

Describing Syntax

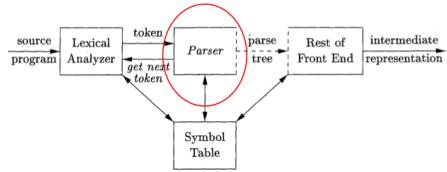
- The syntax of programming language constructs can be specified by **context-free grammars**¹
 - A grammar gives a precise and easy-to-understand **syntactic specification** of a programming language, **defining** its structure
 - For certain grammars, we can **automatically construct** an efficient **parser**
 - A properly designed grammar helps **translate source programs** into correct object code and **detect errors**

¹Can also be specified using BNF (Backus-Naur Form) notation, which basically can be seen as a variant of CFG:
<http://www.cs.nuim.ie/~jpower/Courses/Previous/parsing/node23.html>

可用上下文无关文法解释编程语言。

The Role of the Parser

- The parser obtains a string of tokens from the lexical analyzer and **verifies that the string of token names can be generated by the grammar for the source language**
- Report syntax errors in an intelligent fashion
- For well-formed programs, the parser constructs a parse tree
 - The parse tree need not be constructed explicitly



构建语法树

Classification of Parsers

- Universal parsers (通用语法分析器)**
 - Some methods (e.g., Earley's algorithm¹) can parse any grammar
 - However, they are too inefficient to be used in practice
- Top-down parsers (自顶向下语法分析器)**
 - Construct parse trees from the top (root) to the bottom (leaves)
- Bottom-up parsers (自底向上语法分析器)**
 - Construct parse trees from the bottom (leaves) to the top (root)

Note: Top-down and bottom-up parsing both scan the input from left to right, one symbol at a time. They work only for certain grammars, which are expressive enough.

Context-free Grammar

- A context-free grammar (CFG) consists of four parts:
 - Terminals (终结符号):** Basic symbols from which strings are formed (token names)
 - Nonterminals (非终结符号):** Syntactic variables that denote sets of strings
 - Usually correspond to a language construct, such as `stmt` (statements)
 - One nonterminal is distinguished as the **start symbol (开始符号)**
 - The set of strings denoted by the start symbol is the language generated by the CFG
 - Productions (产生式):** Specify how the terminals and nonterminals can be combined to form strings
 - Format:** $\text{head} \rightarrow \text{body}$
 - head** must be a nonterminal; **body** consists of zero or more terminals/nonterminals
 - Example:** $\text{expression} \rightarrow \text{expression} + \text{term}$

可类比为常量
可类比为变量

也即替换规则

CFG Example

- The grammar below defines simple arithmetic expressions

- Terminal symbols: `id`, `+`, `-`, `*`, `/`, `(`, `)`

- Nonterminals: `expression`, `term` (项), `factor` (因子)

- Start symbol: `expression`

- Productions:

- $\text{expression} \rightarrow \text{expression} + \text{term}$
- $\text{expression} \rightarrow \text{expression} - \text{term}$
- $\text{expression} \rightarrow \text{term}$
- $\text{term} \rightarrow \text{term} * \text{factor}$
- $\text{term} \rightarrow \text{term} / \text{factor}$
- $\text{term} \rightarrow \text{factor}$
- $\text{factor} \rightarrow (\text{expression})$
- $\text{factor} \rightarrow \text{id}$

→ can be read as:
can be in the form, can be replaced by, can be re-written as, can produce, can generate, can make...

Notational Simplification

```
expression → expression + term  
expression → expression - term  
expression → term  
term → term * factor  
term → term / factor  
term → factor  
factor → ( expression )  
factor → id
```

$$\xrightarrow{} \begin{array}{|l|} \hline E \rightarrow E + T \mid E - T \mid T \\ T \rightarrow T * F \mid T / F \mid F \\ F \rightarrow (E) \mid id \\ \hline \end{array}$$

- | is a meta symbol to specify alternatives
- (and) are not meta symbols, they are terminal symbols

Derivations

- Derivation (推导):** Starting with the start symbol, nonterminals are rewritten using productions until only terminals remain

- Example:

- CFG: $E \rightarrow -E \mid E + E \mid E * E \mid (E) \mid id$
- A derivation (a sequence of rewrites) of `-(id)` from `E`
 - $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$

Notations

- \Rightarrow means "derives in one step"

- \Rightarrow^* means "derives in zero or more steps"

- $\alpha \Rightarrow^* \alpha$ holds for any string α
- If $\alpha \Rightarrow^* \beta$ and $\beta \Rightarrow \gamma$, then $\alpha \Rightarrow^* \gamma$
- Example: $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$ can be written as $E \Rightarrow^* -(id)$

- \Rightarrow^+ means "derives in one or more steps"

Terminologies

- If $S \Rightarrow^* \alpha$, where S is the start symbol of a grammar G , we say that α is **sentential form** of G (文法的句型)

- May contain both terminals and nonterminals, and may be empty
- Example: $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+id) \Rightarrow -(id+id)$, here all strings of grammar symbols are sentential forms

- A **sentence** (句子) of G is a sentential form with no nonterminals
- In the above example, only the last string `-(id+id)` is a sentence

- The **language generated** by a grammar is its set of sentences

句型中可含非终结符

句子中不含非终结符

语言即语法的句子的集合。

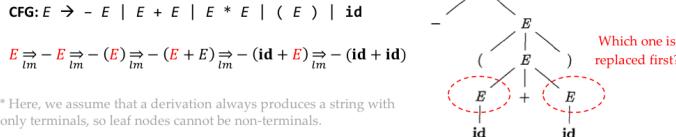
Leftmost/Rightmost Derivations

- At each step of a derivation, we need to choose which nonterminal to replace
- In **leftmost derivations** (最左推导), the leftmost nonterminal in each sentential form is always chosen to be replaced
 - $E \xrightarrow{lm} - E \xrightarrow{lm} - (E) \xrightarrow{lm} - (E + E) \xrightarrow{lm} - (\text{id} + E) \xrightarrow{lm} - (\text{id} + \text{id})$
- In **rightmost derivations** (最右推导), the rightmost nonterminal is always chosen to be replaced
 - $E \xrightarrow{rm} - E \xrightarrow{rm} - (E) \xrightarrow{rm} - (E + E) \xrightarrow{rm} - (E + \text{id}) \xrightarrow{rm} - (\text{id} + \text{id})$

每次最左边的会被替代

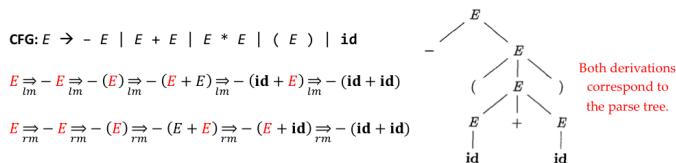
Parse Trees

- A **parse tree** is a graphical representation of a derivation that filters out the order in which productions are applied
 - The **root node** (根结点) is the start symbol of the grammar
 - Each **leaf node** (叶子结点) is labeled by a terminal symbol
 - Each **interior node** (内部结点) is labeled with a nonterminal symbol and represents the application of a production
 - The interior node is labeled with the nonterminal in the head of the production; the children nodes are labeled, from left to right, by the symbols in the body of the production



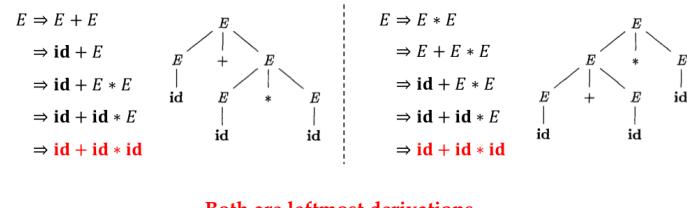
* Here, we assume that a derivation always produces a string with only terminals, so leaf nodes cannot be non-terminals.

- The leaves, from left to right, constitute a **sentential form** of the grammar, which is called the **yield** or **frontier** of the tree
- There is a **many-to-one** relationship between **derivations** and **parse trees**
 - However, there is a **one-to-one** relationship between **leftmost/rightmost derivations** and **parse trees**



Ambiguity (二义性)

- Given a grammar, if there are **more than one parse tree for some sentence**, it is ambiguous.
- Example CFG: $E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$



The left tree corresponds to the commonly assumed precedence.

根节点：起始符号
叶子节点：终结符号
内部节点：非终结符号

叶子从左到右构建了语法的句型

转换规则与解析树是多对一的关系。
但 最左/最右转换规则与解析树是一对一的关系。

对某个句子有不只是一个解析树。

- The grammar of a programming language typically needs to be unambiguous
 - Otherwise, there will be multiple ways to interpret a program
 - Given $E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$, how to interpret $a + b * c$?
- In some cases, it is convenient to use carefully chosen ambiguous grammars, together with disambiguating rules to discard undesirable parse trees
 - For example: multiplication before addition

CFG vs. Regular Expression

• CFGs are more expressive than regular expressions

- Every language that can be described by a regular expression can also be described by a grammar (i.e., every regular language is also a context-free language)
- Some context-free languages cannot be described using regular expressions

需要某些规则来避免二义性。

任何正则表达式都能用上下文无关文法表达

有的上下文无关文法不能被正则表达式表达

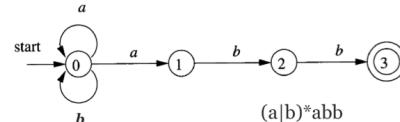
Any Regular Language Can be Described by a CFG

• (Proof by Construction) Each regular language can be accepted by an NFA. We can construct a CFG to describe the language:

- For each state i of the NFA, create a nonterminal symbol A_i
- If state i has a transition to state j on input a , add the production $A_i \rightarrow aA_j$
- If state i goes to state j on input ϵ , add the production $A_i \rightarrow A_j$
- If i is an accepting state, add $A_i \rightarrow \epsilon$
- If i is the start state, make A_i be the start symbol of the grammar

Example: NFA to CFG

- $A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$
- $A_1 \rightarrow bA_2$
- $A_2 \rightarrow bA_3$
- $A_3 \rightarrow \epsilon$



Consider the string **baabb**: The process of the NFA accepting the sentence corresponds exactly to the derivation of the sentence from the grammar

Some Context-Free Languages Cannot be Described Using Regular Expressions

• Example: $L = \{a^n b^n \mid n > 0\}$

- The language L can be described by CFG $S \rightarrow aSb \mid ab$
- L cannot be described by regular expressions. In other words, we cannot construct a DFA to accept L

Proof by Contradiction

- Suppose there is a DFA D that accepts L and D has k states
- When processing $a^{k+1} \dots$, D must enter a state s more than once (D enters one state after processing a symbol)¹
- Assume that D enters the state s after reading the i th and j th a ($i \neq j, i \leq k+1, j \leq k+1$)
- Since D accepts L , $a^j b^j$ must reach an accepting state. There must exist a path labeled b^j from s to an accepting state
- Since a^i reaches the state s and there is a path labeled b^j from s to an accepting state, D will accept $a^i b^j$. Contradiction!!!

¹ $a^{k+1} b^{k+1}$ is a string in L so D must accept it

Grammar Design

- CFGs are capable of describing most, but not all, of the syntax of programming languages
 - “Identifiers should be declared before use” cannot be described by a CFG
 - Subsequent phases must analyze the output of the parser to ensure compliance with such rules
 - Before parsing, we typically apply several transformations to a grammar to make it more suitable for parsing

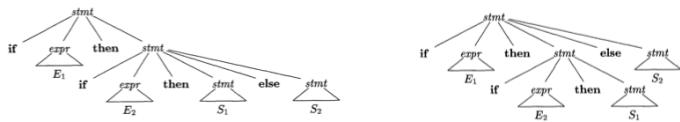
上下文无关文法能描述大多数编程语言。

在parse程序前，需要应用多种转换。

Eliminating ambiguity

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

Two parse trees for `if E1 then if E2 then S1 else S2`



**Which parse tree is preferred in programming?
(i.e., else matches which then?)**

- **Principle of proximity:** match each `else` with the closest unmatched `then`
 - **Idea of rewriting:** A statement appearing between a `then` and an `else` must be matched (must not end with an unmatched `then`)

stmt → *matched_stmt*
 | *open_stmt*

A simple line drawing of a hand with fingers slightly spread, used as a placeholder or icon.

