Algoritmi di Parsing: Top-down Parsing

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

Now let's parse a string

- 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

Recursive Descent Parsing

- Consider the grammar

  $E \to T + E \mid T$

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

- Token stream is: $\text{int}_5 * \text{int}_2$
- Start with top-level non-terminal E

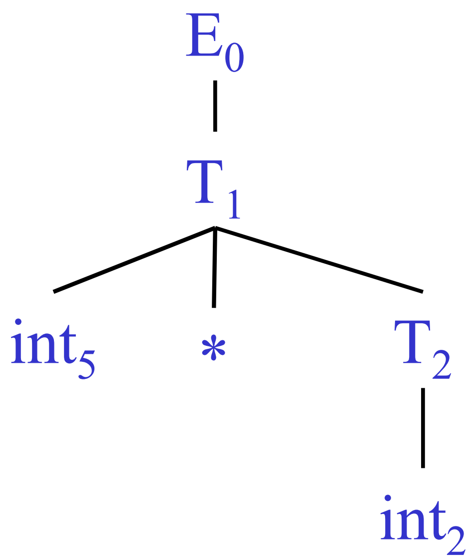
- Try the rules for E in order

Recursive Descent Parsing. Example (Cont.)

- Try $E_0 \to T_1 + E_1$
- Then try a rule for T_1: $T_1 \to (E_2) * T_2$
  - But (does not match input token int_5
- Try $T_1 \to (E_2)$
  - But (does not match input token int_5
- Try $T_1 \to int * T_2$
  - Then all rules for T_2 until $T_2 \to int$. This matches!
- But + after T_1 will be unmatched
  - Backtrack to choice for E_0

Recursive Descent Parsing. Example (Cont.)

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



A Recursive Descent Parser (2)

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

    ```
    bool term(TOKEN tok) { return in[next++] == tok; }
    ```
  - A given production of S (the n^{th})

    bool S_n() { … }
  - Any production of S:

    ```
    bool S() { … }
    ```

- These functions advance next

A Recursive Descent Parser (3)

- For production $E \to T + E$

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

- For production $E \to T$

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

- For all productions of E (with backtracking)

  ```
  bool E() {
    int save = next;
    return   E1()
           || (next = save,  E2());   }
  ```

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); }

   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()
- 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 …

When Recursive Descent Does Not Work

- Consider a production $S \to S\ a$:
  - In the process of parsing S we try the above rule
  - What goes wrong?

- A <u>left-recursive grammar</u> has a non-terminal S (not necessarily the initial one) such that

$$S \Rightarrow^{+} S\alpha \quad \text{for some } \alpha$$

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

Elimination of Left Recursion

- Consider the left-recursive grammar

$$S \to S\,\alpha \mid \beta$$

  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

- Can rewrite using right-recursion

$$S \to \beta\, S'$$

$$S' \to \alpha\, S' \mid \varepsilon$$

Elimination of Left-Recursion. Example

- Consider the grammar

  $S \to S\ 0 \mid 1 \qquad (\ \beta = 1 \text{ and } \alpha = 0\ )$

  can be rewritten as

  $S \to 1\ S'$

  $S' \to 0\ S' \mid \varepsilon$

More Elimination of Left-Recursion

- In general

$$S \to S\,\alpha_1 \mid \dots \mid S\,\alpha_n \mid \beta_1 \mid \dots \mid \beta_m$$

  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 $\beta_1,\dots,\beta_m$ and continue with several instances of $\alpha_1,\dots,\alpha_n$
- Rewrite as

$$S \to \beta_1\,S' \mid \dots \mid \beta_m\,S'$$
$$S' \to \alpha_1\,S' \mid \dots \mid \alpha_n\,S' \mid \varepsilon$$

  (done automatically by ANTLR4)

