40-414 Compiler Design

Introduction to Parsing

Lecture 4

Languages and Automata

- Formal languages are very important in CS
  - Especially in programming languages
- Regular languages
  - The weakest formal languages widely used
  - Many applications
- We will also study context-free languages

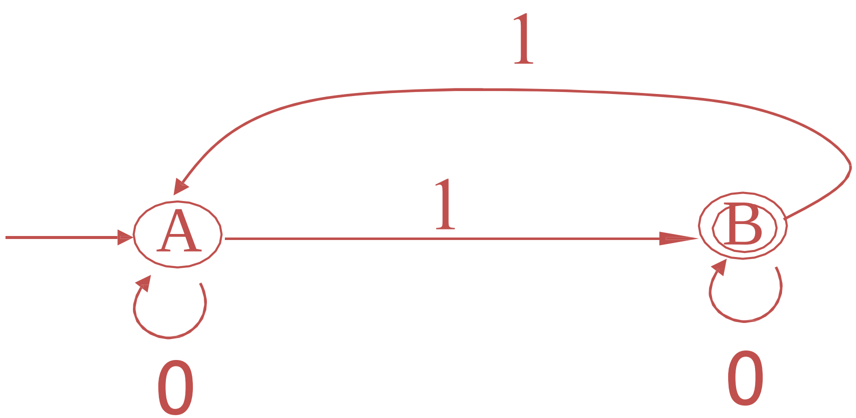
Beyond Regular Languages

- Many languages are not regular

- Strings of balanced parentheses are not regular:

$$\left\{ (^i\,)^i \mid i \geq 0 \right\}$$

What Can Regular Languages Express?



What Can Regular Languages Express?

- Languages requiring counting modulo a fixed integer

- Intuition: A finite automaton that runs long enough must repeat states

- Finite automaton can't remember # of times it has visited a particular state

The Functionality of the Parser

- **Input:** sequence of tokens from lexer

- **Output:** parse tree of the program
  (But some parsers never produce a parse tree . . .)

Example

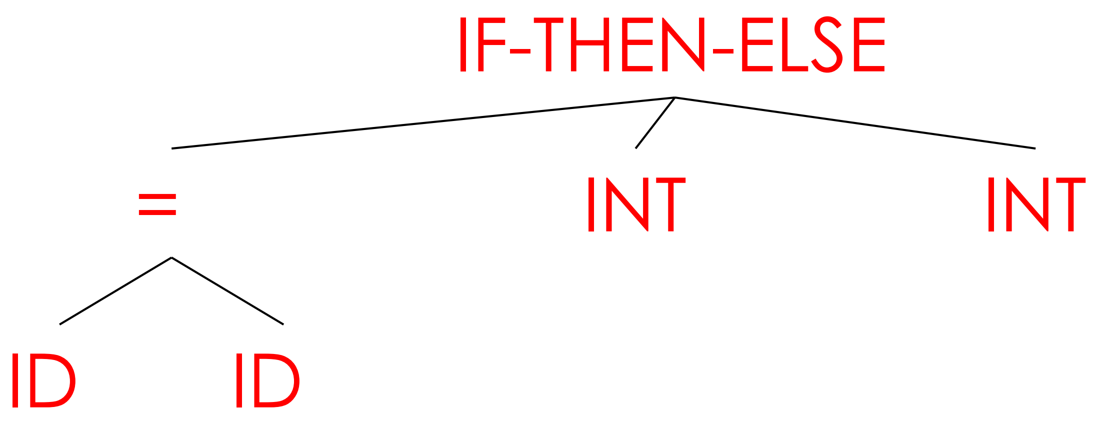
- Cool

  if x = y then 1 else 2 fi

- Parser input

  IF ID = ID THEN INT ELSE INT FI

- Parser output



Comparison with Lexical Analysis

Phase	*Input*	*Output*
Lexer	String of characters	String of tokens
Parser	String of tokens	Parse tree (may be implicit)

The Role of the Parser

- Not all strings of tokens are programs . . .
- . . . parser must distinguish between valid and invalid strings of tokens

- We need
  - A language for describing valid strings of tokens
  - A method for distinguishing valid from invalid strings of tokens

Chomsky Hierarchy

0 Unrestricted	$\alpha A\beta \to \alpha\gamma\beta$
1 Context-Sensitive	$\mid LHS \mid \leq \mid RHS \mid$
2 Context-Free	$\mid LHS \mid = 1$
3 Regular	$\mid RHS \mid = 1$ or 2 , $A \to a \mid aB$, or $A \to a \mid Ba$

Context-Free Grammars

- Programming language constructs have recursive structure

- An STMT is

  if EXPR then STMT else STMT

  while EXPR do STMT end

  …

- Context-free grammars are a natural notation for this recursive structure

CFGs (Cont.)

- A CFG consists of
  - A set of *terminals* T
  - A set of *non-terminals* N
  - A *start symbol* S (a non-terminal)
  - A set of *productions*

$$X \to Y_1 Y_2 \ldots Y_n$$

$$\text{where } X \in N \text{ and } Y_i \in T \cup N \cup \{\varepsilon\}$$

Notational Conventions

- In these lecture notes
  - Non-terminals are written upper-case
  - Terminals are written lower-case
  - The start symbol is the left-hand side of the first production

Example of CFGs

Simple arithmetic expressions:

$$
\begin{array}{lll}
E & \rightarrow & E * E \\
  & | & E + E \\
  & | & (E) \\
  & | & \mathrm{id}
\end{array}
$$

The Language of a CFG

Read productions as replacement rules:

$$X \rightarrow Y_1 \ldots Y_n$$

Means X can be replaced by $Y_1 \ldots Y_n$

Key Idea

1. Begin with a string consisting of the start symbol "S"
2. Replace any non-terminal X in the string by a the right-hand side of some production

$$X \to Y_1 \ldots Y_n$$

3. Repeat (2) until there are no non-terminals in the string

The Language of a CFG (Cont.)

More formally, write

$$X_1 \ldots X_{i-1}\, X_i\, X_{i+1} \ldots X_n \rightarrow X_1 \ldots X_{i-1}\, Y_1 \ldots Y_m\, X_{i+1} \ldots X_n$$

if there is a production

$$X_i \rightarrow Y_1 \ldots Y_m$$

The Language of a CFG (Cont.)

Write

$$X_1 \ldots X_n \rightarrow^* y_1 \ldots y_m$$

if

$$X_1 \ldots X_n \rightarrow \ldots \rightarrow \ldots \rightarrow y_1 \ldots y_m$$

in 0 or more steps

The Language of a CFG

Let G be a context-free grammar with start symbol S. Then the language $L(G)$ of G is:

$$\{a_1 \ldots a_n \mid S \rightarrow^* a_1 \ldots a_n \text{ and every } a_i \text{ is a terminal}\}$$

The sentential forms $SF(G)$ of G is:

$$\{X_1 \ldots X_n \mid S \rightarrow^* X_1 \ldots X_n \text{ and every } X_i \text{ is a terminal or non-terminal}\}$$

Therefore: $L(G) \subset SF(G)$

Terminals

- Terminals are so-called because there are no rules for replacing them

- Once generated, terminals are permanent

- Terminals ought to be tokens of the language

Examples

$L(G)$ is the language of CFG G

Strings of balanced parentheses $\left\{ (^i\,)^i \mid i \geq 0 \right\}$

Two grammars:

$$\begin{array}{lll} S & \to & (S) \\ S & \to & \varepsilon \end{array} \qquad \text{OR} \qquad \begin{array}{lll} S & \to & (S) \\ & \mid & \varepsilon \end{array}$$

Arithmetic Example

Simple arithmetic expressions:

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

Some elements of the language:

id	id + id
(id)	id * id
(id) * id	id * (id)

Notes

The idea of a CFG is a big step. But:

- Membership in a language is "yes" or "no"; also need parse tree of the input

- Must handle errors gracefully

- Need an implementation of CFG's (e.g., bison)

Derivations and Parse Trees

A *derivation* is a sequence of productions

$$S \to \ldots \to \ldots \to \ldots \to \ldots \to \ldots$$

A derivation can be drawn as a tree

- Start symbol is the tree's root
- For a production $X \to Y_1 \ldots Y_n$ add children $Y_1 \ldots Y_n$ to node X

