Ambiguity, Precedence, Associativity & Top-Down Parsing

Lecture 9-10

(From slides by G. Necula & R. Bodik)

Administrivia

- Please let me know if there are continued problems with being able to see other people's stuff.
- *Preliminary* run of test data against any projects handed in by midnight Wednesday.
  - Not final data sets, but may give you an indication.
  - You can submit early and often!
  - Will not test again until midnight Friday.

Remaining Issues

- How do we *find a derivation* of *s* ?
- *Ambiguity:* what if there is *more than one parse tree* (interpretation) for some string *s* ?
- *Errors:* what if there *is no parse tree* for some string *s* ?
- Given a derivation, how do we *construct an abstract syntax tree* from it?

Today, we'll look at the first two.

Ambiguity

- Grammar

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

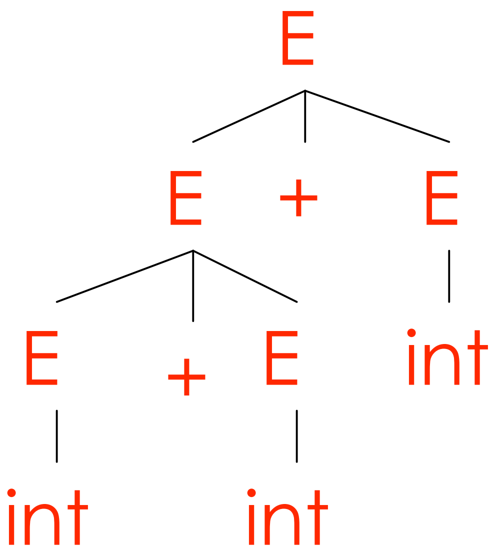
- Strings

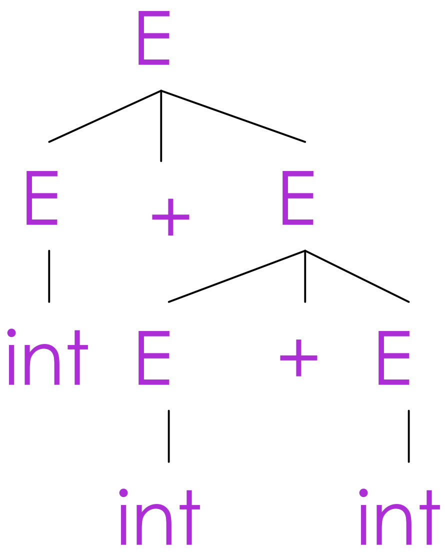
int + int + int

int * int + int

Ambiguity. Example

The string int + int + int has two parse trees



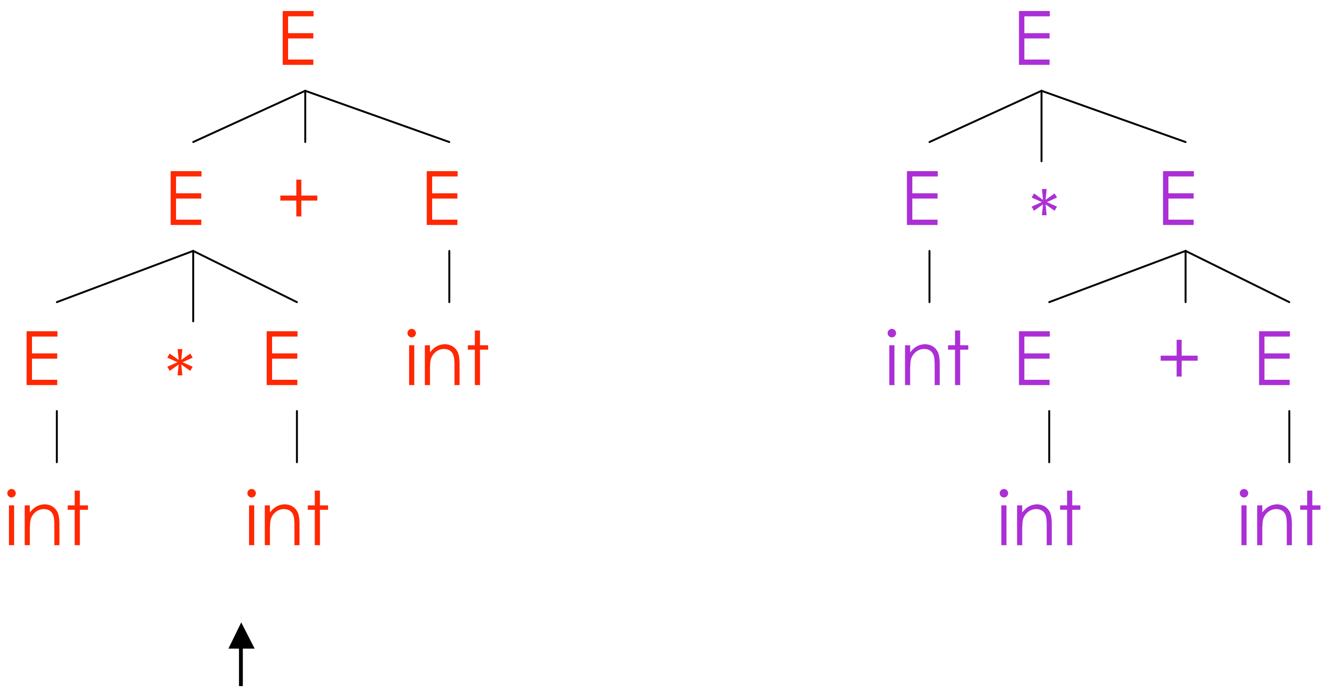


↑

+ is left-associative

Ambiguity. Example

The string int * int + int has two parse trees



* has 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 rightmost or leftmost 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
- Most direct method is to rewrite the grammar unambiguously