General Left Recursion

- The grammar

$$S \to A\ \alpha \mid \delta$$

$$A \to S\ \beta$$

is also left-recursive because

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

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

Algorithm for Left Recursion Elimination

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

For i := 1 to n

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

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

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

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 $T \to \text{int} * T$ and $T \to \text{int}$
  - what goes wrong? (consider our input example int*int)

Predictive parsers

Motivation

- 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)

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)

LL(1) Languages

- In recursive-descent, for each non-terminal 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

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 <u>left-factored</u> before using predictive parsing
  (managed automatically by ANTLR4 LL(*) algorithm)

Left-Factoring Example

- Recall the grammar

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

- Factor out common prefixes of productions

$$E \rightarrow T X$$
$$X \rightarrow + E \mid \varepsilon$$
$$T \rightarrow (E) Y \mid \text{int } Y$$
$$Y \rightarrow * T \mid \varepsilon$$

LL(1) parser (details)

LL(1) parser

- to simplify things, instead of

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

- we'll use a LL(1) <u>table</u> and a <u>parse stack</u>
  - the LL(1) table will replace the switch
  - the parse stack will replace the call stack

LL(1) Parsing Table Example

- Left-factored grammar

$E \to T\ X$

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

$X \to +\ E \mid \varepsilon$

$Y \to *\ T \mid \varepsilon$

- The LL(1) parsing table:

	int	*	+	(	)	$
T	int Y			(E) Y		
E	T X			T X		
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 $E \to T X$"
  - This production can generate an int in the first place
- 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"

Using Parsing Tables

- Method similar to recursive descent, except
  - For each non-terminal S
  - We look at the next token a
  - 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

