



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8讲：语法分析(5)

张献伟

xianweiz.github.io

DCS290, 3/17/2022

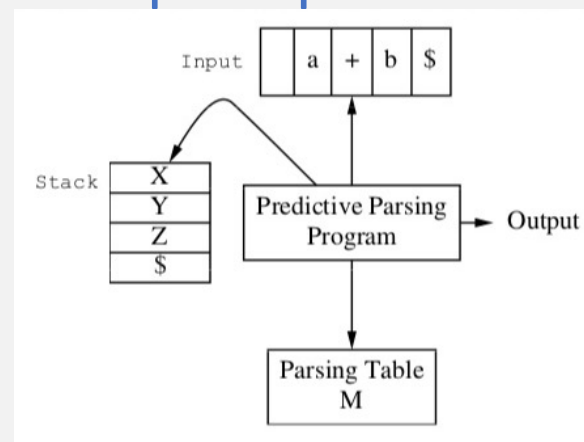


中山大學
SUN YAT-SEN UNIVERSITY



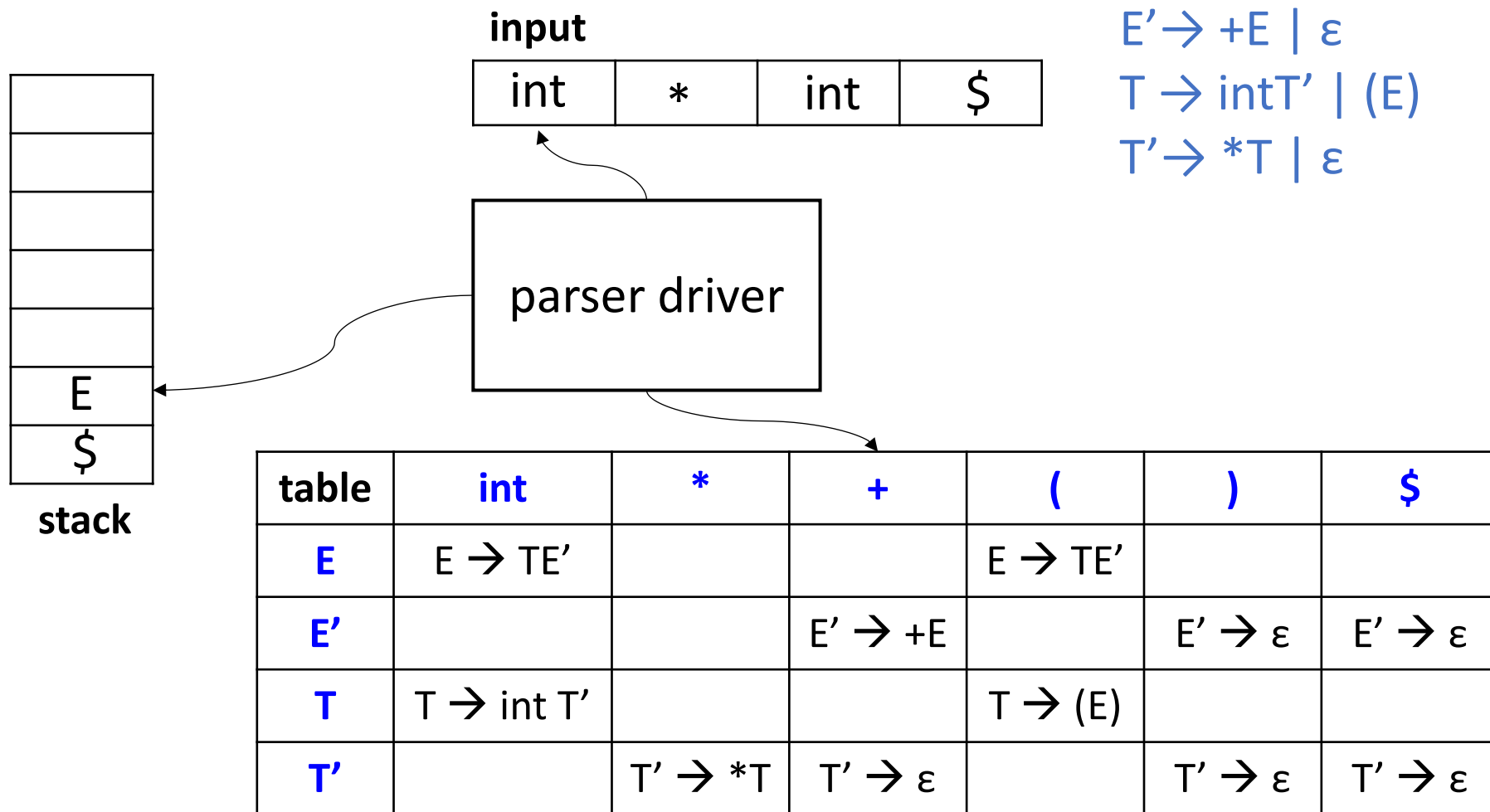
Review Questions (1)

