

Compiler

(CS3104)

Syntax Analysis,
Parsing

Lex – example-1

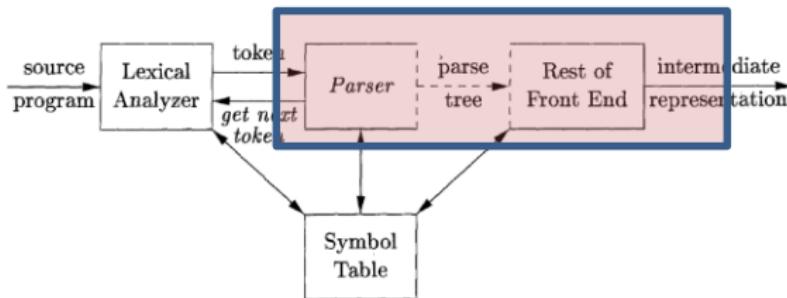
Input file – **input_first**

```
if + 78 else 0
```

Tokens: if, else, op (+,-), number, other

Parsing

- ▶ Every programming language **has precise grammar rules** that describe the **syntactic structure** of well-formed programs
 - ▶ In C, the **rules** states a **program consists of functions**, a **function consist of declarations and statements**, a **statement consists of expressions**, and so on.
- ▶ The **task of a parser** is to
 - (a) **Obtain strings of tokens** from the lexical analyzer and **verify** that the string follows the **rules of the source language**
 - (b) Parser **reports errors** and sometimes **recovers** from it



- **Type checking, semantic analysis and translation actions** can be **interlinked** with parsing
- Implemented as a **single module**.

Parsing

- ▶ Two major classes of parsing
 - ▶ top-down and bottom-up
- ▶ Input to the parser is **scanned from left to right**, one symbol at a time.

$\langle \text{id}, 1 \rangle \xrightarrow{=} \langle \text{id}, 2 \rangle \xrightarrow[+]{\quad} \langle \text{id}, 3 \rangle \xrightarrow{*} \langle 60 \rangle$

- ▶ The **syntax of programming language** constructs can be specified by **context-free grammars**
- ▶ Grammars systematically describe the **syntax** of programming language constructs like **expressions and statements**.

$stmt \rightarrow \text{if} (\ expr \) \ stmt \ \text{else} \ stmt$

- ▶ Quick recall

Context free grammar

- ▶ A CFG is denoted as $G = (N, T, P, S)$

N : Finite set of non-terminals -- **syntactic variables** (stmt, expr)

T : Finite set of terminals ---- **Tokens**, basic symbols from which strings and programs are formed

S : The start symbol -- set of strings it generates is the **language** generated by the grammar

P : Finite set of productions -- specify the manner in which the **terminals and nonterminals can be combined** to form strings

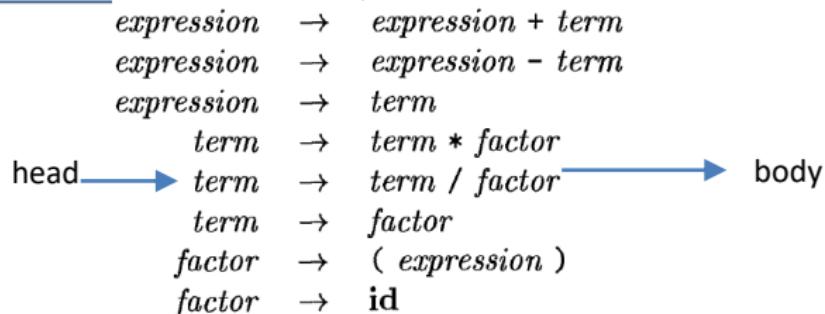
In this grammar, the terminal symbols are

id + - * / ()

Start symbol:

expression

Productions



Task of a parser

Output of the parser is some **representation of the parse tree** for the **stream of tokens as input**, that comes from the lexical analyzer.

- **Top-down** parser works for **LL grammar**
- **Bottom-up** parser works for **LR grammars**
- Only subclasses of grammars
 - But expressive enough to describe **most of the syntactic constructs** of modern programming languages.

Concentrate on parsing **expressions**

- Constructs that begin with keywords like **while** or **int** are relatively easy to parse
 - because the **keyword guides the parsing decisions**
- We therefore **concentrate on expressions**, which present **more of challenge**, because of the **associativity and precedence** of operators

Derivations

The **construction of a parse tree** can be conceptualized as **derivations**

Derivation: Beginning with the **start symbol**, each rewriting step **replaces a nonterminal** by the body of one of its **productions**.

$A \rightarrow \gamma$ is a production

$\alpha A \beta \Rightarrow \alpha \gamma \beta$

If $S \xrightarrow{*} \alpha$, where S is the start symbol of a grammar G , we say that α is a *sentential form* of G .

A **sentence** of G is a sentential form with **no nonterminals**.

The language $L(G)$ generated by a grammar G is its **set of sentences**.

Derivations

The **construction of a parse tree** can be conceptualized as **derivations**

Beginning with the start symbol, each rewriting step replaces a nonterminal by the body of one of its productions.

$$\alpha A \beta \Rightarrow \alpha \bar{\gamma} \beta. \quad A \rightarrow \gamma \text{ is a production}$$

Consider a grammar G

$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \text{id}$$

Derivation

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\text{id} + E) \Rightarrow -(\text{id} + \text{id})$$

1. Derivation of **-(id+id)** from start symbol E
2. **-(id+id)** is a **sentence** of G
3. At **each step** in a derivation, there are **two choices** to be made.
 - **Which nonterminal** to replace? : **leftmost derivations**
 - Accordingly we must **choose** a production

Derivations-- Rightmost derivations

Consider a grammar G

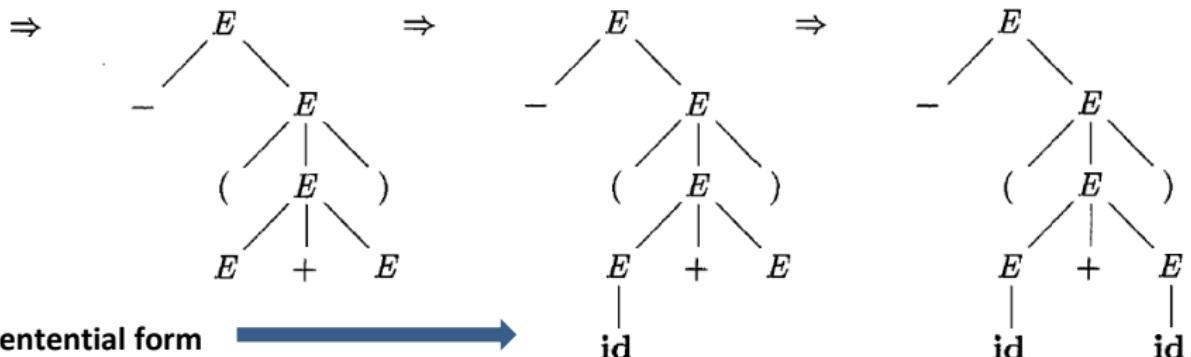
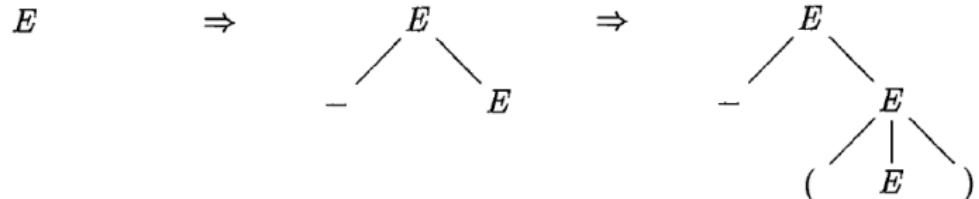
$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \text{id}$$

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(E + \text{id}) \Rightarrow -(\text{id} + \text{id})$$

1. Derivation of $-(\text{id}+\text{id})$ from E
2. $-(\text{id}+\text{id})$ is a sentence of G
3. At each step in a derivation, there are two choices to be made.
 - Which nonterminal to replace?
 - Accordingly we must pick a production → **Rightmost derivations**,

Parse trees

- ▶ A parse tree is a **graphical representation of a derivation** that exhibits
 - ▶ the **order** in which **productions are applied** to replace non-terminals
- ▶ The **internal node** is a **non-terminal A** in the head of the production
 - ▶ The **children of the node** are labelled, from left to right, by the symbols in the **body of the production** by which A was replaced during the derivation
- ▶ **Same parse tree** for leftmost and rightmost derivations

$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \text{id}$$


Sentential form
(leaves of a
parse tree)

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\text{id}+E) \Rightarrow -(\text{id}+\text{id})$$

parse tree for - (id + id)

Ambiguity

- ▶ A grammar that **produces more than one parse tree** for some **sentence** is said to be ***ambiguous***
- ▶ *An ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.*

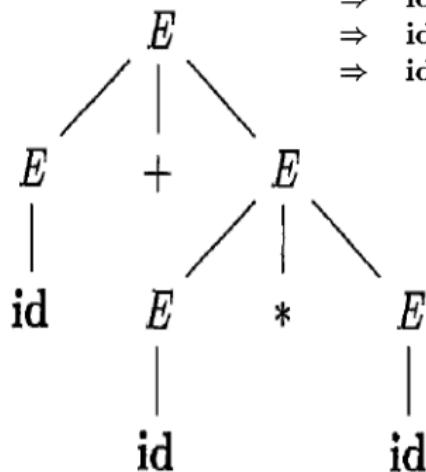
$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$$

$$\begin{array}{ll} E \Rightarrow E + E & E \Rightarrow E * E \\ \Rightarrow \text{id} + E & \Rightarrow E + E * E \\ \Rightarrow \text{id} + E * E & \Rightarrow \text{id} + E * E \\ \Rightarrow \text{id} + \text{id} * E & \Rightarrow \text{id} + \text{id} * E \\ \Rightarrow \text{id} + \text{id} * \text{id} & \Rightarrow \text{id} + \text{id} * \text{id} \end{array}$$

Two distinct leftmost derivations for the sentence **id + id * id**

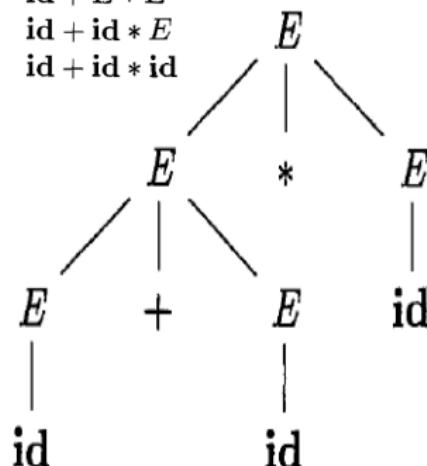
Ambiguity

$$\begin{array}{lcl} E & \Rightarrow & E + E \\ & \Rightarrow & id + E \\ & \Rightarrow & id + E * E \\ & \Rightarrow & id + id * E \\ & \Rightarrow & id + id * id \end{array}$$



(a)

$$\begin{array}{lcl} E & \Rightarrow & E * E \\ & \Rightarrow & E + E * E \\ & \Rightarrow & id + E * E \\ & \Rightarrow & id + id * E \\ & \Rightarrow & id + id * id \end{array}$$



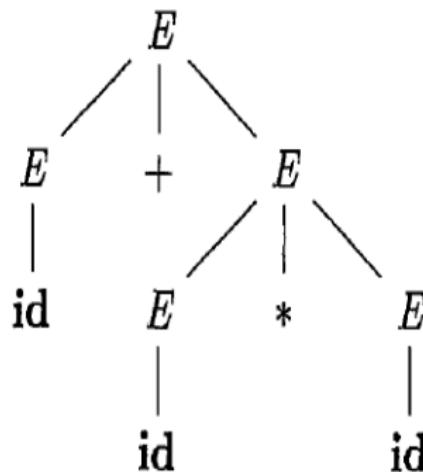
(b)

Two parse trees for $id + id * id$

Ambiguity

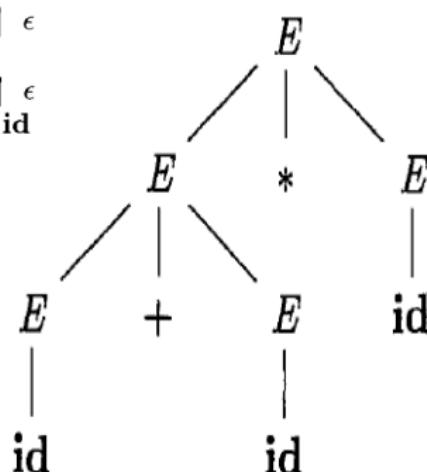
$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \text{id} \end{array}$$

Unambiguous grammar



(a)

Two parse trees for `id+id*id`

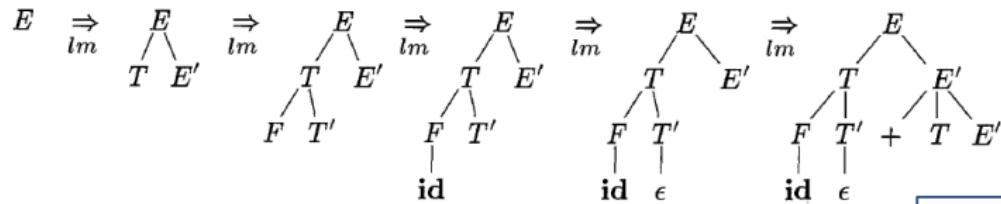


(b)

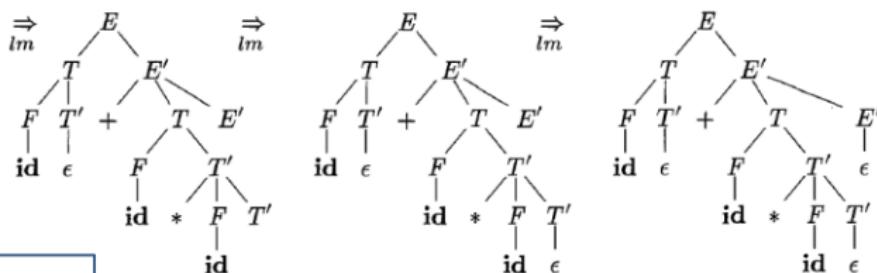
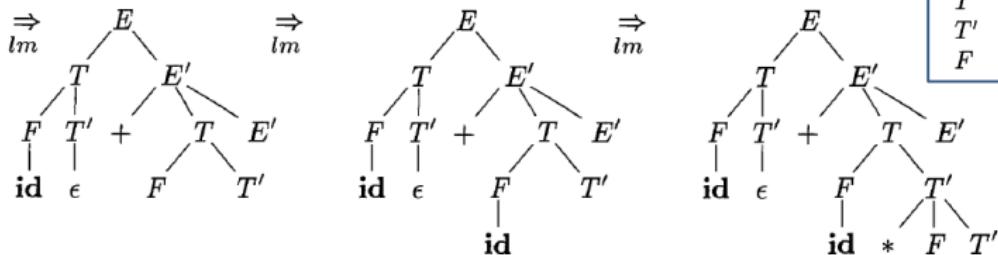
Top-Down Parsing

- Top-down parsing can be viewed as the problem of
- **Constructing a parse tree** for the input string,
 - **starting from the root** and creating the nodes of the parse tree in **preorder**
- Top-down parsing can be viewed as finding a **leftmost derivation** for an input string

$$\begin{array}{lll} E & \rightarrow & T \ E' \\ E' & \rightarrow & + \ T \ E' \mid \epsilon \\ T & \rightarrow & F \ T' \\ T' & \rightarrow & * \ F \ T' \mid \epsilon \\ F & \rightarrow & (\ E \) \mid \text{id} \end{array} \quad \xrightarrow{\hspace{10em}} \text{id} + \text{id} * \text{id}$$

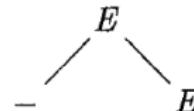
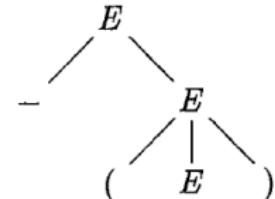


$$\begin{array}{lcl}
 E & \rightarrow & T \ E' \\
 E' & \rightarrow & + \ T \ E' \mid \epsilon \\
 T & \rightarrow & F \ T' \\
 T' & \rightarrow & * \ F \ T' \mid \epsilon \\
 F & \rightarrow & (\ E \) \mid \text{id}
 \end{array}$$

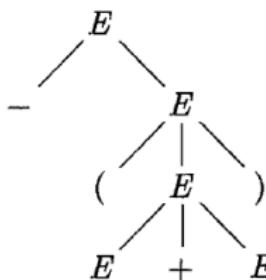
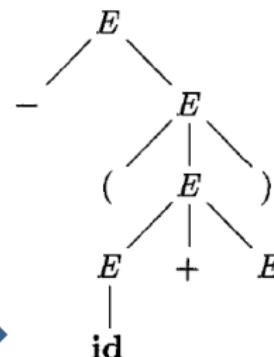
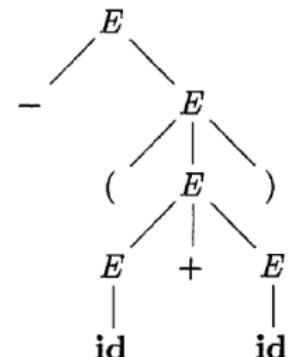


id + id * id

Top-Down Parsing

 E \Rightarrow  \Rightarrow 

parse tree for $- (\text{id} + \text{id})$

 \Rightarrow  \Rightarrow  \Rightarrow 

Derivation

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\text{id}+E) \Rightarrow -(\text{id}+\text{id})$$

parse tree for $- (+ \text{id}) ???$

Top-Down Parsing

A grammar is *left recursive* if it has a nonterminal A such that there is a derivation $A \stackrel{+}{\Rightarrow} A\alpha$ for some string α . Top-down parsing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \text{id} \end{array} \quad \text{Left recursive}$$

$$\begin{array}{lcl} E & \rightarrow & T E' \\ E' & \rightarrow & + T E' \mid \epsilon \\ T & \rightarrow & F T' \\ T' & \rightarrow & * F T' \mid \epsilon \\ F & \rightarrow & (E) \mid \text{id} \end{array} \quad \text{Non-Left recursive}$$

Eliminating left recursion.

production of the form $A \rightarrow A\alpha | \beta$



$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \quad | \quad \epsilon$$

Generalization

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

Immediate left recursion

Eliminating left recursion.

$$\begin{aligned} S &\rightarrow A \ a \mid b \\ A &\rightarrow A \ c \mid S \ d \mid \epsilon \end{aligned}$$

$$S \Rightarrow Aa \Rightarrow Sda$$

Top-Down Parsing

Eliminating left recursion.

INPUT: Grammar G with no cycles or ϵ -productions.

OUTPUT: An equivalent grammar with no left recursion.

