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7讲：语法分析(4)

张献伟

xianweiz.github.io

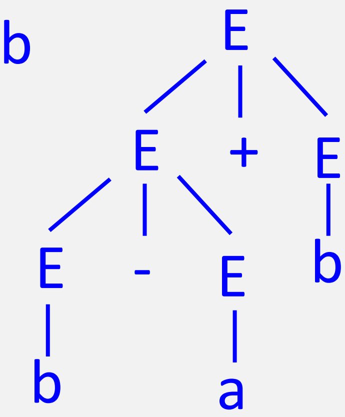
DCS290, 3/15/2022

Quiz Questions

- Q1: for grammar E → E-E | E+E | a | b, and input b-a+b, give one rightmost derivation.

  E → E - E → E - E + E → E - E + b → E - a + b → b - a + b

  E → E + E → E + b → E - E + b → E - a + b → b - a + b



- Q2: plot parse tree of the derivation in Q1.

- Q3: briefly describe top-down parsing.

  Mimics leftmost derivation, expand the start symbol to input string.

- Q4: why top-down parsing cannot handle left recursive grammars?

  Repeatedly expanding without consuming any input symbol.

- Q5: is grammar S → T a| a, T → S left recursive? Why?

  YES. S → T a → S a (indirect left-recursive).

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[候选产生式开始符号唯一]
  - 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，即要匹配的那个（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]
  - Current nonterminal being processed[当前正处理的非终结符]
- Patterns in grammars that prevent predictive parsing[并非总是能预测分析]
  - **Common prefix**[共同前缀]:

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

    Given input terminal(s) α, cannot choose between two rules
  - **Left recursion**[左递归]:

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

    Lookahead symbol changes only when a terminal is matched

    What is the language of the grammar? $\alpha\beta^*$

Rewrite Grammars for Prediction[改写]

- **Left factoring**[左公因子提取]: removes common left prefix
  - In previous example: A→αβ | αγ
  - can be changed to

    A → α A'

    A' → β | γ

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

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

    S' → **else** *stmt* | ε
  - After processing α, A' can can choose between β or γ (assuming β or γ do not start with α)

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→Aβ|α
  - can be changed to

    A → α A'

    A'→βA' | ε
  - After processing α, A' can can choose between β or ε (assuming β doesn't start with α or A' isn't followed by α)

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.)

