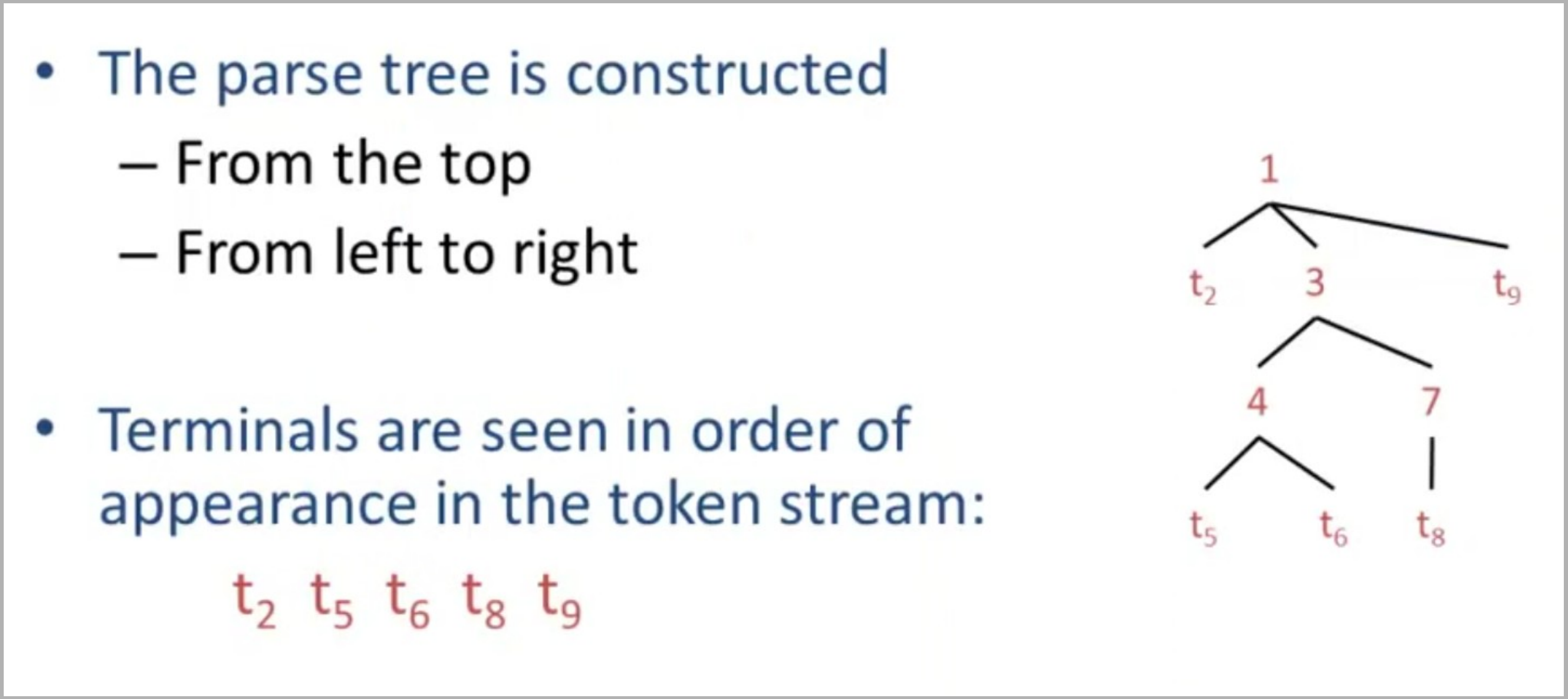
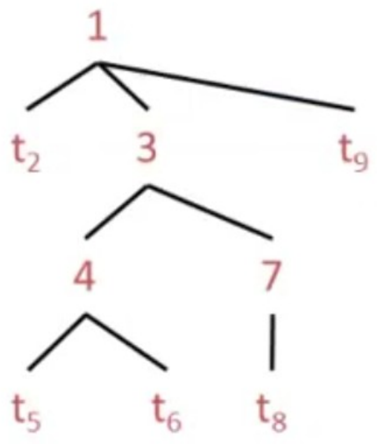
Recursive Descent Parsing Algorithm – top down parsing.



- 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 t_5 t_6 t_8 t_9



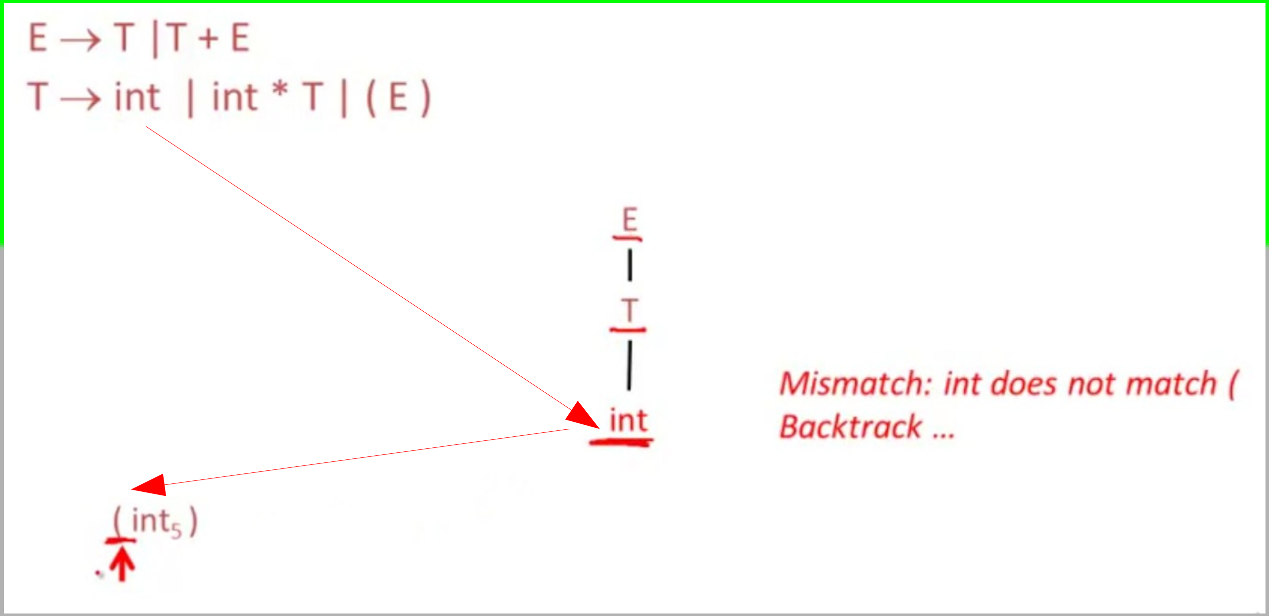
- Consider the grammar

  $E \rightarrow T \mid T + E$

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

- Token stream is: $(\; int_5 \;)$

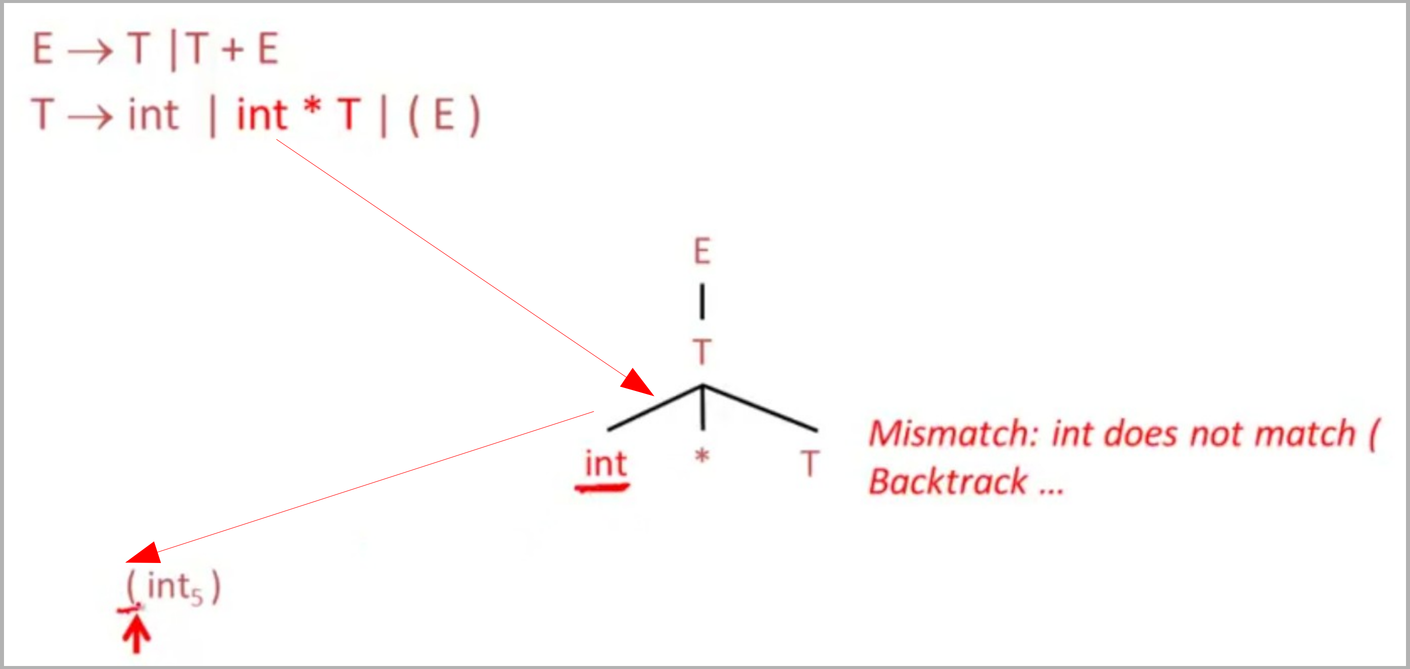
- Start with top-level non-terminal E
  - Try the rules for E in order



