编译原理
Complier Principles

Lecture 4
Syntax Analysis: Top-Down

赵帅

计算机学院

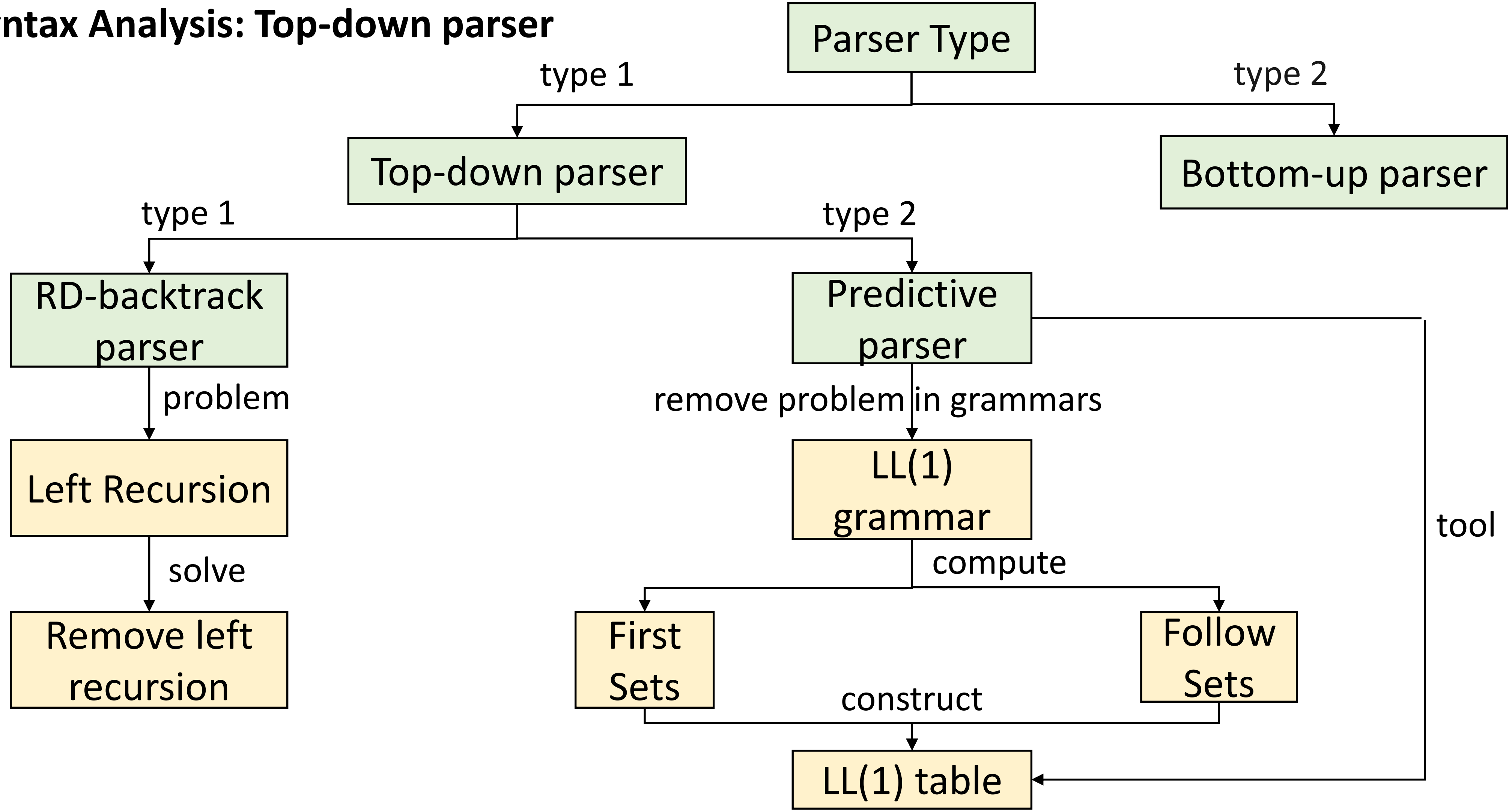
中山大学

Before we start...

So, please pay attention and catch up!

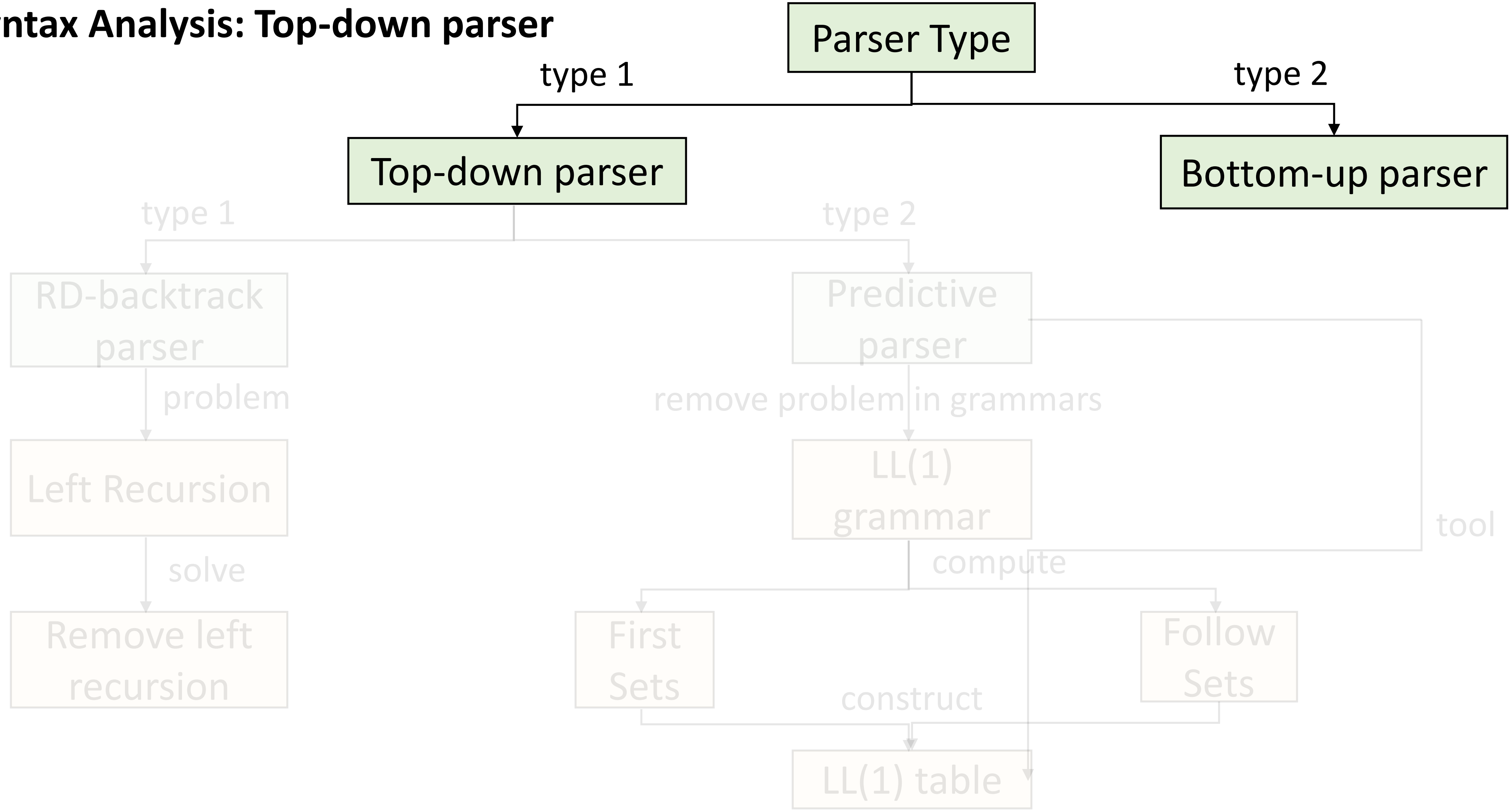
Mind Map[思维导图]

Syntax Analysis: Top-down parser



Mind Map[思维导图]

Syntax Analysis: Top-down parser



Parser Type[语法分析类型]

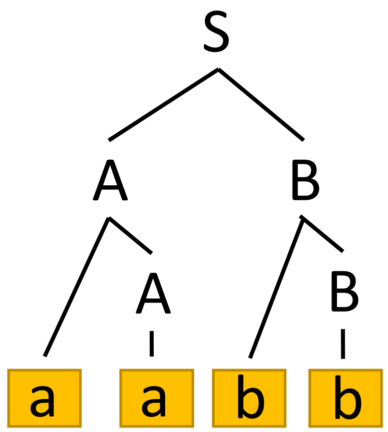
- Most compilers use either **Top-Down** or **Bottom-Up** parsers.
- Bottom-up parsing [自底向上分析]
  - ◆ Begin at the leaves (*the bottom*) and working up towards the root (*the top*).
  - ◆ Tries to reduce[规约] the input string to the start symbol.
  - ◆ Finds reverse order of the rightmost derivation[最右推导的逆过程即最左规约]
    规范推导 规范规约
  - ◆ Parser code structure nothing like grammar.
    - ▫ Very difficult to implement manually.
    - ▫ Automated tools exist to convert to code (e.g., Yacc, Bison).

Parser Type[语法分析类型]

◆Example

- Grammar G(S): S → AB ; A → aA | a ; B → bB | b ;
- **Language?** **$L(G)=\{a^m b^n | m,n \geq 1\}$**
- Sentence: aabb;

S ⇒ AB	(5) *B* reduce to *S*.
⇒ AbB	(4) *bB* reduce to *B*.
⇒ Abb	(3) 2nd *b* reduce to *B*.
⇒ aAbb	(2) *aA* reduce to *A*.
⇒ aabb	(1) 2nd *a* reduce to *A*.



<u>Reverse of rightmost derivation!</u>

Parser Type[语法分析类型]

- Top-Down parsing [自顶向下分析]
  - Starting from the root (*top*) and create the leaves (*down*) of the parse tree in a pre-defined order(depth-first)[深度优先,先根次序/前序].
  - Top-down parsing can be viewed as **finding a leftmost derivation**[寻求最左推导] for an input string. ***Why not rightmost or arbitrary derivation?***
  - **Review:** *In each step of derivation*, the following choices need to be made:
    - Choice of the non-terminal to be replaced. [替换哪个非终结符] **Leftmost!**
    - Choice of the production to be applied for a non-terminal. [使用文法中哪个规则来替换] **Key Problem!**
  - **Question:** *At each step of a top-down parse*, What is the key problem?

Parser Type[语法分析类型]

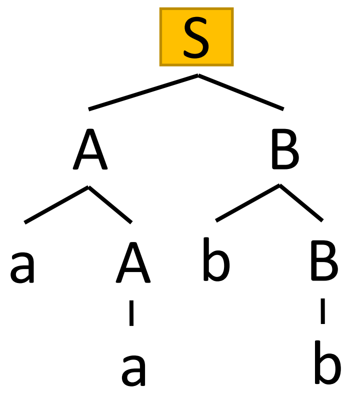
- Top-Down parsing [自顶向下分析]
  - Once a production is chosen, we try to match the terminal symbols in the production body with the input string.
  - Parser code structure closely mimics grammar.
    - Manually implementation is feasible.
    - Automated tools exist to convert to code. (e.g. ANTLR)
- Top-Down **vs.** Bottom-Up [对比]
  - Top-down: easier to understand and implement manually. (E.g. ANTLR)
  - Bottom-up: more powerful, can be implemented automatically. (E.g. YACC/Bison)

Parser Type[语法分析类型]

◆Example

- ◆Grammar G(S): S → AB ; A → aA | a ; B → bB | b ;
- ◆Sentence: aabb;

$S \Rightarrow AB$ (1)
$\Rightarrow aAB$ (2)
$\Rightarrow aaB$ (3)
$\Rightarrow aabB$ (4)
$\Rightarrow aabb$ (5)



$S \Rightarrow AB$ (5) *B* reduce to *S*.
$\Rightarrow AbB$ (4) *bB* reduce to *B*.
$\Rightarrow Abb$ (3) 2^{nd} *b* reduce to *B*.
$\Rightarrow aAbb$ (2) *aA* reduce to *A*.
$\Rightarrow aabb$ (1) 2^{nd} *a* reduce to *A*.