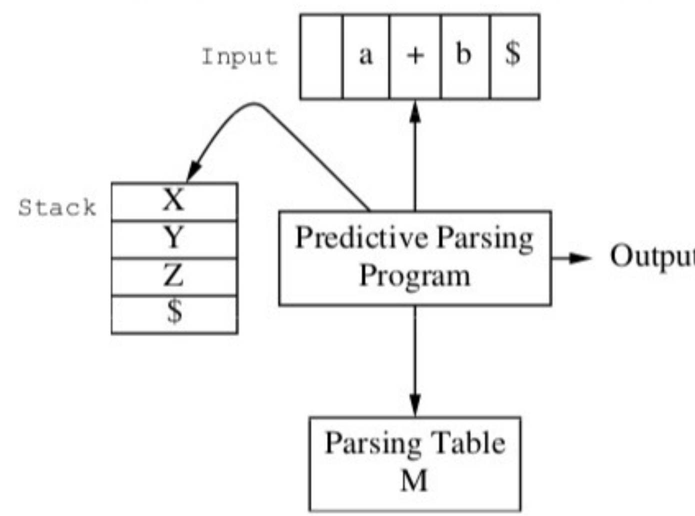
- Recursive LL(1) parser for: A→B | C, B→b, C→c

```
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 matched against the input
- Top of stack = leftmost pending terminal or non-terminal

?: The current token is treated as lookahead token.

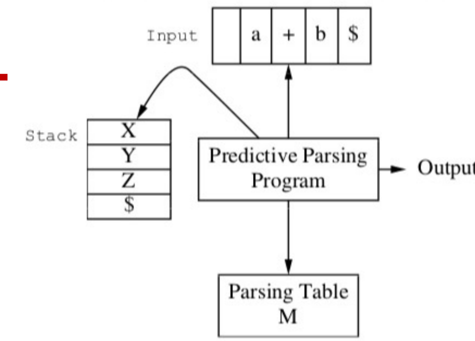
LL(1) Parse Table: Example

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** lists all non-terminals in the grammar
    - I.e., leftmost non-terminal in derivation