$E \rightarrow T \mid T + E$

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

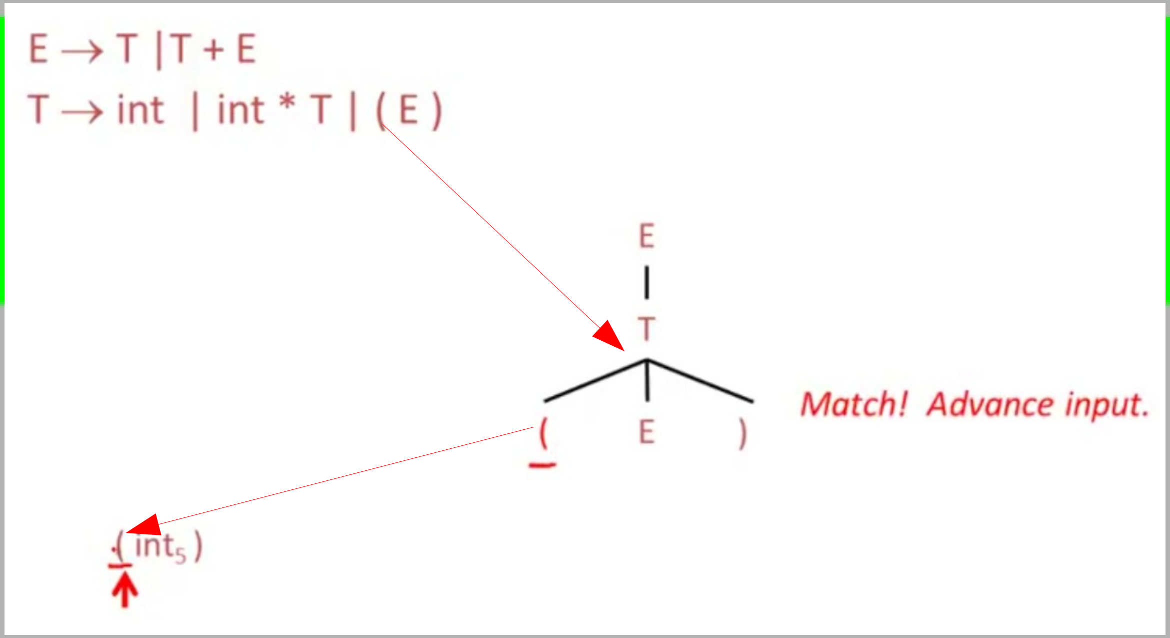
Mismatch: int does not match (
Backtrack ...



$E \rightarrow T \mid T + E$

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

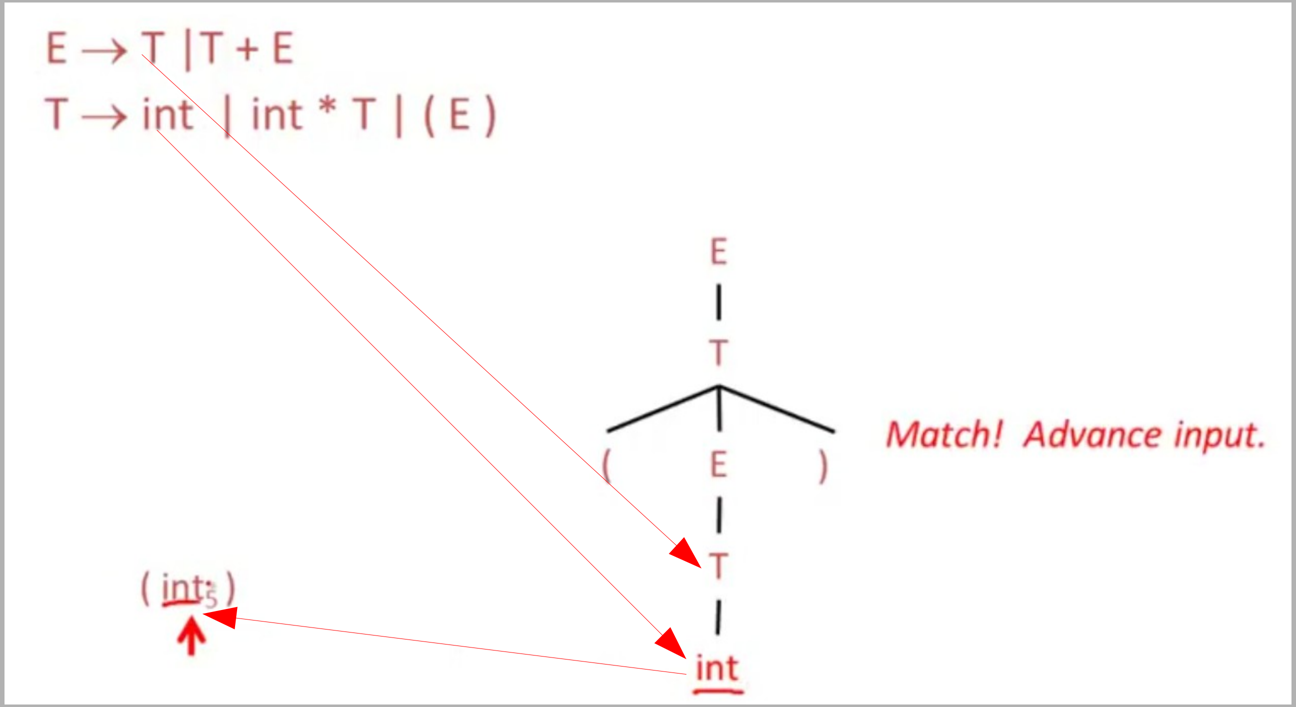
Mismatch: int does not match (
Backtrack ...



$E \rightarrow T \mid T + E$

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

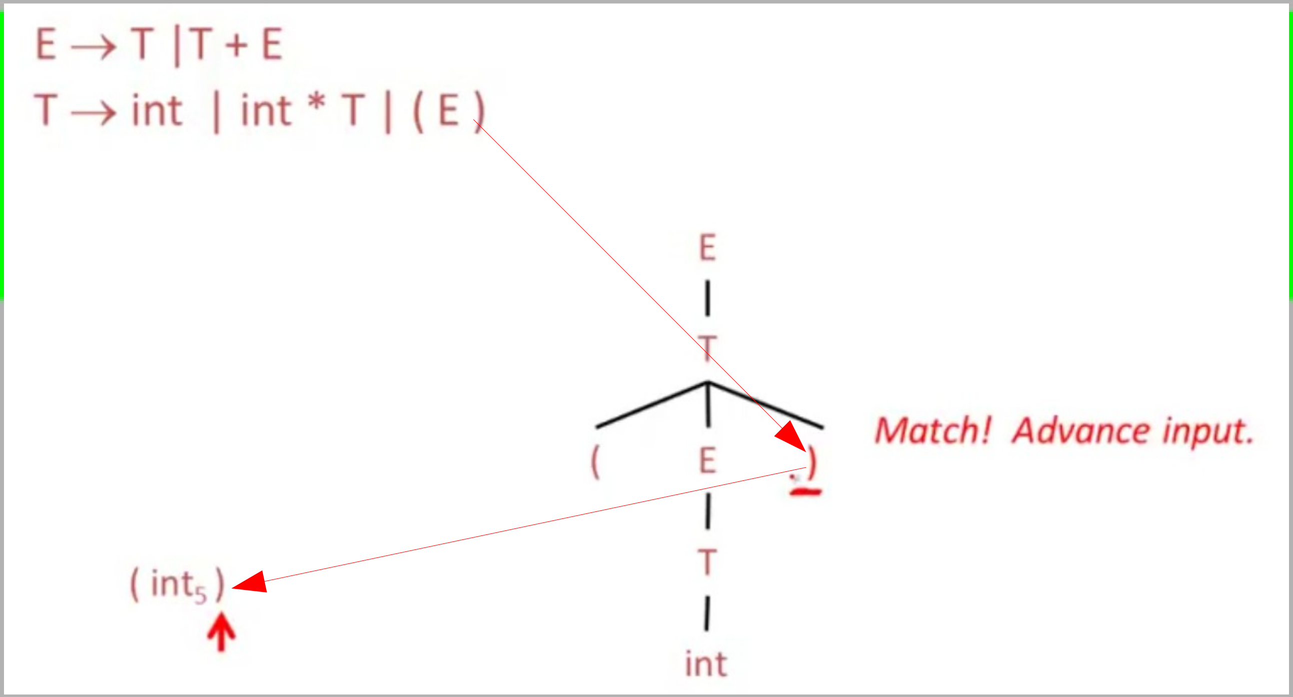
Match! Advance input.



