(COMPILER DESIGN)

UNIT-2

(Pragya Gaur)

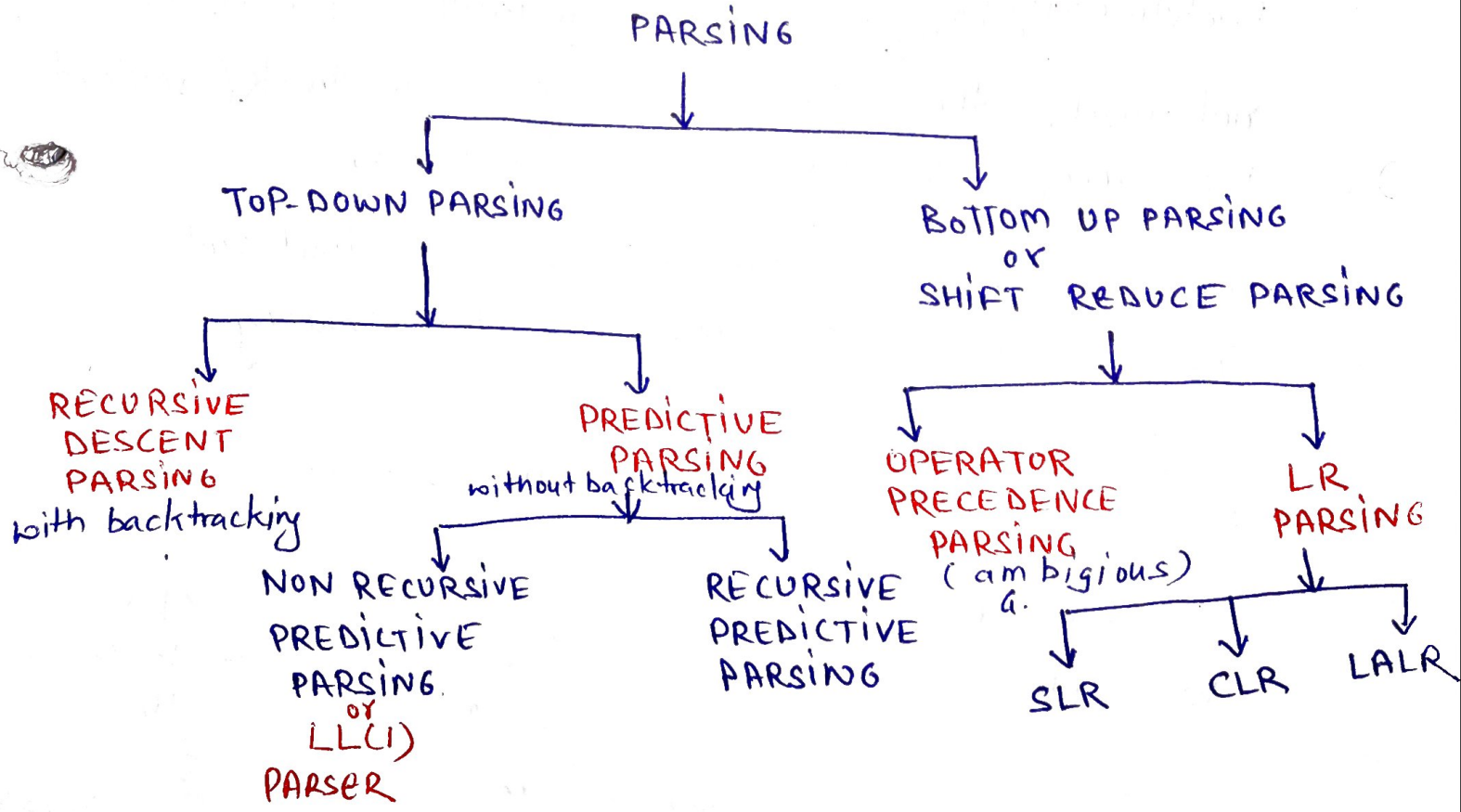
PARSER :- (Parsing is a important phase of Compilation).

A parser is or syntax analyzer is a program that takes the sequence of tokens generated by lexical analyzer as input and group them togethor into syntactic structure.

The goel of parsing is to determine the syntactic validity of source string.