- 1) arrange the nonterminals in some order A_1, A_2, \dots, A_n .
- 2) **for** (each i from 1 to n) {
 - 3) **for** (each j from 1 to $i - 1$) {
 - 4) 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
 - 5) }
 - 6) eliminate the immediate left recursion among the A_i -productions
 - 7) }

Eliminating left recursion.

$$\begin{aligned} S &\rightarrow A \ a \mid b \\ A &\rightarrow A \ c \mid S \ d \mid \epsilon \end{aligned}$$

Unfolding all the left recursions

$$A \rightarrow A \ c \mid A \ a \ d \mid b \ d \mid \epsilon$$

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

$$\begin{aligned} S &\rightarrow A \ a \mid b \\ A &\rightarrow b \ d \ A' \mid A' \\ A' &\rightarrow c \ A' \mid a \ d \ A' \mid \epsilon \end{aligned}$$

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \epsilon \end{aligned}$$

Top-Down Parsing

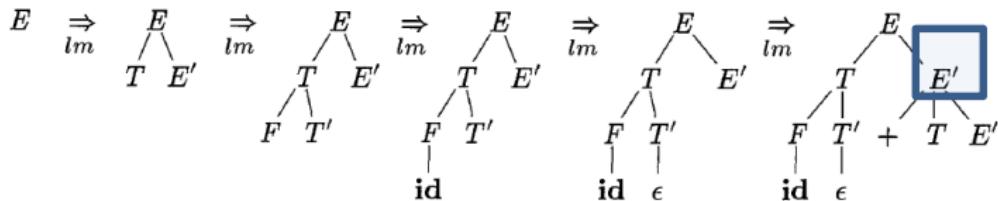
Challenges:

At **each step** of a top-down parse, the key problem is that of **determining the production to be applied** for a nonterminal, say A.

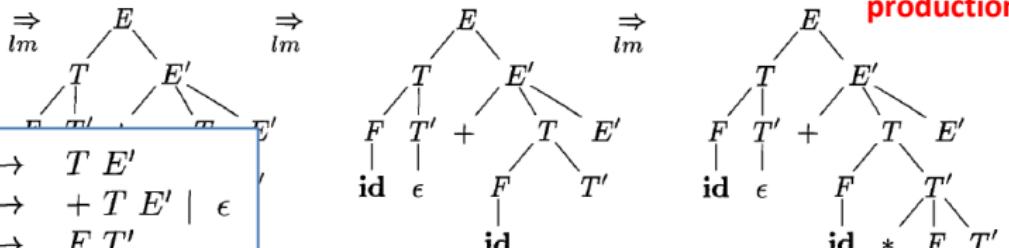
(a) **Recursive descent parsing:** May require **backtracking** to find the **correct A-production** to be applied

(b) **Predictive parsing:** No backtracking!

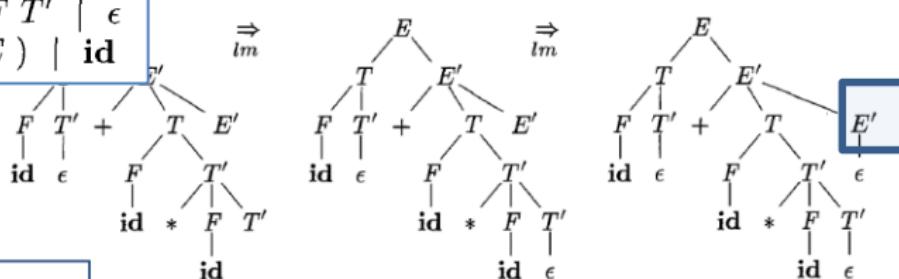
looking ahead at the input a fixed number of symbols (next symbols) – LL(k),
LL(1) grammars



Choose the correct production



$$\begin{array}{lcl} E & \rightarrow & T \ E' \\ E' & \rightarrow & + \ T \ E' \mid \epsilon \\ T & \rightarrow & F \ T' \\ T' & \rightarrow & * \ F \ T' \mid \epsilon \\ F & \rightarrow & (\ E \) \mid \text{id} \end{array}$$



id + id * id

Recursive-Descent Parsing

Nondeterministic

```
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 */;

```
}
```

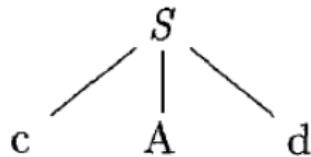


Try other productions!

- (a) A recursive-descent parsing **consists of a set of procedures**, one for each **nonterminal**.
- (b) Execution begins with the **procedure for the start symbol S**,
- (c) **Halts and announces success** if $S()$ returns and its procedure body scans the **entire input string**.
- (d) **Backtracking**: may require **repeated scans over the input**

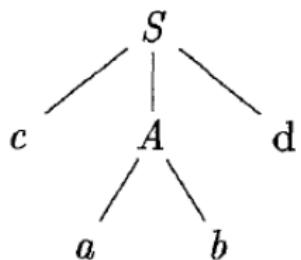
$$\begin{array}{l} S \rightarrow c A d \\ A \rightarrow a b \mid a \end{array}$$

input string $w = cad,$



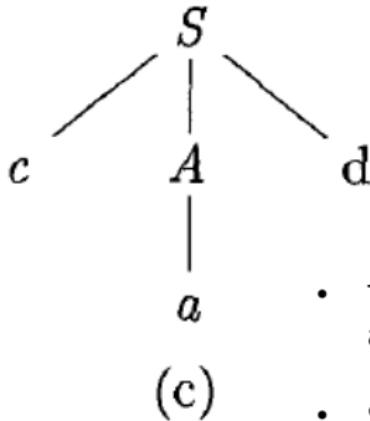
The leftmost leaf, **labeled c** , matches the first symbol of input w (i.e. c), so we advance the input pointer to a

input string $w = cad,$



Now, we expand A using the first alternative $A \rightarrow a b$

- We have a match for the **second input symbol, a** ,
 - So we advance the **input pointer to d** , the third input symbol
 - Compare d against the next leaf, labeled **b**
- Failure !! Backtrack!**



input string $w = cad,$

we must reset the input pointer to position **a**

- The leaf **a** matches the second input symbol of w (i.e. **a**) and the leaf **d** matches the third input symbol **d**
- Since **S()** returns and we have **scanned w and produced a parse tree for w**,
- We halt and **announce successful completion of parsing**

Top-Down Parsing

Challenges:

At **each step** of a top-down parse, the key problem is that of **determining the production to be applied** for a nonterminal, say A.

(a) Recursive descent parsing: May require backtracking to find the correct A-production to be applied

(b) **Predictive parsing:** No backtracking!

looking ahead at the input a fixed number of symbols (**next symbols**) –
LL(k), LL(1) grammars

Basic concept of Predictive parsing

Input string $w=abcd$

One sentential form

$S \Rightarrow aXY\dots$

Another sentential form

$S \Rightarrow aXb$

We know that **b Follows X** in any sentential form



Grammar productions

1. $X \rightarrow bA\dots$



First symbol

2. $X \rightarrow cP \dots$

Grammar productions

1. $X \rightarrow \epsilon$

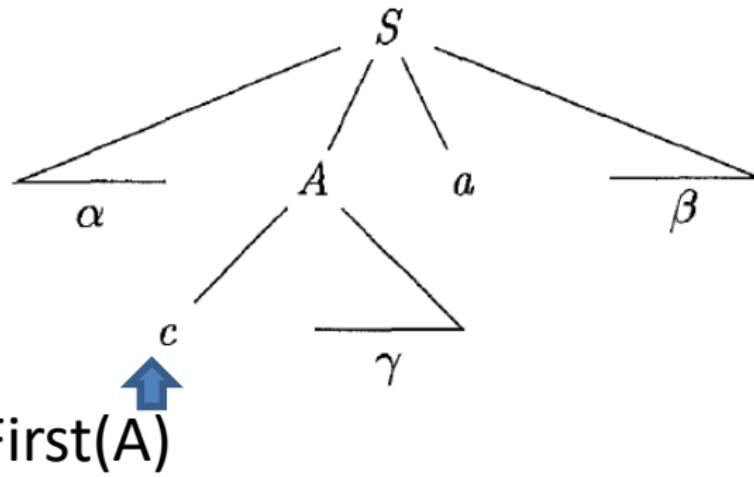
2. $X \rightarrow \dots$

4.4.2 FIRST and FOLLOW

The construction of both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW, associated with a grammar G . During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol. During panic-mode error recovery, sets of tokens produced by FOLLOW can be used as synchronizing tokens.

Define $\text{FIRST}(\alpha)$, where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α . If $\alpha \xrightarrow{*} \epsilon$, then ϵ is also in $\text{FIRST}(\alpha)$. For example, in Fig. 4.15, $A \xrightarrow{*} c\gamma$, so c is in $\text{FIRST}(A)$.

For a preview of how FIRST can be used during predictive parsing, consider two A -productions $A \rightarrow \alpha \mid \beta$, where $\text{FIRST}(\alpha)$ and $\text{FIRST}(\beta)$ are disjoint sets. 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 $\text{FIRST}(\alpha)$ and $\text{FIRST}(\beta)$, not both. For instance, if a is in $\text{FIRST}(\beta)$ choose the production $A \rightarrow \beta$. This idea will



How to compute First(X)

To compute $\text{FIRST}(X)$ for all grammar symbols X , apply the following rules until no more terminals or ϵ can be added to any FIRST set.

1. If X is a terminal, then $\text{FIRST}(X) = \{X\}$.
2. If X is a nonterminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in $\text{FIRST}(X)$ if for some i , a is in $\text{FIRST}(Y_i)$, and ϵ is in all of $\text{FIRST}(Y_1), \dots, \text{FIRST}(Y_{i-1})$; that is, $Y_1 \cdots Y_{i-1} \xrightarrow{*} \epsilon$. If ϵ is in $\text{FIRST}(Y_j)$ for all $j = 1, 2, \dots, k$, then add ϵ to $\text{FIRST}(X)$. For example, everything in $\text{FIRST}(Y_1)$ is surely in $\text{FIRST}(X)$. If Y_1 does not derive ϵ , then we add nothing more to $\text{FIRST}(X)$, but if $Y_1 \xrightarrow{*} \epsilon$, then we add $\text{FIRST}(Y_2)$, and so on.
3. If $X \rightarrow \epsilon$ is a production, then add ϵ to $\text{FIRST}(X)$.

$$\begin{array}{lcl}
 E & \rightarrow & T \ E' \\
 E' & \rightarrow & + \ T \ E' \mid \epsilon \\
 T & \rightarrow & F \ T' \\
 T' & \rightarrow & * \ F \ T' \mid \epsilon \\
 F & \rightarrow & (\ E \) \mid \text{id}
 \end{array}$$

1. $\text{FIRST}(F) = \text{FIRST}(T) = \text{FIRST}(E) = \{(, \text{id}\}$. To see why, note that the two productions for F have bodies that start with these two terminal symbols, **id** and the left parenthesis. T has only one production, and its body starts with F . Since F does not derive ϵ , $\text{FIRST}(T)$ must be the same as $\text{FIRST}(F)$. The same argument covers $\text{FIRST}(E)$.
2. $\text{FIRST}(E') = \{+, \epsilon\}$. The reason is that one of the two productions for E' has a body that begins with terminal $+$, and the other's body is ϵ . Whenever a nonterminal derives ϵ , we place ϵ in FIRST for that nonterminal.
3. $\text{FIRST}(T') = \{*, \epsilon\}$. The reasoning is analogous to that for $\text{FIRST}(E')$.

Basic concept of Predictive parsing

Input string $w=abcd$

One sentential form
 $S \Rightarrow aXY\dots$

Another sentential form
 $S \Rightarrow aXb$

We know that **b Follows X** in any sentential form



Grammar productions

1. $X \rightarrow bA\dots$



First symbol

2. $X \rightarrow cP \dots$

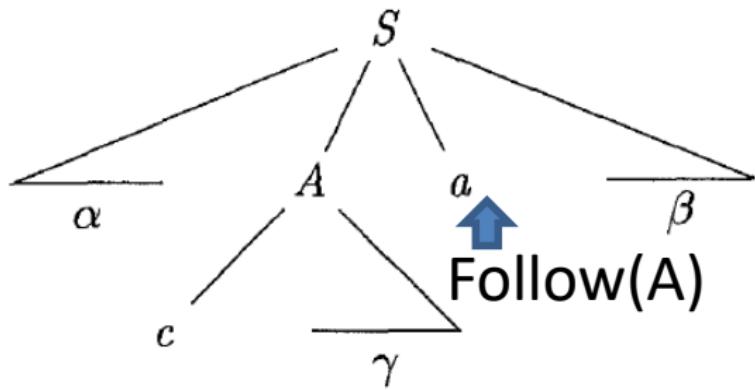
Grammar productions

1. $X \rightarrow \epsilon$

2. $X \rightarrow \dots$

FIRST and FOLLOW

Define $\text{FOLLOW}(A)$, 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, the set of terminals a such that there exists a derivation of the form $S \xrightarrow{*} \alpha A a \beta$, for some α and β , as in Fig. 4.15. Note that there may have been symbols between A and a , at some time during the derivation, but if so, they derived ϵ and disappeared. In addition, if A can be the rightmost symbol in some sentential form, then $\$$ is in $\text{FOLLOW}(A)$; recall that $\$$ is a special “endmarker” symbol that is assumed not to be a symbol of any grammar.



How to compute Follow(A)

$S \rightarrow xAy$

y in $\text{Follow}(A)$

To compute $\text{FOLLOW}(A)$ for all nonterminals A , apply the following rules until nothing can be added to any FOLLOW set.

1. Place $\$$ in $\text{FOLLOW}(S)$, where S is the start symbol, and $\$$ is the input right endmarker.
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in $\text{FIRST}(\beta)$ except ϵ is in $\text{FOLLOW}(B)$.
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$, where $\text{FIRST}(\beta)$ contains ϵ , then everything in $\text{FOLLOW}(A)$ is in $\text{FOLLOW}(B)$.

$S \rightarrow xAy$

$\text{Follow}(A) = y$

$\rightarrow x\alpha B y$

$\text{Follow}(B) = \text{Follow}(A)$

$E \rightarrow T E'$ ↙
 $E' \rightarrow + T E' \mid \epsilon$
 $T \rightarrow F T'$
 $T' \rightarrow * F T' \mid \epsilon$
 $F \rightarrow (E) \mid \text{id}$

$\text{FOLLOW}(E) = \text{FOLLOW}(E') = \{\}, \$\}.$ Since E is the start symbol, $\text{FOLLOW}(E)$ must contain $\$$. The production body (E) explains why the right parenthesis is in $\text{FOLLOW}(E)$. For E' , note that this nonterminal appears only at the ends of bodies of E -productions. Thus, $\text{FOLLOW}(E')$ must be the same as $\text{FOLLOW}(E)$.

$$\begin{array}{lcl}
 E & \rightarrow & T \ E' \\
 E' & \rightarrow & + \ T \ E' \mid \epsilon \quad \leftarrow \\
 T & \rightarrow & F \ T' \\
 T' & \rightarrow & * \ F \ T' \mid \epsilon \\
 F & \rightarrow & (\ E \) \mid \text{id}
 \end{array}$$

$$\text{FIRST}(E') = \{+, \epsilon\}$$

$\text{FOLLOW}(T) = \text{FOLLOW}(T') = \{+,), \$\}$. Notice that T appears in bodies only followed by E' . Thus, everything except ϵ that is in $\text{FIRST}(E')$ must be in $\text{FOLLOW}(T)$; that explains the symbol $+$. However, since $\text{FIRST}(E')$ contains ϵ (i.e., $E' \xrightarrow{*} \epsilon$), and E' is the entire string following T in the bodies of the E -productions, everything in $\text{FOLLOW}(E)$ must also be in $\text{FOLLOW}(T)$. That explains the symbols $\$$ and the right parenthesis. As for T' , since it appears only at the ends of the T -productions, it must be that $\text{FOLLOW}(T') = \text{FOLLOW}(T)$.

$$\begin{array}{lcl}
 E & \rightarrow & T \ E' \\
 E' & \rightarrow & + \ T \ E' \mid \epsilon \\
 T & \rightarrow & F \ T' \quad \leftarrow \\
 T' & \rightarrow & * \ F \ T' \mid \epsilon \\
 F & \rightarrow & (\ E \) \mid \text{id}
 \end{array}$$

$\text{FOLLOW}(F) = \{+, *,), \$\}$. The reasoning is analogous to that for T in point (5).

Follow(F)=Follow(T)

Predictive parsing

Challenges:

At **each step** of a top-down parse, the key problem is that of **determining the production to be applied** for a nonterminal, say A.

(a) Recursive descent parsing: May require backtracking to find the correct A-production to be applied

(b) **Predictive parsing:** No backtracking!

looking ahead at the input a fixed number of symbols (**next symbols**) –
LL(k), LL(1) grammars

Predictive parsing

Parsing table M

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	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

LL(1) grammar => avoid confusion!!

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

First(α) and First(β) Disjoint sets

1. For no terminal a do both α and β derive strings beginning with a .
2. At most one of α and β can derive the empty string.
3. If $\beta \xrightarrow{*} \epsilon$, then α does not derive any string beginning with a terminal in $\text{FOLLOW}(A)$. Likewise, if $\alpha \xrightarrow{*} \epsilon$, then β does not derive any string beginning with a terminal in $\text{FOLLOW}(A)$.

ϵ is in $\text{FIRST}(\alpha)$.

then $\text{FIRST}(\beta)$ and $\text{FOLLOW}(A)$ are disjoint sets.

Basic concept of Predictive parsing

Input string $w=abcd$

One sentential form

$S \Rightarrow aXY\ldots$

Another sentential form

$S \Rightarrow aXb$

We know that **b** Follows **X** in any sentential form **Follow(X)=b**

Grammar productions

1. $X \rightarrow bA\ldots$



First symbol

2. $X \rightarrow bY\ldots$

Grammar productions

1. $X \rightarrow \epsilon$

2. $X \rightarrow \ldots$

3. $X \rightarrow bY\ldots$

Left Factoring

$stmt \rightarrow \text{if } expr \text{ then } stmt \text{ else } stmt$
| $\text{if } expr \text{ then } stmt$

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2$

Left factoring a grammar.

Left Factoring

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \beta_2$$

$$S \rightarrow i \ E \ t \ S \mid i \ E \ t \ S \ e \ S \mid a$$

$$E \rightarrow b$$

$$S \rightarrow i \ E \ t \ S \ S' \mid a$$

$$S' \rightarrow e \ S \mid \epsilon$$

$$E \rightarrow b$$

Parsing table M

INPUT: Grammar G .

OUTPUT: Parsing table M .

METHOD: For each production $A \rightarrow \alpha$ of the grammar, do the following:

1. For each terminal a in $\text{FIRST}(A)$, add $A \rightarrow \alpha$ to $M[A, a]$.



Obvious



Input string $w=bacd$

One sentential form

$S \Rightarrow bAY\dots$

Grammar productions

1. $A \rightarrow aX\dots$



First symbol

2. $A \rightarrow \dots$

INPUT: Grammar G .

OUTPUT: Parsing table M .

METHOD: For each production $A \rightarrow \alpha$ of the grammar, do the following:

1. For each terminal a in $\text{FIRST}(A)$, add $A \rightarrow \alpha$ to $M[A, a]$.
2. If ϵ is in $\text{FIRST}(\alpha)$ then for each terminal b in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, b]$. If ϵ is in $\text{FIRST}(\alpha)$ and $\$$ is in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, \$]$ as well.

Input string $w=abcd$

One sentential form

$S \Rightarrow aAb$

Grammar productions

1. $A \rightarrow \alpha \Rightarrow \epsilon$
2. $A \rightarrow \dots$

We know that **b Follows A** in any sentential form **Follow(A)=b**

INPUT: Grammar G .

OUTPUT: Parsing table M .

METHOD: For each production $A \rightarrow \alpha$ of the grammar, do the following:

1. For each terminal a in $\text{FIRST}(A)$, add $A \rightarrow \alpha$ to $M[A, a]$.
2. If ϵ is in $\text{FIRST}(\alpha)$, then for each terminal b in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, b]$. If ϵ is in $\text{FIRST}(\alpha)$ and $\$$ is in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, \$]$ as well.

If, after performing the above, there is no production at all in $M[A, a]$, then set $M[A, a]$ to **error** (which we normally represent by an empty entry in the table). \square

→ production $E \rightarrow TE'$

$$\text{FIRST}(TE') = \text{FIRST}(T) = \{(), \text{id}\}$$

→ Production $E' \rightarrow +TE'$

$$\text{FIRST}(+TE') = \{+\}$$

→ $E' \rightarrow \epsilon$

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

E	\rightarrow	$T E'$
E'	\rightarrow	$+ T E' \mid \epsilon$
T	\rightarrow	$F T'$
T'	\rightarrow	$* F T' \mid \epsilon$
F	\rightarrow	$(E) \mid \text{id}$

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	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

$$\rightarrow T \rightarrow FT'$$

$\text{First}(FT') = \{ , \text{id} \}$

$$\rightarrow T' \rightarrow *FT'$$

$\text{First}(*FT') = \{ * \}$

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

$\text{Follow}(T') = \{ +,), \$ \}$

E	\rightarrow	$T E'$
E'	\rightarrow	$+ T E' \mid \epsilon$
T	\rightarrow	$F T'$
T'	\rightarrow	$* F T' \mid \epsilon$
F	\rightarrow	$(E) \mid \text{id}$

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	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

$$\rightarrow F \rightarrow (E) \mid \text{id}$$

$\text{First}((E)) = \{ \}$

$\text{First}(\text{id}) = \{ \text{id} \}$

Example of Non-LL(1) grammar

- For every LL(1) grammar, each parsing-table entry uniquely identifies a production or signals an error.
- left-recursive or ambiguous grammars are not LL(1)

$$\begin{array}{l} S \rightarrow iEtSS' \mid a \\ S' \rightarrow eS \mid \epsilon \\ E \rightarrow b \end{array}$$

Input string
i b t i b t a e a

if b
then
if b
then
a
else
a

Example of Non-LL(1) grammar

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

Predictive Parsing

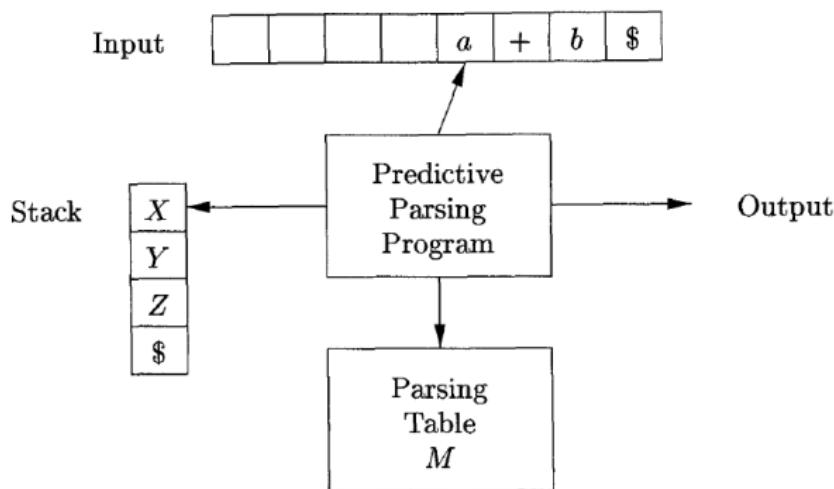
- **Non-recursive** version
 - maintaining a **stack explicitly**, rather than implicitly via recursive calls

INPUT: A string w and a parsing table M for grammar G .

OUTPUT: If w is in $L(G)$, a leftmost derivation of w ; otherwise, an error indication.

Initial configuration

STACK	INPUT
$E\$ \quad id + id * id\$$	



Recursive-Descent Parsing

Nondeterministic

