



中山大學  
SUN YAT-SEN UNIVERSITY



国家超级计算广州中心  
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

# Compilation Principle 编译原理

---

## 第6讲：语法分析(3)

张献伟

[xianweiz.github.io](https://xianweiz.github.io)

DCS290, 03/18/2021

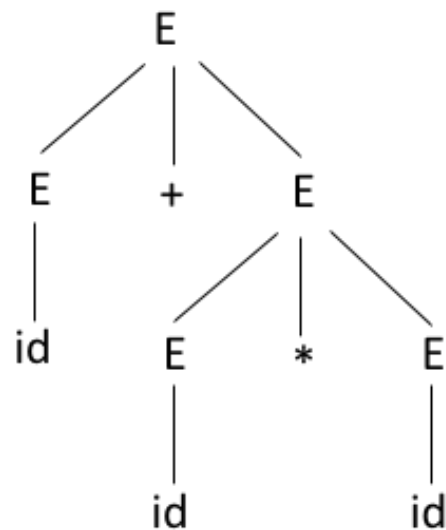


中山大學  
SUN YAT-SEN UNIVERSITY



# Review: Ambiguous Grammar

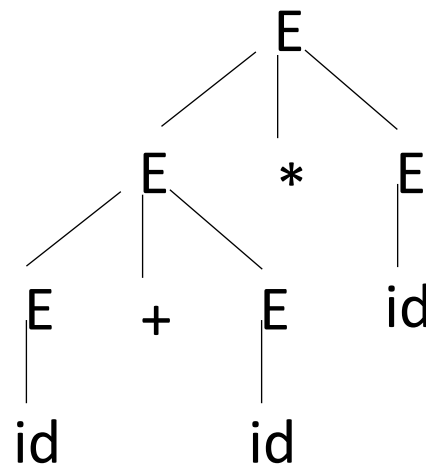
- Grammar  $E \rightarrow E * E \mid E + E \mid (E) \mid id$ 
  - Ambiguous. **Why?**
  - Two distinct leftmost derivations for the sentence  $id + id * id$



- Are the two trees have the same meaning?

- Above:  $id + (id * id)$
- Below:  $(id + id) * id$

- The deepest sub-tree is traversed first, thus higher precedence



# Review: Ambiguity Removal

- How to remove the ambiguity?
- Specify precedence
  - The higher level of the production, the lower priority of operator
  - The lower level of the production, the higher priority of operator
- Specify associativity
  - If the operator is left associative, induce left recursion in its production
  - If the operator is right associative, induce right recursion in its production

$$\begin{array}{l} E \rightarrow E * E \mid E + E \mid (E) \mid \text{id} \end{array} \Rightarrow \begin{array}{l} E \rightarrow E + E \mid T \\ T \rightarrow T * T \mid F \\ F \rightarrow (E) \mid \text{id} \end{array} \Rightarrow \begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid \text{id} \end{array}$$

still possible to get  
 $\text{id} + (\text{id} + \text{id})$   
and  
 $(\text{id} + \text{id}) + \text{id}$   
what if '-' (minus)?

Now, can only have more '+' on left  
E: sum of one or more terms (T)  
T: product of one or more factors (F)  
F: an identifier or a '()'ed expr

# Review: Top-down and Bottom-up

- Consider a CFG grammar G

$S \rightarrow AB$

$A \rightarrow aC$

$B \rightarrow bD$

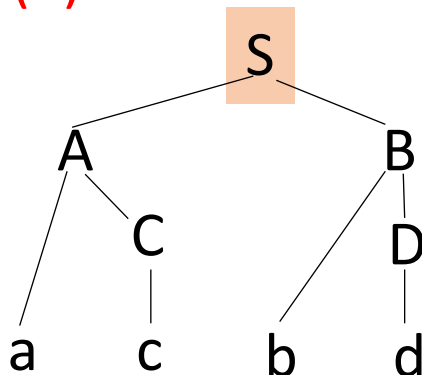
$D \rightarrow d$

$C \rightarrow c$

- This language has only one sentence:  $L(G) = \{acbd\}$

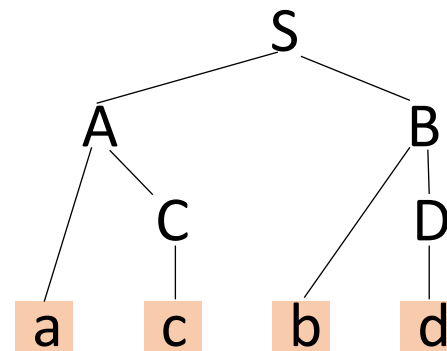
Top-down (Leftmost Derivation)

$S \Rightarrow AB$  (1)  
 $\Rightarrow aCB$  (2)  
 $\Rightarrow acB$  (3)  
 $\Rightarrow acbD$  (4)  
 $\Rightarrow acbd$  (5)



Bottom-up (reverse of rightmost derivation)

$S \Rightarrow AB$  (5)  
 $\Rightarrow AbD$  (4)  
 $\Rightarrow Abd$  (3)  
 $\Rightarrow aCbD$  (2)  
 $\Rightarrow acbd$  (1)



# Preview: Bottom-up Steps

- Consider a CFG grammar G

$S \rightarrow AB$

$A \rightarrow aC$

$B \rightarrow bD$

$D \rightarrow d$

$C \rightarrow c$

| Stack | Input  | Action |
|-------|--------|--------|
| \$    | acbd\$ | Shift  |
| \$a   | cbd\$  | Shift  |
| \$ac  | bd\$   | Reduce |
| \$aC  | bd\$   |        |
| \$A   | bd\$   | Reduce |
| \$Ab  | d\$    | Shift  |
| \$Abd | \$     | Shift  |
| \$AbD | \$     | Reduce |
| \$AB  | \$     | Reduce |
| \$S   | \$     | Reduce |

Bottom-up (reverse of rightmost derivation)

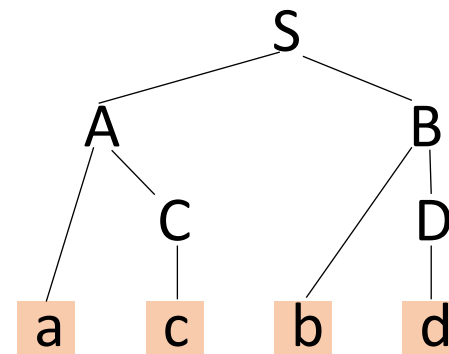
$S \Rightarrow AB$  (5)

$\Rightarrow AbD$  (4)

$\Rightarrow Abd$  (3)

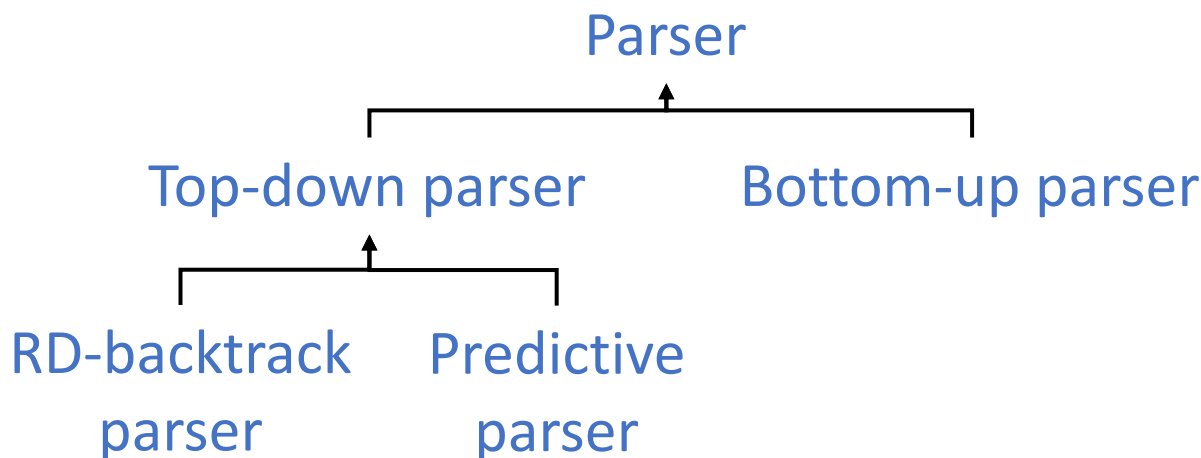
$\Rightarrow aCbD$  (2)

$\Rightarrow acbd$  (1)



# Recursive Descent[递归下降]

- **Recursive descent** is a simple and general parsing strategy
  - Try and backtrack
  - Left-recursion must be eliminated first
    - Can be eliminated automatically using some algorithm
- However it is not popular because of **backtracking**
  - Backtracking requires re-parsing the same string
  - Which is inefficient (can take exponential time)
  - Also undoing semantic actions may be difficult
    - E.g. removing already added nodes in parse tree



# Predictive Parsers[预测分析]

---

- A parser with **no backtracking**: predict correct next production given next input terminal(s)
  - If first terminal of every alternative production is **unique**, then parsing requires no backtracking
  - If not unique, grammar cannot use predictive parsers

$A \rightarrow aBD \mid bBB$

$B \rightarrow c \mid bce$

$D \rightarrow d$

parsing input “**abcd**” requires no backtracking

# Predictive Parsers (cont.)

---

- A predictive parser chooses the production to apply solely on the basis of
    - Next input symbols
    - Current nonterminal being processed
  - Patterns in grammars that prevent predictive parsing
    - **Common prefix**[共同前綴]:  
 $A \rightarrow \alpha\beta \mid \alpha\gamma$   
Given input terminal(s)  $\alpha$ , cannot choose between two rules
    - **Left recursion**[左递归]:  
 $A \rightarrow A\beta \mid \alpha$   
Given input terminal(s)  $\alpha$ , cannot choose between two rules
- What is the language of the grammar?  $\alpha\beta^*$



# Rewrite Grammars for Prediction

- **Left factoring**[左公因子]: removes common left prefix
  - In previous example:  $A \rightarrow \alpha\beta \mid \alpha\gamma$
  - can be changed to
$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta \mid \gamma$$
  - Given input  $\alpha$ ,  $A'$  can choose between  $\beta$  or  $\gamma$   
(Assuming  $\beta$  or  $\gamma$  do not start with  $\alpha$ )
- **Left-recursion removal**: same as for recursive descent
  - In previous example:  $A \rightarrow A\beta \mid \alpha$
  - can be changed to
$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta A' \mid \epsilon$$
  - Given input  $\alpha$ ,  $A'$  can choose between  $\beta$  or  $\epsilon$   
(Assuming  $\beta$  doesn't start with  $\alpha$  or  $A'$  isn't followed by  $\alpha$ )

# LL(k) Parser / Grammar / Language

---

- **LL(k) Parser**

- A predictive parser that uses  $k$  lookahead tokens
- **L**: scans the input from **l**eft to right
- **L**: produces a **l**eftmost derivation
- **k**: using  $k$  input symbols of lookahead at each step to decide

- **LL(k) Grammar**

- A grammar that can be parsed using an LL(k) parser
- $LL(k) \subset CFG$ 
  - Some CFGs are not LL(k): common prefix or left-recursion

- **LL(k) Language**

- A language that can be expressed as an LL(k) grammar

- Many languages are LL(k) ... in fact many are LL(1)!

