Aalto University

Top-Down Parsing

Lecture 4

Announcements

- PA 1
  - Due today (17 June) at 11:59pm
- WA 2
  - Released today
- PA 2
  - Released today

Problems with Top Down Parsing

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

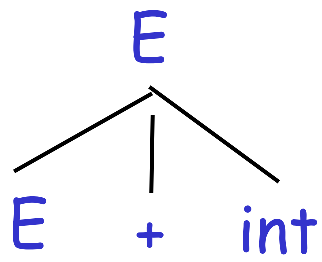
- In there is a production of form $A \to A\alpha$, we say the grammar has left recursion

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

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

Problems with Top Down Parsing (Example)

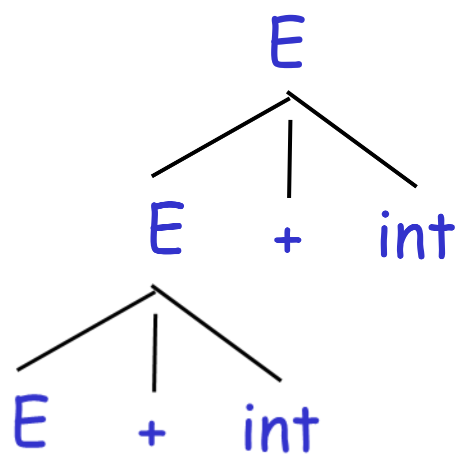
$E \rightarrow E + \text{int} \mid \text{int}$



int + int

Problems with Top Down Parsing (Example)

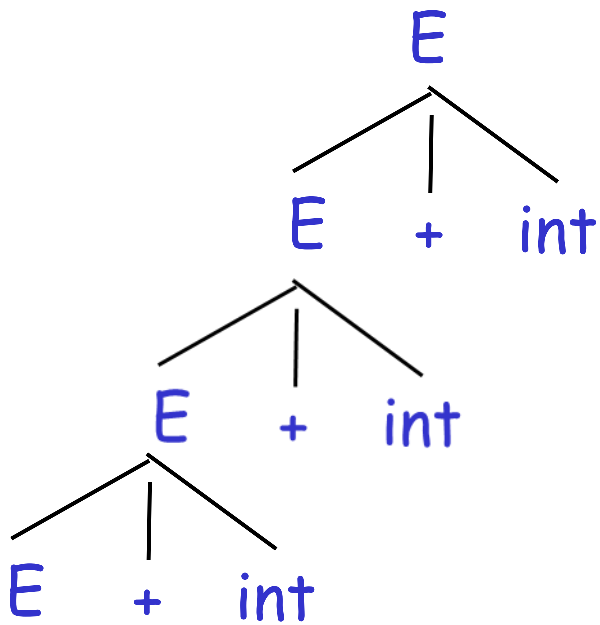
E → E + int | int



int + int

Problems with Top Down Parsing (Example)

E → E + int | 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

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

- All strings derived from S start with one of $\beta_1,\ldots,\beta_m$ and continue with several instances of $\alpha_1,\ldots,\alpha_n$
- Rewrite as

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

General Left Recursion

- The grammar

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

  is also left-recursive because

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

- This left-recursion can also be eliminated

- See Dragon Book for general algorithm
  - Section 4.3.3

Predictive Parsing and Left Factoring

- Consider the grammar

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

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

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

- We need to <u>left-factor</u> the grammar

Left-Factoring Example

- Recall the grammar

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

- Factor out common prefixes of productions

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

Predictive Parsers

- 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"
  - In practice, LL(1) is used

LL(1) Parsing Table Example

- Left-factored grammar

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

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

- The LL(1) parsing table:

next input token

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

leftmost non-terminal $\qquad$ *rhs of production to use*

LL(1) Parsing Table Example (Cont.)