```
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 */;  
    }  
}  
    Try other productions!
```

- (a) A recursive-descent parsing **consists of a set of procedures**, one for each **nonterminal**.
- (b) Execution begins with the **procedure for the start symbol S**,
- (c) **Halts and announces success** if $S()$ returns and its procedure body scans the **entire input string**.
- (d) **Backtracking**: may require **repeated scans over the input**

Predictive Parsing

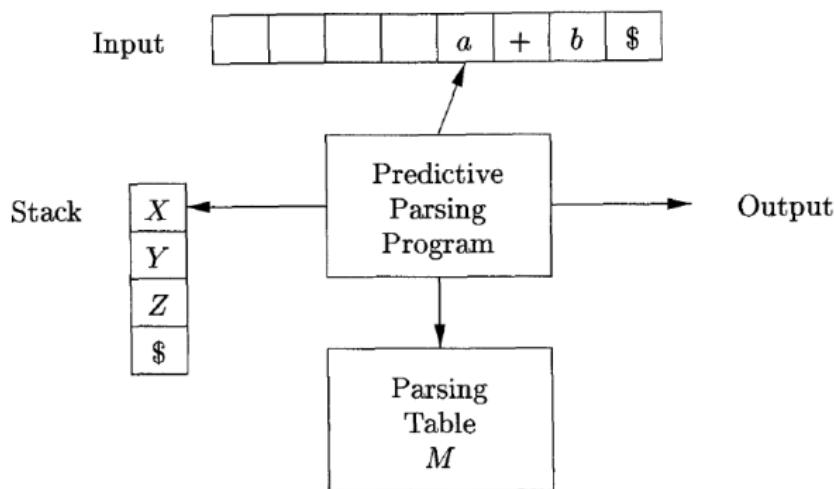
- **Non-recursive** version
 - maintaining a **stack explicitly**, rather than implicitly via recursive calls

INPUT: A string w and a parsing table M for grammar G .

OUTPUT: If w is in $L(G)$, a leftmost derivation of w ; otherwise, an error indication.

Initial configuration

STACK	INPUT
$E\$ \quad id + id * id\$$	



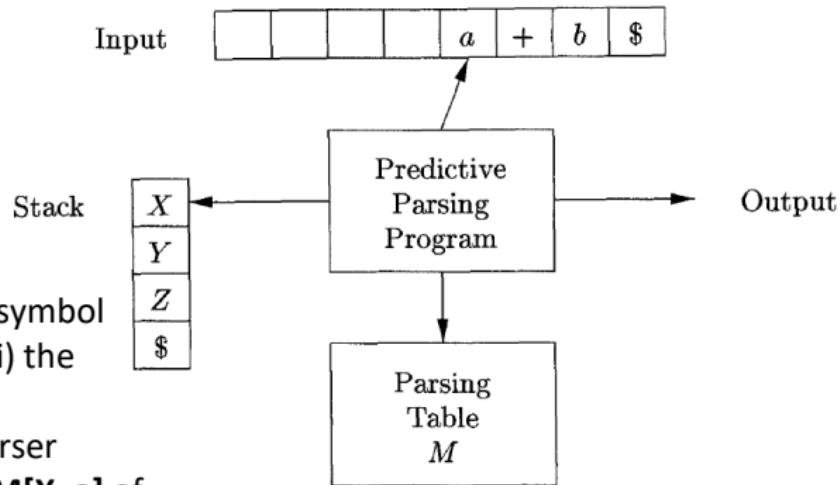
Predictive Parsing

INPUT: A string w and a parsing table M for grammar G .

OUTPUT: If w is in $L(G)$, a leftmost derivation of w ; otherwise, an error indication.

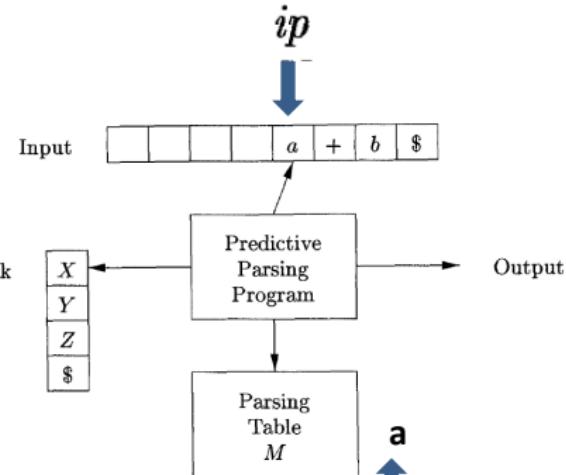
Initial configuration

STACK	INPUT
$E\$$	$id + id * id \$$



- The parser considers (i) the symbol **on top of the stack X**, and (ii) the current **input symbol a**.
- If **X is a nonterminal**, the parser chooses an X -production from $M[X, a]$ of the parsing table.
- Otherwise, it checks for a **match** between the **terminal X** and current **input symbol a**.

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	$F \rightarrow id$				$F \rightarrow (E)$	



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	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

set ip to the first symbol of w ;
 set X to the top stack symbol;
while ($X \neq \$$) { /* stack is not empty */
 if (X is a) pop the stack and advance ip ;
 else if (X is a terminal) *error()*;
 else if ($M[X, a]$ is an error entry) *error()*;
 else if ($M[X, a] = X \rightarrow Y_1 Y_2 \dots Y_k$) {
 output the production $X \rightarrow Y_1 Y_2 \dots Y_k$;
 pop the stack;
 push Y_k, Y_{k-1}, \dots, Y_1 onto the stack, with Y_1 on top;
 }
 }

set X to the top stack symbol;

}

Y_1

→
id + id * id

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	$F \rightarrow id$			$F \rightarrow (E)$		

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 $E' \rightarrow + TE'$
$id +$	$TE' \$$	$id * id \$$	match $+$
$id +$	$FT'E' \$$	$id * id \$$	output $T \rightarrow FT'$
$id +$	$id T'E' \$$	$id * id \$$	output $F \rightarrow id$
$id + id$	$T'E' \$$	$* id \$$	match id
$id + id$	$* FT'E' \$$	$* id \$$	output $T' \rightarrow * FT'$
$id + id *$	$FT'E' \$$	$id \$$	match $*$
$id + id *$	$id T'E' \$$	$id \$$	output $F \rightarrow id$
$id + id * id$	$T'E' \$$	$\$$	match id
$id + id * id$	$E' \$$	$\$$	output $T' \rightarrow \epsilon$
$id + id * id$	$\$$	$\$$	output $E' \rightarrow \epsilon$



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

Leftmost derivation

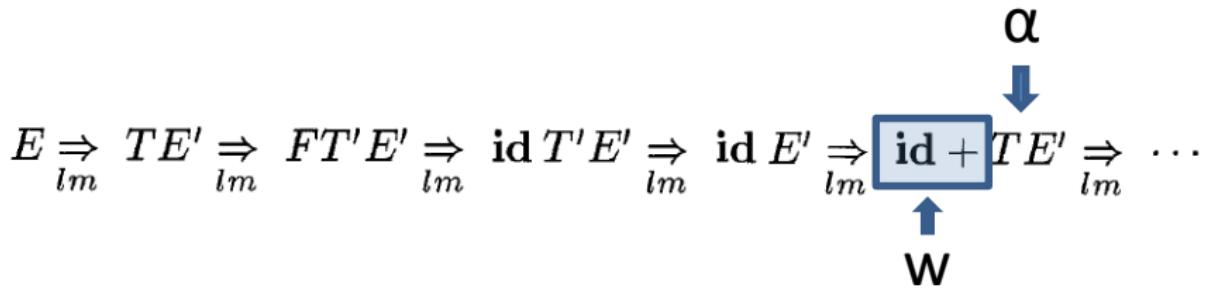
$$E \xrightarrow{lm} TE' \xrightarrow{lm} FT'E' \xrightarrow{lm} \text{id } T'E' \xrightarrow{lm} \text{id } E' \xrightarrow{lm} \text{id} + TE' \xrightarrow{lm} \dots$$

Predictive Parsing

The **stack** contains a sequence of **grammar symbols**

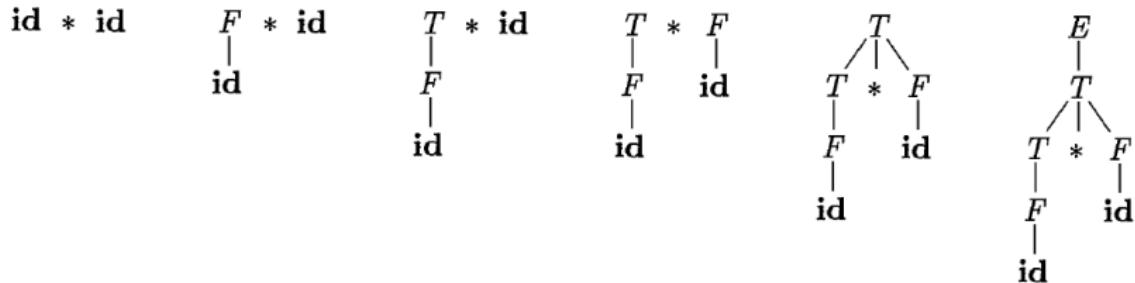
If w is the input that has been matched so far, then the stack holds a sequence of grammar symbols α such that

$$S \xrightarrow[lm]{*} w\alpha$$



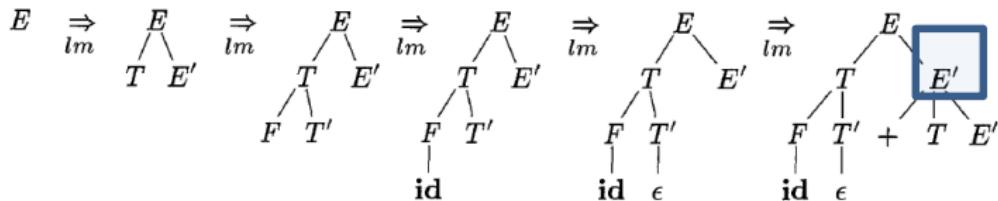
Bottom Up Parsing

- A bottom-up parse corresponds to the **construction of a parse tree** for an **input string**
 - Beginning at the leaves** (the bottom) and working up towards the **root** (the top)

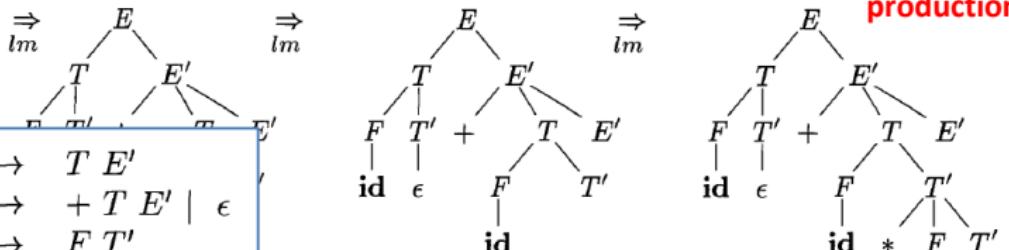


Input . **id * id**

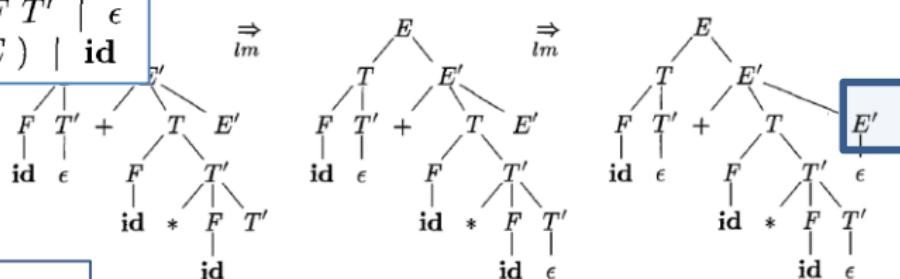
$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \text{id} \end{array}$$



Choose the correct production



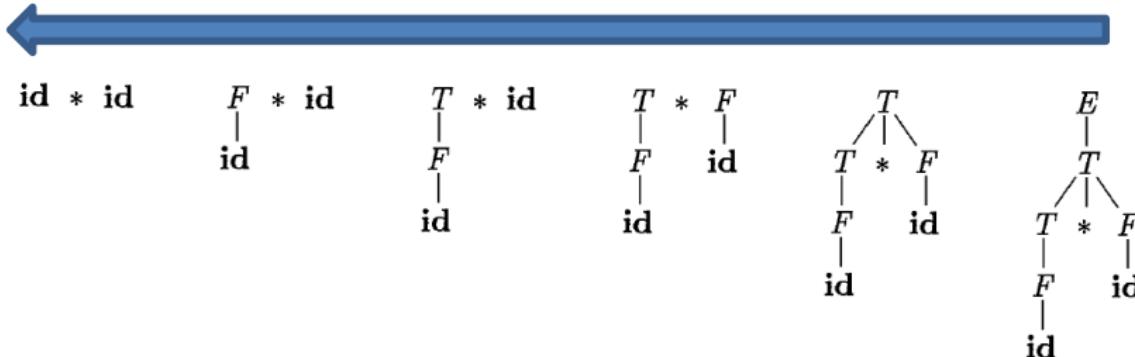
$$\begin{array}{lcl} E & \rightarrow & T \ E' \\ E' & \rightarrow & + \ T \ E' \mid \epsilon \\ T & \rightarrow & F \ T' \\ T' & \rightarrow & * \ F \ T' \mid \epsilon \\ F & \rightarrow & (\ E \) \mid \text{id} \end{array}$$



id + id * id

Bottom Up Parsing

Sentential forms



Derivation --- Rightmost derivation

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \text{id} \Rightarrow F * \text{id} \Rightarrow \text{id} * \text{id}$$

Bottom-up parsing is therefore to construct a **rightmost derivation in reverse**

Reduction

- A specific **substring** of **input** matching the **body** of a production
 - Replaced by the **nonterminal** at the **head** of that production.

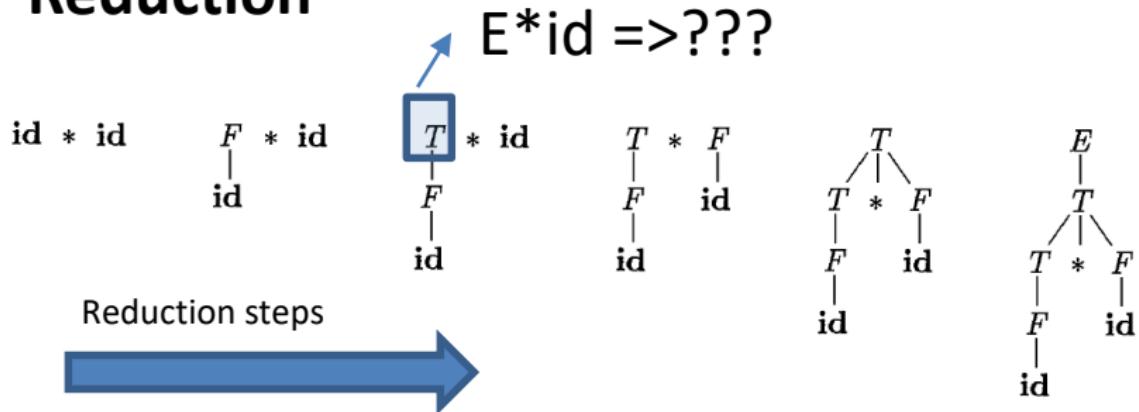
Bcdxy=>Axy Production
 A-> Bcd

- **Bottom-up parsing** as the process of "**reducing**" a string **w** to the **start symbol** of the grammar

Challenges

- (a) **when** to reduce and
- (b) **what production** to apply, as the parse proceeds.

Reduction



Challenges

(a) **when** to reduce and

(b) **what production** to apply, as the parse proceeds.

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

Handle

- “Handle” is a **substring** of **input** that matches the **body of a production**
- Allows reduction => Towards **start symbol**=>reverse of a rightmost derivation

Bcdxy=>Axy

Production
A-> Bcd

Right sentential forms

$\alpha\beta w \Rightarrow \alpha A w$ ↑
Terminals
handle

production $A \rightarrow \beta$

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$id_1 * id_2$	id_1	$F \rightarrow id$
$F * id_2$	F	$T \rightarrow F$
$T * id_2$	id_2	$F \rightarrow id$
$T * F$	$T * F$	$E \rightarrow T * F$

Identifying the **handle** is a challenge

Shift Reduce parsing

Bottom-up parsing in which

- (a) Stack holds **grammar symbols** and
- (b) Input buffer holds the **rest of the string** to be parsed.
- (c) **handle** always appears at the **top of the stack**

Initial config.

STACK
\$

INPUT
 $w \$$



Final config.

STACK
\$ S

INPUT
\$

1. *Shift.* Shift the next input symbol onto the top of the stack.
2. *Reduce.* The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
3. *Accept.* Announce successful completion of parsing.
4. *Error.* Discover a syntax error and call an error recovery routine.

Shift Reduce parsing

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

STACK	INPUT	ACTION
\$	id ₁ * id ₂ \$	shift
\$ id ₁	* id ₂ \$	reduce by $F \rightarrow \text{id}$
\$ F	* id ₂ \$	reduce by $T \rightarrow F$
\$ T	* id ₂ \$	shift
\$ T *	id ₂ \$	shift
\$ T * id ₂	\$	reduce by $F \rightarrow \text{id}$
\$ T * F	\$	reduce by $T \rightarrow T * F$
\$ T	\$	reduce by $E \rightarrow T$
\$ E	\$	accept

Handle always appears at the top of the stack

$$(1) \quad S \xrightarrow[\text{rm}]{*} \alpha A z \xrightarrow[\text{rm}]{*} \alpha \beta B y z \xrightarrow[\text{rm}]{*} \alpha \beta \gamma y z \quad \begin{array}{l} A \rightarrow \beta B y \\ B \rightarrow \gamma \end{array}$$

STACK
\$\alpha\beta\gamma\$

INPUT
\$yz\$

The parser reduces the handle γ to B to reach the configuration

\$\alpha\beta B

\$yz\$

The parser can now shift the string y onto the stack by a sequence of zero or more shift moves to reach the configuration

\$\alpha\beta B y

\$z\$

Handle always appears at the top of the stack

$$(2) \quad S \xrightarrow[\text{rm}]{*} \alpha B x A z \xrightarrow[\text{rm}]{*} \alpha B x y z \xrightarrow[\text{rm}]{*} \alpha \gamma x y z \quad \begin{array}{l} A \rightarrow y \\ B \rightarrow y \end{array}$$

\$\alpha\gamma \qquad \qquad \qquad xyz\$

the handle γ is on top of the stack. After reducing the handle γ to B , the parser can shift the string xy to get the next handle y on top of the stack, ready to be reduced to A :

\$\alpha B x y \qquad \qquad \qquad z\$

Conflict

Shift/reduce conflict: Cannot decide whether to shift or to reduce

Reduce/reduce conflict: Cannot decide which of several reductions to make

Shift/reduce conflict

$$\begin{array}{lcl} \textit{stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\ & | & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \textbf{ else } \textit{stmt} \\ & | & \textbf{other} \end{array}$$

STACK

... **if** *expr* **then** *stmt*

INPUT

else ... \$

Shift Reduce parsing

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

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

LR Parsing

Challenges in shift-reduce parsing

- (a) **when** to reduce and
- (b) **what production** to apply, as the parse proceeds.

Examples:

Simple LR, LR(1), LALR

- LR parser makes **shift-reduce decisions** by LR(0) automaton and maintaining **states**
- State represent sets of **items**

Items

production $A \rightarrow XYZ$ yields the four items

$$A \rightarrow \cdot XYZ$$

$$A \rightarrow X \cdot YZ$$

$$A \rightarrow XY \cdot Z$$

$$A \rightarrow XYZ \cdot$$

production $A \rightarrow \epsilon$ generates only one item, $A \rightarrow \cdot$

Intuitively, an **item** indicates **how much of a production body we have seen** at a given point in the parsing process.

$A \rightarrow \cdot XYZ \rightarrow$ Indicates that **we hope** to see a **string** derivable from **XYZ** on the next input

$A \rightarrow X \cdot YZ \rightarrow$ Indicates that we have **just seen** on the input a **string derivable from X** and that we **hope** next to see a string derivable from **YZ**

$A \rightarrow XYZ \cdot \rightarrow$ Indicates that we have **seen the body XYZ** on input string and that it may be time to **reduce XYZ to A**

Canonical LR(0) collection

- Sets of items => One state
- Collection of sets of items=> *canonical LR(0) collection* => Collection of states

LR(0) automaton: Construct a deterministic **finite automaton** that is used to make **parsing decisions**

To construct the canonical LR(0) collection for a grammar G, we define (a) **augmented grammar** and (b) two functions, **CLOSURE** and **GOTO**

Augmented grammar: If G is a grammar with start symbol S, then the *augmented grammar* G'

start symbol S' and production $S' \rightarrow S$.

Closure of Item Sets

Similar to I



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:

1. Initially, add every item in I to $\text{CLOSURE}(I)$.
2. If $A \rightarrow \alpha \cdot B\beta$ is in $\text{CLOSURE}(I)$ and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow \cdot\gamma$ to $\text{CLOSURE}(I)$, if it is not already there. Apply this rule until no more new items can be added to $\text{CLOSURE}(I)$.

Intuitively, $A \rightarrow \alpha \cdot B\beta$ in $\text{CLOSURE}(I)$ indicates that, at some point in the parsing process, we think we might next see a substring derivable from $B\beta$ as input. The substring derivable from $B\beta$ will have a prefix derivable from B by applying one of the B -productions

We therefore add items for all the B -productions; that is, if $B \rightarrow \gamma$ is a production, we also include $B \rightarrow \cdot\gamma$ in $\text{CLOSURE}(I)$.

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:

1. Initially, add every item in I to $\text{CLOSURE}(I)$.
2. If $A \rightarrow \alpha \cdot B\beta$ is in $\text{CLOSURE}(I)$ and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow \cdot\gamma$ to $\text{CLOSURE}(I)$, if it is not already there. Apply this rule until no more new items can be added to $\text{CLOSURE}(I)$.