Derivation Example

- Grammar

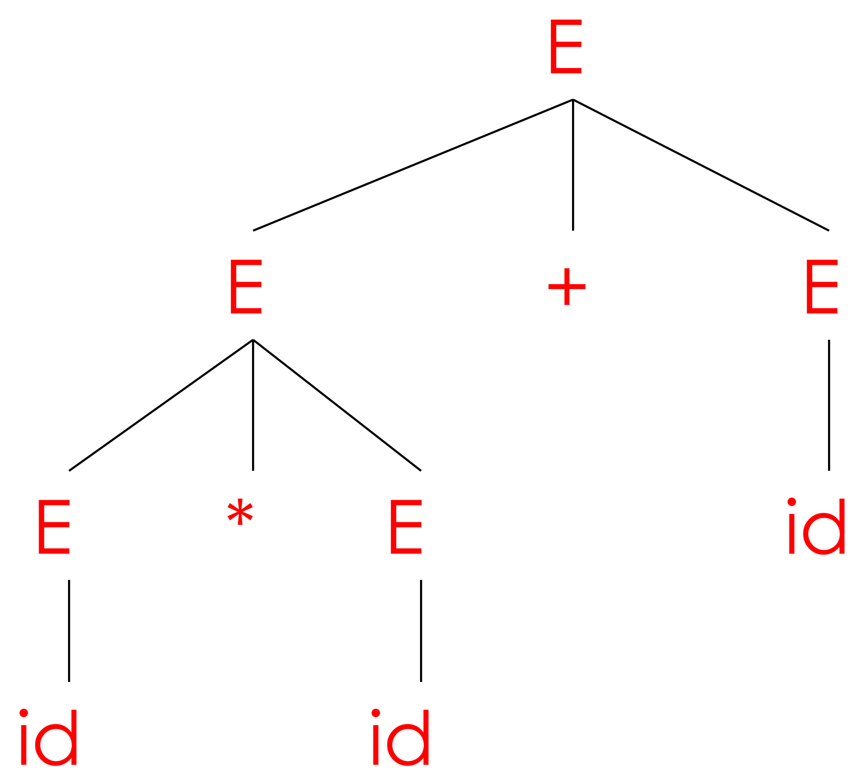
$$E \rightarrow E+E \mid E * E \mid (E) \mid id$$

- String

$$id * id + id$$

Derivation Example (Cont.)

$$
\begin{aligned}
& E \\
\rightarrow\quad & E+E \\
\rightarrow\quad & E * E+E \\
\rightarrow\quad & id * E + E \\
\rightarrow\quad & id * id + E \\
\rightarrow\quad & id * id + id
\end{aligned}
$$

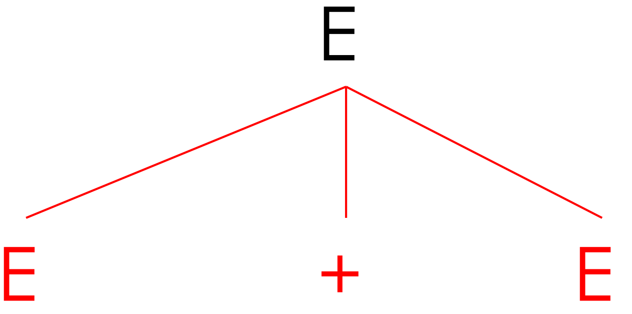


Derivation in Detail (1)

E

E

Derivation in Detail (2)



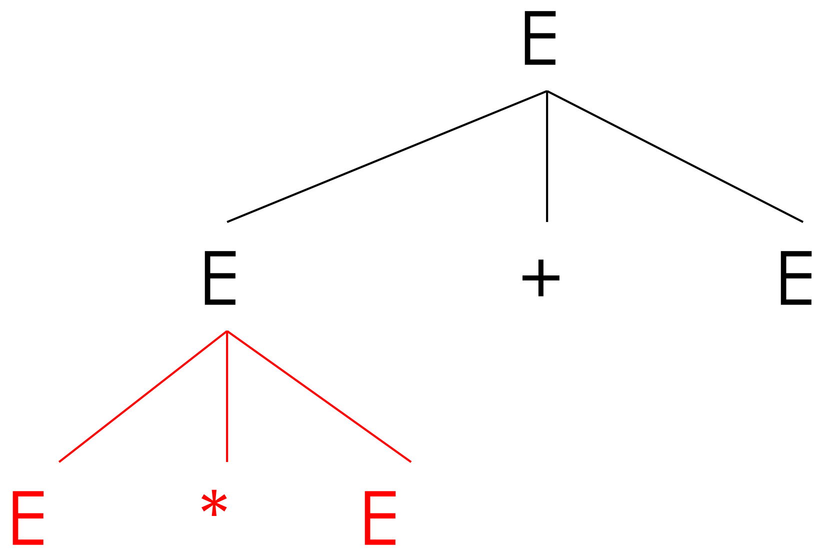
$$
\begin{aligned}
& E \\
\rightarrow\ & E+E
\end{aligned}
$$

Derivation in Detail (3)

$$E$$

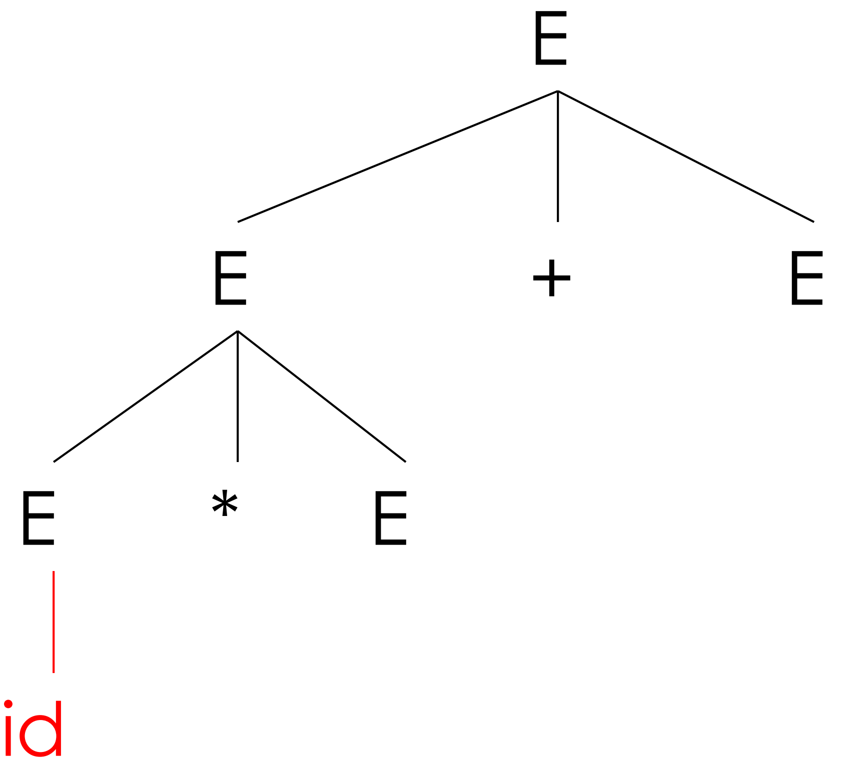
$$\rightarrow \quad E+E$$

$$\rightarrow \quad E * E+E$$



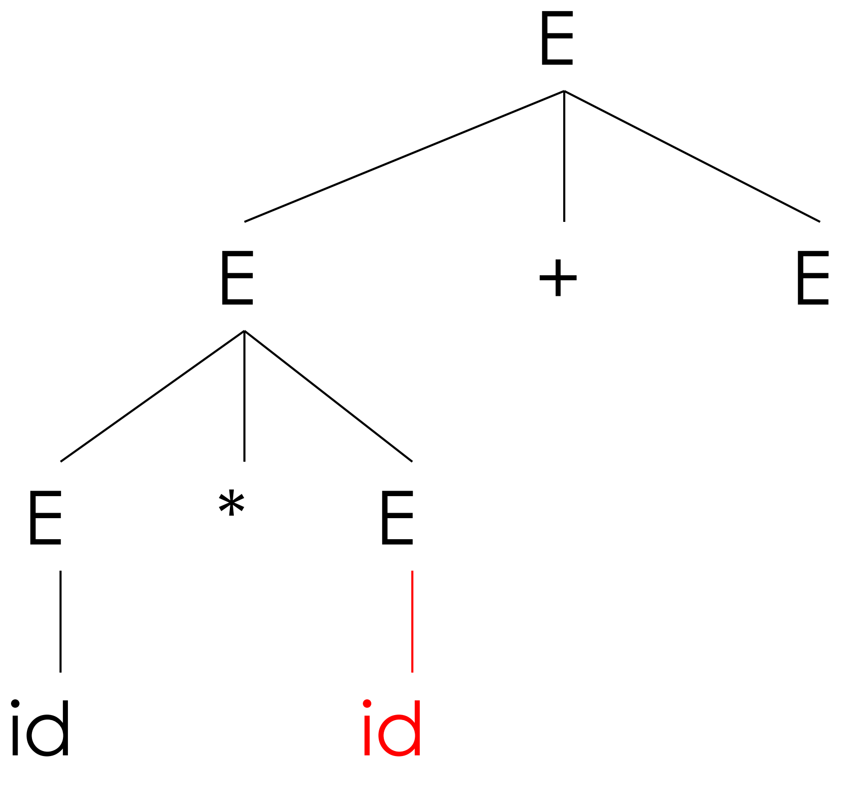
Derivation in Detail (4)

$$
\begin{aligned}
& E \\
\rightarrow \quad & E+E \\
\rightarrow \quad & E * E+E \\
\rightarrow \quad & id * E + E
\end{aligned}
$$