"A Parser is a program that takes a string w as o/p and produce a parse tree for the string if the string is a valid sentence or a error message".

CLASSIFICATION OF PARSER :-

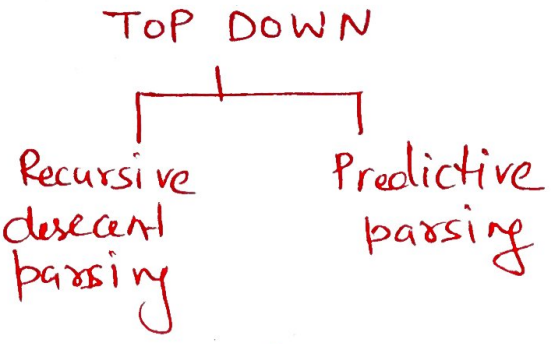


PARSING →

① TOP DOWN PARSING :-

In this strategy the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the i/p.

(B)



(i) RECURSIVE DESCENT PARSING :

It is a common form of top down parsing, it uses recursive procedures to process the i/p. Backtracking is needed in recursive descent parsing.

Backtracking :- If a choice of a production rule does not work, then syntax analyzer backtrack (restart the process) to try other alternatives.

RECURSIVE DESCNT PARSER tries to find out the left most derivation using backtracking.

Consider the example :-

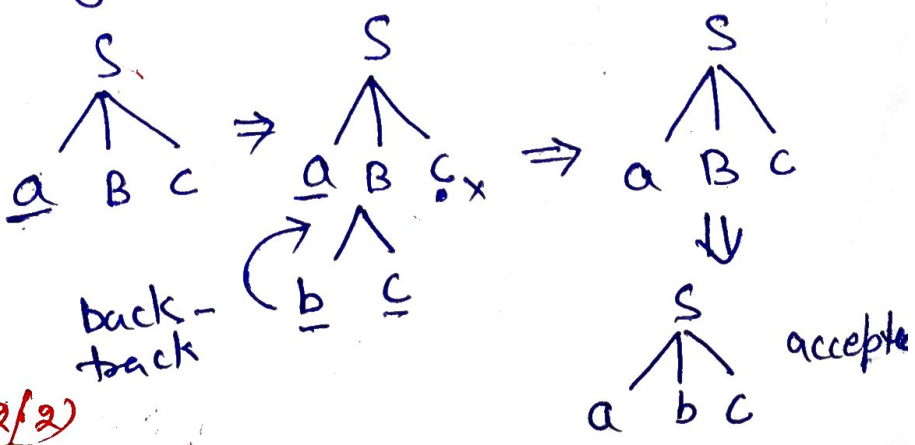
G :

$S \rightarrow aBC$

$B \rightarrow bc \mid b$ ⇒

$w = abc$

Parsing proceeds as -



(II) Predictive Parser : (~~Recursive descent parsing - no backtracking~~)

To make a grammar suitable for predictive parsing remove/eliminate left recursion and left factor the grammar if required.

grammar → eliminate left-Recursio → left factor ⇒ (suitable for predictive parsing) LL(1) Grammar

LL(1) → just looking the current token

- L → left to right scan
- L → leftmost derivation

① Recursive Predictive parsing :-

(Recursive descent parsing - no backtracking)

Each non terminal corresponds to a procedure.

lookhead - holding the current i/p ~~token~~ and a procedure match - action of recognizing the next token in the parsing process.