```
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 (  $B \rightarrow \cdot\gamma$  is not in  $J$  )  
                    add  $B \rightarrow \cdot\gamma$  to  $J$ ;  
    until no more items are added to  $J$  on one round;  
    return  $J$ ;  
}
```

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:

1. Initially, add every item in I to $\text{CLOSURE}(I)$.
2. If $A \rightarrow \alpha \cdot B\beta$ is in $\text{CLOSURE}(I)$ and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow \cdot\gamma$ to $\text{CLOSURE}(I)$, if it is not already there. Apply this rule until no more new items can be added to $\text{CLOSURE}(I)$.

$$\begin{array}{lcl} E' & \rightarrow & E \\ E & \rightarrow & E + T \quad | \quad T \\ T & \rightarrow & T * F \quad | \quad F \\ E & \rightarrow & (E) \quad | \quad \text{id} \end{array}$$

← Augmentation

$$\begin{array}{l} E' \rightarrow \cdot E \\ E \rightarrow \cdot E + T \\ E \rightarrow \cdot T \\ T \rightarrow \cdot T * F \\ T \rightarrow \cdot F \\ F \rightarrow \cdot (E) \\ F \rightarrow \cdot \text{id} \end{array}$$

If I is the set of one item $\{[E' \rightarrow \cdot E]\}$, then $\text{CLOSURE}(I)$ contains

Closure (I)

$E' \rightarrow \cdot E$

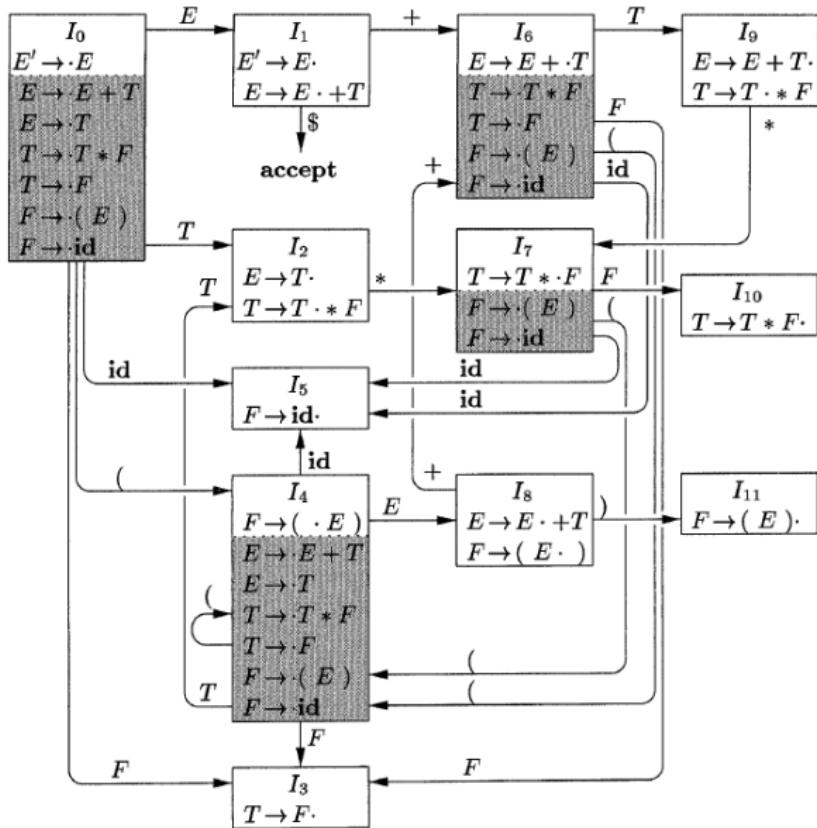
$E \rightarrow \cdot E + T$
 $E \rightarrow \cdot T$
 $T \rightarrow \cdot T * F$
 $T \rightarrow \cdot F$
 $F \rightarrow \cdot (E)$
 $F \rightarrow \cdot \text{id}$

Kernel items: the initial item, $S' \rightarrow \cdot S$, and all items whose dots are not at the left end.

Can be easily derived from Kernel items

Nonkernel items: all items with their dots at the left end, except for $S' \rightarrow \cdot S$.

Closure of Item Sets

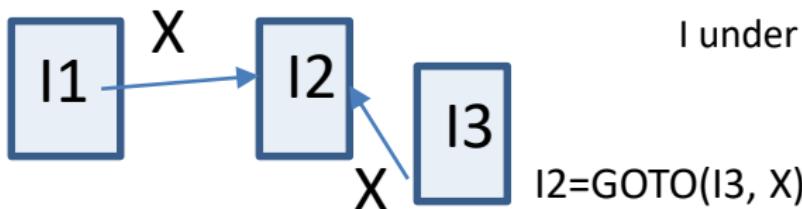


GOTO of Item Sets

- The second useful function is $\text{GOTO}(I, X)$ where I is a set of items and X is a grammar symbol.
- Defines the transitions** in the LR(0) automaton

Assume that $[A \rightarrow \alpha \cdot X \beta]$ is in I .

$\text{GOTO}(I, X)$ is defined to be the closure of the set of all items
 $[A \rightarrow \alpha X \cdot \beta]$



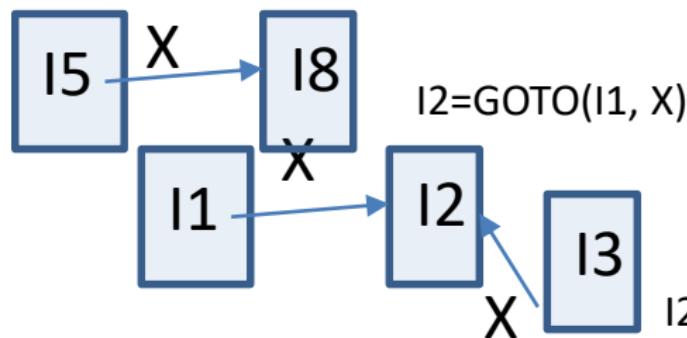
$\text{GOTO}(I, X)$ specifies the **transition** from the state for I under input X

GOTO of Item Sets

- The second useful function is $\text{GOTO}(I, X)$ where I is a set of items and X is a grammar symbol.
- Defines the transitions** in the LR(0) automaton

Assume that $[A \rightarrow \alpha \cdot X \beta]$ is in I .

$\text{GOTO}(I, X)$ is defined to be the closure of the set of all items
 $[A \rightarrow \alpha X \cdot \beta]$



$\text{GOTO}(I, X)$ specifies the **transition** from the state for I under input X

GOTO of Item Sets

If I is the set of two items $\{[E' \rightarrow E\cdot], [E \rightarrow E\cdot + T]\}$

$\text{GOTO}(I, +)$ contains the items

$$E \rightarrow E + \cdot T$$

$$T \rightarrow \cdot T * F$$

$$T \rightarrow \cdot F$$

I1 set

$$F \rightarrow \cdot(E)$$

$$F \rightarrow \cdot\text{id}$$

E'	\rightarrow	E
E	\rightarrow	$E + T \mid T$
T	\rightarrow	$T * F \mid F$
E	\rightarrow	$(E) \mid \text{id}$

Canonical LR(0) collection

LR(0) automaton: Construct a deterministic **finite automaton** that is used to make **parsing decisions**

- **Sets of items => One state**
- **Collection of sets of items=> canonical LR(0) collection => Collection of states**

To construct the canonical LR(0) collection for a grammar G, we define (a) augmented grammar and (b) two functions, **CLOSURE** and **GOTO**

Augmented grammar: If G is a grammar with start symbol S, then the *augmented grammar* G'

start symbol S' and production $S' \rightarrow S$.

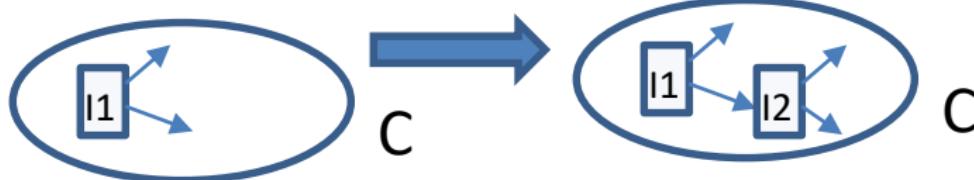
Canonical collection of sets of items

augmented grammar G'

LR(0) automaton

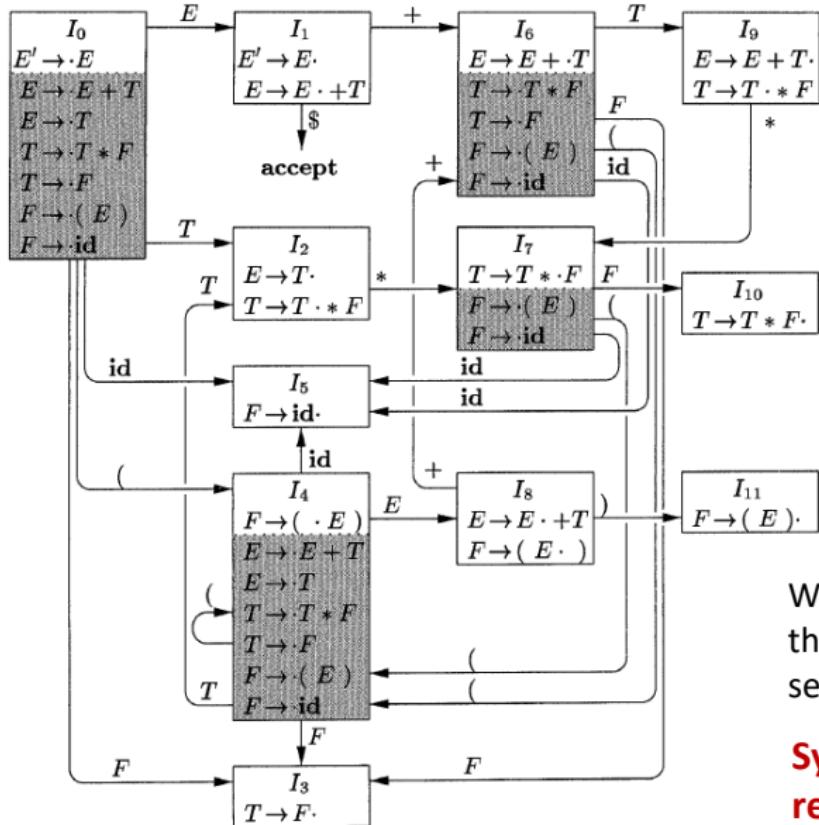
```
void items( $G'$ ) {  
     $C = \text{CLOSURE}(\{[S' \rightarrow \cdot S]\});$   
    repeat  
        for ( each set of items  $I$  in  $C$  )  
            for ( each grammar symbol  $X$  )  
                if (  $\text{GOTO}(I, X)$  is not empty and not in  $C$  )  
                    add 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

}



The start state of the LR(0) automaton is $\text{CLOSURE}(\{[S' \rightarrow \cdot S]\})$

LR(0) automaton



$$\begin{array}{lcl}
 E' & \rightarrow & E \\
 E & \rightarrow & E + T \quad | \quad T \\
 T & \rightarrow & T * F \quad | \quad F \\
 E & \rightarrow & (E) \quad | \quad \text{id}
 \end{array}$$

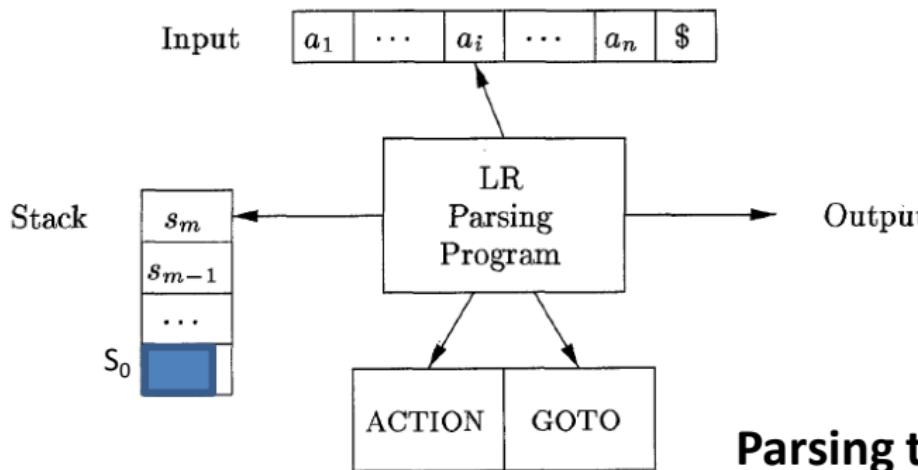
(a) The **states** of this automaton are the sets of items from the **canonical LR(0) collection**.

