

Due Algoritmi di Parsing:

- Top-Down: albero creato dalla radice (ANTLR)
- Bottom-Up: albero creato dalle foglie

Algoritmi di Parsing: Top-down Parsing

Slides based on material
by Ras Bodik available at
<http://inst.eecs.berkeley.edu/~cs164/fa04>

Esempio: la variabile più a sinistra è A. Quale produzione scelgo? Le provo tutte una alla volta, fino a che trovo mismatch con stream di token (se no, vado avanti fino ad arrivare alla stringa in input)

Now let's parse a string

Costruisco derivazioni canoniche sinistre (bottom-up: deriv. canoniche destre). Se ho un mismatch, torno allo stato iniziale e ricomincio da capo, oppure ho il match con la stringa in input

- **recursive descent** parsers compute (left-most) derivations by trying each production in turn
 - until there is a mismatch (**backtracking**)
 - or until it matches derived string with the input string

Mancata Corrispondenza (Backtracking): È qui che i parser a discesa ricorsiva semplici possono utilizzare il backtracking (ritorno sui propri passi): Se una produzione viene scelta e il parsing successivo non riesce a corrispondere al resto della stringa di input (si verifica un mismatch), il parser fa backtracking. Il backtracking significa che il parser annulla tutto il lavoro svolto dalla scelta (riportando indietro il puntatore dell'input) e prova la prossima produzione alternativa per il non-terminale corrente.

Corrispondenza (Successo): Il processo continua fino a quando: Il parser deriva con successo una stringa che corrisponde esattamente all'intera stringa di input oppure il parser raggiunge un punto in cui ha provato tutte le produzioni per un non-terminale e nessuna funziona (un errore di sintassi).

Recursive Descent Parsing

- Consider the grammar

(non ambigua: assoc. a destra per il +, assoc. a sinistra per il *; precedenza del * sul +)

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

$$T \rightarrow (E)^* T \mid (E) \mid \text{int}^* T \mid \text{int}$$

- Token stream is:

$\text{int}_5^* \text{int}_2$

int: literal intero

- Start with top-level non-terminal E

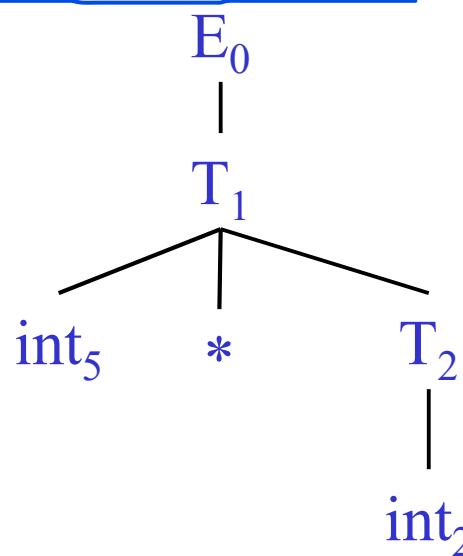
- Try the rules for E in order

Recursive Descent Parsing. Example (Cont.)

- Try $E_0 \rightarrow T_1 + E_1$ Si parte da E e si prova tutte le regole di E
 - Then try a rule for T_1 : $T_1 \rightarrow (E_2)^* T_2$ Prova tutte le regole di T
 - But (does not match) input token int₅
 - Try $T_1 \rightarrow (E_2)$
 - But (does not match) input token int₅
 - Try $T_1 \rightarrow \text{int}^* T_2$
 - Then all rules for T_2 until $T_2 \rightarrow \text{int}$. This matches! Prova tutte le regole di T
 - But + after T_1 will be unmatched
 - Backtrack to choice for E_0
- Ritorna alla scelta delle produzioni per E, cioè prova un'altra produzione per E_0 (al posto di $E_0 \rightarrow T_1 + E_1$, la scarta e prova un'altra produzione $E_0 \rightarrow T_1$)
- ma dopo aver trovato i terminali, ho una stringa finale che non matcha con la stringa in input (backtrack)

Recursive Descent Parsing. Example (Cont.)

- Try $E_0 \rightarrow T_1$
- Follow same steps as before for T_1
 - And succeed with $T_1 \rightarrow \text{int} * T_2$ and $T_2 \rightarrow \text{int}$
 - With the following parse tree



La stringa finale generata è uguale alla stringa in input

A Recursive Descent Parser (2)

- Define boolean functions that check the token string for a match of

- A given token terminal

Controlla che il prossimo token nello stream sia uguale a quello passato come parametro alla funzione

```
bool term(TOKEN tok) { return in[next++] == tok; }
```

- A given production of S (the n^{th})

```
bool  $S_n()$  { ... }
```

La Funzione term() e $S()$ viene data ad ogni produzione (ritorna true se ogni funzione nel predicato ritorna true, cioè ogni sotto stringa fa match)

- Any production of S :

```
bool  $S()$  { ... }
```

Per ogni produzione di ogni variabile non-terminale, si valuta ogni combinazione e si implementa il backtracking

Se c'è anche un solo false, quel percorso di produzioni non vale

- These functions advance next

Le funzioni dei non-terminali sono ricorsive, mentre quelle dei terminali no.

approccio non deterministico gestito attraverso la ricorsione e il backtracking

A Recursive Descent Parser (3)

Affinché $E_1()$ abbia successo, tutte e tre le condizioni devono essere vere e devono avanzare il puntatore next correttamente.

- For production

$$E \rightarrow T + E$$

Da questa singola produzione della variabile non-terminale E , si provano tutte le combinazioni delle variabili e terminali del corpo della produzione

```
bool E1() { return T() && term(PLUS) && E(); }
```

- For production

$$E \rightarrow T$$

Si provano ogni produzione di T

```
bool E2() { return T(); }
```

- For all productions of E (with backtracking)

```
bool E() {
```

```
    int save = next;
```

Si salva la posizione iniziale del puntatore nello stream prima di provare qualsiasi produzione della variabile (l'indice del primo token che E deve analizzare)

```
    return E1()
```

```
    || (next = save, E2()); }
```

Backtracking: Se $E_1()$ restituisce false (fallimento), viene eseguito il blocco successivo: Il puntatore next viene ripristinato alla posizione save (quella iniziale), annullando qualsiasi avanzamento di token che $E_1()$ e le sue chiamate hanno fatto prima di fallire.

$E()$ avrà successo se almeno una delle sue produzioni (E_1 o E_2) ha successo. Il meccanismo di ripristino assicura che il secondo tentativo (E_2) inizi sempre con la stessa porzione di input del primo tentativo (E_1).

A Recursive Descent Parser (4)

- Functions for non-terminal T

```
bool T1() { return term(OPEN) && E() && term(CLOSE) &&  
          term(TIMES) && T(); }  
bool T2() { return term(OPEN) && E() && term(CLOSE); }  
bool T3() { return term(INT) && term(TIMES) && T(); }  
bool T4() { return term(INT); }
```

All'interno di ciascuna, se hanno almeno un non-terminale, si utilizza la ricorsione

All'interno di T, si usa il backtracking

```
bool T() {  
    int save = next;  
    return T1()  
        || (next = save, T2()) || (next = save, T3())  
        || (next = save, T4()); }
```

Recursive Descent Parsing. Notes.