example-

expr → term rest

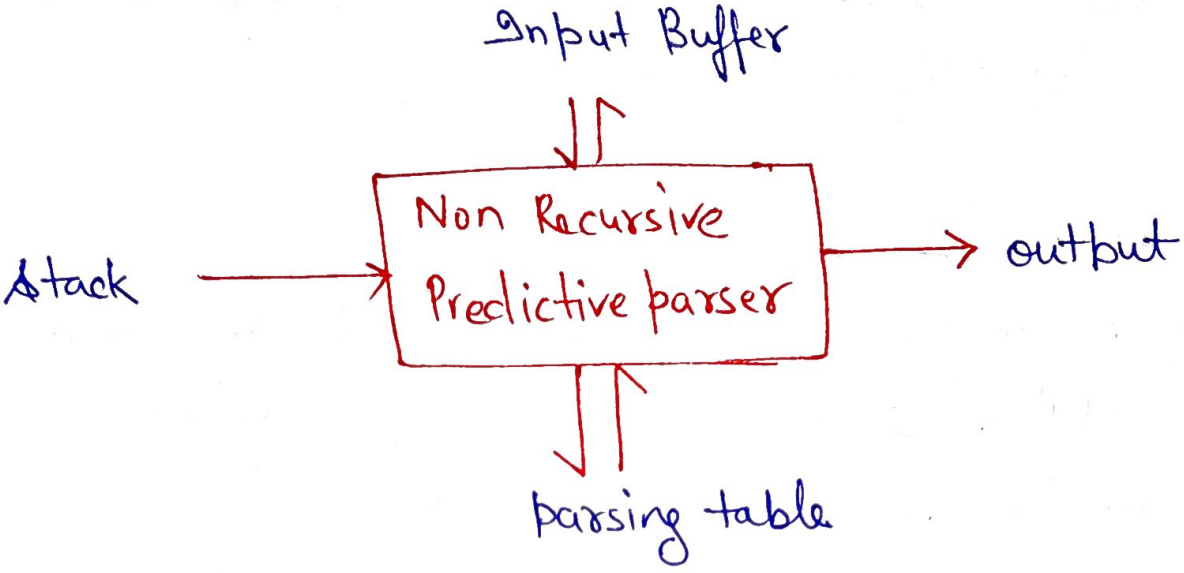
rest → + term rest | - term rest | ∈

term → 0|1|2|---|9

Pragya Gaur

Non Recursive Predictive Parsing:

non-recursive predictive parsing is a table driven parsing.



Input Buffer :.

Input buffer contains the string to be parsed, and the end of string is marked with a special symbol $.

Output :.

output contains a production rule representing a step of the derivation sequence of the string in the Ip buffer.

Stack :

stack contains the grammar symbol.

initially the stack contains only the symbol $ and starting symbol S. ($S)

- when only $ left in the stack, the parsing is complete

parsing table :-

	a	b	$
S	S→aBa		
B	B→ϵ	B→bB	

→ each row is a non-terminal symbol.
→ each column is a terminal symbol or the special end marker $.
→ each entry holds a production rule.

Parser Actions :-

The symbol at the top of the stack and the current symbol in the input string determine the parser action.

There are four possible parser actions

Let X be the symbol at the top of stack and a represents the current symbol in the i/p string.

Action-1

ex →

Stack	i/p	o/p
$	$	accept.

if X and a are $
then parser halts (successful completion)

Action-2

if X and a are same terminal symbol
then parser pops X from the stack and move the next symbol in the i/p buffer

Stack	i/p buffer	o/p
$aBa	abba$	
$aB	bba$	

Action-3

If X is a non-terminal

then parser looks at the parsing table entry M[X, a]

if M[X, a] holds a production rule $X \to Y_1 \dots Y_n$, it pops X from the stack and pushes $Y_n \dots Y_1$ into the stack.

stack	i/p buffer	o/p
$ aB	bba $	B → bB
$ aBb	bba $	

→ All empty entries in the parsing table are errors.

→ If X is a terminal symbol other then a, this is also an error case.

Example :-

G:

S → aBa
B → bB
B → ϵ

w = abba

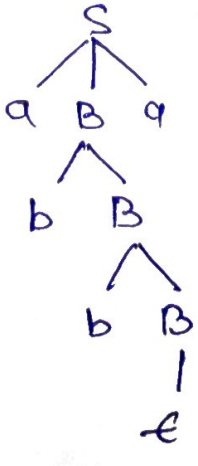
LL(1) parsing table -

	a	b	$
S	S → aBa		
B	B → ϵ	B → bB	

stack	i/p buffer	o/p
$S	abba $	S → aBa
$aBa	abba $	
$aB	bba $	B → bB
$aBb	bba $	
$aB	ba $	B → bB
$aBb	ba $	
$aB	a $	B → ϵ
$a	a $	
$	$	accept.