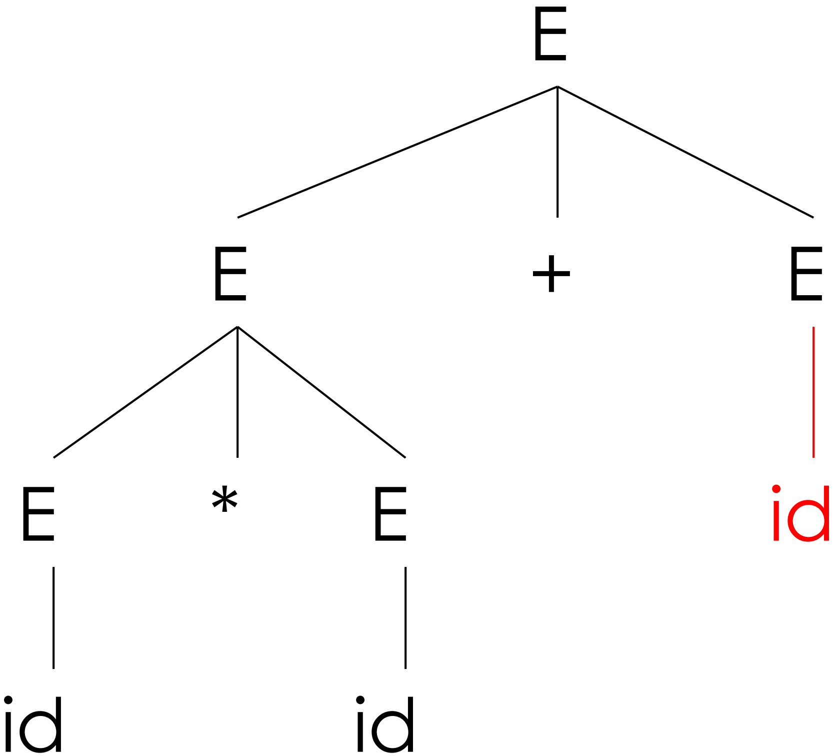
Derivation in Detail (5)

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E*E+E \\
\rightarrow\ & id*E+E \\
\rightarrow\ & id*id+E
\end{aligned}
$$



Derivation in Detail (6)

$$
\begin{aligned}
& E \\
\rightarrow\; & E+E \\
\rightarrow\; & E * E+E \\
\rightarrow\; & id * E + E \\
\rightarrow\; & id * id + E \\
\rightarrow\; & id * id + id
\end{aligned}
$$



Notes on Derivations

- A parse tree has
  - Terminals at the leaves
  - Non-terminals at the interior nodes

- An in-order traversal of the leaves is the original input

- The parse tree shows the association of operations, the input string does not

Left-most and Right-most Derivations

- The example was a *left-most* derivation
  - At each step, replaced the left-most non-terminal

- There is an equivalent notion of a *right-most* derivation

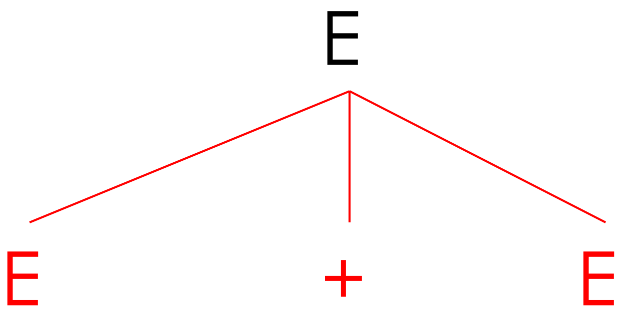
$$
\begin{aligned}
& E \\
\rightarrow\quad & E+E \\
\rightarrow\quad & E+id \\
\rightarrow\quad & E * E + id \\
\rightarrow\quad & E * id + id \\
\rightarrow\quad & id * id + id
\end{aligned}
$$

Right-most Derivation in Detail (1)

E

E

Right-most Derivation in Detail (2)

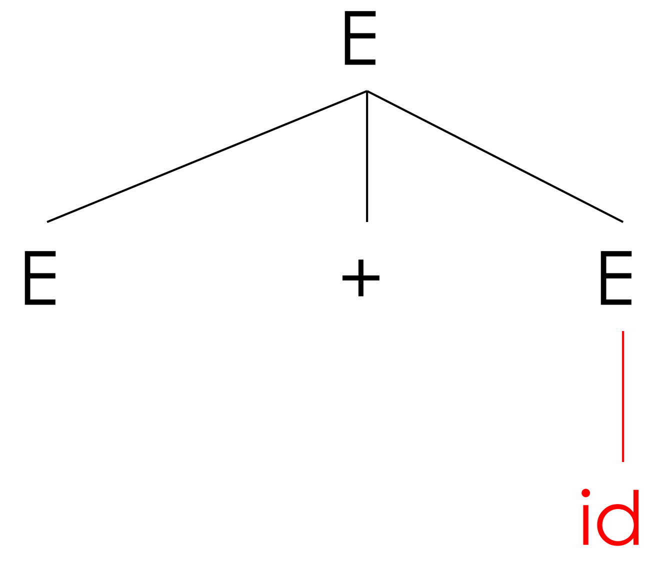


E

$\rightarrow \quad E+E$

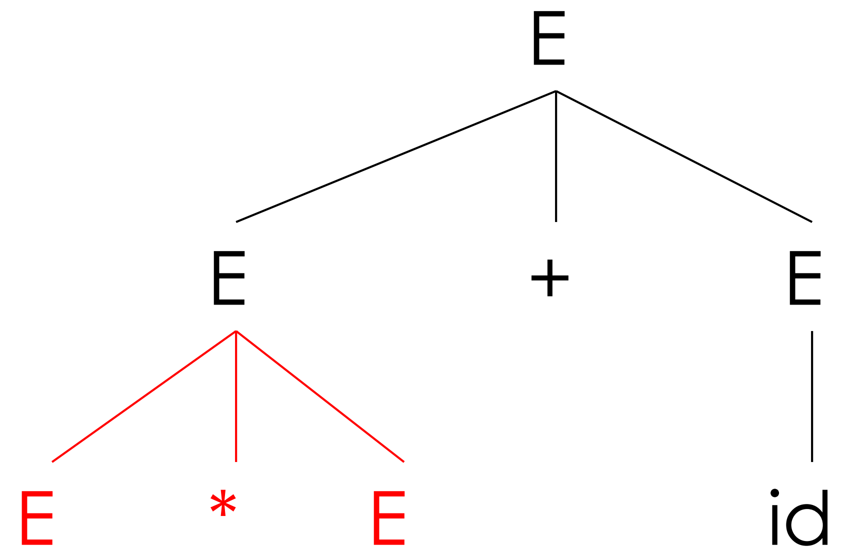
Right-most Derivation in Detail (3)

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id
\end{aligned}
$$



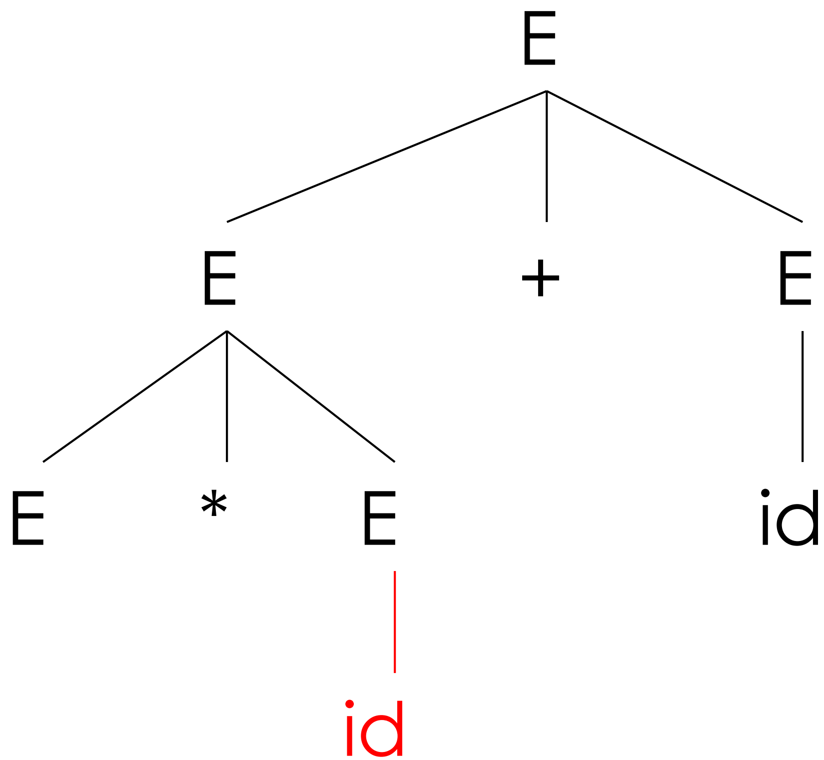
Right-most Derivation in Detail (4)

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id \\
\rightarrow\ & E * E + id
\end{aligned}
$$