$E \rightarrow T \mid T + E$

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

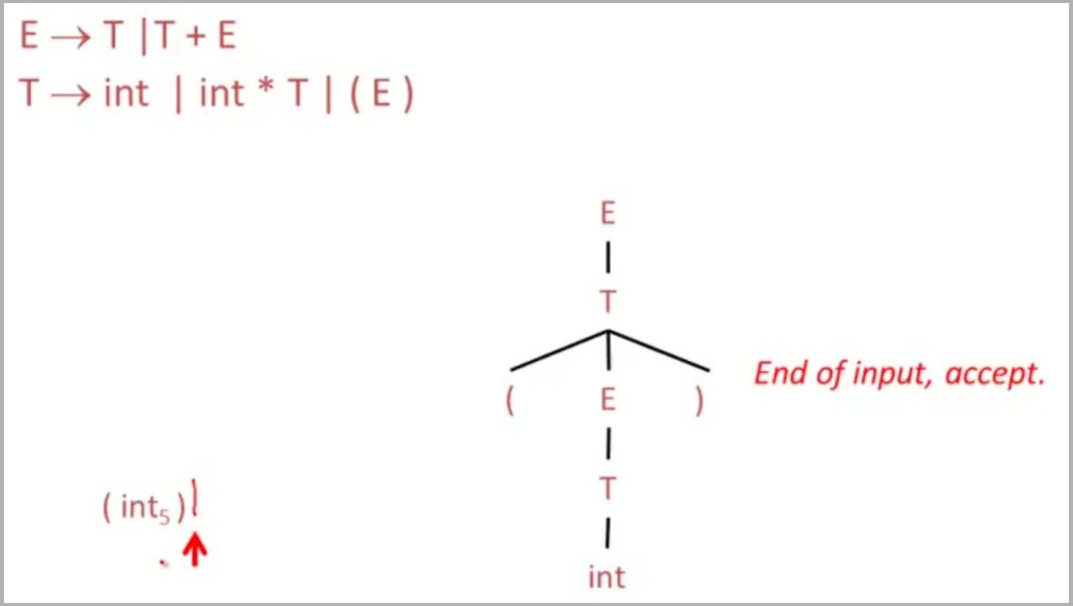
Match! Advance input.



$E \rightarrow T \mid T + E$

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

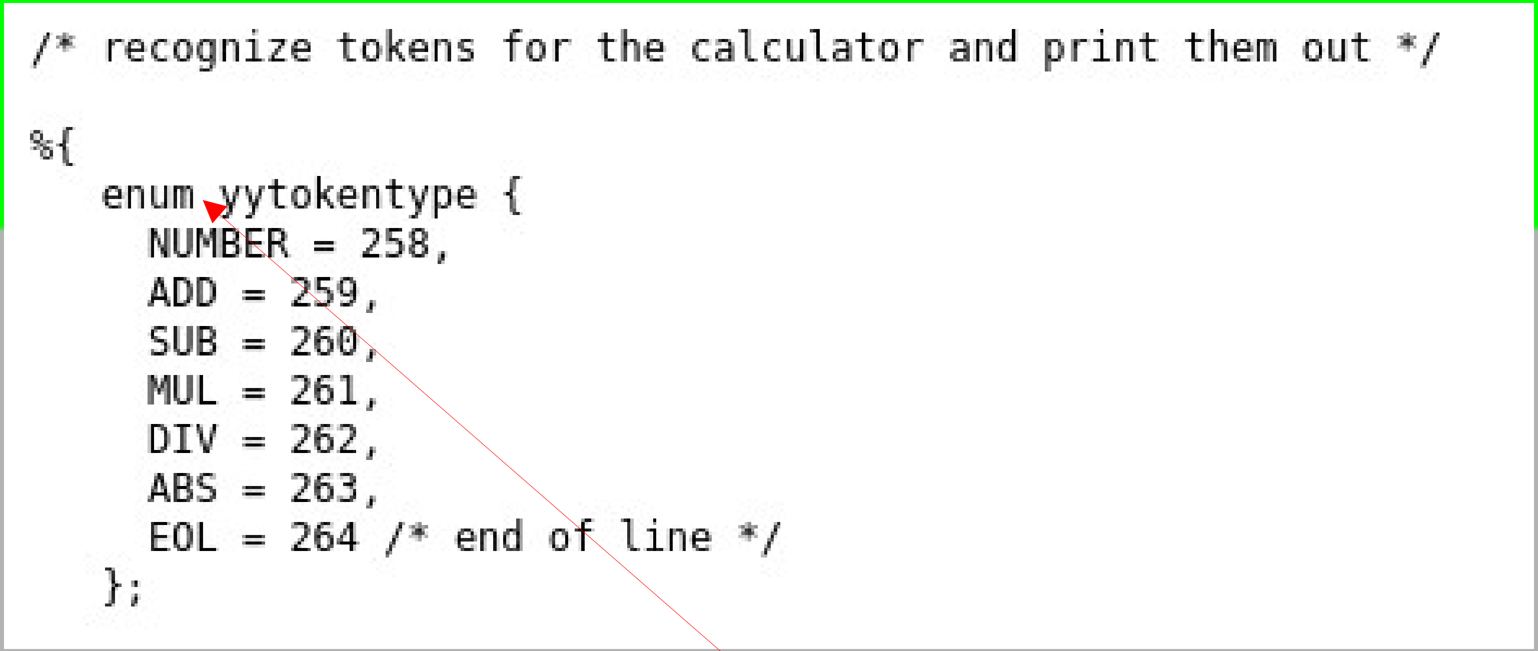
Match! Advance input.



$E \rightarrow T \mid T + E$

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

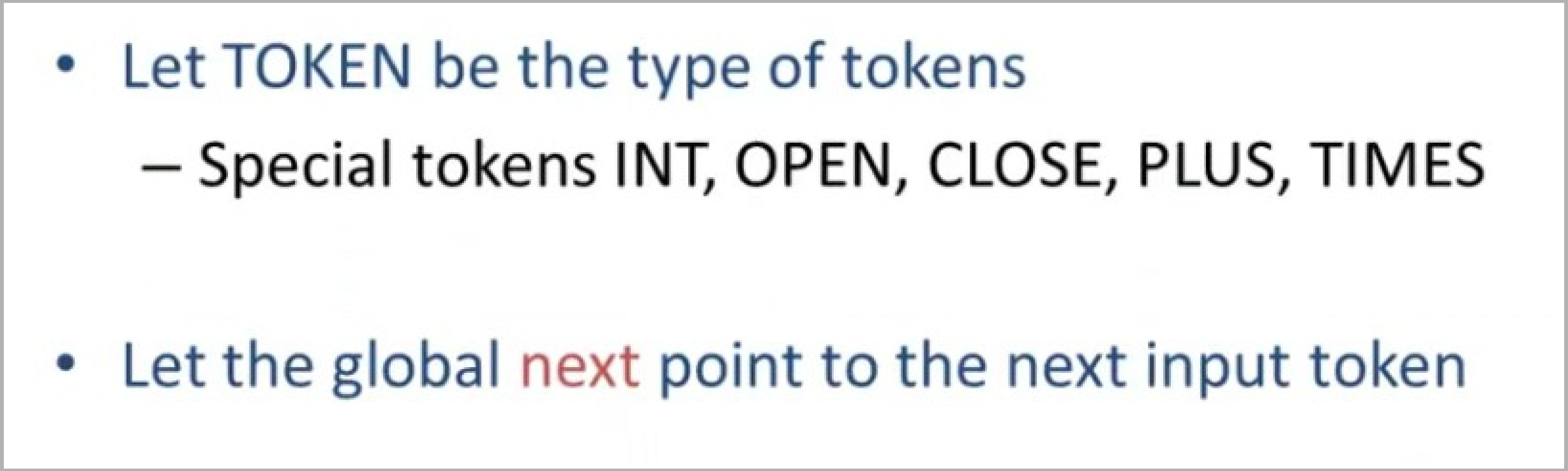
End of input, accept.