Eliminating Left Recursion

- A grammar is **left recursive** if it has a nonterminal A such that there is a derivation $A \xrightarrow{+} A\alpha$ for some string α
 - $S \rightarrow Aa \mid b$
 - $A \rightarrow Ac \mid Sd \mid \epsilon$
 - Because $S \Rightarrow Aa \Rightarrow Sda$
 - **Immediate left recursion (立即左递归):** the grammar has a production of the form $A \rightarrow A\alpha$
 - Top-down parsing methods cannot handle left-recursive grammars (bottom-up parsing methods can handle...)

消除二义性没有基本准则

左邊曰： $A \stackrel{+}{\Rightarrow} A \times$

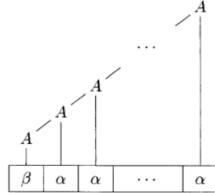
立即左递归：有推导式 $A \rightarrow A\alpha$

自顶向下的解析方法无法解决。

Eliminating Immediate Left Recursion

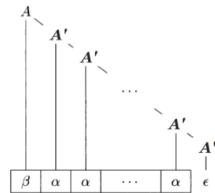
- Simple grammar: $A \rightarrow A\alpha \mid \beta$

- It generates sentences starting with the symbol β followed by zero or more α 's



- Replace the grammar by:

- $A \rightarrow \beta A'$
- $A' \rightarrow \alpha A' \mid \epsilon$
- It is right recursive now



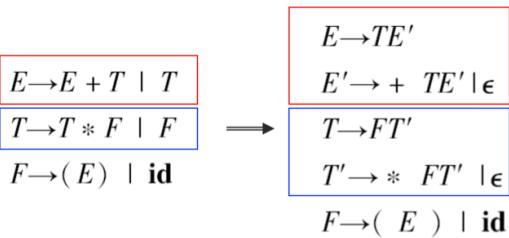
- The general case: $A \rightarrow A\alpha_1 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \dots \mid \beta_n$

- Replace the grammar by:

- $A \rightarrow \beta_1 A' \mid \dots \mid \beta_n A'$
- $A' \rightarrow \alpha_1 A' \mid \dots \mid \alpha_m A' \mid \epsilon$

将左递归转成右递归。

Example



Eliminating Left Recursion

- The technique for eliminating immediate left recursion does not work for the non-immediate left recursions
- The general left recursion eliminating algorithm (iterative)

- Input:** Grammar G with no cycles or ϵ -productions
- Output:** An equivalent grammar with no left recursion

arrange the nonterminals in some order A_1, A_2, \dots, A_n .

```
for ( each i from 1 to n ) {
    for ( each j from 1 to i - 1 ) {
        replace each production of the form  $A_i \rightarrow A_j \gamma$  by the
        productions  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$ , where
         $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all current  $A_j$ -productions
    }
    eliminate the immediate left recursion among the  $A_i$ -productions
}
```

将非立即左递归都转成立即左递归，然后
再应用立即左递归的解决方式。

→ 将推导式展开至终结符或递归

→ 做消除 (应用 general case)

Example

$$S \rightarrow Aa \mid b \quad A \rightarrow Ac \mid Sd \mid \epsilon$$

- Order the nonterminals: S, A

- $i = 1$:

- The inner loop does not run; there is no immediate left recursion among S -productions

- $i = 2$:

- $j = 1$, replace the production $A \rightarrow Sd$ by $A \rightarrow Aad \mid bd$
 - $A \rightarrow Aad \mid bd \mid Ac \mid \epsilon$

- Eliminate immediate left recursion

$$\begin{array}{l} S \rightarrow Aa \mid b \\ A \rightarrow bdA' \mid A' \\ A' \rightarrow cA' \mid ada' \mid \epsilon \end{array}$$

Left Factoring

- If we have the following two productions
 $stmt \rightarrow if\ expr\ then\ stmt\ else\ stmt$
| $if\ expr\ then\ stmt$
- On seeing input **if**, we cannot immediately decide which production to choose
- In general, if $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ are two productions, and the input begins with a nonempty string derived from α . We may defer choosing productions by expanding A to $\alpha A'$ first

$$\begin{array}{l} A \rightarrow \alpha A' \\ A' \rightarrow \beta_1 \mid \beta_2 \end{array}$$

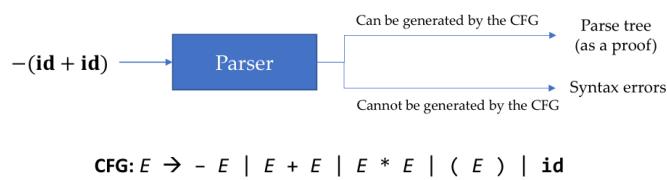
提取出左边的公因子.

Algorithm: Left Factoring a Grammar

- Input:** Grammar G
- Output:** An equivalent left-factored grammar
- For each nonterminal A , find the longest prefix α common to two or more of its alternatives.
- If $\alpha \neq \epsilon$, replace all 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 the above transformation until no two alternatives for a nonterminal have a common prefix

Parsing Revisited

- During program compilation, the syntax analyzer (a.k.a. parser) checks whether the string of token names produced by the lexer can be generated by the grammar for the source language
 - That is, if we can find a parse tree whose frontier is equal to the string, then the parser can declare "success"



若无法生成 parse tree, 即有语法错误.

Top-Down Parsing

- Problem definition:** Constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder (depth-first)

- Two basic actions of top-down parsing algorithms:**

- Predict:** At each step of parsing, determine the production to be applied for the **leftmost nonterminal*** (of the parse tree's frontier)
- Match:** Match the terminals in the chosen production's body with the input string

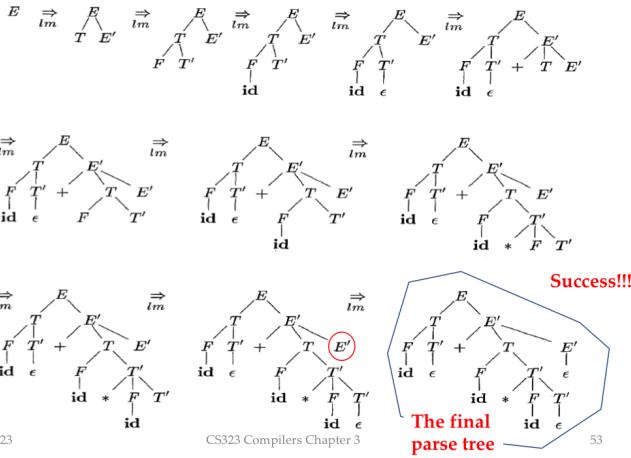
预测: 在每次分析时, 选用正确的推导式

匹配: 将输入字符串与推导式匹配

* So that the sentential forms may contain leading terminals to match with the prefix of the input string

Example

- Grammar:** $E \rightarrow TE' \quad E' \rightarrow +TE' \mid \epsilon \quad T \rightarrow FT' \quad T' \rightarrow *FT' \mid \epsilon \quad F \rightarrow (E) \mid id$
- Input string:** $id + id * id$ The sentential form after rewrite: $id + id * id$



Fall 2023

CS323 Compilers Chapter 3

53

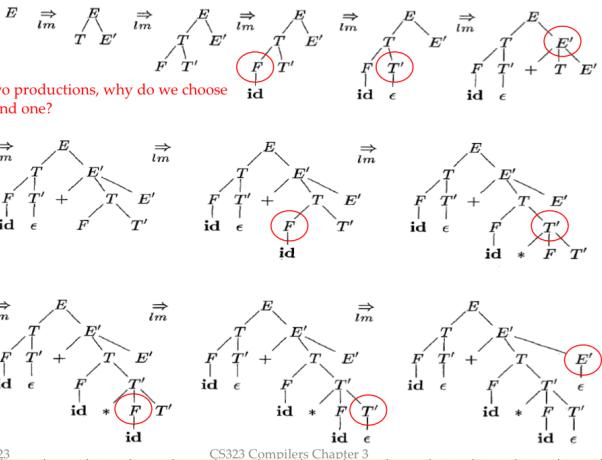
We can make two observations from the example:

- Top-down parsing is equivalent to **finding a leftmost derivation**.
- At each step, the frontier of the tree is a left-sentential form.

自顶向下的分析即为寻找最左推导。

Key decision in top-down parsing: Which production to apply at each step?

Grammar: $E \rightarrow TE' \quad E' \rightarrow +TE' \mid \epsilon \quad T \rightarrow FT' \quad T' \rightarrow *FT' \mid \epsilon \quad F \rightarrow (E) \mid id$



F has two productions, why do we choose the second one?

Fall 2023

CS323 Compilers Chapter 3

为什么选用特定的推导式：需要一定的预测

Bottom-Up Parsing

- Problem definition:** Constructing a parse tree for an input string beginning at the leaves (**terminals**) and working up towards the root (**start symbol of the grammar**)
- Shift-reduce parsing (移入-归约分析)** is a **general style** of bottom-up parsing (using a **stack** to hold grammar symbols). Two basic actions:
 - Shift:** Move an input symbol onto the stack
 - Reduce:** Replace a string at the stack top with a non-terminal that can produce the string (the reverse of a rewrite step in a derivation)

从叶子节点出发，往上合成根节点

移入：将输入字符串入栈中

归约：将栈顶的字符串用非终结符替代。

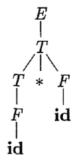
Example

Parsing steps on input $\text{id}_1 * \text{id}_2$

STACK	INPUT	ACTION
\$	$\text{id}_1 * \text{id}_2 \$$	shift
$\$ \text{id}_1$	$* \text{id}_2 \$$	reduce by $F \rightarrow \text{id}$
$\$ F$	$* \text{id}_2 \$$	reduce by $T \rightarrow F$
$\$ T$	$* \text{id}_2 \$$	shift
$\$ T * \text{id}_2$	$\$$	shift
$\$ T * \text{id}_2$	$\$$	reduce by $F \rightarrow \text{id}$
$\$ T * F$	$\$$	reduce by $T \rightarrow T * F$
$\$ T$	$\$$	reduce by $E \rightarrow T$
$\$ E$	$\$$	accept

Success!!!

$$\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid \text{id} \end{array}$$



The final parse tree

Parsing steps on input $\text{id}_1 * \text{id}_2$

STACK	INPUT	ACTION
\$	$\text{id}_1 * \text{id}_2 \$$	shift
$\$ \text{id}_1$	$* \text{id}_2 \$$	reduce by $F \rightarrow \text{id}$
$\$ F$	$* \text{id}_2 \$$	reduce by $T \rightarrow F$
$\$ T$	$* \text{id}_2 \$$	shift
$\$ T * \text{id}_2$	$\$$	shift
$\$ T * \text{id}_2$	$\$$	reduce by $F \rightarrow \text{id}$
$\$ T * F$	$\$$	reduce by $T \rightarrow T * F$
$\$ T$	$\$$	reduce by $E \rightarrow T$
$\$ E$	$\$$	accept

$$\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid \text{id} \end{array}$$

Rightmost derivation:
 $E \Rightarrow T$
 $\Rightarrow T * F$
 $\Rightarrow T * \text{id}$
 $\Rightarrow F * \text{id}$
 $\Rightarrow \text{id} * \text{id}$

We can make two observations from the example:

- Bottom-up parsing is equivalent to finding a rightmost derivation (in reverse).
- At each step, stack + remaining input is a right-sentential form.

Key decisions:

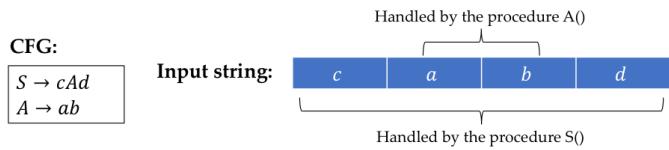
- When to shift? When to reduce?
- Which production to apply when reducing (there could be multiple possibilities)?