- To start the parser
 - Initialize `next` to point to first token
 - Invoke `E()` (variabile iniziale della grammatica)
- Suppose a special character `$` to be put at the end of input string in the `in[]` array
- Parsing is successful if, at end of execution,
`E()` returns true and `next` points to `$`
- Notice how this simulates our backtracking example from lecture

Recursive Descent Parsing. Notes.

- Easy to implement (also by hand)
- But does not always work ...
non funziona sempre

When Recursive Descent Does Not Work

- Consider a production $S \rightarrow S a$:
 - In the process of parsing S we try the above rule
 - What goes wrong?
- A left-recursive grammar has a non-terminal S (not necessarily the initial one) such that
$$S \Rightarrow^+ S\alpha \quad \text{for some } \alpha$$

qualsiasi cosa
- Recursive descent does not work in such cases
 - It goes into an infinite loop

é considerata tale se si ha una produzione dove la testa ricompare più a sinistra nel corpo della produzione

Elimination of Left Recursion

- Consider the left-recursive grammar

$$S \rightarrow S\alpha \mid \beta \quad \xrightarrow{\text{S va in ricorsione sinistra in modo diretto (cioé in uno step va in ricorsione)}}$$

with α not being ϵ and β not starting with S

- S generates all strings starting with a β and followed by a number of α (esempio: beta alfa alfa alfa)

Come risolvere il problema della ricorsione a sinistra? Converto a Ricorsione a destra

- Can rewrite using right-recursion

$$S \rightarrow \beta S'$$

$$S' \rightarrow \alpha S' \mid \epsilon$$

Elimination of Left-Recursion. Example

- Consider the grammar

$$S \rightarrow S0 \mid 1$$

($\beta = 1$ and $\alpha = 0$)

can be rewritten as

$$S \rightarrow 1 S'$$

$$S' \rightarrow 0 S' \mid \epsilon$$

More Elimination of Left-Recursion

- In general

alfa_i: sono stringhe a destra di S
beta_i: sono stringhe senza S

$$S \rightarrow S\alpha_1 \mid \dots \mid S\alpha_n \mid \beta_1 \mid \dots \mid \beta_m$$

with all of α_i not being ε and all of β_i not starting with S



- All strings derived from S start with one of β_1, \dots, β_m and continue with several instances of $\alpha_1, \dots, \alpha_n$
- Rewrite as

$$S \rightarrow \beta_1 S' \mid \dots \mid \beta_m S'$$

$$S' \rightarrow \alpha_1 S' \mid \dots \mid \alpha_n S' \mid \varepsilon$$

(done automatically by ANTLR4)

General Left Recursion

Questa, rispetto a quelle di primo, sono ricorsioni a sinistra indirette (cioé S va in ricorsione, applicando in modo alternato la produzione di S con quelle di A)

- **The grammar**

$$S \rightarrow A \alpha \mid \delta$$

Ricorsione indiretta

$$A \rightarrow S \beta$$

is also left-recursive because

$$S \Rightarrow^+ S \beta \alpha$$

Quando l'algoritmo analizza A_i , trova una regola che inizia con un simbolo già processato A_j (dove $j < i$):

$$A_i \rightarrow A_j \text{ beta}$$

Per rompere questa dipendenza, l'algoritmo sostituisce A_j con tutte le sue definizioni non ricorsive.

Esempio: Grammatica iniziale:

$$A_1 \rightarrow A_2 a \mid b$$

$$A_2 \rightarrow A_1 c \mid d$$

-> Passo $i=1$ (Analisi di $\$A_1\$$):

A_1 non ha ricorsione a sinistra, né diretta né indiretta con A_j dove $j < 1$. -

-> Passo $i=2$ (Analisi di $\$A_2\$$):

A_2 ha la produzione $A_2 \rightarrow A_1 c$. Poiché $j < i$, dobbiamo sostituire.

- Sostituiamo A_1 in $A_2 \rightarrow A_1 c$:

A_1 si espande in $A_2 a \Rightarrow$ la nuova regola è $A_2 \rightarrow (A_2 a)c$

A_1 si espande in $b \Rightarrow$ la nuova regola è $A_2 \rightarrow bc$

Le regole di A_2 diventano: $A_2 \rightarrow A_2 ac \mid bc \mid d$.

Risultato: La ricorsione indiretta $A_2 \rightarrow A_1 c$ è stata trasformata in una ricorsione a sinistra diretta $A_2 \rightarrow A_2 ac$.

- This left-recursion can also be eliminated with the algorithm in the next slide

In A, rimangono i corpi della produzione che, nelle produzioni iniziali, non avevano A. In A', vanno le produzioni che, nelle produzioni iniziali, avevano A.



Ora che tutta la ricorsione indiretta è stata convertita in ricorsione diretta, si applica la tecnica standard, che utilizza un nuovo simbolo per isolare la parte ricorsiva.

Riscriviamo: Introduciamo un nuovo simbolo A' .

- Regole per A (il corpo): A deve espandersi in una delle parti non ricorsive (\beta), seguite dal nuovo simbolo A' (che gestirà la ricorsione). Nell'esempio: $A_2 \rightarrow bcA_2' \mid da_2'$

- Regole per A' (la coda): A' gestisce la ripetizione (\alpha) o termina la derivazione (\epsilon). Nell'esempio: $A_2' \rightarrow acA_2' \mid \epsilon$

A_j (dove $j < i$) Ricorsione Indiretta: Eliminata (grazie al punto 1 dell'algoritmo: la sostituzione). La regola $A_i \rightarrow A_j \beta$ non esiste più.
 A_i (dove $k=i$) Ricorsione Diretta: Eliminata (grazie al punto 2 dell'algoritmo: la riscrittura con A'). La regola $A_i \rightarrow A_i \alpha$ non esiste più.
 A_k (dove $k > i$) Dipendenza "Futura": Consentita. Non costituisce un problema di ricorsione a sinistra, poiché A_k verrà analizzato solo nei passi successivi.

Risolve sia Ricorsione diretta sia Ricorsione indiretta

Algorithm for Left Recursion Elimination

Lista delle variabili

List non-terminal symbols in order: A_1, A_2, \dots, A_n

Per ogni variabile

For $i := 1$ to n

- Replace all $A_i \rightarrow A_j \beta$ such that $j < i$ (i.e. immediate left-recursion for A_j already eliminated) with $A_i \rightarrow \delta_1 \beta \mid \delta_2 \beta \mid \dots \mid \delta_k \beta$ (where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$)
- Eliminate immediate left recursion for A_i (with the previously presented algorithm)

quindi non ci sono più dipendenze a sinistra da A_j con $j < i$.