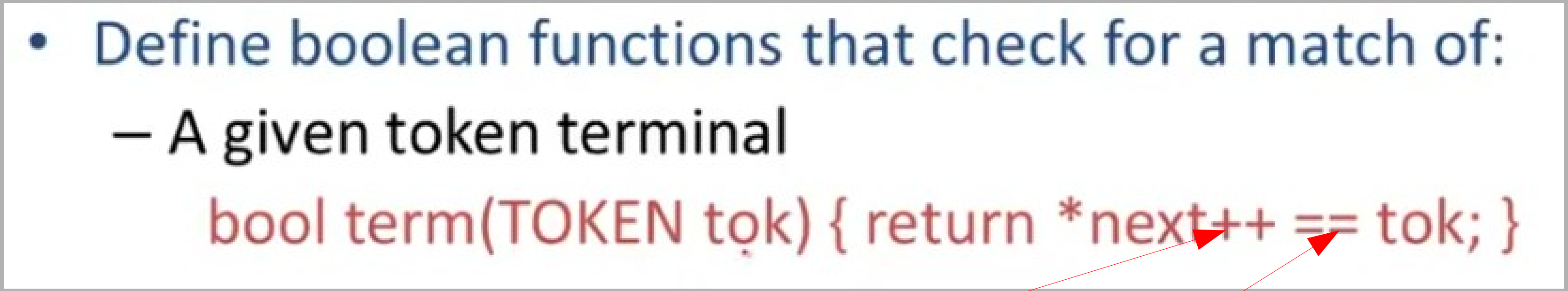
```
/* recognize tokens for the calculator and print them out */

%{
   enum yytokentype {
     NUMBER = 258,
     ADD = 259,
     SUB = 260,
     MUL = 261,
     DIV = 262,
     ABS = 263,
     EOL = 264 /* end of line */
   };
```

Enumeration (or enum) is a user defined data type in C. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.



- Let TOKEN be the type of tokens
  - Special tokens INT, OPEN, CLOSE, PLUS, TIMES
- Let the global next point to the next input token



- Define boolean functions that check for a match of:
  - A given token terminal

```
bool term(TOKEN tok) { return *next++ == tok; }
```

advances next, returns boolean

$E \to T \mid T + E$

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

– The nth production of S: `bool S_n() { … }`

– Try all productions of S: `bool S() { … }`

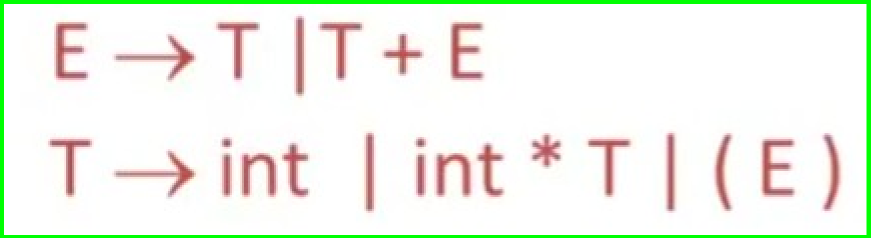
- For production $E \to T$ `bool E_1() { return T(); }`

these advance next

- For production $E \to T + E$

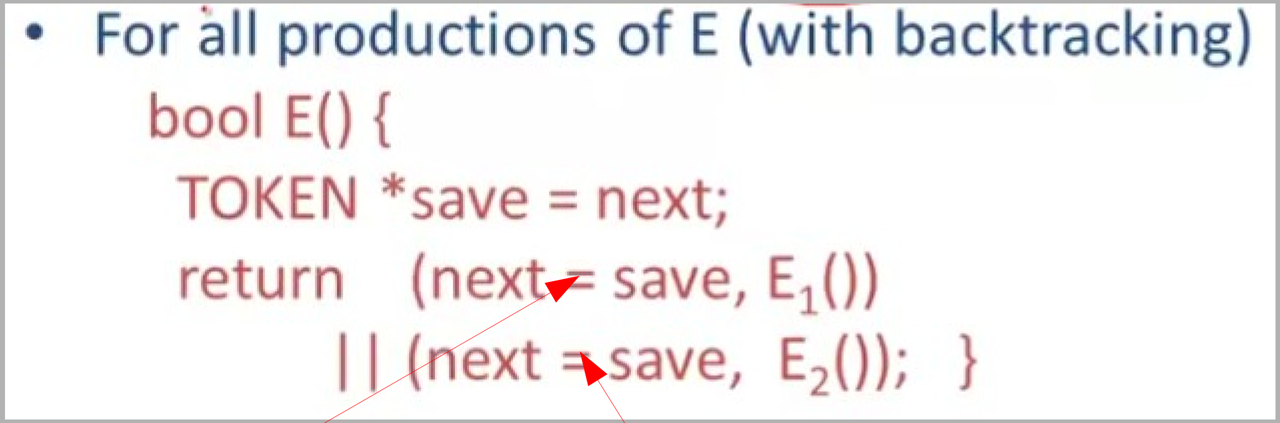
```
bool E_2() { return T() && term(PLUS) && E(); }
```

&& evaluates arguments in left to right order



$E \rightarrow T \mid T + E$

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



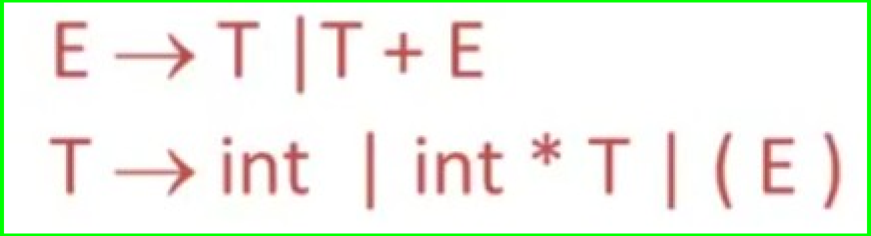
- For all productions of E (with backtracking)

```
bool E() {
 TOKEN *save = next;
 return    (next = save, E1())
           || (next = save,  E2());   }
```