# LL(k) Parser Implementation

---

- Implemented in a recursive or non-recursive fashion
  - Recursive: recursive descent (recursive function calls)
  - Non-recursive: explicit stack to keep track of recursion
- Recursive LL(1) parser for:  $A \rightarrow B \mid C, B \rightarrow b, C \rightarrow c$ 
  - Parser consists of small functions, one for each non-terminal

```
int A() {  
    int token = peekNext(); // lookahead token  
    switch(token) {  
        case 'b': // 'B' starts with 'b'  
            return B();  
        case 'c': // 'C' starts with 'c'  
            return C();  
        default: // Reject  
            return 0;  
    }  
}
```

# LL(k) Parser Implementation (cont.)

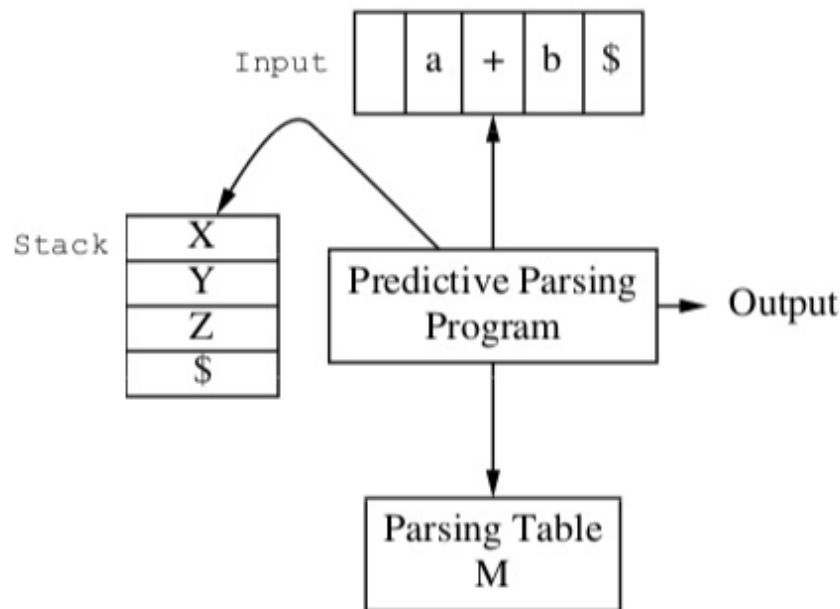
- Recursive LL(1) parser for:  $A \rightarrow B \mid C$ ,  $B \rightarrow b$ ,  $C \rightarrow c$

```
int A() {  
    int token = peekNext(); // lookahead token  
    switch(token) {  
        case 'b': // 'B' starts with 'b'  
            return B();  
        case 'c': // 'C' starts with 'c'  
            return C();  
        default: // Reject  
            return 0;  
    }  
}
```

- Is there a way to express above code more concisely?
  - Non-recursive LL(k) parsers use a **state transition table** (Just like finite automata)
  - Easier to automatically generate a non-recursive parser

# Non-recursive LL(1) Parser

- Table-driven parser: amenable to automatic code generation (just like lexers)
  - **Input buffer**: contains the string to be parsed, followed by \$
  - **Stack**: holds unmatched portion of derivation string
  - **Parse table**  $M[A, b]$ : an entry containing rule “ $A \rightarrow \dots$ ” or error
  - **Parser driver** (a.k.a., predictive parsing program): next action based on (stack top, current token)



# LL(1) Parse Table: Example

| Table | int                            | *                   | +                            | (                   | )                            | \$                           |
|-------|--------------------------------|---------------------|------------------------------|---------------------|------------------------------|------------------------------|
| E     | $E \rightarrow TE'$            |                     |                              | $E \rightarrow TE'$ |                              |                              |
| E'    |                                |                     | $E' \rightarrow +E$          |                     | $E' \rightarrow \varepsilon$ | $E' \rightarrow \varepsilon$ |
| T     | $T \rightarrow \text{int } T'$ |                     |                              | $T \rightarrow (E)$ |                              |                              |
| T'    |                                | $T' \rightarrow *T$ | $T' \rightarrow \varepsilon$ |                     | $T' \rightarrow \varepsilon$ | $T' \rightarrow \varepsilon$ |

- Implementation with 2D parse table
  - **First column** lists all non-terminals in the grammar
  - **First row** lists all possible terminals in the grammar and \$
  - A table entry contains one production
    - One action for each (non-terminal, input) combination
    - It “predicts” the correct action based on one lookahead
    - No backtracking required

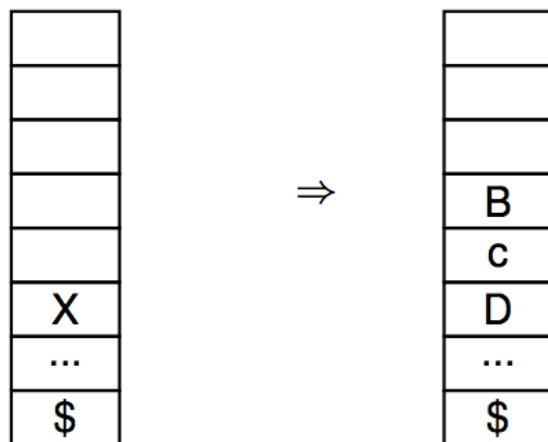
# LL(1) Parsing Algorithm

---

- Initial state
  - **Input** tape: input tokens followed by '\$'
  - **Stack**: start symbol followed by '\$' at bottom
- General idea: repeat one of two actions
  - **Expand** symbol at top of stack by applying a production
  - **Match** terminal symbol at top of stack with input token
- Step-by-step parsing based on  $(X,a)$ 
  - $X$ : symbol at the top of the stack
  - $a$ : current input token
    - If  $X \in T$ , then
      - If  $X == a == \$$ , parser halts with “success”
      - If  $X == a \neq \$$ , successful match, pop  $X$  from stack and advance input head
      - If  $X \neq a$ , parser halts and input is **rejected**
    - if  $X \in N$ , then
      - if  $M[X,a] == 'X \rightarrow \text{RHS}'$ , pop  $X$  and push RHS to stack
      - if  $M[X,a] == \text{empty}$ , parser halts and input is **rejected**

# Push RHS in Reverse Order

- For  $(X, a)$ 
  - $X$ : symbol at the top of the stack
  - $a$ : current input token
- If  $M[X, a] = "X \rightarrow BcD"$



- Performs the leftmost derivation:  $\alpha X \beta \Rightarrow \alpha BcD \beta$ 
  - $\alpha$ : string that has already been matched with input
  - $\beta$ : string yet to be matched, corresponding to the ... above



