Compilers (CS31003)

Lecture 07-08

Pralay Mitra

Curse or Boon 1: Left-Recursion

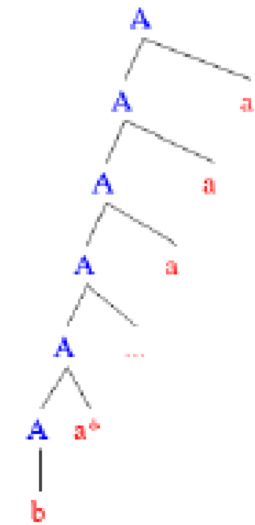
A grammar is left-recursive iff there exists a non-terminal A that can derive to a sentential form with itself as the leftmost symbol. Symbolically,

$$A \quad \Rightarrow^{+} \quad A\alpha$$

We cannot have a recursive descent or predictive parser (with left-recursion in the grammar) because we do not know how long should we recur without consuming an input

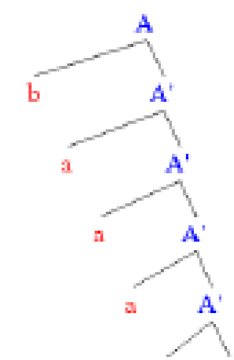
Curse or Boon 1: Left-Recursion

Note that, $\begin{array}{lll} A & \rightarrow & A\alpha \\ A & \rightarrow & \beta \end{array}$ leads to:



$$\begin{array}{llllllll} A\,\$ & \Rightarrow & A\alpha\,\$ & \Rightarrow & A\alpha\alpha\,\$ & \Rightarrow & A\alpha\alpha\alpha\,\$ & \cdots \\ & \Rightarrow & A\alpha^*\,\$ & \Rightarrow & \beta\alpha^*\,\$ & & & \end{array}$$

Removing left-recursion $\begin{array}{lllll} A & \rightarrow & \beta A' & & \\ A' & \rightarrow & \alpha A' & \mid & \epsilon \end{array}$ leads to:



$$\begin{array}{llllllll} A\,\$ & \Rightarrow & \beta A'\,\$ & \Rightarrow & \beta\alpha A'\,\$ & \Rightarrow & \beta\alpha\alpha A'\,\$ & \cdots \\ & \Rightarrow & \beta\alpha^* A'\,\$ & \Rightarrow & \beta\alpha^*\,\$ & & & \end{array}$$

Curse or Boon 2: Left-Recursion

$$
\begin{array}{llll}
1: & E & \rightarrow & E + E \\
2: & E & \rightarrow & E * E \\
3: & E & \rightarrow & (E) \\
4: & E & \rightarrow & \mathbf{id}
\end{array}
$$

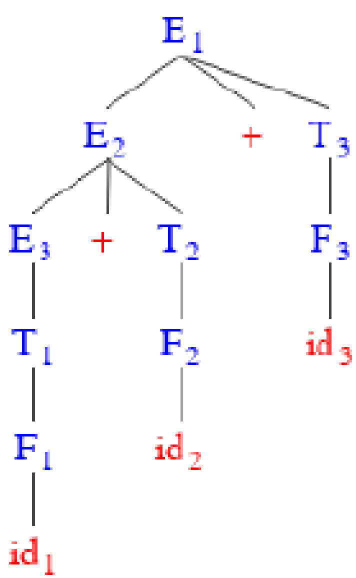
- Ambiguity simplifies. But, ...
  - Associativity is lost
  - Precedence is lost
- Can *Operator Precedence (infix* $\rightarrow$ *postfix)* give us a clue?

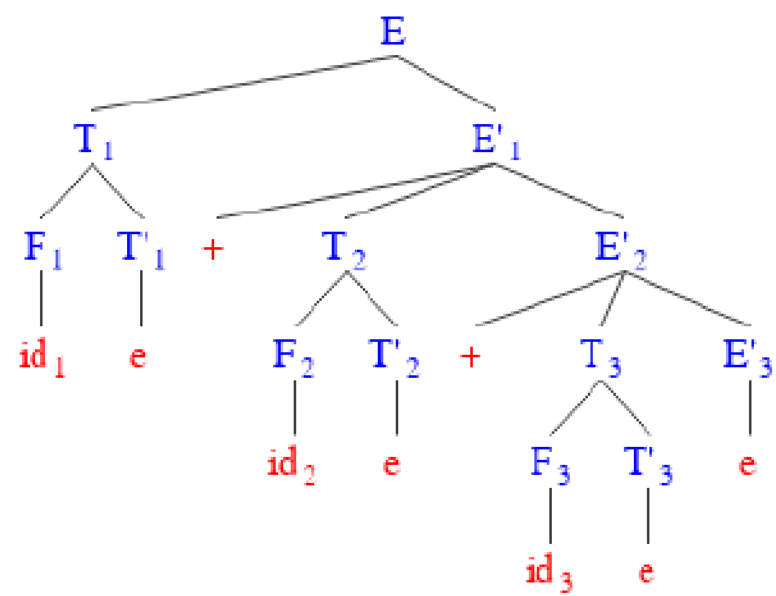
Left-Recursion: Example

Grammar G_1 before Left-Recursion Removal			
1:	E	$\rightarrow$	$E + T$
2:	E	$\rightarrow$	T
3:	T	$\rightarrow$	$T * F$
4:	T	$\rightarrow$	F
5:	F	$\rightarrow$	(E)
6:	F	$\rightarrow$	id

Grammar G_2 after Left-Recursion Removal					
1:	E	$\rightarrow$	$T\ E'$		
2:	E'	$\rightarrow$	$+\ T\ E'$	|	ϵ
3:	T	$\rightarrow$	$F\ T'$		
4:	T'	$\rightarrow$	$*\ F\ T'$	|	ϵ
5:	F	$\rightarrow$	(E)		
6:	F	$\rightarrow$	id		

- These are syntactically equivalent. But what happens semantically?
- Can left recursion be effectively removed?
- What happens to Associativity?





Top-Down parsing

- Action A: Selection of an alternative for the actual leftmost nonterminal and attachment of the right side of the production to the actual tree fragment.