自底向上分析等价于寻找最右推导(逆序)
 每一步的栈+剩余输入即为右句型

Recursive-Descent Parsing (递归下降语义分析)

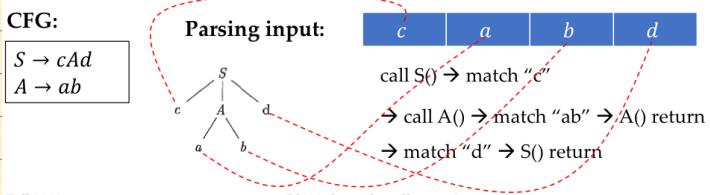
一系列过程，一个过程对应一个非终结符

- A recursive-descent parsing program has a set of procedures, one for each nonterminal
 - The procedure for a nonterminal deals with a substring of the input
- Execution begins with the procedure for the start symbol
 - Announce success if the procedure scans the entire input (the start symbol derives the whole input via applying a series of productions)



A Typical Procedure for A Nonterminal

```
void A() {
    1) Choose an A-production,  $A \rightarrow X_1X_2 \dots X_k$ ; → Predict
    2) for ( i = 1 to k ) {
        3)     if (  $X_i$  is a nonterminal )
        4)         call procedure  $X_i()$ ;
        5)     else if (  $X_i$  equals the current input symbol a )
        6)         advance the input to the next symbol;
        7)     else /* an error has occurred */;
    }
}
```



Backtracking (回溯)

```
void A() {
    1) Choose an A-production,  $A \rightarrow X_1X_2 \dots X_k$ ;
    2) for ( i = 1 to k ) {
        3)     if (  $X_i$  is a nonterminal )
        4)         call procedure  $X_i()$ ;
        5)     else if (  $X_i$  equals the current input symbol a )
        6)         advance the input to the next symbol;
    7)     else /* an error has occurred */; → If there is a failure at line 7, does this mean that there must be syntax errors?
    }
}
```



If there is a failure at line 7, does this mean that there must be syntax errors?



The failure might be caused by a wrong choice of A-production at line 1 !!!

- General recursive-descent parsing may require repeated scans over the input (backtracking)
- To allow backtracking, we need to modify the algorithm

```
void A() {
    1) Choose an A-production,  $A \rightarrow X_1X_2 \dots X_k$ ;
    2) for ( i = 1 to k ) {
        3)     if (  $X_i$  is a nonterminal )
        4)         call procedure  $X_i()$ ;
        5)     else if (  $X_i$  equals the current input symbol a )
        6)         advance the input to the next symbol;
        7)     else /* an error has occurred */;
    }
}
```

Instead of exploring one A-production, we must try each possible production in some order.

错误选取推导式也可能导致error

```

void A() {
    Choose an A-production,  $A \rightarrow X_1X_2 \cdots X_k$ ;
    for ( i = 1 to k ) {
        if (  $X_i$  is a nonterminal )
            call procedure  $X_i()$ ;
        else if (  $X_i$  equals the current input symbol a )
            advance the input to the next symbol;
        else /* an error has occurred */;
    }
}

```

When there is a failure at line 7, return to line 1 and try another A-production.

```

void A() {
    Choose an A-production,  $A \rightarrow X_1X_2 \cdots X_k$ ;
    for ( i = 1 to k ) {
        if (  $X_i$  is a nonterminal )
            call procedure  $X_i()$ ;
        else if (  $X_i$  equals the current input symbol a )
            advance the input to the next symbol;
        else /* an error has occurred */;
    }
}

```

In order to try another A-production, we must reset the input pointer that points to the next symbol to scan (failed trials consume symbols)

Example

- Grammar: $S \rightarrow cAd$ $A \rightarrow ab | a$

- Input string: cad

- S has only one production, apply it

- Grammar: $S \rightarrow cAd$ $A \rightarrow ab | a$

- Input string: cad

- Leftmost leaf matches a in input
- Advance input pointer

- Grammar: $S \rightarrow cAd$ $A \rightarrow ab | a$

- Input string: cad

- Leftmost leaf matches a in input
- Advance input pointer

- Grammar: $S \rightarrow cAd$ $A \rightarrow ab | a$

- Input string: cad

- The leftmost leaf matches c in input
- Advance input pointer

- Grammar: $S \rightarrow cAd$ $A \rightarrow ab | a$

- Input string: cad

- Symbol mismatch
- Go back to try another A-production

- Grammar: $S \rightarrow cAd$ $A \rightarrow ab | a$

- Input string: cad

- Expand A using the first production

- Grammar: $S \rightarrow cAd$ $A \rightarrow ab | a$

- Input string: cad

- Reset input pointer
- Expand A using its second production

The Problem of Left Recursion

Suppose there is only one A-production, $A \rightarrow A\alpha \dots$

```

void A() {
    Choose an A-production,  $A \rightarrow X_1X_2 \cdots X_k$ ;
    for ( i = 1 to k ) {
        if (  $X_i$  is a nonterminal )
            call procedure  $X_i()$ ; Recursively rewriting A without matching any terminals
        else if (  $X_i$  equals the current input symbol a )
            advance the input to the next symbol;
        else /* an error has occurred */;
    }
}

```

If there is **left recursion** in a CFG, a recursive-descent parser may go into an **infinite loop**! Revise the CFG before parsing!!!

遇到错误，重新选取推导式

Before calling A()

Error

在尝试另一个推导式前，要将指针移回

若CFG有左递归，就可能会造成无限递归。
需在parse前修正CFG

Avoid Backtracking

```

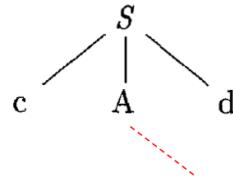
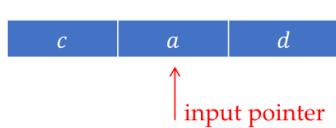
void A() {
    Choose an A-production,  $A \rightarrow X_1X_2 \dots X_k$ ;
    for ( i = 1 to k ) {
        if (  $X_i$  is a nonterminal )
            call procedure  $X_i()$ ;
        else if (  $X_i$  equals the current input symbol  $a$  )
            advance the input to the next symbol;
        else /* an error has occurred */;
    }
}

```

Key problem: At line 1, we make *random choices* (brute force search)

- Grammar: $S \rightarrow cAd \quad A \rightarrow c \mid a$

- Input string: cad



When rewriting A , is it a good idea to choose $A \rightarrow c$?

No! If we look ahead, the next char in the input is a .
 $A \rightarrow c$ is obviously a bad choice!!!

Looking Ahead (FIRST)

- Suppose the input string is $x\alpha\dots$
- Suppose the current sentential form is $x\alpha\beta$
 - α is a non-terminal; β may contain both terminals and non-terminals

If we know the following fact for the productions $A \rightarrow \alpha \mid \gamma$:

- $\alpha \in FIRST(\alpha)$: α derives strings that begin with α
- $\alpha \notin FIRST(\gamma)$: γ derives strings that do not begin with α

* $FIRST(\alpha)$ denotes the set of beginning terminals of strings derived from α

After matching x , which production should we choose to rewrite α ? $A \rightarrow \alpha$

Computing FIRST

- $FIRST(X)$, where X is a grammar symbol
 - If X is a terminal, then $FIRST(X) = \{X\}$
 - If X is a nonterminal and $X \rightarrow \epsilon$, then add ϵ to $FIRST(X)$
 - If X is a nonterminal and $X \rightarrow Y_1Y_2 \dots Y_k$ ($k \geq 1$) is a production
 - If for some i , α is in $FIRST(Y_i)$ and ϵ is in all of $FIRST(Y_1), \dots, FIRST(Y_{i-1})$, then add α to $FIRST(X)$
 - If ϵ is in all of $FIRST(Y_1), \dots, FIRST(Y_k)$, then add ϵ to $FIRST(X)$

可以在这些处做优化.

已知输入串为: $x\alpha\dots$
 当前句型为: $x\alpha\beta$ (α 为非终结符, β 未定)
 有推导式: $A \rightarrow \alpha \mid \gamma$
 若 $\alpha \in FIRST(\alpha)$, 即 α 生成的串的首字母为 α .
 且 $\alpha \notin FIRST(\gamma)$, 即 γ 生成的串的首字母不为 α .
 则选用推导式 $A \rightarrow \alpha$.

- $FIRST(X_1X_2 \dots X_n)$, where $X_1X_2 \dots X_n$ is a string of grammar symbols
 - Add all non- ϵ symbols of $FIRST(X_1)$ to $FIRST(X_1X_2 \dots X_n)$
 - If ϵ is in $FIRST(X_1)$, add non- ϵ symbols of $FIRST(X_2)$ to $FIRST(X_1X_2 \dots X_n)$
 - If ϵ is in $FIRST(X_1)$ and $FIRST(X_2)$, add non- ϵ symbols of $FIRST(X_3)$ to $FIRST(X_1X_2 \dots X_n)$
 - ...
 - If ϵ is in $FIRST(X_i)$ for all i , add ϵ to $FIRST(X_1X_2 \dots X_n)$

$FIRST(X)$ 表示从 X 推导出的字符串的首字符的集合

ϵ 可以在 $FIRST$ 中.

Example

• Grammar

- | | |
|---------------------|----------------------------------|
| $E \rightarrow TE'$ | $E' \rightarrow +TE' \epsilon$ |
| $T \rightarrow FT'$ | $T' \rightarrow *FT' \epsilon$ |
| | $F \rightarrow (E) \text{id}$ |

• FIRST sets

- $\text{FIRST}(F) = \{\), \text{id}\}$
- $\text{FIRST}(T) = \text{FIRST}(F) = \{\), \text{id}\}$
- $\text{FIRST}(E) = \text{FIRST}(T) = \{\), \text{id}\}$
- $\text{FIRST}(E') = \{+, \epsilon\}$
- $\text{FIRST}(T') = \{* , \epsilon\}$
- $\text{FIRST}(TE') = \text{FIRST}(T) = \{\), \text{id}\}$
- ...

Strings derived from F or T
must start with (or id

Looking Ahead (FOLLOW)

- Suppose the input string is $x\alpha\dots$
- Suppose the current sentential form is $xA\beta$
 - A is a non-terminal; β may contain both terminals and non-terminals

If we know that for the production $A \rightarrow \alpha, \epsilon \in \text{FIRST}(\alpha)$, can we choose the production to rewrite A ?

* $\epsilon \in \text{FIRST}(\alpha)$ means that rewriting A to α may result in an empty string (recall when we add ϵ to the FIRST set)

Yes, only if β can derive strings beginning with a , that is, A can be followed by a in some sentential forms (i.e., $a \in \text{FOLLOW}(A)$)

Computing FOLLOW

• Computing FOLLOW set for all nonterminals

- Add $\$$ in $\text{FOLLOW}(S)$, where S is the start symbol and $\$$ is the input right endmarker
- Apply the rules below, until all FOLLOW sets do not change
 1. If there is a production $A \rightarrow \alpha B \beta$, then everything in $\text{FIRST}(\beta)$ except ϵ is in $\text{FOLLOW}(B)$
 2. If there is a production $A \rightarrow \alpha B$ (or $A \rightarrow \alpha B \beta$ and $\text{FIRST}(\beta)$ contains ϵ) then everything in $\text{FOLLOW}(A)$ is in $\text{FOLLOW}(B)$

By definition, ϵ will not appear in any FOLLOW set

Example

• Grammar

- | | |
|---------------------|----------------------------------|
| $E \rightarrow TE'$ | $E' \rightarrow +TE' \epsilon$ |
| $T \rightarrow FT'$ | $T' \rightarrow *FT' \epsilon$ |
| | $F \rightarrow (E) \text{id}$ |

• FOLLOW sets

- $\text{FOLLOW}(E) = \{\$,)\}$
- $\text{FOLLOW}(E') = \{\$,)\}$
- $\text{FOLLOW}(T) = \{+, \$,)\}$
- $\text{FOLLOW}(T') = \{+, \$,)\}$
- $\text{FOLLOW}(F) = \{*, +, \$,)\}$

- \$ is always in FOLLOW(E)
- Everything in FIRST() except \$\epsilon\$ is in FOLLOW(E)

• Grammar

- | | |
|---------------------|----------------------------------|
| $E \rightarrow TE'$ | $E' \rightarrow +TE' \epsilon$ |
| $T \rightarrow FT'$ | $T' \rightarrow *FT' \epsilon$ |
| | $F \rightarrow (E) \text{id}$ |

• FOLLOW sets

- $\text{FOLLOW}(E) = \{\$,)\}$
- $\text{FOLLOW}(E') = \{\$,)\}$
- $\text{FOLLOW}(T) = \{+, \$,)\}$
- $\text{FOLLOW}(T') = \{+, \$,)\}$
- $\text{FOLLOW}(F) = \{*, +, \$,)\}$

- Everything in $\text{FIRST}(E')$ except ϵ is in $\text{FOLLOW}(T)$
- Since $E' \rightarrow \epsilon$, everything in $\text{FOLLOW}(E)$ and $\text{FOLLOW}(E')$ is in $\text{FOLLOW}(T)$

Why need FIRST & FOLLOW?