# Applying LL(1) Parsing to Grammar

---

- Consider the grammar

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

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

- No left recursion
- But require left factoring

- After rewriting grammar, we have

$$E \rightarrow TE'$$

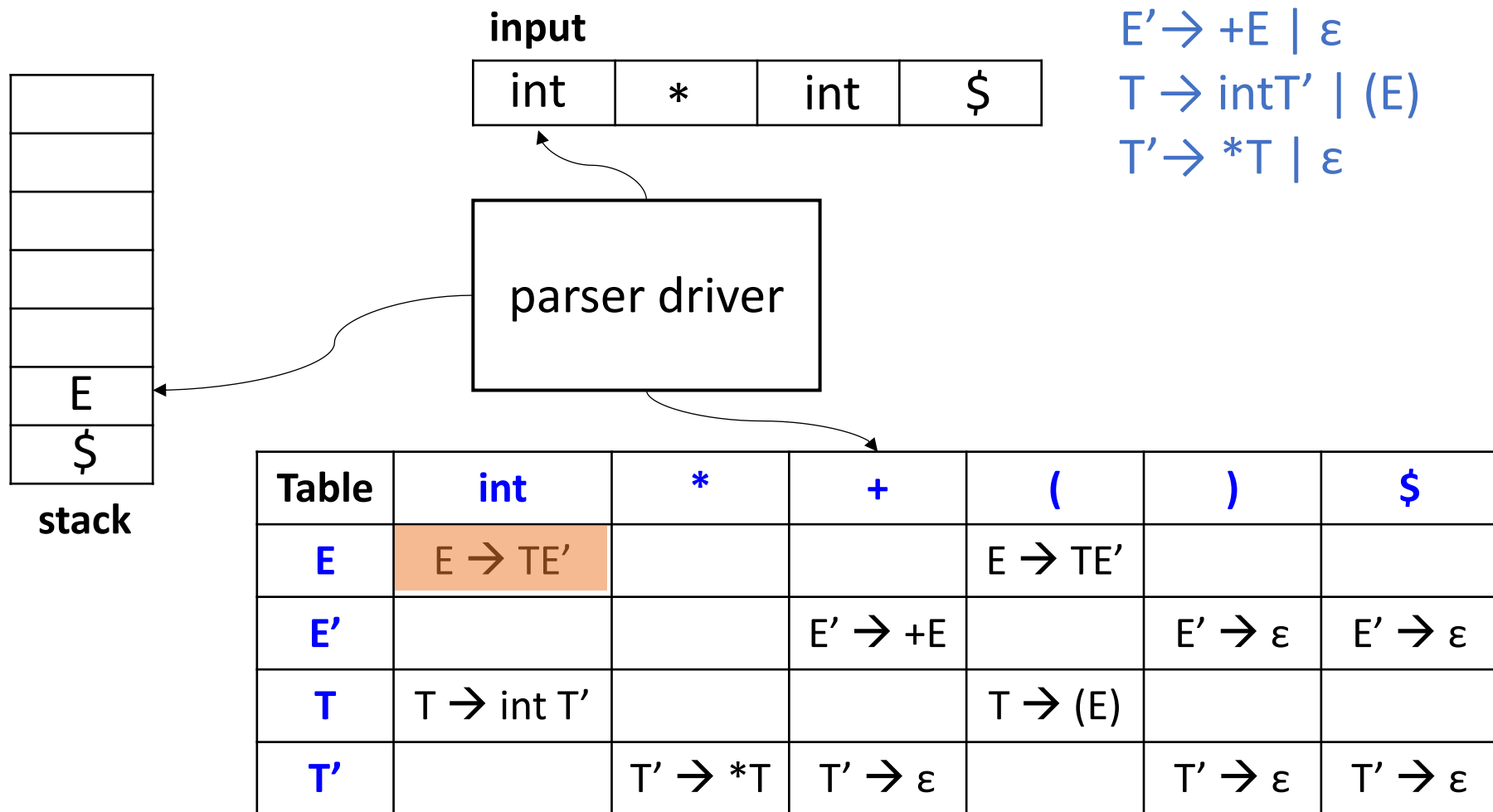
$$E' \rightarrow +E \mid \varepsilon$$

$$T \rightarrow \text{int}T' \mid (E)$$

$$T' \rightarrow *T \mid \varepsilon$$

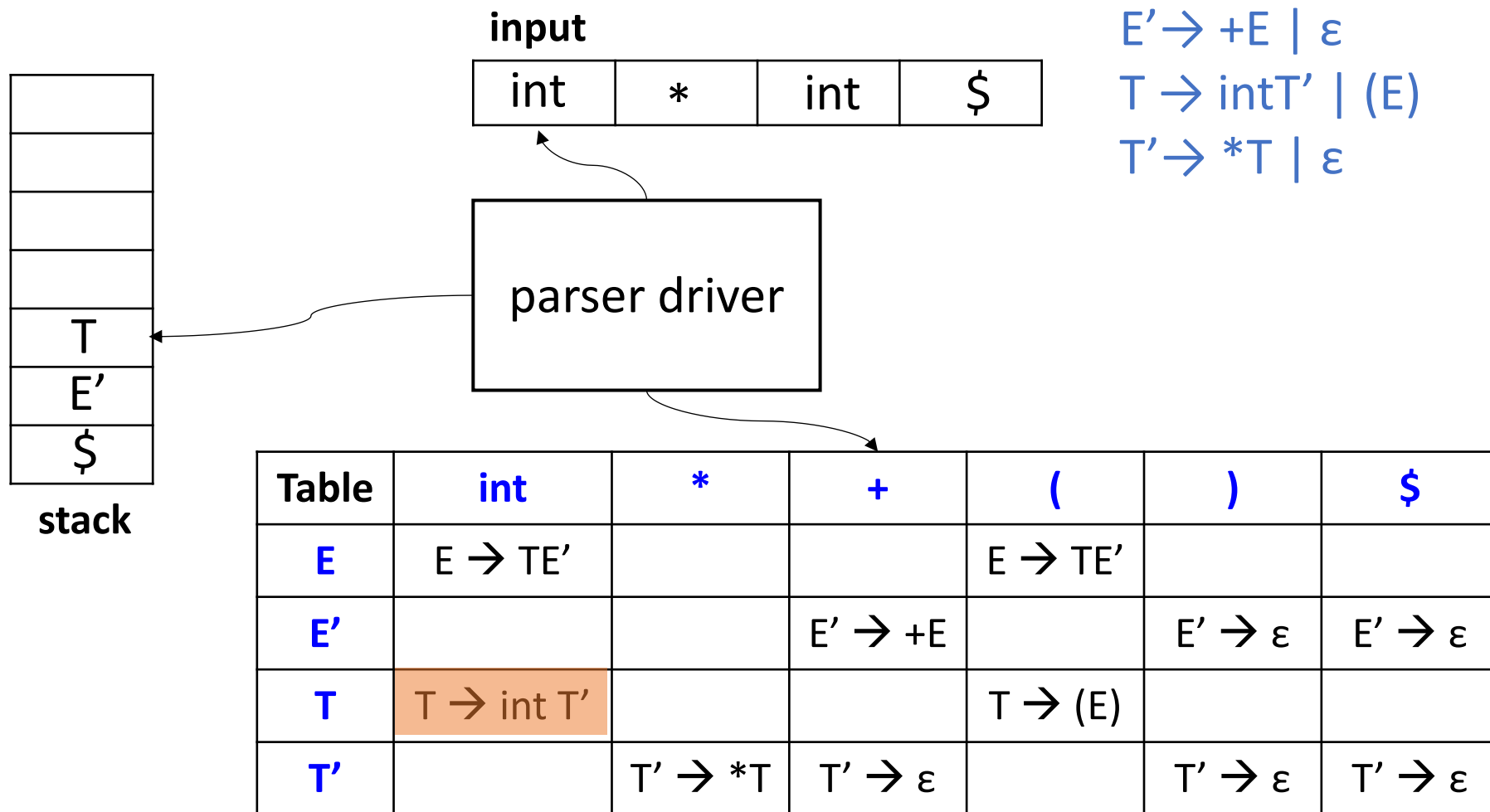
# Using the Parse Table

- To recognize “int \* int”



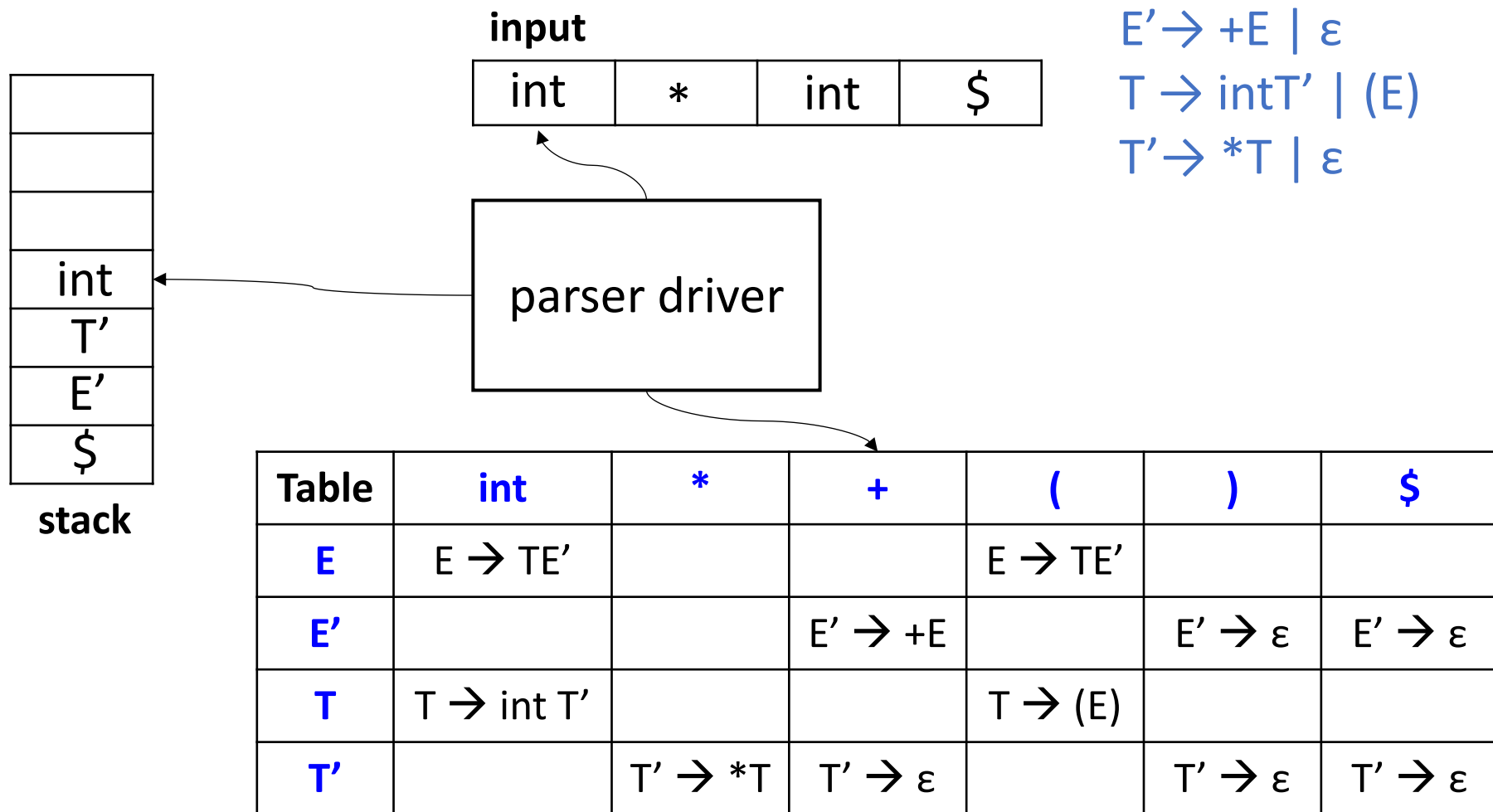
# Using the Parse Table

- To recognize “int \* int”