- Q1: why do we prefer to use Predictive Parser?
Requires no backtracking, more efficient.
- Q2: how to predict the next production to use?
Current nonterminal being processed, next input symbol(s).
- Q3: can predictive parser handle $E \rightarrow E+T \mid \text{int} \mid \text{int}^*T$?
NO. Left recursion, common prefix.
- Q4: 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
- Q5: what is the initial state of the parser?
Input: input tokens followed by \$
Stack: start symbol followed by \$



Use the Parse Table

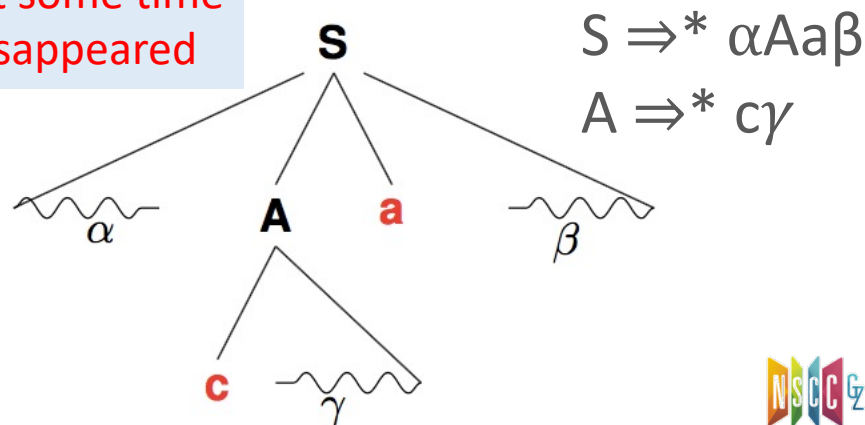
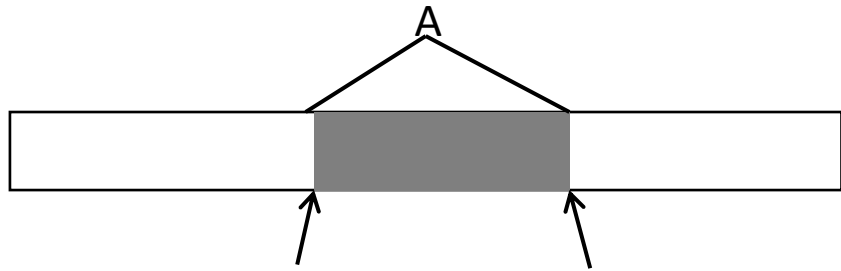
- To recognize “int * int”



To Construct Parse Table[构建解析表]

- 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(α)**: set of terminals that begin strings derived from α
 - E.g., $c \in \text{FIRST}(A)$ or $\text{FIRST}(Aa\beta)$
 - If $A \Rightarrow^* \epsilon$, then ϵ 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 symbol of a sentential form, $\$$ is also in $\text{FOLLOW}(A)$