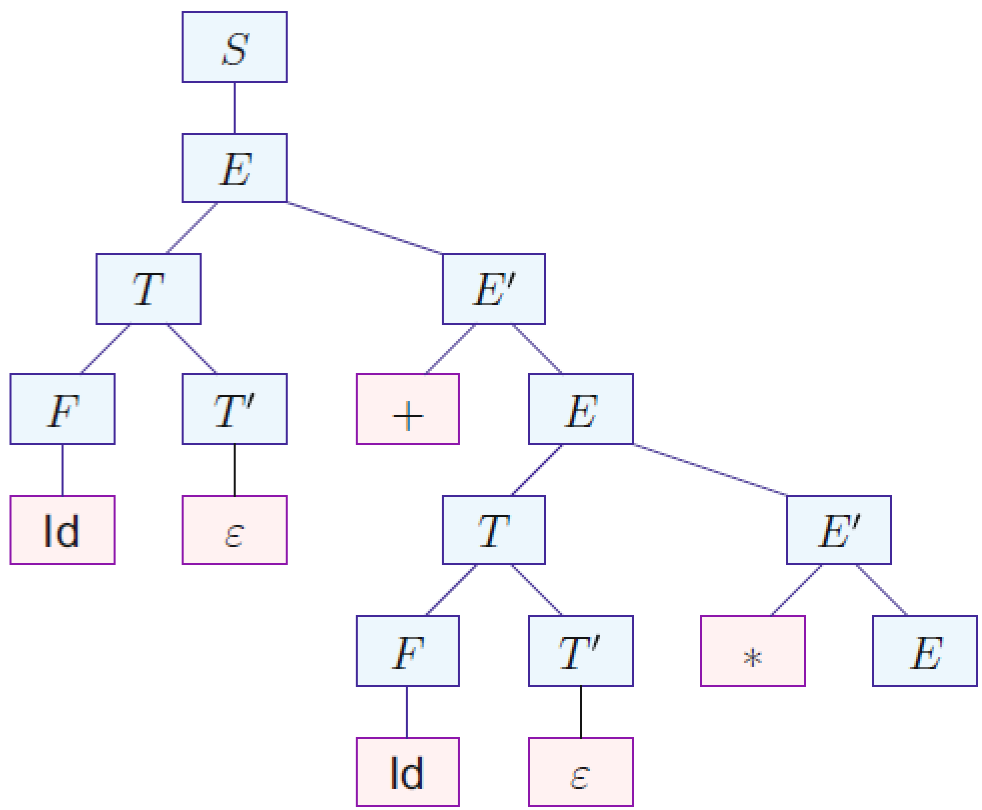
- Action B: Comparison of terminal symbols to the left of the leftmost nonterminal with the remaining input.

$$
\begin{array}{llll}
S \rightarrow E & E' \rightarrow + E \mid \varepsilon & T' \rightarrow * T \mid \varepsilon & \text{Id+Id*Id} \\
E \rightarrow T\, E' & T \rightarrow F\, T' & F \rightarrow (E) \mid \text{Id} &
\end{array}
$$

Top-Down parsing

$S \rightarrow E$ $\quad E' \rightarrow +E \mid \varepsilon$ $\quad T' \rightarrow *T \mid \varepsilon$ $\quad$ Id+Id*Id

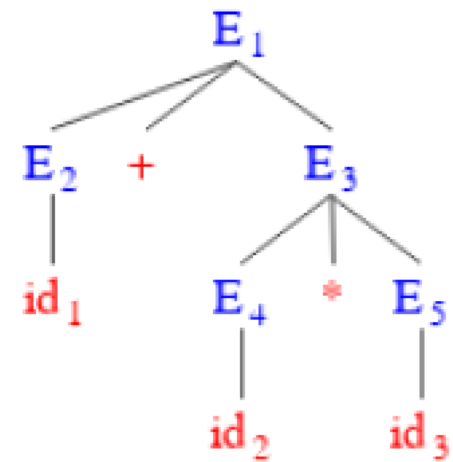
$E \rightarrow T\,E'$ $\quad T \rightarrow F\,T'$ $\quad F \rightarrow (E) \mid \text{Id}$



Ambiguous Derivation of id + id * id

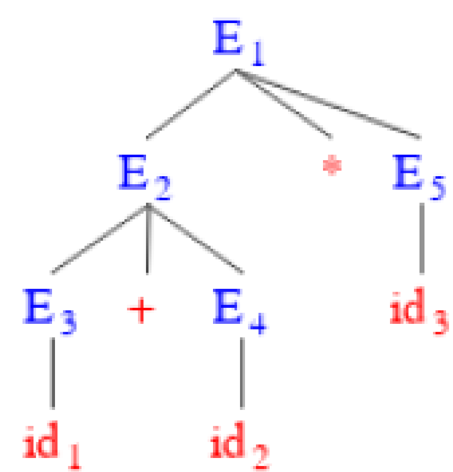
Correct derivation: * has precedence over +

$$
\begin{aligned}
E\ \$ &\Rightarrow \underline{E + E}\ \$ \\
&\Rightarrow E + \underline{E * E}\ \$ \\
&\Rightarrow E + E * \underline{\mathbf{id}}\ \$ \\
&\Rightarrow E + \underline{\mathbf{id}} * \mathbf{id}\ \$ \\
&\Rightarrow \underline{\mathbf{id}} + \mathbf{id} * \mathbf{id}\ \$
\end{aligned}
$$



Wrong derivation: + has precedence over *

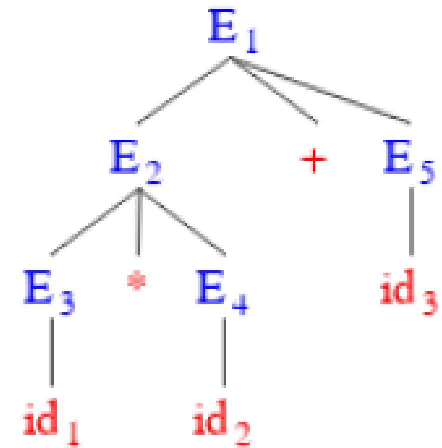
$$
\begin{aligned}
E\ \$ &\Rightarrow \underline{E * E}\ \$ \\
&\Rightarrow E * \underline{\mathbf{id}}\ \$ \\
&\Rightarrow \underline{E + E} * \mathbf{id}\ \$ \\
&\Rightarrow E + \underline{\mathbf{id}} * \mathbf{id}\ \$ \\
&\Rightarrow \underline{\mathbf{id}} + \mathbf{id} * \mathbf{id}\ \$
\end{aligned}
$$