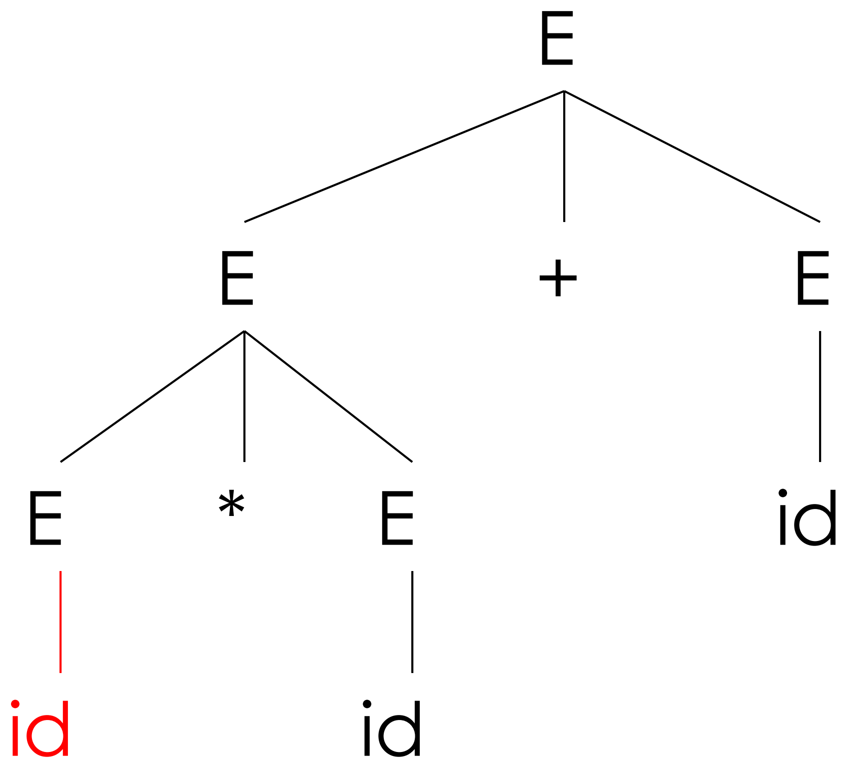
Right-most Derivation in Detail (5)

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id \\
\rightarrow\ & E * E + id \\
\rightarrow\ & E * id + id
\end{aligned}
$$



Right-most Derivation in Detail (6)

$$
\begin{aligned}
& E \\
\rightarrow\; & E+E \\
\rightarrow\; & E+\mathrm{id} \\
\rightarrow\; & E * E + \mathrm{id} \\
\rightarrow\; & E * \mathrm{id} + \mathrm{id} \\
\rightarrow\; & \mathrm{id} * \mathrm{id} + \mathrm{id}
\end{aligned}
$$



Derivations and Parse Trees

- Note that right-most and left-most derivations have the same parse tree

- The difference is the order in which branches are added

Summary of Derivations

- We are not just interested in whether $s \in L(G)$
  - We need a parse tree for s

- A derivation defines a parse tree
  - But one parse tree may have many derivations

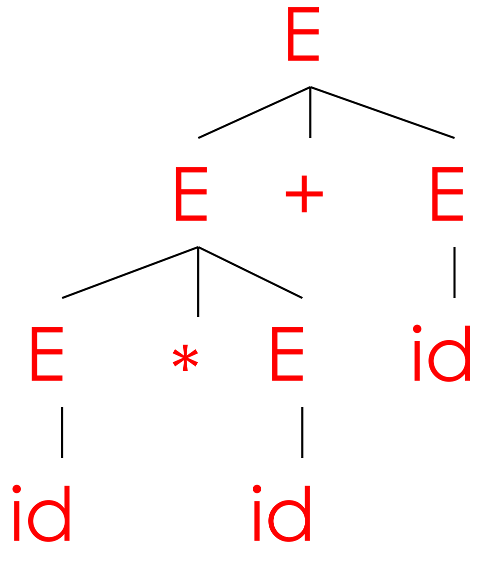
- Left-most and right-most derivations are important in parser implementation

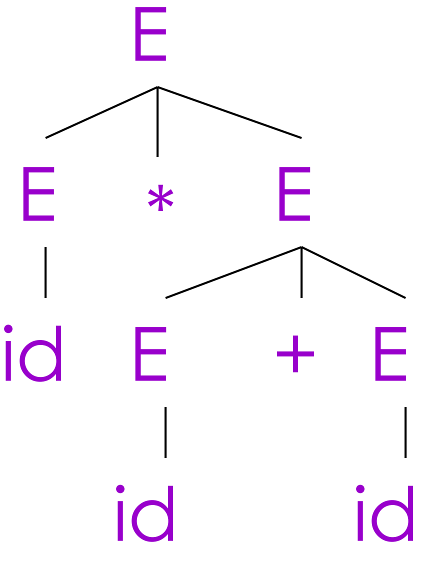
Ambiguity

- Grammar $E \rightarrow E + E \mid E * E \mid (E) \mid id$
- String $id * id + id$

Ambiguity (Cont.)

This string has two parse trees





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 **BAD**
  - Leaves meaning of some programs ill-defined

Dealing with Ambiguity

- There are several ways to handle ambiguity

- Most direct method is to rewrite grammar unambiguously

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

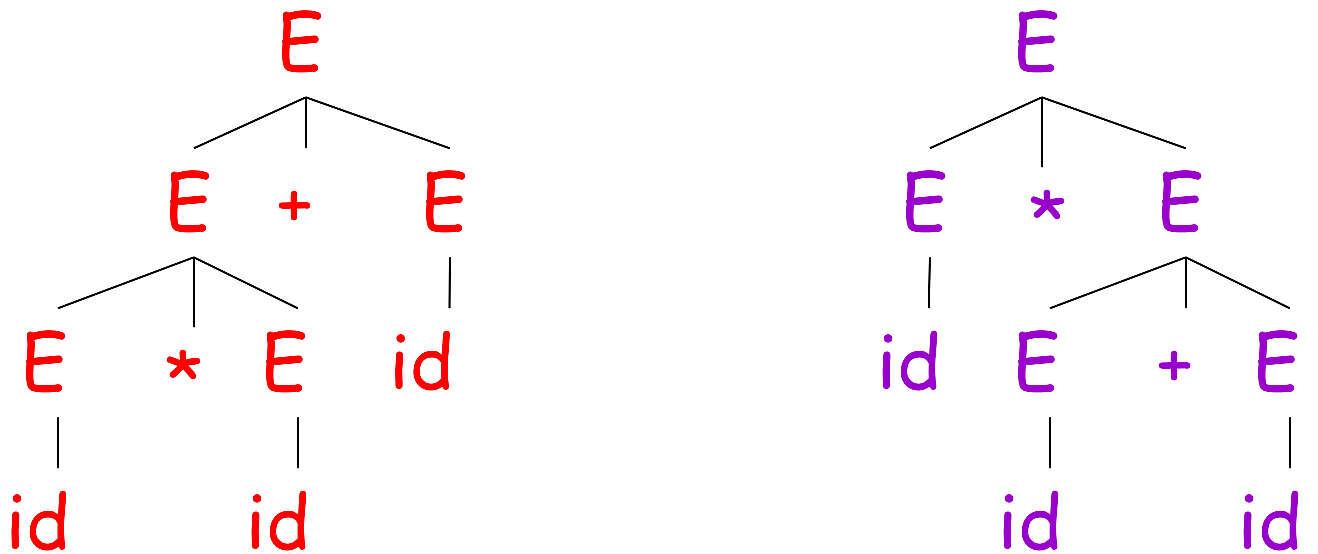
- Enforces precedence of * over +

Ambiguity in Arithmetic Expressions

- Recall the grammar

  $E \rightarrow E + E \mid E * E \mid (E) \mid id$

- The string id * id + id has two parse trees:

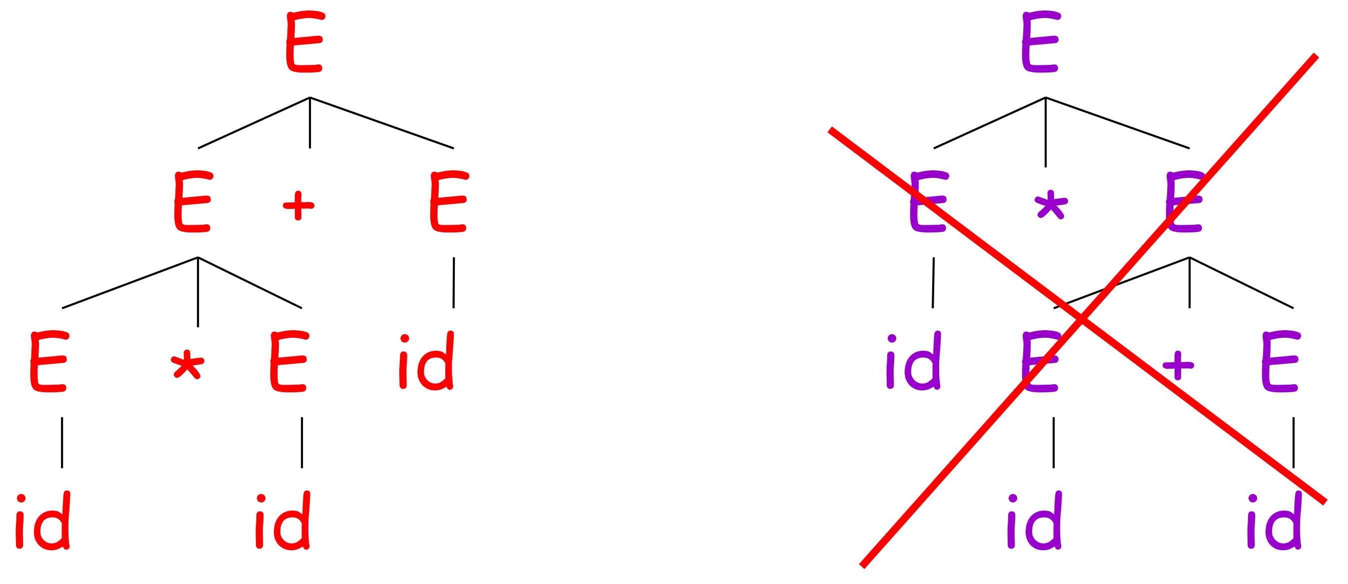


Ambiguity in Arithmetic Expressions

- Recall the grammar

  $E \rightarrow E + E \mid E * E \mid (E) \mid id$

- The string id * id + id has two parse trees:



Ambiguity: The Dangling Else

- Consider the grammar

$$
\begin{aligned}
S \rightarrow\ & \text{if } E \text{ then } S \\
|\ & \text{if } E \text{ then } S \text{ else } S \\
|\ & \text{OTHER}
\end{aligned}
$$

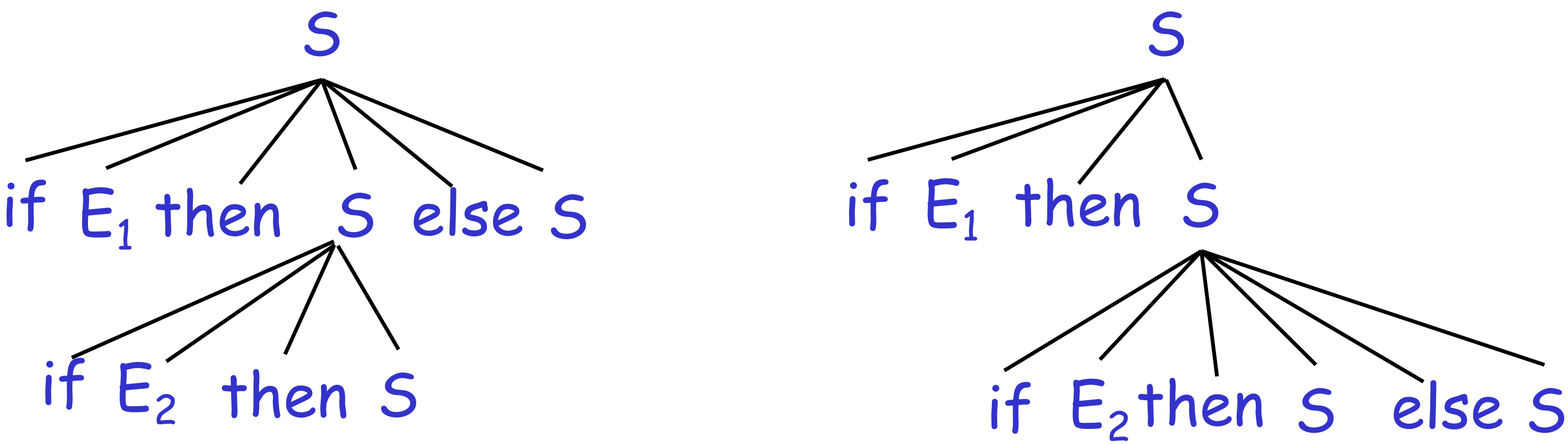
- This grammar is also ambiguous

The Dangling Else: Example

- The expression

  if E_1 then if E_2 then S else S

  has two parse trees



- Typically we want the second form

The Dangling Else: A Fix

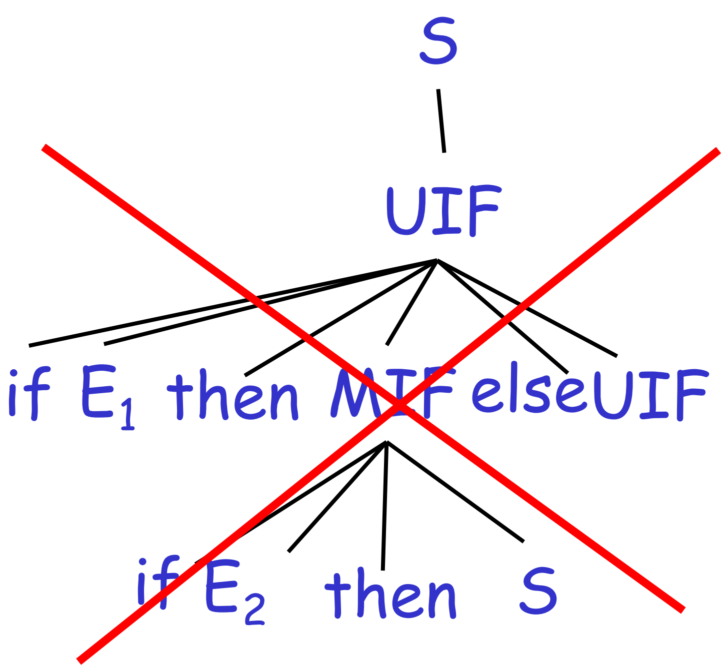
- else matches the closest unmatched then
- We can describe this in the grammar

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

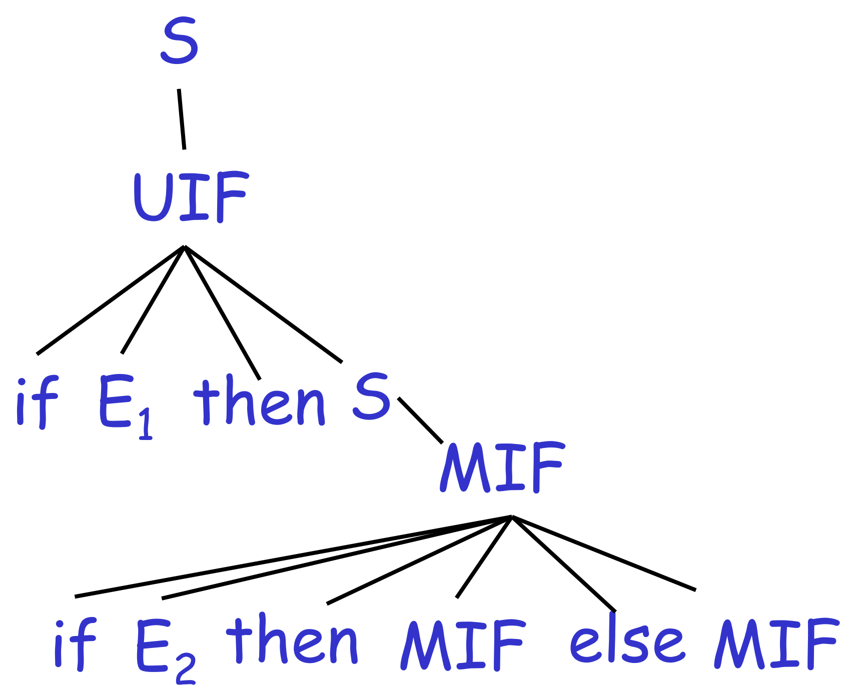
- Describes the same set of strings

The Dangling Else: Example Revisited

- The expression if E_1 then if E_2 then S else S



- Not valid because the then expression is not a MIF



- A valid parse tree (for a UIF)

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

Introduction to Top-Down Parsing

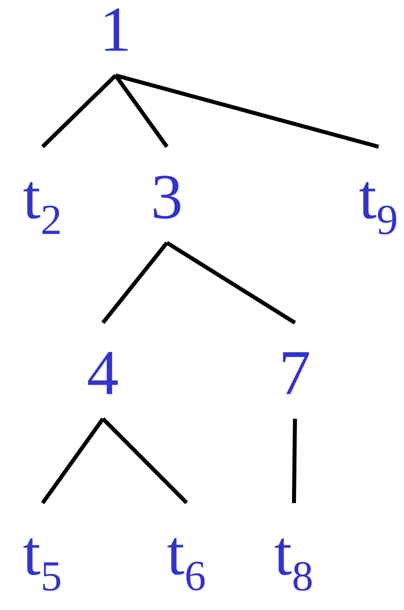
Recursive Descent

Intro to Top-Down Parsing: The Idea

- The parse tree is constructed
  - From the top
  - From left to right

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

  $t_2 \quad t_5 \quad t_6 \quad t_8 \quad t_9$



Recursive Descent Parsing

- Consider the grammar

  $E \rightarrow T \mid T + E$

  $T \rightarrow int \mid int * T \mid (E)$

- Token stream is: (int)