- For a production $\text{head} \rightarrow \text{body}$, when we are trying to rewrite head , if we know $\text{FIRST}(\text{body})$, that is, what terminals can strings derived from body start with, we can decide whether to choose $\text{head} \rightarrow \text{body}$ by looking at the next input symbol.
 - If the next input symbol is in $\text{FIRST}(\text{body})$, the production may be a good choice.
- For a production $\text{head} \rightarrow \epsilon$ (or head can derive ϵ in some steps), when we are trying to rewrite head , if we know $\text{FOLLOW}(\text{head})$, that is, what terminals can follow head in various sentential forms, we can decide whether to choose $\text{head} \rightarrow \epsilon$ by looking at the next input symbol.
 - If the next input symbol is in $\text{FOLLOW}(\text{head})$, the production may be a good choice.

LL(1) Grammars

- Recursive-descent parsers needing no backtracking can be constructed for a class of grammars called **LL(1)**

- 1st L: scanning the input from left to right
- 2nd L: producing a leftmost derivation (top-down parsing)
- 1: using one input symbol of lookahead at each step to make parsing decision

A grammar G is LL(1) if and only if for any two distinct productions $A \rightarrow \alpha \mid \beta$, the following conditions hold:

- There is no terminal a such that α and β derive strings beginning with a
- At most one of α and β can derive the empty string
- If $\beta \stackrel{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with a terminal in $\text{FOLLOW}(A)$ and vice versa

* The three conditions essentially rule out the possibility of applying both productions so that there is a unique choice of production at each "predict" step by looking at the next input symbol

More formally:

- $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$ (conditions 1-2 above)
- If $\epsilon \in \text{FIRST}(\beta)$, then $\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \emptyset$ and vice versa

- For LL(1) grammars, during recursive-descent parsing, the proper production to apply for a nonterminal can be selected by looking only at the current input symbol

Grammar: $\text{stmt} \rightarrow \underline{\text{if(expr)}} \text{ stmt } \underline{\text{else}} \text{ stmt} \mid \underline{\text{while(expr)}} \text{ stmt } \mid \underline{a}$

(1)

(2)

只有当前符号即可选出推导式

Parsing steps for input: $\text{if(expr)} \text{ while(expr)} \text{ a else a}$

- Rewrite the start symbol stmt with ①: $\text{if(expr)} \text{ stmt } \text{else} \text{ stmt}$
- Rewrite the leftmost stmt with ②: $\text{if(expr)} \text{ while(expr)} \text{ stmt } \text{else} \text{ stmt}$
- Rewrite the leftmost stmt with ③: $\text{if(expr)} \text{ while(expr)} \text{ a else} \text{ stmt}$
- Rewrite the leftmost stmt with ③: $\text{if(expr)} \text{ while(expr)} \text{ a else a}$

若下一输入字符在 $\text{FIRST}(\text{body})$ 中时，应用 $\text{head} \rightarrow \text{body}$

若存在 $\text{head} \xrightarrow{*} \epsilon$ ，则需要看 $\text{FOLLOW}(\text{head})$ 。

不需要回溯的递归下降生成 LL(1)，即每次选取推导式时选择唯一

从左至右读

产生最左推导

在每一步提前看下一个符号。

任两个非终结符推导出来的字符串没有相同的前缀字符

Parsing Table

- We can build parsing tables for recursive-descent parsers (**LL parsers**)
- A predictive **parsing table** is a two-dimensional array that determines which production the parser should choose when it sees a nonterminal A and a symbol a on its input stream
- The parsing table of an LL(1) parser has **no entries with multiple productions**

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$		$F \rightarrow (E)$			

LL(1)分析表中，对固定的非终结符A与符号a，没有多个推导式。

Parsing Table Construction

The following algorithm can be applied to any CFG

• **Input:** Grammar G **Output:** Parsing table M

• **Method:**

- For each production $A \rightarrow \alpha$ of G , do the following:

- For each terminal a in $\text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$
- If ϵ is in $\text{FIRST}(\alpha)$, then for each terminal b (including the right endmarker \$) in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, b]$

- Set all empty entries in the table to **error**

Fill the table entries so that when rewriting A , we know what production to choose by checking the next input symbol

先看 FIRST
若 $\epsilon \in \text{FIRST}$, 则看 FOLLOW

Example

• **Grammar**

- $E \rightarrow TE' \quad E' \rightarrow +TE' \mid \epsilon$
- $T \rightarrow FT' \quad T' \rightarrow *FT' \mid \epsilon \quad F \rightarrow (E) \mid id$

• **FIRST sets:** $E, T, F: \{(, id\}$ $E': \{+, \epsilon\}$ $T': \{*, \epsilon\}$

• **FOLLOW sets:** $E, E': \{., \}\} \quad T, T': \{+, ., \}\} \quad F: \{*, +, ., \}\}$

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$		$F \rightarrow (E)$			

For $E \rightarrow TE'$:

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$		$F \rightarrow (E)$			

For $E' \rightarrow \epsilon$:

ϵ in $\text{FIRST}(\epsilon)$
 $\text{FOLLOW}(E')$
 $= \{\$\}$

Conflicts in Parsing Tables

• **Grammar:** $S \rightarrow iEtSS' \mid a \quad S' \rightarrow eS \mid \epsilon \quad E \rightarrow b$

- $\text{FIRST}(eS) = \{e\}$, so we add $S' \rightarrow eS$ to $M[S', e]$
- $\text{FOLLOW}(S') = \{\$, e\}$, so we add $S' \rightarrow \epsilon$ to $M[S', e]$

NON - TERMINAL	INPUT SYMBOL					
	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

- LL(1) grammar is never ambiguous.
- This grammar is not LL(1). The language has no LL(1) grammar !!!

LL(1)语法永远没有歧义

Recursive-Descent Parsing for LL(1) Grammars

```

void A() {
    Choose an A-production,  $A \rightarrow X_1 X_2 \cdots X_k$ ;
    for ( i = 1 to k ) {
        if (  $X_i$  is a nonterminal )
            call procedure  $X_i()$ ;
        else if (  $X_i$  equals the current input symbol  $a$  )
            advance the input to the next symbol;
        else /* an error has occurred */;
    }
}

```

Replace line 1 with: Choose A-production according to the parse table

- Assume input symbol is a , then the choice is the production in $M[A, a]$

How is Recursion Handled?

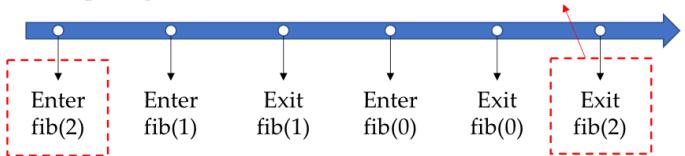
```

int fib(int n) {
    if(n <= 1) return n;
    else {
        int a = fib(n-1) + fib(n-2);
        return a;
    }
}

```

Computing $\text{fib}(2)$:

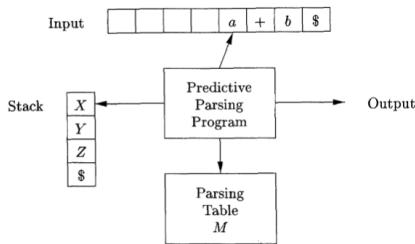
First in, last out



Non-Recursive Predictive Parsing

- A non-recursive predictive parser can be built by **explicitly maintaining a stack** (not implicitly via recursive calls)
 - Input buffer** contains the string to be parsed, ending with \$
 - Stack** holds a sequence of grammar symbols with \$ at the bottom.

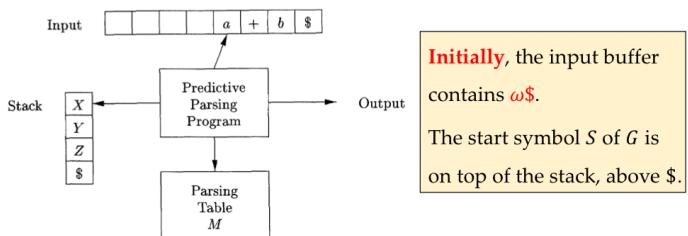
Initially, the stack contains only \$ and the start symbol S on top of \$



显式维护一个栈

Table-Driven Predictive Parsing

- Input:** A string ω and a parsing table M for grammar G
- Output:** If ω is in $L(G)$, a leftmost derivation of ω (input buffer and stack are both empty); otherwise, an error indication



输入：字符串 ω 、语法 G 的 parse table

输出：若 $\omega \in L(G)$ ，则为 ω 的最左推导

若 buffer 和 stack 有一个不为空，则 error

开始时：buffer = $\omega \$$
stack = $S \$$

Example

$$\begin{array}{lll} E \rightarrow TE' & E' \rightarrow +TE' | \epsilon \\ T \rightarrow FT' & T' \rightarrow *FT' | \epsilon \\ F \rightarrow (E) | id \end{array}$$

Example

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	
F	$F \rightarrow id$			$F \rightarrow (E)$		

MATCHED	STACK	INPUT	ACTION
	$E\$$	$id + id * id\$$	
	$TE'\$$	$id + id * id\$$	output $E \rightarrow TE'$
	$FT'E'\$$	$id + id * id\$$	output $T \rightarrow FT'$
id	$id T'E'\$$	$id + id * id\$$	output $F \rightarrow id$
id	$T'E'\$$	$+ id * id\$$	match id
id	$E'\$$	$+ id * id\$$	output $T' \rightarrow \epsilon$
$id +$	$+ TE'\$$	$+ id * id\$$	output $E' \rightarrow + TE'$
\dots	$TE'\$$	$id * id\$$	match $+$
$id + id * id$	$\$$	$\$$	output $E' \rightarrow \epsilon$

There are eight more steps before accepting.

The parser announce success when both stack and input are empty.

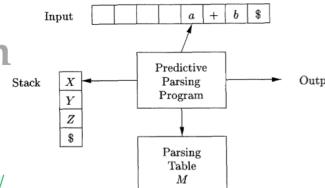
MATCHED	STACK	INPUT	ACTION
	$E\$$	$id + id * id\$$	
Leftmost derivation	$TE'\$$	$id + id * id\$$	output $E \rightarrow TE'$
	$FT'E'\$$	$id + id * id\$$	output $T \rightarrow FT'$
id	$id T'E'\$$	$id + id * id\$$	output $F \rightarrow id$
id	$T'E'\$$	$+ id * id\$$	match id
id	$E'\$$	$+ id * id\$$	output $T' \rightarrow \epsilon$
id	$+ TE'\$$	$+ id * id\$$	output $E' \rightarrow + TE'$
$id +$	$TE'\$$	$id * id\$$	match $+$
\dots	$\$$	$\$$	output $E' \rightarrow \epsilon$

Matched part
+
Stack content
(from top to bottom)
=
A left-sentential form
总是最左句型

Parsing Algorithm

Parsing Algorithm

1. let a be the first symbol of ω ;
2. let X be the top stack symbol;
3. while ($X \neq \$$) { /* stack is not empty */
4. if ($X = a$) pop the stack and let a be the next symbol of ω ;
5. else if (X is a terminal) error(); /* X can only match a , cannot be another terminal */
6. else if ($M[X, a]$ is an error entry) error();
7. else if ($M[X, a] = X \rightarrow Y_1 Y_2 \dots Y_k$) {
8. output the production $X \rightarrow Y_1 Y_2 \dots Y_k$;
9. pop the stack;
10. push Y_k, Y_{k-1}, \dots, Y_1 onto the stack, with Y_1 on top; /* order is critical */
11. }
12. let X be the top stack symbol;
13. }



Shift-Reduce Parsing (Revisit)

- Bottom-up parsing can be seen as a process of "reducing" a string ω to the start symbol of the grammar (a reverse process of derivation)
- Shift-reduce parsing is a general style of bottom-up parsing in which:
 - A **stack** holds grammar symbols
 - An **input buffer** holds the rest of the string to be parsed
 - The **stack content** (from bottom to top) and the **input buffer content** form a **right-sentential form** (assuming no errors)

Initial status:

STACK	INPUT
\$	$\omega\$$

Actions:

Shift
Reduce
Accept
Error

Shift-reduce process:

- The parser **shifts zero or more input symbols onto the stack**, until it is ready to reduce a string β on top of the stack
- Reduce** β to the head of the appropriate production



The parser repeats the above cycle until it has detected an error or the stack contains the start symbol and input is empty

The Challenge (Revisit)

Parsing steps on input $\text{id}_1 * \text{id}_2$

STACK	INPUT	ACTION
\$	$\text{id}_1 * \text{id}_2 \$$	shift
\$ id_1	$* \text{id}_2 \$$	reduce by $F \rightarrow \text{id}$
\$ F	$* \text{id}_2 \$$	reduce by $T \rightarrow F$
\$ T	$* \text{id}_2 \$$	shift
\$ $T *$	$\text{id}_2 \$$	shift
\$ $T * \text{id}_2$	$\$$	reduce by $F \rightarrow \text{id}$
\$ $T * F$	$\$$	reduce by $T \rightarrow T * F$
\$ T	$\$$	reduce by $E \rightarrow T$
\$ E	$\$$	accept

$$\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid \text{id} \end{array}$$

Why shifting * instead of reducing T ?