  - **First row** 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

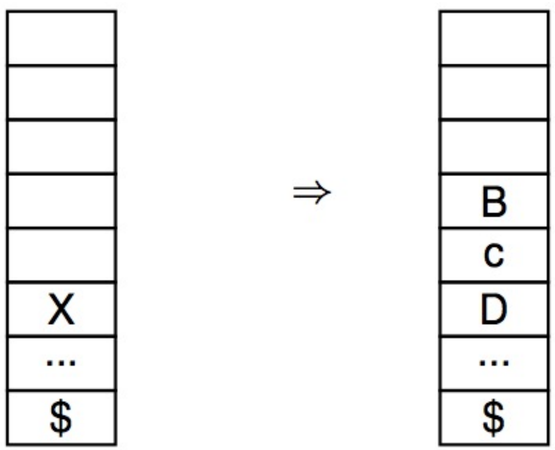
LL(1) Parsing Algorithm[算法]



- 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
      - 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
      - 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"



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

Apply LL(1) Parsing to Grammar[应用]

- Consider the grammar

  E → T+E|T

  T → int*T | int | (E)

  – Left recursion? **NO!**

  – Left factoring? **YES.** E → T+E|T, T → int*T | int

- After rewriting grammar, we have

  E → TE'

  E'→ +E | ε

  T → intT' | (E)

  T'→ *T | ε

Use the Parse Table

- To recognize "int * int"

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

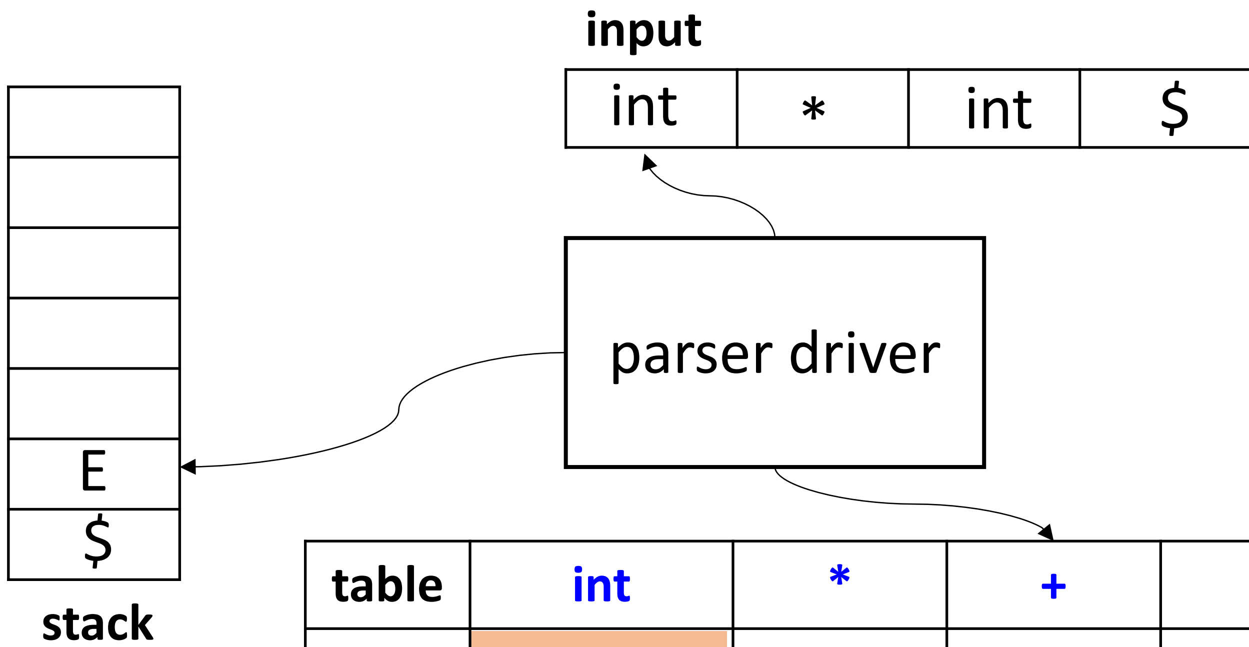
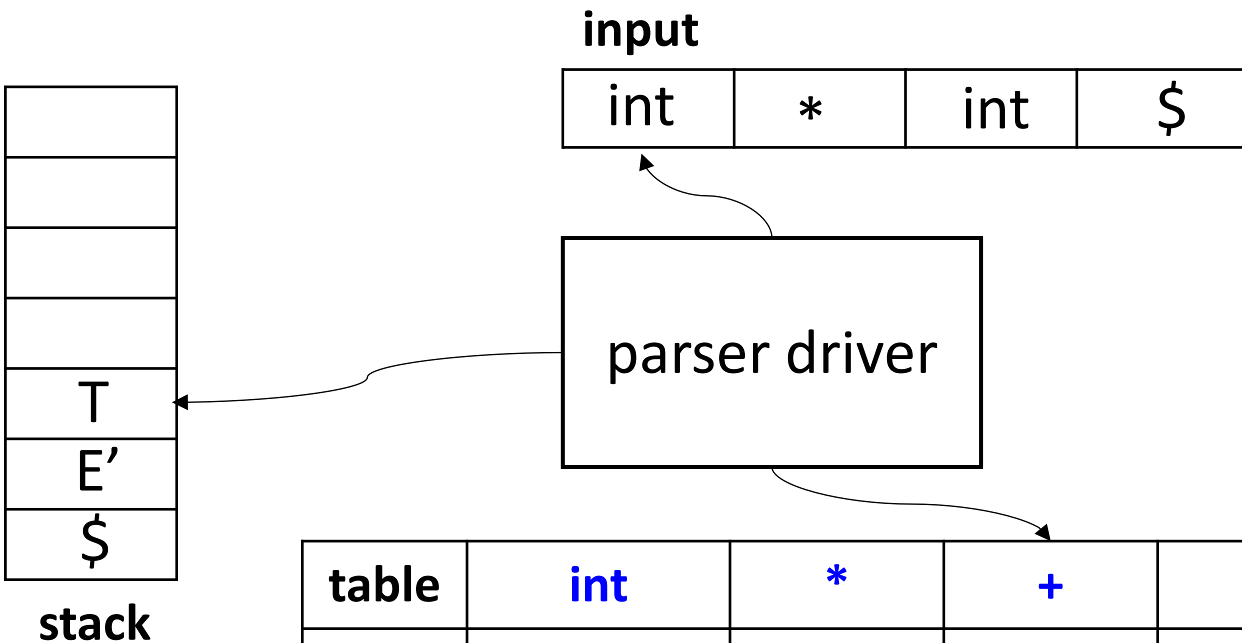


table	int	*	+	(	)	$
E	E → TE'			E → TE'		
E'			E' → +E		E' → ε	E' → ε
T	T → int T'			T → (E)		
T'		T' → *T	T' → ε		T' → ε	T' → ε

Use the Parse Table

- To recognize "int * int"