|| if first branch succeeds, do not bother with second branch

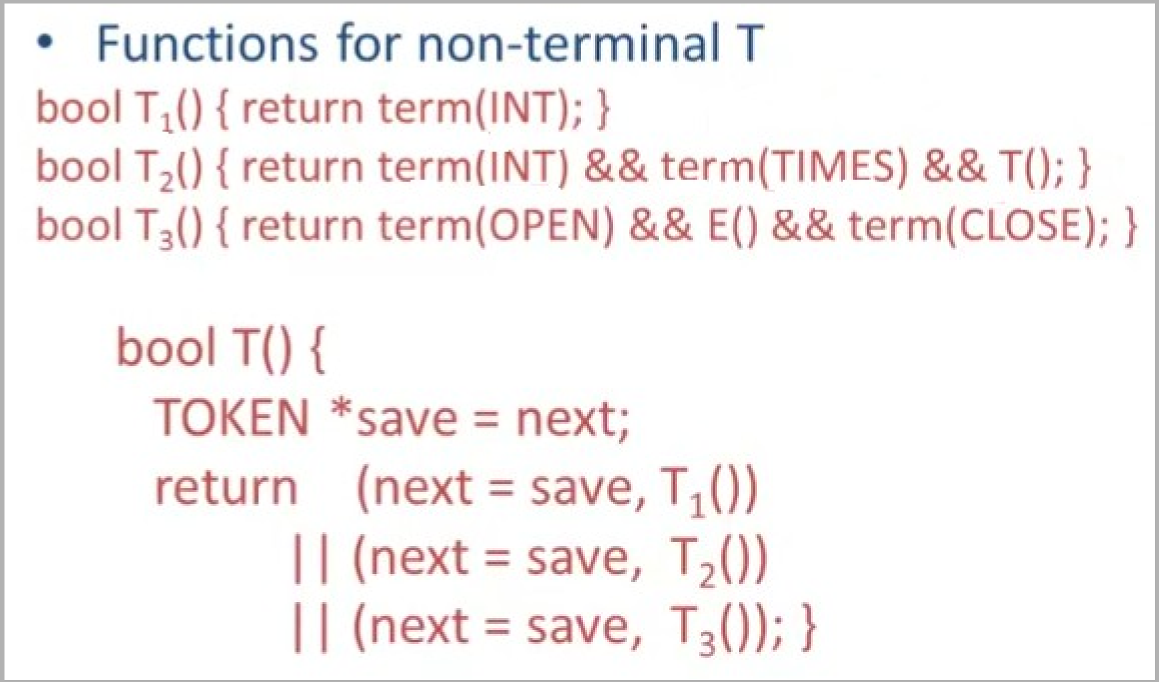
backtracking

if they all fail, the higher level will do the backtracking



$E \rightarrow T \mid T + E$

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

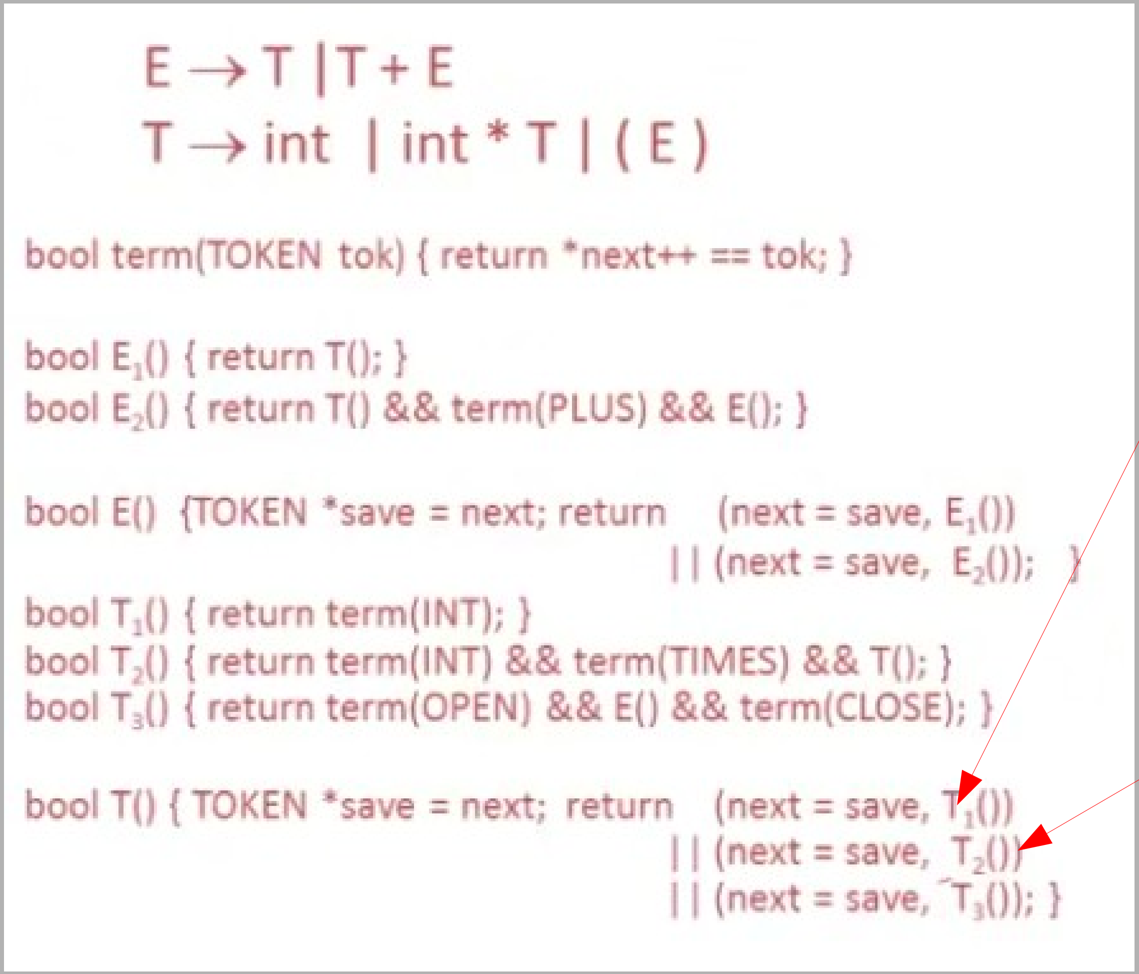


- Functions for non-terminal T

```
bool T1() { return term(INT); }
bool T2() { return term(INT) && term(TIMES) && T(); }
bool T3() { return term(OPEN) && E() && term(CLOSE); }

    bool T() {
      TOKEN *save = next;
      return    (next = save, T1())
             || (next = save,  T2())
             || (next = save,  T3()); }
```

A limitation of recursive descent



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

```
bool term(TOKEN tok) { return *next++ == tok; }

bool E1() { return T(); }
bool E2() { return T() && term(PLUS) && E(); }

bool E()  {TOKEN *save = next; return    (next = save, E1())
                                      || (next = save,  E2());   }
bool T1() { return term(INT); }
bool T2() { return term(INT) && term(TIMES) && T(); }
bool T3() { return term(OPEN) && E() && term(CLOSE); }

bool T() { TOKEN *save = next;  return    (next = save, T1())
                                       || (next = save,  T2())
                                       || (next = save,  T3()); }
```

int * int will be rejected

once a non terminal succeeds
no way to try another production

- If a production for non-terminal X succeeds
  - Cannot backtrack to try a different production for X later
- General recursive-descent algorithms support such "full" backtracking
  - Can implement any grammar

- Presented recursive descent algorithm is not general
  - But is easy to implement by hand
- Sufficient for grammars where for any non-terminal at most one production can succeed
- The example grammar can be rewritten to work with the presented algorithm
  - By *left factoring*

Left Recursion

In the formal language theory of computer science, left recursion is a special case of recursion where a string is recognized as part of a language by the fact that it decomposes into a string from that same language (on the left) and a suffix (on the right).

- Consider a production $S \rightarrow S\ a$

```
bool S_1() { return S() && term(a); }

bool S() { return  S_1(); }
```

non empty sequence of rewrites

- A left-recursive grammar has a non-terminal S

  $S \rightarrow^{+} S\alpha$ for some α

- Recursive descent does not work in such cases

- Consider the left-recursive grammar

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

$S \rightarrow S\alpha \rightarrow S\alpha\alpha \rightarrow S\alpha\alpha\alpha \rightarrow \cdots \rightarrow S\alpha\cdots\alpha \rightarrow \beta\alpha\cdots\alpha$

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

zero or more

- Can rewrite using right-recursion

  $S \rightarrow \beta S'$

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

$S \rightarrow \beta S' \rightarrow \beta\alpha S' \rightarrow \beta\alpha\alpha S' \rightarrow \cdots$
$\rightarrow \beta\alpha\cdots\alpha S' \rightarrow \beta\alpha\cdots\alpha$

[redacted]

- In general

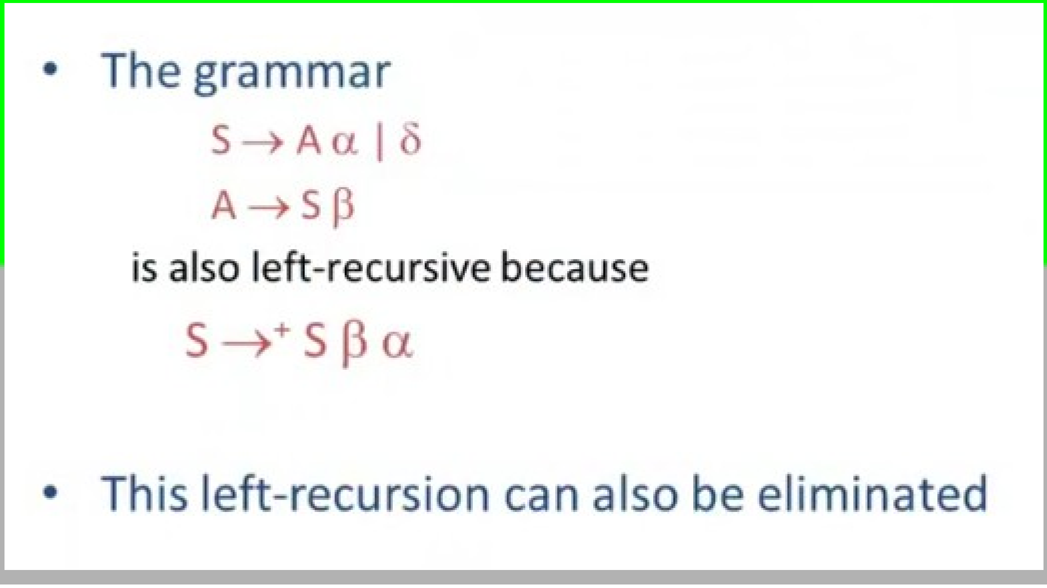
$$S \rightarrow S\,\alpha_1 \mid \dots \mid S\,\alpha_n \mid \beta_1 \mid \dots \mid \beta_m$$