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
      <X rest>  : if T[X,*next] = Y_1...Y_n
                         then stack ← <Y_1... Y_n rest>;
                         else  error ();
      <t rest>   : if t == *(next++)
                         then  stack ← <rest>;
                         else error ();
until stack == < >
```

LL(1) Parsing Example

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

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!)

- We want to generate parsing tables from CFG

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. b belongs to an expansion of *A*
   - Production $A \rightarrow \alpha$ can be used if b can start a string derived from α

     In this case we say that **b is in First(α)**

Or…

Constructing Predictive Parsing Tables (Cont.)

2. Otherwise the **expansion of *A* can be empty** and b belongs to an expansion of γ
   - 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 Follow(*A*)** in this case

   - 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 First(**α**)** in this case

Computing First, Follow sets

First Sets. Example

- Recall the grammar

$$E \to T\,X \qquad\qquad X \to +\,E \mid \varepsilon$$
$$T \to (\,E\,)\,Y \mid \text{int}\;Y \qquad\qquad Y \to *\,T \mid \varepsilon$$

- First sets

First(+) = { + }   First(*) = { * }   First(( ) = { (}

First( )) = {) }   First(int) = { int }

First(T) = {int, (}   First(E) = {int, (}

First(X) = {+, ε }   First(Y) = {*, ε }

Computing First Sets

Definition $\quad First(X) = \{ b \mid X \Rightarrow^* b\alpha\} \cup \{\varepsilon \mid X \Rightarrow^* \varepsilon\}$

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

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

2. For all productions $X \rightarrow A_1 \ldots A_n$ with n>=0
   - Add $First(A_1) - \{\varepsilon\}$ to First(X). Stop if $\varepsilon \notin First(A_1)$
   - Add $First(A_2) - \{\varepsilon\}$ to First(X). Stop if $\varepsilon \notin First(A_2)$
   - …
   - Add $First(A_n) - \{\varepsilon\}$ to First(X). Stop if $\varepsilon \notin First(A_n)$
   - Add ε to First(X)

3. Repeat step 2 until no First set grows

Computing First Sets