output :
Derivation

S ⇒ aBa ⇒ abBa ⇒ abbBa
⇓
abba



pragya Gaur

Constructing LL(1) parsing table :-

to construct LL(1) parsing table two function are required-

1) FIRST()
2) FOLLOW()

FIRST() →

FIRST(α) is a set of terminal symbols which occur as first symbol in strings derived from α.
(α indicates any string of grammar symbols)

Rules to compute FIRST() for any string X:

Rule-1: if X is a terminal symbol
→ FIRST(X) = {X}

Rule-2: If X is a non-terminal symbol -
and $X \to \epsilon$ is a production rule
then ε is in FIRST(X)

Rule-3) If X is a non terminal and $X \to Y_1 \dots Y_n$ is a production rule, then

1) if terminal a in FIRST(Y_i) and ε is in all FIRST(Y_j) (for $j = 1 \dots i-1$), then a is in FIRST(X).
2) if ε is in FIRST(Y_j) (for $j = 1 \dots n$). then ε is in FIRST(X).

Rule-4. · If X is ϵ
then FIRST(X) = $\{\epsilon\}$

Rule-5: If X is $Y_1, Y_2 \ldots Y_n$, then-

1) If terminal a is in FIRST(Y_i) and ϵ is in all FIRST(Y_j) for $j = 1 \ldots i-1$ then a is in FIRST(X).

2) If ϵ is in all FIRST(Y_j) for $j = 1 \ldots n$ then ϵ is in FIRST(X).

Example:-

$S \to aBDh$
$B \to eC$
$C \to bC \mid \epsilon$
$D \to EF$
$E \to g \mid \epsilon$
$F \to f \mid \epsilon$

★ FIRST(F) = ?

$F \to f$
$F \to \epsilon$

FIRST(F) = $\{f, \epsilon\}$

★ FIRST(E) = ?

$E \to g$
$E \to \epsilon$

FIRST(E) = $\{g, \epsilon\}$

★ FIRST(D) = ?

$D \to EF$

FIRST(D) = FIRST(EF)
= (FIRST(E) − ϵ) ∪ {FIRST(F)} ⇒ FIRST(D) = $\{g, f, \epsilon\}$
= $\{g, f\} \cup \{f, \epsilon\}$ ⇒ $\{g, f, \epsilon\}$

★ FIRST(C) = ?

$C \to bC$
$C \to \epsilon$

FIRST(C) = $\{b, \epsilon\}$

★ FIRST(B) = ?

$B \to eC$

FIRST(B) = $\{e\}$

★ FIRST(S) = ?

$S \to aBDh$

FIRST(S) = $\{a\}$

Pragya (2/01)

example→

S→ ACB | CbB | Ba

A→ da | BC

B→ g | ϵ

C→ h | ϵ

★ C→ h

C→ ϵ

FIRST(C) = {h, ϵ}

★ B→ g

B→ ϵ

FIRST(B) = {g, ϵ}

★ A→ da

A→ BC

FIRST(A) = {d} ∪ {FIRST(B) − {ϵ}} ∪ FIRST(C)

= {d} ∪ {g} ∪ {h, ϵ}

= {d, g, h, ϵ}

★ S→ ACB

S→ CbB

S→ Ba

FIRST(S) = {FIRST(ACB)} ∪ {FIRST(CbB)} ∪ {FIRST(Ba)}

= {d, g, h, ϵ} ∪ {h, b} ∪ {g, ϵ, a}

= {a, b, d, g, h, ϵ}

FIRST(ACB) = FIRST(A) & if FIRST(A) contain ϵ

then FIRST(ACB) = (FIRST(A) − {ϵ}) ∪ FIRST(CB)

= {d, g, h} ∪ [{h} ∪ {g}] ⇒ {d, g, h, ϵ}

FOLLOW() →

1) If S is a start symbol
   ⇒ $ is in FOLLOW(S).

2) If A → αBβ is a production rule, then everything in FIRST(β) is in FOLLOW(B) except ∈.

3) If A → αB is a production rule or
   A → αBβ is a production rule and ∈ is in FIRST(β)
   then everything in FOLLOW(A) is in FOLLOW(B).

Example:

S → ACB | CbB | Ba
A → da | BC
B → g | ∈
C → h | ∈

★ FOLLOW(S) = {$}

★ FOLLOW(A) = ?