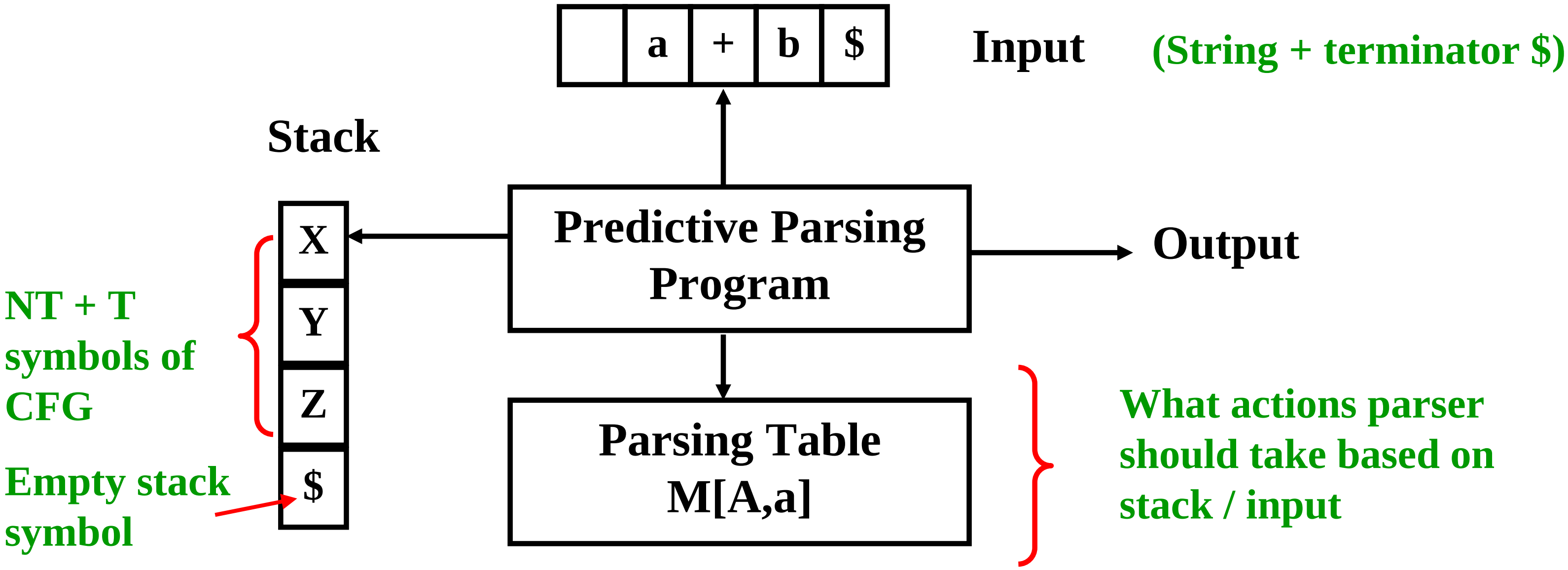
- Consider the [E, int] entry
  - "When current non-terminal is E and next input is int, use production $E \to T X$"
  - This can generate an int in the first position
- Consider the [Y,+] entry
  - "When current non-terminal is Y and current token is +, get rid of Y"
  - Y can be followed by + only if $Y \to \varepsilon$

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"

LL(1) Parsing Algorithm



General parser behavior: X : top of stack a : current token

1. When X=a = $ halt, accept, success

2. When X=a ≠ $, POP X off stack, advance input, go to 1.

3. When X is a non-terminal, examine M[X, a], if it is an error, call recovery routine if M[X, a] = {UVW}, POP X, PUSH U,V,W, and DO NOT advance input

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: The Intuition

- Consider non-terminal A, production $A \to \alpha$, & token t
- $T[A,t] = \alpha$ in two cases:

- If $\alpha \to^* t\ \beta$
  - α can derive a t in the first position
  - We say that $t \in First(\alpha)$

- If $A \to \alpha$ and $\alpha \to^* \varepsilon$ and $S \to^* \beta\ A\ t\ \delta$
  - Useful if stack has A, input is t, and A cannot derive t
  - In this case only option is to get rid of A (by deriving ε)
    - Can work only if t can follow A in at least one derivation
  - We say $t \in Follow(A)$

Constructing LL(1) Parsing Tables

- Construct a parsing table T for CFG G

- For each production $A \to \alpha$ in G do:
  - For each terminal $t \in First(\alpha)$ do
    - $T[A, t] = \alpha$
  - If $\varepsilon \in First(\alpha)$, for each $t \in Follow(A)$ do
    - $T[A, t] = \alpha$
  - If $\varepsilon \in First(\alpha)$ and $\$ \in Follow(A)$ do
    - $T[A, \$] = \alpha$