(b) the **transitions** are given by the **GOTO** function

We say "**state j**" to refer to the state corresponding to the set of items I_j .

Symbol representation : X

LR-Parsing Algorithm



Parsing table

The stack holds a sequence of states $s_0 s_1 \dots s_m$, where s_m is on top.

Where a **shift-reduce** parser **shifts a symbol**, an **LR parser shifts a state**

Top of the stack state (s_m) represents the state of the parser

Role of LR(0) automata in shift-reduce decisions

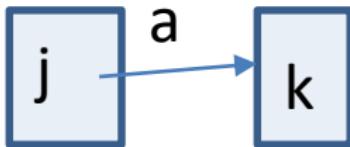
Key Idea

Input: $w=y\mathbf{a}\alpha$

Consider we are in **state j** (maybe after scanning **y symbols**)

Next input **symbol a**

- *If state j has a transition on a.*
 - *Shift (to state k) on next input symbol a*
- *Otherwise, we choose to reduce;*
 - *The items in state j will tell us which production to use*



- All transitions **to state k** must be for the **same grammar symbol a**. Thus, **each state** has a **unique grammar symbol** associated with it (except the start state 0)
- **Multiple states may have same grammar symbol**

Key Idea

States

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	id * id \$	shift to 5
(2)	0 5	\$ id	* id \$	reduce by $F \rightarrow id$
(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 $F \rightarrow id$
(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

Reduction

With symbols,

Reduction is implemented by **popping the body of the production** (the body is **id**) from the stack and **pushing the head of the production (in this case, F)**.

With states, (a) we **pop state 5**, which brings **state 0 to the top** and
(b) look for a **transition on F**, the head of the production.
(c) we **push state 3**

Shift Reduce parsing

Bottom-up parsing in which

- (a) Stack holds **grammar symbols** and
- (b) Input buffer holds the **rest of the string** to be parsed.
- (c) **handle** always appears at the **top of the stack**

Initial config.

STACK
\$

INPUT
 $w \$$



Final config.

STACK
\$ S

INPUT
\$

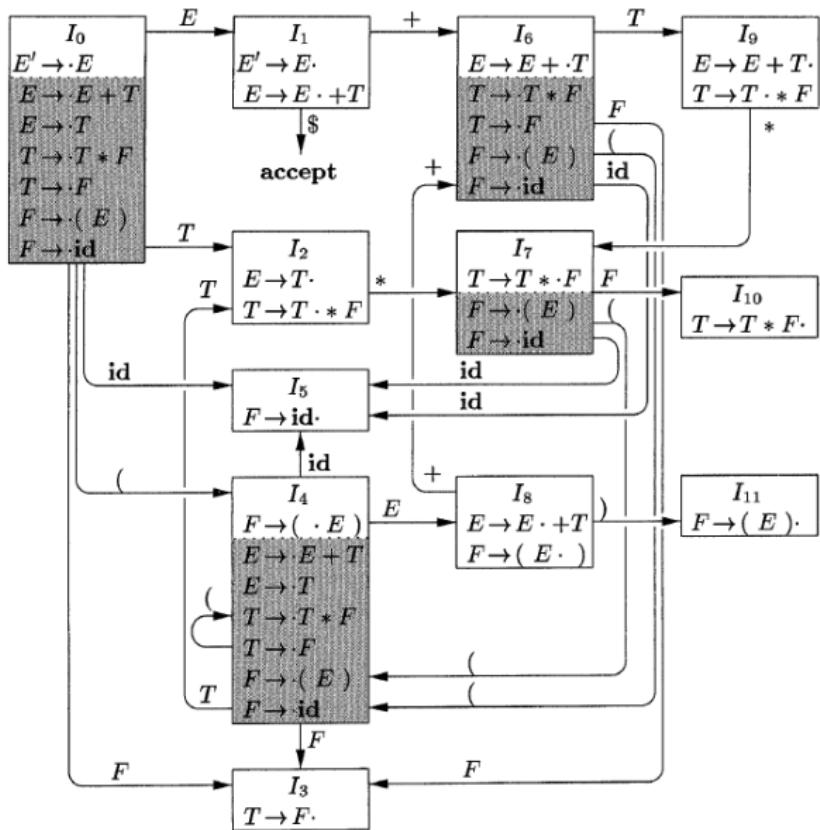
1. *Shift.* Shift the next input symbol onto the top of the stack.
2. *Reduce.* The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
3. *Accept.* Announce successful completion of parsing.
4. *Error.* Discover a syntax error and call an error recovery routine.

Shift Reduce parsing

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

STACK	INPUT	ACTION
\$	id ₁ * id ₂ \$	shift
\$ id ₁	* id ₂ \$	reduce by $F \rightarrow \text{id}$
\$ F	* id ₂ \$	reduce by $T \rightarrow F$
\$ T	* id ₂ \$	shift
\$ T *	id ₂ \$	shift
\$ T * id ₂	\$	reduce by $F \rightarrow \text{id}$
\$ T * F	\$	reduce by $T \rightarrow T * F$
\$ T	\$	reduce by $E \rightarrow T$
\$ E	\$	accept

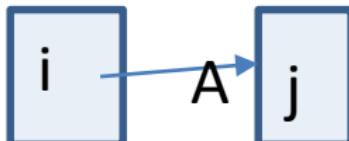
$LR(0)$ automaton



Structure of the LR Parsing Table

The parsing table consists of two parts: a parsing-action function ACTION and a goto function GOTO.

1. The ACTION function takes as arguments a state i and a terminal a (or $\$$, the input endmarker). The value of $ACTION[i, a]$ can have one of four forms:
 - (a) Shift j , where j is a state. The action taken by the parser effectively shifts input a to the stack, but uses state j to represent a .
 - (b) Reduce $A \rightarrow \beta$. The action of the parser effectively reduces β on the top of the stack to head A .  **Pop and push**
 - (c) Accept. The parser accepts the input and finishes parsing.
 - (d) Error. The parser discovers an error in its input and takes some corrective action. We shall have more to say about how such error-recovery routines work in Sections 4.8.3 and 4.9.4.
2. We extend the GOTO function, defined on sets of items, to states: if $GOTO[I_i, A] = I_j$, then GOTO also maps a state i and a nonterminal A to state j .



SLR Parsing table

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

The codes for the actions are:

1. si means shift and stack state i ,

$$(1) \quad E \rightarrow E + T$$

$$(4) \quad T \rightarrow F$$

2. rj means reduce by the production numbered j ,

$$(2) \quad E \rightarrow T$$

$$(5) \quad F \rightarrow (E)$$

3. acc means accept,

$$(3) \quad T \rightarrow T * F$$

$$(6) \quad F \rightarrow \text{id}$$

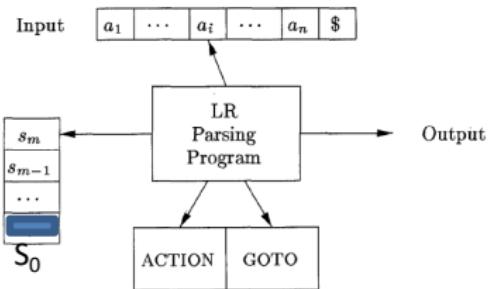
4. blank means error.

LR-parsing algorithm.

METHOD: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer. The parser then executes the program ...

□

```
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  $A \rightarrow \beta$  )
        pop  $|\beta|$  symbols off the stack;
        let state  $t$  now be on top of the stack;
        push GOTO[ $t, A$ ] onto the stack;
        output the production  $A \rightarrow \beta$ ;
    } else if ( ACTION[ $s, a$ ] = accept ) break; /* parsing is done */
    else call error-recovery routine;
}
```



Optional



LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	id * id \$	shift to 5
(2)	0 5	\$ id	* id \$	reduce by $F \rightarrow id$
(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 $F \rightarrow id$
(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

SLR Parsing table

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

The codes for the actions are:

1. si means shift and stack state i ,

$$(1) \quad E \rightarrow E + T$$

$$(4) \quad T \rightarrow F$$

$$(2) \quad E \rightarrow T$$

$$(5) \quad F \rightarrow (E)$$

$$(3) \quad T \rightarrow T * F$$

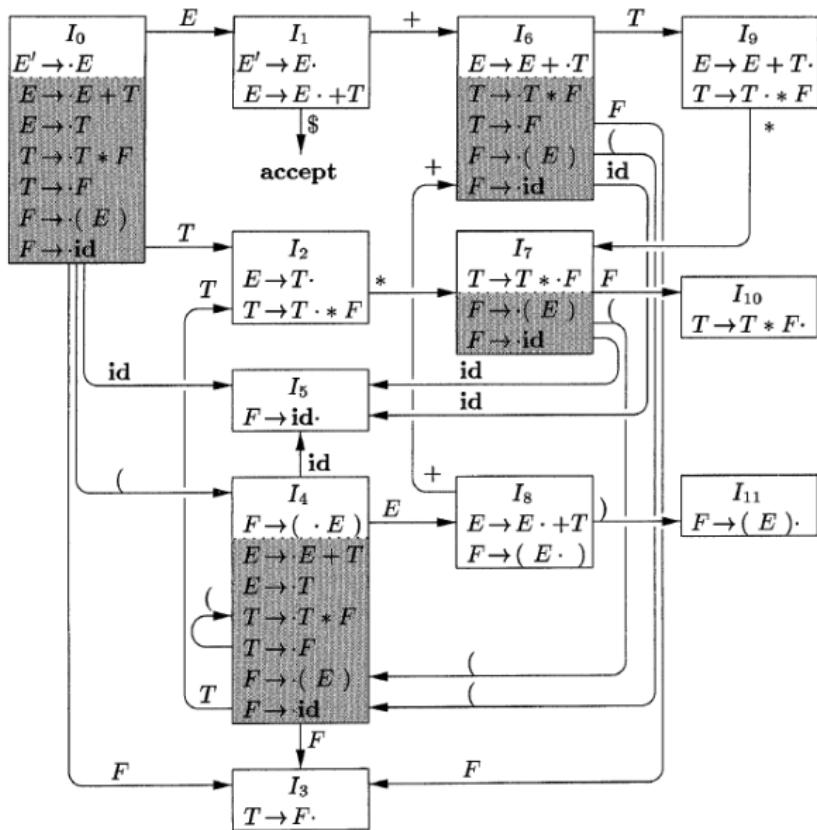
$$(6) \quad F \rightarrow \text{id}$$

2. rj means reduce by the production numbered j ,

3. acc means accept,

4. blank means error.

$LR(0)$ automaton



Constructing SLR-Parsing Tables

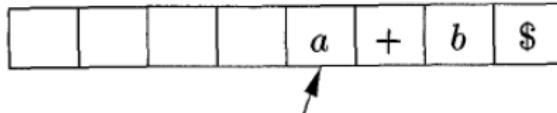
- LR parser using an **SLR-parsing table** as an **SLR parser**
- Same for LR(1), LALR parser
- Step 1: Given a grammar, G , we augment G to produce G' , with a new start symbol S'
- Step 2: Construct **LR(0) items** and **LR(0) automata**
 - We construct **canonical collection of sets of items** for G' together with the GOTO function.
- Step 3: Construct the parsing table
 - Determine the **ACTION** and **GOTO** entries

SLR-Parsing Table: Algorithm

1. Construct $C = \{I_0, I_1, \dots, I_n\}$, the collection of sets of LR(0) items for G' .
2. State i is constructed from I_i . The parsing actions for state i are determined as follows:
 - (a) If $[A \rightarrow \alpha \cdot a\beta]$ 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.
 - (b) If $[A \rightarrow \alpha \cdot]$ is in I_i , then set $\text{ACTION}[i, a]$ to “reduce $A \rightarrow \alpha$ ” for all a in $\text{FOLLOW}(A)$; here A may not be S' .
 - (c) If $[S' \rightarrow S \cdot]$ is in I_i , then set $\text{ACTION}[i, \$]$ to “accept.”

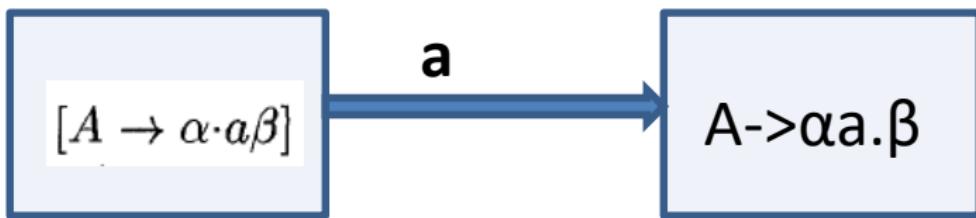
Input string

Input



I_i

I_j



$$\text{GOTO}(I_i, a) = I_j$$

Stack: ... αa looking for an handle

Key Idea States

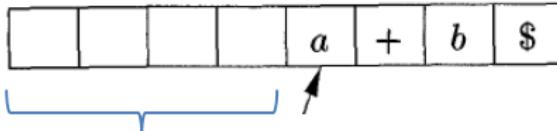
LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	id * id \$	shift to 5
(2)	0 5	\$ id	* id \$	reduce by $F \rightarrow id$
(3)	0 3	\$ F	* id \$	reduce by $T \rightarrow F$
(4)	0 2	\$ T	* id \$	shift to 7
(5)	0 2 7	\$ T *	id \$	shift to 5
(6)	0 2 7 5	\$ T * id	\$	reduce by $F \rightarrow id$
(7)	0 2 7 10	\$ T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

SLR-Parsing Table: Algorithm

1. Construct $C = \{I_0, I_1, \dots, I_n\}$, the collection of sets of LR(0) items for G' .
2. State i is constructed from I_i . The parsing actions for state i are determined as follows:
 - (a) If $[A \rightarrow \alpha \cdot a\beta]$ 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.
 - (b) If $[A \rightarrow \alpha \cdot]$ is in I_i , then set $\text{ACTION}[i, a]$ to “reduce $A \rightarrow \alpha$ ” for all a in $\text{FOLLOW}(A)$; here A may not be S' .
 - (c) If $[S' \rightarrow S \cdot]$ is in I_i , then set $\text{ACTION}[i, \$]$ to “accept.”

Input string

Input



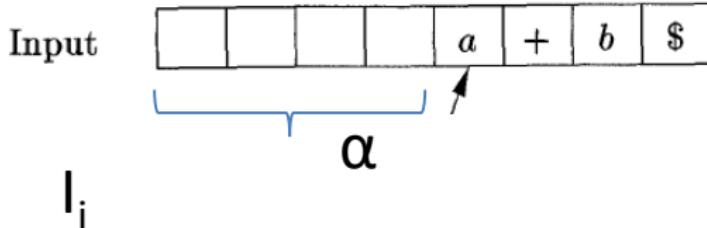
$$[A \rightarrow \alpha \cdot]$$

Stack: ... α ... *May* detected a handle!!

$S \Rightarrow \dots A a \dots \Rightarrow \alpha a$

If this is a sentential form.

Input string



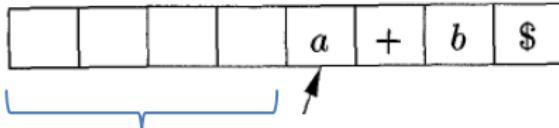
Stack: ... $\alpha\ldots$ *May* detected a handle!!

$S=>..Aa...=>\alpha a$

- If this is a sentential form.
- a follows A

Input string

Input



$|_i$

$$[A \rightarrow \alpha \cdot]$$

Stack: ... $\alpha a ..$ *May* detected a handle!!

$S \Rightarrow ..Aa.. \Rightarrow \alpha a$

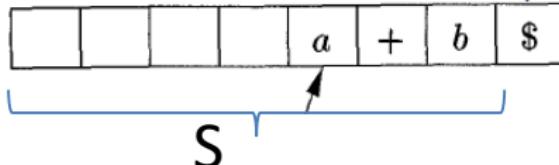
- If this is a sentential form.
- a follows A
- a in $\text{Follow}(A)$!

SLR-Parsing Table: Algorithm

1. Construct $C = \{I_0, I_1, \dots, I_n\}$, the collection of sets of LR(0) items for G' .
2. State i is constructed from I_i . The parsing actions for state i are determined as follows:
 - (a) If $[A \rightarrow \alpha \cdot a\beta]$ 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.
 - (b) If $[A \rightarrow \alpha \cdot]$ is in I_i , then set $\text{ACTION}[i, a]$ to “reduce $A \rightarrow \alpha$ ” for all a in $\text{FOLLOW}(A)$; here A may not be S' .
 - (c) If $[S' \rightarrow S \cdot]$ is in I_i , then set $\text{ACTION}[i, \$]$ to “accept.”

Input string

Input



$[S' \rightarrow S\cdot]$

Done!!

SLR-Parsing Table: Algorithm

1. Construct $C = \{I_0, I_1, \dots, I_n\}$, the collection of sets of LR(0) items for G' .
2. State i is constructed from I_i . The parsing actions for state i are determined as follows:
 - (a) If $[A \rightarrow \alpha \cdot a\beta]$ 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.
 - (b) If $[A \rightarrow \alpha \cdot]$ is in I_i , then set $\text{ACTION}[i, a]$ to “reduce $A \rightarrow \alpha$ ” for all a in $\text{FOLLOW}(A)$; here A may not be S' .
 - (c) 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 SLR(1). The algorithm fails to produce a parser in this case.



SLR Parsing table

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

The codes for the actions are:

1. si means shift and stack state i ,

$$(1) \quad E \rightarrow E + T$$

$$(4) \quad T \rightarrow F$$

2. rj means reduce by the production numbered j ,

$$(2) \quad E \rightarrow T$$

$$(5) \quad F \rightarrow (E)$$

3. acc means accept,

$$(3) \quad T \rightarrow T * F$$

$$(6) \quad F \rightarrow \text{id}$$

4. blank means error.