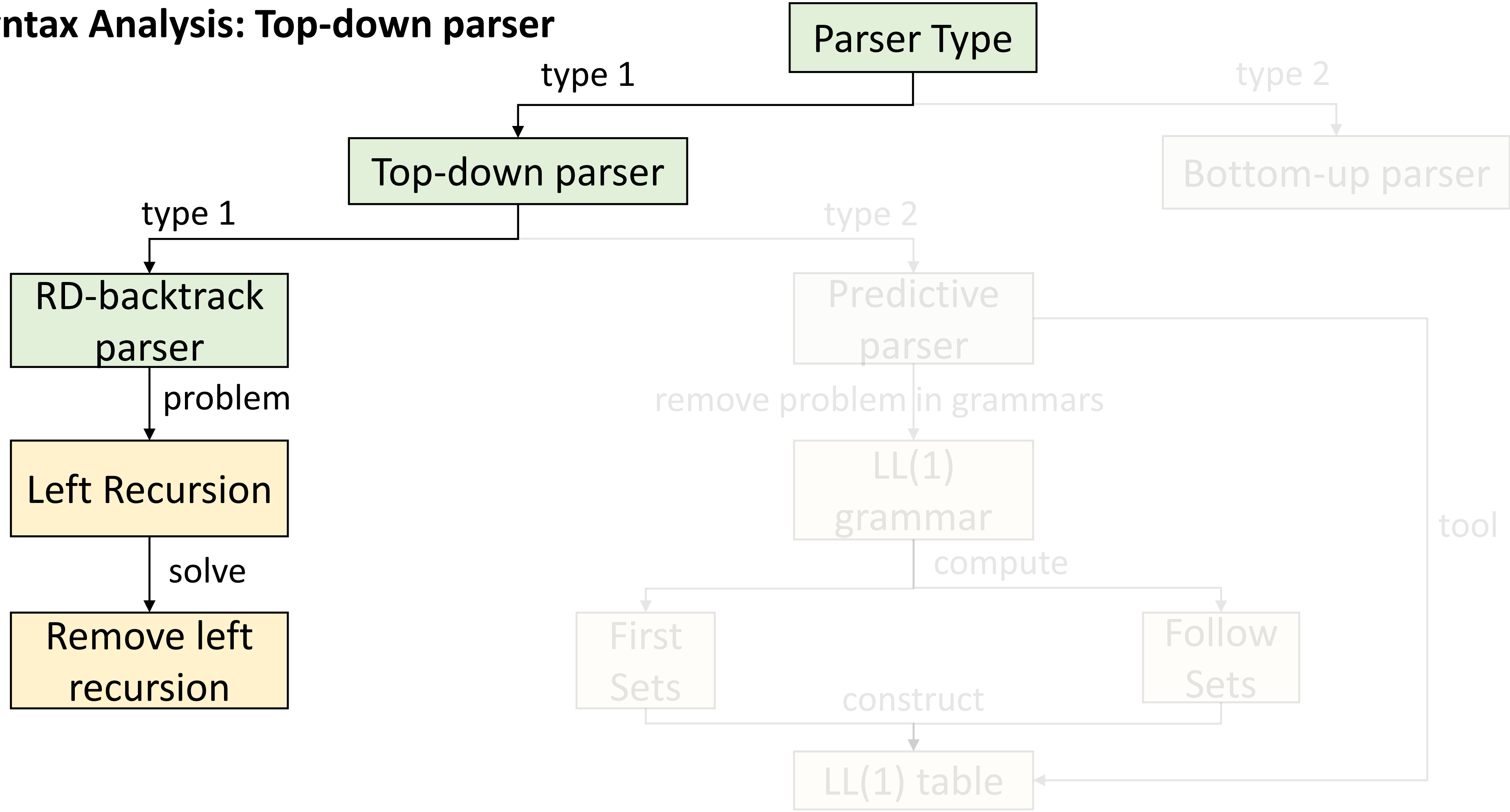
Leftmost derivation ***Leftmost reduction***

Top-Down ***Bottom-Up***

9

Mind Map[思维导图]

Syntax Analysis: Top-down parser



Top-down Parsing[自顶向下分析]

SUN YAT-SEN UNIVERSITY

- **Recursive-descent parsing**[RDP, 递归下降语法分析]
  - A general form[通用形式] of top-down parsing.
  - A recursive-descent parsing program consists of a set of *procedures*, one for each non-terminal.
  - Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string.

```
void A() {
    Choose an A-production, A → X1 X2 ··· Xk;
    for ( i = 1 to k ) {
        if ( Xi is a nonterminal )
            call procedure Xi();
        else if ( Xi equals the current input symbol a )
            advance the input to the next symbol;
        else /* an error has occurred */;
    }
}
```

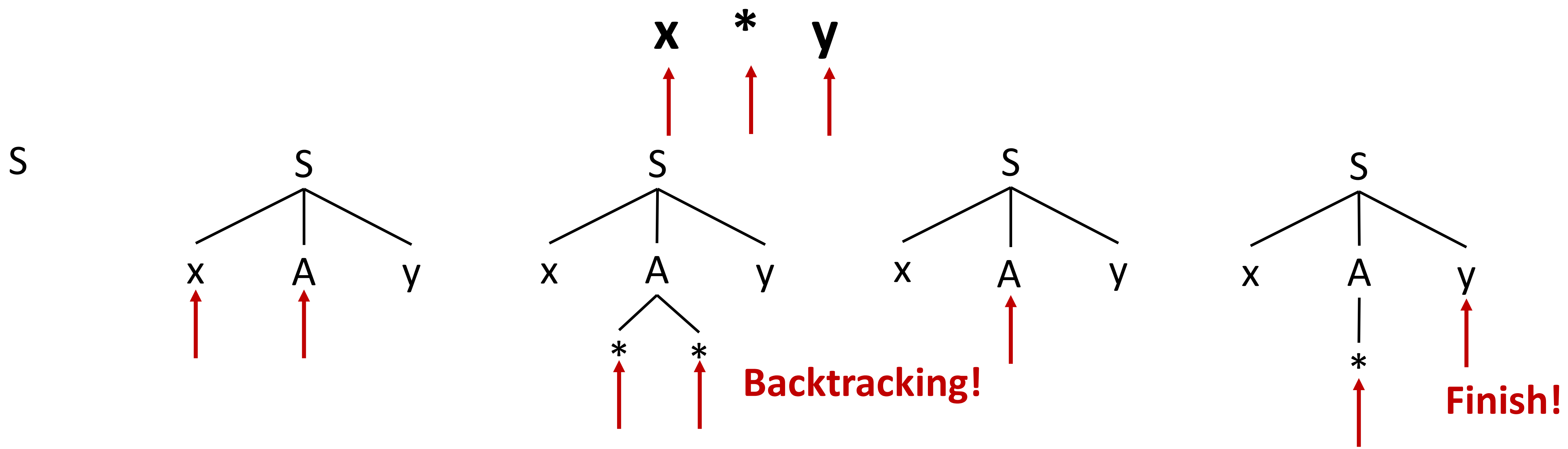
How to choose A-production?
It's not specified, so the pseudocode is nondeterministic[伪代码是不确定的].

RDP with backtracking[回溯]

- RDP may require ***backtracking***.
- **Approach**: for a non-terminal in the derivation, productions are tried in some order until
  - A production is found that generates a portion of the input, or
  - No production is found that generates a portion of the input, in which case backtrack to previous non-terminal.
- Terminals of the derivation are compared against input
  - Match: advance input, continue parsing
  - Mismatch: backtrack, or fail
- Parsing fails if no derivation generates the entire input.

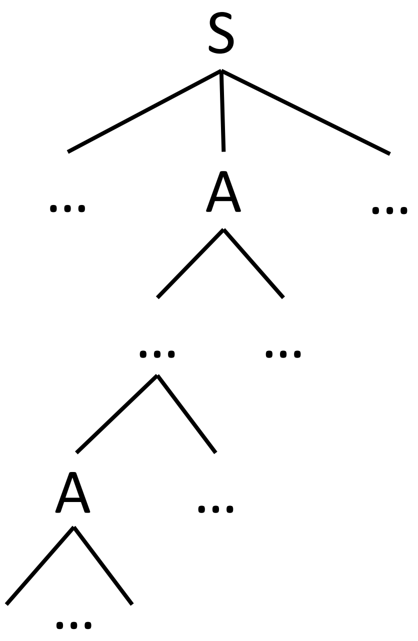
RDP with backtracking[回溯]

- In the analysis process, when a non-terminal is successfully matched with an alternative, **the match may be temporary.**
- If a mismatch occurs, backtracking[回溯] will be performed.
- G[S]: S → xAy; A → ** | *, whether the input string **x * y** is its sentence?



Left Recursion Problem[左递归问题]

- A grammar is left recursive[左递归] if it has a non-terminal **A** such that there is a derivation $\mathbf{A} \overset{+}{\Rightarrow} \mathbf{A\alpha}$.
- Sentence can grow infinitely without consuming input.(Into an infinite loop!)
- Top-down parsing methods cannot handle left-recursive problems.[自顶向下语法分析方法不能处理左递归的文法]



Left Recursion Problem[左递归问题]

- Immediate left recursion [直接/立即左递归]
  - There is a production $A \rightarrow A\alpha$.
- Non-immediate left recursion [间接/非立即左递归]
  - Left recursion involving derivation of 2+ step.
  - $A \rightarrow B\beta$; $B \rightarrow A\alpha$.
- A transformation is needed to *eliminate left recursion*. [需要一个转换方法来消除左递归]
- Rewrite the grammar so that it is right recursive. [改为右递归]

Remove Left Recursion[消除左递归]

- Immediate left recursion[直接左递归的消除]
  - Grammar: $A \rightarrow A\alpha \mid \beta$ ($\alpha \neq \beta$, β doesn't start with A)
  - *rewrite the rule of A as the following form equivalently:*
  - Grammar: $A \rightarrow \beta A'$; $A' \rightarrow \alpha A' \mid \varepsilon$ **(right recursion)**

G[A]: $A \rightarrow A\alpha \mid \beta$

$A \Rightarrow A\alpha$
$\Rightarrow A\alpha\alpha$
$\Rightarrow A\alpha\alpha\alpha$
……
$\Rightarrow A\alpha\ldots\alpha$
$\Rightarrow \beta\alpha\ldots\alpha$

Remove Left Recursion →

G[A]: $A \rightarrow \beta A'$
$A' \rightarrow \alpha A' \mid \varepsilon$

$A \Rightarrow \beta A'$
$\Rightarrow \beta\alpha A'$
$\Rightarrow \beta\alpha\alpha A'$
……
$\Rightarrow \beta\alpha\ldots\alpha A'$
$\Rightarrow \beta\alpha\ldots\alpha$

Remove Left Recursion[消除左递归]

中山大學 SUN YAT-SEN UNIVERSITY

- **Immediate left recursion** can be eliminated by the following technique, which works for any number of A-productions.
  - First, group the productions as

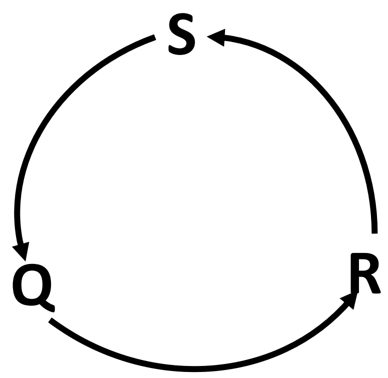
    $\mathbf{A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \ \ldots \mid \beta_n}$ where no β_i begins with an A.
  - Then, replace the A-productions by

    $\mathbf{A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'}$ **AND** $\mathbf{A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon}$
- Exercise-Remove Immediate left recursion.
  - $G_1[E]$: $E \rightarrow \underbrace{\boxed{E+T}}_{\alpha} \mid \underbrace{\boxed{T}}_{\beta}$; $T \rightarrow \underbrace{\boxed{T*F}}_{\alpha} \mid \underbrace{\boxed{F}}_{\beta}$; $F \rightarrow (E) \mid i$.
  - $G_2[E]$: $\mathbf{E \rightarrow TE'; E' \rightarrow +TE' \mid \varepsilon;}$ $\mathbf{T \rightarrow FT'; T' \rightarrow *FT' \mid \varepsilon;}$ $F \rightarrow (E) \mid i$.