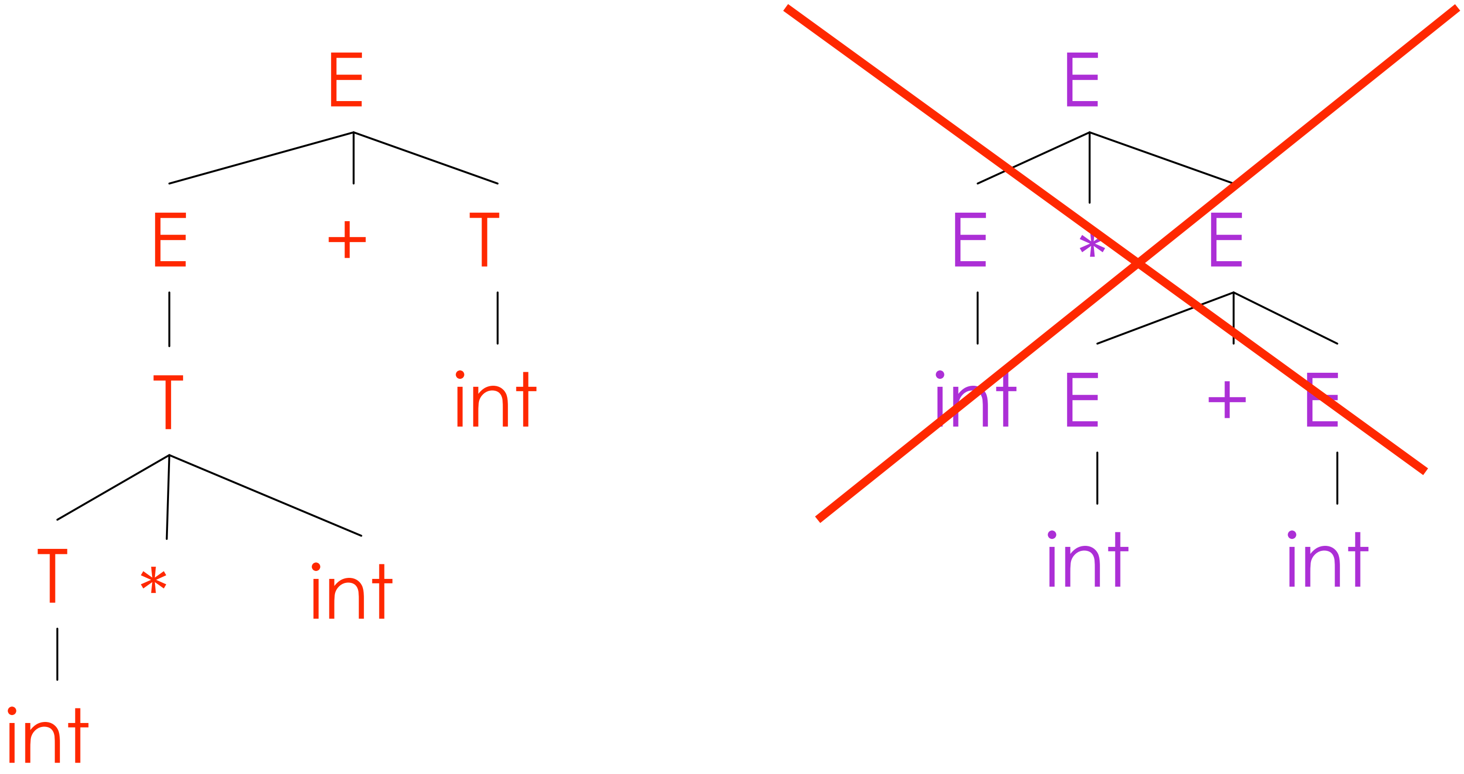
$$E \to E + T \mid T$$

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

- 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

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

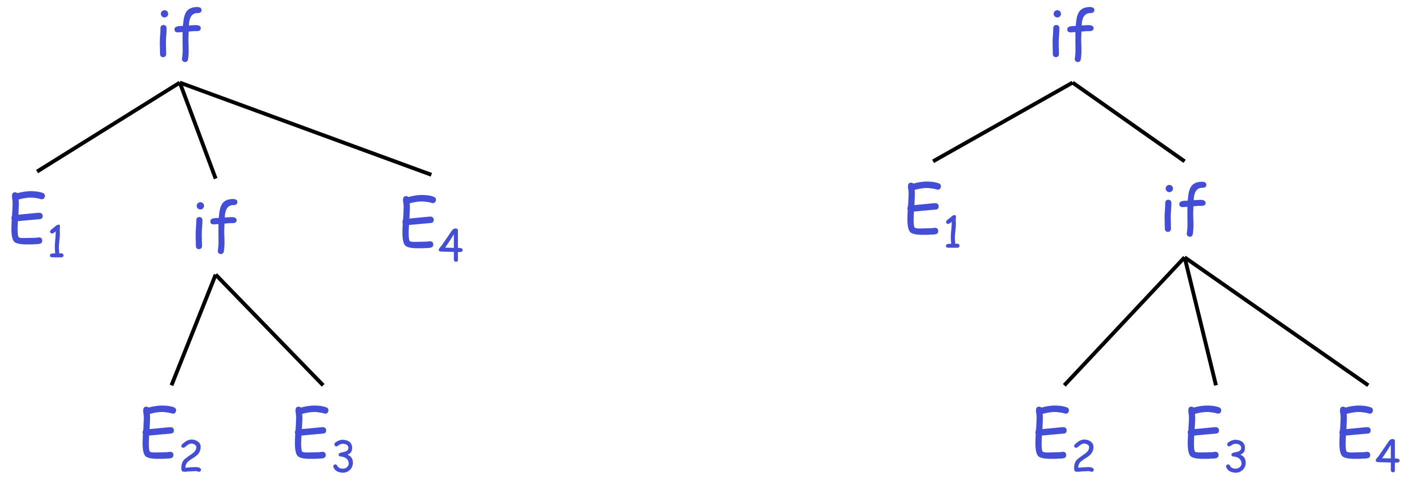
- This grammar is also ambiguous

The Dangling Else: Example

- The expression

  if E_1 then if E_2 then E_3 else E_4

  has two parse trees



- Typically we want the second form

The Dangling Else: A Fix

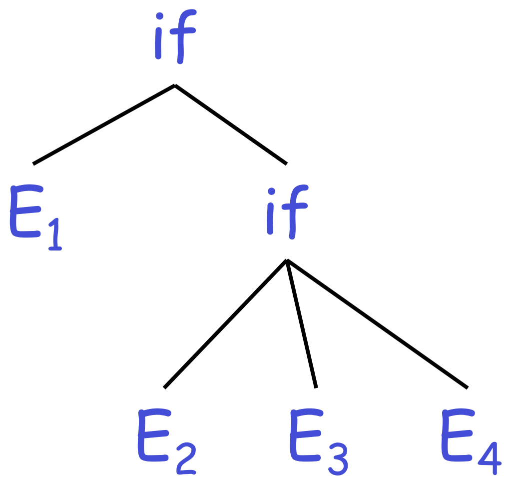