Ambiguous Derivation of id * id + id

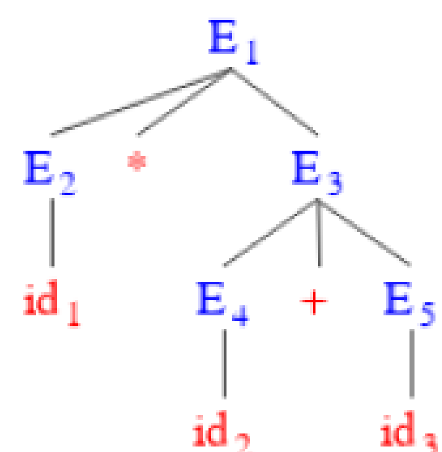
Correct derivation: * has precedence over +

$$
\begin{aligned}
E\,\$ &\Rightarrow \underline{E + E}\ \$ \\
&\Rightarrow E + \underline{\textbf{id}}\ \$ \\
&\Rightarrow \underline{E * E} + \textbf{id}\ \$ \\
&\Rightarrow E * \underline{\textbf{id}} + \textbf{id}\ \$ \\
&\Rightarrow \underline{\textbf{id}} * \textbf{id} + \textbf{id}\ \$
\end{aligned}
$$



Wrong derivation: + has precedence over *

$$
\begin{aligned}
E\,\$ &\Rightarrow \underline{E * E}\ \$ \\
&\Rightarrow E * \underline{E + E}\ \$ \\
&\Rightarrow E * E + \underline{\textbf{id}}\ \$ \\
&\Rightarrow E * \underline{\textbf{id}} + \textbf{id}\ \$ \\
&\Rightarrow \underline{\textbf{id}} * \textbf{id} + \textbf{id}\ \$
\end{aligned}
$$



Remove: Ambiguity and Left-Recursion

$$
\begin{array}{llll}
1: & E & \rightarrow & E + E \\
2: & E & \rightarrow & E * E \\
3: & E & \rightarrow & (E) \\
4: & E & \rightarrow & \text{id}
\end{array}
$$

Removing ambiguity:

Remove: Ambiguity and Left-Recursion

1:	E	$\rightarrow$	$E + E$
2:	E	$\rightarrow$	$E * E$
3:	E	$\rightarrow$	(E)
4:	E	$\rightarrow$	id

Removing ambiguity:

1:	E	$\rightarrow$	$E + T$
2:	E	$\rightarrow$	T
3:	T	$\rightarrow$	$T * F$
4:	T	$\rightarrow$	F
5:	F	$\rightarrow$	(E)
6:	F	$\rightarrow$	id

Removing left-recursion:

Remove: Ambiguity and Left-Recursion

$$
\begin{array}{llll}
1: & E & \rightarrow & E + E \\
2: & E & \rightarrow & E * E \\
3: & E & \rightarrow & (E) \\
4: & E & \rightarrow & \textbf{id}
\end{array}
$$

Removing ambiguity:

$$
\begin{array}{llll}
1: & E & \rightarrow & E + T \\
2: & E & \rightarrow & T \\
3: & T & \rightarrow & T * F \\
4: & T & \rightarrow & F \\
5: & F & \rightarrow & (E) \\
6: & F & \rightarrow & \textbf{id}
\end{array}
$$

Removing left-recursion:

$$
\begin{array}{rlll}
1: & E & \rightarrow & T\ E' \\
2|3: & E' & \rightarrow & +\ T\ E' \mid \epsilon \\
4: & T & \rightarrow & F\ T' \\
5|6: & T' & \rightarrow & *\ F\ T' \mid \epsilon \\
7: & F & \rightarrow & (E) \\
8: & F & \rightarrow & \textbf{id}
\end{array}
$$

A grammar

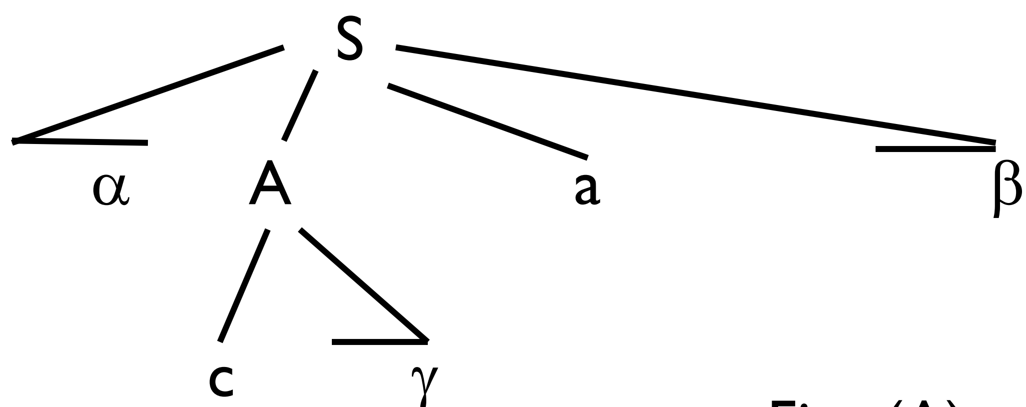
- G_0=({E,T,F}, {+,*,(,),Id}, P_0, E)
- P_0

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

First and Follow

- First(α) is the set of terminals that begin strings derived from α, where α is any string of grammar symbols. If $\alpha \Rightarrow^* \varepsilon$ then ε is also in First(α).

- Follow(*A*), for a non-terminal *A*, is the set of terminals *a* that can appear immediately to the right of *A* in some sentential; that is, the set of terminals *a* such that there exists a derivation of the form $S \Rightarrow^* \alpha A a \beta$ for some α and β.



First(A)=c
Follow(A)=a

First and Follow

First(α):

1. First(X) = {X} if X is a terminal.
2. Add ε to First(X) if there exists X $\rightarrow$ ε.
3. If there is a production X $\rightarrow$ $Y_1Y_2Y_3..Y_k$, k≥1, then place *a* in First(X) if *a* is in First(Y_i) and $Y_1Y_2..Y_{i-1} \Rightarrow^* \varepsilon$.

Follow(A):

1. Follow(S)=$, where S is the start symbol and $ is the input right end marker.
2. If there is a production A $\rightarrow$ $\alpha B\beta$, then everything in First(β) except ε is in Follow(B).
3. If there is a production A $\rightarrow$ $\alpha B \mid \alpha B\beta$, where First($\beta$) contains ε then everything in Follow(A) is in Follow(B).