Definition (n>=0 symbols) $First(X_1X_2 \ldots X_n) = \{ b \mid X_1X_2 \ldots X_n \Rightarrow^* b\alpha\} \cup \{\varepsilon \mid X_1X_2 \ldots X_n \Rightarrow^* \varepsilon\}$

- Add $First(X_1) - \{\varepsilon\}$ to $First(X_1X_2 \ldots X_n)$. Stop if $\varepsilon \notin First(X_1)$
- Add $First(X_2) - \{\varepsilon\}$ to $First(X_1X_2 \ldots X_n)$. Stop if $\varepsilon \notin First(X_2)$

  …

- Add $First(X_n) - \{\varepsilon\}$ to $First(X_1X_2 \ldots X_n)$. Stop if $\varepsilon \notin First(X_n)$
- Add $\{\varepsilon\}$ to $First(X_1X_2 \ldots X_n)$.

Follow Sets. Example

- Recall the grammar

$E \to T X$ $\qquad$ $X \to + E \mid \varepsilon$

$T \to (E) Y \mid \text{int } Y$ $\qquad$ $Y \to * T \mid \varepsilon$

- 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)

Computing Follow Sets

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

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

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

   Add $\text{First}(A_1 \ldots A_n) - \{\varepsilon\}$ to Follow(X)

   and for all productions $Y \to \alpha X$ or $Y \to \alpha X \beta$ with $\varepsilon \in \text{First}(\beta)$

   Add Follow(Y) to Follow(X)