After i -th step, all productions $A_i \rightarrow A_k \beta$ are such that $k > i$

(works if there are no unit productions $A \rightarrow B$ and no epsilon productions $A \rightarrow \epsilon$, but there are techniques to remove such productions - see Chomsky Normal Form)

La garanzia afferma che l'unica volta che una produzione di A_i inizia con un non-terminale (A_k), tale non-terminale A_k non può essere A_i stesso e non può essere nessuno dei non-terminali precedentemente elaborati (A_j dove $j < i$).

Summary of Recursive Descent

- simple parsing strategy
 - left-recursion must be eliminated first
 - ... but that can be done automatically
- unpopular because of backtracking
 - thought to be too inefficient
 - in practice, backtracking is (sufficiently) eliminated by restricting the class of grammars
- so, it's good enough for small languages
 - careful, though: order of productions important even after left-recursion eliminated
 - try to reverse the order of $T \rightarrow \text{int}^* T$ and $T \rightarrow \text{int}$
 - what goes wrong? (consider our input example int^*int)

Con l'Ordine invertito (Caso che Fallisce)

- Chiamata a $T()$
- Prova Regola 1: $T \rightarrow \text{int}$: Il parser trova una corrispondenza con il primo token: int.
- Il parser consuma int e restituisce true. L'input rimanente è: $^*\text{int}$
- Ritorno al Chiamante: Il parser ha analizzato il T e non ha mai avuto la possibilità di vedere lo $*$ e il secondo int.
- Cosa va storto? La regola più breve e meno specifica ha avuto successo prima che il parser potesse tentare la regola più lunga. Il parser riconosce solo la prima parte della stringa e si ferma, lasciando $^*\text{int}$ non analizzato. Il risultato finale è Fallimento, perché il parser non ha consumato tutto l'input.

Questo esempio dimostra che, nelle grammatiche non deterministiche gestite con backtracking, le produzioni più specifiche o più lunghe devono essere provate prima delle produzioni che sono loro prefissi (le più corte). Altrimenti, la regola corta "ruba" l'analisi, causando un fallimento prematuro dell'intera analisi.

Predictive parsers

Motivation

Dal vecchio parser (r.d. parser), si sostituisce il backtracking (il resto rimane uguale)

- Wouldn't it be nice if
 - the r.d. parser just knew which production to expand next?
 - Idea: replace

```
return E1() || (next = save, E2());
```
 - with

```
switch ( something ) {  
    case L1: return E1();  
    case L2: return E2();  
    otherwise: print "syntax error";  
}
```
 - what's "something", L1, L2?
 - the parser will do lookahead (look at next token)

Quel something, che decide quale produzione scegliere tra E1 ed E2, insieme ad L1 ed L2, viene definito tramite look-ahead che mi permette di "vedere nel futuro" (cioè leggo lo stream di input al parser e decido quale produzione scegliere)

Predictive Parsers

- Like recursive-descent but parser can “predict” which production to use
 - By looking at the next few tokens
 - No backtracking
 - Predictive parsers accept $LL(k)$ grammars
 - L means “left-to-right” scan of input
 - L means “leftmost derivation”
 - k means “predict based on k tokens of lookahead”
 - We will analyse $LL(1)$
 - ANTLR uses $LL(*)$, a more sophisticated technique that considers as many tokens as needed (not covered by the slides)
- grammatiche che permettono di determinare in maniera deterministica la prossima produzione da scegliere
- Anti Bottom-Up (LR)
- look-ahead adattivo

Se ho al massimo una produzione per cella
(esistono anche le celle vuote), allora
predizione deterministica (LL(1) grammar)

LL(1) Languages

Affinché una grammatica sia LL(1), deve soddisfare due requisiti principali che eliminano le scelte ambigue:

- Non Ambiguità: La grammatica deve essere non ambigua, il che significa che ogni stringa nel linguaggio ha un solo albero sintattico.
- Assenza di Ricorsione a Sinistra, che impedisce qualsiasi forma di RDP.

- In recursive-descent, for each ~~non-terminal~~^{variabile} and input token there may be a choice of production
- LL(1) means that for each non-terminal and token there is at most one production that could lead to success (possible only if grammar is non ambiguous)
- Can be specified as a 2D table
 - One dimension for current non-terminal to expand
 - One dimension for next token
 - A table entry contains one production

Se una cella è vuota e viene visitata, significa che il parser ha riscontrato un errore di sintassi.

Se la cella è vuota,
significa che non esiste
produzione da usare con
quel token in look-ahead

Quando il parser ha in cima alla sua
pila il non-terminale A ed il prossimo
token in ingresso è a, consulta la cella
 $T[A, a]$ per sapere esattamente quale
singola produzione applicare. Questo
garantisce il funzionamento
deterministico (senza backtracking).

Left factoring

Predictive Parsing and Left Factoring

- Recall the grammar

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

$$T \rightarrow (E) \mid (E)^* T \mid \text{int} \mid \text{int}^* T$$

- Impossible to predict because

- For T two productions start with int and two with (
 - For E it is not clear how to predict

- In general a grammar must be left-factored before using predictive parsing
(managed automatically by ANTLR4 LL(*) algorithm)

 non serve fare la fattorizzazione perché ANTLR usa un look-ahead adattivo (guarda avanti quanto basta per prendere una decisione)

Left-Factoring Example

- Recall the grammar

$$E \rightarrow \underline{T} + E \mid T$$

In E, c'è un prefisso

$$T \rightarrow (\underline{E}) \mid (\underline{E})^* T \mid \underline{\text{int}} \mid \underline{\text{int}}^* T$$

In T, ci sono due prefissi

Si raccolgono i prefissi comuni nei corpi della produzione e creo una variabile che rimuove i prefissi (max prefisso comune, unica produzione che faccio seguire da nuova variabile)

- **Factor out common prefixes of productions**

$$\underline{(E)} \rightarrow T \underline{X}$$

$$\underline{X} \rightarrow + E \mid \epsilon$$

$$\underline{(T)} \rightarrow (\underline{E}) Y \mid \underline{\text{int}} Y$$

I prefissi rimangono nella testa della produzione iniziale, mentre ciò che segue i prefissi vengono fatti spostare in una nuova variabile

$$Y \rightarrow * T \mid \epsilon$$

Essendo in comune con $(E)^* T$, $\text{int}^* T$; l'ho messa una sola volta in Y

Se no, sarebbe stato: $Y \rightarrow * T \mid * T \mid \epsilon$

LL(1) parser (details)

PDA Deterministico perché usa Stack
(PDA) e Tabella (Deterministico)

perché ogni cella punta al massimo ad una sola produzione

LL(1) parser

- to simplify things, instead of

```
switch ( something ) {  
    case L1: return E1();  
    case L2: return E2();  
    otherwise: print "syntax error";  
}
```

Nei parser a discesa ricorsiva scritti a mano, lo switch deve decidere quale produzione scegliere basandosi sul prossimo token di input.

- we'll use a LL(1) table and a parse stack

- the LL(1) table will replace the switch
- the parse stack will replace the call stack

La decisione switch viene completamente sostituita dalla consultazione della Tabella LL(1). Il parser legge l'elemento alla riga del non-terminale corrente e alla colonna del token di input.
Vantaggio: Il programma del parser diventa un piccolo algoritmo fisso, mentre le regole della grammatica sono contenute interamente nella tabella. Ciò rende il parser più generale e la grammatica più facile da modificare.