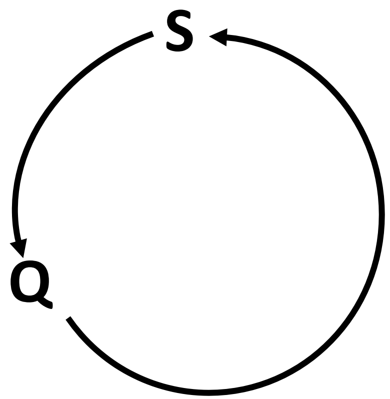
Remove Left Recursion[消除左递归]

- **Non-Immediate left recursion**[非直接左递归的消除]
  - Grammar: **S → Qc | c ; Q → Rb | b ; R → Sa | a.**
  - Although there is no immediate left recursion, **S, Q and R are all left recursion.**

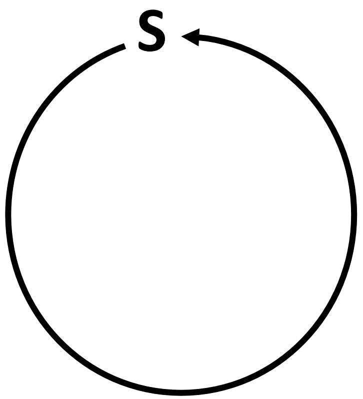
Left Recursion!



S → Qc | c
Q → Rb | b
R → Sa | a



S → Qc | c
Q → Sab | ab | b
R → Sa | a



S → Sabc | abc | bc | c
Q → Sab | ab | b
R → Sa | a

Remove Left Recursion[消除左递归]

- The following algorithm systematically eliminates left recursion from a grammar[直接/间接]. It is guaranteed to work if:
  - the grammar has no cycles (derivations of the form $A \stackrel{+}{\Rightarrow} A$)
  - the grammar has no ε–productions (productions of the form $A \rightarrow ε$).
  - These two can be eliminated systematically from a grammar.
- **Algorithm:** Eliminating left recursion.
  - **INPUT:** Grammar *G* with no cycles or ε–productions.
  - **OUTPUT:** An equivalent grammar with no left recursion.
  - **METHOD:** Apply the following 3 steps to G. Note that the resulting non-left-recursive grammar may have ε–productions.

Remove Left Recursion[消除左递归]

- **Step 1 :** Arrange all non-terminals of grammar G in some order $A_1, A_2, \ldots, A_n$;
- **Step 2 :** *Execute in order obtained in Step 1:*

FOR **i:=1** TO **n** DO

  FOR **j:=1** TO **i-1** DO

    Replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \delta_k\gamma$, where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all current A_j-productions;

    eliminate the immediate left recursion among the A_i-productions;

  END

END

- **Step 3 :** Simplify the grammar obtained from Step 2 --- remove the production rules of non-terminal that can never be reached from the start symbol.

Remove Left Recursion[消除左递归]

- **Example:** Consider Grammar G(S): (**ILR = immediate left recursion**)

  R → Sa | a Q → Rb | b S → Qc | c

- **Step 1 :** non-terminal order: **R,Q,S**; (A_1, A_2, A_3)
- **Step 2 :** (i=1) For R, there is no **ILR**;

  (i=2,j=1) Replace production Q → Rb by the productions R → Sa | a, which generates Q → Sab | ab | b; ***not ILR***

  (i=3,j=1) No operations.

  (i=3,j=2) Replace production S → Qc by the productions Q → Sab | ab | b, which generates S → Sabc | abc | bc | c; ***contain ILR;***

  (continue...)

Remove Left Recursion[消除左递归]

(continue...)

(α)　　(β)

(i=3,j=2) S → Sabc | abc | bc | c　　contains ILR.

Eliminate ILR about S:

S → abcS' | bcS' | cS'　　　S' → abcS' | ε

Q → Sab | ab | b　　　R → Sa | a

- **Step 3 :** Simplify the grammar and get **Final Grammar**:

S → abcS' | bcS' | cS'　S' → abcS' | ε

(Q&R's production is included by S)

Remove Left Recursion[消除左递归]

- **Question: What will happen if the order in step 1 is different**
- Again, consider G(S): S → Qc | c   Q → Rb | b   R → Sa | a
- **Exercise:** If the order in step 1 is **S,Q,R**, the final grammar without ILR is?

(was R, Q, S in the previous example)

S,Q,R		R,Q,S
S → Qc | c Q → Rb | b R → bcaR’ | caR’ | aR’ R’ → bcaR’ | ε	**equivalent** ⟷	S → abcS’ | bcS’ | cS’ S’ → abcS’ | ε

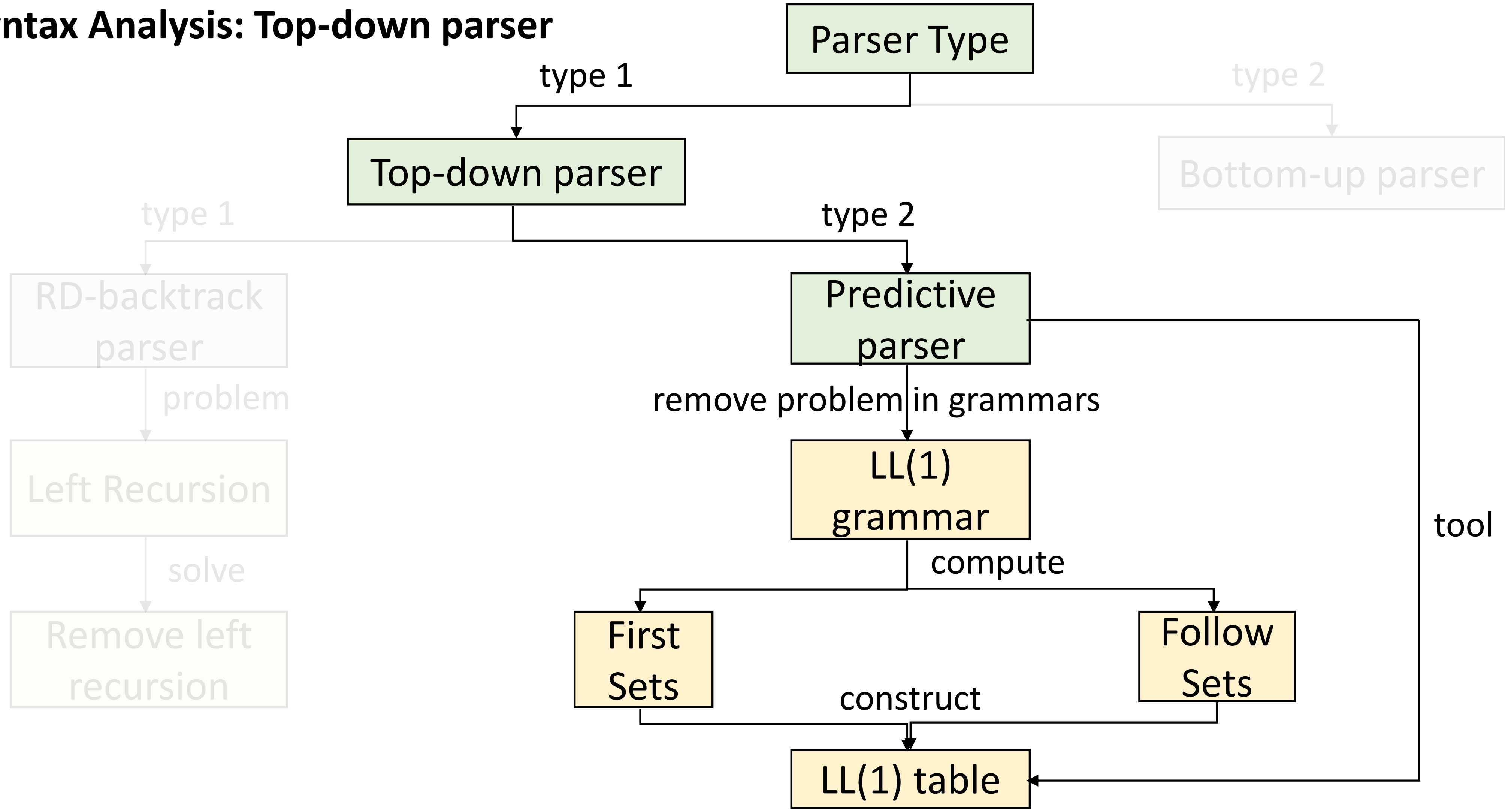
- Different order in Step1 may cause the final grammar different in form[可能在形式上不同], but it is not difficult to prove that they are equivalent.

Recursive-descent parsing[递归下降语法分析]

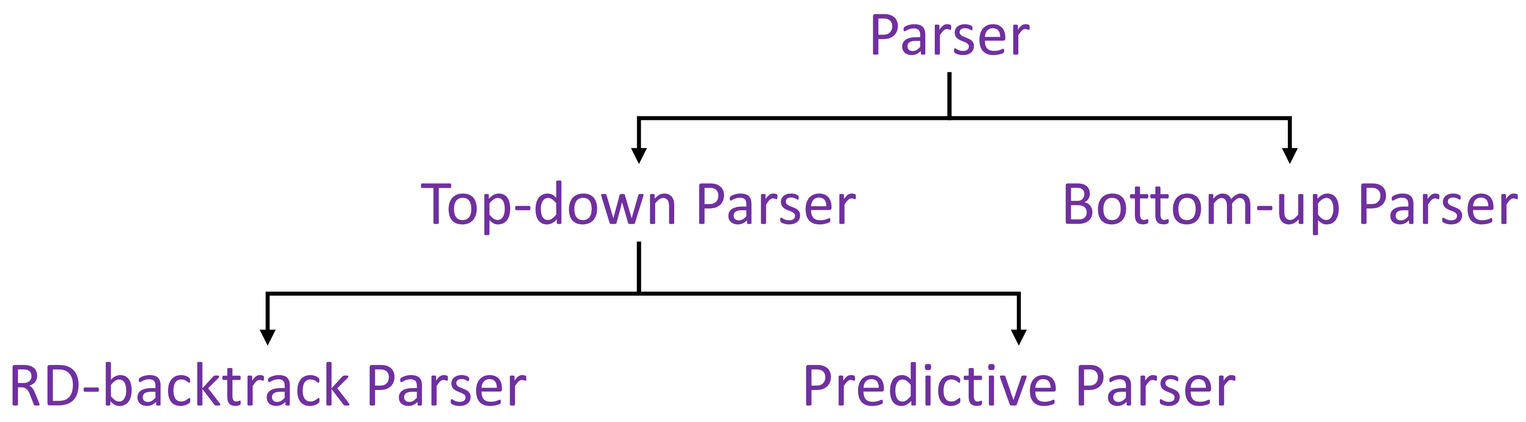
- **Recursive-descent parsing** [RDP, 递归下降语法分析]
  - A general form of top-down parsing.
  - RDP is a simple and general parsing strategy.
    - Left-recursion must be eliminated first.(Can be eliminated automatically using some algorithm)
  - However it is not popular because of **backtracking**.
    - Backtracking requires *re-parsing* the same string.
    - Tried to solve problem *in any possible ways*[穷尽一切可能的试探法] which is inefficient.(can take exponential time)
    - Also removing already added nodes in parse tree is troublesome.