S → A C B
A → α B β

FOLLOW(A) = FIRST(CB)
= {h} ∪ {g, ∈}
= {h, g} ∪ FOLLOW(S)
= {h, g, $}

★ follow(B) = ?

S → A C B
A → α B β

FOLLOW(B) = FOLLOW(S) = {$}

FOLLOW(C) = {$, b, g, h}

S → Ba
A → α B β

FOLLOW(B) = FIRST(a)
= {a}

⇒ FOLLOW(B) = {a, $}

★ FOLLOW(C) = ?

S → A C B
A → α B β

FOLLOW(C) = FIRST(B)
= {g, ∈}

As FIRST(B) contain ∈

FOLLOW(C) = {g} ∪ FOLLOW(S)
= {g, $}

S → C b B
A → α B β

FOLLOW(C) = FIRST(bB)
= {b}

A → BC
A → α B β

FOLLOW(C) = FOLLOW(A) = {h, g, $}

ex. $S \to aBDh$
$B \to eC$
$C \to bC \mid \epsilon$
$D \to EF$
$E \to g \mid \epsilon$
$F \to f \mid \epsilon$

FIRST(F) = {f, ε}
FIRST(E) = {g, ε}
FIRST(D) = FIRST(EF)
= {f, g, ε}
FIRST(C) = {b, ε}
FIRST(B) = {e}
FIRST(S) = {a}

* FOLLOW(S) = {$}

$S \to aBDh$
$A \to \alpha B \beta$

FOLLOW(B) = FIRST(Dh)
= {f, g, h}

FOLLOW(C) = ~~FIRST(bC)~~ ?

$B \to eC$
$A \to \alpha B \beta$

FOLLOW(C) = FOLLOW(B)
= {f, g, h}

$S \to aBDh$
$A \to \alpha B \beta$

FOLLOW(D) = {h}

$D \to EF$

FOLLOW(E) = FIRST(F)
= {f} ∪ FOLLOW(D)
= {f, h}

FOLLOW(F) = ?

$D \to EF$
$A \to \alpha B \beta$

FOLLOW(F) = FOLLOW(D)
= {h}

Rules to Construct Parsing table →

→ for each production rule $A \to \alpha$ of grammar G:
- for each terminal a in FIRST(α)
  Add $A \to \alpha$ to M[A, a]
- If ε is in FIRST(α)
  for each terminal a in FOLLOW(A)
  Add $A \to \alpha$ to M[A, a]
- If ε is in FIRST(α) and $ is in FOLLOW(A)
  Add $A \to \alpha$ to M[A, $]

Pragya Gaur

Example:-

follow ↓ / ↑ first

$E \to TE'$
$E' \to +TE' \mid \epsilon$
$T \to FT'$
$T' \to *FT' \mid \epsilon$
$F \to (E) \mid id$

$FIRST(F) = \{(, id\}$

$FIRST(T') = \{*, \epsilon\}$

$FIRST(T) = FIRST(F) = \{(, id\}$

$FIRST(E') = \{+, \epsilon\}$

$FIRST(E) = FIRST(T) = \{(, id\}$

$FOLLOW(E) = \{\$,)\}$

$FOLLOW(E') = \{\$,)\}$

$FOLLOW(T) = \{+,), \$\}$

$FOLLOW(T') = \{+,), \$\}$

$FOLLOW(F) = \{+, *,), \$\}$

$FOLLOW(F) = FIRST(T') = \{*, \epsilon\}$

Since $FIRST(T')$ contain ϵ $FOLLOW(F)$ also include $FOLLOW(T)$ & $FOLLOW(T')$

$\Rightarrow FOLLOW(F) = \{*, +,), \$\}$

fragya Goel

$FOLLOW(E) = ?$

$F \to (E)$
$A \to \alpha B \beta$

$\boxed{FOLLOW(E) = \{\$,)\}}$

$FOLLOW(E') = ?$

$A \to \alpha B \beta$
$E \to TE'$

$E' \to +TE'$
$A \to \alpha B \beta$

$FOLLOW(E') = FOLLOW(E)$

$\boxed{FOLLOW(E') = \{\$,)\}}$

$FOLLOW(T) = ?$

$A \to \alpha B \beta$
$E \to TE'$ ——— (1)

