Compilation Principle
编 译 原 理

第10讲：语法分析(6)

张献伟

xianweiz.github.io

DCS290, 4/2/2024

Grading[考核标准]

- **编译原理**
  - 课堂参与（15%）- 点名、提问、测试
  - 课程作业（25%）- 5次左右，理论
  - 期末考试（60%）- 闭卷
- **编译器构造实验**
  - 课堂参与（10%）- 签到、练习等
  - Project 1（20%）- Lexical Analysis
  - Project 2（20%）- Syntax/Semantic Analysis
  - Project 3（20%）- IR Generation
  - Project 4（30%）- Code Optimization

⚠️⚠️⚠️ proj1昨晚已经截止，我们会尽快完成批改！未提交的同学：如果截止前填写了问卷，自动宽限5天，请于4.2/周二23:59:59前提交（宽限期内扣除得分的5%），宽限期外每延一天再扣除得分的10%；没有填写问卷的同学，从今天开始每天扣除10%。proj2/3/4将类似执行，具体有所调整。

- 理论
  - 随机点名
    - 缺席优先
  - 随机提问
    - 后排优先
  - 随机测试
    - 不定时间
- 实验
  - 个人完成
    - 杜绝抄袭
  - 按时提交
    - 硬性截止
  - 侧重代码实现
    - 简略报告

Review Questions

- Q1: actions in top-down parsing?

  Expand on non-terminal, compare on terminal.

- Q2: how to build the LL(1) parse table?

  Two sets: FIRST, FOLLOW

- Q3: for the grammar, what is FIRST(A) and FOLLOW(B)?

  FIRST(A) = {b, d, ε}, FO(A) = {a} FOLLOW(B) = {d, a} = FI(D) + FO(A)

- Q4: which one is typically used, LL(0), LL(1), LL(2) …? Why not others?

  LL(1). LL(0) is too weak, LL(k) has a too large table.

- Q5: top-down vs. bottom-up parsing?

  Top-down is based on leftmost derivation;
  bottom-up is the reverse of rightmost derivation.

$S \rightarrow Aa$
$A \rightarrow BD$
$B \rightarrow b$
$B \rightarrow \varepsilon$
$D \rightarrow d$
$D \rightarrow \varepsilon$

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 $t \in \text{FIRST}(\alpha)$

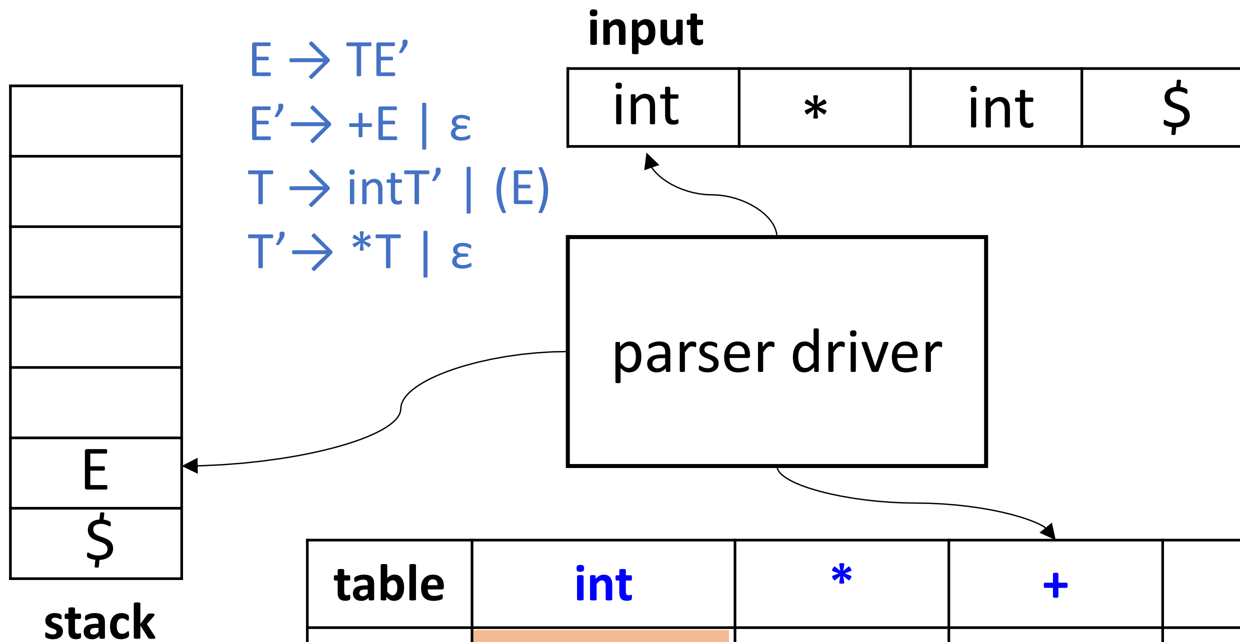
    OR
  - For non-terminal *A* and lookahead *t*, use the production rule $A \rightarrow \alpha$ where $\varepsilon \in \text{FIRST}(\alpha)$ and $t \in \text{FOLLOW}(A)$
  - There can only be ONE such rule
    - Otherwise, the grammar is not LL(1)