Example 1

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

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

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

Example 2

S→ Sa | b

First(S)={ b }

Follow(S)={ $,a }

	a	b	$
S		b, Sa	

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 CFGs are not LL(1)

Notes on LL(1) Grammars

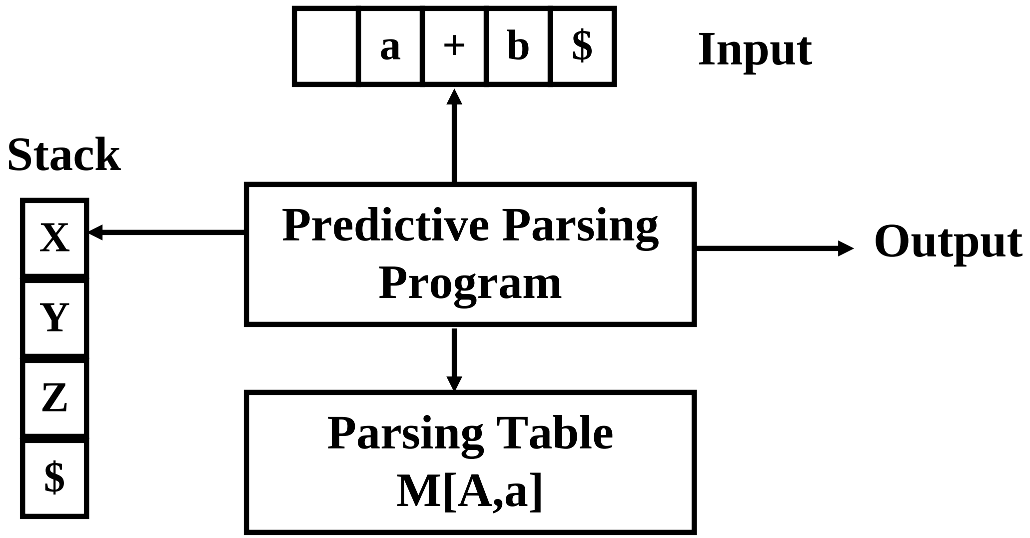
Grammar is LL(1) $\Leftrightarrow$ when for all $A \rightarrow \alpha \mid \beta$

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

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

It may not be possible for a grammar to be manipulated into an LL(1) grammar

Implementing Panic Mode in LL(1)



Error situations include:

1. If X is a terminal and it doesn't match current token.
2. If M[X, Input] is empty – No allowable actions

Panic-Mode Recovery

- Assume in a syntax error, non-terminal *A* is on the top of the stack.

- The choice for a synchronizing set is important.
  - define the synchronizing set of *A* to be Follow(*A*). Then skip input until a token in Follow(*A*) appears and then pop *A* from the stack. Resume parsing...

  - add symbols of FIRST(*A*) to the synchronizing set. In this case, we skip input and once we find a token in FIRST(*A*), we resume parsing from *A*.

Panic-Mode Recovery (Cont.)

Modify the empty cells of the Parsing Table.

1. if M[A, a] = {empty} and a belongs to Follow(A) then we set M[A, a] = "synch"

Error-recovery Strategy :

If A=top-of-the-stack and a=current-token,

1. If A is NT and M[A, a] = {empty} then skip a from the input.
2. If A is NT and M[A, a] = {synch} then pop A.
3. If A is a terminal and A!=a then pop A (This is essentially inserting A before a).