- else matches the closest unmatched then
- We can describe this in the grammar (distinguish between matched and unmatched "then")

```
E →   MIF                    /* all then are matched */
    | UIF                    /* some then are unmatched */
MIF → if E then MIF else MIF
    |  OTHER
UIF → if E then E
    |  if E then MIF else UIF
```

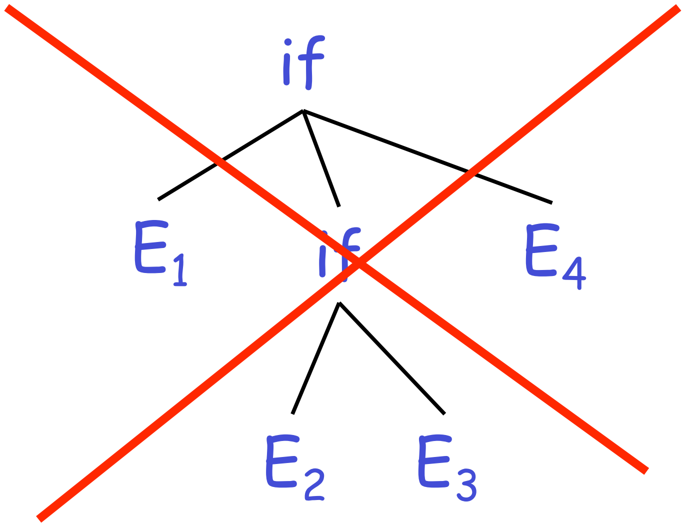
- Describes the same set of strings

The Dangling Else: Example Revisited

- The expression if E_1 then if E_2 then E_3 else E_4



- A valid parse tree (for a UIF)



