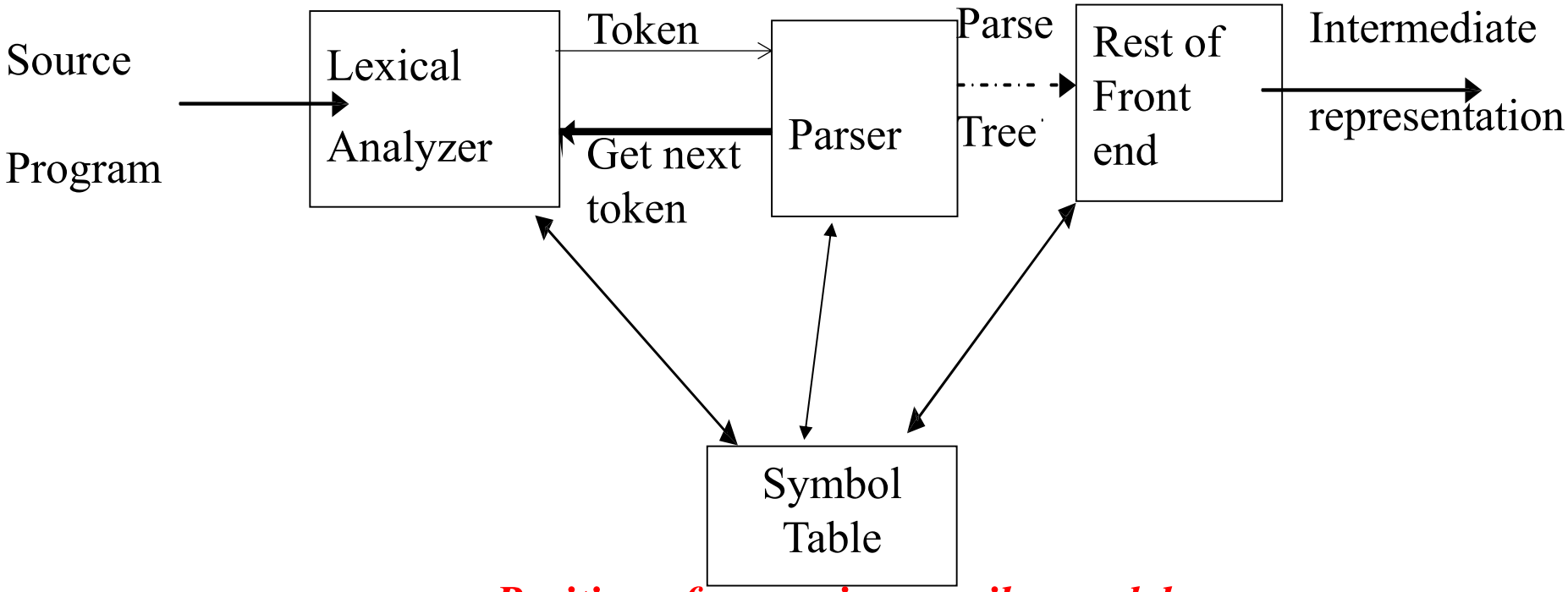
Syntax Analyzer (Parser)
Top Down Parsing

The Role Parser



Position of parser in compiler model

- the parser obtains a string of tokens from the lexical analyzer, as shown in Fig. and verifies that the string of token names can be generated by the grammar for the source language.
- the parser report any syntax errors and to recover from commonly occurring errors to continue processing the remainder of the program.
- for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.

Parsers (cont.)

- We categorize the parsers into two groups:

1. **Top-Down Parser**
   - the parse tree is created top to bottom, starting from the root.
2. **Bottom-Up Parser**
   - the parse is created bottom to top; starting from the leaves

- Both top-down and bottom-up parsers scan the input from left to right (one symbol at a time).
- Efficient top-down and bottom-up parsers can be implemented only for sub-classes of context-free grammars.
  - LL for top-down parsing
  - LR for bottom-up parsing

Left Recursion

- A grammar is ***left recursive*** if it has a non-terminal A such that there is a derivation.

  $A \overset{+}{\Rightarrow} A\alpha$ for some string α

- Top-down parsing techniques **cannot** handle left-recursive grammars.
- So, we have to convert our left-recursive grammar into an equivalent grammar which is not left-recursive.
- The left-recursion may appear in a single step of the derivation (*immediate left-recursion*), or may appear in more than one step of the derivation.