Generally, when to shift/reduce? How to reduce (choosing which production)?

LR Parsing

- LR(k) parsers:** the most prevalent type of bottom-up parsers
 - L: left-to-right scan of the input
 - R: construct a rightmost derivation in reverse
 - k : use k input symbols of lookahead in making parsing decisions
- LR(0) and LR(1) parsers** are of practical interest
 - When $k \geq 2$, the parser becomes too complex to construct (parsing table will be too huge to manage)

stack + buffer = right-sentential

L: 从左至右扫描

R: 反向构造出最右推导

k: 在做分析决定时提前看k个输入字符。

$k \geq 2$ 的分析表难以维护

Advantages of LR Parsers

- Table-driven (like non-recursive LL parsers) and powerful
 - Although it is too much work to construct an LR parser by hand, there are parser generators to construct parsing tables automatically
 - Comparatively, LL parsers tend to be easier to write by hand, but less powerful (handle fewer grammars)
- LR-parsing is the most general nonbacktracking shift-reduce parsing method known
- LR parsers can be constructed to recognize virtually all programming language constructs for which CFGs can be written
- LR grammars can describe more languages than LL grammars
 - Recall the stringent conditions for a grammar to be LL(1)

parser 生成器会自动构建分析表

LR 分析是最一般的非回溯移入归约分析

LR 比 LL 能描述更多的语言.

When to Shift/Reduce?

STACK	INPUT	ACTION
\$	$\text{id}_1 * \text{id}_2 \$$	shift
\$ id_1	$* \text{id}_2 \$$	reduce by $F \rightarrow \text{id}$
\$ F	$* \text{id}_2 \$$	reduce by $T \rightarrow F$
\$ T	$* \text{id}_2 \$$	shift
\$ $T *$	$\text{id}_2 \$$	shift
\$ $T * \text{id}_2$	$\$$	reduce by $F \rightarrow \text{id}$
\$ $T * F$	$\$$	reduce by $T \rightarrow T * F$
\$ T	$\$$	reduce by $E \rightarrow T$
\$ E	$\$$	accept

$$\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid \text{id} \end{array}$$

Parsing input $\text{id}_1 * \text{id}_2$



How does a shift/reduce parser know that T on stack top is a bad choice for reduction (the right action is to shift)?

LR(0) Items

- An LR parser makes shift-reduce decisions by **maintaining states** to keep track of **what have been seen** during parsing
- An **LR(0) item** (item for short) is a production with a dot at some position of the body, indicating how much we have seen at a given time point in the parsing process
 - $A \rightarrow \cdot XYZ$ $A \rightarrow X \cdot YZ$ $A \rightarrow XY \cdot Z$ $A \rightarrow XYZ \cdot$
 - $A \rightarrow X \cdot YZ$: we have just seen on the input a string derivable from X and we hope to see a string derivable from YZ next
 - The production $A \rightarrow \epsilon$ generates only one item $A \rightarrow \cdot$
- States:** sets of LR(0) items (**LR(0) 项集**)

LR(0) 项即是一个产生式，其中有一点用下划线表明有多少字符串已看过。

Canonical LR(0) Collection

- One collection of states (i.e., sets of LR(0) items), called the **canonical LR(0) collection** (**LR(0) 项集规范族**), provides the basis for constructing a DFA to make parsing decisions
- To construct canonical LR(0) collection for a grammar, we need to define:
 - An augmented grammar (增广文法)
 - Two functions: (1) CLOSURE of item sets (项集闭包) and (2) GOTO

Augmented Grammar

- Augmenting a grammar G with start symbol S
 - Introduce a new start symbol S' to take the role of S
 - Add a new production $S' \rightarrow S$
- Obviously, $L(G) = L(G')$
- Benefit:** With the augmentation, acceptance occurs when and only when the parser is about to reduce by $S' \rightarrow S$
 - Otherwise, acceptance could occur at many points since there may be multiple S -productions

为起始字符增加新的推导式

新生成的语言与原本相同

接受发生当且仅当 parser 要归约 $S' \rightarrow S$

Closure of Item Sets

- If I is a set of items for a grammar G , then $\text{CLOSURE}(I)$ is the set of items constructed from I by the two rules
 - Initially, add every item in I to $\text{CLOSURE}(I)$
 - If $A \rightarrow \alpha \cdot B\beta$ is in $\text{CLOSURE}(I)$ and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow \gamma$ to $\text{CLOSURE}(I)$, if it is not already there. Apply this rule until no more new items can be added to $\text{CLOSURE}(I)$
- Intuition:** $A \rightarrow \alpha \cdot B\beta$ indicates that we hope to see a substring derivable from $B\beta$, which has a prefix derivable from B . Therefore, we add items for all B -productions.

1. 全部都在 $\text{CLOSURE}(I)$ 中。

2. 若 $A \rightarrow \alpha \cdot B\beta \in \text{CLOSURE}(I)$, 且有 $B \rightarrow \gamma$, 则将 $B \rightarrow \gamma$ 也加入 $\text{CLOSURE}(I)$ 中。

从 B 推导出前缀。

Algorithm for $\text{CLOSURE}(I)$

// the earlier natural language description is already clear enough

```
SetOfItems CLOSURE(I) {
    J = I;
    repeat
        for ( each item  $A \rightarrow \alpha \cdot B\beta$  in  $J$  )
            for ( each production  $B \rightarrow \gamma$  of  $G$  )
                if (  $B \rightarrow \gamma$  is not in  $J$  )
                    add  $B \rightarrow \gamma$  to  $J$ ;
    until no more items are added to  $J$  on one round;
    return  $J$ ;
}
```

Example

Example

$$\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid \text{id} \end{array}$$

- Augmented grammar
 - $E' \rightarrow E$ $E \rightarrow E + T \mid T$ $T \rightarrow T * F \mid F$ $F \rightarrow (E) \mid \text{id}$
- Computing the closure of the item set $\{[E' \rightarrow \cdot E]\}$
 - Initially, $[E' \rightarrow \cdot E]$ is in the closure
 - Add $[E \rightarrow E + T]$ and $[E \rightarrow \cdot T]$ to the closure
 - Add $[T \rightarrow T * F]$ and $[T \rightarrow \cdot F]$ to the closure
 - Add $[F \rightarrow (E)]$ and $[F \rightarrow \cdot \text{id}]$ and reach fixed point

- $[E' \rightarrow \cdot E]$
- $[E \rightarrow E + T]$
- $[E \rightarrow \cdot T]$
- $[T \rightarrow T * F]$
- $[T \rightarrow \cdot F]$
- $[F \rightarrow (E)]$
- $[F \rightarrow \cdot \text{id}]$

The Function GOTO

The Function GOTO

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

- **GOTO(I, X)**, where I is a set of items and X is a grammar symbol, is defined to be the closure of the set of all items $[A \rightarrow \alpha X \cdot \beta]$ where $[A \rightarrow \alpha \cdot X\beta]$ is in I

- $\text{CLOSURE}(\{[A \rightarrow \alpha X \cdot \beta] \mid [A \rightarrow \alpha \cdot X\beta] \in I\})$

- **Example:** Computing $\text{GOTO}(I, +)$ for $I = \{[E' \rightarrow E \cdot], [E \rightarrow E \cdot + T]\}$

- There is only one item $[E \rightarrow E \cdot + T]$, in which $+$ follows.
- Then compute the $\text{CLOSURE}(\{[E \rightarrow E \cdot + T]\})$, which contains:
 - $[E \rightarrow E \cdot + T]$
 - $[T \rightarrow T * F]$, $[T \rightarrow \text{id}]$
 - $[F \rightarrow (E)]$, $[F \rightarrow \text{id}]$

若上一步在 I 中，则下一步在 GOTO 中。

Constructing Canonical LR(0) Collection

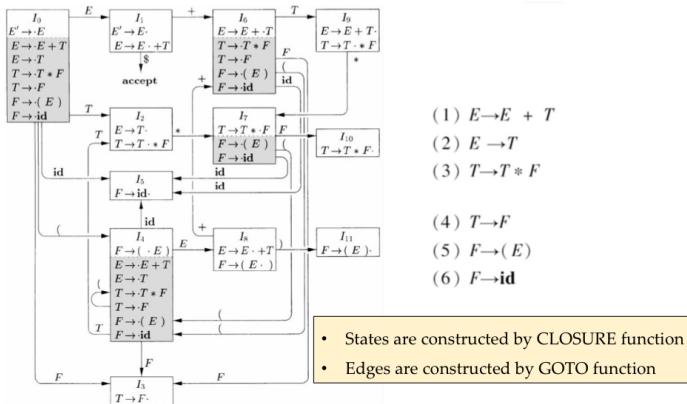
```
void items(G') {
    C = {CLOSURE({[S' → ·S]})}; Initially, there is just one item set  
(i.e., the initial state)
    repeat
        for ( each set of items I in C )
            for ( each grammar symbol X )
                if ( GOTO(I, X) is not empty and not in C )
                    add GOTO(I, X) to C;
    until no new sets of items are added to C on a round;
}
```

Iteratively find all possible GOTO targets
(states in the automaton for parsing)

Example

Example

The canonical LR(0) collection for
the grammar below is $\{I_0, I_1, \dots, I_{11}\}$



LR(0) Automaton

- The central idea behind "Simple LR", or SLR, is constructing the **LR(0) automaton** from the grammar