There may have been symbols between A and a, at some time during derivation, but if so, they derived ϵ and disappeared



Use FIRST and FOLLOW

- Why do we need FIRST and FOLLOW in parsing?
 - FIRST and FOLLOW allow to choose which production to apply, based on the next input symbol
- FIRST[开始集]
 - $\text{FIRST}(\alpha)$: set of terminals that start strings derived from α
 - α : **any string** of grammar symbols
 - 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
 - A : **nonterminal**
 - If there's a derivation of A that results in ϵ
 - 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 $A \Rightarrow^* \epsilon$

 lookahead

Non-terminal A disappears, without consuming any input symbol.

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
 $\Rightarrow aBC$
 $\Rightarrow adBC$
 $\Rightarrow adC$ \times
 \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 ϵ 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 ϵ 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 [非终结符, 非空式]
 - Add a to $\text{FIRST}(X)$, if for some i , a is in $\text{FIRST}(Y_i)$, and ϵ 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 ϵ , then we add nothing more
 - But if $Y_1 \Rightarrow^* \epsilon$, then we add $\text{FIRST}(Y_2)$, and so on
 - Add ϵ to $\text{FIRST}(X)$, if ϵ is in $\text{FIRST}(Y_j)$ for all $j=1, 2, \dots, k$

如果 X 是非终结符，且有产生式形如 $X \rightarrow ABCdEF\dots$ (A, B, C 均为非终结符且包含 ϵ ， d 为终结符)，则需要把 $\text{FIRST}(A)$, $\text{FIRST}(B)$, $\text{FIRST}(C)$, d 加入到 $\text{FIRST}(X)$ 中

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- ϵ symbols to FIRST(α)[当然!]
 - Add FIRST(X_i) – ϵ , $2 \leq i \leq k$, to FIRST(α), if FIRST(X_1), ..., FIRST(X_{k-1}) all contain ϵ [前k-1个都透明]
 - Add non- ϵ symbols of FIRST(X_2), if ϵ is in FIRST(X_1)
 - Add non- ϵ symbols of FIRST(X_3), if ϵ is in FIRST(X_1) and FIRST(X_2)
 - ...
 - Add ϵ to FIRST(α), if FIRST(X_1), ..., FIRST(X_k) all contain ϵ [α 自身可以透明]

FOLLOW

- To compute FOLLOW(A) to all non-terminals A, apply following rules until no terminal or ϵ 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(β) except ϵ is in FOLLOW(B)
 - If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$, where FIRST(β) contains ϵ , then everything in FOLLOW(A) is in FOLLOW(B)
 - Namely, follow sets are defined in terms of follow sets
 - This step has to be repeated until follow sets converge

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 ϵ , 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 ϵ , 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	+,), \$

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



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

Construct LL(1) Parse Table[构建解析表]

- Goal: to put each production into the table entry
- To construct, rule $A \rightarrow \alpha$ is added to $M[A, a]$ if either:
 - For each terminal a in $FIRST(\alpha)$ [首先考虑RHS的FIRST集]
 - If ϵ is in $FIRST(\alpha)$, or $\alpha = \epsilon$, a is in $FOLLOW(A)$ (ϵ -production)[有空产生式, 同时考虑LHS的FOLLOW集]
 - If ϵ is in $FIRST(\alpha)$ and $\$$ is in $FOLLOW(A)$, 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
$T \rightarrow (E)$	(
$T' \rightarrow *T$	*
$E' \rightarrow \varepsilon$	FOLLOW
$T' \rightarrow \varepsilon$	FOLLOW