SLR-Parsing Table: Algorithm

1. Construct $C = \{I_0, I_1, \dots, I_n\}$, the collection of sets of LR(0) items for G' .
2. State i is constructed from I_i . The parsing actions for state i are determined as follows:
 - (a) If $[A \rightarrow \alpha \cdot a\beta]$ 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.
 - (b) If $[A \rightarrow \alpha \cdot]$ is in I_i , then set $\text{ACTION}[i, a]$ to “reduce $A \rightarrow \alpha$ ” for all a in $\text{FOLLOW}(A)$; here A may not be S' .
 - (c) If $[S' \rightarrow S \cdot]$ is in I_i , then set $\text{ACTION}[i, \$]$ to “accept.”
3. The goto transitions for state i are constructed for all nonterminals A using the rule: If $\text{GOTO}(I_i, A) = I_j$, then $\text{GOTO}[i, A] = j$.
4. All entries not defined by rules (2) and (3) are made “error.”
5. The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow \cdot S]$.

SLR-Parsing Table: Example

First consider the set of items I_0 :

$$\begin{aligned} E' &\rightarrow \cdot E \\ E &\rightarrow \cdot E + T \\ E &\rightarrow \cdot T \\ T &\rightarrow \cdot T * F \\ T &\rightarrow \cdot F \\ F &\rightarrow \cdot (E) \\ F &\rightarrow \cdot \text{id} \end{aligned}$$

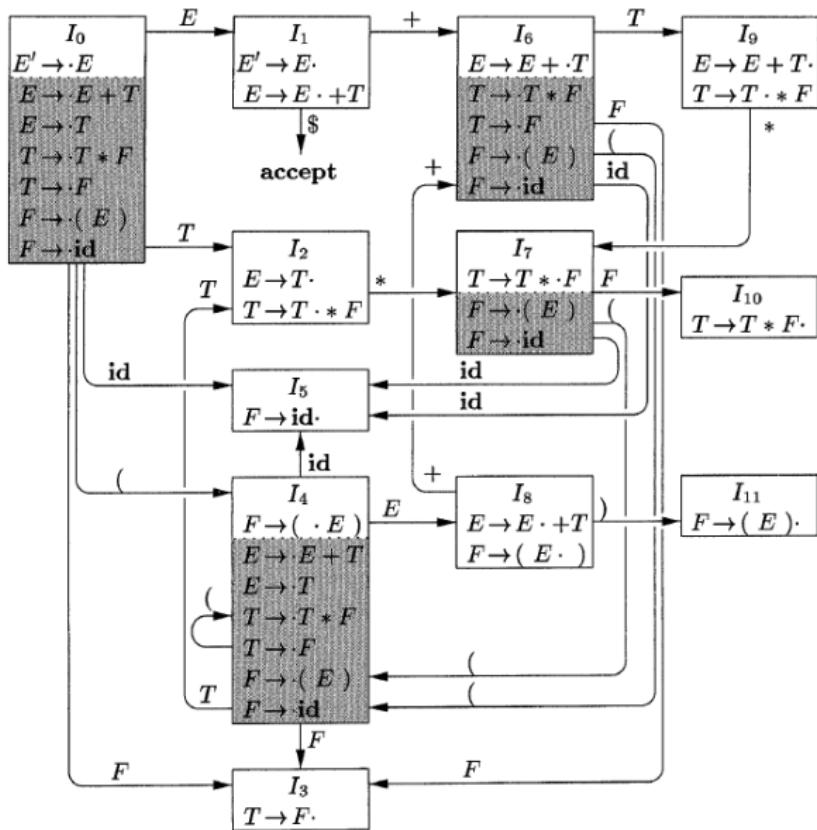
STATE	ACTION					GOTO			
	id	+	*	()	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4			9	3	
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

The item $F \rightarrow \cdot(E)$ gives rise to the entry $\text{ACTION}[0, ()] = \text{shift } 4$, and the item $F \rightarrow \cdot \text{id}$ to the entry $\text{ACTION}[0, \text{id}] = \text{shift } 5$. Other items in I_0 yield no actions. Now consider I_1 :

$$\begin{aligned} E' &\rightarrow E \cdot \\ E &\rightarrow E \cdot + T \end{aligned}$$

The first item yields $\text{ACTION}[1, \$] = \text{accept}$, and the second yields $\text{ACTION}[1, +] = \text{shift } 6$.

$LR(0)$ automaton



SLR-Parsing Table: Example

Next consider I_2 :

$$\begin{aligned}E &\rightarrow T \cdot \\T &\rightarrow T \cdot * F\end{aligned}$$

Since $\text{FOLLOW}(E) = \{\$, +,)\}$, the first item makes

$\text{ACTION}[2, \$] = \text{ACTION}[2, +] = \text{ACTION}[2,]) = \text{reduce } E \rightarrow T$

The second item makes $\text{ACTION}[2, *] = \text{shift 7}$. Continuing in this fashion

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

- (1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$

- (4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow \text{id}$

Non-SLR: Example

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

Grammar

$I_2:$ $S \rightarrow L \cdot = R$
 $R \rightarrow L \cdot$

=

$I_6:$ $S \rightarrow L = \cdot R$
 $R \rightarrow \cdot L$
 $L \rightarrow \cdot * R$
 $L \rightarrow \cdot \text{id}$

ACTION[2, =] \Rightarrow “shift 6.”

Conflicting action!!

FOLLOW(R) contains $= \Rightarrow$ ACTION[2, =] to “reduce $R \rightarrow L$.”

Non-SLR: Where is the problem?

$$\begin{array}{l} S \rightarrow L = R \mid R \\ L \rightarrow *R \mid id \\ R \rightarrow L \end{array}$$

id=*id

Right sentential derivation

S->L=R -> L=L -> L=*&R -> L=*&L -> **L=*&id** -> id=*&id

Stack: \$

Input string: id=*&id\$

SLR parsing

Stack: \$ id

Input string: =*&id\$

Stack: \$ L

Input string: =*&id\$ (Reduction with R->L??)

Stack: \$ L=

Input string: *id\$

Stack: \$ L=*&id

Input string: \$

Stack: \$ S

Input string: \$

Non-SLR: Where is the problem?

$$\begin{array}{l} S \rightarrow L = R \mid R \\ L \rightarrow *R \mid id \\ R \rightarrow L \end{array}$$

id=*id

Right sentential derivation

S->L=R -> L=L -> L=*&R -> L=*&L -> **L=*&id** -> id=*&id

Stack: \$

Input string: id=*&id\$

SLR parsing

Stack: \$ id

Input string: =*&id\$

Stack: \$ L

Input string: =*&id\$ (Reduction with R->L??)

Stack: \$ R

Input string: =*&id\$

Incorrect!

Stack: \$ L=*&id

Input string: \$

Stack: \$ S

Input string: \$

Viable Prefixes

- The **LR(0) automaton** characterizes the **strings of grammar symbols** that can appear on the **stack** of a shift-reduce parser for the grammar.
- The **stack** contents must be a **prefix** of a **right-sentential form**.
- If the **stack holds α** and the **rest of the input is x** , then a sequence of reductions will take αx to S .

$$S \xrightarrow[\text{rm}]{*'} \alpha x.$$

Not all prefixes of right-sentential forms can appear on the **stack**

$$E \xrightarrow[\text{rm}]{*} F * \mathbf{id} \xrightarrow[\text{rm}]{*} (E) * \mathbf{id}$$

The **prefixes** of right sentential forms that can **appear on the stack** of a shift reduce parser are called **viable prefixes**.

Handle always appears at the top of the stack

(1) $S \xrightarrow[\text{rm}]{*} \alpha Az \xrightarrow[\text{rm}]{*} \alpha\beta Byz \xrightarrow[\text{rm}]{*} \boxed{\alpha\beta\gamma}yz$

A-> β By
B-> γ

STACK
 $\$ \alpha\beta\gamma$

INPUT
 $yz\$$

The parser reduces the handle γ to B to reach the configuration

$\$ \alpha\beta B$

$yz\$$

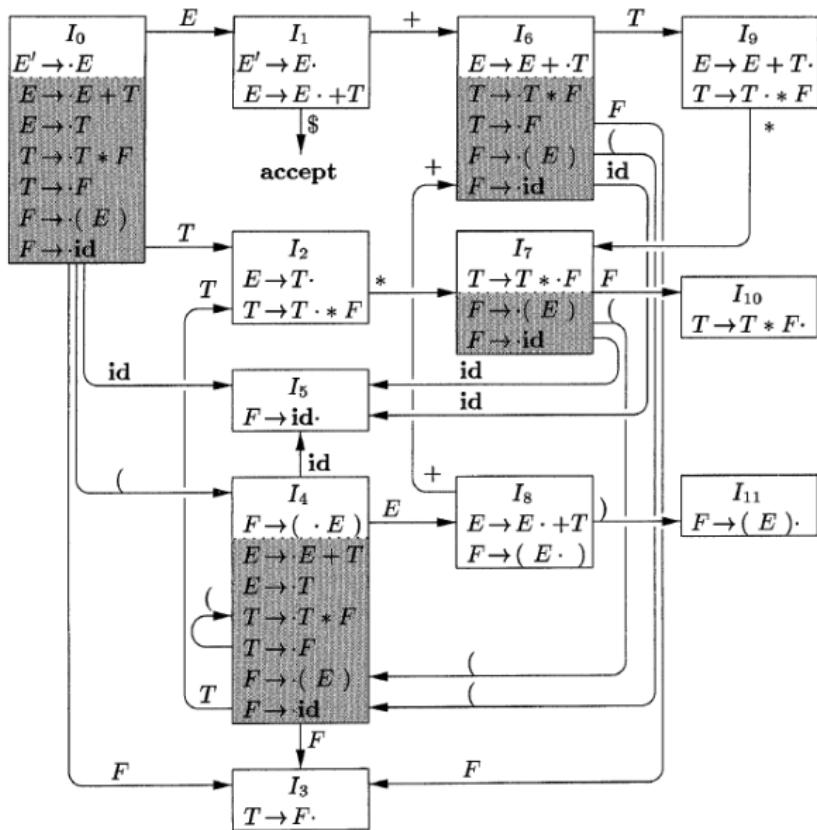
The parser can now shift the string y onto the stack by a sequence of zero or more shift moves to reach the configuration

$\$ \alpha\beta By$

$z\$$

LR(0) automaton

Viable prefix E+T



Viable Prefixes

- the set of **valid items** for a **viable prefix γ** is
 - **Set of items reached from the initial state S along the path labeled γ in the LR(0) automaton**

SLR parsing is based on the fact that LR(0) automata recognize **viable prefixes and valid items**.

We say item $A \rightarrow \beta_1 \cdot \beta_2$ is *valid* for a viable prefix $\alpha\beta_1$ if there is a derivation $S' \xrightarrow[\text{rm}]{*} \alpha Aw \xrightarrow[\text{rm}]{*} \alpha\beta_1\beta_2w$. In general, an item will be valid for many viable prefixes.

$A \rightarrow \beta_1 \cdot \beta_2$ is valid for $\alpha\beta_1$ Viable prefix

if $\beta_2 \neq \epsilon$, Shift

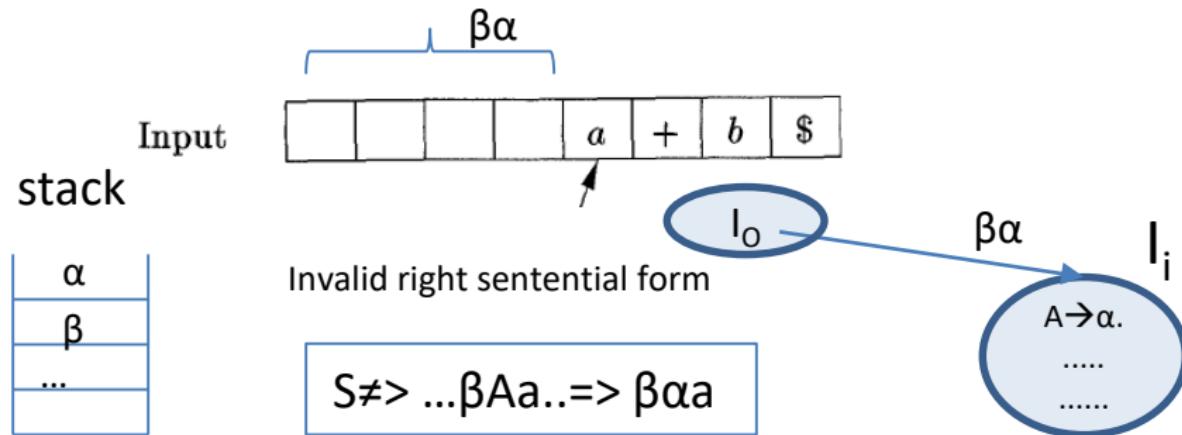
If $\beta_2 = \epsilon$, then it looks as if $A \rightarrow \beta_1$ is the handle,

Reduction

SLR says...

- (b) 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' .

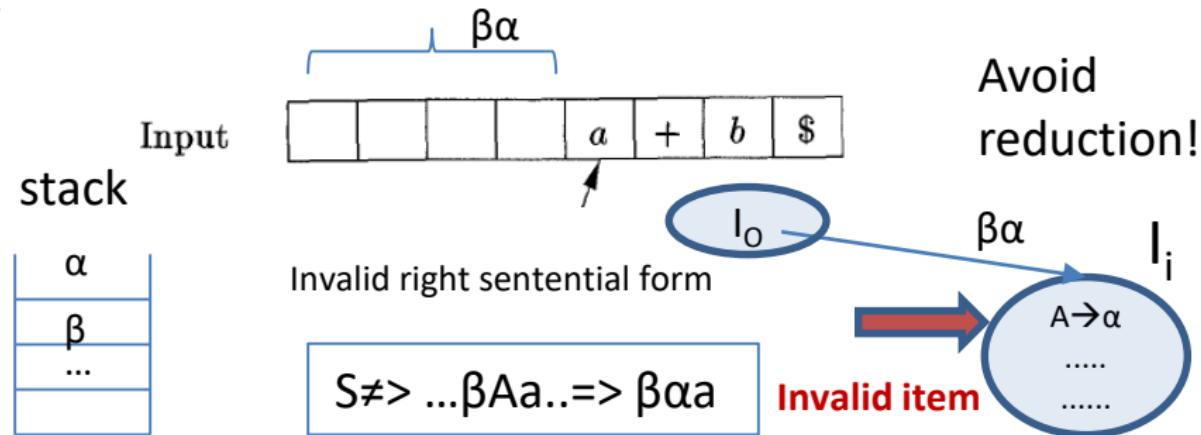
In some situations, however, when state i appears on top of the stack, the viable prefix $\beta\alpha$ on the stack is such that βA cannot be followed by a in any right-sentential form. Thus, the reduction by $A \rightarrow \alpha$ should be invalid on input a .



SLR says...

- (b) 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' .

In some situations, however, when state i appears on top of the stack, the viable prefix $\beta\alpha$ on the stack is such that βA cannot be followed by a in any right-sentential form. Thus, the reduction by $A \rightarrow \alpha$ should be invalid on input a .



Avoid
reduction!

Non-SLR: Where is the problem?

$$\begin{array}{l} S \rightarrow L = R \mid R \\ L \rightarrow *R \mid id \\ R \rightarrow L \end{array}$$

id=*id

Right sentential derivation

S->L=R -> L=L -> L=*&R -> L=*&L -> **L=*&id** -> id=*&id

Stack: \$

Input string: id=*&id\$

SLR parsing

Stack: \$ id

Input string: =*&id\$

Stack: \$ L

Input string: =*&id\$ (Reduction with R->L??)

Stack: \$ L=

Input string: *id\$

Stack: \$ L=*&id

Input string: \$

Stack: \$ S

Input string: \$

Non-SLR: Where is the problem?

$$\begin{array}{l} S \rightarrow L = R \mid R \\ L \rightarrow *R \mid id \\ R \rightarrow L \end{array}$$

id=*id

Right sentential derivation

S->L=R -> L=L -> L=*&R -> L=*&L -> **L=*&id** -> id=*&id

Stack: \$

Input string: id=*&id\$

SLR parsing

Stack: \$ id

Input string: =*&id\$

Stack: \$ L

Input string: =*&id\$ (Reduction with R->L??)

Stack: \$ R

Input string: =*&id\$

Incorrect!

Stack: \$ L=*&id

Input string: \$

Stack: \$ S

Input string: \$

Non-SLR: Example

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

Grammar

$I_2:$ $S \rightarrow L \cdot = R$
 $R \rightarrow L \cdot$

=

$I_6:$ $S \rightarrow L = \cdot R$
 $R \rightarrow \cdot L$
 $L \rightarrow \cdot * R$
 $L \rightarrow \cdot \text{id}$

ACTION[2, =] \Rightarrow “shift 6.”

Conflicting action!!

FOLLOW(R) contains $= \Rightarrow$ ACTION[2, =] to “reduce $R \rightarrow L$.”

Non-SLR: Where is the problem?

- (b) 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' .

FOLLOW(R) contains =

Since $S \Rightarrow L = R \Rightarrow *R = R$ *id=id

It is possible to **carry extra information in the state** that will allow us to **rule out** some of these **invalid reductions**

LR(1) Parser, CLR

- (b) If $[A \rightarrow \alpha \cdot]$ is in I_i , then set $\text{ACTION}[i, a]$ to “reduce $A \rightarrow \alpha$ ” for all a in $\text{FOLLOW}(A)$; here A may not be S' .

$\text{FOLLOW}(R)$ contains =

Since $S \Rightarrow L = R \Rightarrow *R = R$

It is possible to **carry extra information in the state** that will allow us to **rule out** some of these **invalid reductions**

- **Splitting states**
- **Each state** of an LR parser indicates **exactly which input symbols** can **follow** a **handle** α for which there is a possible **reduction to A**
- This **extra information** is incorporated into the state by **redefining items** to include a **terminal symbol** as a **second component**.

LR(1) Parser



The extra information is incorporated into the state by redefining item to include a terminal symbol as a second component. The general form of an item becomes $[A \rightarrow \alpha \cdot \beta, a]$, where $A \rightarrow \alpha\beta$ is a production and a is a terminal or the right endmarker $\$$. We call such an object an *LR(1) item*.

an item of the form $[A \rightarrow \alpha \cdot, a]$ calls for a reduction by $A \rightarrow \alpha$ if the next input symbol is a .

Thus, we are compelled to reduce by $A \rightarrow \alpha$ only on those input symbols a for which $[A \rightarrow \alpha \cdot, a]$ is an LR(1) item in the state on top of the stack. The set of such a 's will always be a subset of $\text{FOLLOW}(A)$,

Look-ahead a is implicit for SLR

lookahead has no effect in an item of the form $[A \rightarrow \alpha \cdot \beta, a]$, where β is not ϵ ,

LR(1) Sets of Items

```
SetOfItems CLOSURE( $I$ ) {
```

```
    repeat
```

```
        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 (  $B \rightarrow \cdot\gamma$  is not in  $J$  )  
                            add  $B \rightarrow \cdot\gamma$  to  $J$ ;
```

```
                until no more items are added to  $J$  on one round;
```

```
            return  $J$ ;
```

```
}
```

```
        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  $I$ ;
```

```
}
```

consider an item of the form $[A \rightarrow \alpha \cdot B\beta, a]$

$$S \xrightarrow{*} \delta Aax \Rightarrow \delta\alpha B\beta ax$$

$\underbrace{}_{\text{by}}$

$$B \rightarrow \eta \qquad \gamma = \delta\alpha$$

$S \xrightarrow{*} \gamma Bby \Rightarrow \gamma\eta by$. Thus, $[B \rightarrow \cdot\eta, b]$ is valid for γ .

b can be any terminal $\text{FIRST}(\beta a)$.