Exercise

Compute First and Follow for the following grammar:

- **E → T E'**
- **E' → + T E' | ε**
- **T → F T'**
- **T' → * F T' | ε**
- **F → (E) | *id***

Do it now

stmt → if *expr* then *stmt*

| if *expr* then *stmt* else *stmt*

| other

Ambiguous

Make it unambiguous.

Item Pushdown Automata (IPDA)

$$
\begin{array}{lll}
(E) & \Delta([X \rightarrow \beta.Y\gamma], \varepsilon) & = \{[X \rightarrow \beta.Y\gamma][Y \rightarrow .\alpha] \mid Y \rightarrow \alpha \in P\} \\
(S) & \Delta([X \rightarrow \beta.a\gamma], a) & = \{[X \rightarrow \beta a.\gamma]\} \\
(R) & \Delta([X \rightarrow \beta.Y\gamma][Y \rightarrow \alpha.], \varepsilon) & = \{[X \rightarrow \beta Y.\gamma]\}.
\end{array}
$$

E/S/R ← Expanding/Shifting/Reducing Transition

Accepting the word *Id+Id*Id*

- $G_0' = (\{S,E,T,F\}, \{+,*,(,),Id\}, P_0', E)$
- P_0' $\quad S \rightarrow E$

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

Pushdown	Remaining input
$[S \rightarrow .E]$	$\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$
$[S \rightarrow .E][E \rightarrow .E + T]$	$\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$
$[S \rightarrow .E][E \rightarrow .E + T][E \rightarrow .T]$	$\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$
$[S \rightarrow .E][E \rightarrow .E + T][E \rightarrow .T][T \rightarrow .F]$	$\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$
$[S \rightarrow .E][E \rightarrow .E + T][E \rightarrow .T][T \rightarrow .F][F \rightarrow .\mathsf{Id}]$	$\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$
$[S \rightarrow .E][E \rightarrow .E + T][E \rightarrow .T][T \rightarrow .F][F \rightarrow \mathsf{Id}.]$	$+\mathsf{Id} * \mathsf{Id}$

Accepting the word *Id+Id*Id*

$[S \to .E][E \to .E+T][E \to .T][T \to F.]$	$+\text{Id} * \text{Id}$
$[S \to .E][E \to .E+T][E \to T.]$	$+\text{Id} * \text{Id}$
$[S \to .E][E \to E.+T]$	$+\text{Id} * \text{Id}$
$[S \to .E][E \to E+.T]$	$\text{Id} * \text{Id}$
$[S \to .E][E \to E+.T][T \to .T*F]$	$\text{Id} * \text{Id}$
$[S \to .E][E \to E+.T][T \to .T*F][T \to .F]$	$\text{Id} * \text{Id}$
$[S \to .E][E \to E+.T][T \to .T*F][T \to .F][F \to .\text{Id}]$	$\text{Id} * \text{Id}$
$[S \to .E][E \to E+.T][T \to .T*F][T \to .F][F \to \text{Id}.]$	$*\text{Id}$
$[S \to .E][E \to E+.T][T \to .T*F][T \to F.]$	$*\text{Id}$
$[S \to .E][E \to E+.T][T \to T.*F]$	$*\text{Id}$
$[S \to .E][E \to E+.T][T \to T*.F]$	Id
$[S \to .E][E \to E+.T][T \to T*.F][F \to .\text{Id}]$	Id
$[S \to .E][E \to E+.T][T \to T*.F][F \to \text{Id}.]$	
$[S \to .E][E \to E+.T][T \to T*F.]$	
$[S \to .E][E \to E+T.]$	
$[S \to E.]$	

Construction of Predictive Parsing Table

- Input: Grammar G
- Output: Parsing Table M

- Method:

For each production A $\rightarrow \alpha$ of the grammar, do the following

1. For each terminal a in First(α), add A$\rightarrow\alpha$ to M[A, a].
2. If ε is in First(α), then for each terminal b in Follow(A), add A$\rightarrow\alpha$ to M[A, b]. If ε is in First(α) and \$ is in Follow(A), add A$\rightarrow\alpha$ to M[A,\$] as well.
3. All the blank M[A, a] entries are marked as error.

LL(1)

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | *id*

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

LL(1)

	id	+	*
E	E → T E'		
E'		E' → + T E'	
T	T → F T'		
T'		T' → ε	T' → * F T'
F	F → Id		

	(	)	$
E	E → T E'		
E'		E' → ε	E' → ε
T	T → F T'		
T'		T' → ε	T' → ε
F	F → (E)		

Matched	Stack	Input	Action
	E$	Id+Id*Id$	
	TE'$	Id+Id*Id$	Output E → TE'
	FT'E'$	Id+Id*Id$	Output T → FT'
	Id T'E'$	Id+Id*Id$	Output F → Id
Id	T'E'$	+Id*Id$	Match Id
Id	E'$	+Id*Id$	Output T' → ε
Id	+TE'$	+Id*Id$	Output E' →+TE'
Id+	TE'$	Id*Id$	Match +
Id+	FT'E'$	Id*Id$	Output T → FT'
Id+	Id T'E'$	Id*Id$	Output F → Id
Id+Id	T'E'$	*Id$	Match Id
Id+Id	* FT'E'$	*Id$	Output T' → *FT'
Id+Id*	FT'E'$	Id$	Match *
Id+Id*	Id T'E'$	Id$	Output F → Id
Id+Id*Id	T'E'$	$	Match Id
Id+Id*Id	E'$	$	Output T' → ε
Id+Id*Id	$	$	Output E' → ε

Syntax error

- 1. Error is localized and reported.
- 2. Error is diagnosed.
- 3. Error is corrected.
- 4. Parser gets back to a state for further error detection.

A = B + (C + D * E ;

Should not go into endless loop while correcting errors.

Whenever the prefix ***u*** of a word has been analyzed without announcing an error, then there exists a word ***w*** such that ***uw*** is a word of the language.

