中山大學 SUN YAT-SEN UNIVERSITY 计算机学院（软件学院） SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

# Compilation Principle
# 编 译 原 理

## 第8讲：语法分析(4)


张献伟

xianweiz.github.io


DCS290, 3/26/2024

# Review Questions

- How does parse tree relate to derivation?

  Parse tree is a graphical repr. of derivation. No orders kept.

- Regard AST tree build, how to classify parser?

  Top-down (root to leaves), bottom-up (leaves to root).

- Explain top-down parsing?

  Build parse tree in a top-down fashion;
  from start symbol to input string, i.e., leftmost derivation.

- How does recursive-descent with backtracking work?

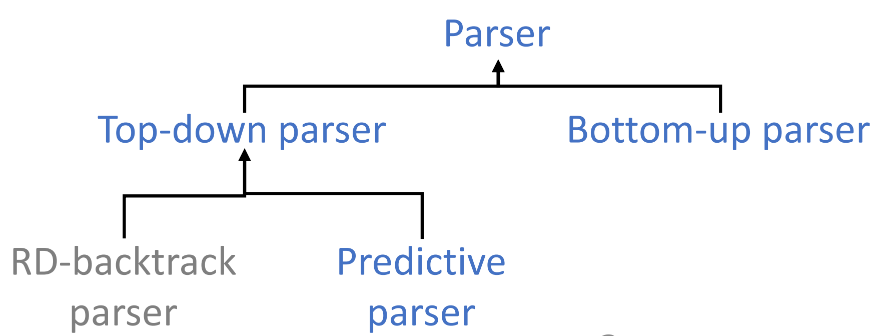
  NT: expand, T: compare; trial and error, backtrack when wrong

- Can $S \rightarrow Ta \mid b$, $T \rightarrow Sd$ be parsed with RD-backtrack?

  No. There exists left recursion.

# Summary of RD-backtrack[小结]

- **RD-backtrack** is a simple and general parsing strategy
  - Left-recursion must be eliminated first
    - Can be eliminated automatically using some algorithm
  - L(Recursive_descent) ≡ L(CFG) ≡ CFL
- 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]

- In recursive descent with backtracking[有回溯]:
  - At each step, many choices of production to use
  - Backtracking used to undo bad choices
- A parser with no backtracking[无回溯]: **predict** correct next production given next input terminal(s)?[以下面一些输入来预测]
  - If first terminal of every alternative production is **unique**, then parsing requires no backtracking[候选产生式首符号唯一]
  - If not unique, grammar cannot use predictive parsers[不唯一]

> A→aBD | bBB
> B→c | bce
> D→d
>
> parsing input “abced” requires no backtracking

?: 如果只往前看一个，那么next terminal其实就是current terminal，即准备匹配的那个（注意backtrack是完全不看）

# Predictive Parsers (cont.)

- A predictive parser chooses the production to apply solely on the basis of[选取产生式的依据]
  - Next input symbol(s)[下一输入符号/终结符；怎么预测？]
  - Current nonterminal being processed[当前非终结符；为谁预测？]
- Patterns in grammars that prevent predictive parsing[并非总是能预测分析]
  - **Common prefix**[共同前缀]: $S \rightarrow cAd$ $A \rightarrow ab \mid a$

    $A \rightarrow \alpha\beta \mid \alpha\gamma$

    Given input terminal(s) α, cannot choose between two rules
  - **Left recursion**[左递归]: $A \rightarrow Ab \mid a$

    $A \rightarrow A\beta \mid \alpha$ 从不匹配（一直展开） input: abbbb

    Lookahead symbol changes only when a terminal is matched

# Rewrite Grammars for Prediction[改写]

- **Left factoring**[左公因子提取]: removes common left prefix
  - In previous example: $A \rightarrow \alpha\beta \mid \alpha\gamma$
  - can be changed to
    - $A \rightarrow \alpha A'$
    - $A' \rightarrow \beta \mid \gamma$
  - After processing $\alpha$, $A'$ can choose between $\beta$ or $\gamma$

  (assuming $\beta$ or $\gamma$ do not start with $\alpha$)

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

☞ *stmt* → **if** *expr* **then** *stmt S'*

*S'* → **else** *stmt* | ε

☞ 推迟选择，直到可区分

- **Left-recursion removal**[左递归消除]: same as recursive descent
  - In previous example: $A \rightarrow A\beta \mid \alpha$
  - can be changed to
    - $A \rightarrow \alpha A'$
    - $A' \rightarrow \beta A' \mid \varepsilon$
  - After processing $\alpha$, $A'$ can choose between $\beta$ or $\varepsilon$

$A \rightarrow Ab \mid a$

$A \rightarrow aA'$

$A' \rightarrow bA' \mid \varepsilon$

逐步匹配

input: abbbb

# LL(k) Parser / Grammar / Language

- **LL(k) Parser**
  - A predictive parser that uses *k* lookahead tokens
  - **L**: scans the input from left to right[从左往右]
  - **L**: produces a leftmost derivation[生成最左推导]
  - **k**: using *k* input symbols of lookahead at each step to decide[向前看k个符号]
- **LL(k) Grammar**
  - A grammar that can be parsed using an LL(k) parser
  - LL(k) ⊂ 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[实现]

- Implemented in a recursive or non-recursive fashion[递归/非递归]
  - Recursive: recursive descent (recursive function calls, implicit stack)
  - Non-recursive: explicit stack to keep track of recursion[栈]
- Recursive LL(1) parser for: A→B | C, B→b, C→c
  - Parser consists of small functions, one for each non-terminal

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

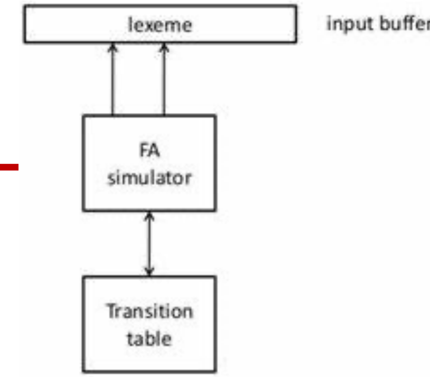
# LL(k) Parser Implementation (cont.)

- Recursive LL(1) parser for: A→B | C, B→b, C→c

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

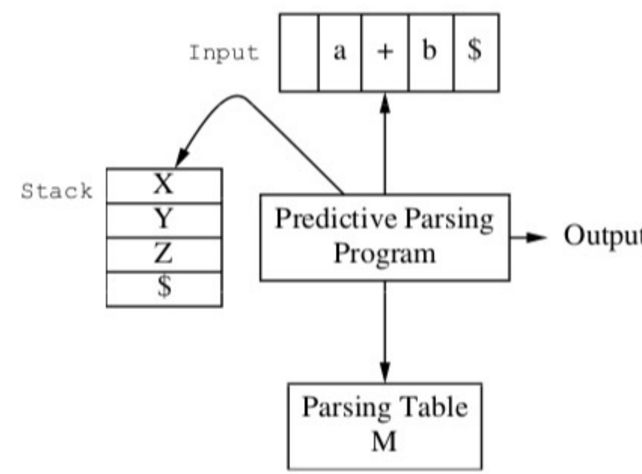
- Is there a way to express above code more concisely?[简洁]
  - Non-recursive LL(k) parsers use a **state transition table** (just like finite automata)[状态转换表]
  - Easier to automatically generate a non-recursive parser[自动化]

# LL(1) Parser[非递归]




- Table-driven parser[表驱动]: 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, $ marks the stack end
  - **Parse table** M[A, b]: an entry containing rule "A→…" or error
  - **Parser driver** (a.k.a., predictive parsing program): next action based on <stack top, current token?>
    - Reject on reaching error state
    - Accept on end of input & empty stack

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






?: The current token is treated as lookahead token.