Closure of Item Sets – LR(1)

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & C \ C \\ C & \rightarrow & c \ C \ | \ d \end{array} \quad [A \rightarrow \alpha \cdot B \beta, a] \quad \text{add } [B \rightarrow \cdot \gamma, b] \text{ for each production } B \rightarrow \gamma \text{ and terminal } b \text{ in FIRST}(\beta a).$$

closure of $\{[S' \rightarrow \cdot S, \$]\}$

we add $[S \rightarrow \cdot CC, \$]$. FIRST(βa) β is ϵ a is $\$$,

Closure of Item Sets

$$\begin{array}{lcl} S' \rightarrow S & & [A \rightarrow \alpha \cdot B \beta, a] \\ S \rightarrow C C & \text{add } [B \rightarrow \cdot \gamma, b] \text{ for each production } B \rightarrow \gamma \text{ and terminal } b \text{ in FIRST}(\beta a). \\ C \rightarrow c C \mid d & & \end{array}$$

closure of $\{[S' \rightarrow \cdot S, \$]\}$

we add $[S \rightarrow \cdot CC, \$]$. FIRST(βa) β is ϵ a is $\$$,

adding all items $[C \rightarrow \cdot \gamma, b]$ for b in FIRST($C \$$)

$$\text{FIRST}(C \$) = \text{FIRST}(C)$$

FIRST(C) contains terminals c and d

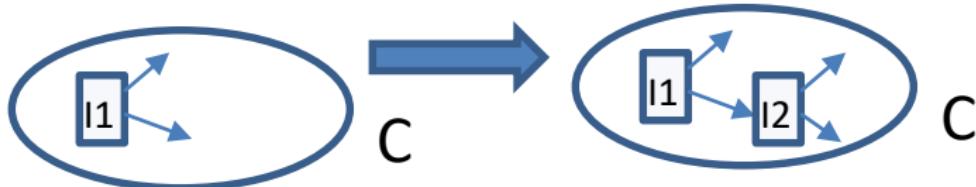
$$\begin{aligned} I_0 : \quad & S \rightarrow \cdot S, \$ \\ & S \rightarrow \cdot CC, \$ \\ & C \rightarrow \cdot cC, c/d \\ & C \rightarrow \cdot d, c/d \end{aligned}$$

LR(1) automation -- GOTO

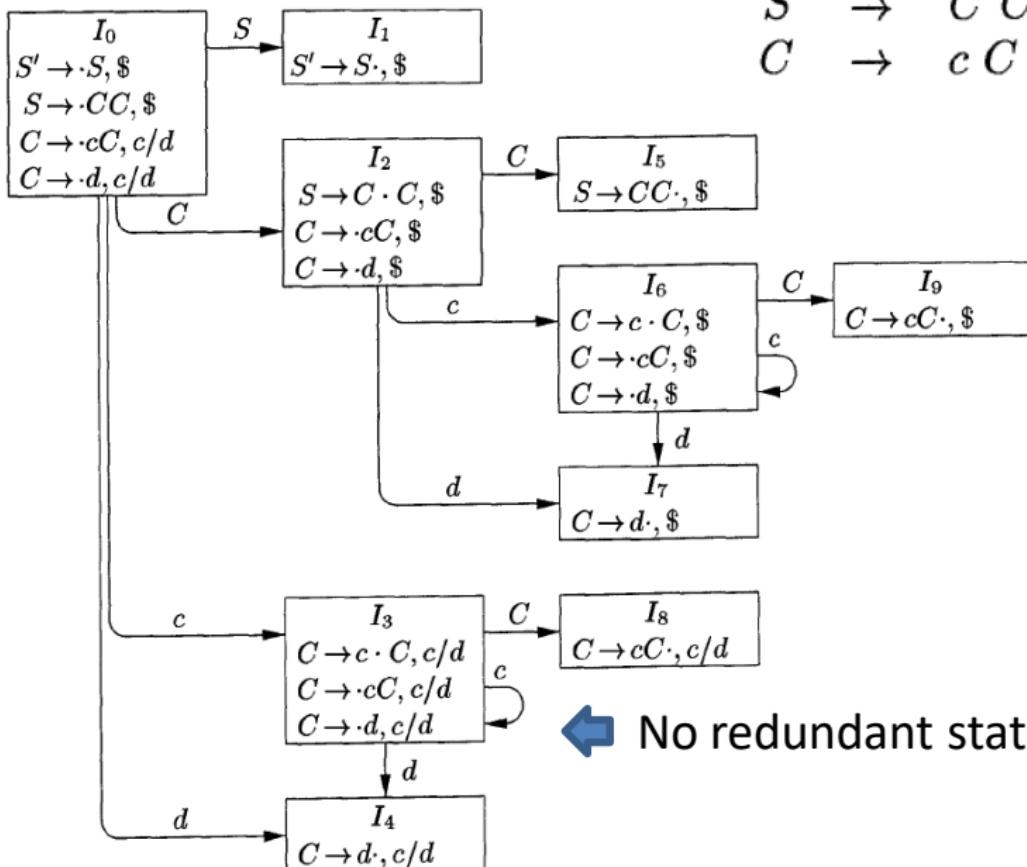
```
SetOfItems GOTO( $I, X$ ) {  
    initialize  $J$  to be the empty set;  
    for ( each item  $[A \rightarrow \alpha \cdot X \beta, a]$  in  $I$  )  
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;  
    return CLOSURE( $J$ );  
}
```

LR(1) automation

```
void items( $G'$ ) {  
    initialize  $C$  to CLOSURE( $\{[S' \rightarrow \cdot S, \$]\}$ );  
    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  $C$ ;  
}
```



LR(1) automation

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & CC \\ C & \rightarrow & cC \mid d \end{array}$$


LR(1) Parsing table

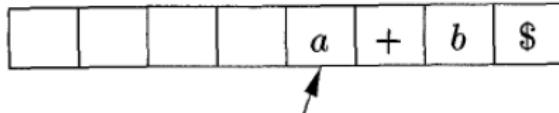
1. Construct $C' = \{I_0, I_1, \dots, I_n\}$, the collection of sets of LR(1) items for G' .
2. State i of the parser is constructed from I_i . The parsing action for state i is determined as follows.
 - (a) If $[A \rightarrow \alpha \cdot a \beta, b]$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ to “shift j .” Here a must be a terminal. **b is not important**
 - (b) If $[A \rightarrow \alpha \cdot, a]$ is in I_i , $A \neq S'$, then set $\text{ACTION}[i, a]$ to “reduce $A \rightarrow \alpha$.”
 - (c) 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 algorithm fails to produce a parser in this case.

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

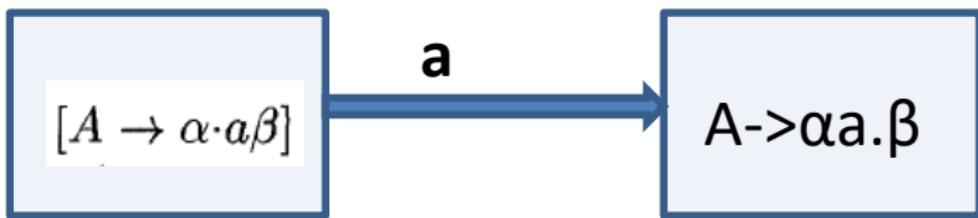
Input string

Input



I_i

I_j

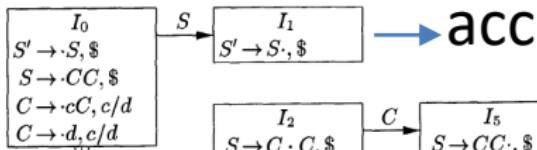


$$\text{GOTO}(I_i, a) = I_j$$

Stack: ... αa expecting an handle

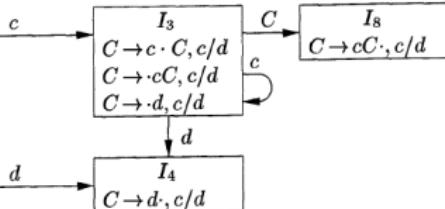
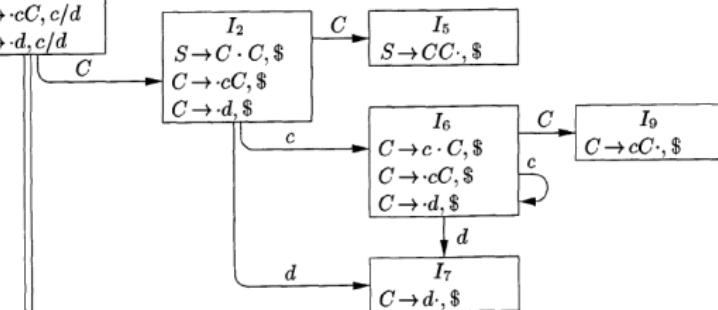
LR(1) Parsing table

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			r3		
8	r2	r2			
9			r2		



acc

Input: dd

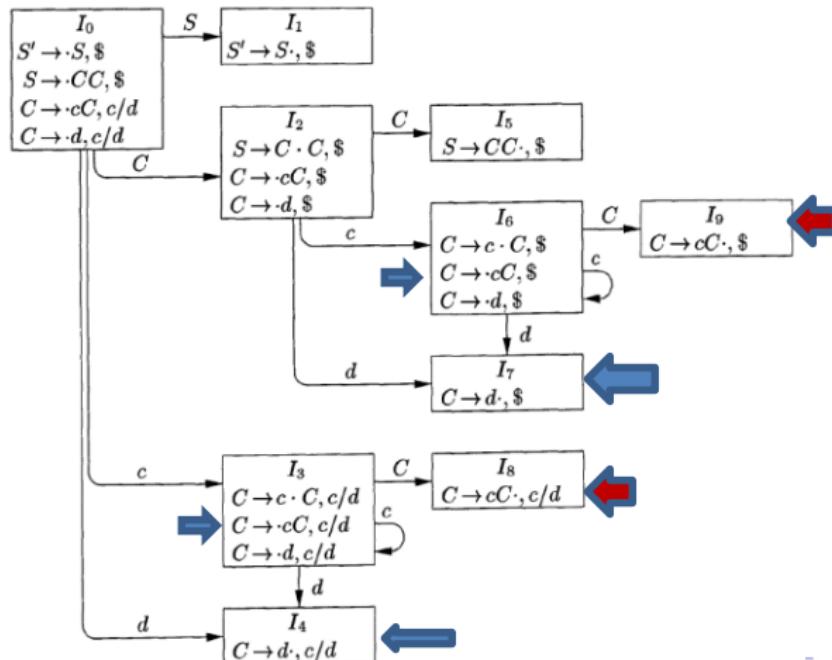


Stack	Input
\$0	dd\$
\$0 4	d\$
\$0 2	d\$ C->d
\$0 2 7	\$
\$0 2 5	\$ C->d
\$0 1	\$ S->CC

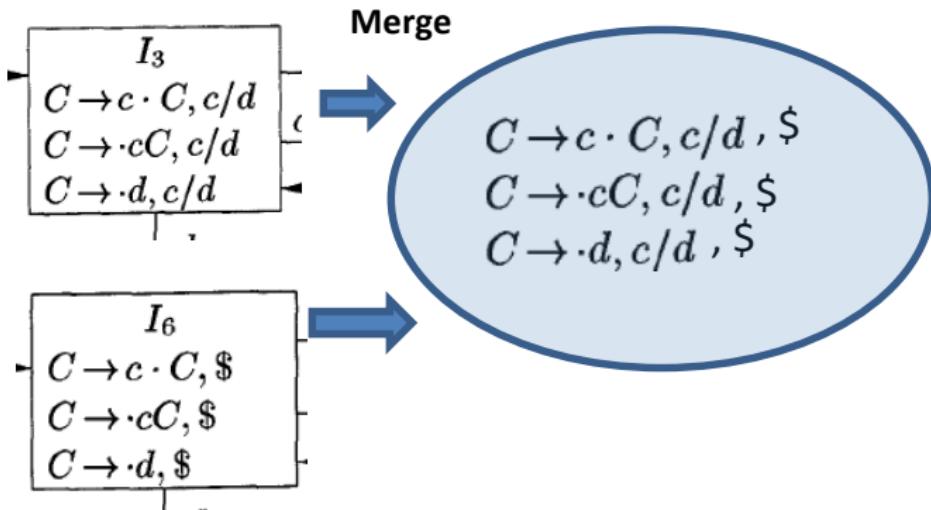
LALR

- Considerably **smaller** than the canonical LR tables
- Most common syntactic constructs of programming languages can be expressed conveniently by an LALR grammar

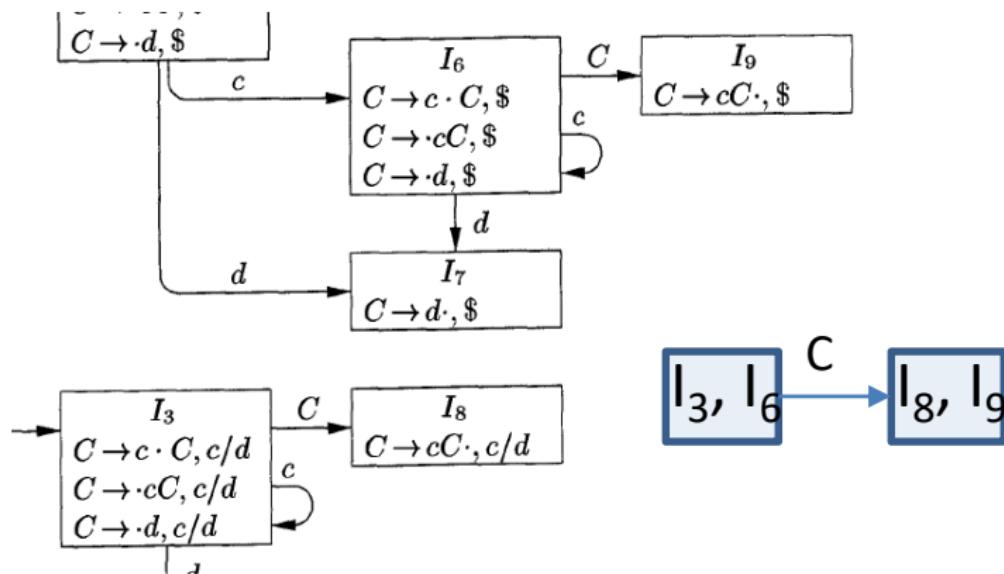
Same core items, different lookahead



- Sets of LR(1) items having the **same core**, that is, set of first components,
- Merge these sets with common cores into one set of **LALR items**.



LALR -- GOTO



- Since the core of $\text{GOTO}(I, X)$ depends **only on the core**,
 - Goto's of merged sets can themselves be merged.
- Thus, there is no problem revising the GOTO function as we merge sets of items.

LR(1) automation -- GOTO

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

LALR Parsing table

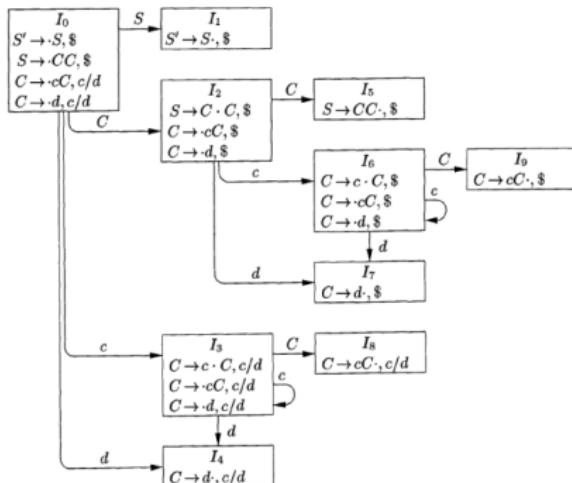
1. Construct $C = \{I_0, I_1, \dots, I_n\}$, the collection of sets of LR(1) items.
2. For each core present among the set of LR(1) items, find all sets having that core, and replace these sets by their union.
3. Let $C' = \{J_0, J_1, \dots, J_m\}$ be the resulting sets of LR(1) items. The parsing actions for state i are constructed from J_i in the same manner as in Algorithm 4.56. If there is a parsing action conflict, the algorithm fails to produce a parser, and the grammar is said not to be LALR(1).
4. The GOTO table is constructed as follows. If J is the union of one or more sets of LR(1) items, that is, $J = I_1 \cap I_2 \cap \dots \cap I_k$, then the cores of $\text{GOTO}(I_1, X), \text{GOTO}(I_2, X), \dots, \text{GOTO}(I_k, X)$ 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 $\text{GOTO}(I_1, X)$. Then $\text{GOTO}(J, X) = K$.

$I_{36}:$ $C \rightarrow c \cdot C, \quad c/d/\$$
 $C \rightarrow \cdot cC, \quad c/d/\$$
 $C \rightarrow \cdot d, \quad c/d/\$$

$I_{47}:$ $C \rightarrow d \cdot, \quad c/d/\$$

$I_{89}:$ $C \rightarrow cC \cdot, \quad c/d/\$$

STATE	ACTION			GOTO	
	<i>c</i>	<i>d</i>	$\$$	<i>S</i>	<i>C</i>
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		



LALR conflicts

LALR item

$$[A \rightarrow \alpha \cdot, a]$$
$$[B \rightarrow \beta \cdot a \gamma, b]$$

Shift reduce conflict on a

- Shares same core in LR(1)!!
- Same conflict for LR(1)!

LR(1) items

$$\{[A \rightarrow c \cdot, d], [B \rightarrow c \cdot, e]\}$$
$$\{[A \rightarrow c \cdot, e], [B \rightarrow c \cdot, d]\}$$

No Reduce-reduce conflict on d, e

LALR item!

$$\begin{array}{l} A \rightarrow c \cdot, d/e \\ B \rightarrow c \cdot, d/e \end{array}$$

Reduce-reduce conflict on d, e!

Efficient Construction of LALR Parsing Tables

- First, we can represent any set of LR(0) or LR(1) items I by its kernel, that is, by those items that are either the initial item — $[S' \rightarrow \cdot S]$ or $[S' \rightarrow \cdot S, \$]$ — or that have the dot somewhere other than at the beginning of the production body.
- We can construct the LALR(1)-item kernels from the LR(0)-item kernels by a process of propagation and spontaneous generation of lookaheads, that we shall describe shortly.
- If we have the LALR(1) kernels, we can generate the LALR(1) parsing table by closing each kernel, using the function CLOSURE of Fig. 4.40, and then computing table entries by Algorithm 4.56, as if the LALR(1) sets of items were canonical LR(1) sets of items.

We must attach the proper lookaheads to the LR(0) items in the kernels, to create the kernels of the sets of LALR(l) items.

LR(0) item $B \rightarrow \gamma\cdot\delta$

1. There is a set of items I , with a kernel item $A \rightarrow \alpha\cdot\beta, a$, and $J = \text{GOTO}(I, X)$, and the construction of

$$\text{GOTO}(\text{CLOSURE}(\{[A \rightarrow \alpha\cdot\beta, a]\}), X)$$

as given in Fig. 4.40, contains $[B \rightarrow \gamma\cdot\delta, b]$, regardless of a . Such a lookahead b is said to be generated *spontaneously* for $B \rightarrow \gamma\cdot\delta$.