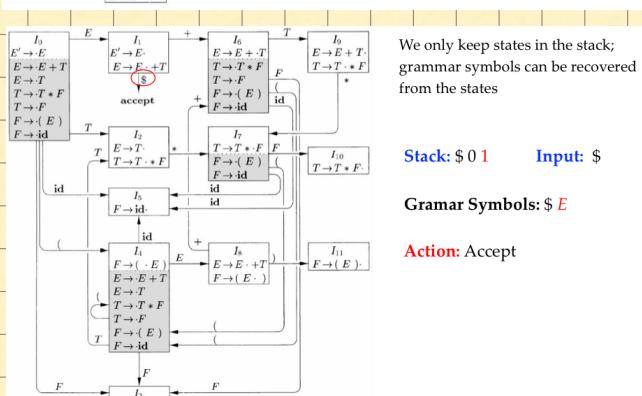
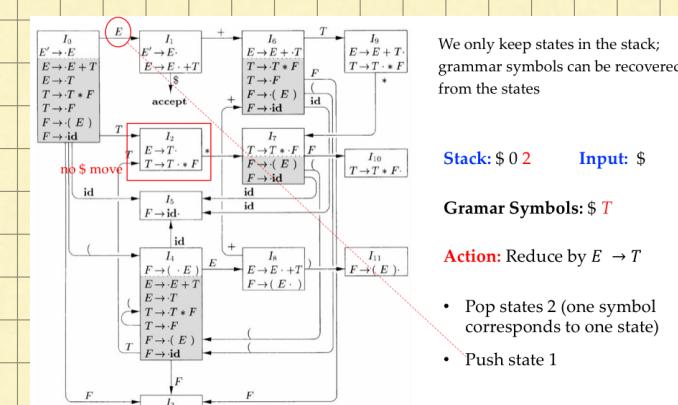
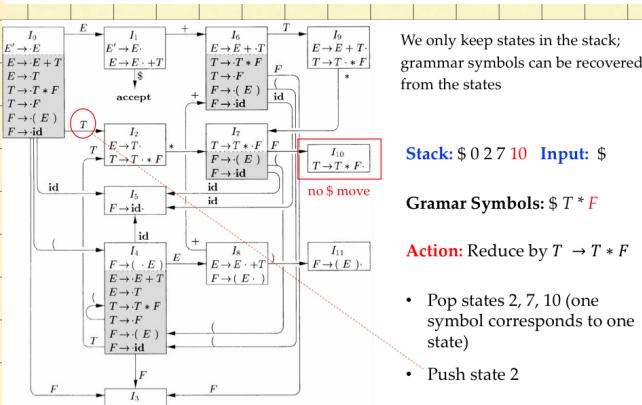
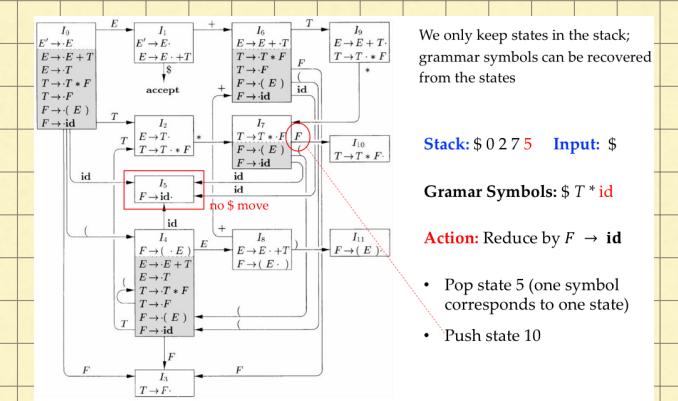
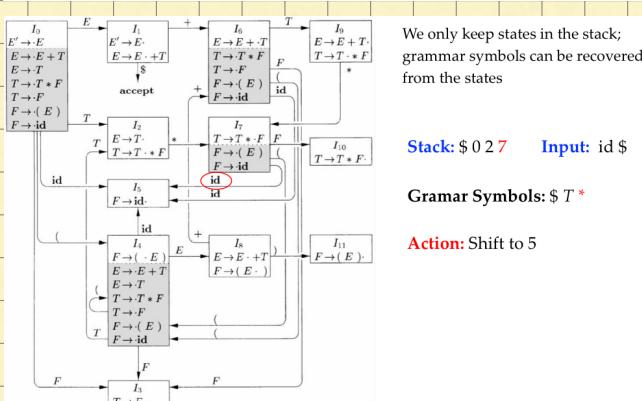
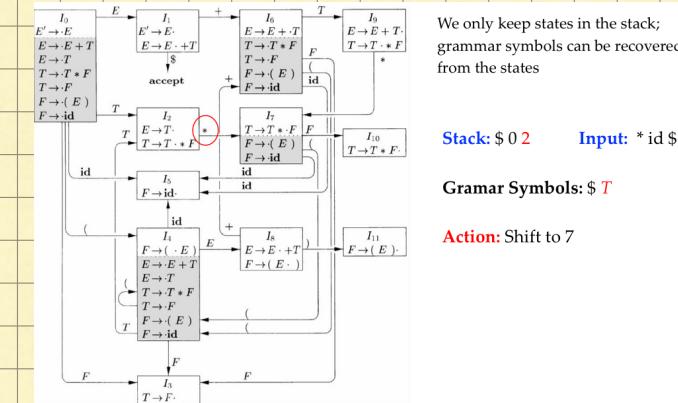
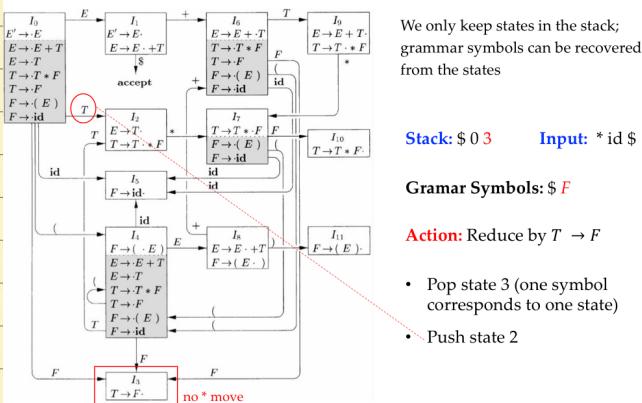
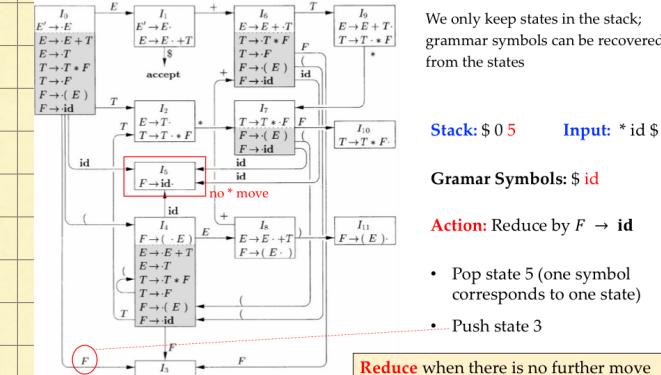
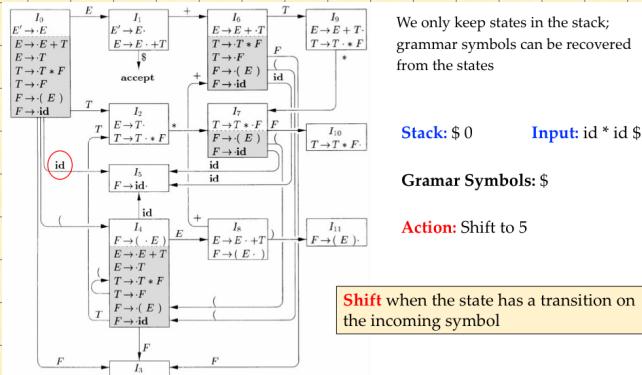
- The **states** are the item sets in the canonical LR(0) collection
- The **transitions** are given by the GOTO function
- The start state is $\text{CLOSURE}(\{[S' \rightarrow \cdot S]\})$

LR(0) automaton can effectively help make shift-reduce decisions.

SLR 就是根据文法构造出 LR(0) 自动机。
 状态：规范 LR(0) 项集中的元素。
 转移：GOTO
 起始状态： $\text{CLOSURE}(\{[S' \rightarrow \cdot S]\})$

Example: Parsing $\text{id} * \text{id}$

有 RLR shift, 无 RLR reduce

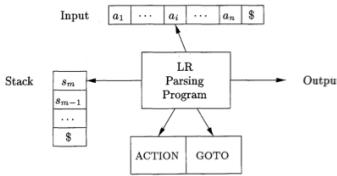


- The complete parsing steps

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$		id * id \$ shift to 5
(2)	0 5	\$ id		* id \$ reduce by $F \rightarrow \text{id}$
(3)	0 3	\$ F		* id \$ reduce by $T \rightarrow F$
(4)	0 2	\$ T		* id \$ shift to 7
(5)	0 2 7	\$ T *		id \$ shift to 5
(6)	0 2 7 5	\$ T * id		\$ reduce by $F \rightarrow \text{id}$
(7)	0 2 7 10	\$ T * F		\$ reduce by $T \rightarrow T * F$
(8)	0 2	\$ T		\$ reduce by $E \rightarrow T$
(9)	0 1	\$ E		\$ accept

LR Parser Structure

- An LR parser consists of an **input**, an **output**, a **stack**, a **driver program**, and a **parsing table** (ACTION + GOTO)
- The **driver program** is the same for all LR parsers; only the parsing table changes from one parser to another (depending on the parsing algorithm)
- The stack holds a sequence of states
 - In SLR, the stack holds states from the LR(0) automaton
- The parser decides the next action based on (1) the state at the top of the stack and (2) the next terminal read from the input buffer



Fall 2023

153

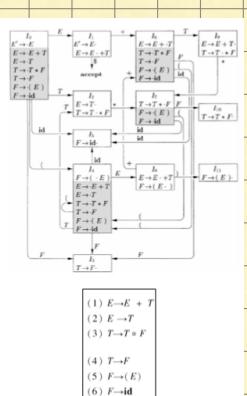
Parsing Table : ACTION + GOTO

- The **ACTION** function takes two arguments: (1) a state i and (2) a terminal a (or $\$$)
- ACTION**[i, a] can have one of the four types of values:
 - Shift j** : shift input a to the stack, and uses state j to represent a
 - Reduce $A \rightarrow \beta$** : reduce β on the top of the stack to non-terminal A
 - Accept**: The parser accepts the input and finishes parsing
 - Error**: syntax errors exist
- The **GOTO** function is obtained from the one defined on sets of items: if $\text{GOTO}(I_i, A) = I_j$, then $\text{GOTO}(i, A) = j$

Parsing Table Example

Parsing Table Example

STATE	ACTION				GOTO			
	id	+	*	()	\$	E	T	F
0	s5					1	2	3
1	s6				acc			
2	r2	s7			r2	r2		
3	r4	r4			r4	r4		
4	s5				s4		8	2
5	r6	r6			r6	r6		3
6	s5				s4		9	
7	s5				s4			10
8	s6				s11			
9	r1	s7			r1	r1		
10	r3	r3			r3	r3		
11	r5	r5			r5	r5		



- s5**: shift by pushing state 5 **r3**: reduce using production No. 3
- GOTO entries for terminals are not listed, can be checked in ACTION part

LR Parser Configurations (状态)

- "**Configuration**" is notation for representing the complete state of the parser (stack status + input status). A **configuration** is a pair:

Stack contents $(s_0 s_1 \dots s_m, a_i a_{i+1} \dots a_n \$)$ Remaining input

- By construction, each state (except s_0) in an LR parser corresponds to a set of items and a grammar symbol (the symbol that leads to the state transition, i.e., the symbol on the incoming edge)

- Suppose X_i is the grammar symbol for state s_i
- Then $X_0 X_1 \dots X_m a_i a_{i+1} \dots a_n$ is a **right-sentential form** (assume no errors)

LR parser 的驱动程序是一样的。
parsing table 随 parser 不同而不同。

(1) 栈顶的状态

(2) 从输入中读入的下一终结字符

Configuration 表示了 parser 的状态

每一个状态对应着一个进入的 symbol

Behavior of the LR Parser

- For the configuration $(s_0 s_1 \dots s_m, a_i a_{i+1} \dots a_n \$)$, the LR parser checks ACTION $[s_m, a_i]$ in the parsing table to decide the parsing action
 - shift s :** shift the next state s onto the stack, entering the configuration $(s_0 s_1 \dots s_m s, a_{i+1} \dots a_n \$)$
 - reduce $A \rightarrow \beta$:** execute a reduce move, entering the configuration $(s_0 s_1 \dots s_{m-r}s, a_i a_{i+1} \dots a_n \$)$, where $r = \text{length of } \beta$, and $s = \text{GOTO}(s_{m-r}, A)$
 - accept:** parsing successful
 - error:** the parser has found an error and calls an error recovery routine

LR-Parsing Algorithm

- Input:** The parsing table for a grammar G and an input string ω
- Output:** If ω is in $L(G)$, the reduction steps of a bottom-up parse for ω ; otherwise, an error indication
- Initial configuration:** $(s_0, \omega \$)$