Immediate Left-Recursion

$A \rightarrow A\,\alpha \mid \beta$ where β does not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta\,A'$

$A' \rightarrow \alpha\,A' \mid \varepsilon$ an equivalent grammar

In general,

$A \rightarrow A\,\alpha_1 \mid \ldots \mid A\,\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ where $\beta_1 \ldots \beta_n$ do not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta_1\,A' \mid \ldots \mid \beta_n\,A'$

$A' \rightarrow \alpha_1\,A' \mid \ldots \mid \alpha_m\,A' \mid \varepsilon$ an equivalent grammar

Immediate Left-Recursion -- Example

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id \mid (E)$

$\Downarrow$ eliminate immediate left recursion

$E \rightarrow T\ E'$

$E' \rightarrow +T\ E' \mid \varepsilon$

$T \rightarrow F\ T'$

$T' \rightarrow *F\ T' \mid \varepsilon$

$F \rightarrow id \mid (E)$

Left-Recursion -- Problem

- A grammar cannot be immediately left-recursive, but it still can be left-recursive.
- By just eliminating the immediate left-recursion, we may not get a grammar which is not left-recursive.

$S \rightarrow Aa \mid b$
$A \rightarrow Sc \mid d$ This grammar is not immediately left-recursive, but it is still left-recursive.

$\underline{S} \Rightarrow Aa \Rightarrow \underline{S}ca$ or
$\underline{A} \Rightarrow Sc \Rightarrow \underline{A}ac$ causes to a left-recursion

- So, we have to eliminate all left-recursions from our grammar

Eliminate Left-Recursion -- Algorithm

- Arrange non-terminals in some order: $A_1 \ldots A_n$
- **for** i **from** 1 **to** n **do** {
  - **for** j **from** 1 **to** i-1 **do** {

    replace each production

    $A_i \rightarrow A_j \gamma$

    by

    $A_i \rightarrow \alpha_1 \gamma \mid \ldots \mid \alpha_k \gamma$

    where $A_j \rightarrow \alpha_1 \mid \ldots \mid \alpha_k$

  }
  - eliminate immediate left-recursions among A_i productions

}

Eliminate Left-Recursion -- Example

$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid f$

- Order of non-terminals: S, A

for S:
- we do not enter the inner loop.
- there is no immediate left recursion in S.

for A:
- Replace $A \rightarrow Sd$ with $A \rightarrow Aad \mid bd$
  So, we will have $A \rightarrow Ac \mid Aad \mid bd \mid f$
- Eliminate the immediate left-recursion in A
  $A \rightarrow bdA' \mid fA'$
  $A' \rightarrow cA' \mid adA' \mid \varepsilon$

So, the resulting equivalent grammar which is not left-recursive is:

$S \rightarrow Aa \mid b$
$A \rightarrow bdA' \mid fA'$
$A' \rightarrow cA' \mid adA' \mid \varepsilon$

Eliminate Left-Recursion – Example2

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow G\hat{}F \mid G$

$G \rightarrow id \mid (E)$

Left-Factoring

- A predictive parser (a top-down parser without backtracking) insists that the grammar must be *left-factored*.

  grammar ➔ a new equivalent grammar suitable for predictive parsing

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

- when we see `if`, we cannot know which production rule to choose to re-write *stmt* in the derivation.

Left-Factoring (cont.)

- In general,

  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ where α is non-empty and the first symbols of β_1 and β_2 (if they have one)are different.

- when processing α we cannot know whether expand

  A to $\alpha\beta_1$ or

  A to $\alpha\beta_2$

- But, if we re-write the grammar as follows

  $A \rightarrow \alpha A'$

  $A' \rightarrow \beta_1 \mid \beta_2$ so, we can immediately expand A to $\alpha A'$

Left-Factoring -- Algorithm

- For each non-terminal A with two or more alternatives (production rules) with a common non-empty prefix, let say

$$A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m$$

convert it into

$$A \rightarrow \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_m$$
$$A' \rightarrow \beta_1 \mid \ldots \mid \beta_n$$

Left-Factoring – Example1

$A \rightarrow \underline{a}bB \mid \underline{a}B \mid cdg \mid cdeB \mid cdfB$

$\Downarrow$

$A \rightarrow aA' \mid \underline{cd}g \mid \underline{cd}eB \mid \underline{cd}fB$