Parse Table / Example

	id	+	*	(	)	$
E	T E′			T E′	synch	synch
E′		+ T E′			∈	∈
T	F T′	synch		F T′	synch	synch
T′		∈	* F T′		∈	∈
F	id	synch	synch	(E)	synch	synch

Pop top of stack NT for "synch" cells

Skip current-token for empty cells

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Parsing Example

	id	+	*	(	)	$
E	T E'			T E'	synch	synch
E'		+ T E'			∈	∈
T	F T'	synch		F T'	synch	synch
T'		∈	* F T'		∈	∈
F	id	synch	synch	(E)	synch	synch

STACK	INPUT	Remark
E $	+ id * + id $	error, skip +
E $	id * + id $	
T E' $	id * + id $	
F T' E' $	id * + id $	
id T' E' $	id * + id $	
T' E' $	* + id $	
* F T' E' $	* + id $	
F T' E' $	+ id $	

Possible Error Msg:
"Misplaced +
I am skipping it"

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Parsing Example (Cont.)

	id	+	*	(	)	$
E	T E'			T E'	synch	synch
E'		+ T E'			∈	∈
T	F T'	synch		F T'	synch	synch
T'		∈	* F T'		∈	∈
F	id	synch	synch	(E)	synch	synch

STACK	INPUT	Remark
F T' E' $	+ id $	error, M[F,+] = synch, F is popped
T' E' $	+ id $	
E' $	+ id $	
+ T E' $	+ id $	
T E' $	id $	
F T' E' $	id $	
id T' E' $	id $	
T' E' $	$	
E' $	$	
$	$	

Possible Error Msg: "Missing Term"

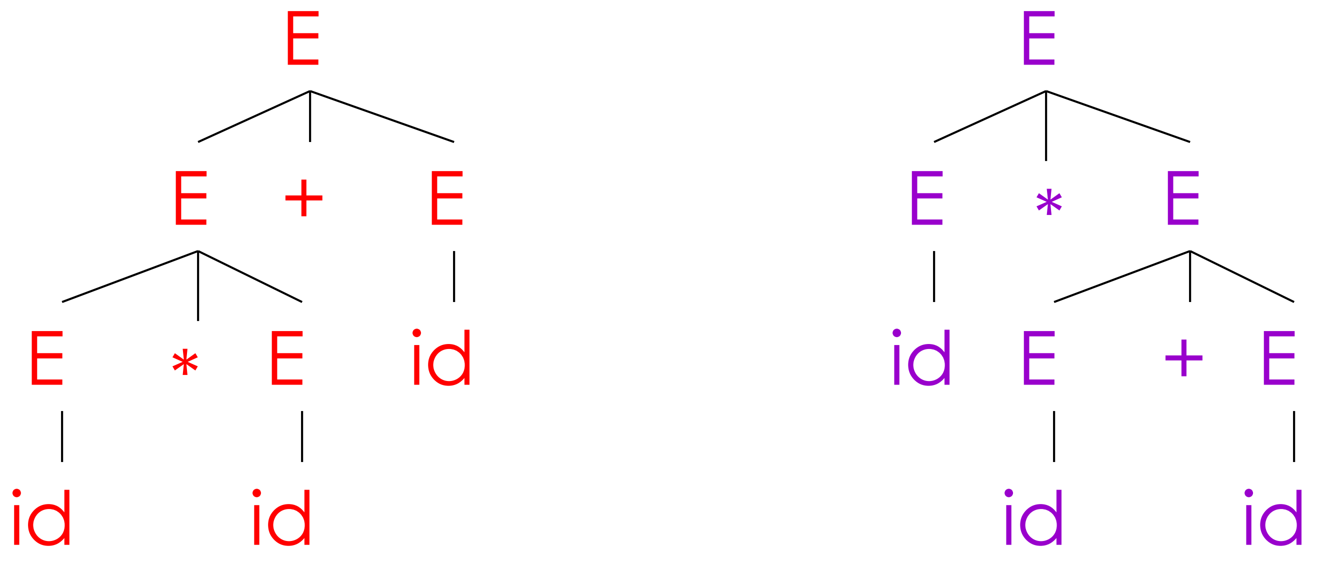
$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