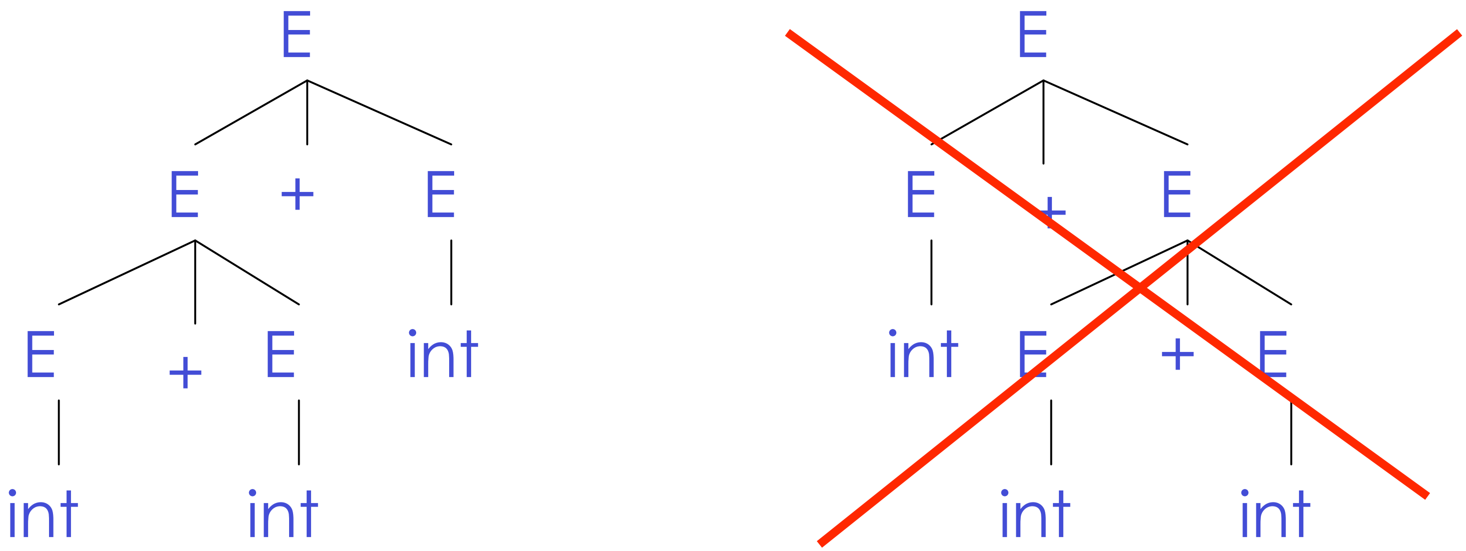
- Not valid because the then expression is not a MIF

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
  - But we need disambiguation mechanisms
- Instead of rewriting the grammar
  - Use the more natural (ambiguous) grammar
  - Along with disambiguating declarations
- Most tools allow *precedence and associativity declarations* to disambiguate grammars
- Examples …

Associativity Declarations

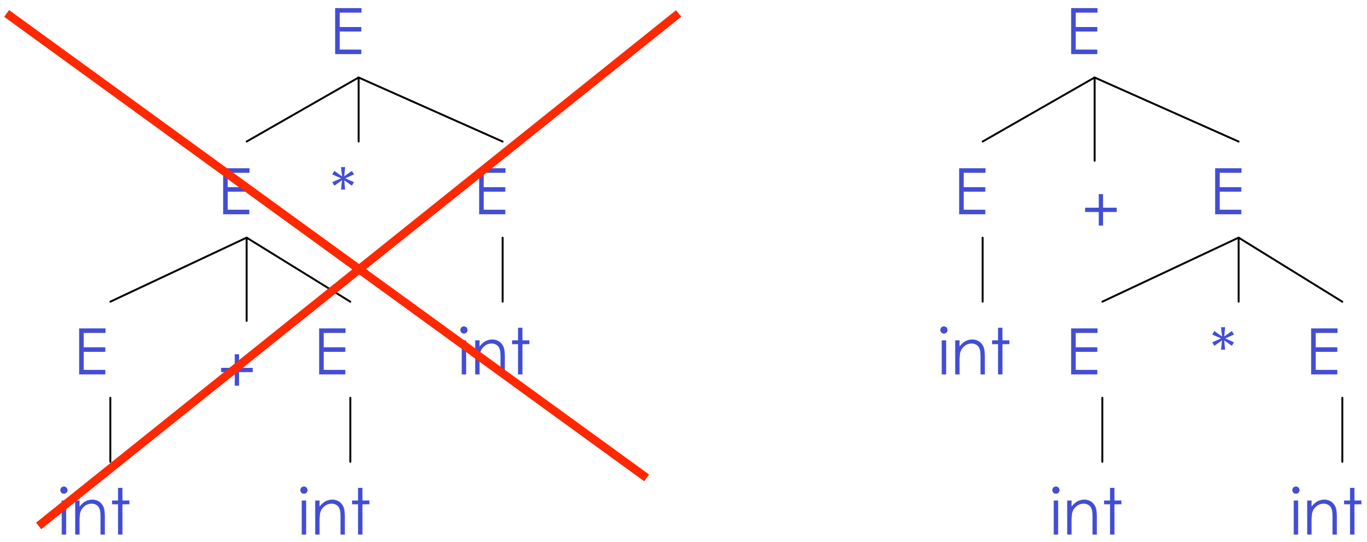
- Consider the grammar $E \rightarrow E + E \mid \text{int}$
- Ambiguous: two parse trees of int + int + int