# LL(1) Parse Table: Example

| table | int | * | + | ( | ) | $ |
|---|---|---|---|---|---|---|
| E | E → TE’ | | | E → TE’ | | |
| E’ | | | E’ → +E | | E’ → ε | E’ → ε |
| T | T → int T’ | | | T → (E) | | |
| T’ | | T’ → *T | T’ → ε | | T’ → ε | T’ → ε |

E → TE’
E’→ +E | ε
T → intT’ | (E)
T’→ *T | ε

- Implementation with 2D parse table
  - **First column** (each row) lists all <u>non-terminals</u> in the grammar
    - I.e., leftmost non-terminal in derivation
  - **First row** (each col) lists all possible <u>terminals</u> in the grammar and ‘$’
    - I.e., next input token
  - A **table entry** contains one <u>production</u>
    - One action for each <non-terminal, input> combination
    - It “predicts” the correct action based on one lookahead
    - No backtracking required

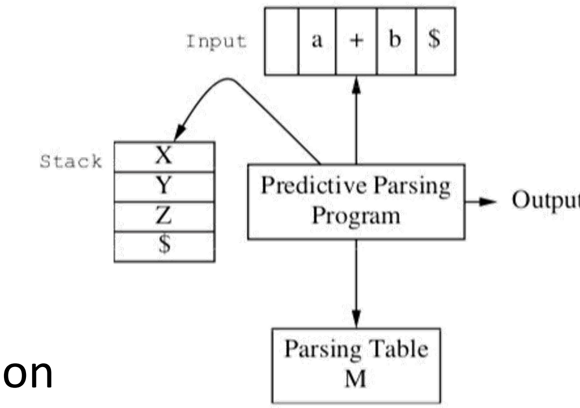
# LL(1) Parse Table: Example

| table | int | * | + | ( | ) | $ |
|---|---|---|---|---|---|---|
| E | E → TE’ | | | E → TE’ | | |
| E’ | | | E’ → +E | | E’ → ε | E’ → ε |
| T | T → int T’ | | | T → (E) | | |
| T’ | | T’ → *T | T’ → ε | | T’ → ε | T’ → ε |

E → TE’
E’→ +E | ε
T → intT’ | (E)
T’→ *T | ε

- Implementation with 2D parse table
  - **First column** (each row) lists all non-terminals in the grammar
    - I.e., leftmost non-terminal in derivation
  - **First row** (each col) lists all possible terminals in the grammar and ‘$’
    - I.e., next input token
  - 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

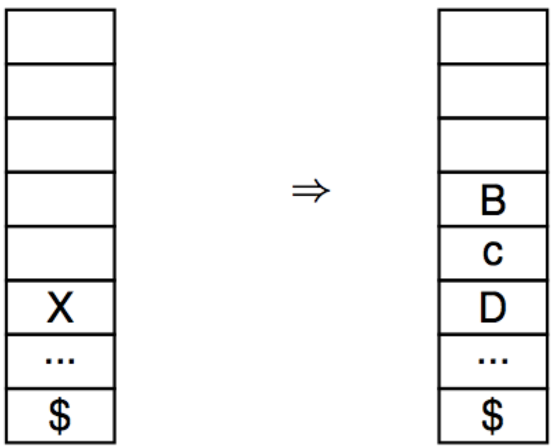
# LL(1) Parsing Algorithm[算法]




- Initial state[初始态]
  - **Input** tape: input tokens followed by '$'
  - **Stack**: start symbol followed by '$' at bottom
- General idea[总体思路]: 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[每步操作] parsing based on <X, a>
  - X: symbol at the top of the stack
  - a: current input token
    - If X ∈ T, then[终结符-比较]
      - If X == a == $, parser halts with "success"
      - If X == a != $, successful match, pop X from stack and advance input head
      - If X != a, parser halts and input is rejected
    - If X ∈ N, then[非终结符-展开]
      - If M[X,a] == "X→RHS", pop X and push RHS to stack
      - If M[X,a] == empty, parser halts and input is rejected

# Push RHS in Reverse Order[逆序入栈]

- For <X, a>
  - X: symbol at the top of the stack
  - a: current input token
- If M[X,a] = “X → BcD”




**逆序入栈：**最左符号需要被最先展开或比较（即，最左推导），因此需在靠近栈顶位置

- Performs the <u>leftmost derivation</u>: $\alpha$ 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

# Apply LL(1) Parsing to Grammar[应用]

- Consider the grammar

  $E \rightarrow T+E|T$

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

  - Left recursion? **NO!**
  - Left factoring? **YES.** $E \rightarrow T+E|T$, $T \rightarrow int*T \mid int$

- After rewriting grammar, we have

  $E \rightarrow TE'$

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

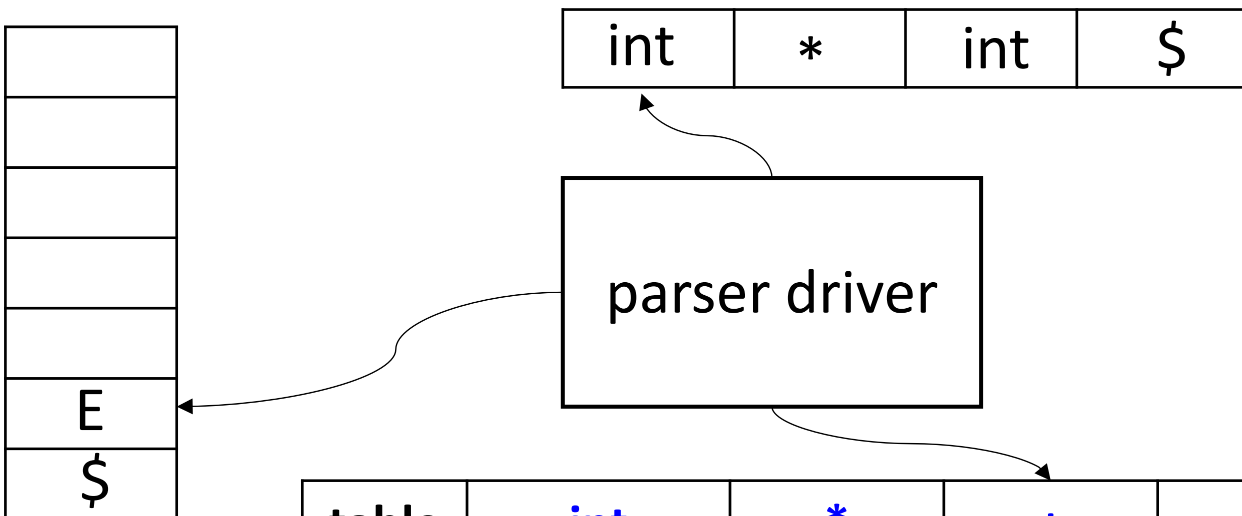
  $T \rightarrow intT' \mid (E)$

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

# Use the Parse Table

- To recognize "int * int"

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

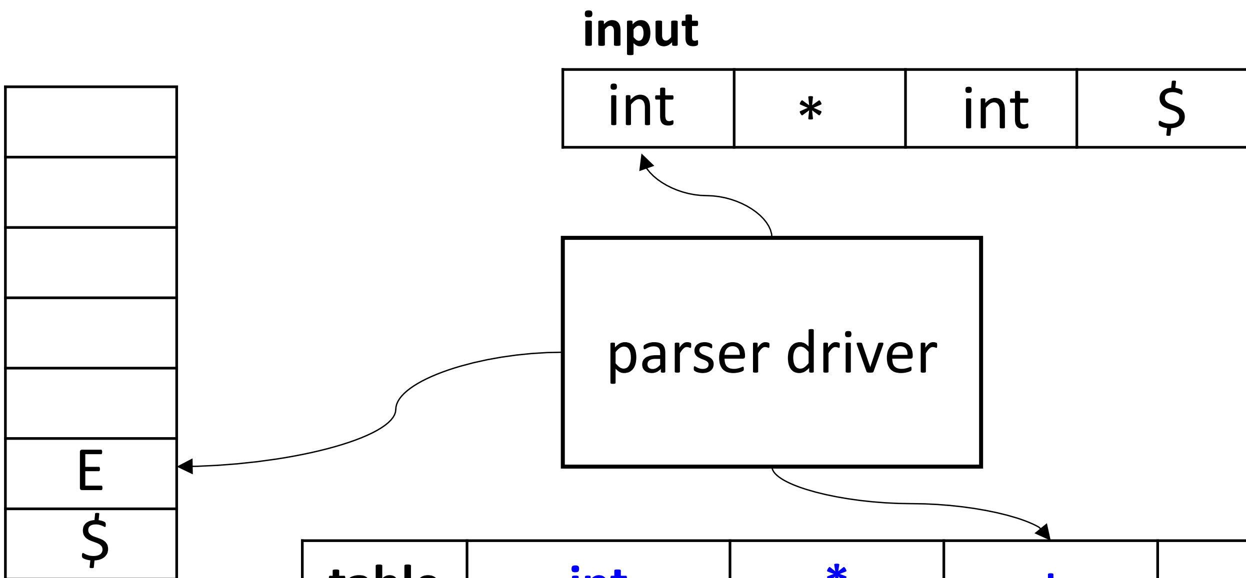



| table | int | * | + | ( | ) | $ |
|---|---|---|---|---|---|---|
| E | E → TE' | | | E → TE' | | |
| E' | | | E' → +E | | E' → ε | E' → ε |
| T | T → int T' | | | T → (E) | | |
| T' | | T' → *T | T' → ε | | T' → ε | T' → ε |