Error recovery in predictive parsing

E → T E'
E' → + T E' | *ε*
T → F T'
T' → * F T' | *ε*
F → (E) | *id*

	id	+	*	(	)	$
E	E → T E'			E → T E'	*synch*	*synch*
E'		E' → + T E'			E' → ε	E' → ε
T	T → F T'	*synch*		T → F T'	*synch*	*synch*
T'		T' → ε	T' → * F T'		T' → ε	T' → ε
F	F → Id	*synch*	*synch*	F → (E)	*synch*	*synch*

LL(1) parser (Home work)

Construct the predictive parsing table

$0:$	S	$\rightarrow$	E
$1:$	E	$\rightarrow$	TE'
$2:$	E'	$\rightarrow$	ε
$3:$	E'	$\rightarrow$	$+E$
$4:$	T	$\rightarrow$	FT'
$5:$	T'	$\rightarrow$	ε
$6:$	T'	$\rightarrow$	$*T$
$7:$	F	$\rightarrow$	(E)
$8:$	F	$\rightarrow$	Id

G=({S,E,S'}, {*i*, *t*, *a*, *e*, *b*}, P, S)

P:

S → *i* E *t* S S' | a

S' → *e* S | ε

E → *b*

Parsers

- Top-down parsing:
  - The non-confirmed part of the prediction starts with a nonterminal.
  - Termination upon prediction and confirmation of whole input.
  - $\gamma_1\beta\gamma_2$ is produced/derived from $\gamma_1 A\gamma_2$ when $A \rightarrow \beta$ is a production rule.

- Bottom-up parsing:
  - The non-confirmed part of the prediction starts with a nonterminal.
  - It either reduce or shift to next input symbol.
  - $\gamma_1 A\gamma_2$ is reduced from $\gamma_1\beta\gamma_2$ when $A \rightarrow \beta$ is a production rule.

Bottom-up Parser

- Read the next input symbol (*shift*)
- Reduce the right side of a production $X \rightarrow \alpha$ at the top of the pushdown by the left side X of the production (*reduce*).

Bottom-up parsing

- Bottom-up parsing:
  - The non-confirmed part of the prediction starts with a nonterminal.
  - It either reduce or shift to next input symbol.
  - $\gamma_1 A \gamma_2$ is reduced from $\gamma_1 \beta \gamma_2$ when $A \rightarrow \beta$ is a production rule.

Item Pushdown Automata (IPDA)

$$
\begin{aligned}
&(E) \quad \Delta([X \to \beta.Y\gamma], \varepsilon) &&= \{[X \to \beta.Y\gamma][Y \to .\alpha] \mid Y \to \alpha \in P\} \\
&(S) \quad \Delta([X \to \beta.a\gamma], a) &&= \{[X \to \beta a.\gamma]\} \\
&(R) \quad \Delta([X \to \beta.Y\gamma][Y \to \alpha.], \varepsilon) &&= \{[X \to \beta Y.\gamma]\}.
\end{aligned}
$$

E/S/R ← Expanding/Shifting/Reducing Transition

Definitions

- **Item for** A→XYZ

  A → .XYZ | X.YZ | XY.Z | XYZ.

- **Closure**

  If I is a set of items for a grammar G, then CLOSURE(I) is the set of items constructed from I by:

  (i) Add every item in I to CLOSURE(I)

  (ii) $\forall\ A \rightarrow \alpha.B\beta \in$ CLOSURE(I) $\wedge\ B \rightarrow \gamma$

  Add $B \rightarrow .\gamma$ to CLOSURE(I) if it is not there.

- **Action**
- **Goto**

  GOTO(I,X) is defined to be the closure of the set of all items [$A \rightarrow \alpha X.\beta$] such that [$A \rightarrow \alpha.X\beta$] is in I.

Definitions

- **Item for** A→XYZ

  A → .XYZ | X.YZ | XY.Z | XYZ.

- **Kernel Items, Non-kernel Items**
- **Closure**

  If I is a set of items for a grammar G, then CLOSURE(I) is the set of items constructed from I by:

  (i) Add every item in I to CLOSURE(I)

  (ii) $\forall\ A \rightarrow \alpha.B\beta \in$ CLOSURE(I) $\wedge\ B \rightarrow \gamma$

  Add $B \rightarrow .\gamma$ to CLOSURE(I) if it is not there.

- **Action**
- **Goto**

  GOTO(I,X) is defined to be the closure of the set of all items [$A \rightarrow \alpha X.\beta$] such that [$A \rightarrow \alpha.X\beta$] is in I.

Definitions

Handle: A substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.

Reliable prefix: A reliable prefix is a prefix of a right sentential-form that does not properly extend beyond the handle.

Right Sentential Form	Handle	Reducing Production
Id * Id	Id	F → Id
F * Id	F	T → F
T * Id	Id	F → Id
T * F	T * F	T → T * F
T	T	E → T

Bottom-up parsing

S → E
E → E+T | T
T → T*F | F
F → (E) | Id

In an unambiguous grammar, the handle of a right sentential-form is the uniquely determined word that the *bottom-up* parser should replace by a nonterminal in the next reduction step to arrive at a rightmost derivation.

A reliable prefix is a prefix of a right sentential-form that does not properly extend beyond the handle.

Right sentential-form	Handle	Reliable prefixess	Reason
$E + F$	F	E, $E+$, $E+F$	$S \underset{rm}{\Longrightarrow} E \underset{rm}{\Longrightarrow} E + T \underset{rm}{\Longrightarrow} E + F$
$T * \mathrm{Id}$	Id	T, $T*$, $T * \mathrm{Id}$	$S \overset{3}{\underset{rm}{\Longrightarrow}} T * F \underset{rm}{\Longrightarrow} T * \mathrm{Id}$

Bottom-up parsing

S → E
E → E+T | T
T → T*F | F
F → (E) | Id

A reliable prefix is a prefix of a right sentential-form that does not properly extend beyond the handle.