1. Repeat step 2 until no Follow set grows

Follow Sets. Example

- Recall the grammar

$E \rightarrow T\,X$ $\qquad\qquad X \rightarrow +\,E \mid \varepsilon$

$T \rightarrow (\,E\,)\,Y \mid \text{int}\,Y$ $\qquad\qquad Y \rightarrow *\,T \mid \varepsilon$

- Follow sets

Follow(E) = {$,)} $\qquad$ Follow(X) = {$,) }

Follow(T) = {+, $,)} $\qquad$ Follow(Y) = {+, $,)}

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 $b \in First(\alpha)$ do
    - $T[A, b] = \alpha$
  - If $\varepsilon \in First(\alpha)$, for each $b \in Follow(A)$ do
    - $T[A, b] = \alpha$

  (remember that last rule applies also to \$:
  If $\varepsilon \in First(\alpha)$ and $\$ \in Follow(A)$ do
    - $T[A, \$] = \alpha$)

Constructing LL(1) Tables. Example

- Recall the grammar

$E \to T\,X \qquad\qquad X \to +\,E \mid \varepsilon$

$T \to (\,E\,)\,Y \mid \text{int}\ Y \qquad\qquad Y \to *\,T \mid \varepsilon$

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

- Where in the line of Y we put $Y \to \varepsilon$?
  - In the coloumns of Follow(Y) = { $, +,) }

Notes on LL(1) Parsing Tables