Mind Map[思维导图]

Syntax Analysis: Top-down parser



Predictive Parsing[预测分析]



- **Predictive Parsing**[预测分析法]
  - A special case of recursive-descent parsing **without backtracking**[无回溯].
  - Predictive parsing chooses the correct production by looking ahead at the input a fixed number of symbols, typically we may look only at one.(that is ,the next input symbol)
  - Restrictions on the grammar to avoid backtracking.[LL(k)]

Predictive Parsing[预测分析]

- **A parser with no backtracking** [无回溯]**:** select the correct alternative through given next input terminal(s)[下一个输入符号/终结符]
- 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).
  - Current non-terminal being processed.

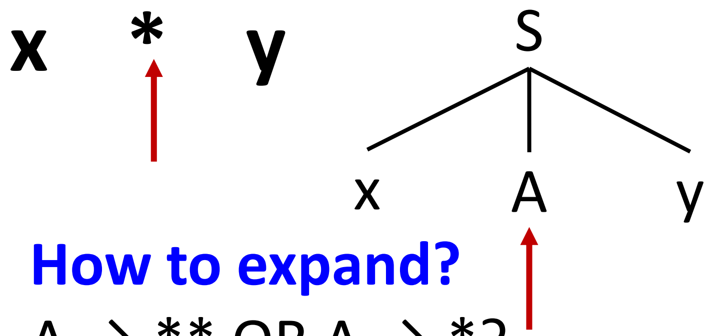
G(S): S → aBD | bBB; B → c | bce; D → d
parsing input "abced" requires no backtracking

What if S → aBD | aBB   **Common Prefix[共同前缀]**

Given input terminal(s) a, cannot choose between two rules.

Common Prefix[共同前缀]

- G[S]: S → xAy; A → ** | *, If the current matching symbol is *, the next step is to expand A, and A → α_1 | α_2 |…| α_n . How to choose α_i?



How to expand?

A → ** OR A → *?

  - ◆(1) If there is a unique α_i with * as the head, replace using this unique α_i.
  - ◆(2) If there are multiple α_i with * as the head, the substitution is not unique[替换是不唯一的], ***backtracking*** may happen.

- **Left factoring**[提取左公因子]: Rewrite the productions to ***defer the decision*** until enough of the input has been seen that we can make the right choice. [推后决定直至可选择]

Left factoring[提取公共左因子]

SUN YAT-SEN UNIVERSITY 中山大學

- For each non-terminal A, find *the longest prefix α* common to two or more of its alternatives. If $\alpha \neq \varepsilon$, **replace** all of the A-productions $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α, by $A \rightarrow \alpha A' \mid \gamma$; $A' \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$.
- *Repeatedly apply this transformation until no two alternatives for a non-terminal have a common prefix.*
- **Example:** G[S]: S → abc |abd |ae;
  - **Step1:** S → aA’; A’ → bc |bd | e;
  - **Step2:** S → aA’; A’ → bB’ | e; B’ → c | d;

Predictive Parsing[预测分析]

- Patterns in grammars that prevent predictive parsing [并非总是能预测分析]：
  - ◆ **Left Recursion Problem** [左递归问题]
    - Lookahead symbol changes **only** when a terminal is matched.
    - *Q: How to solve? A:* **Remove Left Recursion**.
  - ◆ **Common Prefix**[共同前缀] —Cause **Backtracking Problem**.
    - *Q: How to solve? A:* **Left factoring**.
  - ◆ Question: After left factoring, Can we *completely* avoid the backtracking and do predictive parsing? What if S → EBD | FBB; E → a.... ; F → a...

G(S): S → aBD | bBB; B → c | bce; D → d **FIRST && FOLLOW!**
parsing input “abced” requires no backtracking

FIRST[终结首符集]

- During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol.

- **FIRST(α)** [终结首符集] (α is *any string* of grammar symbols)
  - ◆ Define FIRST(α) to be the set of terminals that begin strings[串的终结首符的集合] derived from α. **FIRST(α) = {a | $\alpha \overset{*}{\Rightarrow} a\ldots$, a $\in V_T$}**
  - ◆ AND if $\alpha \overset{*}{\Rightarrow} \varepsilon$, then ε is also in FIRST(α).
  - Consider two(or more) A-productions A → α | β, **where FIRST(α) and FIRST(β) are disjoint sets** [不相交的集合]. We can then choose between these A-productions by looking at the next input symbol *a*, since *a* can be in at most one of FIRST(α) and FIRST(β), not both.

FOLLOW[后继终结符号集]

- **Question:** And then, If for an input symbol ***a*** and the non-terminal **A** ($A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$) , ***but*** **for any α_i , *a* $\notin$ FIRST (α_i),** in this case, how to choose α_i, or draw a conclusion of grammatical errors?
- **FOLLOW(A):** For *nonterminal* A, to be the set of terminals *a* that can appear immediately to the right of A in some sentential form; that is, $FOLLOW(A) = \{a \mid S \overset{*}{\Rightarrow} \ldots Aa\ldots, a \in V_T\}$. [在某些句型中紧跟在A右边的终结符号的集合]
  - In addition, if A can be the rightmost symbol in some sentential form, then \$ is in FOLLOW(A); recall that \$ is a special “endmarker”[结束标记] symbol that is assumed not to be a symbol of any grammar.

FOLLOW[后继终结符号集]

- **Question:** For an input symbol ***a*** and the non-terminal **A** (A → α_1 | α_2 |…| α_n) , ***but*** **for any α_i , *a* ∉ FIRST (α_i),** in this case, how to choose α_i, or draw a conclusion of grammatical errors?
- **Answer:** if ε ∈ FIRST (α_i), then when *a* ∈ FOLLOW (A), select A → α_i, otherwise, grammatical error.
- **Example:** G(S): S → aA | d ;   A → bAS | ε.   String : *abd*.

S	*abd*
⇒ aA	*abd*
⇒ abAS	*abd*
⇒ abS	*abd*
⇒ abd	

FIRST(aA)= {a}
FIRST(d) = {d}
FIRST(bAS)= {b}

HOW TO compute FIRST & FOLLOW?

Compute FIRST[计算FIRST集合的方法]

- To compute FIRST(X) for all grammar symbols X, **apply the following rules until no more terminals or ε can be added to any FIRST set:**
  - **Rule1:** If $X \in V_T$, then FIRST(X) = {X}. [X是终结符]
  - **Rule2:** If $X \in V_N$ and $X \to \varepsilon$ exists, then add ε to FIRST(X). [非终结符,空式]
  - **Rule3:** If $X \in V_N$ and $X \to Y_1Y_2Y_3 \ldots Y_k$, then [非终结符,非空式]
    - If for some Y_i and a terminal a: ① ε ∈ all of FIRST(Y_1),..., FIRST(Y_{i-1}), i.e., $Y_1 \ldots Y_{i-1} \overset{*}{\Rightarrow} \varepsilon$; ② *a* ∈ **FIRST($Y_i$)** \ {ε}. Then, *a* ∈ FIRST(X).
    - E.g.,
      - Everything in FIRST(Y_1) \ {ε} is surely in FIRST(X).
      - If Y_1 doesn't derive ε, then we add nothing more.
      - But if $Y_1 \overset{*}{\Rightarrow} \varepsilon$, then we add FIRST($Y_2$) \ {ε} , and so on...
    - Add ε to FIRST(X), if ε is in FIRST(Y_i) for all i=1,2,...k.

Compute FIRST[计算FIRST集合的方法]

- Next, we can compute FIRST for any string $\alpha = X_1X_2\ldots X_n$ [符号串]
  - ◆ Add all non-ε symbols of FIRST(X_1) to FIRST(α).
  - ◆ Add non-ε symbols of FIRST(X_i), $2 \le i \le n$, to FIRST(α), if FIRST(X_1), ..., FIRST(X_{i-1}) all contain ε. [前k-1个都包含空串]
  - ◆ Add ε to FIRST(α), if FIRST(X_1), ..., FIRST(X_n) all contain ε.

Compute FIRST[计算FIRST集合的方法]

- **Example:** G[E]: E → TE’ ; E’ → +TE’ | ε ; T → FT’ ; T’ → *FT’ | ε ; F → (E) | **id** **, compute the FIRST set of each non-terminal.**
  - FIRST(E):{
  - FIRST(T):{
  - FIRST(E’):{ + , ε
  - FIRST(T’):{ * , ε
  - FIRST(F):{ (, **id**

Apply rules for the first time:

E → TE’ *FIRST(T) add to FIRST(E), FIRST(T) doesn’t contain ε*

E’ → +TE’ *FIRST(+) add to FIRST(E’), FIRST(+) doesn’t contain ε*

E’ → ε *ε add to FIRST(E’)*

T → FT’ *FIRST(F) add to FIRST(T), FIRST(F) doesn’t contain ε*

T’ → *FT’ *FIRST(*) add to FIRST(T’), FIRST(*) doesn’t contain ε*

T’ → ε *ε add to FIRST(T’)*

F → (E) *FIRST(‘(’) add to FIRST(F), FIRST(‘(’) doesn’t contain ε*

F → **id** *FIRST(**id**) add to FIRST(F)*

Compute FIRST[计算FIRST集合的方法]

中山大學 SUN YAT-SEN UNIVERSITY

- **Example:** G[E]: E → TE' ; E' → +TE' | ε ; T → FT' ; T' → *FT' | ε ; F → (E) | **id , compute the FIRST set of each non-terminal.**
  - FIRST(E):{
  - FIRST(T):{ (, **id**
  - FIRST(E'):{ +, ε
  - FIRST(T'):{ *, ε
  - FIRST(F):{ (, **id**

It is necessary to determine whether the FIRST set has changed after each rule application. first time YES!

Apply rules for the second time:

E → TE' *FIRST(T) add to FIRST(E), FIRST(T) doesn't contain ε*

E' → +TE' *FIRST(+) add to FIRST(E'), FIRST(+) doesn't contain ε*

E' → ε *ε add to FIRST(E')*

T → FT' *FIRST(F) add to FIRST(T), FIRST(F) doesn't contain ε*

T' → *FT' *FIRST(*) add to FIRST(T'), FIRST(*) doesn't contain ε*

T' → ε *ε add to FIRST(T')*

F → (E) *FIRST('(') add to FIRST(F), FIRST('(') doesn't contain ε*

F → **id** *FIRST(**id**) add to FIRST(F)*

Compute FIRST[计算FIRST集合的方法]

- **Example:** G[E]: E → TE’ ; E’ → +TE’ | ε ; T → FT’ ; T’ → *FT’ | ε ; F → (E) | **id , compute the FIRST set of each non-terminal.**
  - FIRST(E):{ (, **id**
  - FIRST(T):{ (, **id**
  - FIRST(E’):{ +, ε
  - FIRST(T’):{ *, ε
  - FIRST(F):{ (, **id**

It is necessary to determine whether the FIRST set has changed after each rule application. second time YES!

Apply rules for the third time:

E → TE’ *FIRST(T) add to FIRST(E), FIRST(T) doesn’t contain ε*

E’ → +TE’ *FIRST(+) add to FIRST(E’), FIRST(+) doesn’t contain ε*

E’ → ε *ε add to FIRST(E’)*

T → FT’ *FIRST(F) add to FIRST(T), FIRST(F) doesn’t contain ε*

T’ → *FT’ *FIRST(*) add to FIRST(T’), FIRST(*) doesn’t contain ε*

T’ → ε *ε add to FIRST(T’)*

F → (E) *FIRST(‘(’) add to FIRST(F), FIRST(‘(’) doesn’t contain ε*

F → **id** *FIRST(**id**) add to FIRST(F)*

Compute FIRST[计算FIRST集合的方法]

- **Example:** G[E]: E → TE' ; E' → +TE' | ε ; T → FT' ; T' → *FT' | ε ; F → (E) | **id , compute the FIRST set of each non-terminal.**

  - FIRST(E):{ (, **id** }
  - FIRST(T):{ (, **id** }
  - FIRST(E'):{ +, ε }
  - FIRST(T'):{ *, ε }
  - FIRST(F):{ (, **id** }

It is necessary to determine whether the FIRST set has changed after each rule application. third time YES!