$A' \rightarrow bB \mid B$

$\Downarrow$

$A \rightarrow aA' \mid cdA''$

$A' \rightarrow bB \mid B$

$A'' \rightarrow g \mid eB \mid fB$

Left-Factoring – Example2

$A \rightarrow ad \mid a \mid ab \mid abc \mid b$

$\Downarrow$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid b \mid bc$

$\Downarrow$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid bA''$

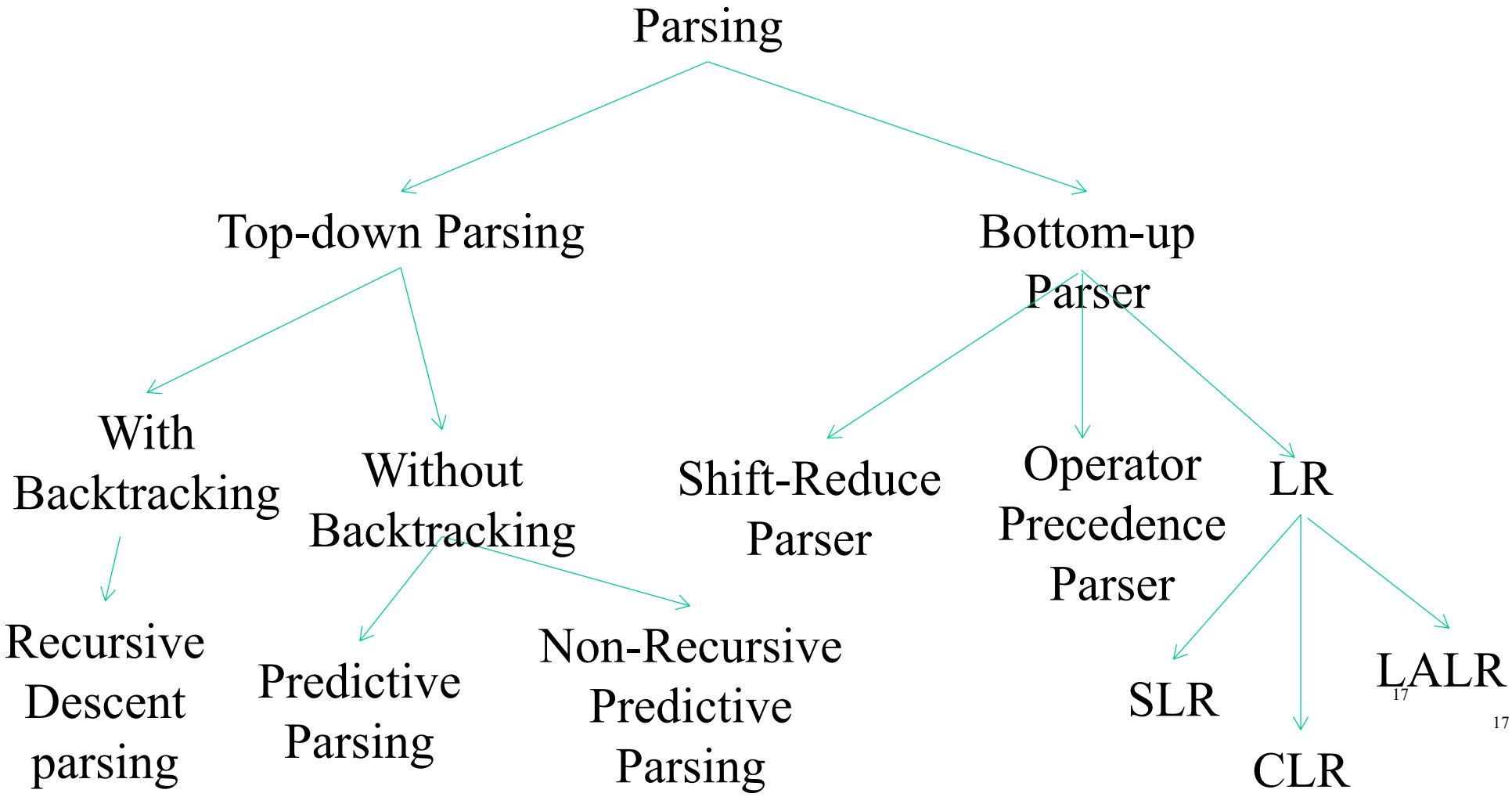
$A'' \rightarrow \varepsilon \mid c$

Left-Factoring – Example3

- $S \rightarrow iEtS \mid iEtSeS \mid a$
- $E \rightarrow b$

$\Downarrow$

- $S \rightarrow iEtSS' \mid a$
- $S' \rightarrow eS \mid \varepsilon$
- $E \rightarrow b$