E → 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

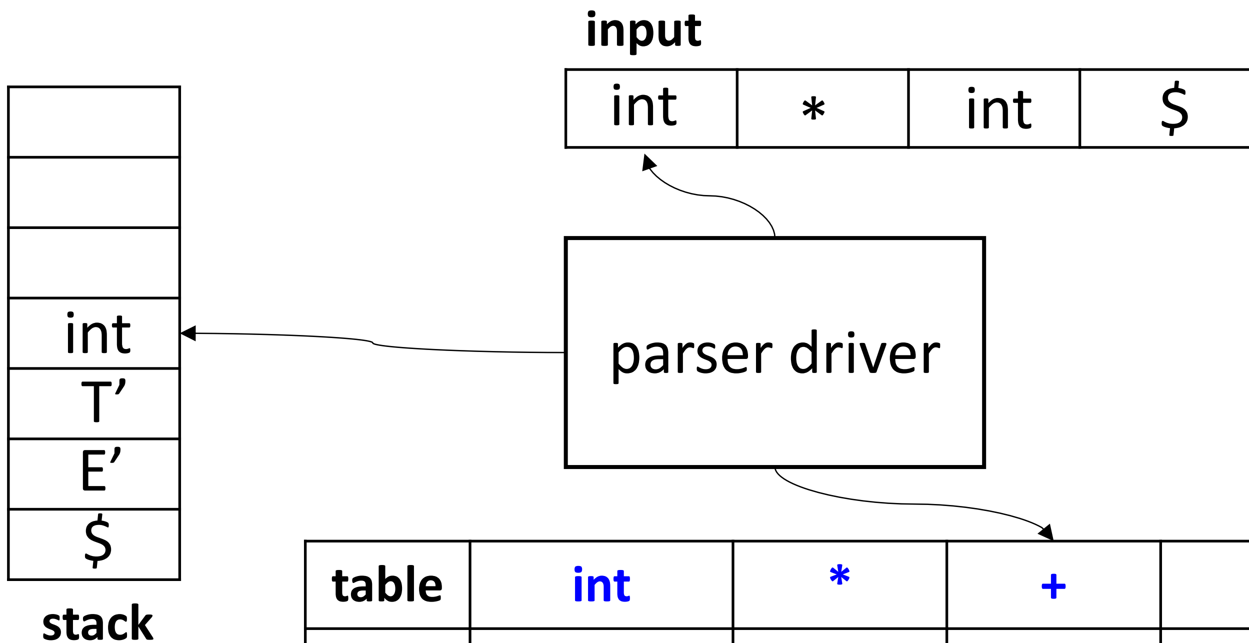


table	int	*	+	(	)	$
E	E → TE'			E → TE'		
E'			E' → +E		E' → ε	E' → ε
T	T → int T'			T → (E)		
T'		T' → *T	T' → ε		T' → ε	T' → ε

Use the Parse Table

- To recognize "int * int"

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

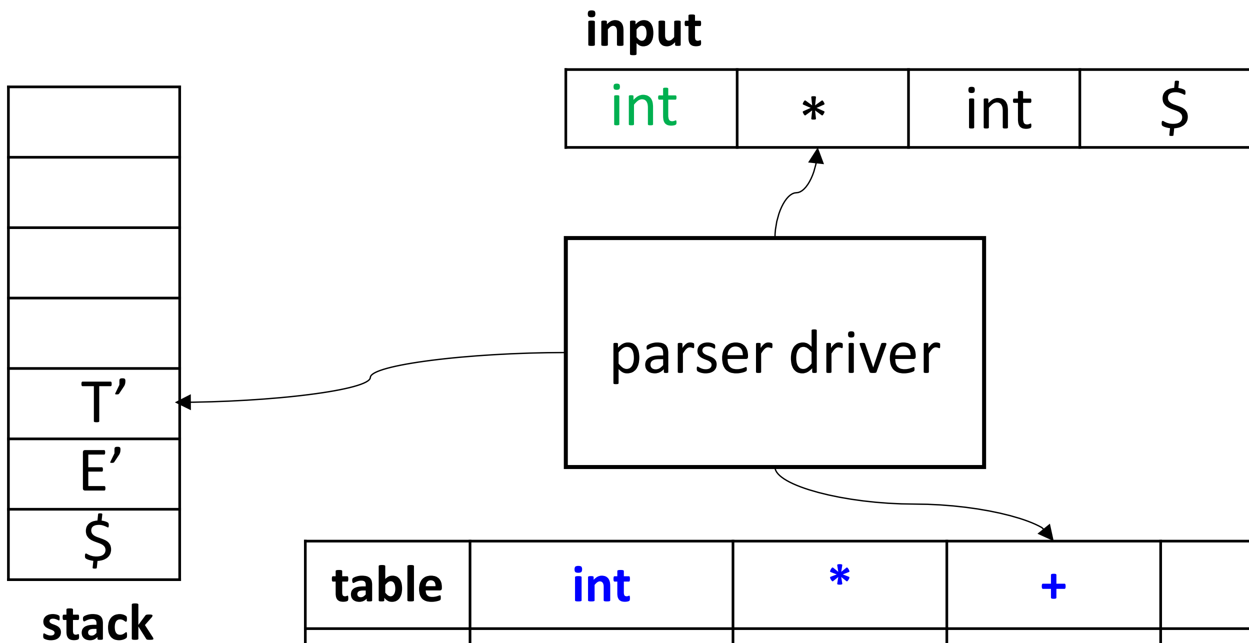


table	int	*	+	(	)	$
E	E → TE'			E → TE'		
E'			E' → +E		E' → ε	E' → ε
T	T → int T'			T → (E)		
T'		T' → *T	T' → ε		T' → ε	T' → ε

Use the Parse Table