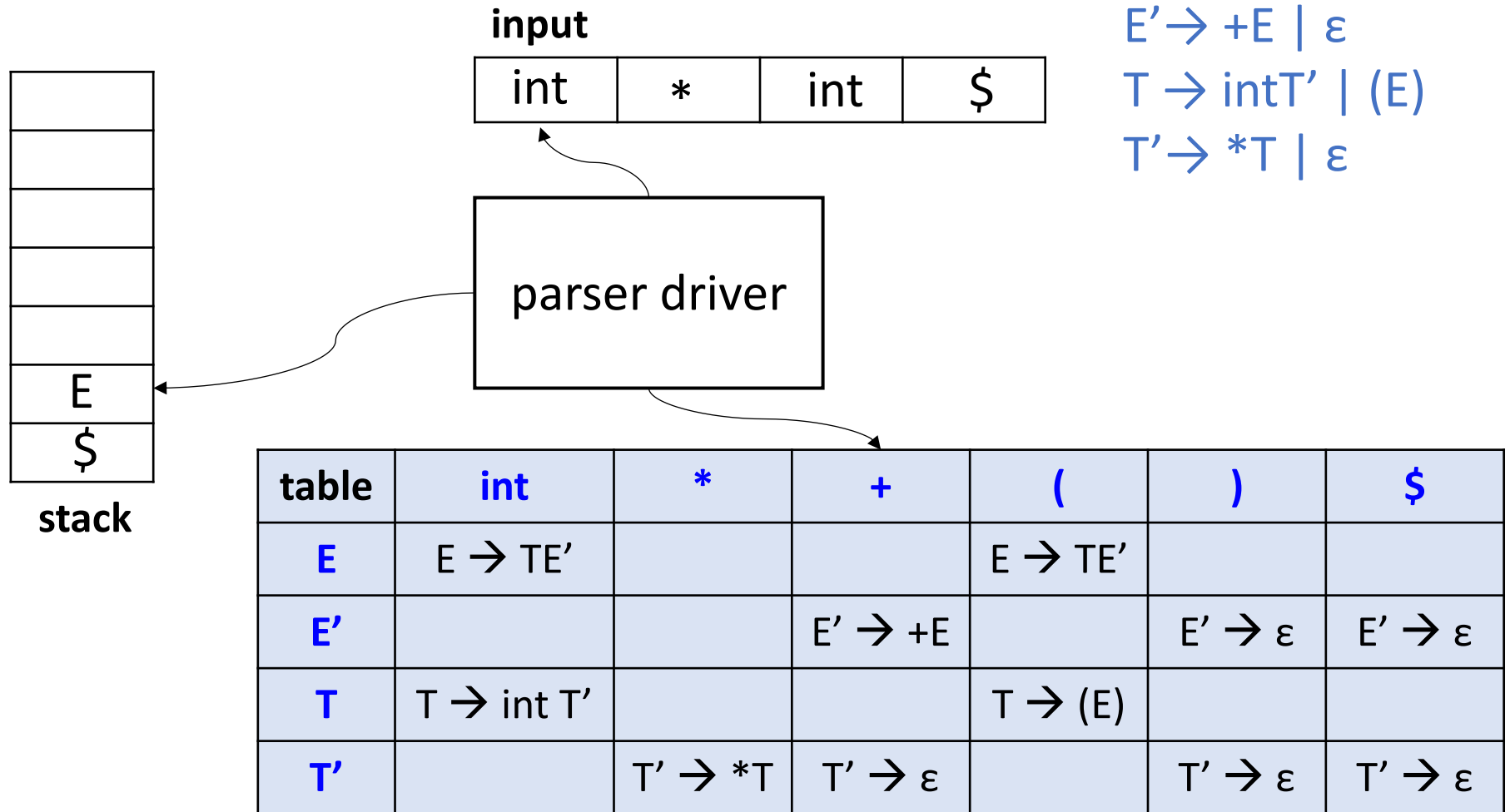
Symbol	FIRST	FOLLOW
E	int, (), \$
E'	+, ε), \$
T	int, (+,), \$
T'	*, ε	+,), \$

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

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 Table [already examined]

- To recognize “int * int”



$$E \rightarrow TE'$$

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

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

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

Determine If Grammar is LL(1)[判断]

- Observation[直观依据]

- 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)

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

- 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
- Check each rule as if the table is getting constructed

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

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

} 保证预测的唯一性

Non-LL(1) Grammars[非LL(1)文法]

- 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[复杂度]

- **Linear** time and space relative to length of input[线性]
- **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 ...