- Start with top-level non-terminal E
  - Try the rules for E in order

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

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

E

(int)

↑

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

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

$$
\begin{array}{c}
E \\
| \\
T
\end{array}
$$

(int)

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

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

E

T

int

Mismatch: int is not (!
Backtrack …

(int)

Recursive Descent Parsing

$E \to T \mid T + E$

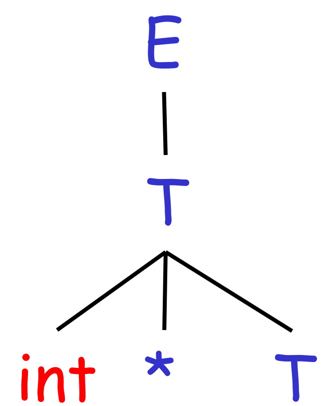
$T \to \text{int} \mid \text{int} * T \mid (E)$

(int)

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

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



Mismatch: int is not (!
Backtrack …

(int)

Recursive Descent Parsing

$E \to T \mid T + E$

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

E

|

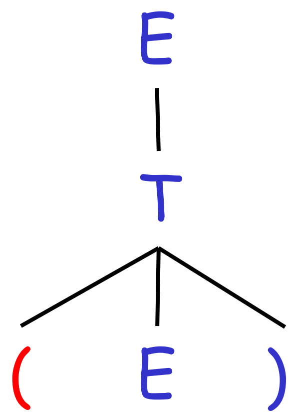
T

(int)

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

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



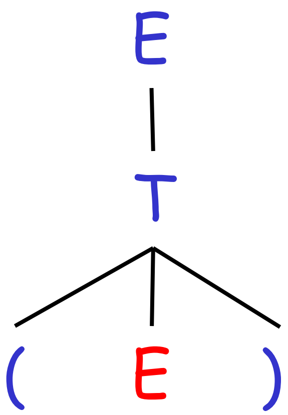
Match! Advance input.

(int)

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

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

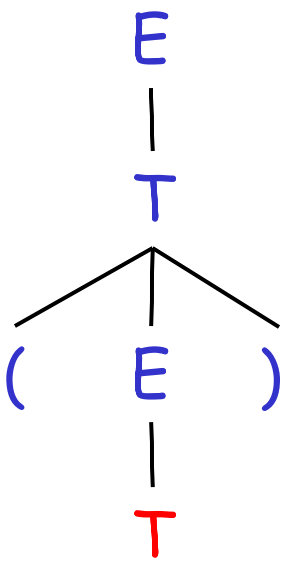


(int)

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

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

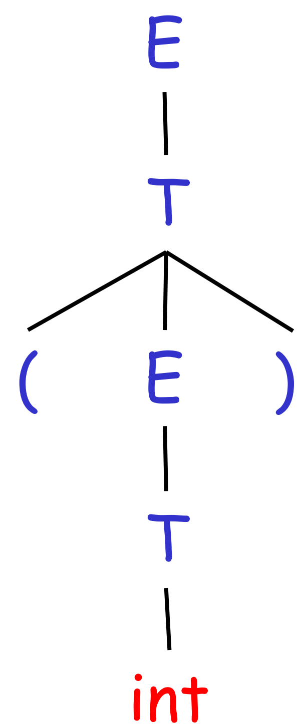


(int)

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

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



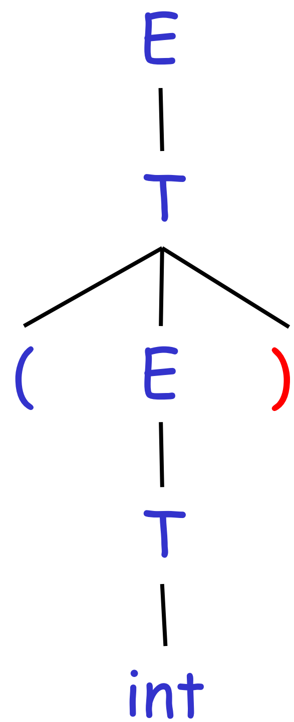
Match! Advance input.

(int)

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$



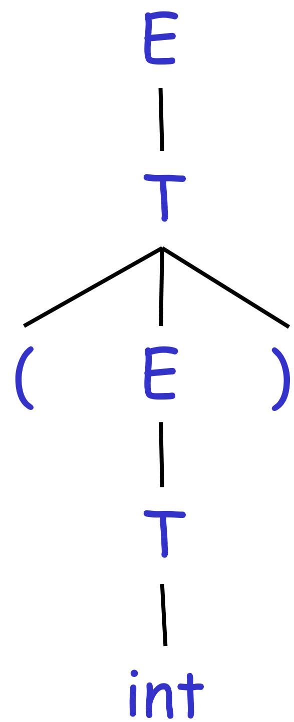
Match! Advance input.

(int)

Recursive Descent Parsing

E → T | T + E

T → int | int * T | (E)



End of input, accept.

(int)

Problems with Top Down Parsing

- Left Recursion in CFG may cause parser to loop forever!

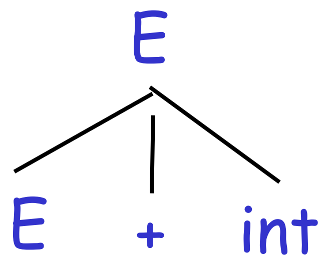
- In there is a production of form $A \to A\alpha$, we say the grammar has left recursion

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

- Solution: Remove Left Recursion...
  - without changing the Language defined by the Grammar.

Problems with Top Down Parsing (Example)

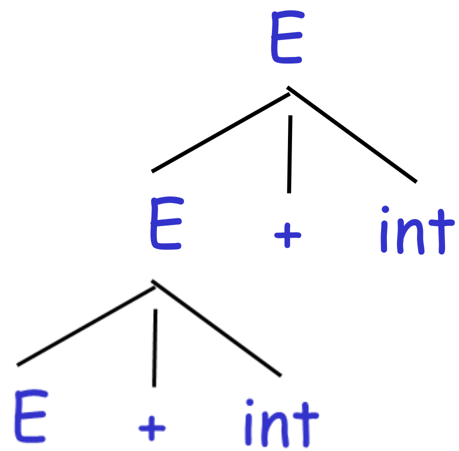
$E \rightarrow E + \text{int} \mid \text{int}$



int + int

Problems with Top Down Parsing (Example)

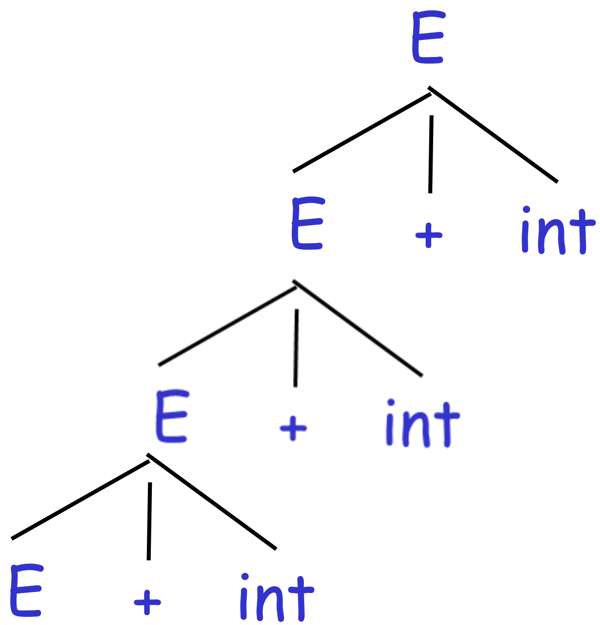
$E \rightarrow E + \text{int} \mid \text{int}$



int + int

Problems with Top Down Parsing (Example)

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



int + int

Elimination of Left Recursion

- Consider the left-recursive grammar

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

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

- Can rewrite using right-recursion

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

More Elimination of Left-Recursion

- In general

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

- 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

$$S \to \beta_1\,S' \mid \dots \mid \beta_m\,S'$$

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

General Left Recursion

- The grammar

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

  $A \rightarrow S\,\beta$

  is also left-recursive because

  $S \rightarrow^{+} S\,\beta\,\alpha$

- This left-recursion can also be eliminated

- See Dragon Book for general algorithm
  - Section 4.3.3

Predictive Top-Down Parsing

Parser Never Backtracks

For Example, Consider:

Type $\rightarrow$ Simple **Start symbol**

| $\uparrow$ **id**

| **array** [Simple] **of** Type

Simple $\rightarrow$ **integer**

| **char**

| **num dotdot num**

Suppose input is :

array [**num dotdot num**] **of integer**

Parsing would begin with

Type $\rightarrow$ **???**

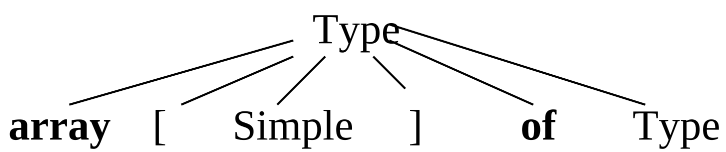
Predictive Parsing Example

Lookahead symbol

Input : **array** [**num dotdot num**] **of integer**

Type

?