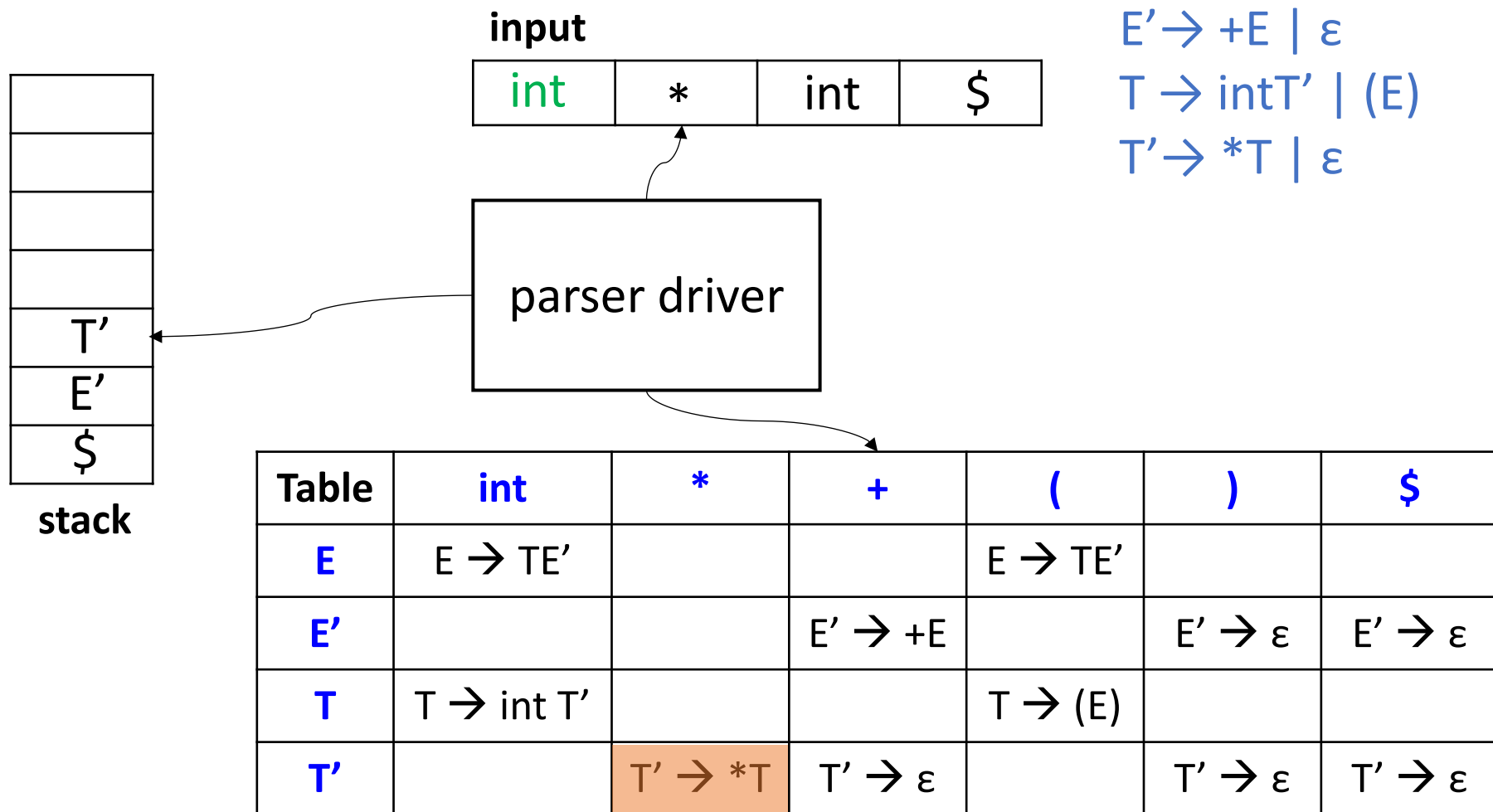
# Using the Parse Table

- To recognize “int \* int”



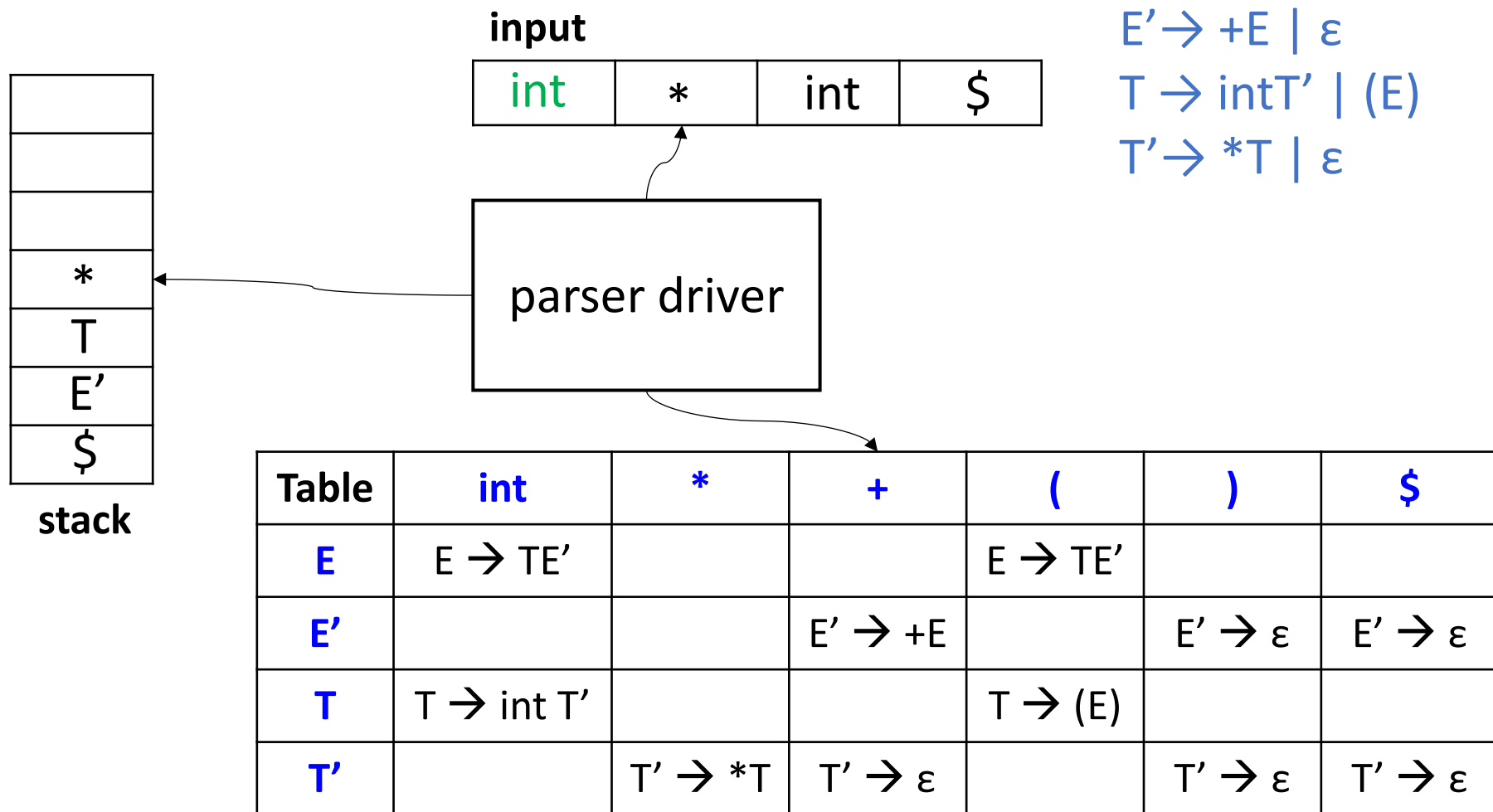
# Using the Parse Table

- To recognize “int \* int”