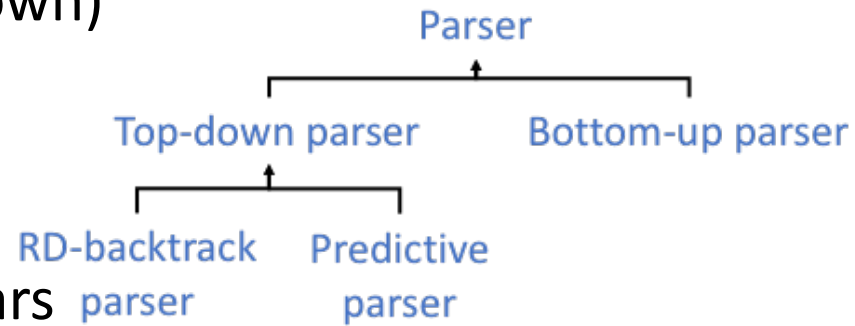
- LL(1) table-driven parser is basically DFA + Stack
 - Capable to count \Rightarrow CFG is more powerful than RE
- We have studied LL(1), what about LL(0), LL(2) or LL(k)?
- Is **LL(0)** useful at all?
 - Grammar where rules can be **predicted with no lookahead**
 - \Rightarrow That means, there can 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[小结]

- **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)

Bottom-up Parsing[自底向上]

- Begins at leaves and works to the top[叶子到根]
 - Bottom-up: **reduces**[归约] input string to start symbol
 - In the opposite direction from top-down
 - Top-down: expands start symbol to input string
 - In reverse order of rightmost derivation (In effect, builds tree from left to right, just like top-down)



- More powerful than top down
 - Don't need left factored grammars
 - Can handle left recursion
 - Can express a larger set of languages
 - And just as efficient

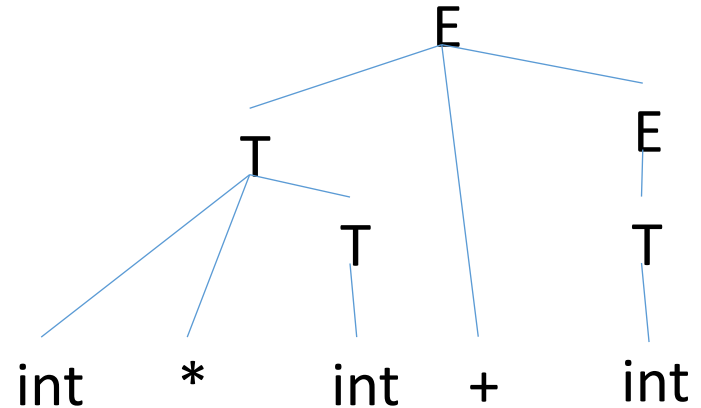
Example

- Grammar

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

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

- String: $\text{int} * \text{int} + \text{int}$



- The rightmost derivation of the parse tree

$$- E \Rightarrow T + E \Rightarrow T + T \Rightarrow T + \text{int} \Rightarrow \text{int} * T + \text{int} \Rightarrow \text{int} * \text{int} + \text{int}$$

- To recognize the string via bottom-up parsing

$$- \text{int} * \text{int} + \text{int} \Rightarrow \text{int} * T + \text{int} \Rightarrow T + \text{int} \Rightarrow T + T \Rightarrow T + E \Rightarrow E$$