- Left-associativity declaration: %left '+'

Precedence Declarations

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



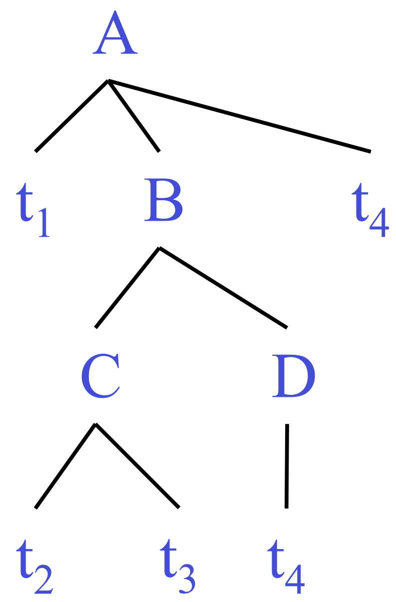
- Precedence declarations: %left '+'
  %left '*'

How It's Done I: Intro to Top-Down Parsing

- Terminals are seen in order of appearance in the token stream:

  $t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5$

- The parse tree is constructed
  - From the top
  - From left to right
- … As for leftmost derivation



Top-down Depth-First Parsing

- Consider the grammar

  $E \rightarrow T + E \mid T$

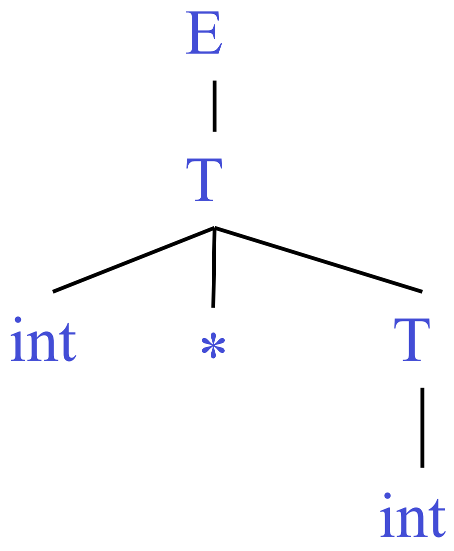
  $T \rightarrow (E) \mid \text{int} \mid \text{int} * T$
- Token stream is: int * int
- Start with top-level non-terminal E
- Try the rules for E in order

Depth-First Parsing. Example int * int

• Start with start symbol	E
• Try E → T + E	T + E
• Then try a rule for T → (E)	(E) + E
– But (≠ input int; backtrack to	T + E
• Try T → int . Token matches.	int + E
– But + ≠ input *; back to	T + E
• Try T → int * T	int*T+E
– But (skipping some steps) + can't be matched	
• Must backtrack to	E

Depth-First Parsing. Example int * int

- Try E → T
- Follow same steps as before for T
  - And succeed with T → int * T and T → int
  - With the following parse tree



Depth-First Parsing

- Parsing: given a string of tokens $t_1\ t_2\ \ldots\ t_n$, find a leftmost derivation (and thus, parse tree)
- Depth-first parsing: Beginning with start symbol, try each production exhaustively on leftmost non-terminal in current sentential form and recurse.

Depth-First Parsing of $t_1\ t_2\ \ldots\ t_n$

- At a given moment, have sentential form that looks like this: $t_1\ t_2\ \ldots\ t_k\ A\ \ldots,\ 0 \le k \le n$
- Initially, $k=0$ and $A\ldots$ is just start symbol
- Try a production for A: if $A \rightarrow BC$ is a production, the new form is $t_1\ t_2\ \ldots\ t_k\ B\ C\ \ldots$
- Backtrack when leading terminals aren't prefix of $t_1\ t_2\ \ldots\ t_n$ and try another production
- Stop when no more non-terminals and terminals all matched (accept) or no more productions left (reject)

When Depth-First Doesn't Work Well

- Consider productions $S \rightarrow S\ a \mid a$:
  - In the process of parsing S we try the above rules
  - Applied consistently in this order, get infinite loop
  - Could re-order productions, but search will have lots of backtracking and general rule for ordering is complex
- Problem here is *left-recursive grammar:* one that has a non-terminal S