SUN YAT-SEN UNIVERSITY 中山大學

NSCC GZ

LL(1) Parser

- To recognize "int * int"



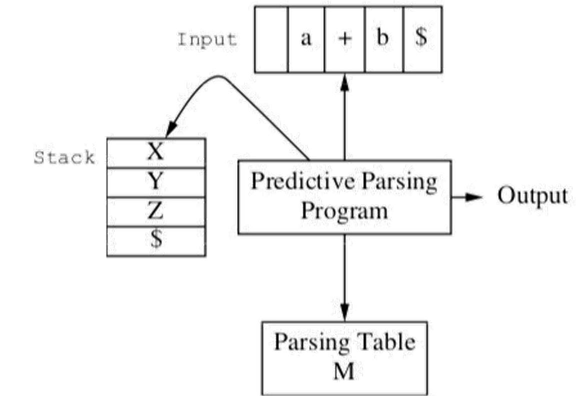


table	int	*	+	(	)	$
E	E → TE'			E → TE'		
E'			E' → +E		E' → ε	E' → ε
T	T → int T'			T → (E)		
T'		T' → *T	T' → ε		T' → ε	T' → ε

LL(1) Parse Table

A → α (RHS)	FIRST
E → TE’	int, (
E’ → +E	+
T → intT’	int
T → (E)	(
T’ → *T	*
E’ → ε	FOLLOW
T’ → ε	FOLLOW

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

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

Derivation

- Consider a CFG grammar G

  S→AB A→aC B→bD D→d C→c

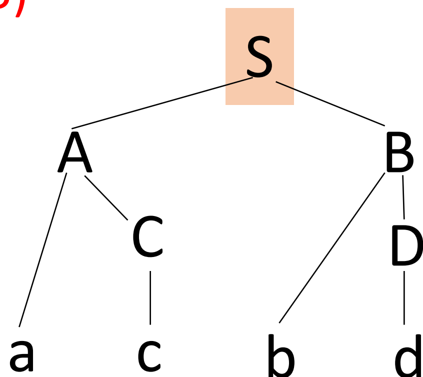
- This language has only one sentence: L(G) = {acbd}

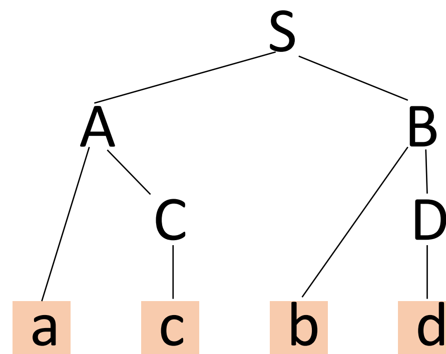
Top-down (leftmost derivation)

S ⇒ AB (1)
⇒ aCB (2)
⇒ acB (3)
⇒ acbD (4)
⇒ acbd (5)

Bottom-up (reverse of rightmost derivation)

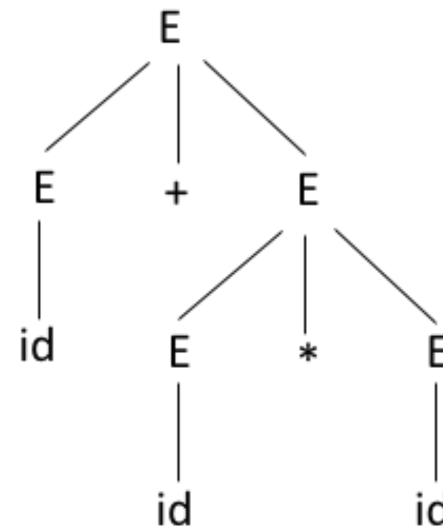
S ⇒ AB (5)
⇒ AbD (4)
⇒ Abd (3)
⇒ aCbd (2)
⇒ acbd (1)