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

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

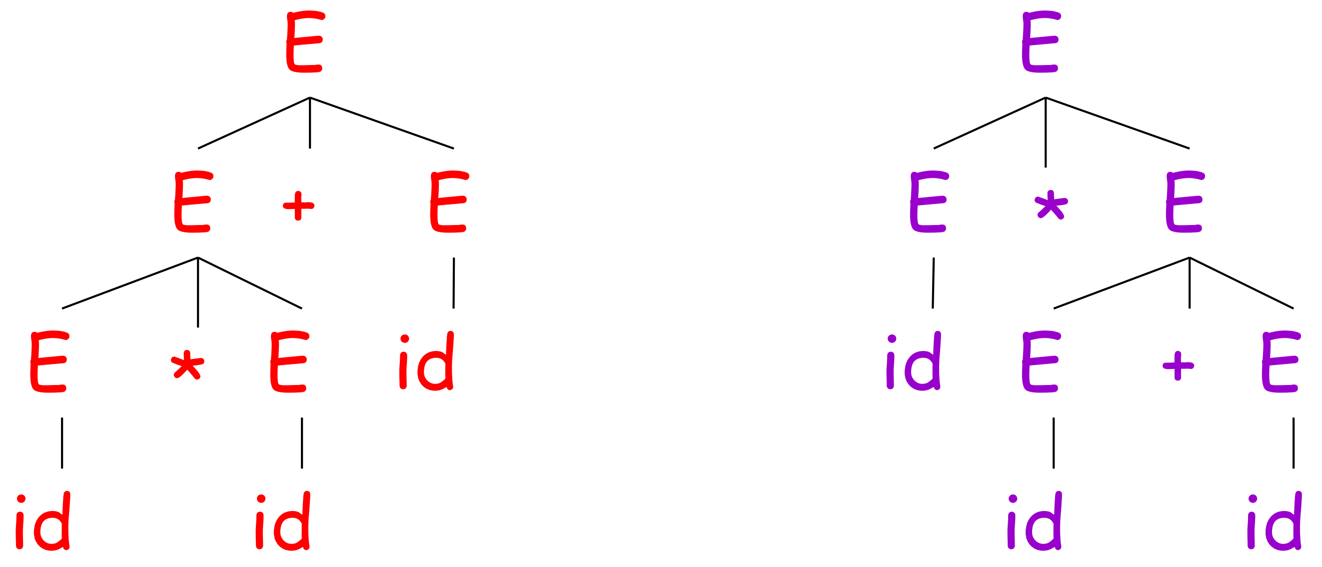
- Enforces precedence of * over +

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:

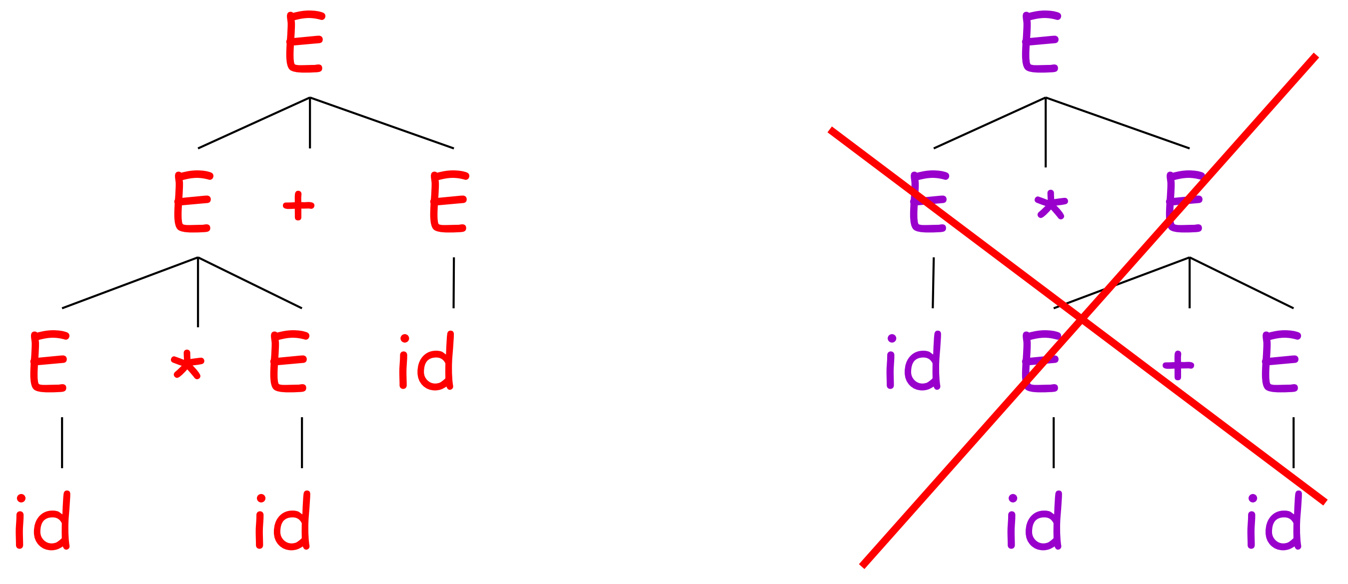


Ambiguity in Arithmetic Expressions

- 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

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

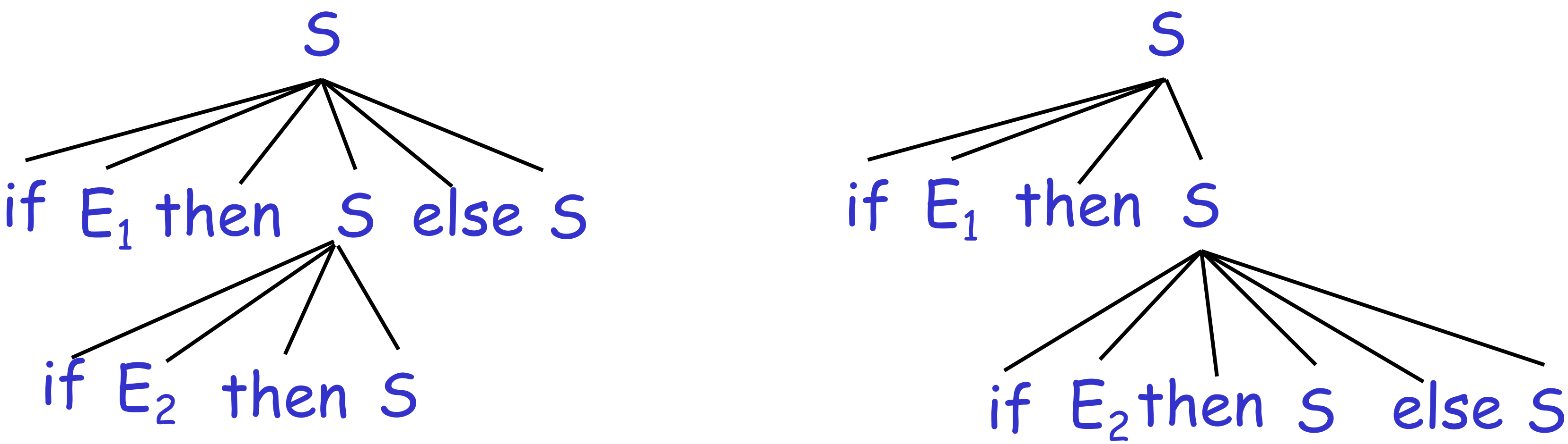
- This grammar is also ambiguous

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

$S \rightarrow$ MIF /* all then are matched */
  | UIF /* some then is unmatched */

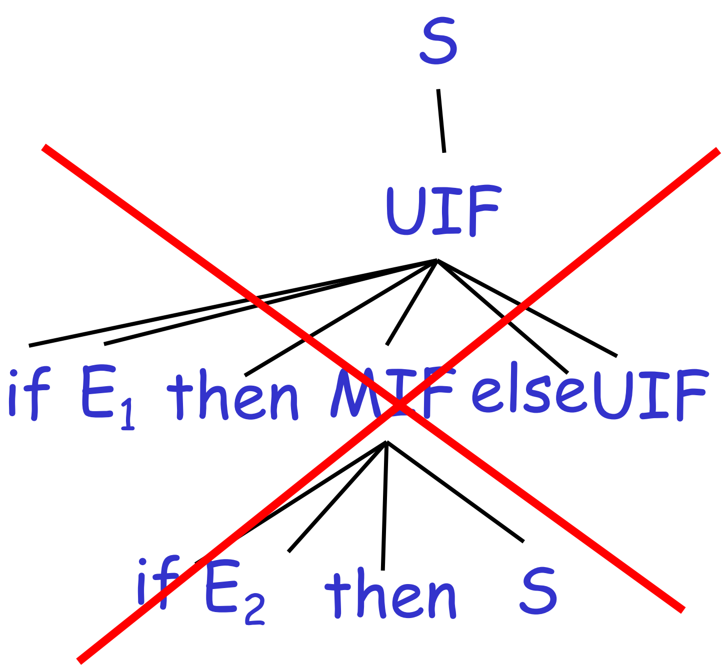
MIF $\rightarrow$ if E then MIF else MIF
  | OTHER