# Use the Parse Table

- To recognize "int * int"

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

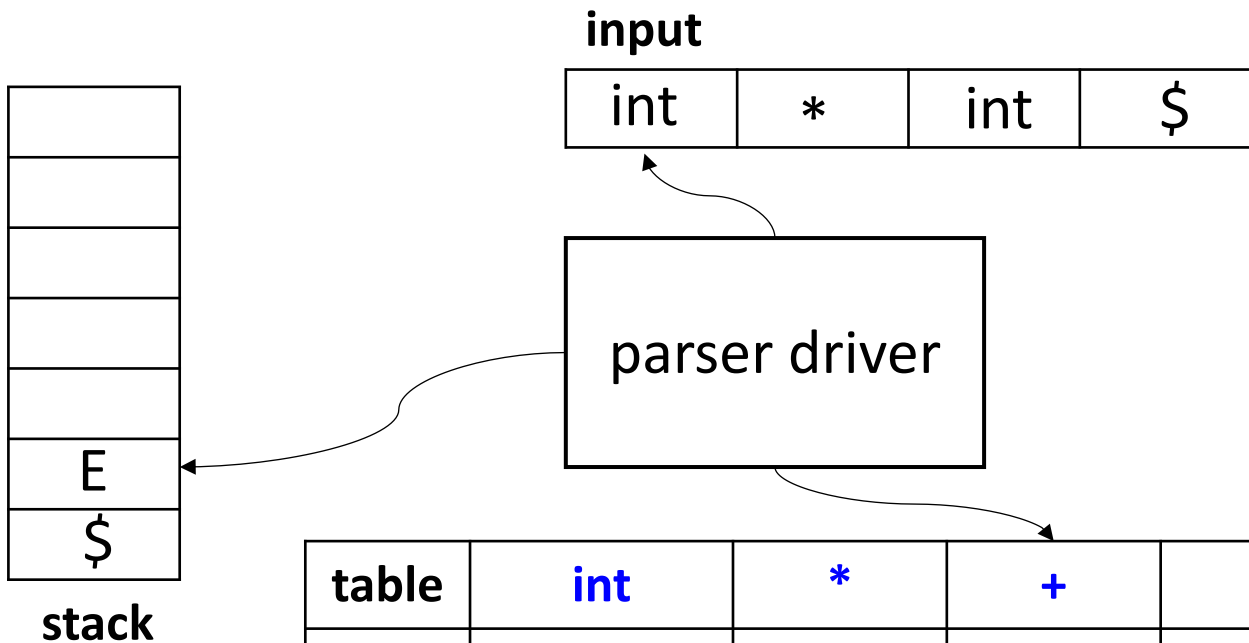



| table | int | * | + | ( | ) | $ |
|---|---|---|---|---|---|---|
| **E** | $E \rightarrow TE'$ | | | $E \rightarrow TE'$ | | |
| **E'** | | | $E' \rightarrow +E$ | | $E' \rightarrow \varepsilon$ | $E' \rightarrow \varepsilon$ |
| **T** | $T \rightarrow int\ T'$ | | | $T \rightarrow (E)$ | | |
| **T'** | | $T' \rightarrow *T$ | $T' \rightarrow \varepsilon$ | | $T' \rightarrow \varepsilon$ | $T' \rightarrow \varepsilon$ |

# Use the Parse Table

- To recognize "int * int"

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

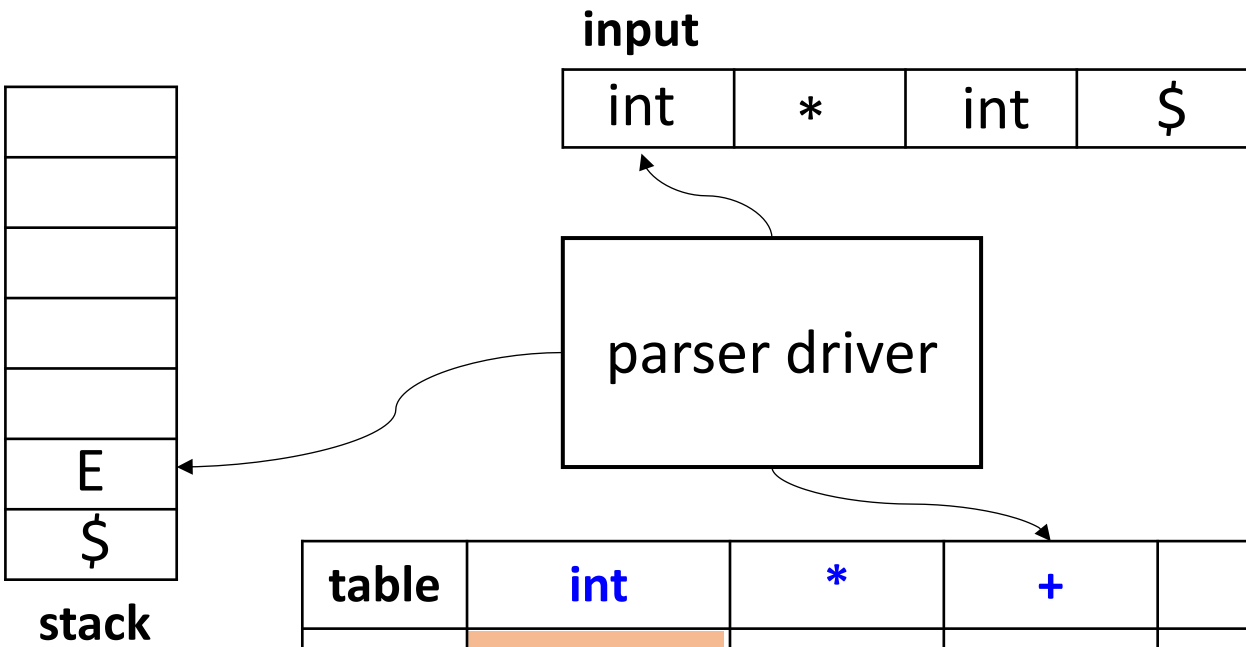



| table | int | * | + | ( | ) | $ |
|---|---|---|---|---|---|---|
| E | E → TE' | | | E → TE' | | |
| E' | | | E' → +E | | E' → ε | E' → ε |
| T | T → int T' | | | T → (E) | | |
| T' | | T' → *T | T' → ε | | T' → ε | T' → ε |

# Use the Parse Table

- To recognize "int * int"

E → TE'
E'→ +E | ε
T → intT' | (E)
T'→ *T | ε




| table | int | * | + | ( | ) | $ |
|---|---|---|---|---|---|---|
| E | E → TE' | | | E → TE' | | |
| E' | | | E' → +E | | E' → ε | E' → ε |
| T | T → int T' | | | T → (E) | | |
| T' | | T' → *T | T' → ε | | T' → ε | T' → ε |