Apply rules for the 4th time:

E → TE' *FIRST(T) add to FIRST(E), FIRST(T) doesn't contain ε*

E' → +TE' *FIRST(+) add to FIRST(E'), FIRST(+) doesn't contain ε*

E' → ε *ε add to FIRST(E')*

T → FT' *FIRST(F) add to FIRST(T), FIRST(F) doesn't contain ε*

T' → *FT' *FIRST(*) add to FIRST(T'), FIRST(*) doesn't contain ε*

T' → ε *ε add to FIRST(T')*

F → (E) *FIRST('(') add to FIRST(F), FIRST('(') doesn't contain ε*

F → **id** *FIRST(**id**) add to FIRST(F)*

It is necessary to determine whether the FIRST set has changed after each rule application. 4th time NO!

Compute FOLLOW[计算FOLLOW集合的方法]

- To compute FOLLOW(A) for all non-terminals A, **apply the following rules until nothing can be added to any FOLLOW set.**
  - ◆**Rule1:** Place \$ in FOLLOW(S), where S is the start symbol.
  - ◆**Rule2:** If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ε is in FOLLOW(B).
  - ◆**Rule3:** 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).
- **Example:** G[E]: E → TE' ; E' → +TE' | ε ; T → FT' ; T' → *FT' | ε ; F → (E) | id , **compute the FOLLOW set of each non-terminal.**
  - FIRST(E):{ (, **id** }; FIRST(T):{ (, **id** } ; FIRST(E'):{ +, ε } ; FIRST(T'):{ *, ε } ; FIRST(F):{ (, **id** }.

Compute FOLLOW[计算FOLLOW集合的方法]

- **Example:** G[E]: E → TE’ ; E’ → +TE’ | ε ; T → FT’ ; T’ → *FT’ | ε ; F → (E) | **id , compute the FOLLOW set of each non-terminal.**
  - FIRST(E):{ (, **id** };FIRST(T):{ (, **id** };FIRST(E’):{ +, ε };FIRST(T’):{ *, ε };FIRST(F):{ (, **id** }.
  - FOLLOW(E):{ $
  - FOLLOW(T):{ +, $
  - FOLLOW(E’):{ $
  - FOLLOW(T’):{
  - FOLLOW(F):{

Apply rules for the first time:

- Place $ in FOLLOW(E), since E is the start symbol.
- E → TE’
  - FIRST(E’) except ε is in FOLLOW(T).
  - Everything in FOLLOW(E) is in FOLLOW(E’).
  - Since FIRST(E’) contains ε, then everything in FOLLOW(E) is in FOLLOW(T).
- E’ → +TE’ | ε
  - FIRST(E’) except ε is in FOLLOW(T).
  - Everything in FOLLOW(E’) is in FOLLOW(E’).
  - Since FIRST(E’) contains ε, then everything in FOLLOW(E’) is in FOLLOW(T).

Compute FOLLOW[计算FOLLOW集合的方法]

- **Example:** G[E]: E → TE' ; E' → +TE' | ε ; T → FT' ; T' → *FT' | ε ; F → (E) | i , **compute the FOLLOW set of each non-terminal.**
  - FIRST(E):{ (, **id** };FIRST(T):{ (, **id** };FIRST(E'):{ +, ε };FIRST(T'):{ *, ε };FIRST(F):{ (, **id** }.
  - FOLLOW(E):{ $
  - FOLLOW(T):{ +, $
  - FOLLOW(E'):{ $
  - FOLLOW(T'):{+, $
  - FOLLOW(F):{ *, +, $

Apply rules for the first time:

- T → FT'
  - FIRST(T') except ε is in FOLLOW(F).
  - Everything in FOLLOW(T) is in FOLLOW(T').
  - Since FIRST(T') contains ε, then everything in FOLLOW(T) is in FOLLOW(F).
- T' → *FT' | ε
  - FIRST(T') except ε is in FOLLOW(F).
  - Everything in FOLLOW(T') is in FOLLOW(T').
  - Since FIRST(T') contains ε, then everything in FOLLOW(T') is in FOLLOW(F).

Compute FOLLOW[计算FOLLOW集合的方法]

- **Example:** G[E]: E → TE' ; E' → +TE' | ε ; T → FT' ; T' → *FT' | ε ; F → (E) | i , **compute the FOLLOW set of each non-terminal.**
  - FIRST(E):{ (, **id** };FIRST(T):{ (, **id** };FIRST(E'):{ +, ε };FIRST(T'):{ *, ε };FIRST(F):{ (, **id** }.
  - FOLLOW(E):{ $,)
  - FOLLOW(T):{ +, $,)
  - FOLLOW(E'):{ $,)
  - FOLLOW(T'):{ +, $
  - FOLLOW(F):{ *, +, $

Apply rules for the first time:

- F → (E) | **id**
  - FIRST(')') except ε is in FOLLOW(E).

Since FOLLOW sets has changed at the first time, **Apply rules for the second time:**

- E → TE'
  - FIRST(E') except ε is in FOLLOW(T).
  - Everything in FOLLOW(E) is in FOLLOW(E').
  - Since FIRST(E') contains ε, then everything in FOLLOW(E) is in FOLLOW(T).

Compute FOLLOW[计算FOLLOW集合的方法]

- **Example:** G[E]: E → TE’ ; E’ → +TE’ | ε ; T → FT’ ; T’ → *FT’ | ε ; F → (E) | i , **compute the FOLLOW set of each non-terminal.**
  - FIRST(E):{ (, **id** };FIRST(T):{ (, **id** };FIRST(E’):{ +, ε };FIRST(T’):{ *, ε };FIRST(F):{ (, **id** }.
  - FOLLOW(E):{ $,)
  - FOLLOW(T):{ +, $,)
  - FOLLOW(E’):{ $,)
  - FOLLOW(T’):{ +, $,)
  - FOLLOW(F):{ *, +, $,)

- E’ → +TE’ | ε
  - FIRST(E’) except ε is in FOLLOW(T).
  - Everything in FOLLOW(E’) is in FOLLOW(E’).
  - Since FIRST(E’) contains ε, then everything in FOLLOW(E’) is in FOLLOW(T).
- T → FT’
  - FIRST(T’) except ε is in FOLLOW(F).
  - Everything in FOLLOW(T) is in FOLLOW(T’).
  - Since FIRST(T’) contains ε, then everything in FOLLOW(T) is in FOLLOW(F).

Compute FOLLOW[计算FOLLOW集合的方法]

- **Example:** G[E]: E → TE' ; E' → +TE' | ε ; T → FT' ; T' → *FT' | ε ; F → (E) | i , **compute the FOLLOW set of each non-terminal.**
  - FIRST(E):{ (, **id** };FIRST(T):{ (, **id** };FIRST(E'):{ +, ε };FIRST(T'):{ *, ε };FIRST(F):{ (, **id** }.
  - FOLLOW(E):{ $,) }
  - FOLLOW(T):{ +, $,) }
  - FOLLOW(E'):{ $,) }
  - FOLLOW(T'):{ +, $,) }
  - FOLLOW(F):{ *, +, $,) }

- T' → *FT' | ε
  - FIRST(T') except ε is in FOLLOW(F).
  - Everything in FOLLOW(T') is in FOLLOW(T').
  - Since FIRST(T') contains ε, then everything in FOLLOW(T') is in FOLLOW(F).
- F → (E) | **id**
  - FIRST(')') except ε is in FOLLOW(E).

Since FOLLOW sets has changed at the second time, **Apply rules for the third time:**

- **EXERCISE! –Ans: no FOLLOW set will be changed.**

LL(1) Grammar[LL(1)文法]

- *Predictive parsers*, that is, recursive-descent parsers needing no backtracking, can be constructed for a class of grammars called LL(1).
- A grammar G is LL(1) **if and only if** whenever any two distinct productions of G [G的任意两个不同的产生式], $A \to \alpha \mid \beta$, the following conditions hold:
  - For no terminal *a* do both α and β derive strings beginning with *a*.
  - At most one of α and β can derive the empty string.
  - If $\beta \stackrel{*}{\Rightarrow} \varepsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A). Likewise, if $\alpha \stackrel{*}{\Rightarrow} \varepsilon$, then β does not derive any string beginning with a terminal in FOLLOW(A).

LL(1) Grammar[LL(1)文法]

- The first two conditions are equivalent to the statement that FIRST(α) and FIRST(β) are disjoint sets [不相交的集合].
- The third condition is equivalent to stating that if ε is in FIRST(β), then FIRST(α) and FOLLOW(A) are disjoint sets, and likewise if ε is in FIRST(α).
- LL(1) Grammar does not contain left recursion. [LL(1)文法不含左递归]
- LL(1) Grammar is not ambiguous. [LL(1)文法不是二义的]

LL(1)/LL(k) Grammar[LL(1)/LL(k)文法]

- **LL (1) grammar.**
  - **L: The first "L" in LL(1) stands for scanning the input from left to right.**
  - **L: The second "L" for producing a leftmost derivation.**
  - **1: The "1" for using one input symbol of lookahead at each step to make parsing action decisions.**
- **LL (k) grammar.**
  - k: using **k** input symbols of lookahead at each step to make parsing action decisions.
- Many languages are LL(k), in fact many are LL(1).
- **Is LL(0) useful at all?**
  - Grammar where rules can be predicted with no lookahead.
  - ⇒ That means, there can only be one rule per non-terminal.
  - ⇒ That means, this language can have only one string.

LL(1) Parser Implementation[实现]

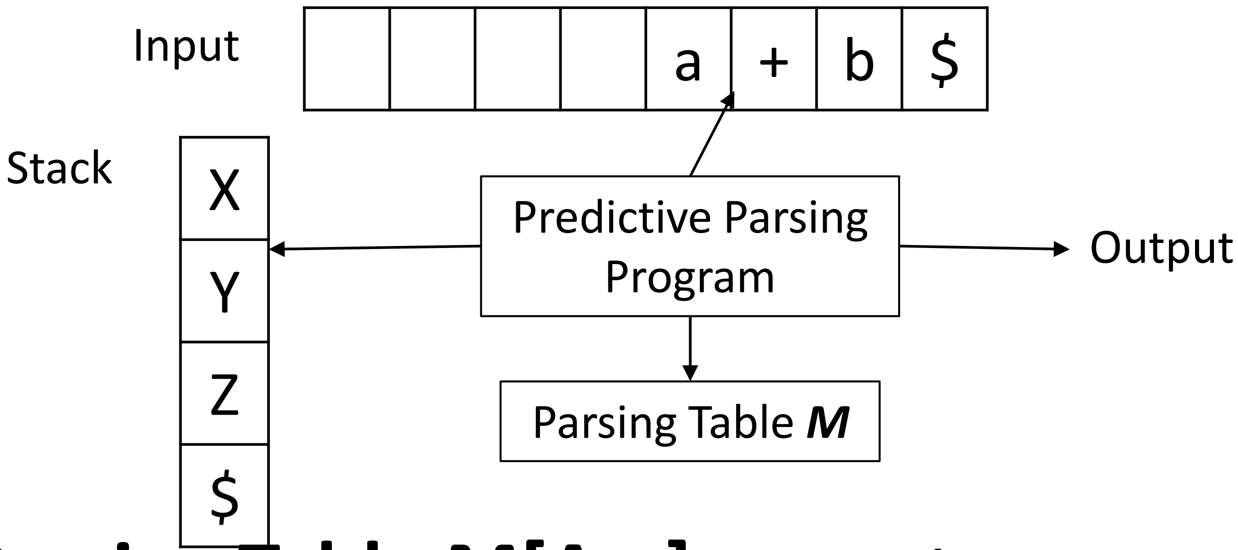
- *Recursive* LL(1) parser for G[S]: S → A | B; A → a; B → b.
  - maintaining a stack implicitly. [隐式维护栈]

```
void S(){
        token = Next(); // lookahead
        if(token == a) // 'A' starts with 'a'
                A(); // call procedure A()
        else if (token == b) // 'B' starts with 'b'
                B(); // call procedure B()
        else
                return; // error, reject.
}
```

- Is there a way to express above code more concisely? [简洁]
  - *Non-recursive* LL(1) parsers
    - Use a predictive parsing table. [预测分析表]
    - maintaining a stack explicitly. [显式维护栈]
    - Table-driven parser. [表驱动]

Non-recursive LL(1) Parser [非递归]

- **Input buffer**: contains the string to be parsed, followed by the endmarker $.
- **Stack:** holds a sequence of grammar symbols and the symbol $ to mark the bottom of the stack. It may contain:
  - **Terminals** that have yet to matched against the input symbol.
  - **Non-terminals** that have yet to be expanded.