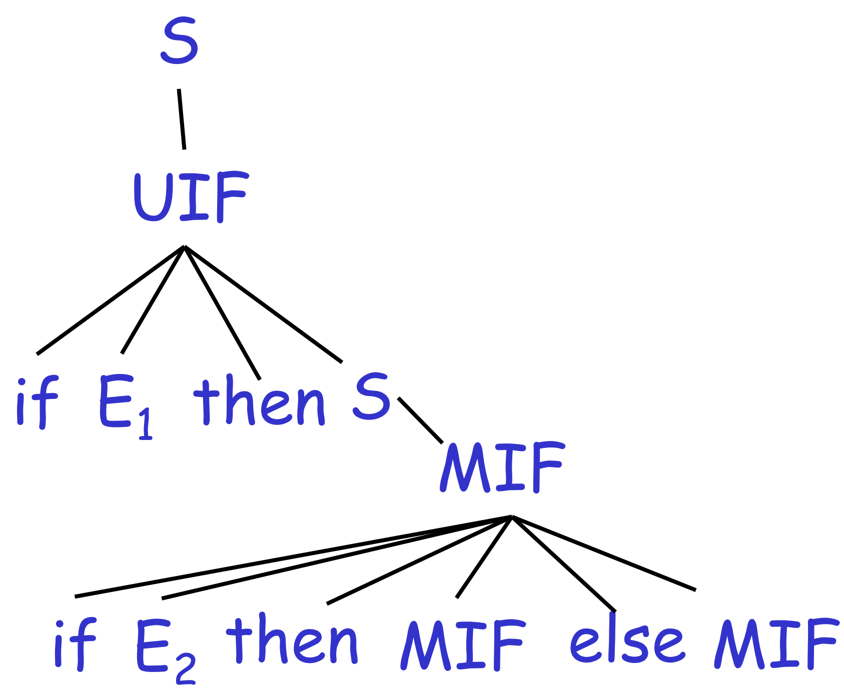
UIF $\rightarrow$ if E then S
  | if E then MIF else UIF

- 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

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
  ```

Question?

Choose the next parse state given the grammar, parse table, and current state below. The initial string is:

if true then { true } else { if false then { false } } $

	if	then	else	{	}	true	false	$
E	if B then { E } E'				ε	B	B	ε
E'			else { E }		ε			ε
B						true	false	

E → if B then { E } E'
  | B | ε
E' → else { E } | ε
B → true | false

	Stack	Input
Current	E' $	else { if false then { false } } $
○	$	$
○	else {E} $	else { if false then { false } } $
○	E} $	if false then { false } } $
○	else {if B then {E} E' } $	else { if false then { false } } $

Question?

For the given grammar, find the First and Follow of Non-terminals and the Parse table

$S \rightarrow$ i E t S S′ | a	First(S) =	Follow(S) =
$S' \rightarrow$ e S | $\in$	First(S′) =	Follow(S′) =
$E \rightarrow$ b	First(E) =	Follow(E) =

	a	b	e	i	t	$
S						
S′						
E						

Question?

For the given grammar, find the First and Follow of Non-terminals and the Parse table

$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \epsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \epsilon$
$F \rightarrow (\ E\ ) \mid id$

First(E,T,F) =

First(E') =

First(T') =

Follow(E) =

Follow(E') =

Follow(T) =

Follow(T') =

Follow(F) =

	id	+	*	(	)	$
E						
E'						
T						
T'						
F						

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:

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

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 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?

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

$S \to A$

$A \to B | C$

$B \to (C)$

$C \to B + C | D$

$D \to 1 | 0$

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

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

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