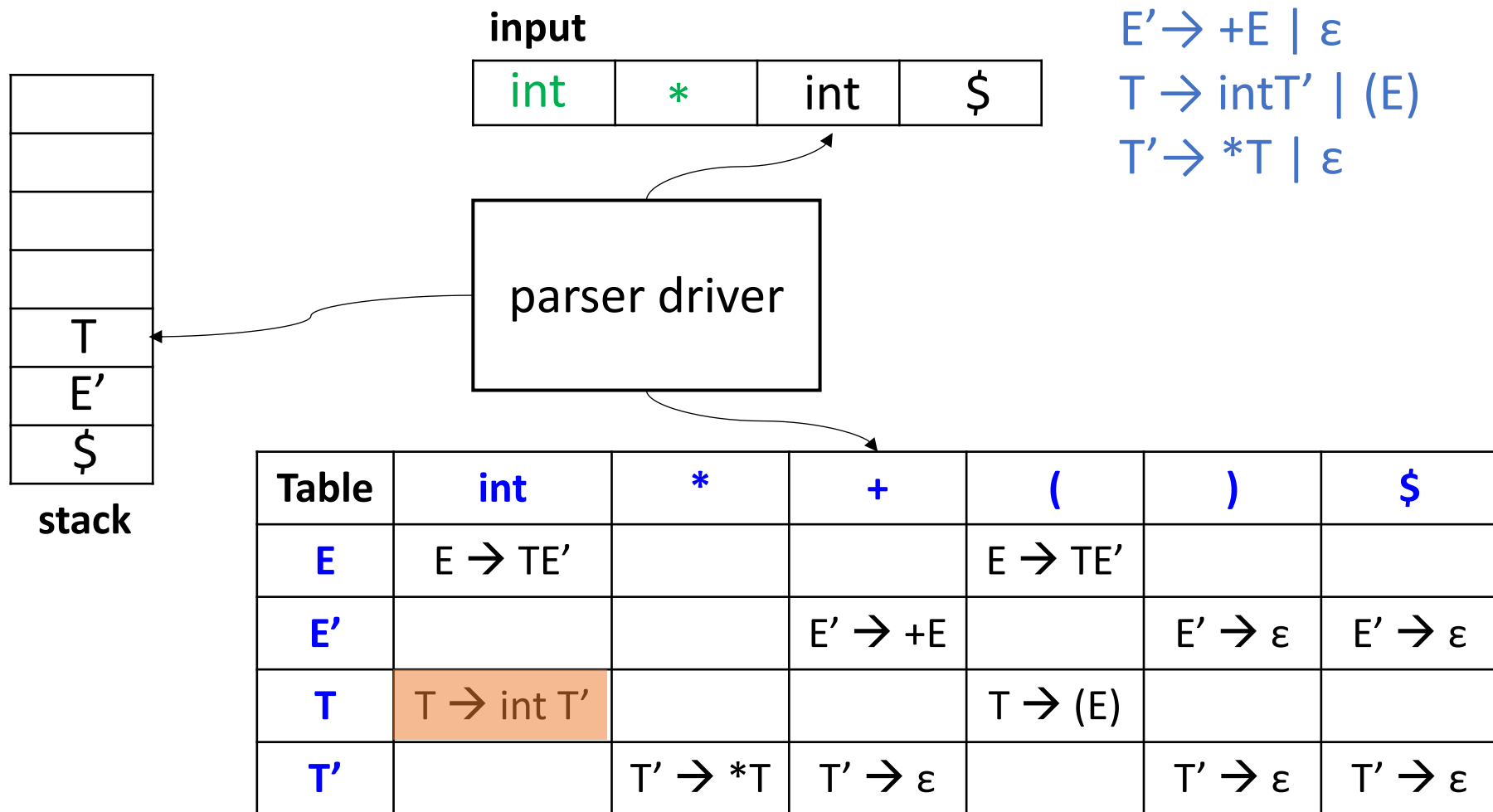
# Using the Parse Table

- To recognize “int \* int”



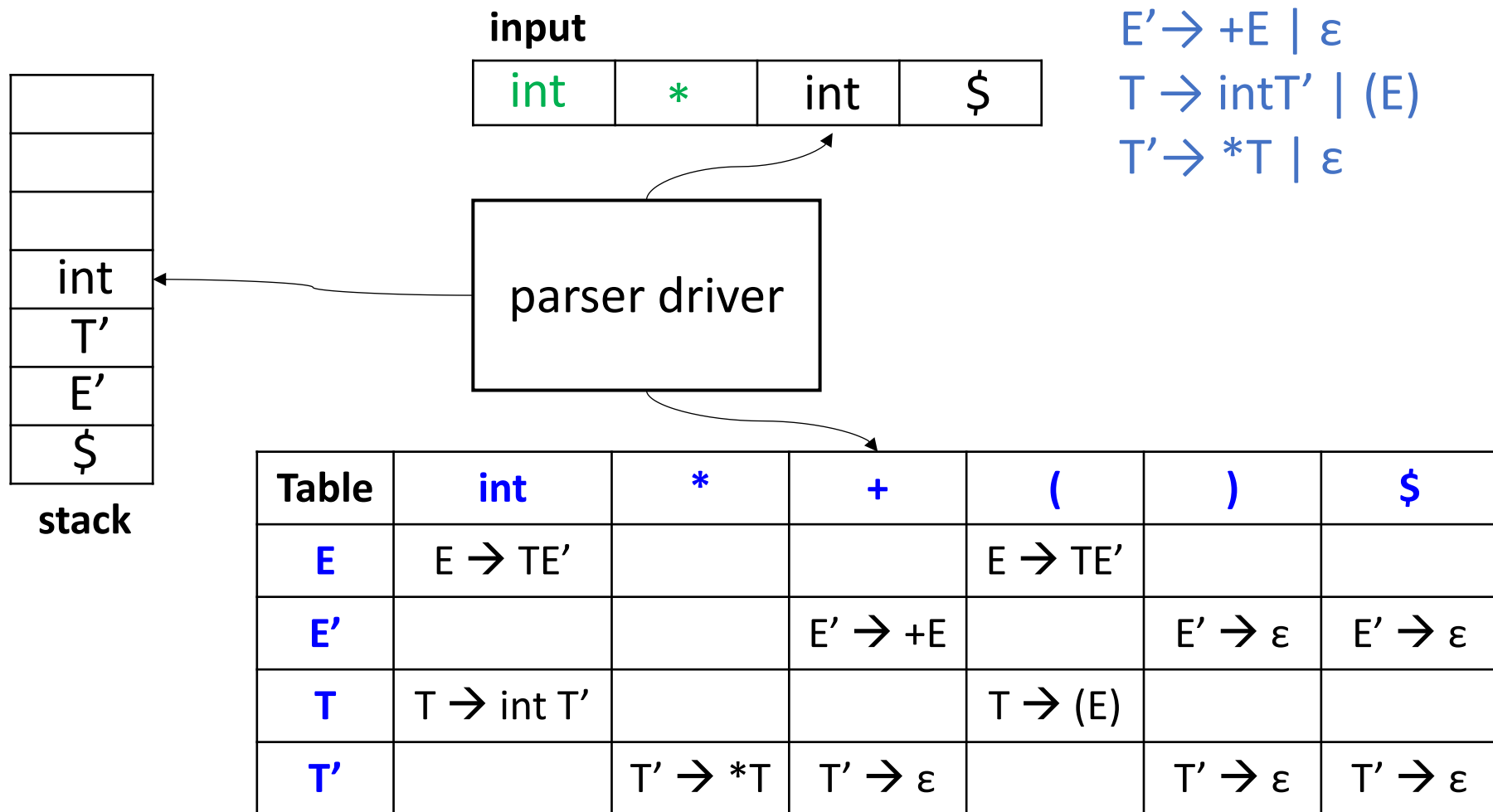
# Using the Parse Table

- To recognize “int \* int”



# Using the Parse Table

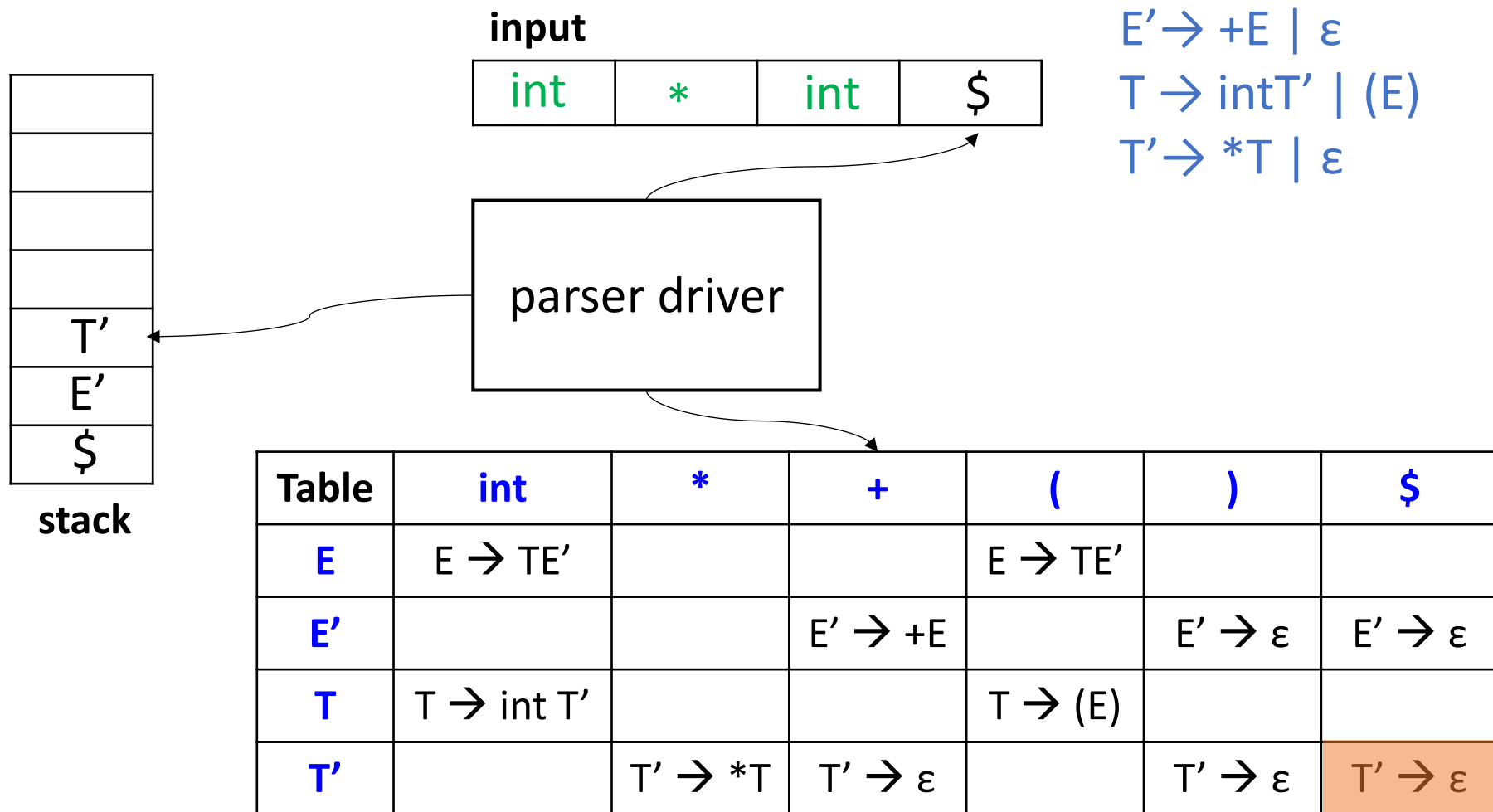
- To recognize “int \* int”