Reliable prefix	Valid item	Reason
$E+$	$[E \to E + .T]$	$S \underset{rm}{\Longrightarrow} E \underset{rm}{\Longrightarrow} E+T$
	$[T \to .F]$	$S \overset{*}{\underset{rm}{\Longrightarrow}} E+T \underset{rm}{\Longrightarrow} E+F$
	$[F \to .\mathrm{Id}]$	$S \overset{*}{\underset{rm}{\Longrightarrow}} E+F \underset{rm}{\Longrightarrow} E+\mathrm{Id}$
$(E+($	$[F \to (.E)]$	$S \overset{*}{\underset{rm}{\Longrightarrow}} (E+F) \underset{rm}{\Longrightarrow} (E+(E))$
	$[T \to .F]$	$S \overset{*}{\underset{rm}{\Longrightarrow}} (E+(.T) \underset{rm}{\Longrightarrow} (E+(F))$
	$[F \to .\mathrm{Id}]$	$S \overset{*}{\underset{rm}{\Longrightarrow}} (E+(F) \underset{rm}{\Longrightarrow} (E+(\mathrm{Id}))$

Bottom-up parsing

S → E

E → E+T | T

T → T*F | F

F → (E) | Id

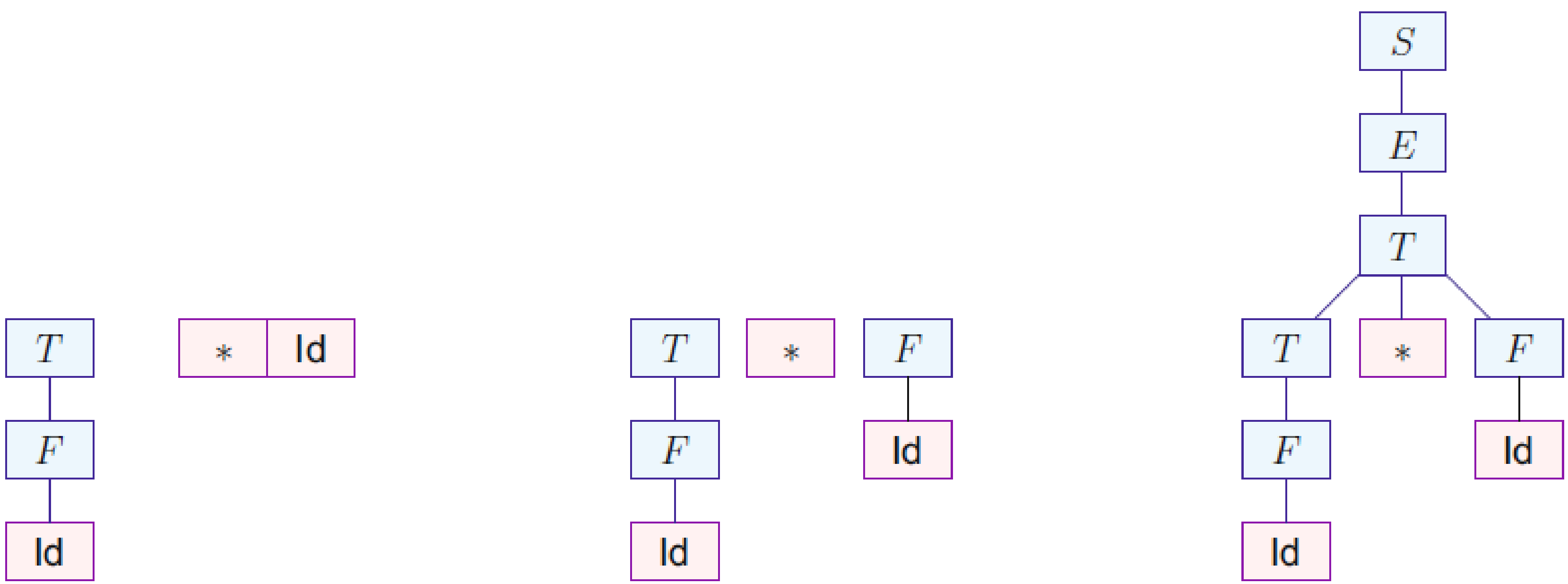
Id * Id

Pushdown	Input	Erroneous alternative actions
	Id $*$ Id	
Id	$*$ Id	
F	$*$ Id	Reading of $*$ misses a required reduction
T	$*$ Id	Reduction of T to E leads into a dead end
T $*$	Id	
T $*$ Id		
$T * F$		Reduction of F to T leads into a dead end
T		
E		
S		

Bottom-up parsing

S → E
E → E+T | T
T → T*F | F
F → (E) | Id

Id * Id



Shift-Reduce parsing

S → E
E → E+T | T
T → T*F | F
F → (E) | Id

Shift / Reduce / Accept / Error

Stack	Input	Action
$	Id*Id$	Shift
$Id	*Id$	Reduce F → Id
$F	*Id$	Reduce T → F
$T	*Id$	Shift
$T*	Id$	Shift
$T*Id	$	Reduce F →Id
$T*F	$	Reduce T → T*F
$T	$	Reduce E → T
$E	$	Accept

Shift/Reduce conflict
Reduce/Reduce conflict

LR parsing

si ← shift and stack state i,
rj ← reduce by the production j,
Acc ← accept
Blank ← error

(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → Id

State	Action						Goto		
	Id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

LR parsing

(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → Id

	Stack	Symbol	Input	Action
1	0		Id*Id+Id$	shift
2	0 5	Id	*Id+Id$	reduce F → Id
3	0 3	F	*Id+Id$	reduce T → F
4	0 2	T	*Id+Id$	shift
5	0 2 7	T*	Id+Id$	shift
6	0 2 7 5	T*Id	+Id$	reduce F → Id
7	0 2 7 10	T*F	+Id$	reduce T → T*F
8	0 2	T	+Id$	reduce E → T
9	0 1	E	+Id$	shift
10	0 1 6	E+	Id$	shift
11	0 1 6 5	E+Id	$	reduce F → Id
12	0 1 6 3	E+F	$	reduce T → F
13	0 1 6 9	E+T	$	reduce E → E+T
14	0 1	E	$	accept

LR(0) Automaton

I0: E'→.E
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.id

E' → E
E → E + T | T
T → T * F | F
F → (E) | id

LR(0) Automaton

I0: E'→.E
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.id

I1: E'→E.
E→E.+T

E' → E
E → E + T | T
T → T * F | F
F → (E) | id

LR(0) Automaton

I0: E'→.E
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.id

I1: E'→E.
E→E.+T

I2: E→T.
T→T.*F

E' → E
E → E + T | T
T → T * F | F
F → (E) | id

LR(0) Automaton

I0: E'→.E
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.id

I1: E'→E.
E→E.+T

I2: E→T.
T→T.*F

I3: T→F.

E' → E
E → E + T | T
T → T * F | F
F → (E) | id

LR(0) Automaton

I0: E’→.E
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.id

I1: E’→E.
E→E.+T

I2: E→T.
T→T.*F

I4: F→(.E)
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.id

I3: T→F.