Start symbol

Type → simple

| ↑ **id**

| **array** [Simple] **of** Type

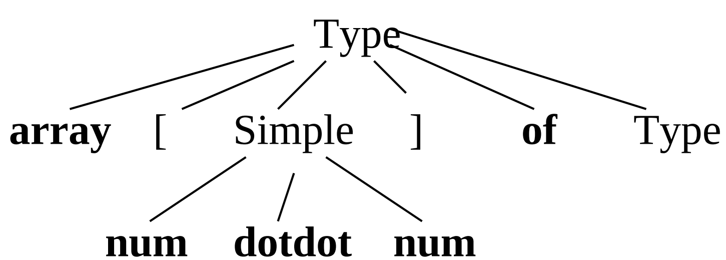
Simple → **integer**

| **char**

| **num dotdot num**

Lookahead symbol

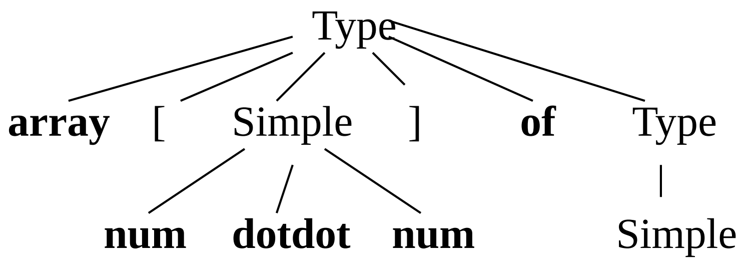
Input : **array** [**num dotdot num**] **of integer**



Predictive Parsing Example

Lookahead symbol

Input : **array** [**num dotdot num**] **of integer**



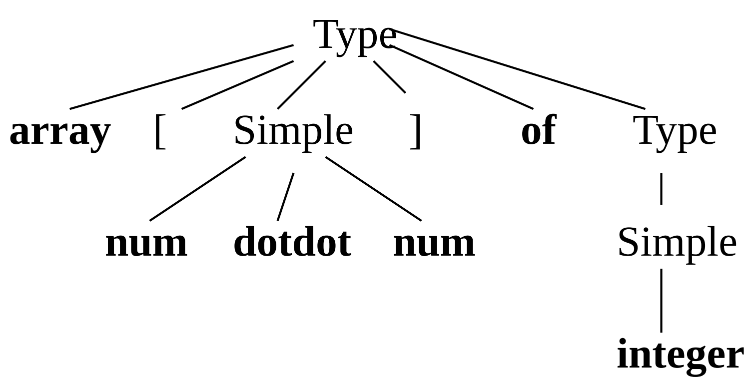
Start symbol

Type → Simple
| ↑ **id**
| **array** [Simple] **of** Type

Simple → **integer**
| **char**
| **num dotdot num**

Lookahead symbol

Input : **array** [**num dotdot num**] **of integer**



Predictive Recursive Descent

- Parser is implemented by N + 1 subroutines, where N is the number grammar non-terminals

- There is one subroutine for attempting to Match tokens in the input stream

- There is also one subroutine for each non-terminal with two tasks:
  1. Deciding on the next production to use
  2. Applying the selected production

Predictive Recursive Descent (Cont.)

Procedure "Match" checks if the token matches the expected input

```
procedure Match ( expected_token ) ;
  {
        if  lookahead = expected_token then
                lookahead  :=  get_next_token
        else error
   }
```

Predictive Recursive Descent (Cont.)

- The subroutine for each non-terminals has two tasks:
  1. Selecting the appropriate production
  2. Applying the chosen production

- Selection is done based on the result of a number of if-then-else statements
- Applying a production is implemented by calling the match procedure or other subroutines, based on the rhs of the production

Predictive Recursive Descent (Cont.)

Subroutine "Simple" for the given example:

Type → Simple
  | ↑ **id**
  | **array** [Simple] **of** Type
Simple → **integer**
  | **char**
  | **num dotdot num**

```
procedure Simple ;
  { if lookahead = integer then call Match ( integer );
      else if  lookahead = char then call Match ( char );
          else if lookahead = num
              then { call Match ( num ); call Match ( dotdot );
                  call Match ( num ) }
              else error
  }
```

Predictive Recursive Descent (Cont.)

Subroutine "Type" for the given example:

Type $\rightarrow$ Simple
  | $\uparrow$ **id**
  | **array** [Simple] **of** Type
Simple $\rightarrow$ **integer**
  | **char**
  | **num dotdot num**

```
procedure Type ;
   {if lookahead  is in { integer, char, num } then call Simple;
       else if lookahead  = '↑'  then  { call Match ( '↑' ) ;  call Match( id ) }
          else if lookahead  = array
                 then { call Match( array ); call Match( '[' ); call Simple;
                         call Match( ']' ); call Match( of ); call Type }
              else  error
   }
```

How to write tests for selecting the appropriate production rule?

Basic Tools:

First: Let α be a string of grammar symbols. First(α) is the set that includes every terminal that appears leftmost in α or in any string originating from α.
NOTE: If $\alpha \Rightarrow \in$, then $\in$ is First(α).

Follow: Let A be a non-terminal. Follow(A) is the set of terminals a that can appear directly to the right of A in some sentential form. ($S \Rightarrow \alpha A a \beta$, for some α and β).
NOTE: If $S \Rightarrow \alpha A$, then \$ is Follow(A).

Computing First Sets

Definition

$$\text{First}(X) = \{ t \mid X \to^* t\alpha \} \cup \{ \varepsilon \mid X \to^* \varepsilon \}$$

Algorithm sketch:

1. $\text{First}(t) = \{ t \}$
2. $\varepsilon \in \text{First}(X)$
   - if $X \to \varepsilon$
   - if $X \to A_1 \ldots A_n$ and $\varepsilon \in \text{First}(A_i)$ for $1 \le i \le n$
3. $\text{First}(\alpha) \subseteq \text{First}(X)$ if $X \to A_1 \ldots A_n\, \alpha$
   - and $\varepsilon \in \text{First}(A_i)$ for $1 \le i \le n$

Computing First(X) for all Grammar Symbols

1. If X is a terminal, First(X) = {X}
2. If $X \rightarrow \in$ is a production rule, add $\in$ to First(X)
3. If X is a non-terminal, and $X \rightarrow Y_1Y_2 \ldots Y_k$ is a production rule

   Place First(Y_1) - $\in$ in First(X)

   if $Y_1 \Rightarrow \in$, Place First(Y_2) - $\in$ in First(X)

   if $Y_2 \Rightarrow \in$, Place First(Y_3) - $\in$ in First(X)

   …

   if $Y_{k-1} \Rightarrow \in$, Place First(Y_k) in First(X)

NOTE: As soon as $Y_i \overset{*}{\not\Rightarrow} \in$, Stop.

Repeat above steps until no more elements are added to any First set.

First Sets. Example

- Recall the grammar

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

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

- 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:

$$\text{Follow}(X) = \{ t \mid S \rightarrow^* \beta\, X\, t\, \delta \}$$

- Intuition
  - If $X \rightarrow A\ B$ then $\text{First}(B) \subseteq \text{Follow}(A)$ and $\text{Follow}(X) \subseteq \text{Follow}(B)$
    - if $B \rightarrow^* \varepsilon$ then $\text{Follow}(X) \subseteq \text{Follow}(A)$
  - If S is the start symbol then $\$ \in \text{Follow}(S)$

Computing Follow Sets (Cont.)

Algorithm sketch:

1. $\$ \in \text{Follow}(S)$
2. $\text{First}(\beta) - \{\varepsilon\} \subseteq \text{Follow}(X)$
   - For each production $A \to \alpha\ X\ \beta$
3. $\text{Follow}(A) \subseteq \text{Follow}(X)$
   - For each production $A \to \alpha\ X\ \beta$ where $\varepsilon \in \text{First}(\beta)$

Follow Sets. Example

- Recall the grammar

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

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

- Follow sets

Follow(E) = {), $}

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

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

Follow(X) = {), $}

Conditions for R.D. Parsing to be Predictive

R.D. parsing is Predictive when for all $A \rightarrow \alpha \mid \beta$

1. First(α) $\cap$ First(β) = $\varnothing$;
   besides, only one of α or β can derive ε

2. if α derives ε, then Follow(A) $\cap$ First(β) = $\varnothing$

Predictive Parsing and Left Factoring

- Consider the grammar

  $E \to T + E \mid T$

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

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

- We need to left-factor the grammar

Left-Factoring Example

- Recall the grammar

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

- Factor out common prefixes of productions

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

Error Handling

- Purpose of the compiler is
  - To detect non-valid programs
  - To translate the valid ones
- Many kinds of possible errors (e.g. in C)

Error kind	Example	Detected by ...
Lexical	... $ ...	Lexer
Syntax	... x *% ...	Parser
Semantic	... int x; y = x(3); ...	Type checker
Correctness	your favorite program	Tester/User