Parsing
Top-down Parsing
Bottom-up Parser
With Backtracking
Without Backtracking
Shift-Reduce Parser
Operator Precedence Parser
LR
Recursive Descent parsing
Predictive Parsing
Non-Recursive Predictive Parsing
SLR
LALR
CLR

17

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.
- Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.
- Top-down parser
  - Recursive-Descent Parsing
    - Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
    - It is a general parsing technique, but not widely used.
    - Not efficient
  - Predictive Parsing
    - no backtracking
    - efficient
    - needs a special form of grammars (LL(1) grammars).
    - Recursive Predictive Parsing is a special form of Recursive Descent parsing without backtracking.
    - Non-Recursive (Table Driven) Predictive Parser is also known as LL(1) parser.

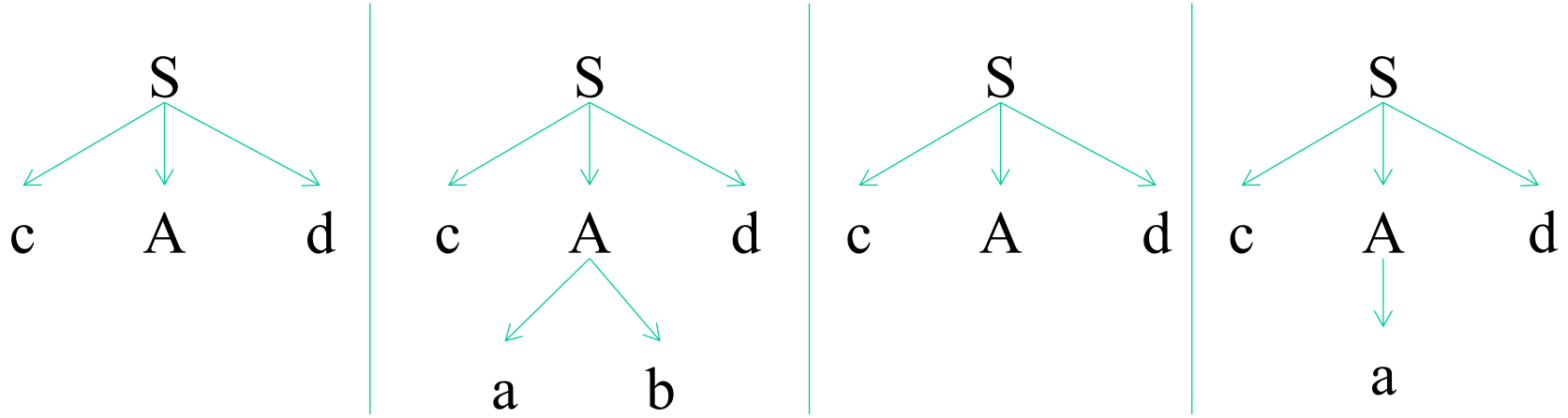
Recursive-Descent Parsing (uses Backtracking)

- Backtracking is needed.
- It tries to find the left-most derivation.

Consider the grammar G : S → cAd

A → ab | a and the input string w=cad.

The parse tree can be constructed using the following top-down approach :



Step : 1 Step : 2 Step : 3 Step : 4

- To construct a parse tree top-down for the input string w = cad, begin with a tree consisting of a single node labeled S, and the input pointer pointing to c, the first symbol of w. S has only one production, so we use it to expand S and obtain the tree (step1).
- The leftmost leaf, labeled c, matches the first symbol of input w, so we advance the input pointer to a, the second symbol of w, and consider the next leaf, labeled A.(step2)
- We have a match for the second input symbol, a, so we advance the input pointer to d, the third input symbol, and compare d against the next leaf, labeled b.
- Since b does not match d, we report failure and go back to A to see whether there is another alternative for A that has not been tried, but that might produce a match.
- In going back to A, we must reset the input pointer to position 2 .(step 3)
- The second alternative for A produces the tree of Fig. The leaf a matches the second symbol of w and the leaf d matches the third symbol.(step 4)
- Since we have produced a parse tree for w, we halt and announce successful completion of parsing.