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

zero or more

- Rewrite as

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



- The grammar

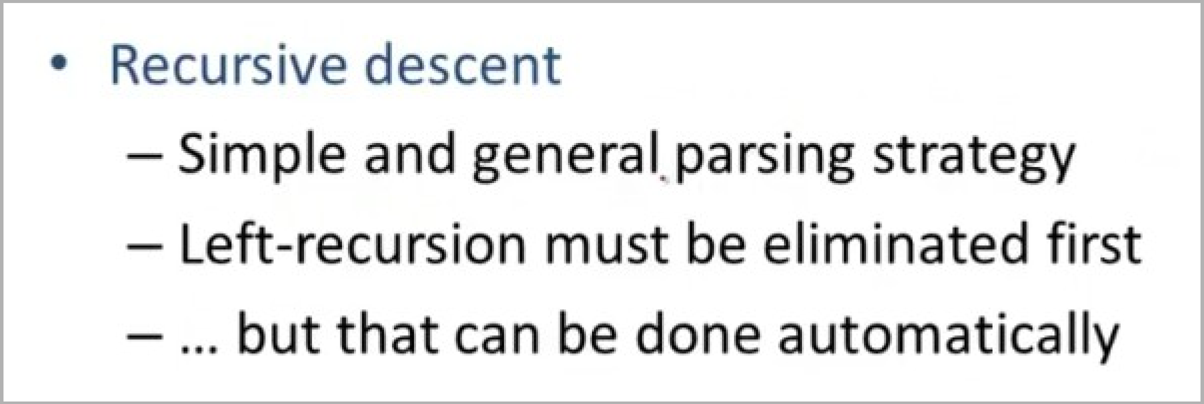
  $S \to A\,\alpha \mid \delta$

  $A \to S\,\beta$

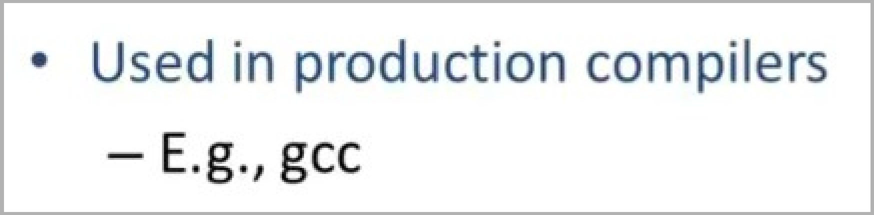
  is also left-recursive because

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

- This left-recursion can also be eliminated

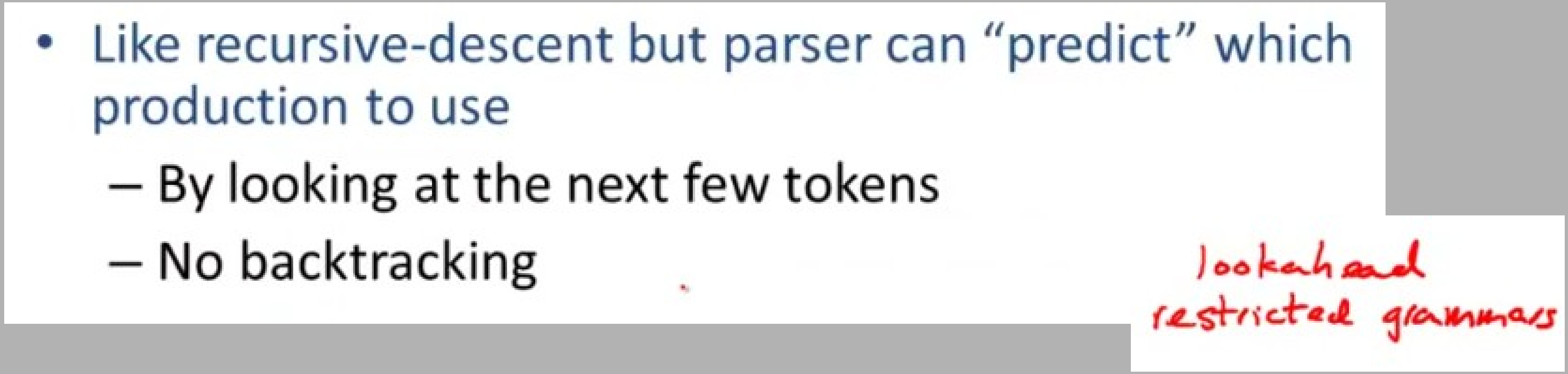


- Recursive descent
  - Simple and general parsing strategy
  - Left-recursion must be eliminated first
  - … but that can be done automatically



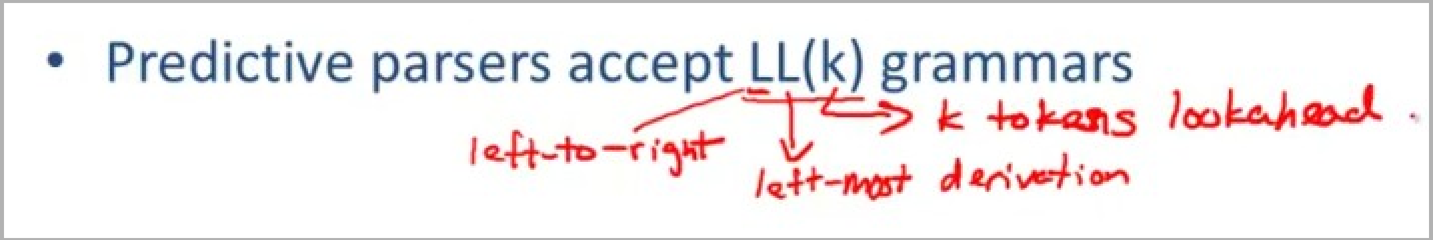
- Used in production compilers
  - E.g., gcc

Predictive Parsers



- Like recursive-descent but parser can "predict" which production to use
  - By looking at the next few tokens
  - No backtracking

lookahead
restricted grammars

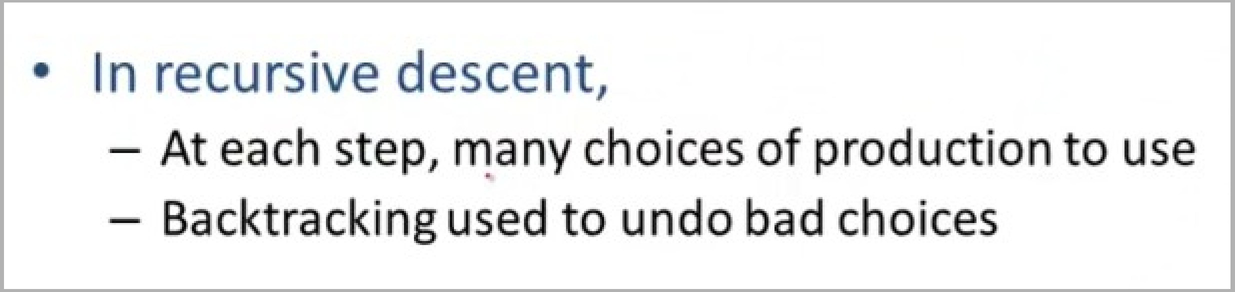


- Predictive parsers accept LL(k) grammars

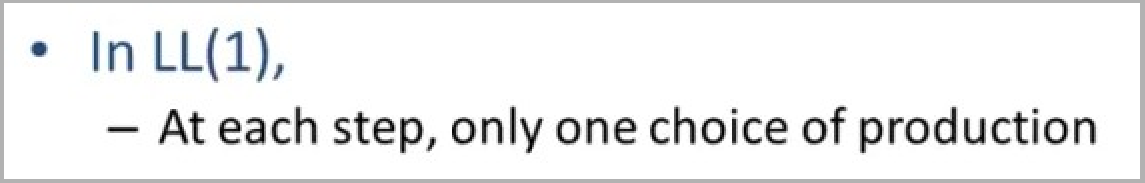
left-to-right
left-most derivation
k tokens lookahead.

always k=1

A deterministic model of computation is a model of computation such that the successive states of the machine and the operations to be performed are completely determined by the preceding state.



- In recursive descent,
  - At each step, many choices of production to use
  - Backtracking used to undo bad choices



- In LL(1),
  - At each step, only one choice of production

LL(1) grammars cannot be left recursive since the leftmost nonterminal is the same as the LHS. This would result in infinite recursion.