- 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

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

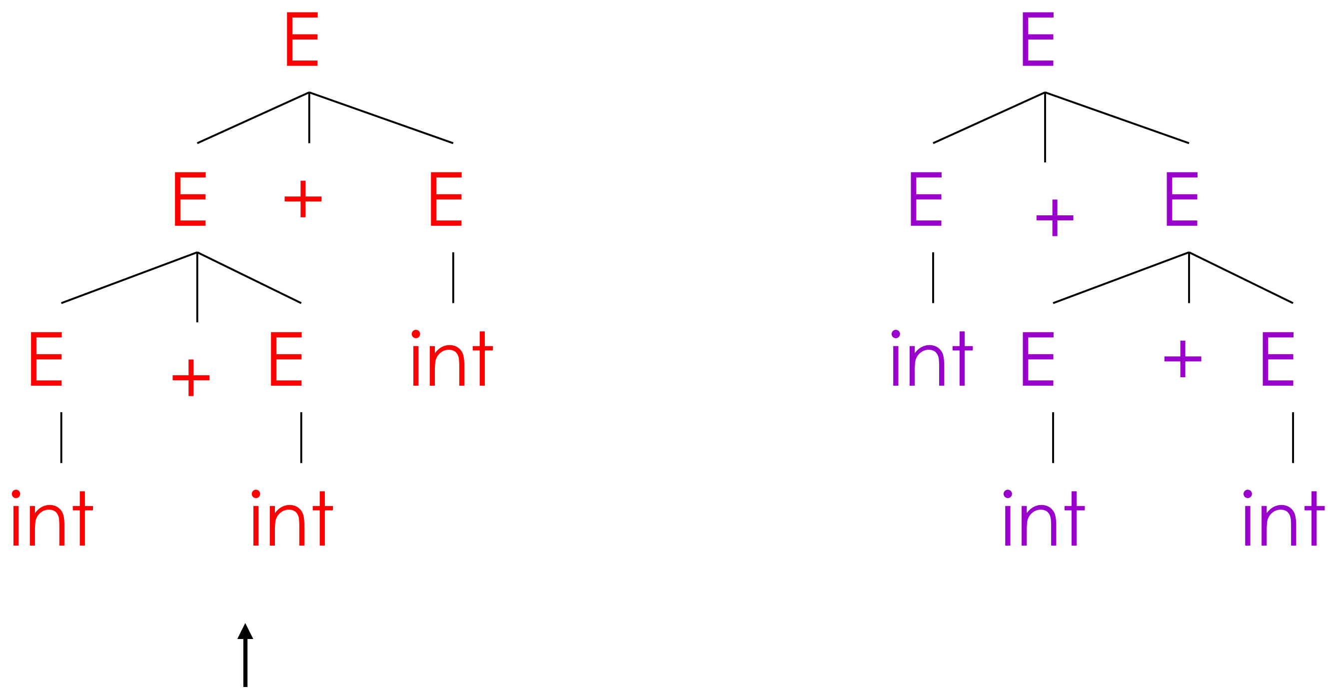
- Strings

  int + int + int

  int * int + int

Ambiguity. Example

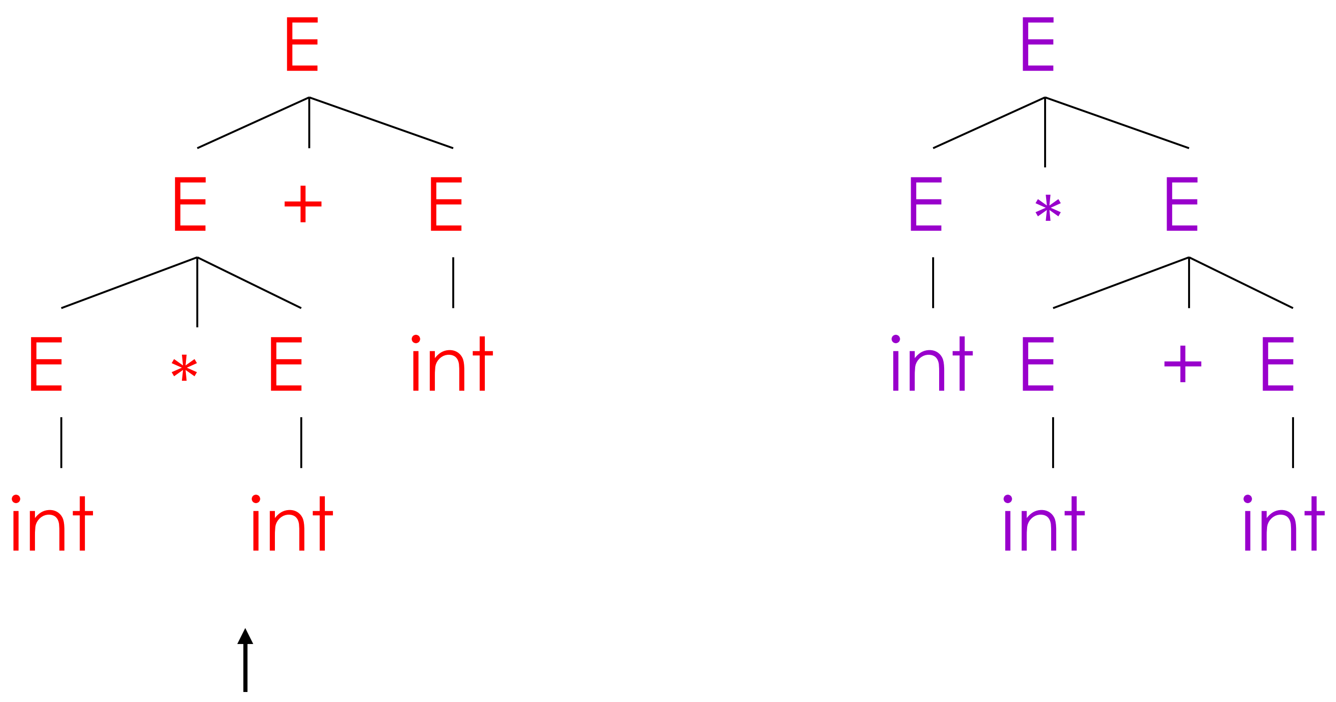
This string has two parse trees



+ is associated to the left

Ambiguity. Example

This string has two parse trees



* is given higher precedence than +

Ambiguity (Cont.)

- 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 <u>bad</u>
  - Leaves meaning of some programs ill-defined
- Ambiguity is <u>common</u> in programming languages
  - Arithmetic expressions
  - IF-THEN-ELSE

Dealing with Ambiguity

- There are several ways to handle ambiguity

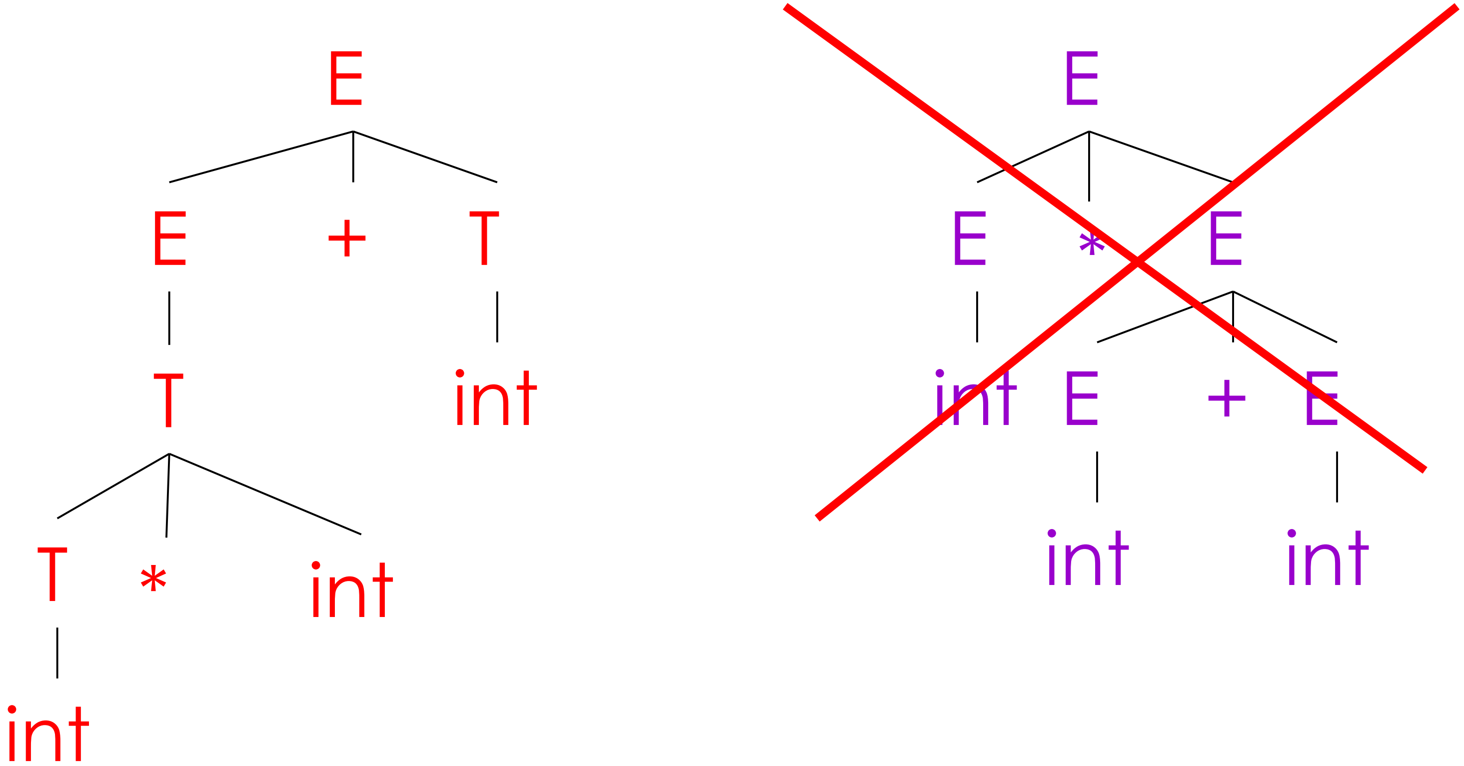
- Most direct method is to rewrite the grammar unambiguously

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

- Enforces precedence of * over +
- Enforces left-associativity of + and *

Ambiguity. Example

The int * int + int has only one parse tree now



Ambiguity: The Dangling Else

- Consider the grammar

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