Rifiuto Stringa: cella vuota (syntax error)
Accetto Stringa: non ho più nulla nello strato di input

LL(1) Parsing Table Example

- Left-factored grammar

$$E \rightarrow TX$$

$$X \rightarrow + E \mid \epsilon$$

$$T \rightarrow (E)Y \mid \text{int } Y$$

$$Y \rightarrow * T \mid \epsilon$$

- The LL(1) parsing table:

	int	*	+	()	\$
T	int Y			(E)Y		
E	TX			TX		
X			+ E		ϵ	ϵ
Y		* T	ϵ		ϵ	ϵ

LL(1) Parsing Table Example (Cont.)

- Consider the [E, int] entry
 - "When current non-terminal is E and next input is int, use production $E \rightarrow TX$ "
 - This production can generate an int in the first place La tabella ci dice che solo la produzione $E \rightarrow TX$ (che deriva in un Termine T) ha int nel suo insieme FIRST (cioè, T può iniziare con un int).
- Consider the [Y, +] entry
 - "When current non-terminal is Y and current token is +, get rid of Y"
 - We'll see later why this is so

LL(1) Parsing Tables. Errors

- Blank entries indicate error situations
 - Consider the $[E, *]$ entry
 - “There is no way to derive a string starting with $*$ from non-terminal E”

Syntax Error

Using Parsing Tables

- Method similar to recursive descent, except
 - For each non-terminal $S \downarrow$ on top of the stack → Si considera il non-terminale S che è attualmente in cima allo stack.
 - We look at the next token a → Si guarda il token di lookahead a (il prossimo input). La cella $T[S, a]$ indica l'unica produzione da applicare, garantendo il determinismo.
 - And choose the production shown at $[S, a]$ ↑
- We use a stack to keep track of pending non-terminals (as in leftmost derivations)
- We reject when we encounter an error state
- We accept when we encounter end-of-input

Il simbolo in cima alla pila è confrontato con il token di input.
Se è un Non Terminale (S), si applica la produzione $T[S, a]: S$ viene rimosso (pop) e i simboli del lato destro della produzione sono inseriti (push) sulla pila in ordine inverso.
Se è un Terminale (t) e $t=a$, entrambi vengono rimossi (pop), e si avanza all'input successivo.

LL(1) Parsing Algorithm

- add \$ at the end of the array of tokens
 - initialize next pointing to the first token
 - initialize stack = < S \$ >

repeat

case stack of

(in cima allo stack ho:)

Variabile

<X rest> : if $T[X, *next] = y_1 \dots y_n$

```
then stack ← < y1... yn rest >;  
else error();
```

non vado avanti in input (next, puntatore sullo stream, non va avanti; modifco solo lo stack)

Sostituisco la Variabile in cima allo stack con la produzione presente nella Tabella, considerando il Token in look-ahead. Se la cella è vuota, ho errore sintattico

Terminale

< t rest > : if t == *(next++)

```
then stack ← <rest>;  
else error();
```

Rimuovo il terminale dallo stack, se effettua match con quello in input (puntato dal next). Se non fa match, ho un errore sintattico

until stack == < >

Se mi rimane solo il \$ nello stream di input, pulisco lo stack fino ad arrivare a \$ ed essendo \$ un terminale, fa match con l'input e lo stack diventa vuoto, così l'algoritmo termina (quindi riconosce per pila vuota)

LL(1) Parsing Example

Viene utilizzata la tabella insieme allo stack

Stack	Input	Action
E \$	int * int \$	T X
T X \$	int * int \$	int Y
int Y X \$	int * int \$	terminal
Y X \$	* int \$	* T
* T X \$	* int \$	terminal
T X \$	int \$	int Y
int Y X \$	int \$	terminal
Y X \$	\$	ϵ
X \$	\$	ϵ
\$	\$	terminal
		ACCEPT



Tutti i linguaggi LL(1) sono linguaggi liberi dal contesto, ma non tutti i linguaggi liberi dal contesto sono linguaggi LL(1) (cioè, esistono CFG che non sono LL(1)).

Se si ha piú di una produzione in almeno una cella, significa che 1 token in look-ahead non basta

Constructing Parsing Tables

- LL(1) languages are those defined by a parsing table for the LL(1) algorithm
- No table entry can be multiply defined (deterministic parsing algorithm!)

Non é possibile avere piú di una produzione in una cella
- We want to generate parsing tables from CFG

Generare da CFG

Problema: Dato il non-terminale A in cima alla pila, ed il token di lookahead b nell'input, quale produzione A devo scegliere?

Constructing Predictive Parsing Tables

- Consider the state $S\$ \Rightarrow^* \beta A \gamma$
 - With **b** the next token
 - Trying to match input string $\beta b \delta$
- There are two possibilities:
1. \sqrt{b} belongs to an expansion of **A** in uno o più passi
 - Production $A \rightarrow \alpha$ can be used if b can start a string derived from α cioè se il token b fa parte dell'insieme dei primi simboli che α può generare, allora scegli la produzione $A \rightarrow \alpha$

In this case we say that **b** is in $\text{First}(\alpha)$
- simbolo iniziale della grammatica
- beta: È la porzione della stringa di input che è già stata analizzata (corrisponde ai terminali già rimossi dalla pila e consumati dall'input).
- A: È il non-terminale in cima alla pila
- gamma: È la stringa di simboli terminali e non-terminali che si trovano sotto \$A\$ sulla pila e che devono ancora essere espansi/corrisposti.
- beta: La parte di input già consumata e che corrisponde alla beta nella derivazione.
- b: È il token di lookahead (il simbolo di input successivo), quello su cui il parser sta basando la sua decisione.
- delta: È il resto della stringa di input che deve ancora essere analizzato.

Or...

Constructing Predictive Parsing Tables (Cont.)