# Use the Parse Table

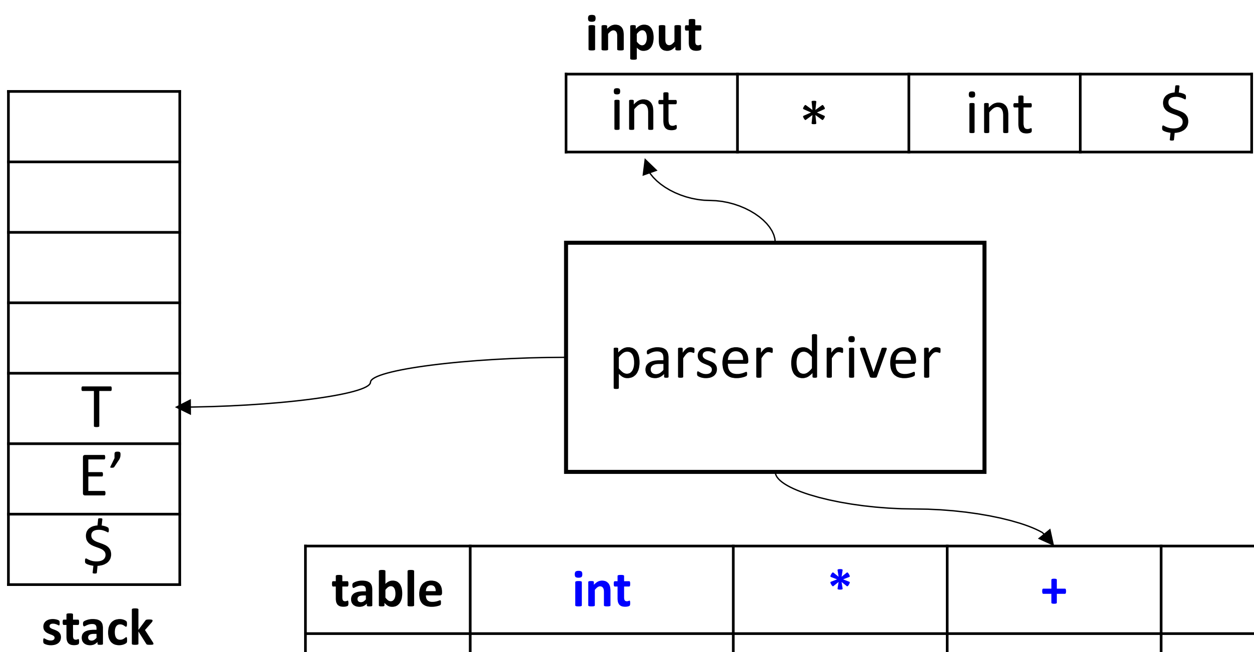
- To recognize "int * int"

$E \rightarrow TE'$

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

$T \rightarrow intT' \mid (E)$

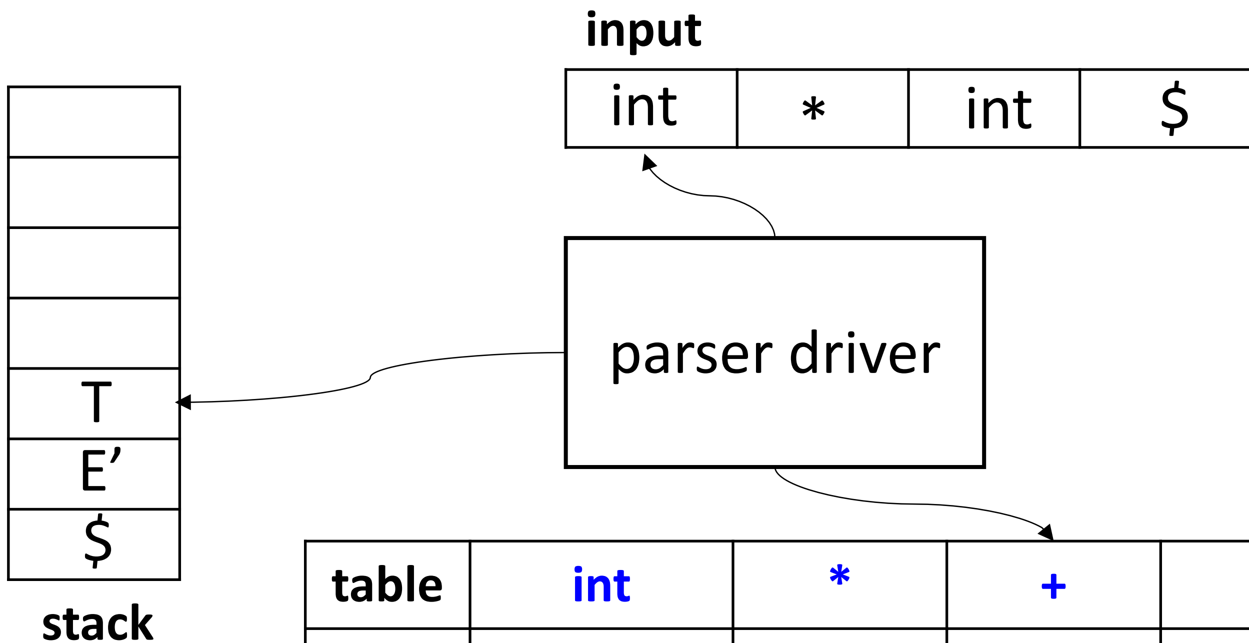
$T' \rightarrow *T \mid \varepsilon$




| table | int | * | + | ( | ) | $ |
|---|---|---|---|---|---|---|
| E | $E \rightarrow TE'$ | | | $E \rightarrow TE'$ | | |
| E' | | | $E' \rightarrow +E$ | | $E' \rightarrow \varepsilon$ | $E' \rightarrow \varepsilon$ |
| T | $T \rightarrow int\ T'$ | | | $T \rightarrow (E)$ | | |
| T' | | $T' \rightarrow *T$ | $T' \rightarrow \varepsilon$ | | $T' \rightarrow \varepsilon$ | $T' \rightarrow \varepsilon$ |

# Use the Parse Table

- To recognize “int * int”



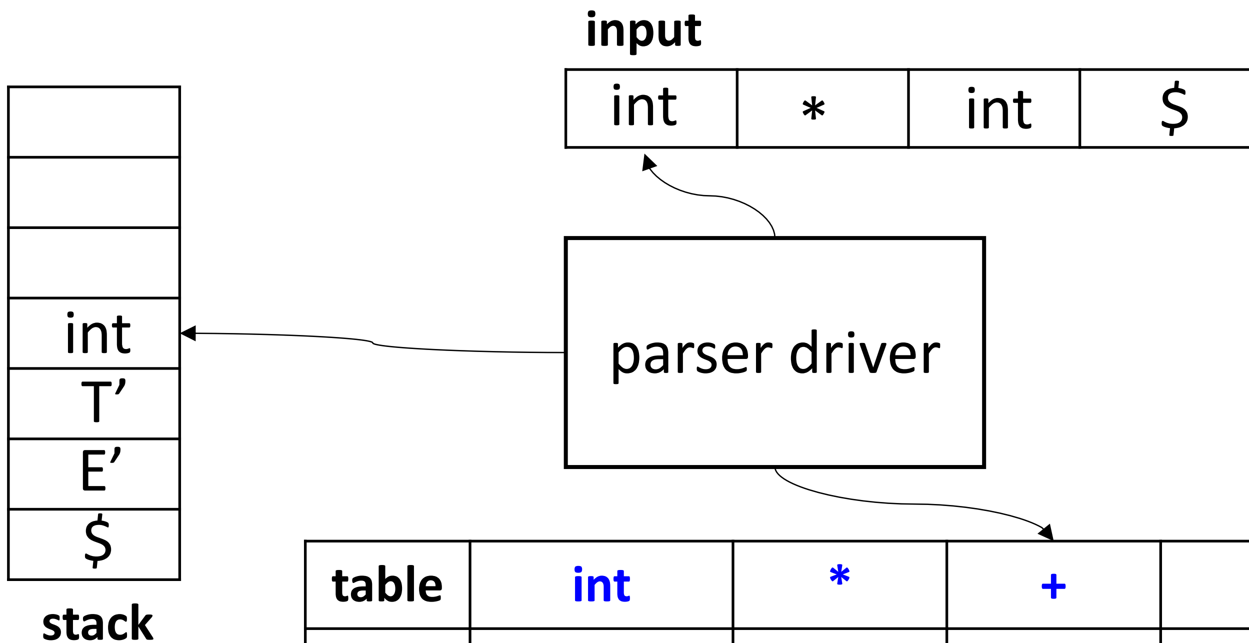

E → TE’
E’→ +E | ε
T → intT’ | (E)
T’→ *T | ε

| table | int | * | + | ( | ) | $ |
|---|---|---|---|---|---|---|
| **E** | E → TE’ | | | E → TE’ | | |
| **E’** | | | E’ → +E | | E’ → ε | E’ → ε |
| **T** | T → int T’ | | | T → (E) | | |
| **T’** | | T’ → *T | T’ → ε | | T’ → ε | T’ → ε |

# Use the Parse Table

- To recognize "int * int"