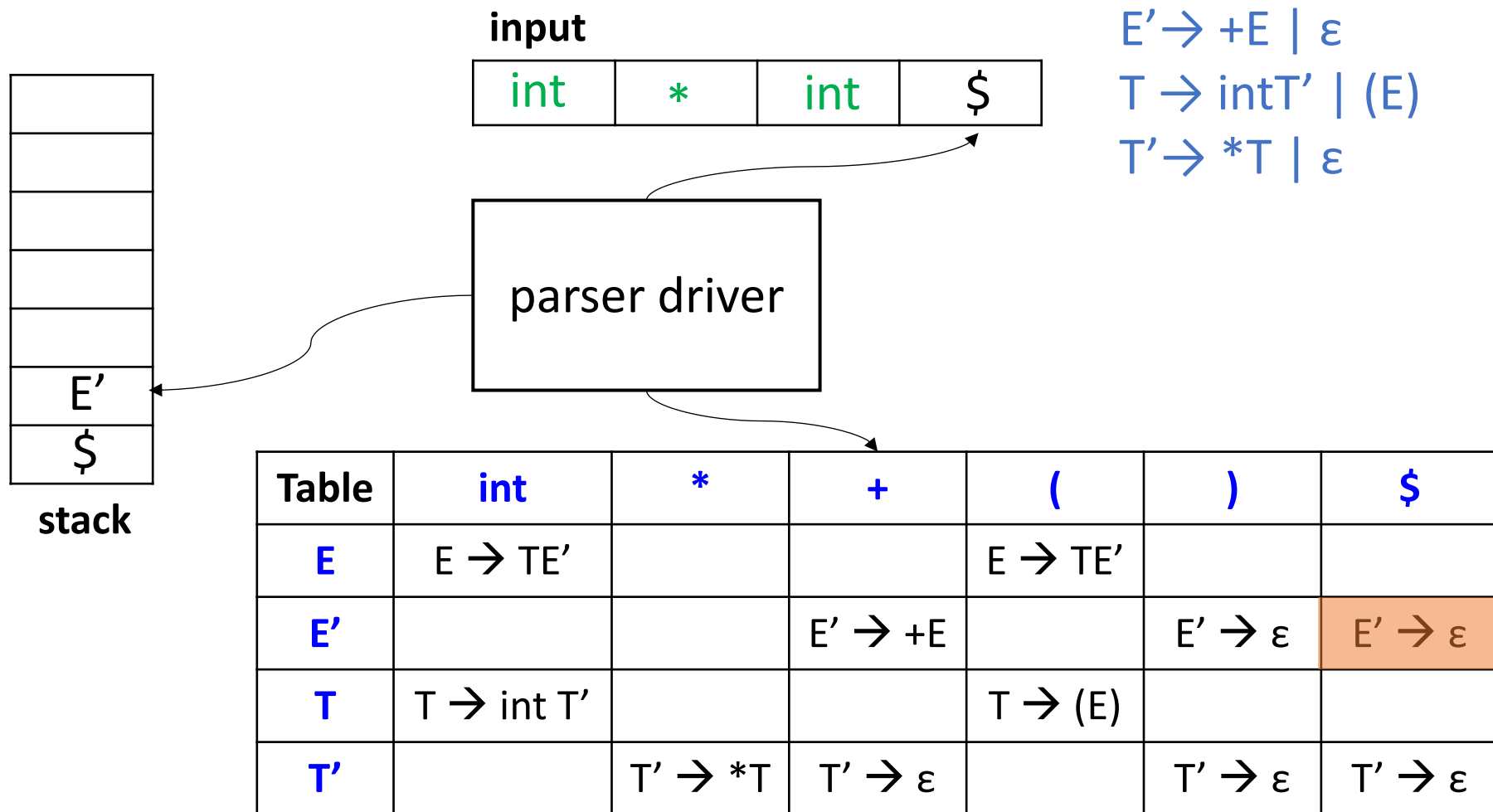
# Using the Parse Table

- To recognize “int \* int”



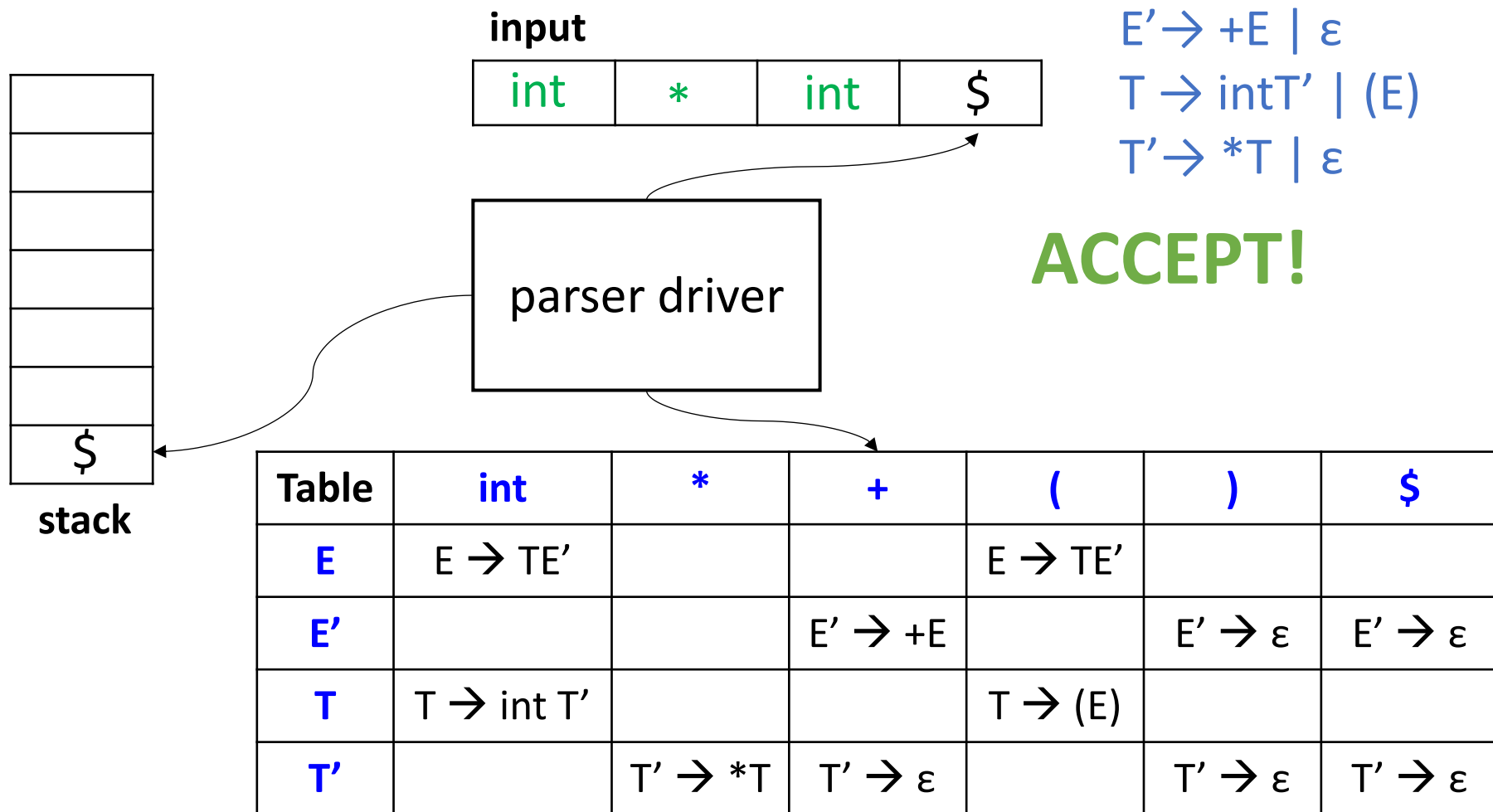
# Using the Parse Table

- To recognize “int \* int”



# Using the Parse Table

- To recognize “int \* int”



# Recognizing Sequence

| Stack        | Input        | Action                         |
|--------------|--------------|--------------------------------|
| E \$         | int * int \$ | $E \rightarrow TE'$            |
| T E' \$      | int * int \$ | $T \rightarrow \text{int } T'$ |
| int T' E' \$ | int * int \$ | match                          |
| T' E' \$     | * int \$     | $T' \rightarrow *T$            |
| * T E' \$    | * int \$     | match                          |
| T E' \$      | int \$       | $T \rightarrow \text{int } T'$ |
| int T' E' \$ | int \$       | match                          |
| T' E' \$     | \$           | $T' \rightarrow \epsilon$      |
| E' \$        | \$           | $E' \rightarrow \epsilon$      |
| \$           | \$           | Halt and accept                |

$E \rightarrow TE'$

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

$T \rightarrow \text{int}T' \mid (E)$

$T' \rightarrow *T \mid \epsilon$

- Contents of stack correspond to remaining input
- Actions correspond to productions in leftmost derivation

# Review Questions (1)

- What is Recursive Descent?

Parsing by trying and backtracking to produce the leftmost derivation

- Why do we prefer to use Predictive Parser?

Requires no backtracking, more efficient

- How to predict the next production to use?

Next input symbol, current nonterminal being processed

- What are the grammar requirements of predictive parse?

No common prefix, no left recursion [唯一性]

- What does LL(k) mean?

L: scans the input from left to right

L: produces a leftmost derivation

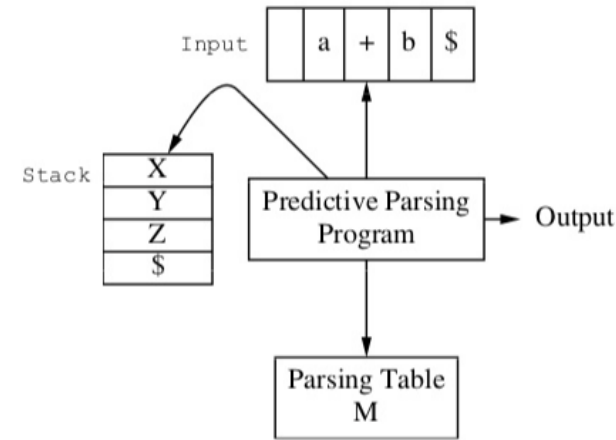
K: using k input symbols of lookahead

# Review Questions (2)

- What is the initial state of the parser?