Syntax Error Handling

- Error handler should
  - Report errors accurately and clearly
  - Recover from an error quickly
  - Not slow down compilation of valid code

- Good error handling is not easy to achieve

Approaches to Syntax Error Recovery

- From simple to complex
  - Panic mode
  - Error productions
  - Automatic local or global correction

Error Recovery: Panic Mode

- Simplest, most popular method

- When an error is detected:
  - Discard tokens until one with a clear role is found
  - Continue from there

- Such tokens are called <u>synchronizing</u> tokens
  - Typically the statement or expression terminators

Syntax Error Recovery: Error Productions

- Idea: specify in the grammar known common mistakes

- Essentially promotes common errors to alternative syntax

- Example:
  - Write 5 x instead of 5 * x
  - Add the production $E \to \ldots \mid E\ E$

- Disadvantage
  - Complicates the grammar

Error Recovery: Local and Global Correction

- Idea: find a correct "nearby" program
  - Try token insertions and deletions
  - Exhaustive search

- Disadvantages:
  - Hard to implement
  - Slows down parsing of correct programs
  - "Nearby" is not necessarily "the intended" program

Syntax Error Recovery: Past and Present

- Past
  - Slow recompilation cycle (even once a day)
  - Find as many errors in one cycle as possible

- Present
  - Quick recompilation cycle
  - Users tend to correct one error/cycle
  - Complex error recovery is less compelling
  - Panic-mode seems enough

Question?

How many strings does the following grammar generate?

$$A \to B\ B$$
$$B \to C\ C$$
$$C \to 1 \mid 2$$

- 7
- 15
- 2
- 8
- 16
- 4

Question?

How many strings does the following grammar generate?

$$A \rightarrow B\ B$$
$$B \rightarrow C\ C$$
$$C \rightarrow 1 \mid 2 \mid \varepsilon$$

- 16
- 31
- 15
- 12
- 64
- 63
- 32
- 11

Question?

Which of the following is a valid derivation of the given grammar?

$S \rightarrow aXa$

$X \rightarrow \varepsilon \mid bY$

$Y \rightarrow \varepsilon \mid cXc \mid d$

- S
  aXa
  abYa
  acXca
  acca

- S
  aa

- S
  aXa
  abYa
  abcXca
  abcbYca
  abcbdca

- S
  aXa
  abYa
  abcXcda
  abccda

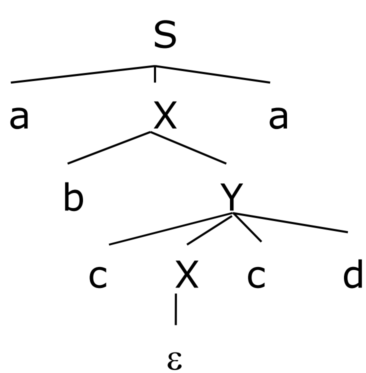
Question?

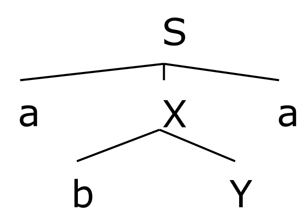
Which of the following is a valid parse tree for the given grammar?

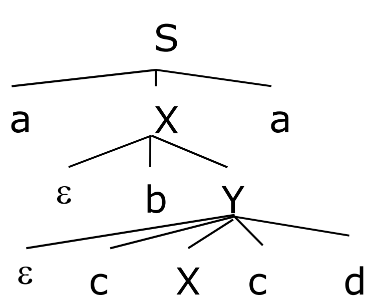
$S \rightarrow aXa$

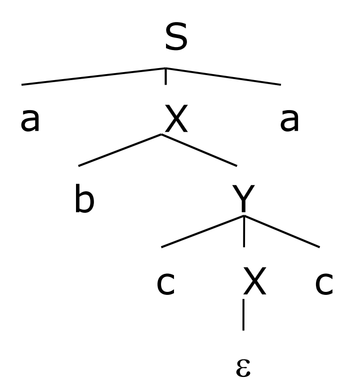
$X \rightarrow \varepsilon \mid bY$

$Y \rightarrow \varepsilon \mid cXc \mid d$









Question?

Choose the grammar that correctly eliminates left recursion from the given grammar: $E \rightarrow E + T \mid T$

$T \rightarrow id \mid (E)$

- $E \rightarrow E + id \mid E + (E) \mid id \mid (E)$

- $E \rightarrow TE'$
  $E' \rightarrow + TE' \mid \varepsilon$
  $T \rightarrow id \mid (E)$

- $E \rightarrow E' + T \mid T$
  $E' \rightarrow id \mid (E)$
  $T \rightarrow id \mid (E)$

- $E \rightarrow id + E \mid E + T \mid T$
  $T \rightarrow id \mid (E)$

Question?

Consider the following grammar. Adding which one of the following rules will cause the grammar to be left-recursive? [Choose all that apply]

$S \to A$

$A \to B \mid C$

$B \to (C)$

$C \to B + C \mid D$

$D \to 1 \mid 0$

- $D \to A$
- $A \to D$
- $B \to C$
- $D \to B$
- $C \to 1\ C$

Question?

Which of the following grammars are ambiguous?

- ☐ $S \rightarrow SS \mid a \mid b$
- ☐ $E \rightarrow E + E \mid id$
- ☐ $S \rightarrow Sa \mid Sb$
- ☐ $E \rightarrow E' \mid E' + E$

  $E' \rightarrow -E' \mid id \mid (E)$

Question?

Choose the unambiguous version of the given ambiguous grammar: $S \rightarrow SS \mid a \mid b \mid \varepsilon$

- $S \rightarrow Sa \mid Sb \mid \varepsilon$
- $S \rightarrow SS'$
  $S' \rightarrow a \mid b$
- $S \rightarrow S \mid S'$
  $S' \rightarrow a \mid b$
- $S \rightarrow Sa \mid Sb$

Question?

Consider the following grammar. How many unique parse trees are there for the string 5 * 3 + (2 * 7) + 4?

- 2
- 1
- 7
- 8
- 5
- 4

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

Question?

Which of the following statements are true about the given grammar?

$S \rightarrow a\ T\ U\ b \mid \varepsilon$
$T \rightarrow c\ U\ c \mid b\ U\ b \mid a\ U\ a$
$U \rightarrow S\ b \mid c\ c$

Choose all that are correct.

- The follow set of S is { \$, b }
- The first set of U is { a, b, c }
- The first set of S is {ε, a, b }
- The follow set of T is { a, b, c }

Question?

Consider the following grammar:

$$S \rightarrow A\ (\ S\ )\ B \mid \varepsilon$$
$$A \rightarrow S \mid S\ B\ x \mid \varepsilon$$
$$B \rightarrow S\ B \mid y$$

What are the first and follow sets of S

- First: { x, y, (, ε } Follow: { y, x, (,) }
- First: { x, ε } Follow: { $, y, x, (,) }
- First: { y, (, ε } Follow: { $, y, (,)}
- First: { x, y, (, ε } Follow: { $, y, x, (,) }
- First: { x, y, (} Follow: { $, y, x, (,) }
- First: { x, (} Follow: { $, y, x }

Question?

Choose the derivation that is a valid recursive descent parse for the string id + id in the given grammar. Moves that are followed by backtracking are given in red.

$$E \rightarrow E' \mid E' + E$$

$$E' \rightarrow -E' \mid id \mid (E)$$

- E
E'
E' + E
id + E
id + E'
id + id

- E
E'
-E'
id
(E)
E' + E
-E' + E
id + E
id + E'
id + -E'
id + id

- E
E' + E
id + E
id + E'
id + id

- E
E'
id
E' + E
id + E
id + E'
id + id

Question?

Choose the grammar that correctly eliminates left recursion from the given grammar: $E \to E + T \mid T$

$T \to id \mid (E)$

- $E \to E + id \mid E + (E) \mid id \mid (E)$

- $E \to TE'$
  $E' \to + TE' \mid \varepsilon$
  $T \to id \mid (E)$

- $E \to E' + T \mid T$
  $E' \to id \mid (E)$
  $T \to id \mid (E)$

- $E \to id + E \mid E + T \mid T$
  $T \to id \mid (E)$

Question?

Choose the alternative that correctly left factors "if" statements in the given grammar

```
EXPR → if BOOL then { EXPR }
     | if BOOL then { EXPR } else { EXPR }
     | ...
BOOL → true | false
```

- ```
  EXPR → if true then { EXPR }
       | if false then { EXPR }
       | if true then { EXPR } else { EXPR }
       | if false then { EXPR } else { EXPR }
       | ...
  ```
- ```
  EXPR → EXPR' | EXPR' else { EXPR }
  EXPR' → if BOOL then { EXPR }
        | ...
  BOOL → true | false
  ```
- ```
  EXPR → if BOOL EXPR'
       | ...
  EXPR' → then { EXPR }
        | then { EXPR } else { EXPR }
  BOOL → true | false
  ```
- ```
  EXPR → if BOOL then { EXPR } EXPR'
       | ...
  EXPR' → else { EXPR } | ε
  BOOL → true | false
  ```