E → TE'
E'→ +E | ε
T → intT' | (E)
T'→ *T | ε




| table | int | * | + | ( | ) | $ |
|---|---|---|---|---|---|---|
| E | E → TE' | | | E → TE' | | |
| E' | | | E' → +E | | E' → ε | E' → ε |
| T | T → int T' | | | T → (E) | | |
| T' | | T' → *T | T' → ε | | T' → ε | T' → ε |

# Use the Parse Table

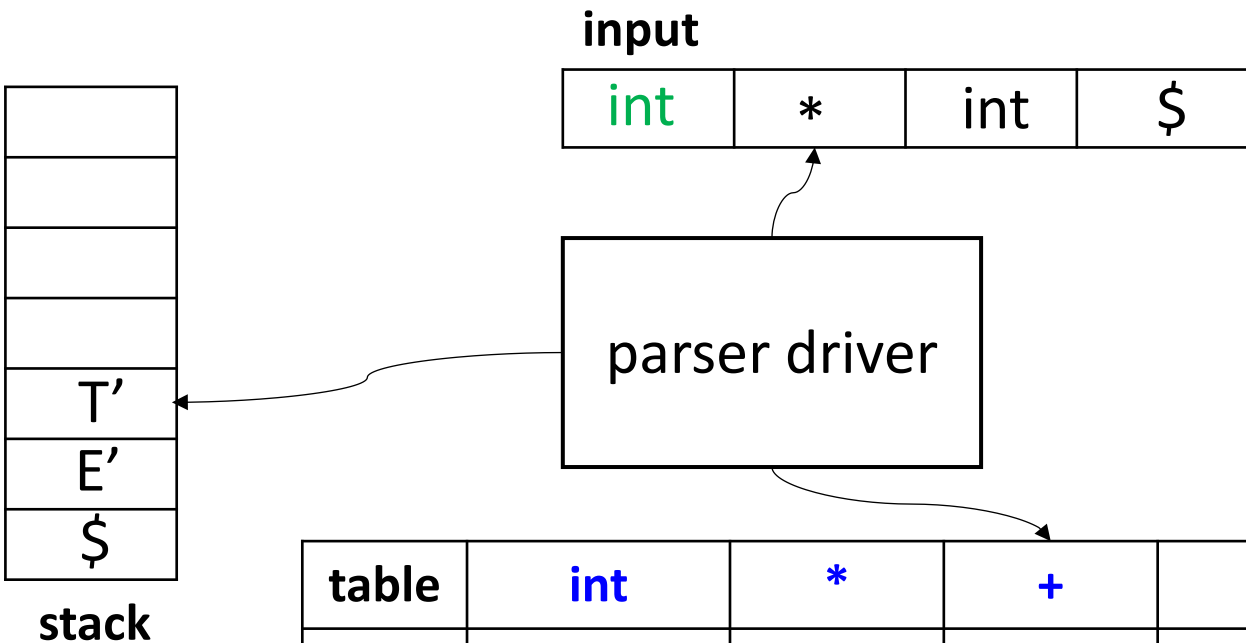
- To recognize "int * int"

$E \rightarrow TE'$

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

$T \rightarrow intT' \mid (E)$

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

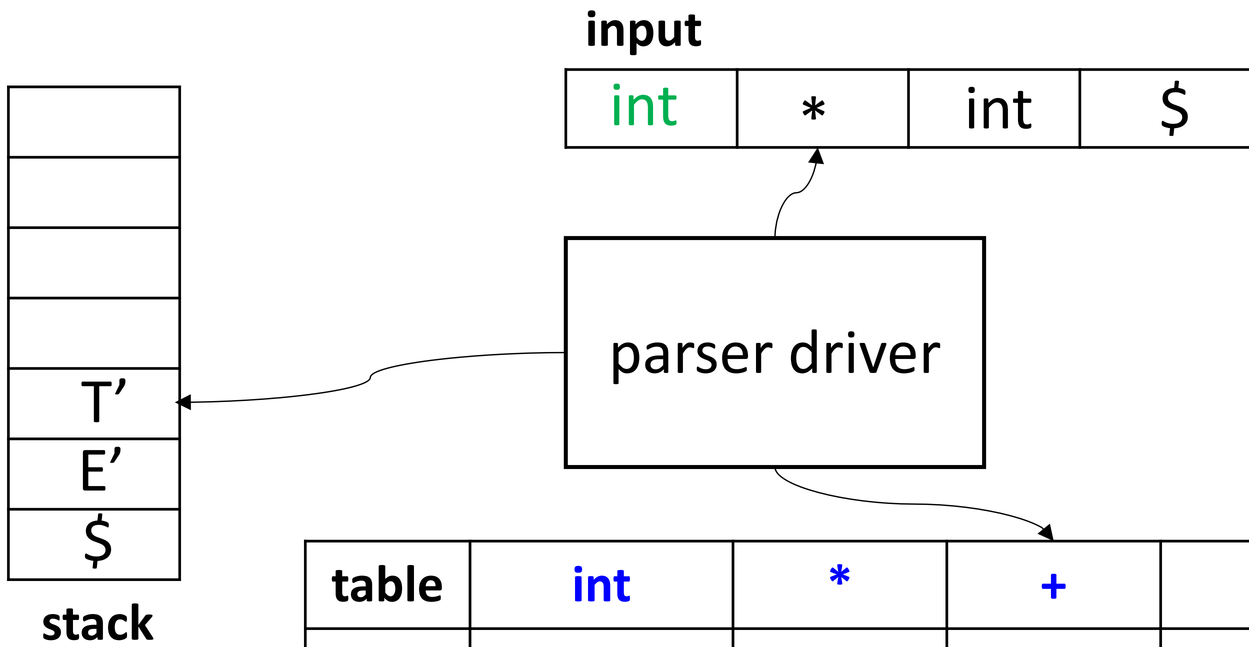



| table | int | * | + | ( | ) | $ |
|---|---|---|---|---|---|---|
| E | $E \rightarrow TE'$ | | | $E \rightarrow TE'$ | | |
| E' | | | $E' \rightarrow +E$ | | $E' \rightarrow \varepsilon$ | $E' \rightarrow \varepsilon$ |
| T | $T \rightarrow int\ T'$ | | | $T \rightarrow (E)$ | | |
| T' | | $T' \rightarrow *T$ | $T' \rightarrow \varepsilon$ | | $T' \rightarrow \varepsilon$ | $T' \rightarrow \varepsilon$ |

# Use the Parse Table

- To recognize “int * int”

E → TE’
E’→ +E | ε
T → intT’ | (E)
T’→ *T | ε




| table | int | * | + | ( | ) | $ |
|---|---|---|---|---|---|---|
| **E** | E → TE’ | | | E → TE’ | | |
| **E’** | | | E’ → +E | | E’ → ε | E’ → ε |
| **T** | T → int T’ | | | T → (E) | | |
| **T’** | | T’ → *T | T’ → ε | | T’ → ε | T’ → ε |

# Use the Parse Table

- To recognize "int * int"