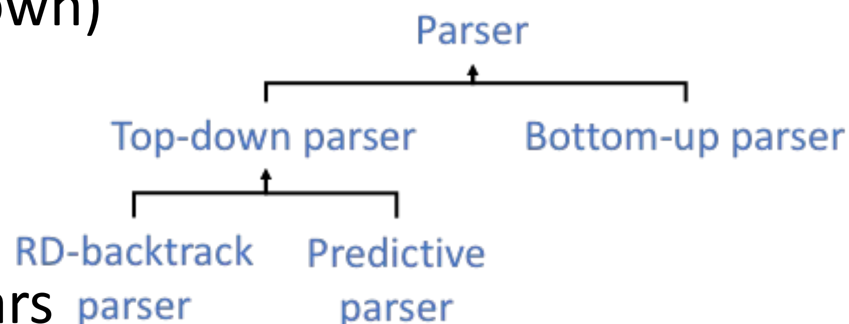
Parser Types[分析器类型]

- **Grammar** is used to derive string or construct **parser**
- Most compilers use either **top-down** or **bottom-up** parsers
- Top-down parsing[自顶向下分析]
  - Starts from root and expands into leaves
    - Tries to expand start symbol to input string
    - Finds leftmost derivation[最左推导]
  - In each step
    - Which non-terminal to replace?[哪个符号？]
    - Which production of the non-terminal to use?[哪个规则？]
  - Parser code structure closely mimics grammar
    - Amenable to implementation by hand
    - Automated tools exist to convert to code (e.g. ANTLR)



Bottom-up Parsing[自底向上]

- Begins at leaves and works to the top[叶子到根]
  - Bottom-up: **reduce**s[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

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

- To recognize "int * int"

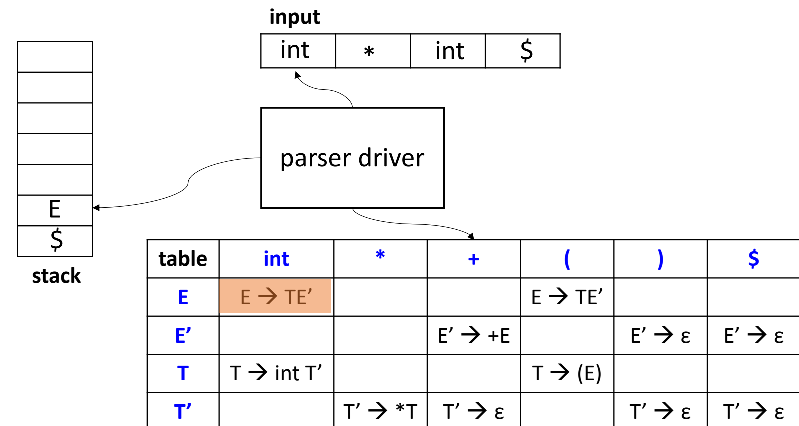


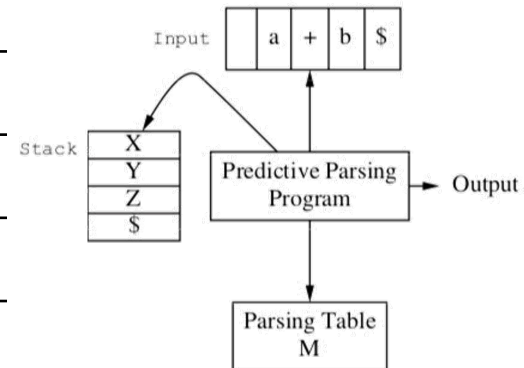
table	int	*	+	(	)	$
E	E → TE'			E → TE'		
E'			E' → +E		E' → ε	E' → ε
T	T → int T'			T → (E)		
T'		T' → *T	T' → ε		T' → ε	T' → ε

Recognize Sequence[解析过程]