Produzione di A dove

2. ~~Otherwise~~[✓] the expansion of A can be empty and b belongs to an expansion of γ (A annullabile direttamente/indirettamente)
- Means that b can appear after A in a derivation of the form $S\$ \Rightarrow^* \beta A b \omega$
 - We say that b is in $\text{Follow}(A)$ in this case (b dopo A in un cammino di derivazione)
 - Which production of A can we use in this case?
 - $A \rightarrow \alpha$ can be used if α can expand to ϵ
 - We say that ϵ is in $\text{First}(\alpha)$ in this case

Computing First, Follow sets

First Sets. Example

- Recall the grammar

$$E \rightarrow TX$$

$$T \rightarrow (E)Y \mid \text{int} Y$$

$$X \rightarrow + E \mid \varepsilon$$

$$Y \rightarrow * T \mid \varepsilon$$

- First sets (nei First Sets, si mettono solo i terminali)

$$\text{First}(+) = \{+\} \quad \text{First}(*) = \{*\} \quad \text{First}(()) = \{(())\}$$

$$\text{First}()) = \{)\}$$

$$\text{First}(\text{int}) = \{\text{int}\}$$

I First Sets dei terminali sono i singoletti dei terminali stessi

$$\text{First}(T) = \{\text{int}, ()\}$$

$$\text{First}(X) = \{+, \varepsilon\}$$

$$\text{First}(E) = \{\text{int}, ()\}$$

$$\text{First}(Y) = \{*, \varepsilon\}$$

→ Si guarda la testa delle produzioni: si prendono solo i terminali. Se ho una variabile come primo carattere, prendo il First Set della Variabile. Se la produzione ha più di un carattere, faccio il First Sets del primo carattere e faccio il First Sets dei successivi, se nel precedente First Set c'era Epsilon

Se ho un epsilon nel First set della variabile X, non significa che X inizia con epsilon, ma significa che X è nullable

Computing First Sets

Definition

$$\text{First}(X) = \{ b \mid X \Rightarrow^* ba \} \cup \{ \epsilon \mid X \Rightarrow^* \epsilon \}$$

(where X is a non-terminal or terminal symbol)

1. $\text{First}(b) = \{ b \}$

I First Sets dei terminali hanno se stessi

2. For all productions $X \rightarrow A_1 \dots A_n$ with $n > 0$

- Add $\text{First}(A_1) - \{\epsilon\}$ to $\text{First}(X)$. Stop if $\epsilon \notin \text{First}(A_1)$
- Add $\text{First}(A_2) - \{\epsilon\}$ to $\text{First}(X)$. Stop if $\epsilon \notin \text{First}(A_2)$
- ...
- Add $\text{First}(A_n) - \{\epsilon\}$ to $\text{First}(X)$. Stop if $\epsilon \notin \text{First}(A_n)$
- Add ϵ to $\text{First}(X)$

Aggiungo al first set di X, i first set dei caratteri presenti nella produzione. Mi fermo ad aggiungere se l'attuale first set non aveva epsilon. Se arrivo in fondo alla produzione e l'ultima ha epsilon, aggiungo epsilon al first set di X

3. Repeat step 2 until no First set grows

Se un First Set dipende dal contenuto di un altro, cioè A dipende da B, allora fermarsi nel riempire A e riempire B, cercando di concludere per quel first set

L'Algoritmo dei First Sets è iterativo. I First sets, all'inizio, della grammatica sono insiemi vuoti. Mano mano, riempio e vado alla prossima iterazione se nella precedente ho aggiunto dei caratteri in un first set. Mi fermo se non aggiungo nulla in quella iterazione.

Computing First Sets

per una stringa

Definition ($n \geq 0$ symbols) $\text{First}(X_1 X_2 \dots X_n) =$
 $\{ b \mid X_1 X_2 \dots X_n \Rightarrow^* b\alpha \} \cup \{ \varepsilon \mid X_1 X_2 \dots X_n \Rightarrow^* \varepsilon \}$

- Add $\text{First}(X_1) - \{\varepsilon\}$ to $\text{First}(X_1 X_2 \dots X_n)$. Stop if $\varepsilon \notin \text{First}(X_1)$
- Add $\text{First}(X_2) - \{\varepsilon\}$ to $\text{First}(X_1 X_2 \dots X_n)$. Stop if $\varepsilon \notin \text{First}(X_2)$
- ...
- Add $\text{First}(X_n) - \{\varepsilon\}$ to $\text{First}(X_1 X_2 \dots X_n)$. Stop if $\varepsilon \notin \text{First}(X_n)$
- Add $\{\varepsilon\}$ to $\text{First}(X_1 X_2 \dots X_n)$.

Simile alla creazione di un First set di una testa di produzione con più produzioni

Il calcolo dei First Sets mi serve per il calcolo dei Follow Sets, per definire la Parsing Table

Follow Sets. Example

- Recall the grammar

$$E \rightarrow TX$$

$$T \rightarrow (E)Y \mid \text{int} Y$$

$$X \rightarrow +E \mid \varepsilon$$

$$Y \rightarrow *T \mid \varepsilon$$

- Follow sets (examples showing the idea)

Follow(+) obtained by adding First(E) only

that is, simply, Follow(+) = { int, (})

Follow() obtained by adding First(Y) - { ε }

and by adding Follow(T)

Follow(Y) obtained by adding Follow(T) only

(the same set is added two times)

Perché nel first di Y ho epsilon e quindi vado avanti ma essendo alla fine della produzione vado a prendere il Follow della testa della produzione

Computing Follow Sets

Definition

$$\text{Follow}(X) = \{ b \mid S\$ \Rightarrow^* \beta X b \delta \}$$

Se dopo X ho una stringa, faccio il First della stringa, non la scompongo nei vari caratteri

1. Add \$ to Follow(S) (if S is the start non-terminal)

Per definizione, il Follow di S contiene \$

2. For all productions $Y \rightarrow \alpha X A_1 \dots A_n$ with $n >= 1$

Add First($A_1 \dots A_n$) - { ϵ } to Follow(X)

and for all productions $Y \rightarrow \alpha X$ or $Y \rightarrow \alpha X \beta$ with

$\epsilon \in \text{First}(\beta)$

Add Follow(Y) to Follow(X)

Nel Follow, va il First del carattere che segue X. Se il first contiene epsilon, vado avanti e metto nel follow anche il First di quello dopo

1. Repeat step 2 until no Follow set grows

L'algoritmo dei Follow Sets è iterativo. Tutti i follow sets, all'inizio, sono vuoti tranne S, che ha \$, essendo variabile iniziale. Continuo fino a che non smettono di crescere i follow sets (l'algoritmo termina quando nella iterazione attuale non ho aggiunto nulla).

Se X è alla fine della produzione o dopo X ho una variabile/terminale che contiene epsilon nel suo first, aggiungo al Follow di X il follow della testa di produzione

Follow Sets. Example

- Recall the grammar

$$E \rightarrow TX$$

$$T \rightarrow (E)Y \mid \text{int} Y$$

$$X \rightarrow + E \mid \varepsilon$$

$$Y \rightarrow * T \mid \varepsilon$$

- Follow sets

$$\text{Follow}(E) = \{\$,)\}$$

$$\text{Follow}(T) = \{+, \$, ,)\}$$

$$\text{Follow}(X) = \{\$,)\}$$

$$\text{Follow}(Y) = \{+, \$, ,)\}$$

E ed X dipendono
uno dall'altro,
quindi hanno
follow uguali e
non possono
crescere di più
(stessa cosa T e Y)

$\text{Follow}() = \{*, +, \$, ,)\}$ (we will not use Follow
on terminals, though)

Constructing the LL(1) parsing table

Constructing LL(1) Parsing Tables

- Construct a parsing table T for CFG G
 - For each production $A \rightarrow \alpha$ in G do:
 - For each terminal $b \in \text{First}(\alpha)$ do
 - $T[A, b] = \alpha$
 - If $\epsilon \in \text{First}(\alpha)$, for each $b \in \text{Follow}(A)$ do
 - $T[A, b] = \alpha$
- (remember that last rule applies also to $\$$:
- If $\epsilon \in \text{First}(\alpha)$ and $\$ \in \text{Follow}(A)$ do
- $T[A, \$] = \alpha$)

nel senso che se ho $\$$ in $\text{Follow}(A)$, metto la produzione di A in $T[A, \$]$