- A recursive-descent parsing program consists of a set of procedures, one for each nonterminal.
- Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string.

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

- Each non-terminal corresponds to a procedure.

Ex: $A \rightarrow aBb$ (This is only the production rule for A)

```
proc A {
    - match the current token with a, and move to the next token;
    - call 'B';
    - match the current token with b, and move to the next token;
}
```

OR

```
Proc A()
{
if(input_token=='a')
{
      -move to the next token(input_token++)
      -call B()
      if(input_token=='b')
       {
         -move to the next token(input_token++)
        }
      else /* an error has occurred * /
}
}
```

A → aBb | bAB

```
Proc A()
{
if(input_token==‘a')
{
      -move to the next token(input_token++)
      -call B()
      if(input_token==‘b’)
       {
         -move to the next token(input_token++)
        }
      else /* an error has occurred * /
} else if(input_token==‘b')
{
      -call A()
      -call B()
}
}
```

Write the procedure for the non-terminals of the following grammar to make recursive descent parsing

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id \mid (E)$

⇓ eliminate immediate left recursion

$E \rightarrow T\ E'$

$E' \rightarrow +T\ E' \mid \varepsilon$

$T \rightarrow F\ T'$

$T' \rightarrow *F\ T' \mid \varepsilon$

$F \rightarrow id \mid (E)$

$E \rightarrow T\ E'$

```
proc E() {
        - call T();
        - call E'();

    }
```

$E' \rightarrow +T\ E' \mid \varepsilon$

```
Proc E'()
{
if(input_token=='+')
{
    -move to the next token(input_token++)
    -call T()
    -call E'()
}
else
{
        return
}
}
```

$T \rightarrow F\ T'$

```
proc T {
        - call F();
        - call T'();

    }
```

$T' \rightarrow *F\ T' \mid \varepsilon$

```
Proc T'()
{
if(input_token=='*')
{
-move to the next token(input_token++)
-call F()
-call T'()
}
else
{
return
}
}
```

$F \rightarrow id \mid (E)$

```
Proc F()
{
if(input_token=='id')
{
-move to the next token(input_token++)
}
else
if(input_token=='(')
{
-move to the next token(input_token++)
-call E()
if(input_token==')'){
-move to the next token(input_token++)
}else /* an error has occurred * /
}
}
```

Write the procedure for the non-terminals of the following grammars to make recursive descent parsing