Matched	Stack (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

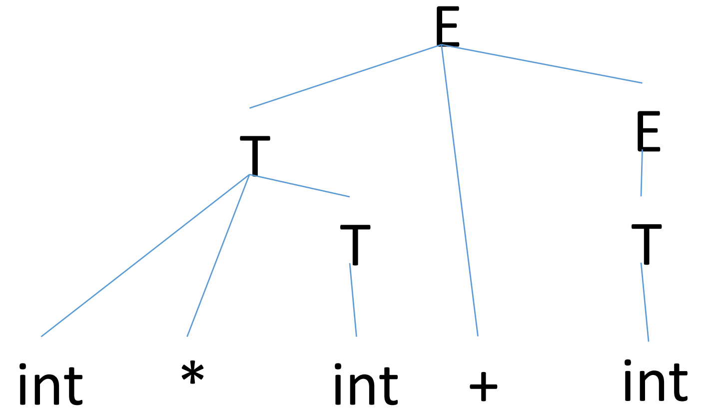
Example

- Grammar

  E → T+E | T

  T → int*T | int | (E)

- String: int * int + int



- The rightmost derivation of the parse tree
  - E ⇒ T + E ⇒ T + T ⇒ T + int ⇒ int * T + int ⇒ int * int + int

- To recognize the string via bottom-up parsing
  - int * int + int ⇒ int * T + int ⇒ T + int ⇒ T + T ⇒ T + E ⇒ E

Example (cont.)

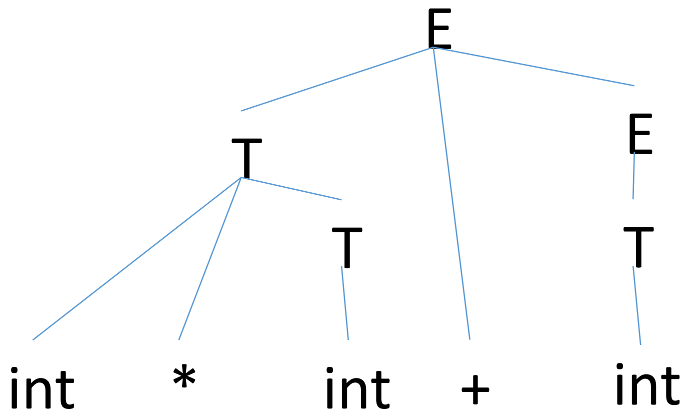
- Grammar

  E → T+E | T

  T → int*T | int | (E)

- String

  int * int + int



Step	Operation
# int * int + int	Shift
int # * int + int	Shift
int * # int + int	Shift
int * int # + int	Reduce T → int
int * T # + int	Reduce T → int*T
T # + int	Shift
T + # int	Shift
T + int #	Reduce T → int
T + T #	Reduce E → T
T + E #	Reduce E → T+E
E #	

Top-down vs. Bottom-up

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

Top-down

Step	Operation
# int * int + int	Shift
int # * int + int	Shift
int * # int + int	Shift
int * int # + int	Reduce T → int
int * T # + int	Reduce T → int*T
T # + int	Shift
T + # int	Shift
T + int #	Reduce T → int
T + T #	Reduce E → T
T + E #	Reduce E → T+E
E #	

Bottom-up

Bottom-up: Overview

- An important fact:
  - Let αβω be a step of a bottom-up parse[bottom-up解析过程中的一步]
  - Assume the next reduction is by X → β[下一步，把β归约到X]
  - Then ω is a string of terminals[i.e., ω一定是句子]
- Why? αXω → αβω 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)[右侧尚未被解析，i.e., 最右推导中已被完全展开]
  - Left substring has terminals and non-terminals[左侧已有解析，i.e., 最右推导中尚未被完全展开]
- The dividing point is marked by a #
  - The # is not part of the string[仅作为标示]
  - Initially, all input is unexamined #$x_1x_2 \dots x_n$[输入尚未有任何解析]

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
  - Recall: expand and compare operations for top-down parsing

- **Shift**[移入]: move # one place to the right
  - Shifts a terminal to the left string[向左侧移入终结符，推进解析]

    ABC#xyz ⇒ ABCx#yz

- **Reduce**[归约]: apply an inverse production at the right end of the left string[左侧串的右端进行归约]
  - If E → Cx is a production, then

    ABCx#yz ⇒ ABE#yz

Stack[栈]

- Left string can be stored into a **stack**
  - Top of the stack is the #[左右串的分界]
- **Shift** pushes a terminal on the stack
- **Reduce** does the following:
  - pops zero or more symbols off 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 ← RHS)[产生式逆向使用]