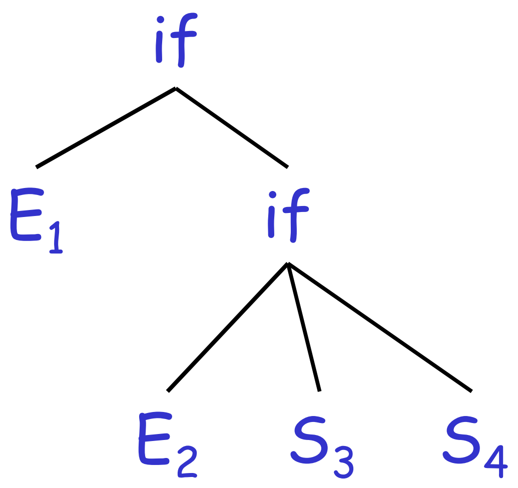
- This grammar is also ambiguous

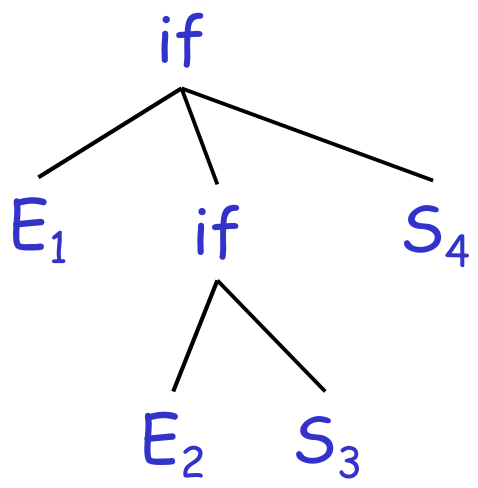
The Dangling Else: Example

- 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")

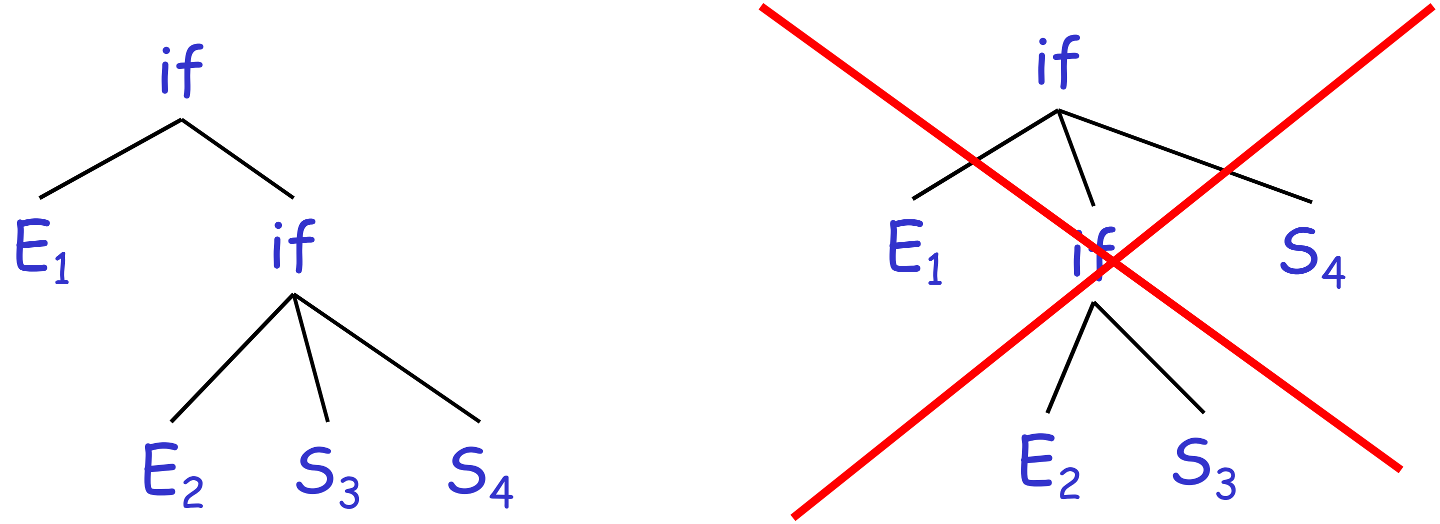
MIF /* if where all then are matched */

$S \rightarrow$ if E then S
| if E then MIF else S
$MIF \rightarrow$ if E then MIF else MIF

- 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

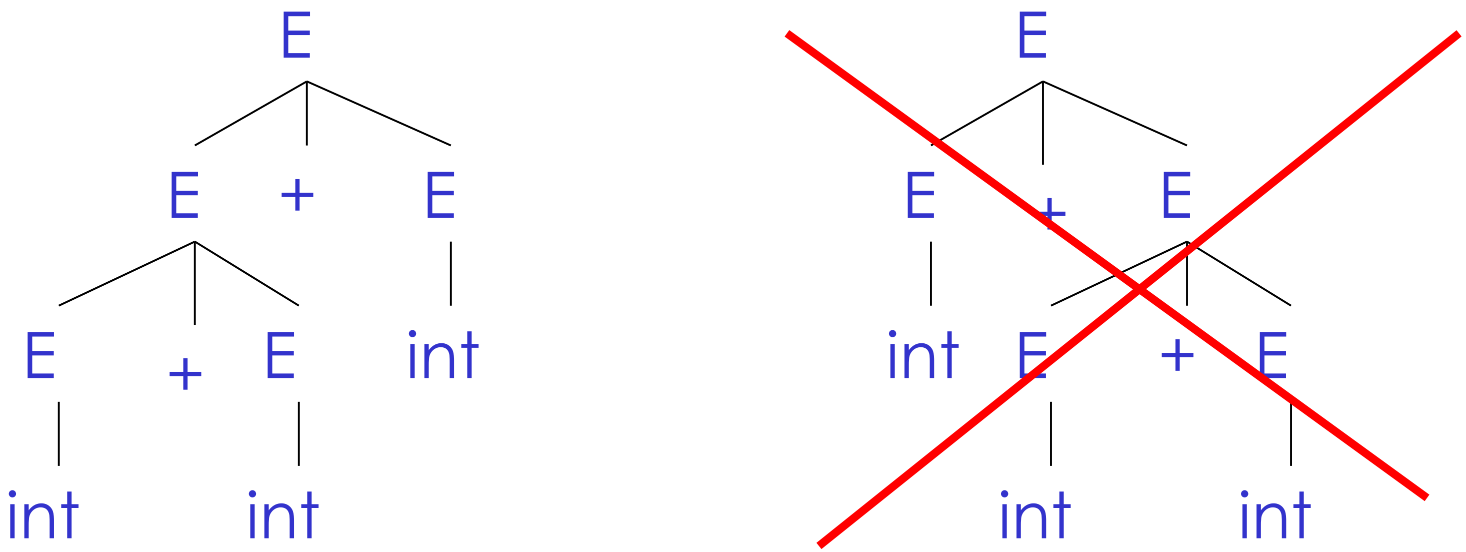
- Instead of rewriting the grammar
  - Use the more natural (ambiguous) grammar
  - Along with disambiguating declarations

- LR (bottom-up) parsers and, recently, also LL (top-down) parsers allow
  - **precedence and associativity declarations** to disambiguate grammars

- Examples …

Associativity Declarations

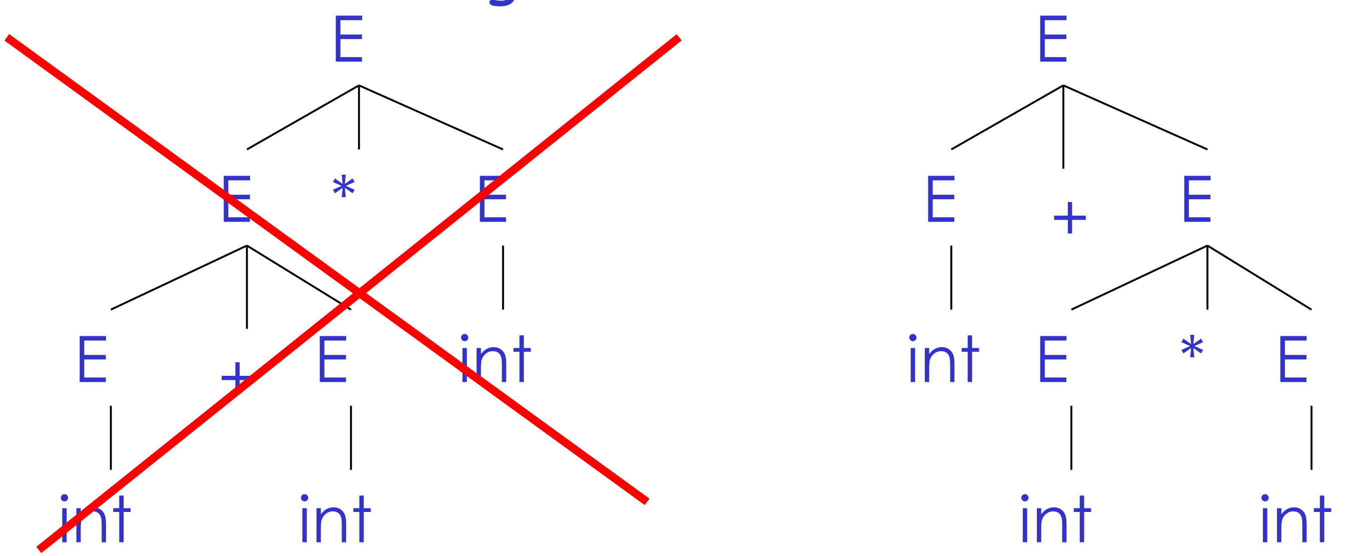
- Consider the grammar $E \rightarrow E + E \mid int$
- Ambiguous: two parse trees of int + int + int



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

Precedence Declarations

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



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