2. As a special case, lookahead $\$$ is generated spontaneously for the item $S' \rightarrow \cdot S$ in the initial set of items.

LR(1) Sets of Items

```
SetOfItems CLOSURE( $I$ ) {
```

```
    repeat
```

```
        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 (  $B \rightarrow \cdot\gamma$  is not in  $J$  )
```

```
                            add  $B \rightarrow \cdot\gamma$  to  $J$ ;
```

```
            until no more items are added to  $J$  on one round;
```

```
            return  $J$ ;
```

```
}
```

```
        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  $I$ ;
```

```
}
```

consider an item of the form $[A \rightarrow \alpha \cdot B\beta, a]$

$$S \xrightarrow{*} \delta Aax \Rightarrow \delta\alpha B\beta ax$$

by

$$B \rightarrow \eta \quad \gamma = \delta\alpha$$

$S \xrightarrow{*} \gamma Bby \Rightarrow \gamma\eta by$. Thus, $[B \rightarrow \cdot\eta, b]$ is valid for γ .

b can be any terminal $\text{FIRST}(\beta a)$.

We must attach the proper lookaheads to the LR(0) items in the kernels, to create the kernels of the sets of LALR(I) items.

LR(0) item $B \rightarrow \gamma\cdot\delta$

3. All is as in (1), but $a = b$, and $\text{GOTO}(\text{CLOSURE}(\{[A \rightarrow \alpha\cdot\beta, b]\}), X)$, as given in Fig. 4.40, contains $[B \rightarrow \gamma\cdot\delta, b]$ only because $A \rightarrow \alpha\cdot\beta$ has b as one of its associated lookaheads. In such a case, we say that lookaheads *propagate* from $A \rightarrow \alpha\cdot\beta$ in the kernel of I to $B \rightarrow \gamma\cdot\delta$ in the kernel of J . Note that propagation does not depend on the particular lookahead symbol; either all lookaheads propagate from one item to another, or none do.

LR(1) automation -- GOTO

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

We need to determine the spontaneously generated lookaheads for each set of LR(0) items, and also to determine which items propagate lookaheads from which. The test is actually quite simple. Let $\#$ be a symbol not in the grammar at hand. Let $A \rightarrow \alpha \cdot \beta$ be a kernel LR(0) item in set I . Compute, for each X , $J = \text{GOTO}(\text{CLOSURE}(\{[A \rightarrow \alpha \cdot \beta, \#]\}), X)$. For each kernel item in J , we examine its set of lookaheads. If $\#$ is a lookahead, then lookaheads propagate to that item from $A \rightarrow \alpha \cdot \beta$. Any other lookahead is spontaneously generated.

INPUT: The kernel K of a set of LR(0) items I and a grammar symbol X .

OUTPUT: The lookaheads spontaneously generated by items in I for kernel items in $\text{GOTO}(I, X)$ and the items in I from which lookaheads are propagated to kernel items in $\text{GOTO}(I, X)$.

METHOD:

```
for ( each item  $A \rightarrow \alpha\cdot\beta$  in  $K$  ) {
     $J := \text{CLOSURE}(\{[A \rightarrow \alpha\cdot\beta, \#]\})$ ;
    if (  $[B \rightarrow \gamma\cdot X \delta, a]$  is in  $J$ , and  $a$  is not  $\#$  )
        conclude that lookahead  $a$  is generated spontaneously for item
         $B \rightarrow \gamma X \cdot \delta$  in  $\text{GOTO}(I, X)$ ;
    if (  $[B \rightarrow \gamma\cdot X \delta, \#]$  is in  $J$  )
        conclude that lookaheads propagate from  $A \rightarrow \alpha\cdot\beta$  in  $I$  to
         $B \rightarrow \gamma X \cdot \delta$  in  $\text{GOTO}(I, X)$ ;
}
```

- We are now ready to **attach lookaheads** to the kernels of the sets of **LR(0) items** to form the **sets of LALR(I) items**.
- First, we know that **\$ is a lookahead for $S' \rightarrow .S$** in the initial set of LR(0) items.
- Algorithm gives us **all the lookaheads** generated **spontaneously**.
- After listing all those **lookaheads**, we must **allow them to propagate** until no further propagation is possible.
- Keep track of "**new**" **lookaheads** that have **propagated into** an item but which have **not yet propagated out**.

METHOD:

1. Construct the kernels of the sets of LR(0) items for G . If space is not at a premium, the simplest way is to construct the LR(0) sets of items, as in Section 4.6.2, and then remove the nonkernel items. If space is severely constrained, we may wish instead to store only the kernel items for each set, and compute GOTO for a set of items I by first computing the closure of I .
2. Apply Algorithm 4.62 to the kernel of each set of LR(0) items and grammar symbol X to determine which lookaheads are spontaneously generated for kernel items in $\text{GOTO}(I, X)$, and from which items in I lookaheads are propagated to kernel items in $\text{GOTO}(I, X)$.

3. Initialize a table that gives, for each kernel item in each set of items, the associated lookaheads. Initially, each item has associated with it only those lookaheads that we determined in step (2) were generated spontaneously.
4. Make repeated passes over the kernel items in all sets. When we visit an item i , we look up the kernel items to which i propagates its lookaheads, using information tabulated in step (2). The current set of lookaheads for i is added to those already associated with each of the items to which i propagates its lookaheads. We continue making passes over the kernel items until no more new lookaheads are propagated.

$$\begin{array}{lcl} S & \rightarrow & L = R \mid R \\ L & \rightarrow & *R \mid \text{id} \\ R & \rightarrow & L \end{array}$$

$$\begin{array}{ll} I_0: & S' \rightarrow \cdot S \\ & S \rightarrow \cdot L = R \\ & S \rightarrow \cdot R \\ & L \rightarrow \cdot * R \\ & L \rightarrow \cdot \text{id} \\ & R \rightarrow \cdot L \end{array}$$

$$\begin{array}{ll} I_5: & L \rightarrow \text{id} \cdot \\ I_6: & S \rightarrow L = \cdot R \\ & R \rightarrow \cdot L \\ & L \rightarrow \cdot * R \\ & L \rightarrow \cdot \text{id} \end{array}$$

$$I_1: \quad S' \rightarrow S \cdot$$

$$I_7: \quad L \rightarrow *R \cdot$$

$$\begin{array}{ll} I_2: & S \rightarrow L \cdot = R \\ & R \rightarrow L \cdot \end{array}$$

$$I_8: \quad R \rightarrow L \cdot$$

$$I_3: \quad S \rightarrow R \cdot$$

$$I_9: \quad S \rightarrow L = R \cdot$$

$$\begin{array}{ll} I_4: & L \rightarrow *R \\ & R \rightarrow \cdot L \\ & L \rightarrow \cdot * R \\ & L \rightarrow \text{id} \end{array}$$

$$\begin{array}{rcl}
 S' & \rightarrow & S \\
 S & \rightarrow & L = R \mid R \\
 L & \rightarrow & *R \mid \mathbf{id} \\
 R & \rightarrow & L
 \end{array}$$

$$I_0: \quad S' \rightarrow \cdot S$$

$$I_5: \quad L \rightarrow \mathbf{id} \cdot$$

$$I_1: \quad S' \rightarrow S \cdot$$

$$I_6: \quad S \rightarrow L = \cdot R$$

$$\begin{array}{l}
 I_2: \quad S \xrightarrow{\cdot} L = R \\
 \quad \quad R \rightarrow L \cdot
 \end{array}$$

$$I_7: \quad L \rightarrow *R \cdot$$

$$I_3: \quad S \rightarrow R \cdot$$

$$I_8: \quad R \rightarrow L \cdot$$

$$I_4: \quad L \rightarrow * \cdot R$$

$$I_9: \quad S \rightarrow L = R \cdot$$

$$I_0: \quad S' \rightarrow \cdot S$$

first compute $\text{CLOSURE}(\{[S' \rightarrow \cdot S, \#]\})$, which is

$$\begin{array}{ll} S' \rightarrow \cdot S, \# & L \rightarrow \cdot * R, \#/ = \\ S \rightarrow \cdot L = R, \# & L \rightarrow \cdot \mathbf{id}, \#/ = \\ S \rightarrow \cdot R, \# & R \rightarrow \cdot L, \# \end{array}$$

Among the items in the closure, we see two where the lookahead $=$ has been generated spontaneously. The first of these is $L \rightarrow \cdot * R$. This item, with $*$ to the right of the dot, gives rise to $[L \rightarrow * \cdot R, =]$. That is, $=$ is a spontaneously generated lookahead for $L \rightarrow * \cdot R$, which is in set of items I_4 . Similarly, $[L \rightarrow \cdot \mathbf{id}, =]$ tells us that $=$ is a spontaneously generated lookahead for $L \rightarrow \mathbf{id} \cdot$ in I_5 .

As $\#$ is a lookahead for all six items in the closure, we determine that the item $S' \rightarrow \cdot S$ in I_0 propagates lookaheads to the following six items:

$$\begin{array}{ll} S' \rightarrow S \cdot \text{ in } I_1 & L \rightarrow * \cdot R \text{ in } I_4 \\ S \rightarrow L \cdot = R \text{ in } I_2 & L \rightarrow \mathbf{id} \cdot \text{ in } I_5 \\ S \rightarrow R \cdot \text{ in } I_3 & R \rightarrow L \cdot \text{ in } I_2 \end{array}$$

FROM	TO
$I_0: S' \rightarrow \cdot S$	$I_1: S' \rightarrow S \cdot$
	$I_2: S \rightarrow L \cdot = R$
	$I_2: R \rightarrow L \cdot$
	$I_3: S \rightarrow R \cdot$
	$I_4: L \rightarrow * \cdot R$
	$I_5: L \rightarrow \mathbf{id} \cdot$

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

SET	ITEM	LOOKAHEADS			
		INIT	PASS 1	PASS 2	PASS 3
$I_0:$	$S' \rightarrow \cdot S$	\$	\$	\$	\$
$I_1:$	$S' \rightarrow S \cdot$		\$	\$	\$
$I_2:$	$S \rightarrow L = R$ $R \rightarrow L \cdot$		\$	\$	\$
$I_3:$	$S \rightarrow R \cdot$		\$	\$	\$
$I_4:$	$L \rightarrow * \cdot R$	=	=/\$	=/\$	=/\$
$I_5:$	$L \rightarrow \text{id} \cdot$	=	=/\$	=/\$	=/\$
$I_6:$	$S \rightarrow L = \cdot R$			\$	\$
$I_7:$	$L \rightarrow * R \cdot$		=	=/\$	=/\$
$I_8:$	$R \rightarrow L \cdot$		=	=/\$	=/\$
$I_9:$	$S \rightarrow L = R \cdot$				\$

Using Ambiguous Grammars

Unambiguous grammar

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$$

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

- This grammar is **ambiguous** because it does not specify the **associativity or precedence** of the operators + and *.
- The unambiguous grammar gives + **lower precedence than ***, and makes both operators **left associative**.
- we might **prefer to use the ambiguous grammar**
 - the parser for the unambiguous grammar will spend a substantial fraction of its **time reducing by the productions $E \rightarrow T$ and $T \rightarrow F$** ,
 - whose sole function is to enforce associativity and precedence.
- The parser for the ambiguous grammar **will not waste time reducing** by these single productions (productions whose body consists of a single nonterminal).

$I_0:$ $E' \rightarrow \cdot E$
 $E \rightarrow \cdot E + E$
 $E \rightarrow \cdot E * E$
 $E \rightarrow \cdot(E)$
 $E \rightarrow \cdot \text{id}$

$I_1:$ $E' \rightarrow E \cdot$
 $E \rightarrow E \cdot + E$
 $E \rightarrow E \cdot * E$

$I_2:$ $E \rightarrow (\cdot E)$
 $E \rightarrow \cdot E + E$
 $E \rightarrow \cdot E * E$
 $E \rightarrow \cdot(E)$
 $E \rightarrow \cdot \text{id}$

$I_3:$ $E \rightarrow \text{id} \cdot$

$I_4:$ $E \rightarrow E + \cdot E$
 $E \rightarrow \cdot E + E$
 $E \rightarrow \cdot E * E$
 $E \rightarrow \cdot(E)$
 $E \rightarrow \cdot \text{id}$

$I_5:$ $E \rightarrow E * \cdot E$
 $E \rightarrow \cdot E + E$
 $E \rightarrow \cdot E * E$
 $E \rightarrow \cdot(E)$
 $E \rightarrow \cdot \text{id}$

$I_6:$ $E \rightarrow (E \cdot)$
 $E \rightarrow E \cdot + E$
 $E \rightarrow E \cdot * E$

$I_7:$ $E \rightarrow E + E \cdot$
 $E \rightarrow E \cdot + E$
 $E \rightarrow E \cdot * E$

$I_8:$ $E \rightarrow E * E \cdot$
 $E \rightarrow E \cdot + E$
 $E \rightarrow E \cdot * E$

$I_9:$ $E \rightarrow (E) \cdot$

Conflicts



$\text{Follow}(E) = \{+, *\}$

Conflict resolution

However, these problems can be **resolved using the precedence and associativity** information for + and *.

Consider the input **id + id * id**, which causes a parser to enter state 7 after processing **id + id**;

In particular the parser reaches a configuration

PREFIX	STACK	INPUT
$E + E$	0 1 4 7	* id \$

If * takes precedence over +, the parser **should shift * onto the stack**

Thus the **relative precedence of + followed by *** uniquely determines how the parsing action conflict between **reducing $E \rightarrow E + E$ and shifting on *** in **state 7** should be resolved.

Conflict resolution

Problems can be resolved using the **associativity information** for +.

Consider the input **id + id + id**, which causes a parser to enter state 7 after processing **id + id**;

In particular the parser reaches a configuration

PREFIX	STACK	INPUT
$E + E$	0 1 4 7	+ id \$

associativity of the + operator determines how this conflict should be resolved. If + is left associative, the correct action is to reduce by $E \rightarrow E + E$. That is, the **id** symbols surrounding the first + must be grouped first. Again this choice coincides with what the SLR parser for the unambiguous grammar would do.

Conflict resolution

However, these problems can be resolved using the precedence and associativity information for + and *.

Consider the input **id * id + id**, which causes a parser to enter **state 8** after processing **id * id**;

In particular the parser reaches a configuration

PREFIX	STACK	INPUT
$E * E$	0 1 4 '8	+ id \$

that * is left associative and takes precedence over +, we can argue that state 8, which can appear on top of the stack only when $E * E$ are the top three grammar symbols, should have the action reduce $E \rightarrow E * E$ on both + and * inputs. In the case of input +, the reason is that * takes precedence over +, while in the case of input *, the rationale is that * is left associative.

$I_0:$ $E' \rightarrow \cdot E$
 $E \rightarrow \cdot E + E$
 $E \rightarrow \cdot E * E$
 $E \rightarrow \cdot(E)$
 $E \rightarrow \cdot \text{id}$

$I_1:$ $E' \rightarrow E \cdot$
 $E \rightarrow E \cdot + E$
 $E \rightarrow E \cdot * E$

$I_2:$ $E \rightarrow (\cdot E)$
 $E \rightarrow \cdot E + E$
 $E \rightarrow \cdot E * E$
 $E \rightarrow \cdot(E)$
 $E \rightarrow \cdot \text{id}$

$I_3:$ $E \rightarrow \text{id} \cdot$

$I_4:$ $E \rightarrow E + \cdot E$
 $E \rightarrow \cdot E + E$
 $E \rightarrow \cdot E * E$
 $E \rightarrow \cdot(E)$
 $E \rightarrow \cdot \text{id}$

$I_5:$ $E \rightarrow E * \cdot E$
 $E \rightarrow \cdot E + E$
 $E \rightarrow \cdot E * E$
 $E \rightarrow \cdot(E)$
 $E \rightarrow \cdot \text{id}$

$I_6:$ $E \rightarrow (E \cdot)$
 $E \rightarrow E \cdot + E$
 $E \rightarrow E \cdot * E$

$I_7:$ $E \rightarrow E + E \cdot$
 $E \rightarrow E \cdot + E$
 $E \rightarrow E \cdot * E$

$I_8:$ $E \rightarrow E * E \cdot$
 $E \rightarrow E \cdot + E$
 $E \rightarrow E \cdot * E$

$I_9:$ $E \rightarrow (E) \cdot$

Conflicts



$\text{Follow}(E) = \{+, *\}$

STATE	ACTION					GOTO
	id	+	*	()	
0	s3			s2		1
1		s4	s5			acc
2	s3			s2		6
3		r4	r4		r4	r4
4	s3			s2		7
5	s3			s2		8
6		s4	s5		s9	
7		r1	s5		r1	r1
8		r2	r2		r2	r2
9		r3	r3		r3	r3

Conflict resolution

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

an abstraction of this grammar, where i stands for **if** $expr$ **then**, e stands for **else**, and a stands for “all other productions.” We can then write the grammar, with augmenting production $S' \rightarrow S$, as

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & i \ S \ e \ S \mid i \ S \mid a \end{array} \tag{4.67}$$

$I_0:$	$S' \rightarrow \cdot S$	$I_3:$	$S \rightarrow a\cdot$
	$S \rightarrow \cdot iSeS$		
	$S \rightarrow \cdot iS$	$I_4:$	$S \rightarrow iS \cdot eS \quad S \rightarrow iS \cdot \quad \leftarrow$
	$S \rightarrow \cdot a$	$I_5:$	$S \rightarrow iSe \cdot S$
$I_1:$	$S' \rightarrow S \cdot$		$S \rightarrow \cdot iSeS$
$I_2:$	$S \rightarrow i \cdot SeS$		$S \rightarrow \cdot iS$
	$S \rightarrow i \cdot S$		$S \rightarrow \cdot a$
	$S \rightarrow \cdot iSeS$	$I_6:$	$S \rightarrow iSeS \cdot$
	$S \rightarrow \cdot iS$		
	$S \rightarrow \cdot a$		

guity in (4.67) gives rise to a shift/reduce conflict in I_4 . There, $S \rightarrow iS \cdot eS$ calls for a shift of e and, since $\text{FOLLOW}(S) = \{e, \$\}$, item $S \rightarrow iS \cdot$ calls for reduction by $S \rightarrow iS$ on input e .

if *expr then stmt else stmt.*

STACK

... **if** *expr then stmt*

INPUT

else ... \$

should we shift **else** onto the stack (i.e., shift *e*) or reduce **if expr then stmt** (i.e, reduce by $S \rightarrow iS$)?

The answer is that we should shift **else**, because it is "associated" with the previous **then**.

We conclude that the shift/reduce conflict should be resolved in favor of shift on input **else**

STATE	ACTION				GOTO
	<i>i</i>	<i>e</i>	<i>a</i>	\$	
0	s2		s3		1
1				acc	
2	s2		s3		4
3		r3		r3	
4		s5		r2	
5	s2		s3		6
6		r1		r1	