- **Parsing Table M[A,a]:** an entry containing production "A→…" or error.
- **Predictive Parsing Program:** Execute the action according to **<stack top, current input symbol>**

LL(1) Parse Table [预测分析表]

中山大學 SUN YAT-SEN UNIVERSITY

G[E]:

E → TE’

E’ → +TE’ | ε

T → FT’

T’ → *FT’ | ε

F → (E) | **id**

	id	+	*	(	)	$
E	E → TE’			E → TE’		
E’		E’ → +TE’			E’ → ε	E’ → ε
T	T → FT’			T → FT’		
T’		T’ → ε	T’ → *FT’		T’ → ε	T’ → ε
F	F → **id**			F → (E)		

Next input symbol, lists all possible terminals and $

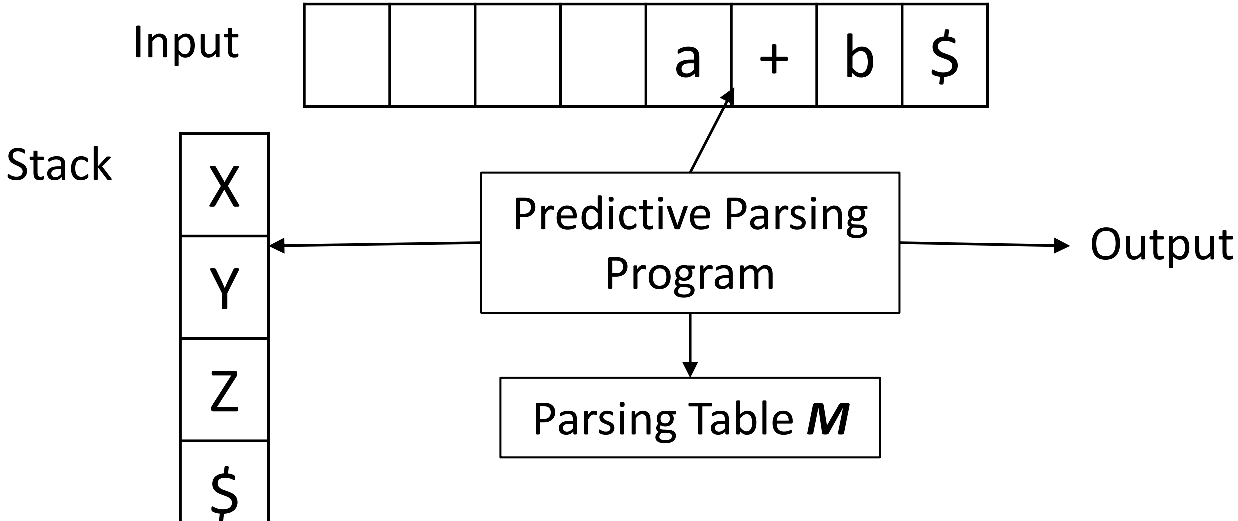
all non-terminals in the grammar

One action for each **<non-terminal, next input>,** it “predicts” the correct action based on one lookahead

- Reject on reaching error state
- Accept on end of input & empty stack

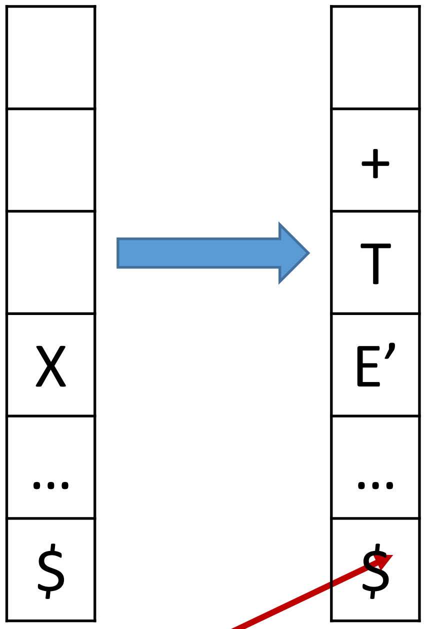
LL(1) Parsing Algorithm [非递归算法]

- Initial state [初始态]
  - **Input:** A string *w* and a parsing table *M* for grammar G. **Input Buffer**: w$.
  - **Stack:** start symbol followed by '$' at bottom.
  - Assume **X:** symbol at the top of the stack, **a:** current input symbol.
- 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



LL(1) Parsing Algorithm [非递归算法]

- Algorithm Step-by-Step based on **<X, a>**:
  - X: symbol at the top of the stack
  - a: current input token
  - If X ∈ V_T, [栈顶符号为终结符] and
    - X == a == \$, declare **SUCCESS**, stop parsing.
    - X == a != \$, **pop X from stack** and move the current input symbol forward one.
    - X != a, declare **ERROR**, input is rejected, stop parsing.
  - If X ∈ V_N, [栈顶符号为非终结符] and
    - M[X, a] has a production about X, pop X and ***push right side of production to stack.***
    - M[X, a] == empty, declare **ERROR**, input is rejected, stop parsing.



If M[X,a] = "X → +TE'"

Push to stack in Reverse Order [逆序入栈]

If M[X,a] = "X → ε"

Only pop and push nothing

Use the Parse Table [使用表进行预测分析]

- When using parse table for predictive parsers, if the grammar does not conform to the specification of LL (1) grammar:
  - the grammar should be rewritten into LL (1) grammar by removing left recursion and backtracking, then the parse table should be constructed.
- **Example:** consider G[E] to recognize "**id + id * id**"

G[E]:
E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | **id**

	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → **id**			F → (E)		

Use the Parse Table [使用表进行预测分析]

- recognize “**id + id * id**”

Input Buffer

		id	+	**id**	*	**id**	$

Stack

E
$

Predictive Parsing Program

	id	+	*	(	)	$
E	E → TE’			E → TE’		
E’		E’ → +TE’			E’ → ε	E’ → ε
T	T → FT’			T → FT’		
T’		T’ → ε	T’ → *FT’		T’ → ε	T’ → ε

G[E]:

E → TE’

E’ → +TE’ | ε

T → FT’

T’ → *FT’ | ε

F → (E) | **id**

Use the Parse Table [使用表进行预测分析]

- recognize “**id** + **id** * **id**”

Input Buffer

		id	+	**id**	*	**id**	$

Stack

T
E’
$

Predictive Parsing Program

G[E]:

E → TE’

E’ → +TE’ | ε

T → FT’

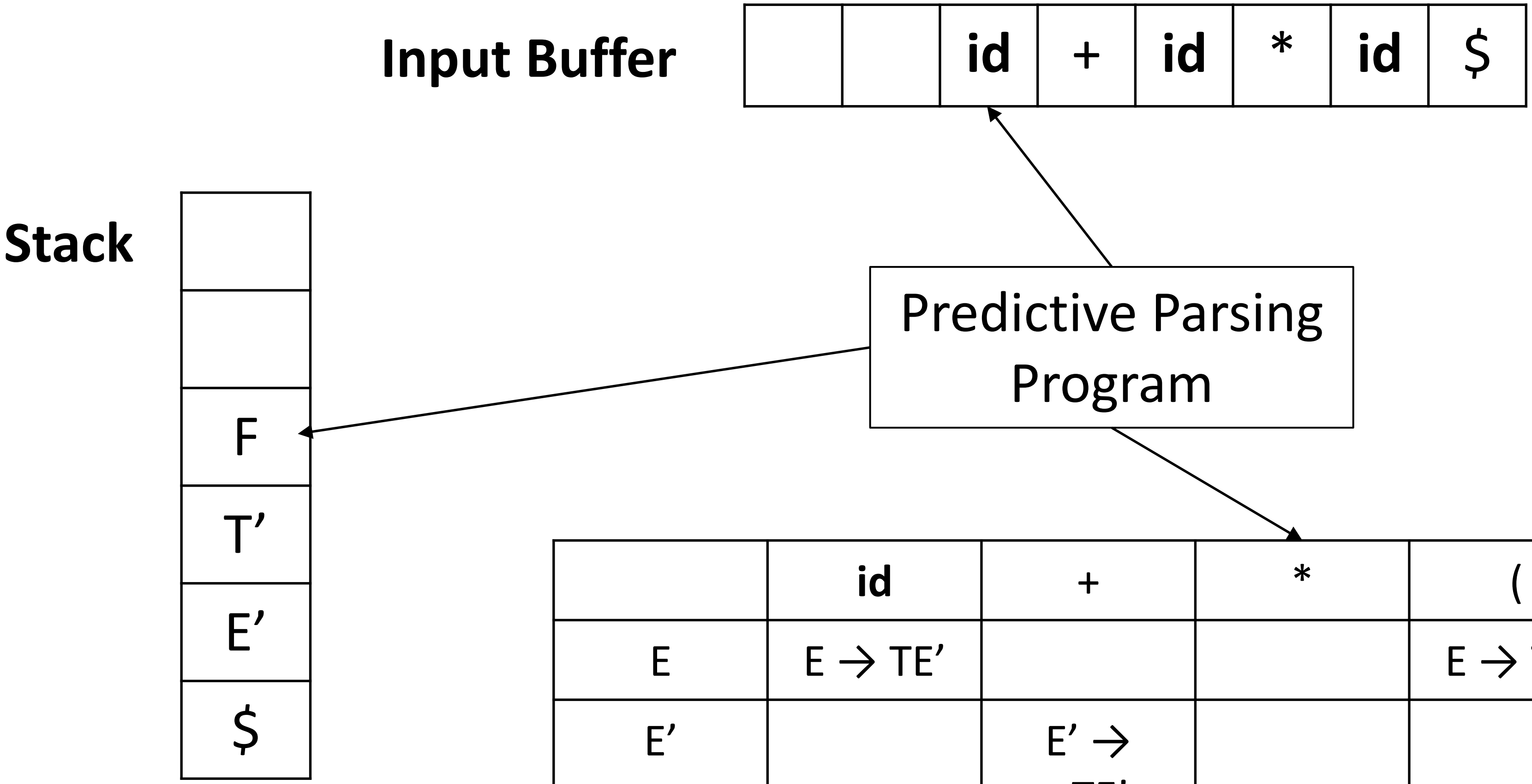
T’ → *FT’ | ε

F → (E) | **id**

	id	+	*	(	)	$
E	E → TE’			E → TE’		
E’		E’ → +TE’			E’ → ε	E’ → ε
T	T → FT’			T → FT’		
T’		T’ → ε	T’ → *FT’		T’ → ε	T’ → ε

Use the Parse Table [使用表进行预测分析]

- recognize “**id + id * id**”



G[E]:

E → TE’

E’ → +TE’ | ε

T → FT’