Constructing LL(1) Tables. Example

- Recall the grammar

$$E \rightarrow TX$$

$$X \rightarrow + E \mid \epsilon$$

$$T \rightarrow (E)Y \mid \text{int } Y$$

$$Y \rightarrow * T \mid \epsilon$$

- Where in the line of Y we put $Y \rightarrow *T$?
 - In the columns of $\text{First}(*T) = \{ * \}$

- Where in the line of Y we put $Y \rightarrow \epsilon$?
 - In the columns of $\text{Follow}(Y) = \{ \$, +,) \}$



Perché subito il $\text{Follow}(Y)$? Perché il First di epsilon contiene solo epsilon e se nel fist c'è epsilon devo considerare anche le colonne che hanno come terminale quelli presenti in Follow di Y

Notes on LL(1) Parsing Tables

Varie
Motivazioni per
cui G non è LL(1)

ha piú di una produzione

- If any entry is multiply defined then G is not LL(1)
 - If G is ambiguous
 - If G (with no useless variables) is left recursive
 - If G (with no useless variables) is not left-factored
 - And in other cases as well
- Most programming language grammars are not LL(1)
- There are tools that build LL(1) tables and there are fully declarative parser generators that use the LL approach

Few words about Ambiguity

- Grammar

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{int}$$

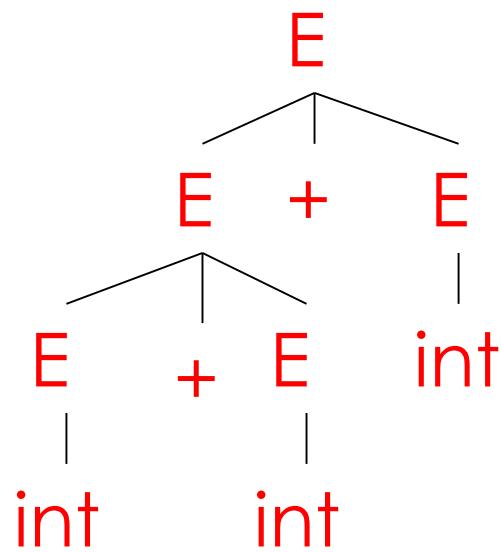
- Strings

int + int + int

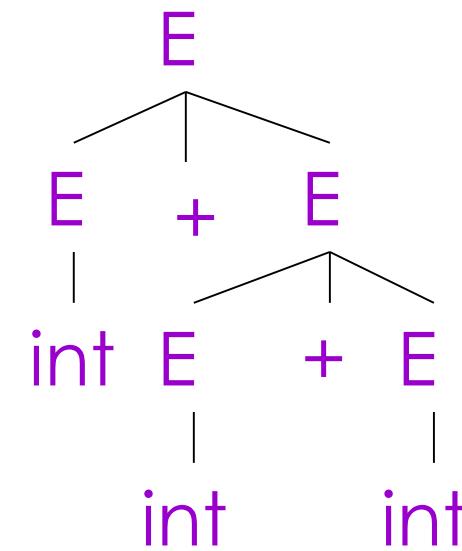
int * int + int

Ambiguity. Example

This string has two parse trees



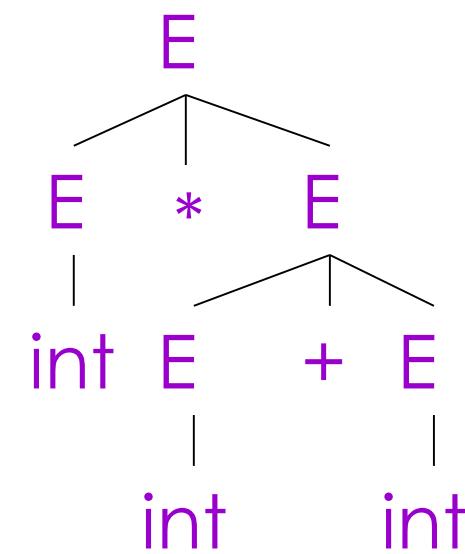
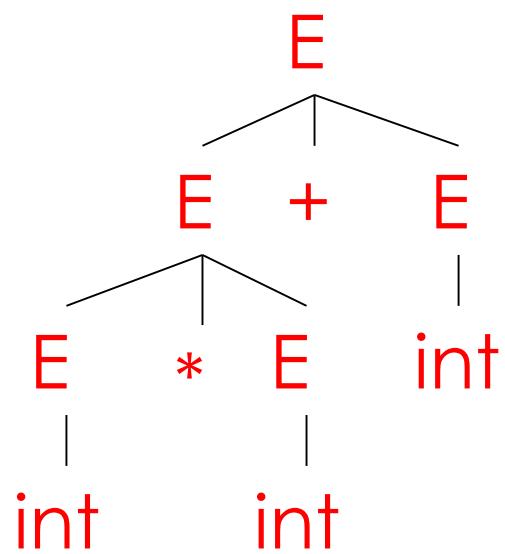
+ is associated to the left



↑
qua il più è associativo a destra

Ambiguity. Example

This string has two parse trees



* is given higher precedence than +

il + ha più precedenza rispetto al *

Ambiguity (Cont.)

L'ambiguità sintattica, cioè avere più di un Albero Sintattico, è un problema perché genera ambiguità semantica, impedendo al compilatore di sapere quale operazione eseguire per prima.