Input: input tokens followed by \$

Stack: start symbol followed by \$



- General idea of the table-driven parse?

Expand on non-terminal, match on terminal

- How do we expand?

If  $M[X, a] = "X \rightarrow \text{RHS}"$ , pop X and push RHS to stack

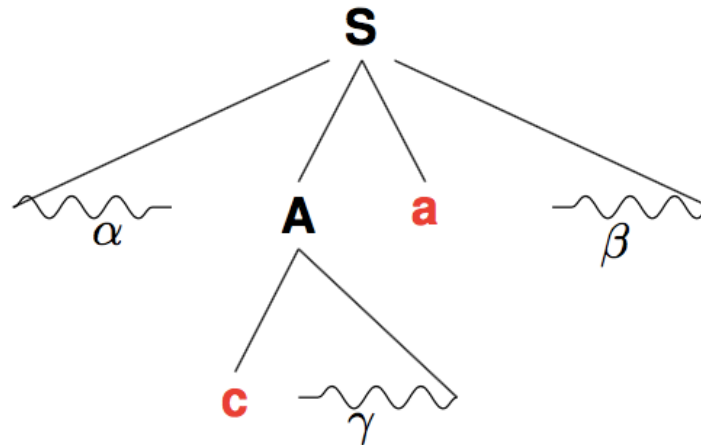
- What are stored in the parsing table?

Actions the parser should take based on input token and stack top

| Table | int                            | *                   | +                         | (                   | )                         | \$                        |
|-------|--------------------------------|---------------------|---------------------------|---------------------|---------------------------|---------------------------|
| E     | $E \rightarrow TE'$            |                     |                           | $E \rightarrow TE'$ |                           |                           |
| E'    |                                |                     | $E' \rightarrow +E$       |                     | $E' \rightarrow \epsilon$ | $E' \rightarrow \epsilon$ |
| T     | $T \rightarrow \text{int } T'$ |                     |                           | $T \rightarrow (E)$ |                           |                           |
| T'    |                                | $T' \rightarrow *T$ | $T' \rightarrow \epsilon$ |                     | $T' \rightarrow \epsilon$ | $T' \rightarrow \epsilon$ |

# To Construct Parsing Table

- The parsing table stores the actions the parser should take based on the input token and the stack top
- The parsing table can be constructed using two sets
  - **FIRST(A)**: set of terminals that begin strings derived from A
    - E.g.,  $c \in \text{FIRST}(A)$
    - If  $A \Rightarrow^* \epsilon$ , then  $\epsilon$  is also in  $\text{FIRST}(A)$
  - **FOLLOW(A)**: set of terminals that can appear following A
    - E.g.,  $a \in \text{FOLLOW}(A)$
    - If A is rightmost of a sentential form, then  $\$$  is also in  $\text{FOLLOW}(A)$



# Use FIRST and FOLLOW

---

- Why do we need FIRST and FOLLOW in parsing?
- FIRST[开始集]
  - $\text{FIRST}(\alpha)$ : set of terminals that start strings derived from  $\alpha$
  - Consider  $A \rightarrow \alpha|\beta$ , where  $\text{FIRST}(\alpha)$  and  $\text{FIRST}(\beta)$  are disjoint sets
  - We can then choose by looking at the next input symbol  $a$ 
    - since  $a$  can be in at most  $\text{FIRST}(\alpha)$  or  $\text{FIRST}(\beta)$ , not both
- FOLLOW[后继集]
  - $\text{FOLLOW}(A)$ : set of terminals that can appear right after  $A$
  - If there's a derivation of  $A$  that results in  $\varepsilon$ 
    - In this case,  $A$  could be replaced by nothing and the next token would be the first token of the symbol following  $A$  in the sentence being parsed
    - Thus, parser needs to consider to choose the path  $A \Rightarrow^* \varepsilon$



# Example

Grammar:

$S \rightarrow aBC$

$B \rightarrow bC$   $b \in \text{FIRST}(B)$

$B \rightarrow dB$   $d \in \text{FIRST}(B)$

$B \rightarrow \epsilon$

$C \rightarrow c$   $c \in \text{FOLLOW}(B)$

$C \rightarrow a$   $a \in \text{FOLLOW}(B)$

$D \rightarrow e$

Input:  $ada$

$S$   $\uparrow \uparrow \uparrow$   
 $\Rightarrow aBC$   
 $\Rightarrow adBC$   
 $\Rightarrow adC$   
 $\Rightarrow ada$

Input:  $ade$

$S$   
 $\Rightarrow aBC$   
 $\Rightarrow adBC$   
 $\Rightarrow adC$   ~~$X$~~   
 $\Rightarrow$

- Both FIRST and FOLLOW should be used to construct the parsing table

# FIRST

---

- Compute  $\text{FIRST}(X)$  for all grammar symbols  $X$ , apply the following rules until no terminal or  $\epsilon$  can be added to any FIRST set
  - If  $X \in T$ , then  $\text{FIRST}(X) = \{X\}$  [终结符]
  - If  $X \in N$  and  $X \rightarrow \epsilon$  exists, then add  $\epsilon$  to  $\text{FIRST}(X)$  [非终结符, 空式]
  - If  $X \in N$  and  $X \rightarrow Y_1 Y_2 Y_3 \dots Y_k$ , then
    - Add  $a$  to  $\text{FIRST}(X)$ , if for some  $i$ ,  $a$  is in  $\text{FIRST}(Y_i)$ , and  $\epsilon$  is in all of  $\text{FIRST}(Y_1), \dots, \text{FIRST}(Y_{i-1})$ , i.e.,  $Y_1 \dots Y_{i-1} \Rightarrow^* \epsilon$ . E.g.,
      - Everything in  $\text{FIRST}(Y_1)$  is surely in  $\text{FIRST}(X)$
      - If  $Y_1$  doesn't derive  $\epsilon$ , then we add nothing more
      - But if  $Y_1 \Rightarrow^* \epsilon$ , then we add  $\text{FIRST}(Y_2)$ , and so on
    - Add  $\epsilon$  to  $\text{FIRST}(X)$ , if  $\epsilon$  is in  $\text{FIRST}(Y_j)$  for all  $j=1, 2, \dots, k$

# FIRST(cont.)

---

- Compute FIRST( $X$ ) for all grammar symbols  $X$  [符号]
- Next, we can compute FIRST for any string  $\alpha = X_1X_2...X_n$  [字符串]
  - Add FIRST( $X_1$ ) all non- $\epsilon$  symbols to FIRST( $\alpha$ )
  - Add FIRST( $X_i$ ) –  $\epsilon$ ),  $2 \leq i \leq k$ , to FIRST( $\alpha$ ), if FIRST( $X_1$ ), ..., FIRST( $X_{k-1}$ ) all contain  $\epsilon$ 
    - Add non- $\epsilon$  symbols of FIRST( $X_2$ ), if  $\epsilon$  is in FIRST( $X_1$ )
    - Add non- $\epsilon$  symbols of FIRST( $X_3$ ), if  $\epsilon$  is in FIRST( $X_1$ ) and FIRST( $X_2$ )
    - ...
  - Add  $\epsilon$  to FIRST( $\alpha$ ), if FIRST( $X_1$ ), ..., FIRST( $X_k$ ) all contain  $\epsilon$

# FOLLOW

---

- To compute FOLLOW(A) to all non-terminals A, apply following rules until no terminal or  $\epsilon$  can be added to any FOLLOW set
  - Place  $\$$  in FOLLOW(S), where S is the start symbol
  - If there is a production  $A \rightarrow \alpha B \beta$ , then everything in FIRST( $\beta$ ) except  $\epsilon$  is in FOLLOW(B)
  - If there is a production  $A \rightarrow \alpha B$ , or a production  $A \rightarrow \alpha B \beta$ , where FIRST( $\beta$ ) contains  $\epsilon$ , then everything in FOLLOW(A) is in FOLLOW(B)

# Example: FIRST and FOLLOW

- $\text{FIRST}(T) = \text{FIRST}(E) = \{\text{int}, (\}$ 
  - E has only one production, and its body starts with T
  - T doesn't derive  $\epsilon$ , E is same with T
- $\text{FIRST}(E') = \{+, \epsilon\}$
- $\text{FIRST}(T') = \{*, \epsilon\}$
- $\text{FOLLOW}(E) = \text{FOLLOW}(E') = \{), \$\}$ 
  - E is start symbol, thus \$ must be contained; production body (E)
  - E' appears at the ends of E-productions, same as  $\text{FOLLOW}(E)$
- $\text{FOLLOW}(T) = \text{FOLLOW}(T') = \{+, ), \$\}$ 
  - +: T appears in bodies only followed by E', thus  $\text{FIRST}(E') - \epsilon$
  - ), \$:  $\text{FIRST}(E')$  contains  $\epsilon$ , and E' is the entire str following T, so  $\text{FOLLOW}(E')$  is in  $\text{FOLLOW}(T)$
  - T' is only at ends of T-productions,  $\text{FOLLOW}(T') = \text{FOLLOW}(T)$

$$E \rightarrow TE'$$

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

$$T \rightarrow \text{int}T' \mid (E)$$

$$T' \rightarrow *T \mid \epsilon$$

# Example: FIRST and FOLLOW (cont.)

| Symbol | FIRST  | FOLLOW   |
|--------|--------|----------|
| E      | int, ( | ), \$    |
| E'     | +, ε   | ), \$    |
| T      | int, ( | +, ), \$ |
| T'     | *, ε   | +, ), \$ |

$E \rightarrow TE'$   
 $E' \rightarrow +E \mid \varepsilon$   
 $T \rightarrow intT' \mid (E)$   
 $T' \rightarrow *T \mid \varepsilon$

| $A \rightarrow \alpha$ (RHS) | FIRST  |
|------------------------------|--------|
| $E \rightarrow TE'$          | int, ( |
| $E' \rightarrow +E$          | +      |
| $T \rightarrow intT'$        | int    |
| $T \rightarrow (+E)$         | (      |
| $T' \rightarrow *T$          | *      |