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

Elimination of Left Recursion

- Consider the left-recursive grammar

$$S \rightarrow S\ \alpha \mid \beta$$

- S generates all strings starting with a β and followed by a number of α

- Can rewrite using right-recursion

$$S \rightarrow \beta\ S'$$
$$S' \rightarrow \alpha\ S' \mid \varepsilon$$

Elimination of left Recursion. Example

- Consider the grammar

$S \rightarrow 1 \mid S\,0$ ($\beta = 1$ and $\alpha = 0$)

can be rewritten as

$S \rightarrow 1\,S'$

$S' \rightarrow 0\,S' \mid \varepsilon$

More Elimination of Left Recursion

- In general

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

- 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

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

General Left Recursion

- The grammar

  $S \rightarrow A\ \alpha \mid \delta$ (1)

  $A \rightarrow S\ \beta$ (2)

  is also left-recursive because

  $S \rightarrow^{+} S\ \beta\ \alpha$
- This left recursion can also be eliminated by first substituting (2) into (1)
- There is a general algorithm (e.g. Aho, Sethi, Ullman §4.3)
- But personally, I'd just do this by hand.

An Alternative Approach

- Instead of reordering or rewriting grammar, can use *top-down breadth-first search.*

  S → S a | a   String: aaa

S

S a   ~~a~~   (string not all matched)

S a a   ~~a a~~

S a a a   a a a

Summary of Top-Down Parsing So Far

- Simple and general parsing strategy
  - Left recursion must be eliminated first
  - … but that can be done automatically
  - Or can use breadth-first search
- But backtracking (depth-first) or maintaining list of possible sentential forms (breadth-first) can make it slow
- Often, though, we can avoid both …

Predictive Parsers

- Modification of depth-first parsing in which parser “predicts” 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”
- In practice, LL(1) is used

LL(1) Languages

- Previously, for each non-terminal and input token there may be a choice of production
- LL*(k)* means that for each non-terminal and *k* tokens, there is only one production that could lead to success

Recursive Descent: Grammar as Program

- In recursive descent, we think of a grammar as a program.
- Each non-terminal is turned into a procedure
- Each right-hand side transliterated into part of the procedure body for its non-terminal
- First, define
  - next() current token of input
  - scan(*t*) check that next()=*t* (else ERROR), and then read new token.

Recursive Descent: Example

$P \rightarrow S\ \$$ $S \rightarrow T\ S'$

$S' \rightarrow +\ S \mid \varepsilon$ $T \rightarrow \text{int} \mid (\ S\ )$

(\$ = end marker)

```
def P (): S(); scan($)
def S(): T(); S'()
def S'():
    if next() == '+': scan('+'); S()
    elif next() in [')', $]: pass
    else: ERROR
def T():
    if next () == int: scan(int)
    elif next() == '(': scan('('); S(); scan (')')
    else: ERROR
```

But where do tests come from?

Predicting Right-hand Sides

- The if-tests are conditions by which parser *predicts* which right-hand side to use.
- In our example, used only next symbol (*LL(1));* but could use more.
- 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

But First: Left Factoring

- With the grammar

  E → T + E | T

  T → int | int * T | (E)

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

- A grammar must be *left-factored* before use for predictive parsing

Left-Factoring Example

- Starting with the grammar

  $E \rightarrow T + E \mid T$

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

- Factor out common prefixes of productions

  $E \rightarrow T X$

  $X \rightarrow + E \mid \varepsilon$

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

  $Y \rightarrow * T \mid \varepsilon$

LL(1) Parsing Table Example

- Left-factored grammar

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

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

- The LL(1) parsing table ($ is a special end marker):

	int	*	+	(	)	$
T	int Y			(E)		
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 first 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
- We reject when we encounter an error state
- We accept when we encounter end-of-input

LL(1) Parsing Algorithm

```
initialize stack = <S,$>
repeat
   case stack of
      <X, rest>  : if T[X,next()] == $Y_1 \ldots Y_n$:
                        stack ← <$Y_1 \ldots Y_n$ rest>;
                     else:  error ();
      <t, rest>   : scan (t); stack ← <rest>;
until stack == < >
```

LL(1) Parsing Example

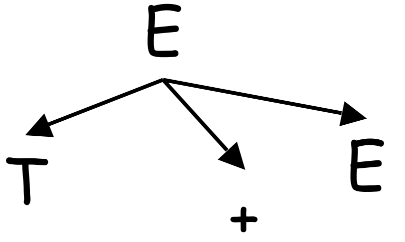
Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	terminal
Y X $	* int $	* T
* T X $	* int $	terminal
T X $	int $	int Y
int Y X $	int $	terminal
Y X $	$	ε
X $	$	ε
$	$	ACCEPT