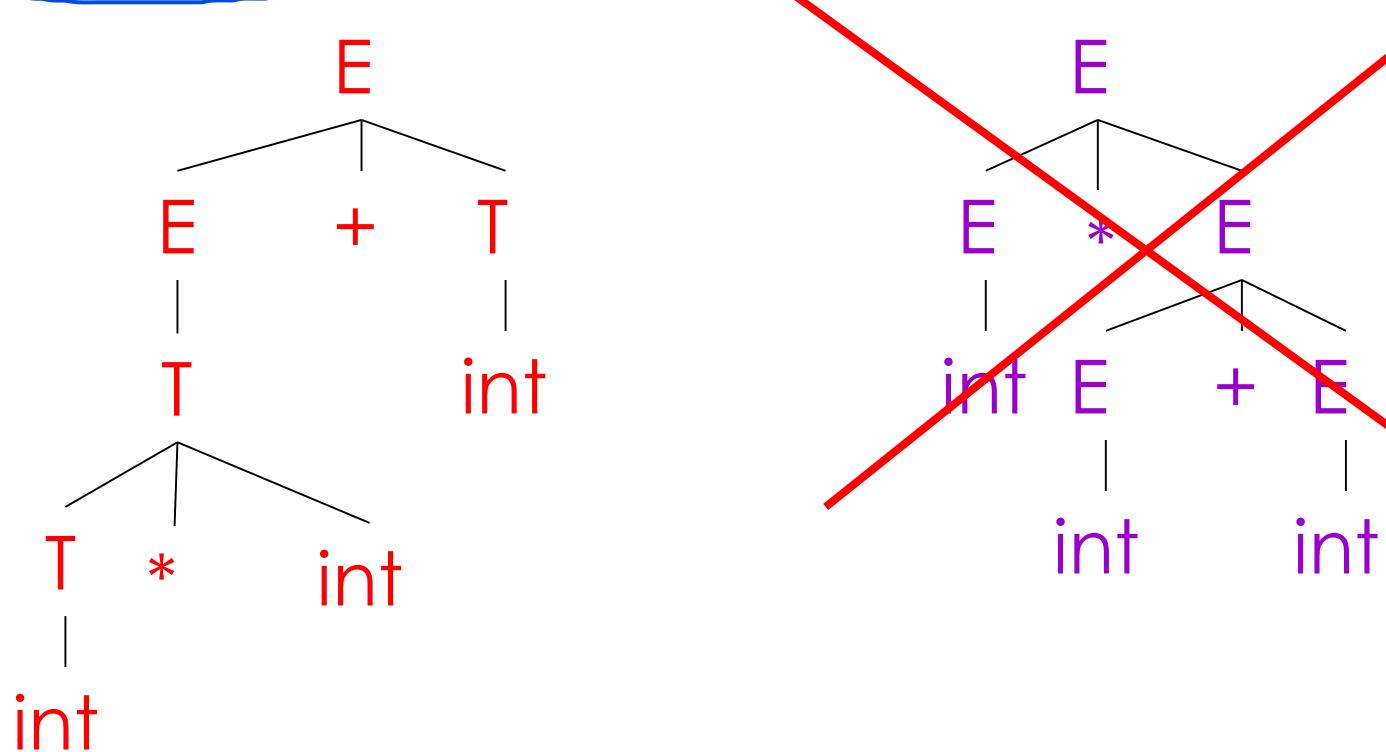
- A grammar is *ambiguous* if it has more than one parse tree for some string
 - Equivalently, there is more than one right-most or left-most derivation for some string
- Ambiguity is bad
 - Leaves meaning of some programs ill-defined
- Ambiguity is common in programming languages
 - Arithmetic expressions
 - IF-THEN-ELSE

Dealing with Ambiguity

- There are several ways to handle ambiguity
- 1 • Most direct method is to rewrite the grammar unambiguously cioé trovare una grammatica non ambigua equivalente a quella ambigua
- $$E \rightarrow E + T \mid T$$
- $$T \rightarrow T^* \text{ int } \mid T^* (E) \mid \text{ int } \mid (E)$$
- 2 [• Enforces precedence of * over +
• Enforces left-associativity of + and *

Ambiguity. Example

The $\text{int} * \text{int} + \text{int}$ has only one parse tree now



Ambiguity: The Dangling Else

- Consider the grammar

$S \rightarrow \text{if } E \text{ then } S$

| $\text{if } E \text{ then } S \text{ else } S$

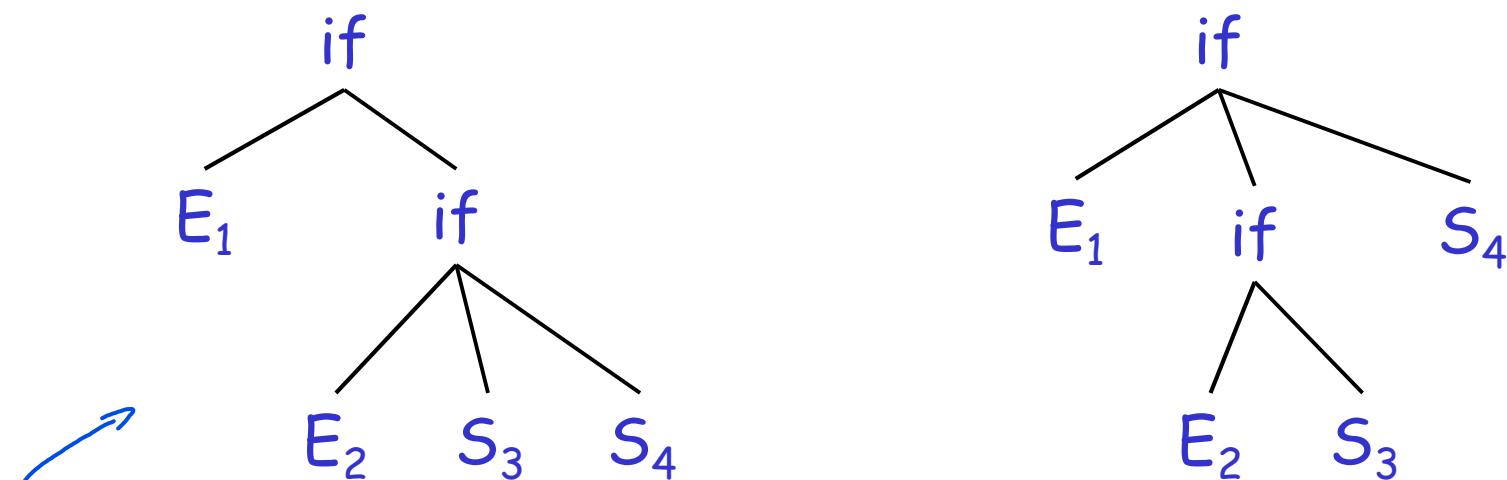
- This grammar is also ambiguous

The Dangling Else: Example

- The expression

if E_1 then if E_2 then S_3 else S_4

has two (abstract) parse trees



- Typically we want the first form

The Dangling Else: A Fix

- else matches the closest unmatched then
- We can describe this in the grammar
(distinguish between matched and unmatched “then”)

MIF

/* if where all then are matched */

$S \rightarrow \text{if } E \text{ then } S$

| $\text{if } E \text{ then MIF else } S$

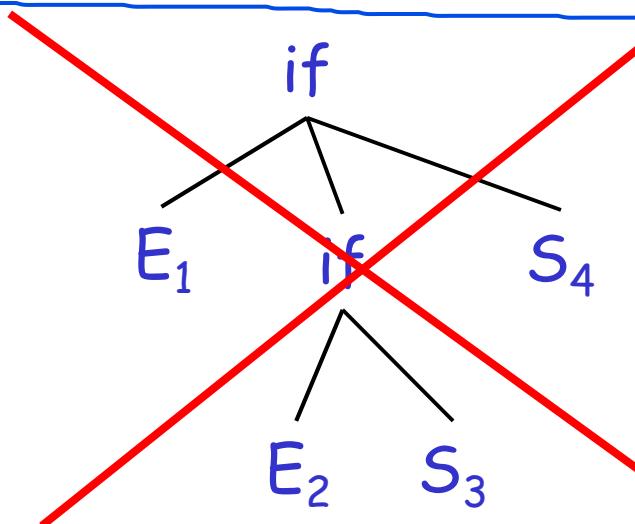
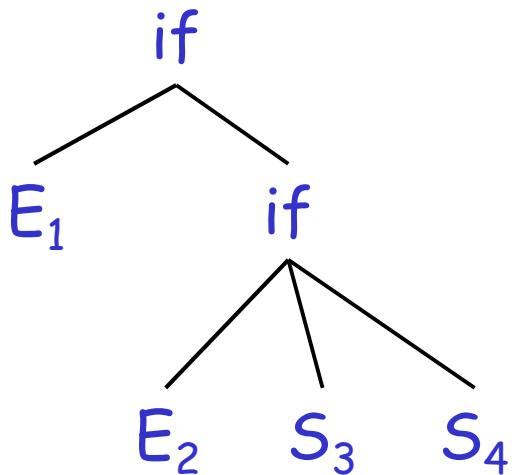
$MIF \rightarrow \text{if } E \text{ then } \text{MIF} \text{ else } \text{MIF}$