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 → T+E | T

    T → int*T | int | (E)

#int * int + int	Shift
int# * int + int	Shift
int * #int + int	Shift
… … …	✓

#int * int + int	Shift
int# * int + int	Reduce T → int
T # * int + int	Shift
… … …	✗

  - Consider the step int # * int + int
  - We could reduce by T → int giving T#*int + 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]

The Example

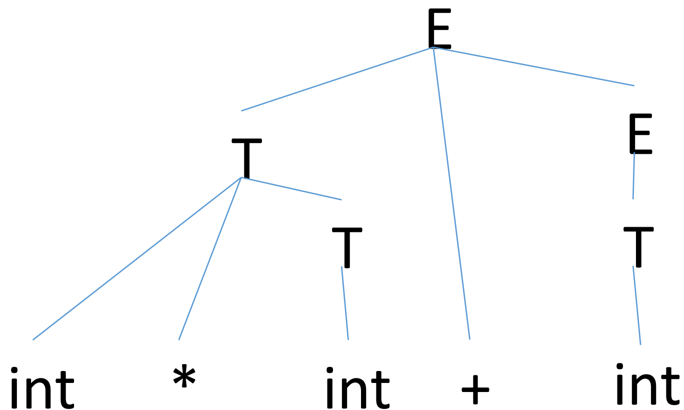
- Grammar

  E → T+E | T

  T → int*T | int | (E)

- String

  int * int + int



Step	Operation
# int * int + int	Shift
int # * int + int	Shift
int * # int + int	Shift
int * int # + int	Reduce T → int
int * T # + int	Reduce T → int*T
T # + int	Shift
T + # int	Shift
T + int #	Reduce T → int
T + T #	Reduce E → T
T + E #	Reduce E → T+E
E #	

Why not just E → T?
(int * T => int * E)

Handle[句柄]

- A **handle** of a sentential form is a substring α such that:
  - *α* matches the RHS of a production *A -> α* ; and[能匹配上规则]
  - replacing *α* by the LHS *A* represents a step in the reverse of a rightmost derivation of S[且能推进解析]
- Definition: let αβω be a sentential form where:
  - α, β is a string of terminals and non-terminals (yet to be derived)
  - ω is a string of terminals (already derived)
  - Then β is a **handle** of αβw if: $S \Rightarrow^*_{rm} \alpha X \omega \Rightarrow \alpha\beta\omega$ by a rightmost/rm derivation (apply rule X→β)
- ☞ We only want to reduce at handles, and there is exactly one handle per sentential form
  - But where to find it?

Some Concepts[一些概念]

- A **right-sentential form**[最右句型] is a sentential form that occurs in the rightmost derivation of some sentence
- A **phrase**[短语] is a subsequence of a sentential form that is eventually "reduced" to a single non-terminal[能归约到单个非终结符]
  - β is a phrase of the right sentential form γ iff $S \Rightarrow^*_{rm} \gamma = \alpha_1 A \alpha_2 \Rightarrow^+ \alpha_1 \beta \alpha_2$
  - a string consisting of all of the leaves of the partial parse tree that is rooted at one particular internal node of the whole parse tree[一个句型的语法树中任一子树叶结点所组成的符号串都是该句型的短语]
- A **simple phrase**[直接短语] is a phrase that is reduced in a single step
  - β is a simple phrase of the right sentential form γ iff $S \Rightarrow^*_{rm} \gamma = \alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2$
- The **handle** is the leftmost simple phrase[最左直接短语]

Handle: Example

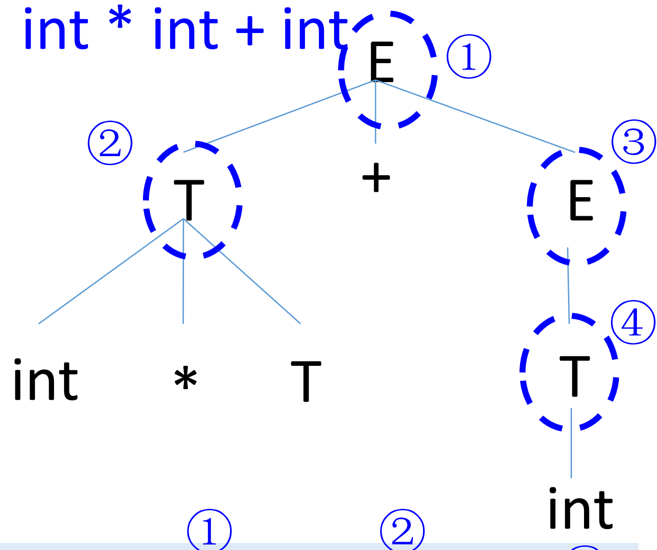
- Grammar

  E → T+E | T

  T → int*T | int | (E)