E’ → E
E → E + T | T
T → T * F | F
F → (E) | id

LR(0) Automaton

I0: E'→.E
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.id

I1: E'→E.
E→E.+T

I2: E→T.
T→T.*F

I5: F→id.

I4: F→(.E)
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.id

I3: T→F.

E' → E
E → E + T | T
T → T * F | F
F → (E) | id

LR(0) Automaton

I0: E'→.E
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.id

I1: E'→E.
E→E.+T

I6: E→E+.T
T→.T*F
T→.F
F→.(E)
F→.id

I2: E→T.
T→T.*F

I7: T→T*.F
F→.(E)
F→.id

I5: F→id.

I4: F→(.E)
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.id

I3: T→F.

E' → E
E → E + T | T
T → T * F | F
F → (E) | id

LR(0) Automaton

I0: E'→.E
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.id

I1: E'→E.
E→E.+T

I2: E→T.
T→T.*F

I5: F→id.

I4: F→(.E)
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.id

I3: T→F.

I6: E→E+.T
T→.T*F
T→.F
F→.(E)
F→.id

I7: T→T*.F
F→.(E)
F→.id

I8: E→E.+T
F→(E.)

I9: E→E+T.
T→T.*F

I10: T→T*F.

I11: F→(E).

E' → E
E → E + T | T
T → T * F | F
F → (E) | id

LR(0) Automaton

Homework:
Add GOTO(*I*,+)

I0: E'→.E
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.id

I1: E'→E.
E→E.+T

I2: E→T.
T→T.*F

I3: T→F.

I4: F→(.E)
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.id

I5: F→id.

I6: E→E+.T
T→.T*F
T→.F
F→.(E)
F→.id

I7: T→T*.F
F→.(E)
F→.id

I8: E→E.+T
F→(E.)

I9: E→E+T.
T→T.*F

I10: T→T*F.

I11: F→(E).

E' → E
E → E + T | T
T → T * F | F
F → (E) | id

Canonical LR (1) parsing table

Input: An augmented grammar *G'*.

Output: Canonical LR parsing table with *Action* and *Goto* for *G'*

Method:

1. Construct $C' = \{I_0, I_1, ..\}$, the collection of sets of LR(1) items for *G'*.
2. State *i* of the parser is constructed from I_i. The parsing action for state *i* is:
   (a) If $[A \rightarrow \alpha.a\beta, b]$ is in I_i and $Goto(I_i,a)=I_j$, then $Action[i,a]$ is "*shift j*". Here *a* must be a terminal.
   (b) If $[A \rightarrow \alpha., a]$ is in I_i, $A \neq S'$, then $Action[i,a]$ is "*reduce* $A \rightarrow \alpha.$".
   (c) If $[S' \rightarrow S., \$]$ is in I_i, then $Action[i,\$]$ is "*accept*".

3. The *Goto* transition for state *i* are constructed for all non-terminals *A* using the rule: If $Goto(I_i,A)=I_j$, then $Goto[i,A]=j$.
4. All blank entries are error.
5. Initial state is $[S' \rightarrow .S,\$]$.

Canonical LR (1) parsing table

Input: An augmented grammar G'.

Output: Canonical LR parsing table with *Action* and *Goto* for G'

Method:

1. Construct $C'=\{I_0,I_1, ..\}$, the collection of sets of LR(1) items for G'.
2. State i of the parser is constructed from I_i. The parsing action for state i is:
   (a) If $[A \rightarrow \alpha.a\beta, b]$ is in I_i and $Goto(I_i,a)=I_j$, then $Action[i,a]$ is "*shift j*". Here a must be a terminal.
   (b) If $[A \rightarrow \alpha., a]$ is in I_i, $A \neq S'$, then $Action[i,a]$ is "*reduce* $A \rightarrow \alpha.$".
   (c) If $[S' \rightarrow S., \$]$ is in I_i, then $Action[i,\$]$ is "*accept*".
3. The *Goto* transition for state i are constructed for all non-terminals A using the rule: If $Goto(I_i,A)=I_j$, then $Goto[i,A]=j$.
4. All blank entries are error.
5. Initial state is $[S' \rightarrow .S,\$]$.

LR(0)????
LR(2)?????

An example

(0) S' → S
(1) S → CC
(2) C → cC
(3) C → d

I0: S'→.S,$
S→.CC,$
C→.cC,c/d
C→.d,c/d

I1: S'→S.,$

I2: S→C.C,$
C→.cC,c/d
C→.d,c/d

I3: C→c.C,c/d
C→.cC,c/d
C→.d,c/d

I4: C→d.,c/d

I5: S→CC.,$

I6: C→c.C,$
C→.cC,$
C→.d,$

I7: C→d.,$

I8: C→cC.,c/d

I9: C→cC.,$

si ← shift and stack state i,

rj ← reduce by the production j,

Acc ← accept

Blank ← error

An example

(0) S' → S
(1) S → CC
(2) C → cC
(3) C → d

I0: S'→.S,$
S→.CC,$
C→.cC,c/d
C→.d,c/d

I1: S'→S.,$

I2: S→C.C,$
C→.cC,c/d
C→.d,c/d

I3: C→c.C,c/d
C→.cC,c/d
C→.d,c/d

I4: C→d.,c/d

I5: S→CC.,$

I6: C→c.C,$
C→.cC,$
C→.d,$

I7: C→d.,$

I8: C→cC.,c/d

I9: C→cC.,$

State	Action			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

si ← shift and stack state i,

rj ← reduce by the production j,

Acc ← accept

Blank ← error

Error recovery in LR parsing

State	Action						Goto
	Id	+	*	(	)	$	E
0	s3	e1	e1	s2	e2	e1	1
1	e3	s4	s5	e3	e2	acc	
2	s3	e1	e1	s2	e2	e1	6
3	r4	r4	r4	r4	r4	r4	
4	s3	e1	e1	s2	e2	e1	7
5	s3	e1	e1	s2	e2	e1	8
6	e3	s4	s5	e3	s9	e4	
7	r1	r1	s5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

e1: Missing Operand
e2: Unbalanced right parenthesis
e3: Missing operator
e4: Missing right parenthesis.