Constructing Parsing Tables

- LL(1) languages are those definable by a parsing table for the LL(1) algorithm
- No table entry can be multiply defined
- Once we have the table
  - Can create table-driver or recursive-descent parser
  - The parsing algorithms are simple and fast
  - No backtracking is necessary
- We want to generate parsing tables from CFG

Top-Down Parsing. Review

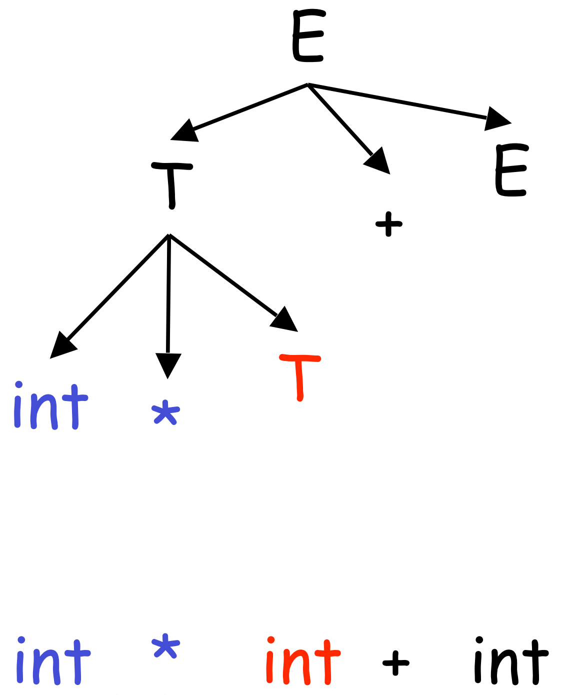
- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal



int * int + int

Top-Down Parsing. Review

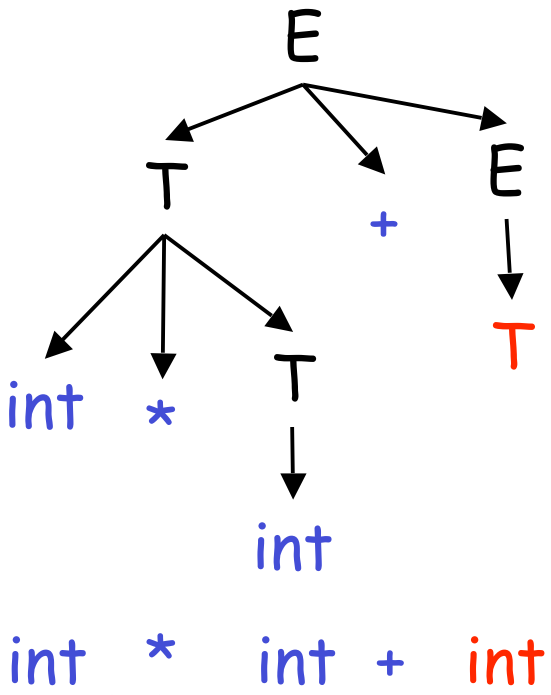
- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal



- The leaves at any point form a string $\beta A \gamma$
  - β contains only terminals
  - The input string is $\beta b \delta$
  - The prefix β matches
  - The next token is b

Top-Down Parsing. Review

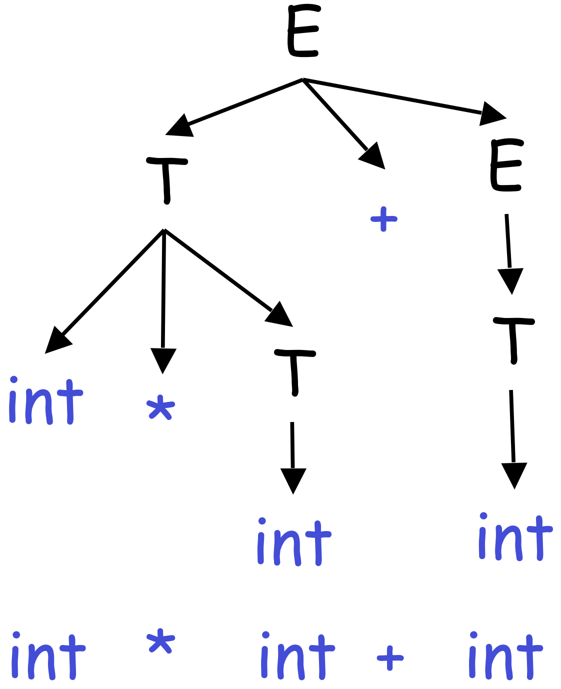
- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal



- The leaves at any point form a string βAγ
  - β contains only terminals
  - The input string is βbδ
  - The prefix β matches
  - The next token is b

Top-Down Parsing. Review

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal



- The leaves at any point form a string $\beta A \gamma$
  - β contains only terminals
  - The input string is $\beta b \delta$
  - The prefix β matches
  - The next token is b

Constructing Predictive Parsing Tables

- Consider the state $S \rightarrow^* \beta A \gamma$
  - With b the next token
  - Trying to match $\beta b \delta$

There are two possibilities:

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

     In this case we say that $b \in \text{First}(\alpha)$

Or…

Constructing Predictive Parsing Tables (Cont.)

