioni aritmetiche

EVAL(NODE v)

```
1: if left(v) ≠ nil then
2:     x ← eval(left(v))
3:     if label(v) = UNARY_MINUS then
4:         return −x
5:     y ← eval(right(v))
6:     return apply(label(v), x, y)
7: else
8:     return symlookup(label(v))
```