a) $E \rightarrow E+T \mid T$

$T \rightarrow TF \mid F$

$F \rightarrow F^* \mid a \mid b$

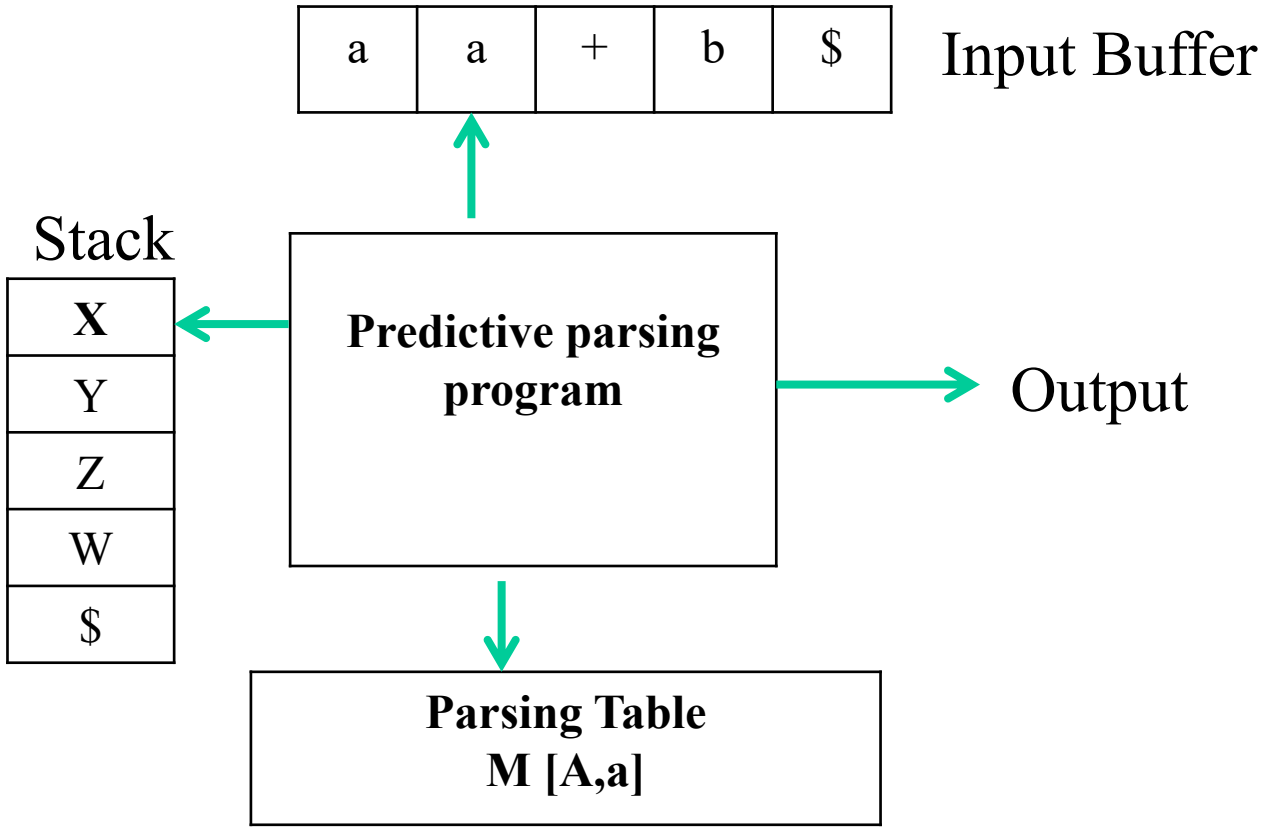
b) $E \rightarrow E+T \mid T$

$T \rightarrow V*F \mid V$

$V \rightarrow id$

Non-Recursive Predictive Parsing -- LL(1) Parser

- Non-Recursive predictive parsing is a table-driven parser.
- It is a top-down parser.
- It is also known as LL(1) Parser



LL(1) Parser

input buffer

- our string to be parsed. We will assume that its end is marked with a special symbol $.

output

- a production rule representing a step of the derivation sequence (left-most derivation) of the string in the input buffer.

stack

- contains the grammar symbols
- at the bottom of the stack, there is a special end marker symbol $.
- initially the stack contains only the symbol $ and the starting symbol S.

  $S ← initial stack
- when the stack is emptied (ie. only $ left in the stack), the parsing is completed.

parsing table

- a two-dimensional array M[A,a]
- each row is a non-terminal symbol
- each column is a terminal symbol or the special symbol $
- each entry holds a production rule.

LL(1) Parser – Parser Actions

- The symbol at the top of the stack (say **X**) and the current symbol in the input string (say **a**) determine the parser action.
- There are four possible parser actions.

1. If **X** and **a** are $ ➔ parser halts (successful completion)

2. If **X** and **a** are the same terminal symbol (different from $)

   ➔ parser pops X from the stack, and moves the next symbol in the input buffer.

3. If **X** is a non-terminal

   ➔ parser looks at the parsing table entry M[X,a]. If M[X,a] holds a production rule $X \rightarrow Y_1 Y_2 ... Y_k$, it pops X from the stack and pushes $Y_k, Y_{k-1}, ..., Y_1$ into the stack. The parser also outputs the production rule $X \rightarrow Y_1 Y_2 ... Y_k$ to represent a step of the derivation.

4. none of the above ➔ error
   - all empty entries in the parsing table are errors.
   - If X is a terminal symbol different from a, this is also an error case.