LR(k) parser

We call a CFG G an $LR(k)$-grammar, if in each of its rightmost derivations $S' = \alpha_0 \underset{rm}{\Longrightarrow} \alpha_1 \underset{rm}{\Longrightarrow} \alpha_2 \cdots \underset{rm}{\Longrightarrow} \alpha_m = v$ and each right sentential-forms α_i occurring in the derivation

- the handle can be localized, and
- the production to be applied can be determined

$$S' \overset{*}{\underset{rm}{\Longrightarrow}} \alpha X w \underset{rm}{\Longrightarrow} \alpha\beta w \quad \text{and}$$

$$S' \overset{*}{\underset{rm}{\Longrightarrow}} \gamma Y x \underset{rm}{\Longrightarrow} \alpha\beta y \quad \text{and}$$

$$w|_k = y|_k \quad \text{implies} \quad \alpha = \gamma \wedge X = Y \wedge x = y.$$

Resolving conflicts

- Precedence
- Associativity

Homework

Draw LR parsing for following grammar

G=({E,T,F},{+,*,Id,(,)},P,E)

P: $E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid Id$

An expression grammar

$G_1=(\{E\}, \{+,*,(,),Id\}, P_1, E)$

P_1 $E \rightarrow E + E \mid E * E \mid (E) \mid Id$

$G_0=(\{E,T,F\}, \{+,*,(,),Id\}, P_0, E)$

P_0
$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid Id \end{aligned}$$

Structure of a Bison program

```
%{
C declarations
 %}
Bison declarations

%%
Grammar rules

%%
Additional C code
```

Reverse polish notation calculator

```
%{
#define YYSTYPE double
#include <ctype.h>
#include <stdio.h>
%}
%token NUM
%% /* Grammar rules and actions follow */
input:    /* empty */
        | input line
;
line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1); }
;
exp:      NUM          { $$ = $1;         }
        | exp exp '+'     { $$ = $1 + $2; }
        | exp exp '-'     { $$ = $1 - $2;  }
        | exp exp '*'     { $$ = $1 * $2;  }
        | exp exp '/'     { $$ = $1 / $2;   }
;
%%
```

```
yylex ()
{
  int c;

while ((c = getchar ()) == ' ' || c == '\t') ;
if (c == '.' || isdigit (c))    {
      ungetc (c, stdin);
      scanf ("%lf", &yylval);
      return NUM;
    }
if (c == EOF)  return 0;
return c;
}

yyerror (char *s)  /* Called by yyparse on error */
{
  printf ("%s\n", s);
}

main ()
{
  yyparse ();
}
```

Compilation and Execution

```
$ bison firstProg.y
$ cc firstProg.tab.c -lm -o firstProg
$ ./firstProg
….
$
```

Infix notation calculator - I

```
%{
#define YYSTYPE double
#include <ctype.h>
#include <stdio.h>
%}

/* Bison Declaration */
%token NUM
%left '-' '+‘ '*' '/'



%%
```

Unary minus?

```
input:    /* empty */
        | input line
;

line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1); }
;

exp:      NUM                    { $$ = $1;
}
        | exp '+' exp    { $$ = $1 + $3;  }
        | exp '-' exp    { $$ = $1 - $3;  }
        | exp '*' exp    { $$ = $1 * $3;  }
        | exp '/' exp    { $$ = $1 / $3;  }
        | '(' exp ')'    { $$ = $2;       }
;
%%
```

Infix notation calculator - I

```
%{
#define YYSTYPE double
#include <ctype.h>
#include <stdio.h>
%}

/* Bison Declaration */
%token NUM
%left '-' '+‘ '*' '/'
%left NEG     /* negation--unary
minus */

%%
```

Operator precedence?

```
input:    /* empty */
        | input line
;

line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1);
}
;

exp:      NUM                          { $$ = $1;
}
        | exp '+' exp     { $$ = $1 + $3; }
        | exp '-' exp       $$ = $1 - $3;  }
        | exp '*' exp     { $$ = $1 * $3; }
        | exp '/' exp     { $$ = $1 / $3;  }
      /* Unary minus    */
        | '-' exp %prec NEG    { $$ = -$2;}
        | '(' exp ')'            { $$ = $2;  }
;
%%
```

Infix notation calculator - I

```
%{
#define YYSTYPE double
#include <ctype.h>
#include <stdio.h>
%}

/* Bison Declaration */
%token NUM
%left '-' '+‘
%left '*' '/'
%left NEG     /* negation--unary
minus */

%%
```

Right associativity?

```
input:    /* empty */
        | input line
;

line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1);
}
;

exp:      NUM                     { $$ = $1;
}
        | exp '+' exp     { $$ = $1 + $3;    }
        | exp '-' exp     { $$ = $1 - $3;    }
        | exp '*' exp     { $$ = $1 * $3;    }
        | exp '/' exp     { $$ = $1 / $3;    }
      /* Unary minus    */
        | '-' exp %prec NEG     { $$ = -$2;
}
        | '(' exp ')'           { $$ = $2;    }
;
%%
```

Infix notation calculator - I

```
%{
#define YYSTYPE double
#include <ctype.h>
#include <stdio.h>
%}

/* Bison Declaration */
%token NUM
%left '-' '+‘
%left '*' '/'
%left NEG     /* negation--unary
minus */

%%
```

Syntax error?

```
input:    /* empty */
        | input line
;

line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1);
}
        | error '\n'  { yyerrok; }
;

exp:      NUM                    { $$ = $1;
}
        | exp '+' exp     { $$ = $1 + $3;    }
        | exp '-' exp      { $$ = $1 - $3;    }
        | exp '*' exp     { $$ = $1 * $3;    }
        | exp '/' exp     { $$ = $1 / $3;    }
       /* Unary minus    */
        | '-' exp %prec NEG     { $$ = -$2;
}
        | '(' exp ')'             { $$ = $2;
}
;
%%
```

Syntax Tree

```
extern int yylineno;            // interface to lexer  from lexer
void yyerror(char s*, …);

/* node of Abstract Syntax Tree (AST) */
struct AST {
        int nodetype;
        struct AST *l;
        struct AST *r;
};

struct AST *newast(int nodetype, struct AST *l, struct AST *r);        // build
an AST

double eval(struct AST *);              // evaluate an AST
void treefree(struct AST *);            // free AST
```

Bison

```
%{
#include <stdio.h>
#include <stdlib.h>
#include “bison1.h”
%}

%union {
		struct AST *a;
		double d;
}

%token <d> NUMBER
%token EOL

%type <a> exp factor term
```