T’ → *FT’ | ε

F → (E) | **id**

	id	+	*	(	)	$
E	E → TE’			E → TE’		
E’		E’ → +TE’			E’ → ε	E’ → ε
T	T → FT’			T → FT’		
T’		T’ → ε	T’ → *FT’		T’ → ε	T’ → ε

Use the Parse Table [使用表进行预测分析]

- recognize “**id + id * id**”

Input Buffer

		id	+	**id**	*	**id**	$

Stack

id
T’
E’
$

Predictive Parsing Program

	id	+	*	(	)	$
E	E → TE’			E → TE’		
E’		E’ → +TE’			E’ → ε	E’ → ε
T	T → FT’			T → FT’		
T’		T’ → ε	T’ → *FT’		T’ → ε	T’ → ε

G[E]:

E → TE’

E’ → +TE’ | ε

T → FT’

T’ → *FT’ | ε

F → (E) | **id**

Use the Parse Table [使用表进行预测分析]

- recognize “**id + id * id**”

Input Buffer

		id	+	**id**	*	**id**	$

Stack

T’
E’
$

Predictive Parsing Program

G[E]:

E → TE’

E’ → +TE’ | ε

T → FT’

T’ → *FT’ | ε

F → (E) | **id**

	id	+	*	(	)	$
E	E → TE’			E → TE’		
E’		E’ → +TE’			E’ → ε	E’ → ε
T	T → FT’			T → FT’		
T’		T’ → ε	T’ → *FT’		T’ → ε	T’ → ε

Use the Parse Table [使用表进行预测分析]

- recognize "**id + id * id**"

G[E]:

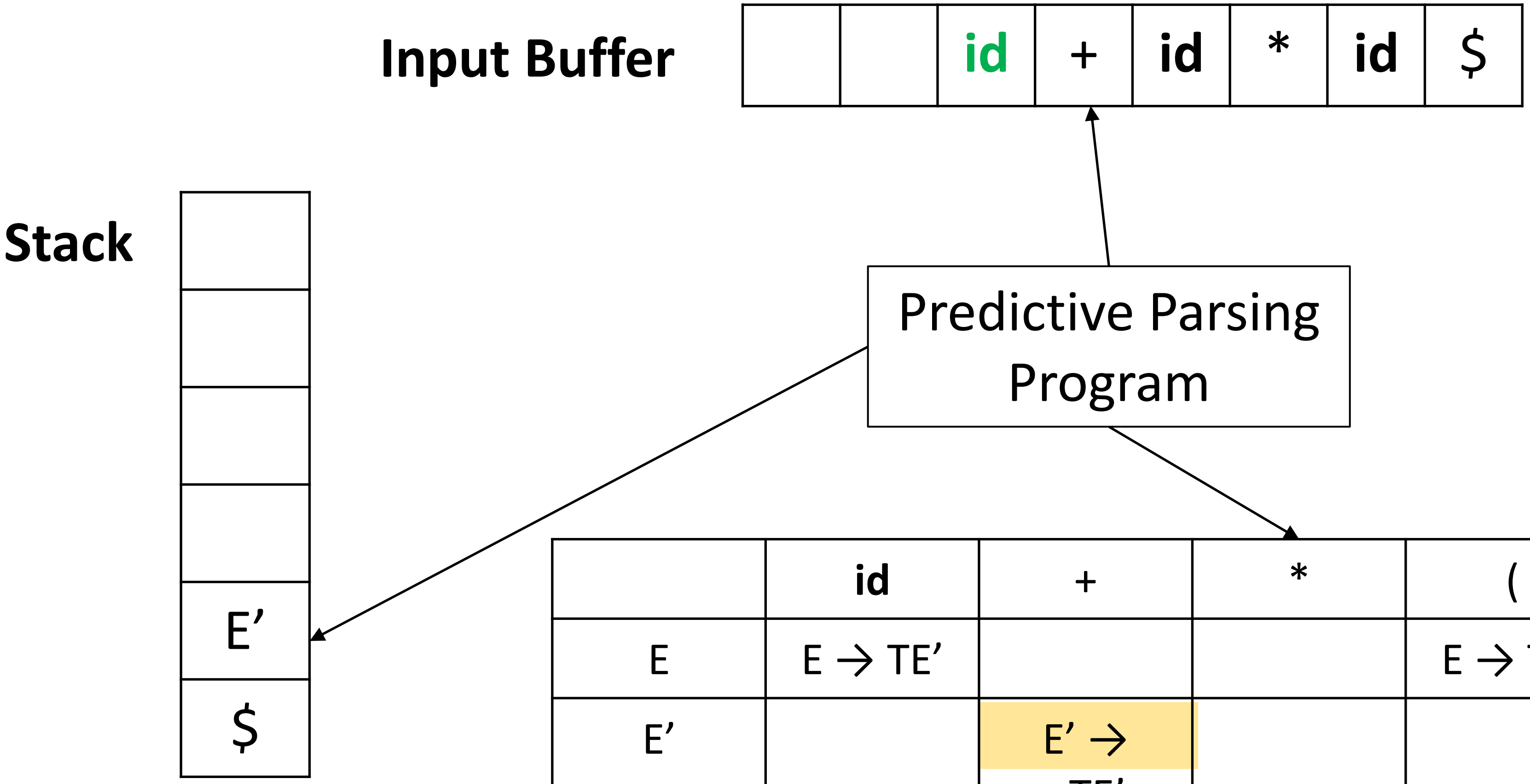
E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | **id**



	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε

Use the Parse Table [使用表进行预测分析]

- recognize “**id + id * id**”

Input Buffer

		id	+	**id**	*	**id**	$

Stack

+
T
E’
$

Predictive Parsing Program

	id	+	*	(	)	$
E	E → TE’			E → TE’		
E’		E’ → +TE’			E’ → ε	E’ → ε
T	T → FT’			T → FT’		
T’		T’ → ε	T’ → *FT’		T’ → ε	T’ → ε

G[E]:

E → TE’

E’ → +TE’ | ε

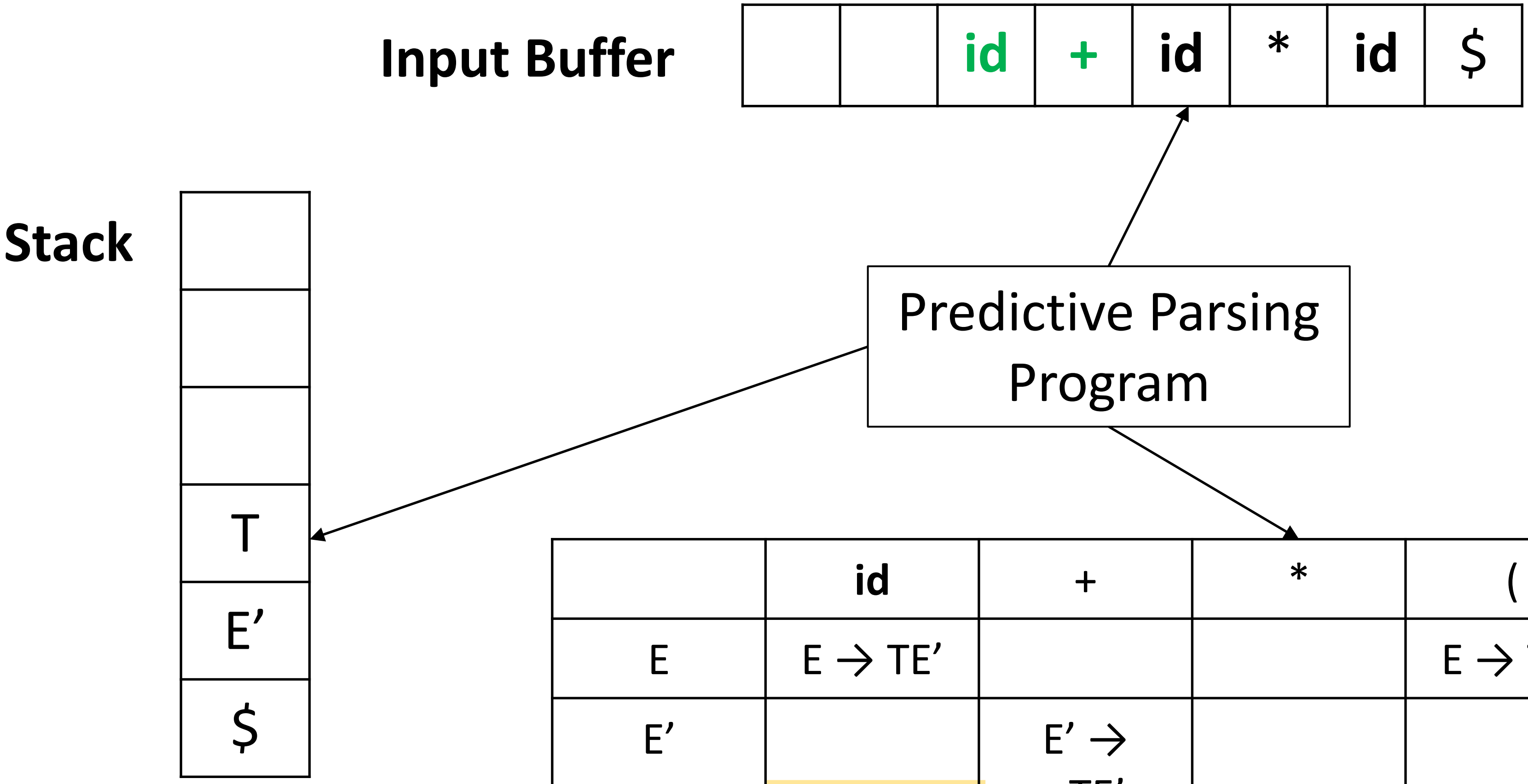
T → FT’

T’ → *FT’ | ε

F → (E) | **id**

Use the Parse Table [使用表进行预测分析]

- recognize “**id + id * id**”



G[E]:

E → TE’

E’ → +TE’ | ε

T → FT’