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

**input**

| int | * | int | $ |
|---|---|---|---|

parser driver

| |
|---|
| |
| |
| |
| * |
| T |
| E' |
| $ |

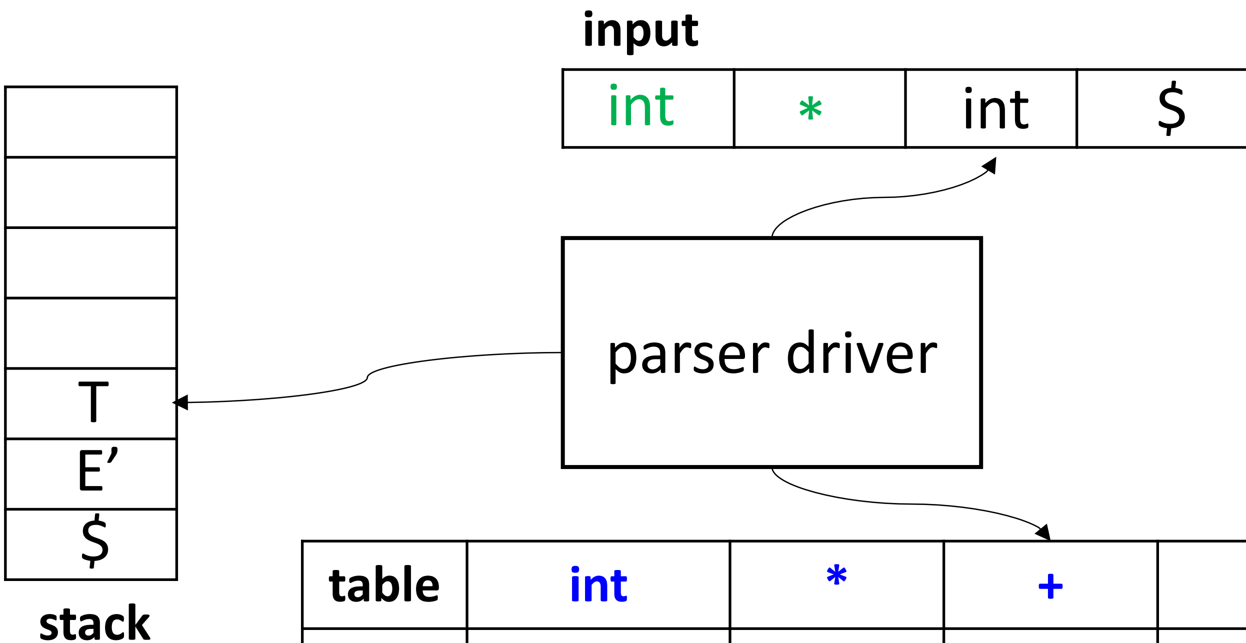
**stack**

| **table** | **int** | ***** | **+** | **(** | **)** | **$** |
|---|---|---|---|---|---|---|
| **E** | $E \rightarrow TE'$ | | | $E \rightarrow TE'$ | | |
| **E'** | | | $E' \rightarrow +E$ | | $E' \rightarrow \varepsilon$ | $E' \rightarrow \varepsilon$ |
| **T** | $T \rightarrow int\ T'$ | | | $T \rightarrow (E)$ | | |
| **T'** | | $T' \rightarrow *T$ | $T' \rightarrow \varepsilon$ | | $T' \rightarrow \varepsilon$ | $T' \rightarrow \varepsilon$ |

# Use the Parse Table

- To recognize “int * int”

E → TE’
E’→ +E | ε
T → intT’ | (E)
T’→ *T | ε

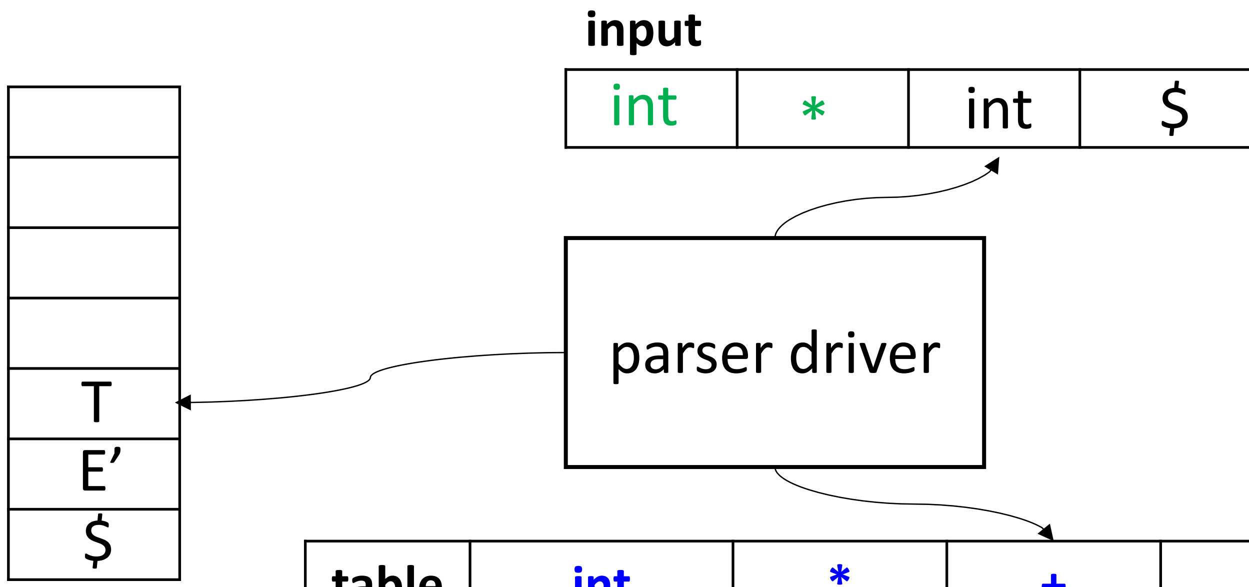



| table | int | * | + | ( | ) | $ |
|---|---|---|---|---|---|---|
| **E** | E → TE’ | | | E → TE’ | | |
| **E’** | | | E’ → +E | | E’ → ε | E’ → ε |
| **T** | T → int T’ | | | T → (E) | | |
| **T’** | | T’ → *T | T’ → ε | | T’ → ε | T’ → ε |

# Use the Parse Table

- To recognize "int * int"

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



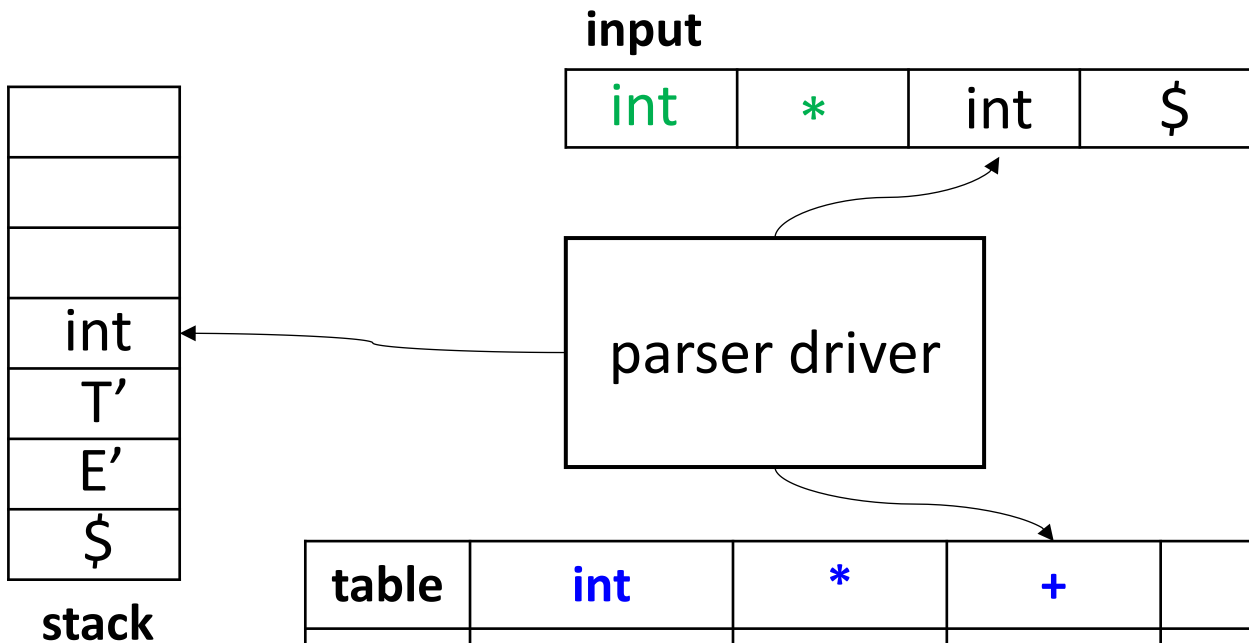

stack

| table | int | * | + | ( | ) | $ |
|---|---|---|---|---|---|---|
| **E** | $E \rightarrow TE'$ | | | $E \rightarrow TE'$ | | |
| **E'** | | | $E' \rightarrow +E$ | | $E' \rightarrow \varepsilon$ | $E' \rightarrow \varepsilon$ |
| **T** | $T \rightarrow int\ T'$ | | | $T \rightarrow (E)$ | | |
| **T'** | | $T' \rightarrow *T$ | $T' \rightarrow \varepsilon$ | | $T' \rightarrow \varepsilon$ | $T' \rightarrow \varepsilon$ |

# Use the Parse Table

- To recognize "int * int"