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
    let s be the state on top of the stack;
    if ( ACTION[s, a] = shift t ) {
        push t onto the stack;
        let a be the next input symbol;
    } else if ( ACTION[s, a] = reduce A → β ) {
        pop |β| symbols off the stack;
        let state t now be on top of the stack;
        push GOTO[t, A] onto the stack;
        output the production A → β;
    } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
    else call error-recovery routine;
}
```

Fall 2023

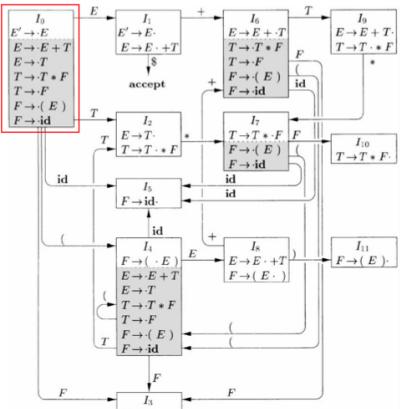
Constructing SLR-Parsing Tables

- The SLR-parsing table for a grammar G can be constructed based on the LR(0) item sets and LR(0) automaton
 - Construct the canonical LR(0) collection $\{I_0, I_1, \dots, I_n\}$ for the augmented grammar G'
 - State i is constructed from I_i . ACTION can be determined as follows:
 - If $[A \rightarrow \alpha \cdot a\beta]$ is in I_i and $\text{GOTO}[I_i, a] = I_j$, then set ACTION $[i, a]$ to "shift j " (here a must be a terminal)
 - If $[A \rightarrow \alpha \cdot]$ is in I_i , then set ACTION $[i, a]$ to "reduce $A \rightarrow \alpha$ " for all a in FOLLOW(A); here A may not be S'
 - If $[S' \rightarrow S \cdot]$ is in I_i , then set ACTION $[i, \$]$ to "accept"
 - The goto transitions for state i are constructed for all nonterminals A using the rule: If $\text{GOTO}(I_i, A) = I_j$, then $\text{GOTO}(i, A) = j$
 - All entries not defined in steps 2 and 3 are set to "error"
 - Initial state is the one constructed from the item set containing $[S' \rightarrow S]$

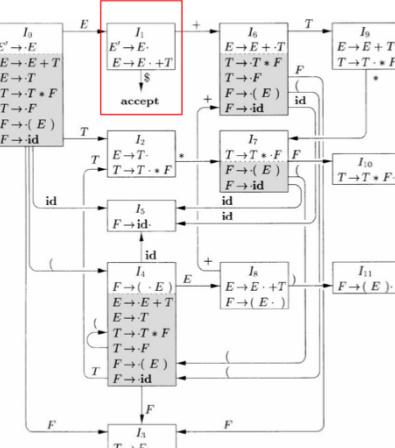
If there is no conflict during the parsing table construction (i.e., multiple actions for a table entry), the grammar is **SLR(1)**

构造分析表时若没有矛盾，则为SLR(1)

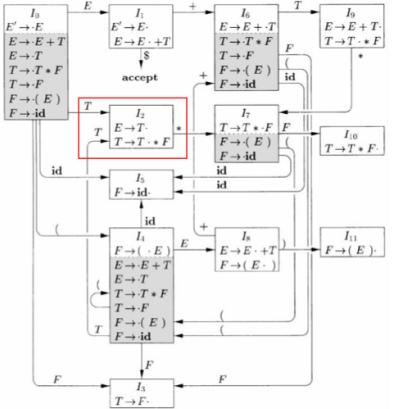
Example



- ACTION($0, \epsilon$) = s_4 (shift 4)
 - ACTION($0, \mathbf{id}$) = s_5
 - GOTO[$0, E$] = 1
 - GOTO[$0, T$] = 2
 - GOTO[$0, F$] = 3



- ACTION(1, +) = s6
 - ACTION(1, \$) = accept



- ACTION(2,*) = $s7$
 - ACTION(2, \$) = reduce $E \rightarrow T$
 - ACTION(2, +) = reduce $E \rightarrow T$
 - ACTION(2,)) = reduce $E \rightarrow T$

$$\text{FOLLOW}(E) = \{\$, +,)\}$$

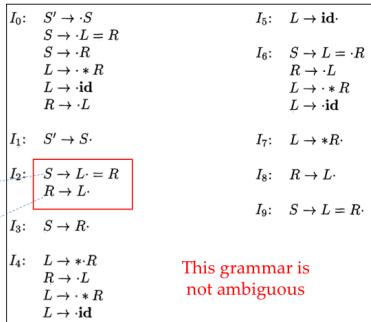
Non-SLR Grammar

- Grammar

- $S \rightarrow L = R \mid R$
 - $L \rightarrow^* R \mid \mathbf{id}$
 - $R \rightarrow L$

- For item set I_3 :

- According to item #1:
ACTION[2,=] is "s6"
 - According to item #2:
ACTION[2,=] is "reduce $R \rightarrow L$ "
(FOLLOW(R) contains =)



CLR and LALR will succeed on a larger collection of grammars, including the above one. However, there exist unambiguous grammars for which every LR parser construction method will encounter conflicts.

Weakness of the SLR Method

- In SLR, the state i calls for reduction by $A \rightarrow \alpha$ if (1) the item set I_i contains item $[A \rightarrow \alpha \cdot]$ and (2) input symbol a is in $\text{FOLLOW}(A)$
 - In some situations, after reduction, the content $\beta\alpha$ on stack top would become βA that cannot be followed by a in any right-sentential form* (i.e., only requiring " a is in $\text{FOLLOW}(A)$ " is not enough)

* Although SLR algorithm requires a to belong to FOLLOW(A), it is still too casual as the stack content below A is not considered.

Example

Example: Parsing $\text{id} = \text{id}$

- $S \rightarrow L = R \mid R$
- $L \rightarrow *R \mid \text{id}$
- $R \rightarrow L$

$I_0: S' \rightarrow \cdot S$	$I_5: L \rightarrow \text{id} \cdot$
$S \rightarrow \cdot L = R$	$I_6: S \rightarrow L = \cdot R$
$S \rightarrow \cdot R$	$R \rightarrow \cdot L$
$L \rightarrow \cdot *R$	$L \rightarrow \cdot *R$
$L \rightarrow \cdot \text{id}$	$L \rightarrow \text{id}$
$R \rightarrow \cdot L$	
$I_1: S' \rightarrow S \cdot$	$I_7: L \rightarrow *R \cdot$
$I_2: S \rightarrow L = R$	$I_8: R \rightarrow L \cdot$
$R \rightarrow L \cdot$	$I_9: S \rightarrow L = R \cdot$
$I_3: S \rightarrow R \cdot$	
$I_4: L \rightarrow *R$	
$R \rightarrow \cdot L$	
$L \rightarrow \cdot *R$	
$L \rightarrow \cdot \text{id}$	

Stack	Symbols	Input	Action
\$0		$\text{id} = \text{id}$	Shift 5
\$05	id	$= \text{id}$	Reduce by $L \rightarrow \text{id}$
\$02	L	$= \text{id}$	Suppose reduce by $R \rightarrow L$
\$03	R	$= \text{id}$	Error!

Cannot shift, cannot reduce since $\text{FOLLOW}(S) = \{\$\}$

Problem: SLR reduces too casually

How to know if a reduction is a good move?
Utilize the next input symbol to precisely determine whether to call for a reduction.

LR(1) Item

- Idea:** Carry more information in the state to rule out some invalid reductions (**splitting LR(0) states**)

- General form of an LR(1) item: $[A \rightarrow \alpha \cdot \beta, a]$

- $A \rightarrow \alpha\beta$ is a production and a is a terminal or $\$$
- "1" refers to the length of the 2nd component: the *lookahead* (向前看字符)*
- The lookahead symbol has no effect if β is not ϵ since it only helps determine whether to reduce (α will be inherited during state transitions)
- An item of the form $[A \rightarrow \alpha \cdot, a]$ calls for a reduction by $A \rightarrow \alpha$ only if the next input symbol is a (the set of such a 's is a **subset** of $\text{FOLLOW}(A)$)

*: LR(0) items do not have lookahead symbols, and hence they are called LR(0)

Constructing LR(1) Item Sets(1)

- Constructing the collection of LR(1) item sets is essentially the same as constructing the canonical collection of LR(0) item sets. The only differences lie in the **CLOSURE** and GOTO functions.

```
SetOfItems CLOSURE(I) {
    repeat
        for ( each item  $[A \rightarrow \alpha \cdot B\beta, a]$  in I )
            for ( each production  $B \rightarrow \gamma$  in  $G'$  )
                for ( each terminal  $b$  in FIRST( $\beta a$ ) )
                    add  $[B \rightarrow \cdot \gamma, b]$  to set I;
    until no more items are added to I;
    return I;
}

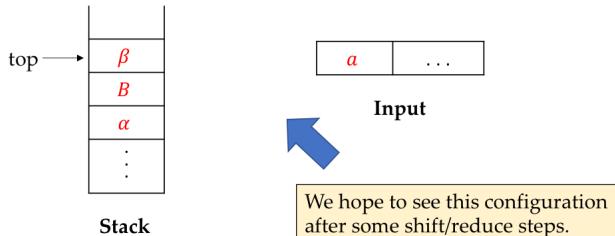
SetOfItems CLOSURE(I) {
    J = I;
    repeat
        for ( each item  $A \rightarrow \alpha \cdot B\beta$  in J )
            for ( each production  $B \rightarrow \gamma$  of  $G'$  )
                if (  $B \rightarrow \gamma$  is not in J )
                    add  $B \rightarrow \gamma$  to J;
    until no more items are added to J on one round;
    return J;
}
```

It only generates the new item $[B \rightarrow \cdot \gamma, b]$ from $[A \rightarrow \alpha \cdot B\beta, a]$ if b is in $\text{FIRST}(\beta a)$

Why b should be in $\text{FIRST}(\beta a)$?

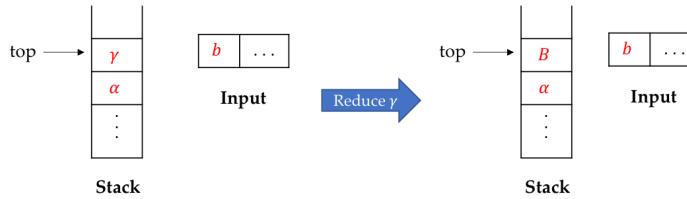
- The item $[A \rightarrow \alpha \cdot B\beta, a]$ will derive $[A \rightarrow \alpha B\beta \cdot, a]$, which calls for reduction when the stack top contains $\alpha B\beta$ and the next input symbol is a

1 指多看一个字符。
该字符只帮助决定是否归约。



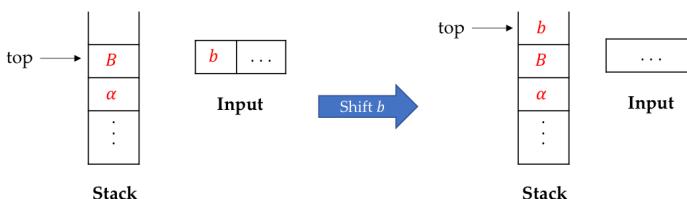
反证法

- When generating the item $[B \rightarrow \gamma, b]$ from $[A \rightarrow \alpha \cdot B\beta, a]$, suppose we allow that b is not in $FIRST(\beta a)$
 - We add the item $[B \rightarrow \gamma, b]$ because we hope that at certain time point during parsing, when we see γ on stack top and b as the next input symbol, we can first reduce γ to B so that in some later step the stack top would contain $\alpha B\beta$ (then we can further reduce it to A)



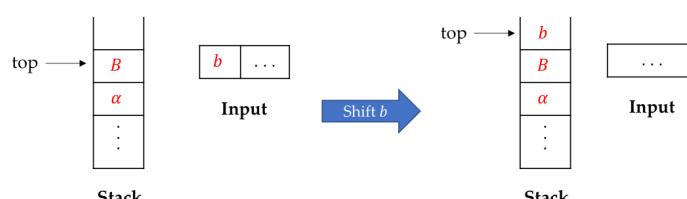
- If we reduce γ to B, the next action would be “shift b to the stack”

- Because the production $A \rightarrow \alpha B \beta$ tells us that we are ready for reduction only when we see $\alpha B \beta$ on stack top



- Since b is not in $\text{FIRST}(\beta\alpha)$, the stack top will never become the form $\alpha B \beta$, which means we will never be able to reduce $\alpha B \beta$ to A .

- Then why should we generate $[B \rightarrow \gamma, b]$ from $[A \rightarrow \alpha \cdot B\beta, a]$ in the first place???



Constructing LR(1) Item Sets 12

- Constructing the collection of LR(1) item sets is essentially the same as constructing the canonical collection of LR(0) item sets. The only differences lie in the CLOSURE and GOTO functions.