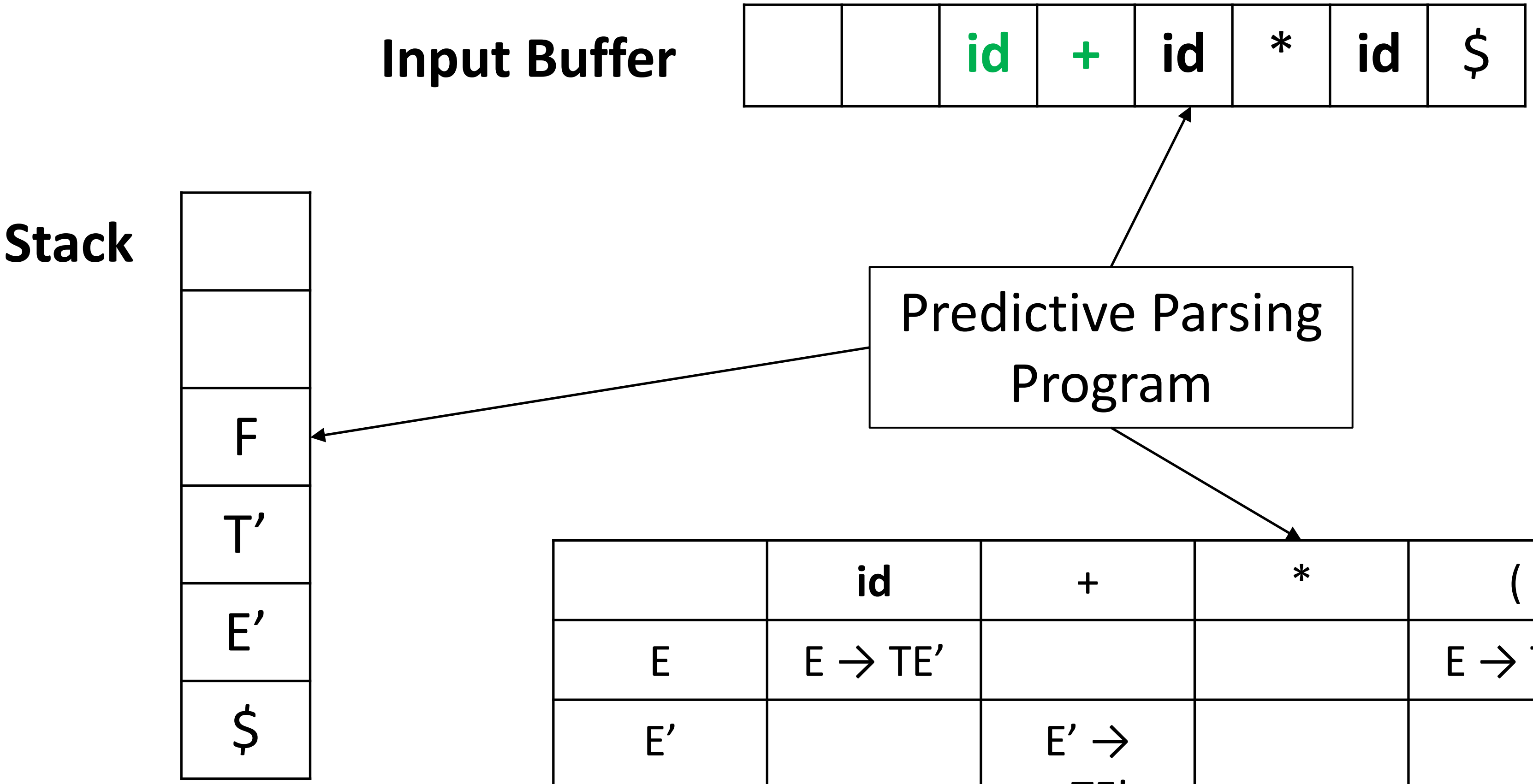
T’ → *FT’ | ε

F → (E) | **id**

	id	+	*	(	)	$
E	E → TE’			E → TE’		
E’		E’ → +TE’			E’ → ε	E’ → ε
T	T → FT’			T → FT’		
T’		T’ → ε	T’ → *FT’		T’ → ε	T’ → ε

Use the Parse Table [使用表进行预测分析]

- recognize “**id + id * id**”



G[E]:

E → TE’

E’ → +TE’ | ε

T → FT’

T’ → *FT’ | ε

F → (E) | **id**

	id	+	*	(	)	$
E	E → TE’			E → TE’		
E’		E’ → +TE’			E’ → ε	E’ → ε
T	T → FT’			T → FT’		
T’		T’ → ε	T’ → *FT’		T’ → ε	T’ → ε

Use the Parse Table [使用表进行预测分析]

- recognize “**id + id * id**”

Input Buffer

		id	+	id	*	**id**	$

Stack

id
T’
E’
$

Predictive Parsing Program

	id	+	*	(	)	$
E	E → TE’			E → TE’		
E’		E’ → +TE’			E’ → ε	E’ → ε
T	T → FT’			T → FT’		
T’		T’ → ε	T’ → *FT’		T’ → ε	T’ → ε

G[E]:

E → TE’

E’ → +TE’ | ε

T → FT’

T’ → *FT’ | ε

F → (E) | **id**

Use the Parse Table [使用表进行预测分析]

- recognize "**id** + **id** * **id**"

Input Buffer

		id	+	id	*	id	$

Stack

T'
E'
$

Predictive Parsing Program

	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε

G[E]:

E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | **id**

Use the Parse Table [使用表进行预测分析]

- recognize “**id + id * id**”

Input Buffer

		id	+	id	*	id	$

Stack

*
F
T’
E’
$

Predictive Parsing Program

	id	+	*	(	)	$
E	E → TE’			E → TE’		
E’		E’ → +TE’			E’ → ε	E’ → ε
T	T → FT’			T → FT’		
T’		T’ → ε	T’ → *FT’		T’ → ε	T’ → ε

G[E]:

E → TE’

E’ → +TE’ | ε

T → FT’

T’ → *FT’ | ε

F → (E) | **id**

Use the Parse Table [使用表进行预测分析]

中山大學 SUN YAT-SEN UNIVERSITY

- recognize “**id + id * id**”

Input Buffer

		id	+	id	*	**id**	$

Stack

F
T’
E’
$

Predictive Parsing Program

G[E]:

E → TE’

E’ → +TE’ | ε

T → FT’

T’ → *FT’ | ε

F → (E) | **id**

	id	+	*	(	)	$
E	E → TE’			E → TE’		
E’		E’ → +TE’			E’ → ε	E’ → ε
T	T → FT’			T → FT’		
T’		T’ → ε	T’ → *FT’		T’ → ε	T’ → ε

Use the Parse Table [使用表进行预测分析]

- recognize “**id + id * id**”

Input Buffer

		id	+	id	*	id	$

Stack

id
T’
E’
$

Predictive Parsing Program

	id	+	*	(	)	$
E	E → TE’			E → TE’		
E’		E’ → +TE’			E’ → ε	E’ → ε
T	T → FT’			T → FT’		
T’		T’ → ε	T’ → *FT’		T’ → ε	T’ → ε

G[E]:

E → TE’

E’ → +TE’ | ε

T → FT’

T’ → *FT’ | ε

F → (E) | **id**

Use the Parse Table [使用表进行预测分析]

- recognize “**id + id * id**”

Input Buffer

		id	+	id	*	id	$

Stack

T’
E’
$

Predictive Parsing Program

	id	+	*	(	)	$
E	E → TE’			E → TE’		
E’		E’ → +TE’			E’ → ε	E’ → ε
T	T → FT’			T → FT’		
T’		T’ → ε	T’ → *FT’		T’ → ε	T’ → ε

G[E]:

E → TE’

E’ → +TE’ | ε

T → FT’

T’ → *FT’ | ε

F → (E) | **id**

Use the Parse Table [使用表进行预测分析]

- recognize “**id + id * id**”

Input Buffer

		id	+	id	*	id	$

Stack

E’
$

Predictive Parsing Program

G[E]:

E → TE’

E’ → +TE’ | ε

T → FT’

T’ → *FT’ | ε

F → (E) | **id**

	id	+	*	(	)	$
E	E → TE’			E → TE’		
E’		E’ → +TE’			E’ → ε	E’ → ε
T	T → FT’			T → FT’		
T’		T’ → ε	T’ → *FT’		T’ → ε	T’ → ε

Use the Parse Table [使用表进行预测分析]

- recognize “**id + id * id**”

Input Buffer

		id	+	id	*	id	$

Stack

$

Predictive Parsing Program

ACCEPT!!

	id	+	*	(	)	$
E	E → TE’			E → TE’		
E’		E’ → +TE’			E’ → ε	E’ → ε
T	T → FT’			T → FT’		
T’		T’ → ε	T’ → *FT’		T’ → ε	T’ → ε

G[E]:

E → TE’

E’ → +TE’ | ε

T → FT’

T’ → *FT’ | ε

F → (E) | **id**

Use the Parse Table [使用表进行预测分析]

中山大學 SUN YAT-SEN UNIVERSITY

Matched	Stack	Input	Action
	E\$	**id** + **id** * **id**\$	E → TE'
	TE'\$	**id** + **id** * **id**\$	T → FT'
	FT'E'\$	**id** + **id** * **id**\$	F → **id**
	idT'E'\$	**id** + **id** * **id**\$	match **id**
id	T'E'\$	+ **id** * **id**\$	T' → ε
id	E'\$	+ **id** * **id**\$	E' → +TE'
id	+TE'\$	+ **id** * **id**\$	match +
id +	TE'\$	**id** * **id**\$	T → FT'
id +	FT' E'\$	**id** * **id**\$	F → **id**
id +	**id**T' E'\$	**id** * **id**\$	match **id**
id + **id**	T' E'\$	* **id**\$	T' → *FT'
id + **id**	*FT'E'\$	* **id**\$	match *

Matched	Stack	Input	Action
id + **id** *	FT'E'\$	**id**\$	F → **id**
id + **id** *	**id**T'E'\$	**id**\$	match **id**
id + **id** * **id**	T'E'\$	\$	T' → ε
id + **id** * **id**	E'\$	\$	E' → ε
id + **id** * **id**	\$	\$	**ACCEPT**

The parser mimics a leftmost derivation.

HOW TO construct LL(1) Parse Table?

Construct LL(1) Parse Table [构建预测分析表]

- Use **FIRST** and **FOLLOW** sets into a predictive parsing table M[A,a], and the algorithm is based on the following idea:
  - The production $A \rightarrow \alpha$ is chosen if the next input symbol *a* is in FIRST (α).
  - The only complication occurs when $\alpha = \varepsilon$ or, more generally, $\alpha \stackrel{*}{\Rightarrow} \varepsilon$. In this case:
    - we should again choose $A \rightarrow \alpha$, if the current input symbol is in FOLLOW(A),
    - or if the \$ on the input has been reached and \$ is in FOLLOW (A).

Construct LL(1) Parse Table [构建预测分析表]

- ***Algorithm***: For each production A → α of the grammar, do the following:
  - For each terminal *a* in FIRST(α), add A → α to M[A, a].
  - If ε ∈ FIRST(α), then for each terminal *b* in FOLLOW(A), add A → α to M[A,b]. If ε ∈ FIRST(α) and $ ∈ FOLLOW(A), add A → α 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 we normally represent by an empty entry in the table).

Construct LL(1) Parse Table [构建预测分析表]

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

G[E]:

E → TE’

E’ → +TE’ \| ε

T → FT’

T’ → *FT’ \| ε

F → (E) \| **id**

Production	**FIRST(α)**
E → TE’	(, i
E’ → +TE’	+
T → FT’	(, i
T’ → *FT’	*
F → (E)	(
F → **id**	**id**
E’ → ε	FOLLOW
T’ → ε	FOLLOW

	id	+	*	(	)	$
E	E → TE’			E → TE’		
E’		E’ → +TE’			E’ → ε	E’ → ε
T	T → FT’			T → FT’		
T’		T’ → ε	T’ → *FT’		T’ → ε	T’ → ε

Example!

Determine If Grammar is LL(1)[判断LL(1)文法]

- 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).

	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → **id**			F → (E)		

- Two methods to determine if a
  - ◆ 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) if and only if for any two distinct productions A → α|β:

    - ◆ FIRST(α) ∩ FIRST(β) = ϕ
    - ◆ If β derives ε, then FIRST(α) ∩ FOLLOW(A) = ϕ….(Mentioned before)

Non-LL(1) Grammar [非LL(1)文法]

- Assume that a grammar is not LL(1). How to solve?
  - Case1- the language may still be LL(1)
    - Try to rewrite grammar to LL(1) grammar [*remove left-recursion & left-factoring*]
    - Try to remove ambiguity in grammar.
  - Case2- If Case-1 fails, language may not be LL(1)
    - It's 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.
- **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.

LL(1) Time and Space Complexity[复杂度]

- **Space:** smaller than input (after removing $X \rightarrow \varepsilon$)
  - Right side of production is always longer or equal to left side of production
    - Derivation string expands monotonically.
    - Derivation string is always shorter than final input string.
  - Stack is a subset of derivation string (unmatched portion)
- LL(k)'s size of parse table = $O(|N| * |T|^k)$[prevent LL(2) ... LL(k) from wide usage]
  - N = number of non-terminals, T = number of terminals

Summary

- Top-down Parsing; RDP with backtracking, predictive parsing.
- Left Recursion Problem, remove left recursion: immediate / non-immediate.
- Left-factoring.
- FIRST, FOLLOW.
- LL(1)/LL(k) Grammar.
- Recursive / Non-recursive LL(1) parser implementation.
- Use LL(1) Parse Table.
- Construct LL(1) Parse Table.

Mind Map[思维导图]

Syntax Analysis: Top-down parser