E → TE'
E' → +E | ε
T → intT' | (E)
T' → *T | ε

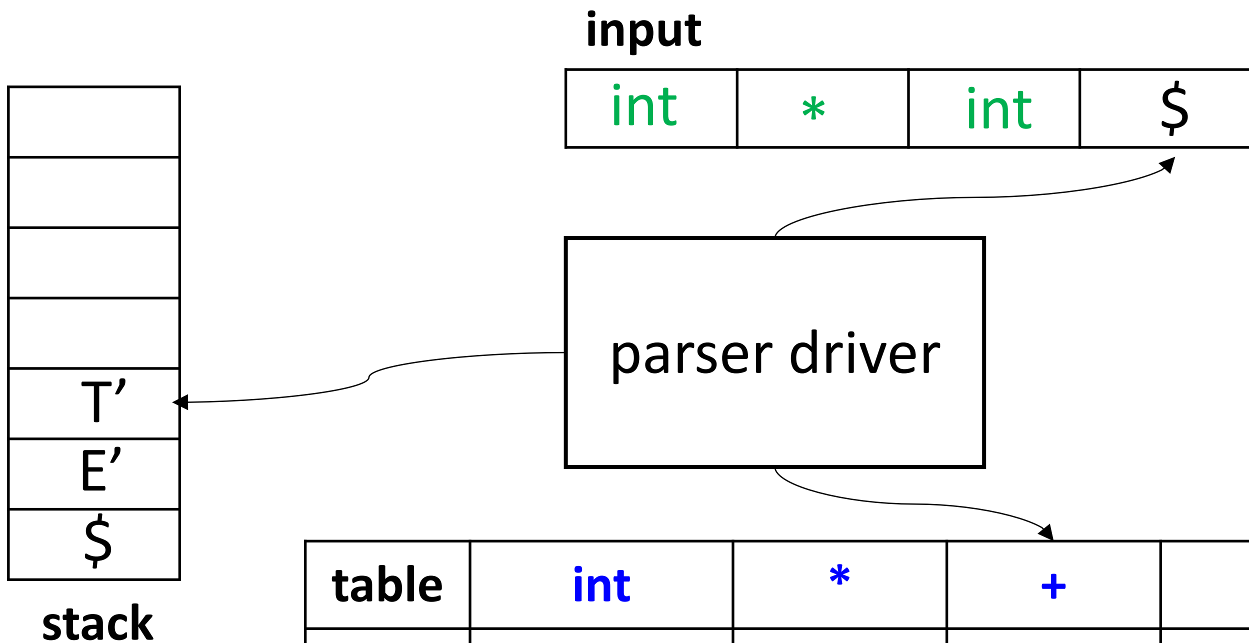



| table | int | * | + | ( | ) | $ |
|---|---|---|---|---|---|---|
| E | E → TE' | | | E → TE' | | |
| E' | | | E' → +E | | E' → ε | E' → ε |
| T | T → int T' | | | T → (E) | | |
| T' | | T' → *T | T' → ε | | T' → ε | T' → ε |

# Use the Parse Table

- To recognize "int * int"

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




| table | int | * | + | ( | ) | $ |
|---|---|---|---|---|---|---|
| E | E → TE’ | | | E → TE’ | | |
| E’ | | | E’ → +E | | E’ → ε | E’ → ε |
| T | T → int T’ | | | T → (E) | | |
| T’ | | T’ → *T | T’ → ε | | T’ → ε | T’ → ε |

# Use the Parse Table

- To recognize “int * int”

$E \to TE'$

$E' \to +E \mid \varepsilon$

$T \to intT' \mid (E)$

$T' \to *T \mid \varepsilon$

**input**

| | | | |
|---|---|---|---|
| int | * | int | $ |

parser driver

**stack**

| |
|---|
| |
| |
| |
| |
| T’ |
| E’ |
| $ |

| **table** | **int** | ***** | **+** | **(** | **)** | **$** |
|---|---|---|---|---|---|---|
| **E** | E → TE’ | | | E → TE’ | | |
| **E’** | | | E’ → +E | | E’ → ε | E’ → ε |
| **T** | T → int T’ | | | T → (E) | | |
| **T’** | | T’ → *T | T’ → ε | | T’ → ε | T’ → ε |

# Use the Parse Table

- To recognize "int * int"

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

**input**

| int | * | int | $ |
|---|---|---|---|

parser driver

**stack**

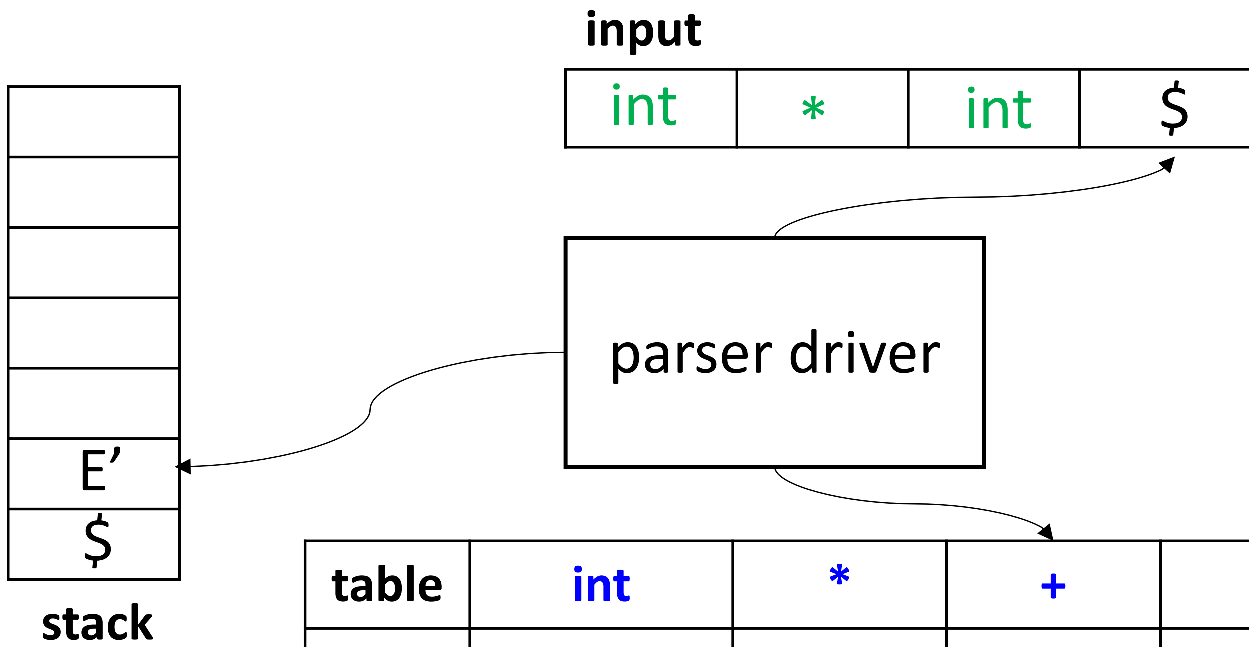
| |
|---|
| |
| |
| |
| |
| |
| E' |
| $ |

| **table** | **int** | ***** | **+** | **(** | **)** | **$** |
|---|---|---|---|---|---|---|
| **E** | $E \rightarrow TE'$ | | | $E \rightarrow TE'$ | | |
| **E'** | | | $E' \rightarrow +E$ | | $E' \rightarrow \varepsilon$ | $E' \rightarrow \varepsilon$ |
| **T** | $T \rightarrow int\ T'$ | | | $T \rightarrow (E)$ | | |
| **T'** | | $T' \rightarrow *T$ | $T' \rightarrow \varepsilon$ | | $T' \rightarrow \varepsilon$ | $T' \rightarrow \varepsilon$ |

# Use the Parse Table

- To recognize "int * int"