In questo caso, si è scelto/creata
una grammatica equivalente a
quella di prima, ma non ambigua

- Describes the same set of if-then-else strings

The Dangling Else: Example Revisited

- The expression if E_1 then if E_2 then S_3 else S_4



- A valid parse tree
- Not valid because the then expression is not a MIF

Ambiguity

- No general techniques for handling ambiguity
- Impossible to convert automatically an ambiguous grammar to an unambiguous one
- Used with care, ambiguity can simplify the grammar
 - Sometimes allows more natural definitions
 - We need disambiguation mechanisms

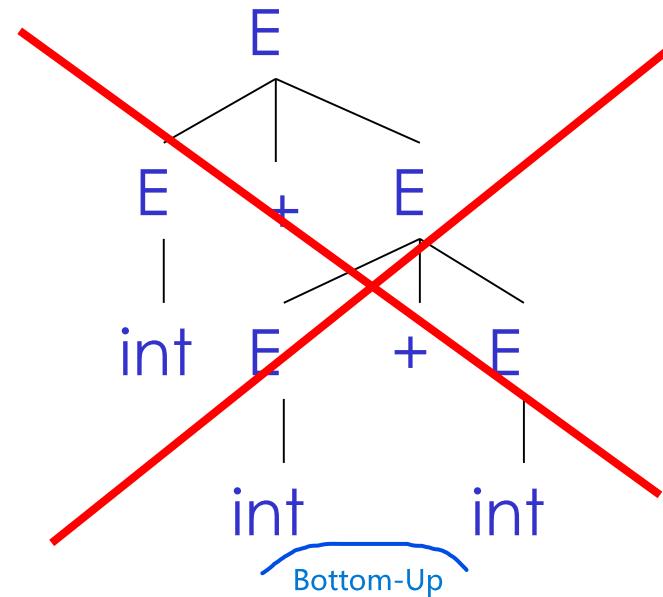
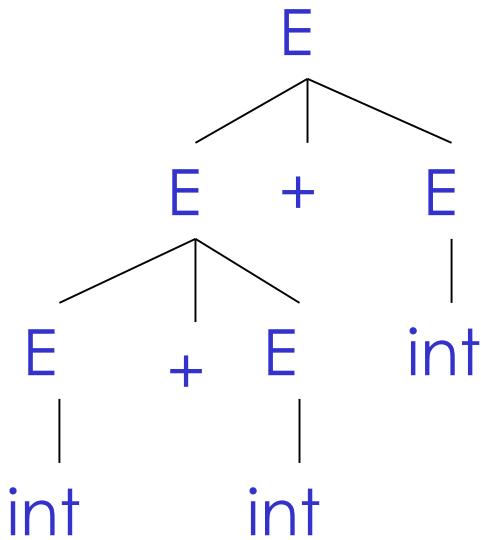
Precedence and Associativity Declarations

- Instead of rewriting the grammar
 - Use the more natural (ambiguous) grammar
 - Along with disambiguating declarations
- LR (bottom-up) parsers and, recently, also LL (top-down) parsers allow
 - precedence and associativity declarations to disambiguate grammars
- Examples ...

Associativity Declarations

- Consider the grammar
- Ambiguous: two parse trees of $\text{int} + \text{int} + \text{int}$

$$E \rightarrow E + E \mid \text{int}$$

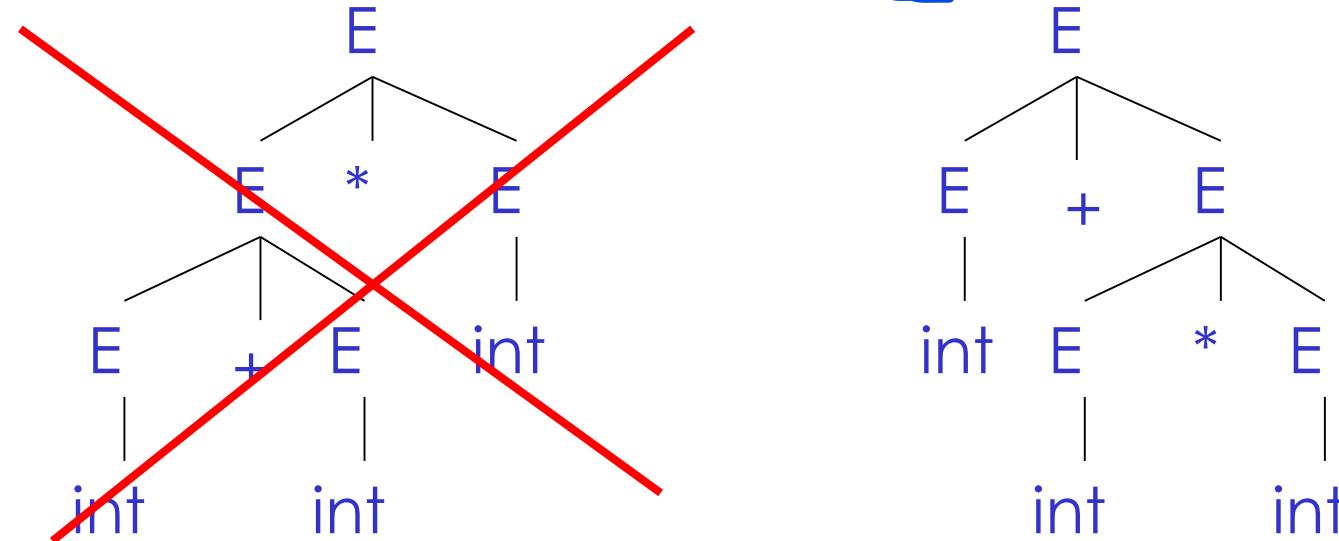


- Left-associativity declaration in LR: `%left +`
(default in ANTLR if `<assoc=right>` not specified)

Top-Down

Precedence Declarations

- Consider the grammar $E \rightarrow E * E \mid E + E \mid \text{int}$
- And the string int + int * int



- Precedence declarations in LR → **%left +**
(in ANTLR based on production
order: first one has higher priority)
- %left ***

Le produzioni più in alto
hanno maggior priorità