2. b does not belong to an expansion of A
   - The expansion of A is empty and b belongs to an expansion of γ (e.g., bω)
   - Means that b can appear after A in a derivation of the form $S \rightarrow^* \beta A b \omega$
   - We say that b $\in$ Follow(A) in this case

   - What productions can we use in this case?
     - Any $A \rightarrow \alpha$ can be used if α can expand to ε
     - We say that $\varepsilon \in$ First(A) in this case

Summary of Definitions

- For $b \in T$, the set of terminals; α a sequence of terminal & non-terminal symbols, S the start symbol, $A \in N$, the set of non-terminals:
- $\mathrm{FIRST}(\alpha) \subseteq T \cup \{ \varepsilon \}$

  $b \in \mathrm{FIRST}(\alpha)$ iff $\alpha \to^* b \ldots$

  $\varepsilon \in \mathrm{FIRST}(\alpha)$ iff $\alpha \to^* \varepsilon$
- $\mathrm{FOLLOW}(A) \subseteq T$

  $b \in \mathrm{FOLLOW}(A)$ iff $S \to^* \ldots A\ b \ldots$

Computing First Sets

Definition $First(X) = \{ b \mid X \rightarrow^* b\alpha\} \cup \{\varepsilon \mid X \rightarrow^* \varepsilon\}$, X any grammar symbol.

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

2. For all productions $X \rightarrow A_1 \ldots A_n$
   - 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 $First(X)$

Computing First Sets, Contd.

- That takes care of single-symbol case.
- In general:

$$
\begin{aligned}
&\mathrm{FIRST}(X_1\ X_2 \ldots X_k) = \\
&\qquad \mathrm{FIRST}(X_1) \\
&\quad \cup\ \mathrm{FIRST}(X_2) \quad \text{if } \varepsilon \in \mathrm{FIRST}(X_1) \\
&\quad \cup\ \ldots \\
&\quad \cup\ \mathrm{FIRST}(X_2) \quad \text{if } \varepsilon \in \mathrm{FIRST}(X_1 X_2 \ldots X_{k-1}) \\
&\quad -\ \{\ \varepsilon\ \} \text{ unless } \varepsilon \in \mathrm{FIRST}(X_i) \quad \forall\ i
\end{aligned}
$$

First Sets. Example

- For the grammar

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

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

- First sets

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

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

First(int) = { int } $\qquad$ First(X) = {+, ε }

First(+) = { + } $\qquad$ First(Y) = {*, ε }

First(*) = { * }

Computing Follow Sets

Definition $\text{Follow}(X) = \{ b \mid S \rightarrow^* \beta X b \omega \}$

1. Compute the First sets for all non-terminals first
2. Add \$ to Follow(S) (if S is the start non-terminal)

3. For all productions $Y \rightarrow \ldots X A_1 \ldots A_n$
   - Add $\text{First}(A_1) - \{\varepsilon\}$ to Follow(X). Stop if $\varepsilon \notin \text{First}(A_1)$
   - Add $\text{First}(A_2) - \{\varepsilon\}$ to Follow(X). Stop if $\varepsilon \notin \text{First}(A_2)$
   - …
   - Add $\text{First}(A_n) - \{\varepsilon\}$ to Follow(X). Stop if $\varepsilon \notin \text{First}(A_n)$
   - Add Follow(Y) to Follow(X)

Follow Sets. Example

- For the grammar

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

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

- Follow sets

Follow(E) = {), $}

Follow(X) = {$,) }

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

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

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 $\alpha \rightarrow^{*} \varepsilon$, for each $b \in Follow(A)$ do
    - $T[A, b] = \alpha$

Constructing LL(1) Tables. Example

- For the grammar

  E → T X   X → + E | ε
  T → (E) | int Y   Y → * T | ε

- Where in the line of Y do we put Y →* T ?
  - In the lines of First(*T) = { * }
- Where in the line of Y do we put Y → ε ?
  - In the lines 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 is left recursive
  - If G 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

Recursive Descent for Real

- So far, have presented a purist view.
- In fact, use of recursive descent makes life simpler in many ways if we "cheat" a bit.
- Here's how you *really* handle left recursion in recursive descent, for $S \rightarrow S\ A \mid R$:

```
def S():
    R ()
    while next() ∈ FIRST(A):
        A()
```

- It's a program: all kinds of shortcuts possible.

Review

- For some grammars there is a simple parsing strategy
  - Predictive parsing (LL(1))
  - Once you build the LL(1) table, you can write the parser by hand

- Next: a more powerful parsing strategy for grammars that are not LL(1)