Left factoring is removing the common left factor that appears in two productions of the same non-terminal.

It is done to avoid back-tracing by the parser.

Suppose the parser has a look-ahead consider this example

A -> qB | qC
where A,B,C are non-terminals and q is a sentence. In this case, the parser will be confused as to which of the two productions to choose and it might have to back-trace.
After left factoring, the grammar is converted to

A -> qD

D -> B | C

In this case, a parser with a look-ahead will always choose the right production.

- Recall 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 left-factor the grammar

$E \rightarrow T X$

$E \rightarrow T X$

$X \rightarrow + E \mid \epsilon$

commom prefix

suffixes

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

$Y \rightarrow * T \mid \epsilon$

- Left-factored grammar

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

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

- The LL(1) parsing table:

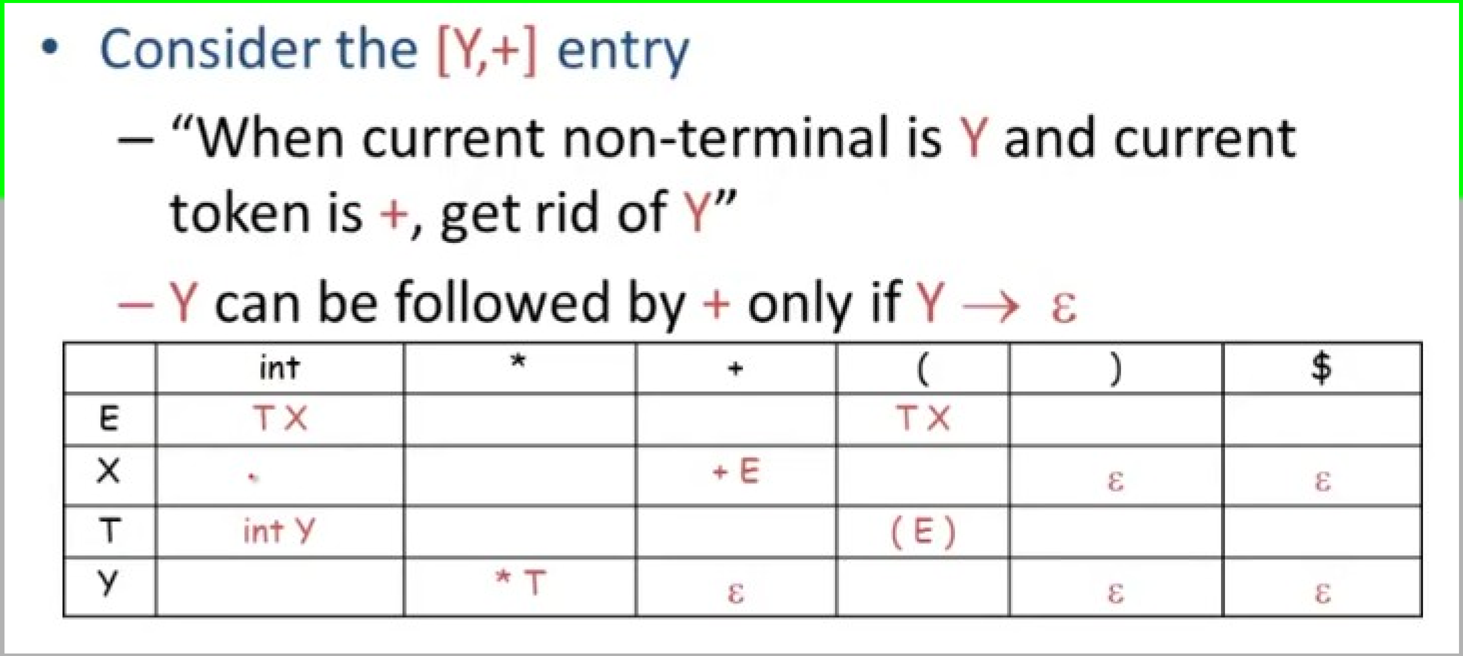
next input token

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

leftmost non-terminal *rhs of production to use*

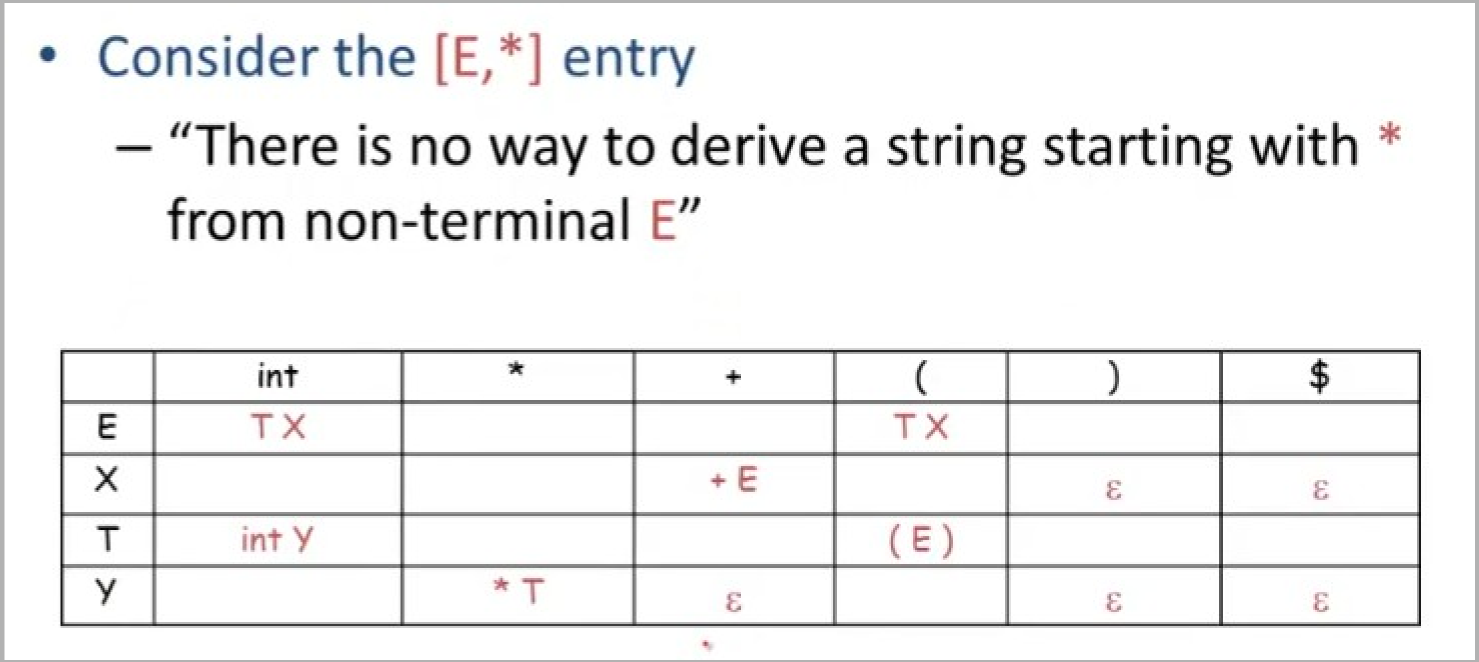
- Consider the [E, int] entry
  - "When current non-terminal is E and next input is int, use production $E \rightarrow T\,X$"