$E' \to +TE'$ ——— (2)
$A \to \alpha B \beta$

from (1), (2)

$FOLLOW(T) = FIRST(E') = \{+, \epsilon\}$

As $FIRST(E')$ contain ϵ $FOLLOW(T)$ also include $FOLLOW(E)$ and $FOLLOW(E')$

$FOLLOW(T) = \{+,), \$\}$

$FOLLOW(T') = ?$

$T \to FT'$
$T' \to *FT'$

$FOLLOW(T') = FOLLOW(T) = \{+,), \$\}$

$FOLLOW(F)$
$T \to FT'$ ——— (1)
$T' \to *FT'$ ——— (2)

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

① $E \to TE'$
FIRST(TE') = {(, id}
but $E \to TE'$ in
M[E, (] & M[E, id]

② $E' \to +TE'$
FIRST(+TE') = {+}
but $E' \to +TE$ in M[E', +]

③ $E' \to \epsilon$
FOLLOW(E') = {$,)}
but $E' \to \epsilon$ in M[E',)] & M[E', $]

④ $T \to FT'$
FIRST(FT') = {(, id}
but $T \to FT'$ in M[T, (] & M[T, id]

⑤ $T' \to *FT'$
FIRST(*FT') = {*}
but $T' \to *FT'$ in M[T', *]

⑥ $T' \to \epsilon$
FOLLOW(T') = {+,), $}, but $T' \to \epsilon$ in M[T', +], M[T',)] & M[T', $]

⑦ ~~$T' \to \epsilon$~~ $F \to (E)$
~~First~~ FIRST[(E)] = {(}
but $F \to (E)$ in M[F, (]

⑧ $f \to id$
FIRST(id) = {id
but $F \to id$ in M[F, id]

LL(1) Grammar :-

A grammar whose parsing table has no multiple-defined enteries is said to be LL(1) grammar.

rules:-

A grammar G is LL(1) if and only if the following conditions hold for two distinctive production rules - $A \to \alpha$ & $A \to \beta$

1) Both α & β cannot derive strings starting with same terminal symbols -

$\Rightarrow$ $FIRST(\alpha) \cap FIRST(\beta) = \phi$

2) At most one of α and β can derive to ϵ

if β can derive to ϵ, then α cannot derive to any string starting with a terminal in FOLLOW(A)

$\Rightarrow$ If FIRST(β) contain ϵ and FIRST(α) does not contain ϵ

then $FIRST(\alpha) \cap FOLLOW(A) = \phi$

example →

$S \to ACB \mid CbB$

$A \to da \mid BC$

$B \to g \mid \epsilon$

$C \to h \mid \epsilon$

for S $A \to \alpha$ $A \to B$

$S \to ACB \mid S \to CbB$

$\Rightarrow FIRST(ACB) \cap FIRST(CbB)$

$\Rightarrow \{d\} \cap \{h, \epsilon\} \Rightarrow FIRST(ACB) \cap FOLLOW(S)$

$= \phi$ $= \{d\} \cap \{\$\}$

$= \phi$

for A $A \to da \mid A \to BC$

$FIRST(da) \cap FIRST(BC$

$\Rightarrow \{d\} \cap \{g, \epsilon\} \Rightarrow FIRST(da) \cap FOLLOW(B)$

$= \phi$ $\Rightarrow \{d\} \cap \{h, \$\}$

$\Rightarrow \phi$

Thus the Grammar is not LL(1)

Pragya Gaur

(2/14)

NOT LL(1) Grammar :-

ex.

$S \rightarrow iCtSE \mid a$

$E \rightarrow eS \mid \epsilon$

$C \rightarrow b$

FOLLOW(S) = {$, e}

FOLLOW(E) = {$, e}

FOLLOW(C) = {t}

	a	b	e	i	t	$
S	S→a			S→iCtSE		
E			E→eS E→ε			E→ε
C		C→b				

A grammar having ~~2 entries~~ multiple entries in a block is not LL(1) grammar.

→ A left recursive grammar can not be LL(1)

→ A grammar which is not left factored, can not be LL(1) grammar.

→ An ambigious grammar can not be LL(1)