Bottom-up: Overview

- An important fact:
 - Let $\alpha\beta\omega$ be a step of a bottom-up parse
 - Assume the next reduction is by $X \rightarrow \beta$
 - Then ω is a string of terminals [i.e., 句子]
- **Why?** $\alpha X\omega \rightarrow \alpha\beta\omega$ is a step in a rightmost derivation
- **Idea:** split string into two substrings
 - Right substring is as yet unexamined by parsing (a string of terminals)[右侧尚未处理]
 - Left substring has terminals and non-terminals[左侧已有处理]
- The dividing point is marked by a **#**
 - The **#** is not part of the string
 - Initially, all input is unexamined $\#x_1x_2 \dots x_n$

Bottom-up: Shift-Reduce[移入-归约]

- Bottom-up parsing is also known as **Shift-Reduce** parsing
 - Involves two types of operations: shift and reduce
- **Shift**[移入]: move # one place to the right
 - Shifts a terminal to the left string[向左侧移入终结符]
 $ABC\#xyz \Rightarrow ABCx\#yz$
- **Reduce**[归约]: apply an inverse production at the right end of the left string[左侧的右端进行规约]
 - If $E \rightarrow Cx$ is a production, then
 $ABCx\#yz \Rightarrow ABE\#yz$

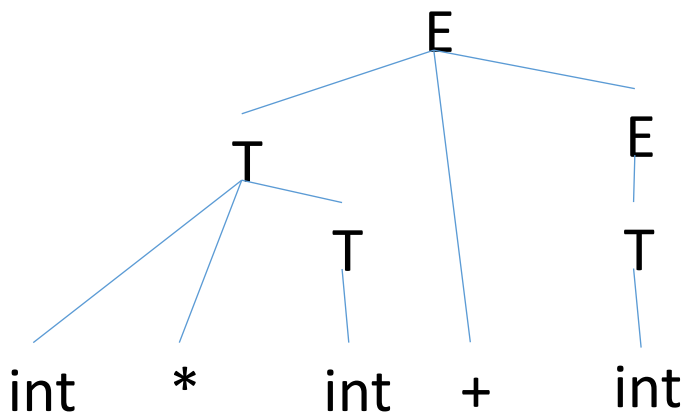
The Example

- Grammar

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

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

- String

$$\text{int} * \text{int} + \text{int}$$


Step	Operation
#int * int + int	Shift
int# * int + int	Shift
int * #int + int	Shift
int * int # + int	Reduce $T \rightarrow \text{int}$
int * T # + int	Reduce $T \rightarrow \text{int} * T$
T # + int	Shift
T + # int	Shift
T + int #	Shift
T + T #	Reduce $T \rightarrow \text{int}$
T + E #	Reduce $E \rightarrow T$
E #	Reduce $E \rightarrow T + E$

Stack[栈]

- Left string can be stored into a **stack**
 - Top of the stack is the **#**
- **Shift** pushes a terminal on the stack
- **Reduce** does the following:
 - pops zero or more symbols off of the stack
 - production rhs[pop出了产生式RHS]
 - pushes a non-terminal on the stack
 - production lhs[push进了产生式LHS]
 - just reverts production ($LHS \leftarrow RHS$)

Step
#int * int + int
int# * int + int
int * #int + int
int * int # + int
int * T # + int
T # + int
T + # int
T + int #
T + T #
T + E #
E #

Key Issue[一个关键问题]

- How to decide when to shift or reduce?

- Example grammar:

$E \rightarrow T + E \mid T$

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

#int * int + int	Shift
int# * int + int	Shift
int * #int + int	Shift
... ..	

#int * int + int	Shift
int# * int + int	Reduce $T \rightarrow \text{int}$
T # * int + int	Shift
... ..	

- Consider the step $\text{int} \# * \text{int} + \text{int}$

- We could reduce by $T \rightarrow \text{int}$ giving $T \# * \text{int} + \text{int}$

- **A fatal mistake:** no way to reduce to the start symbol E

- Intuition: want to reduce only if the result can still be reduced to the start symbol[必须在对的方向上]