- To recognize "int * int"

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

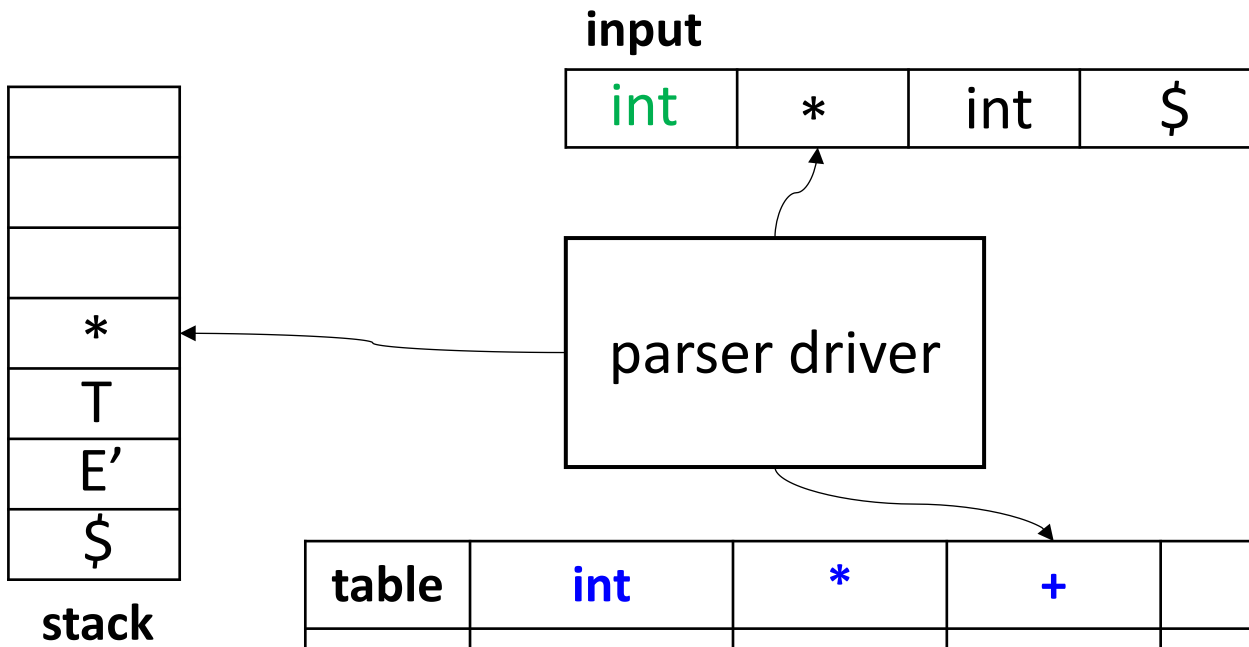


table	int	*	+	(	)	$
E	E → TE'			E → TE'		
E'			E' → +E		E' → ε	E' → ε
T	T → int T'			T → (E)		
T'		T' → *T	T' → ε		T' → ε	T' → ε

Use the Parse Table

- To recognize "int * int"

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

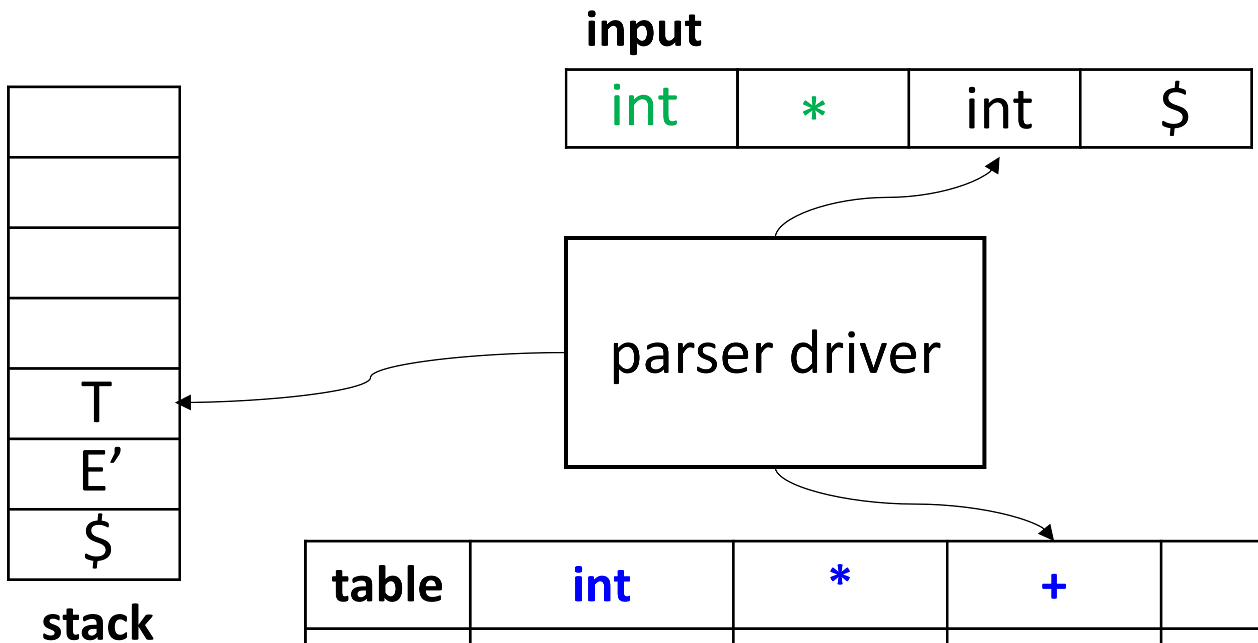


table	int	*	+	(	)	$
E	E → TE'			E → TE'		
E'			E' → +E		E' → ε	E' → ε
T	T → int T'			T → (E)		
T'		T' → *T	T' → ε		T' → ε	T' → ε