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



- 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 $Y \to \varepsilon$

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



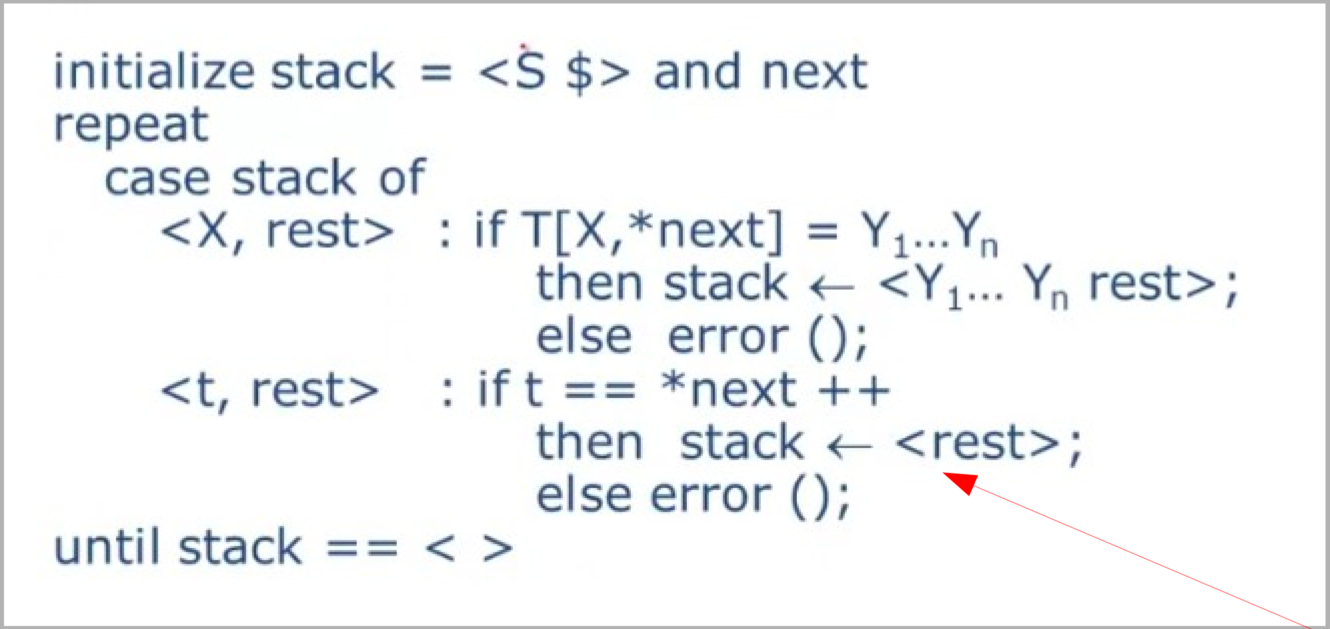
- Consider the [E,*] entry
  - "There is no way to derive a string starting with * from non-terminal E"

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

- Method similar to recursive descent, except
  - For the leftmost non-terminal S
  - We look at the next input token a
  - And choose the production shown at [S,a]

- A stack records frontier of parse tree
  - Non-terminals that have yet to be expanded
  - Terminals that have yet to matched against the input
  - Top of stack = leftmost pending terminal or non-terminal

- Reject on reaching error state
- Accept on end of input & empty stack



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

terminal on top of stack matches input, pop and advance input

- Left-factored grammar

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

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

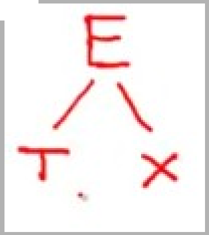
- The LL(1) parsing table:

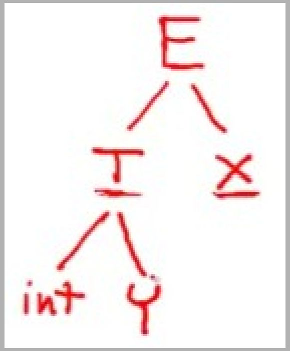
next input token

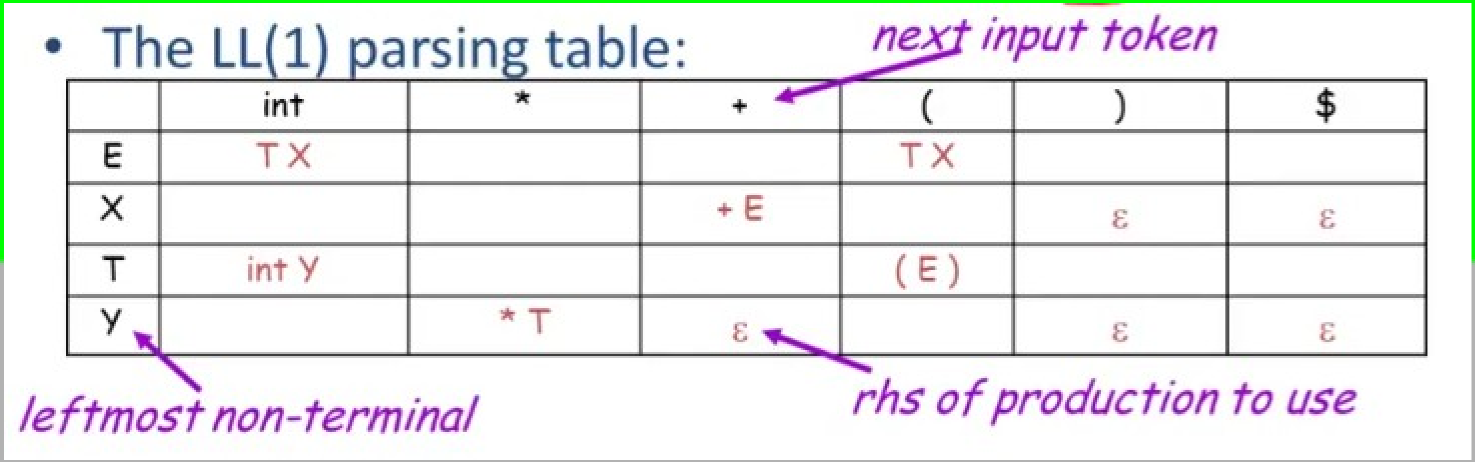
	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

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

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y





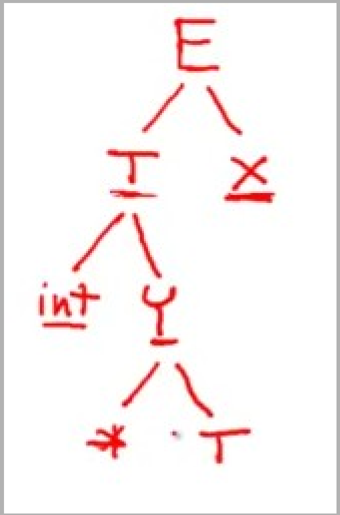


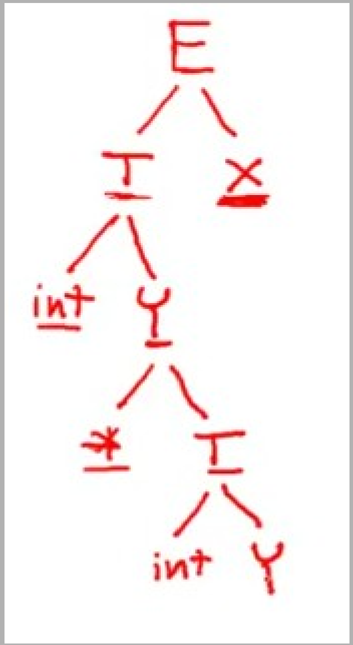
- The LL(1) parsing table:

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

next input token
leftmost non-terminal
rhs of production to use

int Y X $	int * int $	terminal
Y X $	* int $	* T
* T X $	* int $	terminal
T X $	int $	int Y





- Left-factored grammar

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

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

- The LL(1) parsing table:

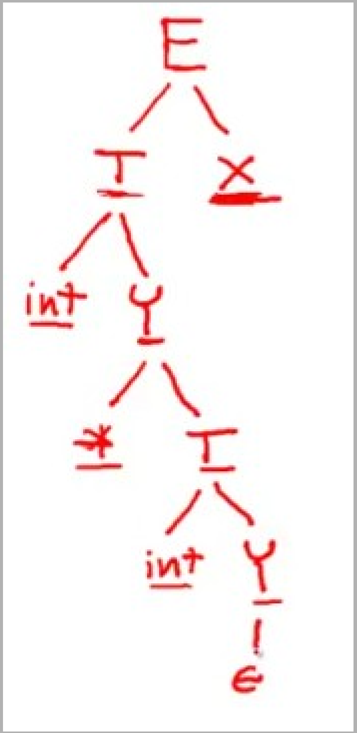
next input token

	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*

int Y X $	int $	terminal
Y X $	$	ε
X $	$	ε

– "When current non-terminal is Y and current token is +, get rid of Y"



- Left-factored grammar

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

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

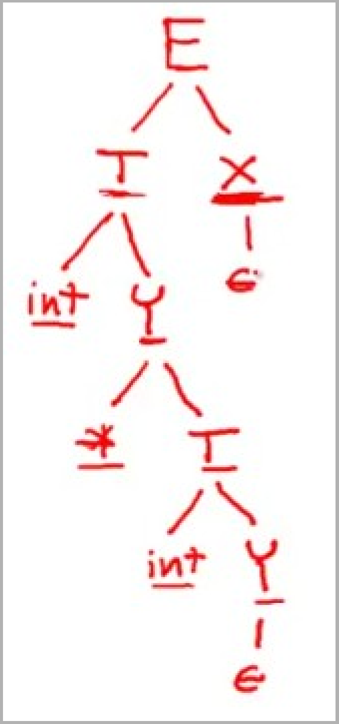
- The LL(1) parsing table:

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

rhs of production to use



X $	$	ε
$	$	ACCEPT