```

SetOfItems GOTO( $I, X$ ) {
    initialize  $J$  to be the empty set;
    for ( each item  $[A \rightarrow \alpha X \beta, a]$  in  $I$  )
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;
    return CLOSURE( $J$ );
}

```

GOTO(I, X) in LR(0) item sets:
 The closure of the set of all items
 $[A \rightarrow \alpha X \cdot \beta]$ where $[A \rightarrow \alpha \cdot X\beta]$ is
 in I .

The lookahead symbols are passed to new items from existing items.

Constructing LR(1) Item Sets (3)

```

void items(G) {
    C = CLOSURE( { [S' → ·S] } );
repeat
    for ( each set of items I in C )
        for ( each grammar symbol X )
            if ( GOTO(I, X) is not empty and not in C )
                add GOTO(I, X) to C;
until no new sets of items are added to C on a round;

```

Constructing the collection of LR(0) item sets

```

void items(G') {
    initialize C to {CLOSURE({[S' → ·S, $]})};
    repeat
        for ( each set of items I in C )
            for ( each grammar symbol X )
                if ( GOTO(I, X) is not empty and not in C
                    add GOTO(I, X) to C;
    until no new sets of items are added to C;
}

```

Constructing the collection of LR(1) item sets

GOTO并无太大变化

LR(1) Item Sets Example

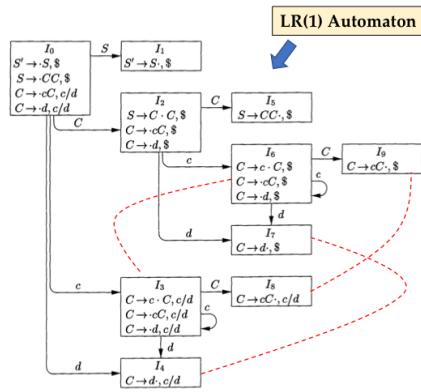
- Augmented grammar:

$$S' \rightarrow S \quad S \rightarrow CC \quad C \rightarrow cC \mid d$$

- Constructing I_0 item set and GOTO function:

- $I_0 = \text{CLOSURE}(\{S' \rightarrow S, \$\}) =$
 - $\{[S' \rightarrow S, \$], [S \rightarrow CC, \$], [C \rightarrow cC, c/d], [C \rightarrow d, c/d]\}$
- $\text{FIRST}(\$) = \{\$\}$
- $\text{FIRST}(C\$) = \{c, d\}$
- $\text{GOTO}(I_0, S) = \text{CLOSURE}(\{[S' \rightarrow S \cdot, \$]\}) = \{[S' \rightarrow S \cdot, \$]\}$
- $\text{GOTO}(I_0, C) = \text{CLOSURE}(\{[S \rightarrow C \cdot, C, \$]\}) =$
 - $\{[S \rightarrow C \cdot, C, \$], [C \rightarrow cC, \$], [C \rightarrow d, \$]\}$
- $\text{FIRST}(\$) = \{\$\}$
- $\text{GOTO}(I_0, c) = \text{CLOSURE}(\{[C \rightarrow c \cdot, C, c/d]\}) =$
 - $\{[C \rightarrow c \cdot, C, c/d], [C \rightarrow cC, c/d], [C \rightarrow d, c/d]\}$
- $\text{GOTO}(I_0, d) = \text{CLOSURE}(\{[C \rightarrow d \cdot, c/d]\}) = \{[C \rightarrow d \cdot, c/d]\}$

The GOTO Graph Example



Constructing Canonical LR(1) Parsing Tables

1. Construct $C' = \{I_0, I_1, \dots, I_n\}$, the collection of LR(1) item sets for the augmented grammar G'

2. State i of the parser is constructed from I_i . Its parsing action is determined as follows:

- If $[A \rightarrow \alpha \cdot a\beta, b]$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ to "shift j ".
Here, a must be a terminal.
- If $[A \rightarrow \alpha \cdot a, a]$ is in I_i , $A \neq S'$, then set $\text{ACTION}[i, a]$ to "reduce $A \rightarrow \alpha$ ".
More restrictive than SLR
- If $[S' \rightarrow S \cdot, \$]$ is in I_i , then set $\text{ACTION}[i, \$]$ to "accept"

If any conflicting actions result from the above rules, we say the grammar is not LR(1)

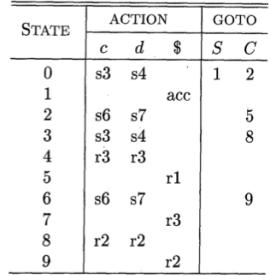
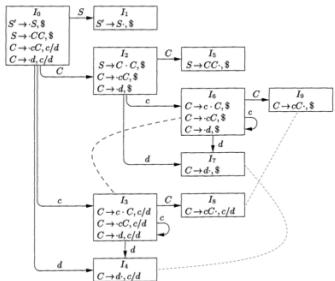
- The goto transitions for state i are constructed from all nonterminals A using the rule: If $\text{GOTO}(I_i, A) = I_j$, then $\text{GOTO}(i, A) = j$
- All entries not defined in steps (2) and (3) are made "error"
- The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow S, \$]$

需要多看的那个字符，才能 reduce.

初始状态为 $[S' \rightarrow S, \$]$

LR(1) Parsing Table Example

Grammar: $S' \rightarrow S$ $S \rightarrow CC$ $C \rightarrow cC \mid d$



Three pairs of states can be seen as being split from the corresponding LR(0) states:

(3, 6) (4, 7) (8, 9)

Lookahead LR (LALR) Method

- SLR(1) is not powerful enough to handle a large collection of grammars (recall the previous unambiguous grammar)
- LR(1) has a huge set of states in the parsing table (states are too fine-grained)
- LALR(1) is often used in practice
 - Keeps the lookahead symbols in the items
 - Its number of states is the same as that of SLR(1)
 - Can deal with most common syntactic constructs of modern programming languages

SLR(1) 不能解决有的文法

LR(1) 的状态集过大。

Merging States in LR(1) Parsing Tables

- State 4:
 - Reduce by $C \rightarrow d$ if the next input symbol is c or d
 - Error if $\$$
- State 7:
 - Reduce by $C \rightarrow d$ if the next input symbol is $\$$
 - Error if c or d



Can we merge states 4 and 7 so that the parser can reduce for all input symbols?

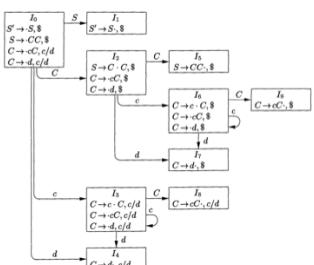
STATE	ACTION			GOTO	
	c	d	$\$$		
0	s3	s4		1	2
1			acc		
2	s6	s7		5	
3	s3	s4		8	
4	r3	r3			
5			r1		
6	s6	s7		9	
7			r3		
8	r2	r2			
9			r2		

- $I_4: C \rightarrow d \cdot, c/d$
- $I_7: C \rightarrow d \cdot, \$$

The Basic Idea of LALR

- Look for sets of LR(1) items with the same **core**
 - The core of an LR(1) item set is the set of the first components
 - The core of I_4 and I_7 is $\{C \rightarrow d \cdot\}$
 - The core of I_3 and I_6 is $\{C \rightarrow c \cdot C, C \rightarrow cC, C \rightarrow d\}$

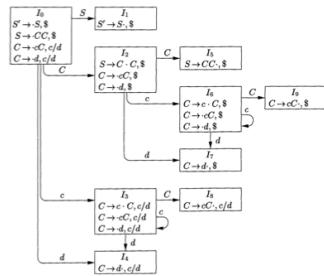
有相同核的 LR(1) 集合合并



- Look for sets of LR(1) items with the same **core**
 - The core of an LR(1) item set is the set of the first components
 - The core of I_4 and I_7 is $\{C \rightarrow d \cdot\}$
 - The core of I_3 and I_6 is $\{C \rightarrow c \cdot C, C \rightarrow \cdot cC, C \rightarrow \cdot d\}$
 - In general, a core is a set of LR(0) items

- We may merge the LR(1) item sets with common cores into one set of items

- Since the core of $\text{GOTO}(I, X)$ depends only on the core of I , the goto targets of merged sets also have the same core and hence can be merged



Consider I_3 and I_6 :

- The core $\{C \rightarrow c \cdot C, C \rightarrow \cdot cC, C \rightarrow \cdot d\}$ determines state transition targets
- Before merging, $\text{GOTO}(I_3, C) = I_9$, $\text{GOTO}(I_6, C) = I_8$
- After merging, I_3 and I_6 become I_{36} , I_8 and I_9 become I_{89} , and $\text{GOTO}(I_{36}, C) = I_{89}$

Conflicts Caused by State Merging

- Merging states in an LR(1) parsing table may cause conflicts
- **Merging does not cause shift/reduce conflicts**
 - Suppose after merging there is shift/reduce conflict on lookahead a
 - There is an item $[A \rightarrow \alpha \cdot, a]$ in a merged set calling for a reduction by $A \rightarrow \alpha$
 - There is another item $[B \rightarrow \beta \cdot ay, ?]$ in the set calling for a shift
 - Since the cores of the sets to be merged are the same, there must be a set containing both $[A \rightarrow \alpha \cdot, a]$ and $[B \rightarrow \beta \cdot ay, ?]$ before merging
 - Then before merging, there is already a shift/reduce conflict on a according to LR(1) parsing table construction algorithm. The grammar is not LR(1).
Contradiction!!!
- **Merging states may cause reduce/reduce conflicts**

合并会造成移入/归约冲突

Example of Conflicts

- An LR(1) grammar:
 - $S' \rightarrow S \quad S \rightarrow aAd \mid bBd \mid aBe \mid bAe \quad A \rightarrow c \quad B \rightarrow c$
- Language: $\{acd, bcd, ace, bce\}$
- One set of valid LR(1) items
 - $\{[A \rightarrow c \cdot, d], [B \rightarrow c \cdot, e]\}$
- Another set of valid LR(1) items
 - $\{[B \rightarrow c \cdot, d], [A \rightarrow c \cdot, e]\}$
- After merging, the new item set: $\{[A \rightarrow c \cdot, d/e], [B \rightarrow c \cdot, d/e]\}$
 - **Conflict:** reduce c to A or B when the next input symbol is d/e ?

Constructing LALR Parsing Table

- Construct $C = \{I_0, I_1, \dots, I_n\}$, the collection of sets of LR(1) items
- For each core present among a set of LR(1) items, find all sets having that core, and replace these sets by their union
- Let $C' = \{J_0, J_1, \dots, J_m\}$ be the resulting collection after merging.
 - The parsing actions for state i are constructed from J_i following the LR(1) parsing table construction algorithm.
 - If there is a conflict, this algorithm fails to produce a parser and the grammar is not LALR(1)

Basic idea: Merging states in LR(1) parsing table; If there is no reduce-reduce conflict, the grammar is LALR(1), otherwise not LALR(1).

- Construct the GOTO table as follows:

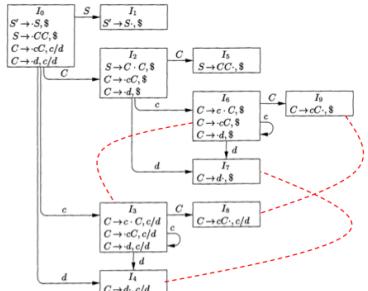
- If J is the union of one or more sets of LR(1) items, that is $J = I_1 \cup I_2 \cup \dots \cup I_k$, then the cores of $\text{GOTO}(I_1, X)$, $\text{GOTO}(I_2, X)$, ..., $\text{GOTO}(I_k, X)$ are the same, since I_1, I_2, \dots, I_k all have the same core.
- Let K be the union of all sets of items having the same core as $\text{GOTO}(I_1, X)$
- $\text{GOTO}(J, X) = K$

LALR Parsing Table Example

- Merging item sets

- I_{36} : $[C \rightarrow c \cdot C, c/d/\$], [C \rightarrow cC, c/d/\$], [C \rightarrow d, c/d/\$]$
- I_{47} : $[C \rightarrow d \cdot, c/d/\$]$
- I_{89} : $[C \rightarrow cC \cdot, c/d/\$]$

- $\text{GOTO}(I_{36}, C) = I_{89}$



STATE	ACTION			GOTO	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7		5	
3	s3	s4		8	
4	r3	r3			
5			r1		
6	s6	s7		9	
7	r2	r2	r3		
8	r2	r2	r3		
9			r2		

STATE	ACTION			GOTO	
	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47		5	
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		



Comparisons Among LR Parsers

- The languages (grammars) that can be handled
 - $\text{CLR} > \text{LALR} > \text{SLR}$
- # states in the parsing table
 - $\text{CLR} > \text{LALR} = \text{SLR}$
- Driver programs
 - $\text{SLR} = \text{CLR} = \text{LALR}$