# Construct LL(1) Parse Table

---

- To construct, rule  $A \rightarrow \alpha$  is added to  $M[A, a]$  if either:
  - For each terminal  $a$  in  $\text{FIRST}(\alpha)$
  - If  $\epsilon$  is in  $\text{FIRST}(\alpha)$ , or  $\alpha = \epsilon$ ,  $a$  is in  $\text{FOLLOW}(A)$  (Epsilon production)
- If  $\epsilon$  is in  $\text{FIRST}(\alpha)$  and  $\$$  is in  $\text{FOLLOW}(A)$ , add  $A \rightarrow \alpha$  to  $M[A, \$]$  as well
- If after performing the above, there is no production at all in  $M[A, a]$ , then set  $M[A, a]$  to error
  - Which is normally represented by an empty entry in the table

# Construct LL(1) Parse Table (cont.)

| $A \rightarrow \alpha$ (RHS) | FIRST  |
|------------------------------|--------|
| $E \rightarrow TE'$          | int, ( |
| $E' \rightarrow +E$          | +      |
| $T \rightarrow intT'$        | int    |
| $T \rightarrow (E)$          | (      |
| $T' \rightarrow *T$          | *      |
| $E' \rightarrow \epsilon$    | FOLLOW |
| $T' \rightarrow \epsilon$    | FOLLOW |

| Symbol | FIRST         | FOLLOW   |
|--------|---------------|----------|
| E      | int, (        | ), \$    |
| E'     | +, $\epsilon$ | ), \$    |
| T      | int, (        | +, ), \$ |
| T'     | *, $\epsilon$ | +, ), \$ |

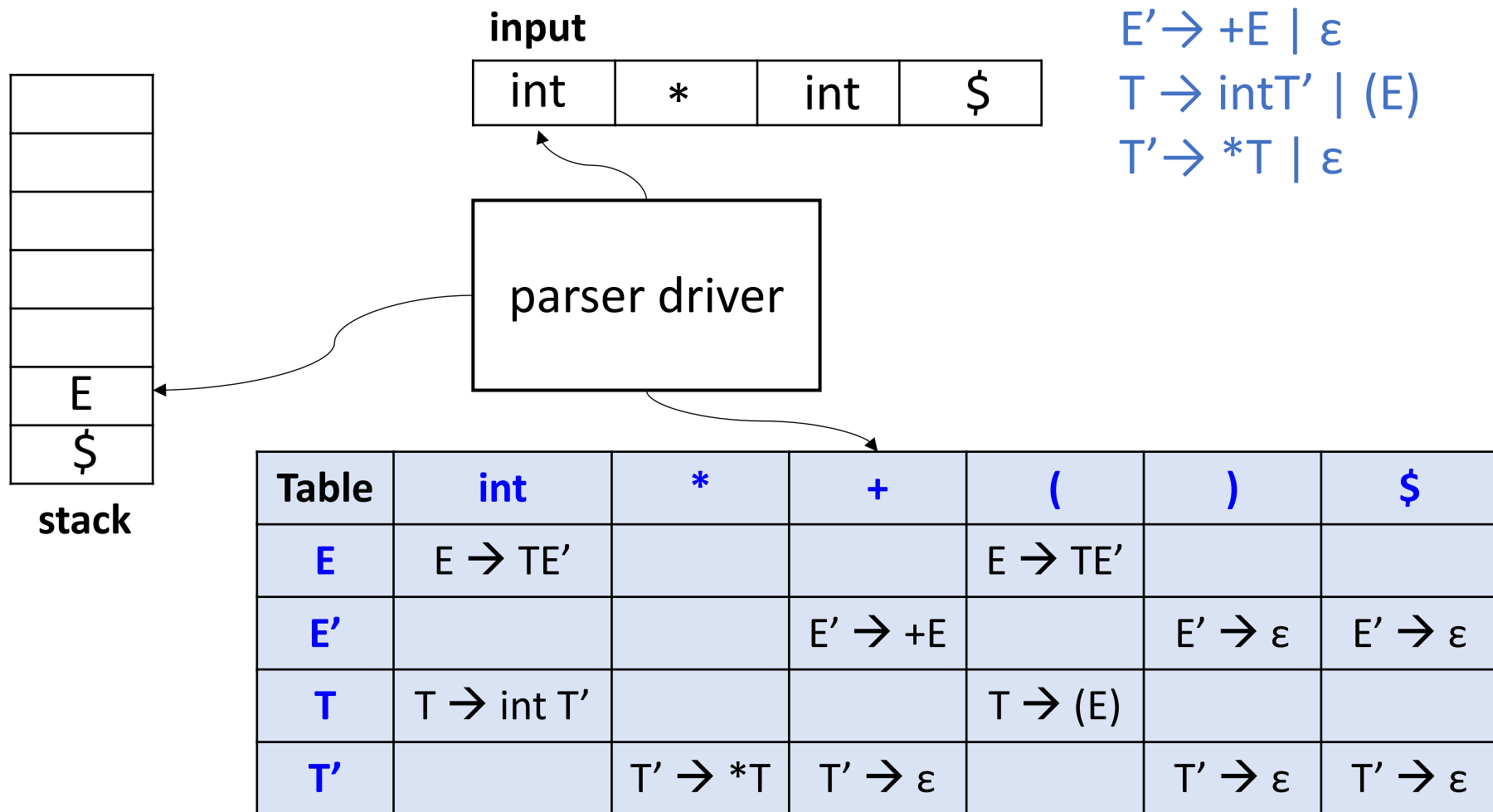
$E \rightarrow TE'$   
 $E' \rightarrow +E | \epsilon$   
 $T \rightarrow intT' | (E)$   
 $T' \rightarrow *T | \epsilon$

| Table | int                    | *                   | +                         | (                   | )                         | \$                        |
|-------|------------------------|---------------------|---------------------------|---------------------|---------------------------|---------------------------|
| E     | $E \rightarrow TE'$    |                     |                           | $E \rightarrow TE'$ |                           |                           |
| E'    |                        |                     | $E' \rightarrow +E$       |                     | $E' \rightarrow \epsilon$ | $E' \rightarrow \epsilon$ |
| T     | $T \rightarrow int T'$ |                     |                           | $T \rightarrow (E)$ |                           |                           |
| T'    |                        | $T' \rightarrow *T$ | $T' \rightarrow \epsilon$ |                     | $T' \rightarrow \epsilon$ | $T' \rightarrow \epsilon$ |



# Use the Table [already examined]

- To recognize “int \* int”



# Determine if Grammar is LL(1)

- Observation

- If a grammar is LL(1), then each of its LL(1) table entry contains **at most one rule**
- Otherwise, it is not LL(1).

- Two methods to determine if a grammar is LL(1) or not

- Construct LL(1) table, and check if there is a multi-rule entry
- Checking each rule as if the table is getting constructed.

G is LL(1) **iff** for a rule  $A \rightarrow \alpha | \beta$

- $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \phi$
- At most one of  $\alpha$  and  $\beta$  can derive  $\epsilon$
- If  $\beta$  derives  $\epsilon$ , then  $\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \phi$

} 保证预测的唯一性

# Non-LL(1) Grammars

---

- Suppose a grammar is not LL(1). What then?
- Case-1: the language may still be LL(1).
  - Try to **rewrite grammar** to LL(1) grammar:
    - Apply left-factoring
    - Apply left-recursion removal
  - Try to **remove ambiguity** in grammar:
    - Encode precedence into rules
    - Encode associativity into rules
- Case-2: If Case-1 fails, language may not be LL(1)
  - Impossible to resolve conflict at the grammar level
  - Programmer chooses which rule to use for conflicting entry (if choosing that rule is always semantically correct)
  - Otherwise, use a more powerful parser (e.g. LL(k), LR(1))

# LL(1) Time and Space Complexity

---

- **Linear** time and space relative to length of input
- **Time**: each input symbol is consumed within a constant number of steps
  - If symbol at top of stack is a terminal:
    - Matched immediately in one step
  - If symbol at top of stack is a non-terminal:
    - Matched in at most  $N$  steps, where  $N$  = number of rules
    - Since no left-recursion, cannot apply same rule twice without consuming input
- **Space**: smaller than input (after removing  $X \rightarrow \epsilon$ )
  - RHS is always longer or equal to LHS
    - Derivation string expands monotonically
    - Derivation string is always shorter than final input string
  - Stack is a subset of derivation string (unmatched portion)

# Some Thoughts ...

---

- LL(1) table-driven parser is basically DFA + Stack
  - Capable to count  $\Rightarrow$  CFG is more powerful than RE
- We have studied LL(1), what about LL(0), LL(2) or LL(k)?
- Is **LL(0)** useful at all?
  - Grammar where rules can be **predicted with no lookahead**
  - $\Rightarrow$  That means, there can only be one rule per non-terminal
  - $\Rightarrow$  That means, this language can have only one string
- What would prevent LL(2) ... LL(k) from wide usage?
  - Size of parse table =  $O(|N| * |T|^k)$ 
    - where  $N$  = set of non-terminals,  $T$  = set of terminals

# Summary: Predictive Parser

---

- **FIRST** and **FOLLOW** sets are used to construct **predictive parsing tables**
- Intuitively, **FIRST** and **FOLLOW** sets guide the choice of rules
  - For non-terminal  $A$  and lookahead  $t$ , use the production rule  $A \rightarrow \alpha$  where  $t \in \text{FIRST}(\alpha)$   
OR
  - For non-terminal  $A$  and lookahead  $t$ , use the production rule  $A \rightarrow \alpha$  where  $\epsilon \in \text{FIRST}(\alpha)$  and  $t \in \text{FOLLOW}(A)$
  - There can only be ONE such rule
    - Otherwise, the grammar is not LL(1)