Use the Parse Table

- To recognize "int * int"

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

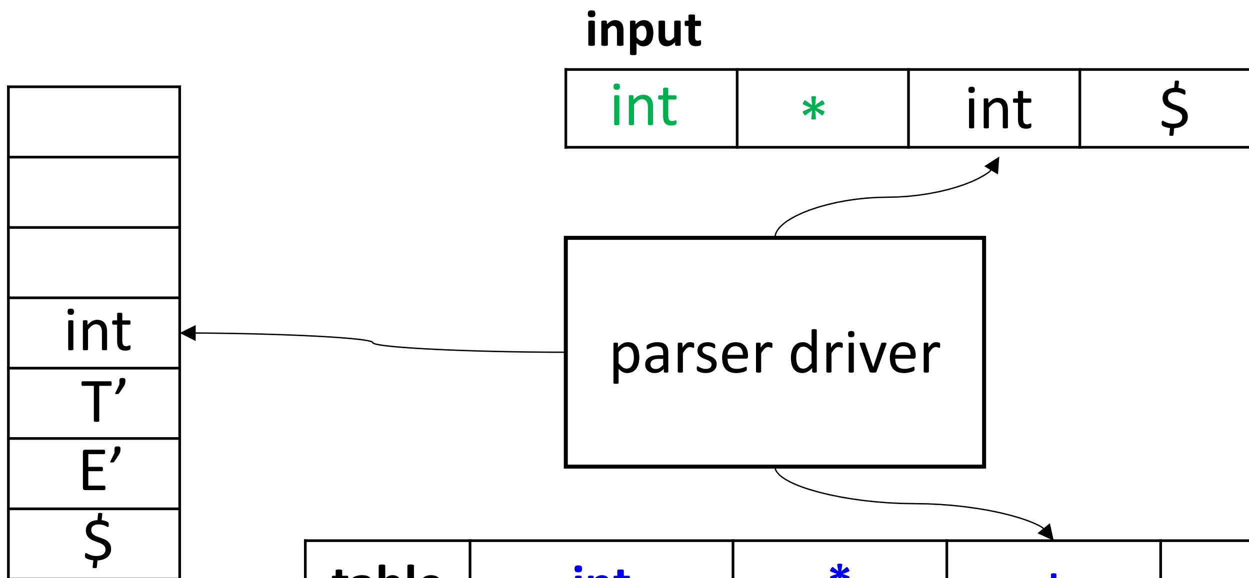
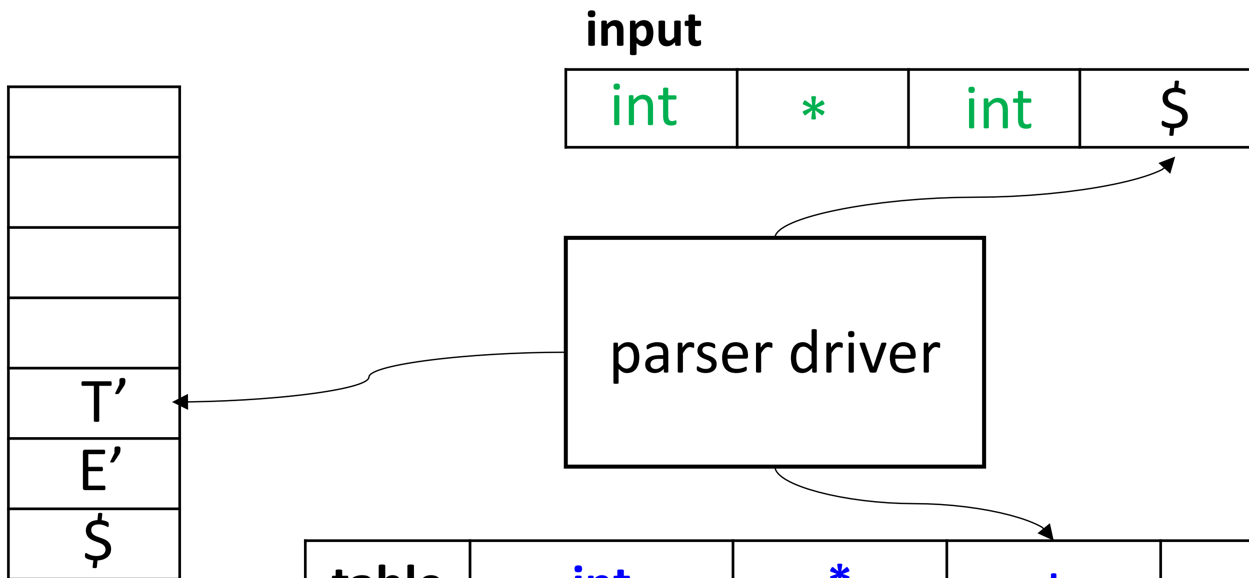


table	int	*	+	(	)	$
E	E → TE'			E → TE'		
E'			E' → +E		E' → ε	E' → ε
T	T → int T'			T → (E)		
T'		T' → *T	T' → ε		T' → ε	T' → ε

Use the Parse Table

- To recognize "int * int"

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



stack

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