- String

  int * int + int



Phrase: int * T + int, int * T, int

Simple phrase: int * T, int

Handle: int * T

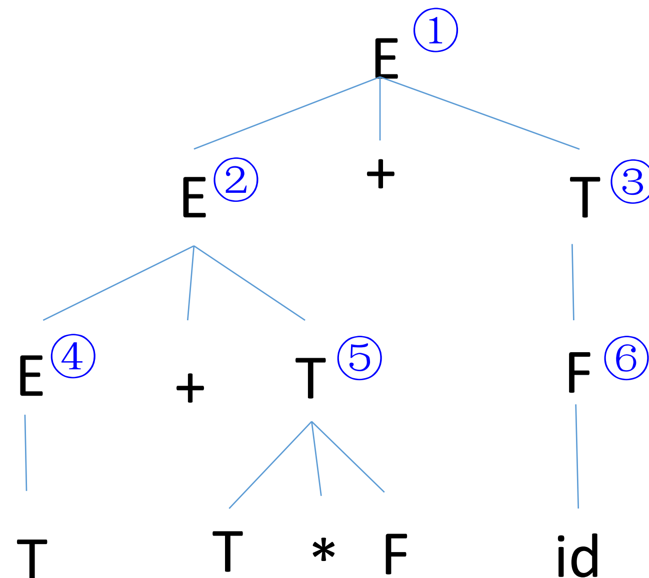
Step	Operation
# int * int + int	Shift
int # * int + int	Shift
int * # int + int	Shift
int * int # + int	Reduce T → int
int * T # + int	Reduce T → int*T
T # + int	Shift
T + # int	Shift
T + int #	Reduce T → int
T + T #	Reduce E → T
T + E #	Reduce E → T+E
E #	

One More Example

- Grammar
  - E → E+T|T
  - T → T*F | F
  - F → (E) | id
- Sentential form:
  - T + T*F + id
- Phrase：
  - T+T*F+id①
  - T+T*F②
  - T④
  - T*F⑤
  - id③⑥

Handle

Simple phrase



中山大學 SUN YAT-SEN UNIVERSITY

NSCC GZ

Handle Always Occurs at Stack Top

- Why can't a handle occur on right side of #?[不在栈外]
  - It can
  - But handle will eventually be shifted in, placing it at top of stack
  - In int * # int + int ⇒ int * int # + int, int is eventually shifted to the top (i.e., left to #)
- Why can't a handle occur on left side of #, i.e., in middle of the stack?[不在栈'中']
  - Can int * int + # int occur? NOPE.
  - Means parser shifted when it could have reduced when the handle was on top[本应归约却移入]
  - If eagerly reduces when handle is at top of stack, never occurs

int * # int + int	Shift
int * **int** # + int	Reduce T → int
int * T # + int	Reduce T → int*T

- ☞ Makes life easier for parser (need only access top of stack)

Viable Prefix[活前缀]

- In shift-reduce parsing, the stack contents are always a **viable prefix**[活前缀/可动前缀]
  - A prefix of some right-sentential form that ends no further right than the end of the handle of that right-sentential form
    - A viable prefix has a handle at its rightmost end
  - Stack content is always a viable prefix, guaranteeing the shift / reduce is on the right track[活前缀说明移进归约是正确的]
- 定义：一个可行前缀是一个最右句型的前缀，并且它没有越过该最右句型的最右句柄的右端
  - 举例：S => bBa => bbAa，这里句柄是 bA，因此可行前缀包括 bA 的所有前缀（包括 b, bb, bbA），但不能是 bbAa（因为越过了句柄）

... => T + int => int * T + int
Handle: int * T
Viable prefix: int, int *, int * T

int * # int + int	Shift
int * int # + int	Reduce T → int
int * T # + int	Reduce T → int*T
T # + int	Shift