E → TE'
E' → +E | ε
T → intT' | (E)
T' → *T | ε

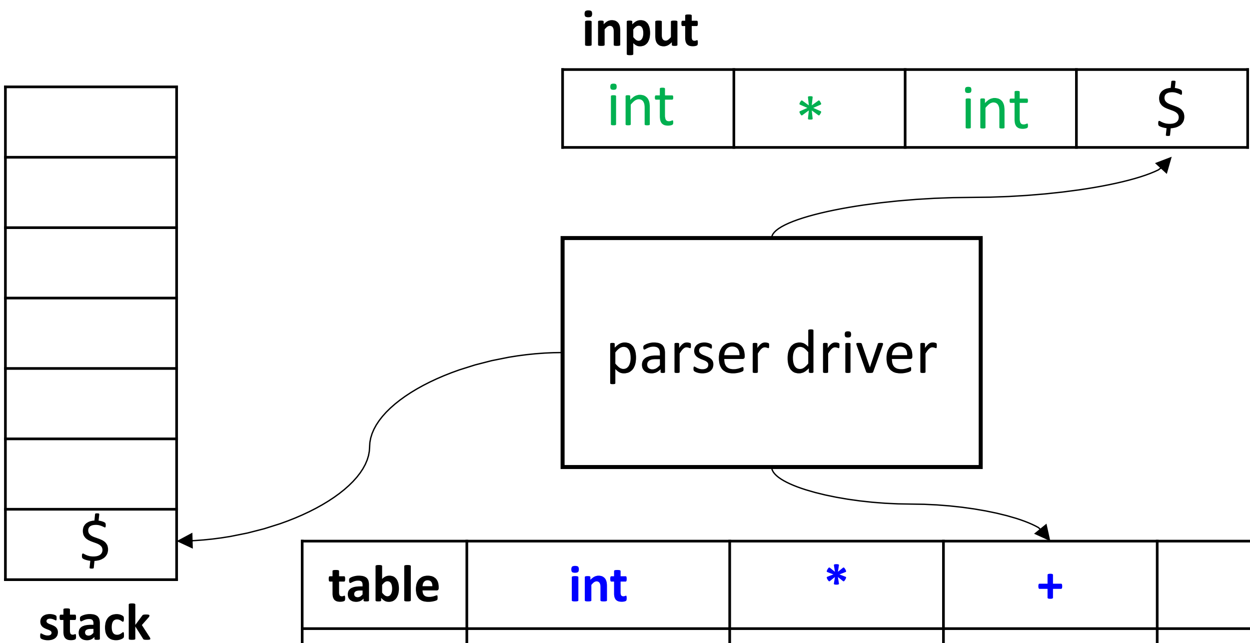



| table | int | * | + | ( | ) | $ |
|---|---|---|---|---|---|---|
| **E** | E → TE' | | | E → TE' | | |
| **E'** | | | E' → +E | | E' → ε | E' → ε |
| **T** | T → int T' | | | T → (E) | | |
| **T'** | | T' → *T | T' → ε | | T' → ε | T' → ε |

# Use the Parse Table

- To recognize "int * int"

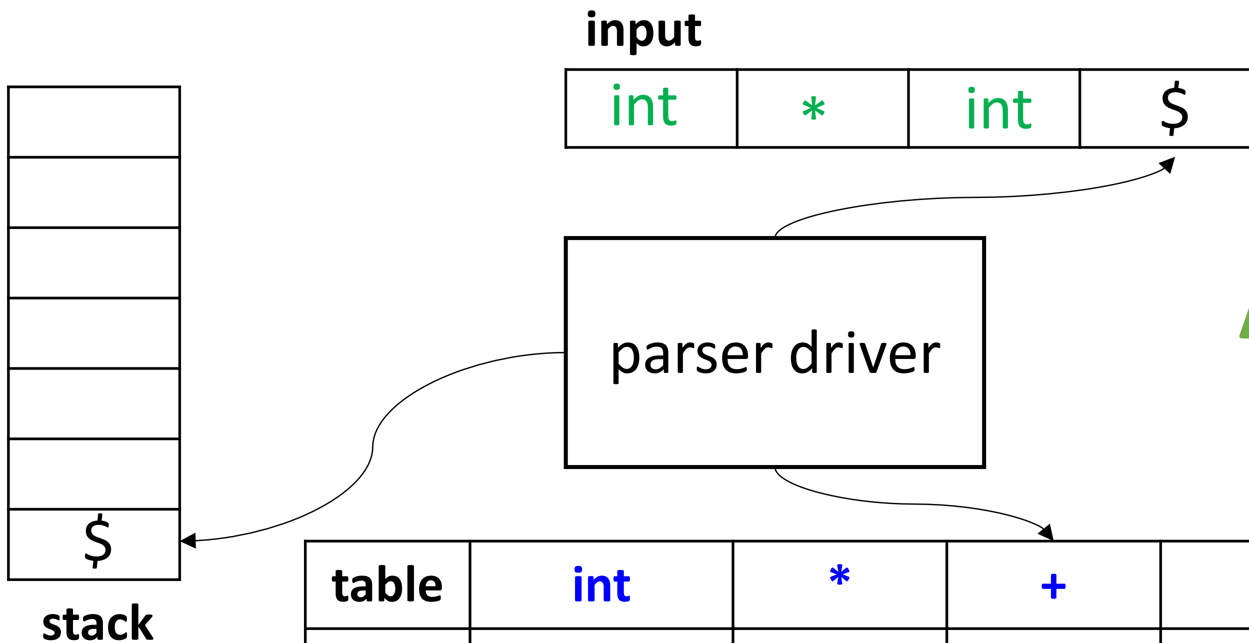
$E \rightarrow TE'$
$E' \rightarrow +E \mid \varepsilon$
$T \rightarrow intT' \mid (E)$
$T' \rightarrow *T \mid \varepsilon$




| table | int | * | + | ( | ) | $ |
|---|---|---|---|---|---|---|
| **E** | $E \rightarrow TE'$ | | | $E \rightarrow TE'$ | | |
| **E'** | | | $E' \rightarrow +E$ | | $E' \rightarrow \varepsilon$ | $E' \rightarrow \varepsilon$ |
| **T** | $T \rightarrow int\ T'$ | | | $T \rightarrow (E)$ | | |
| **T'** | | $T' \rightarrow *T$ | $T' \rightarrow \varepsilon$ | | $T' \rightarrow \varepsilon$ | $T' \rightarrow \varepsilon$ |

# Use the Parse Table

- To recognize "int * int"




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

**ACCEPT!**

| table | int | * | + | ( | ) | $ |
|---|---|---|---|---|---|---|
| **E** | $E \rightarrow TE'$ | | | $E \rightarrow TE'$ | | |
| **E'** | | | $E' \rightarrow +E$ | | $E' \rightarrow \varepsilon$ | $E' \rightarrow \varepsilon$ |
| **T** | $T \rightarrow int\ T'$ | | | $T \rightarrow (E)$ | | |
| **T'** | | $T' \rightarrow *T$ | $T' \rightarrow \varepsilon$ | | $T' \rightarrow \varepsilon$ | $T' \rightarrow \varepsilon$ |

# Recognize Sequence[解析过程]

| Matched | Stack (unmatched) | Input | Action |
|---|---|---|---|
| | E $ | int * int $ | E → TE’ |
| | T E’ $ | int * int $ | T → int T’ |
| int | int T’ E’ $ | int * int $ | match |
| int | T’ E’ $ | * int $ | T’ → *T |
| int | * T E’ $ | * int $ | match |
| int * | T E’ $ | int $ | T → int T’ |
| int * | int T’ E’ $ | int $ | match |
| int * int | T’ E’ $ | $ | T’ → ε |
| int * int | E’ $ | $ | E’ → ε |
| int * int | $ | $ | Halt-accept |

E → TE’
E’→ +E | ε
T → intT’ | (E)
T’→ *T | ε

Input: int * int

- ‘Matched + Stack’ constructs the sentential form[句型]
- Actions correspond to productions in leftmost derivation

# LL(1) Parse Table: Example

| table | int | * | + | ( | ) | $ |
|---|---|---|---|---|---|---|
| E | E → TE’ | | | E → TE’ | | |
| E’ | | | E’ → +E | | E’ → ε | E’ → ε |
| T | T → int T’ | | | T → (E) | | |
| T’ | | T’ → *T | T’ → ε | | T’ → ε | T’ → ε |

E → TE’
E’→ +E | ε
T → intT’ | (E)
T’→ *T | ε

- Implementation with 2D parse table
  - **First column** (each row) lists all <u>non-terminals</u> in the grammar
    - I.e., leftmost non-terminal in derivation
  - **First row** (each col) lists all possible <u>terminals</u> in the grammar and ‘$’
    - I.e., next input token
  - A **table entry** contains one <u>production</u>
    - One action for each <non-terminal, input> combination
    - It “predicts” the correct action based on one lookahead
    - No backtracking required