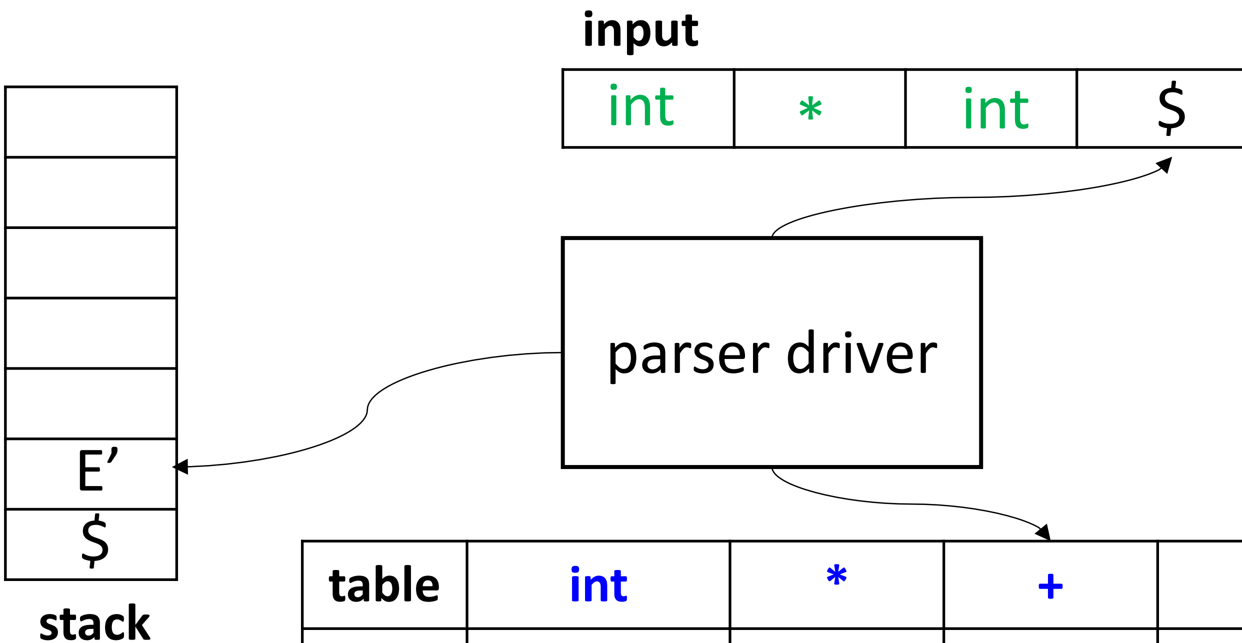
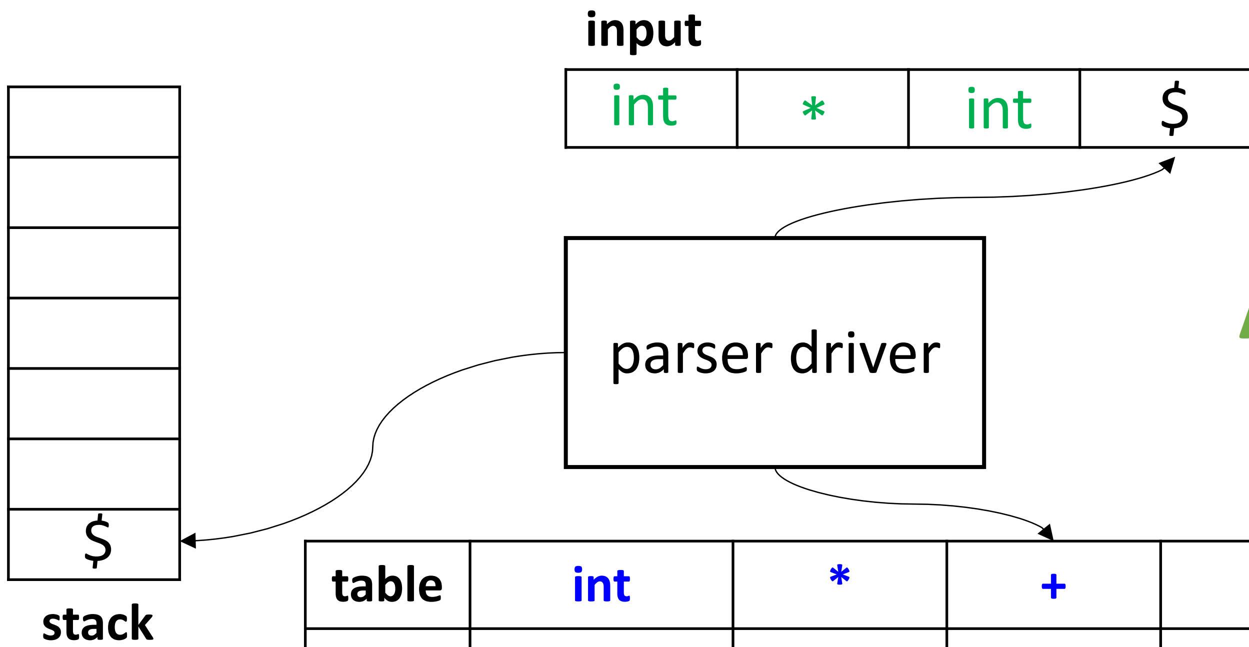


table	int	*	+	(	)	$
E	E → TE'			E → TE'		
E'			E' → +E		E' → ε	E' → ε
T	T → int T'			T → (E)		
T'		T' → *T	T' → ε		T' → ε	T' → ε

Use the Parse Table

- To recognize "int * int"



E → TE'

E'→ +E | ε

T → intT' | (E)

T'→ *T | ε

ACCEPT!

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

Recognize Sequence[解析过程]

Matched	Stack	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 and 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