LL(1) Parser – Example1

$S \rightarrow aBa$

$B \rightarrow bB \mid \varepsilon$

	a	b	$
S	$S \rightarrow aBa$		
B	$B \rightarrow \varepsilon$	$B \rightarrow bB$	

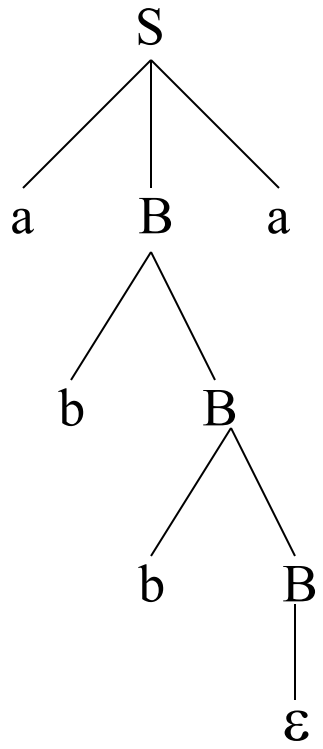
LL(1) Parsing Table

stack	**input**	**output**
$S	abba$	$S \rightarrow aBa$
$aBa	abba$	
$aB	bba$	$B \rightarrow bB$
$aBb	bba$	
$aB	ba$	$B \rightarrow bB$
$aBb	ba$	
$aB	a$	$B \rightarrow \varepsilon$
$a	a$	
$	$	accept, successful completion

LL(1) Parser – Example1 (cont.)

Outputs: $S \rightarrow aBa$ $B \rightarrow bB$ $B \rightarrow bB$ $B \rightarrow \varepsilon$

Derivation(left-most): $S \Rightarrow aBa \Rightarrow abBa \Rightarrow abbBa \Rightarrow abba$



parse tree

LL(1) Parser – Example2

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$

	id	+	*	(	)	**$**
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

LL(1) Parser – Example2

stack	input	output
$E	id+id$	$E \rightarrow TE'$
$E'T	id+id$	$T \rightarrow FT'$
$E'T'F	id+id$	$F \rightarrow id$
$ E'T'id	id+id$	
$ E'T'	+id$	$T' \rightarrow \varepsilon$
$ E'	+id$	$E' \rightarrow +TE'$
$ E'T+	+id$	
$ E'T	id$	$T \rightarrow FT'$
$ E'T'F	id$	$F \rightarrow id$
$ E'T'id	id$	
$ E'T'	$	$T' \rightarrow \varepsilon$
$ E'	$	$E' \rightarrow \varepsilon$
$	$	accept

Constructing LL(1) Parsing Tables

- Two functions are used in the construction of LL(1) parsing tables:
  - FIRST FOLLOW

- **FIRST(α)** is a set of the terminal symbols which occur as first symbols in strings derived from α where α is any string of grammar symbols.
- if α derives to ε, then ε is also in FIRST(α) .

- **FOLLOW(A)** is the set of the terminals which occur immediately after (follow) the *non-terminal A* in the strings derived from the starting symbol.
  - a terminal a is in FOLLOW(A) if $S \stackrel{*}{\Rightarrow} \alpha A a \beta$
  - $ is in FOLLOW(A) if $S \stackrel{*}{\Rightarrow} \alpha A$

Compute FIRST for Any String X

Rules for first( ):

1. If *X* is terminal, then FIRST(*X*) is {X}.
2. If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(*X*).
3. If *X* is non-terminal and $X \rightarrow a\alpha$ is a production then add *a* to FIRST(X).
4. If X is non-terminal and $X \rightarrow Y1\ \ Y2 \ldots Yk$ is a production, then place *a* in FIRST(*X*) if for some *i*, *a* is in FIRST(*Yi*), and ε is in all of FIRST(*Y1*),…,FIRST(*Yi-1*);

   If ε is in FIRST(*Yj*) for all j=1,2,..,k, then add ε to FIRST(*X*).

FIRST Example

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$

FIRST(E) = FIRST(T)= FIRST(F)= { (,id}

FIRST(E′) = {+, ε}

FIRST(T′) = {*, ε}

FIRST(TE′) = { (,id}
FIRST(+TE′) = {+}
FIRST(ε) = {ε}
FIRST(FT′) = { (,id}
FIRST(*FT′) = {*}
FIRST(ε) = {ε}
FIRST((E)) = { (}
FIRST(id) = {id}

Compute FOLLOW (for non-terminals)

- If S is the start symbol ➔ $ is in FOLLOW(S)

- if $A \rightarrow \alpha B \beta$ is a production rule
  ➔ everything in FIRST(β) is FOLLOW(B) except ε

- If ($A \rightarrow \alpha B$ is a production rule) or
  ($A \rightarrow \alpha B \beta$ is a production rule and ε is in FIRST(β))
  ➔ everything in FOLLOW(A) is in FOLLOW(B).

We apply these rules until nothing more can be added to any follow set.

FOLLOW Example

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

FOLLOW(E) = { $,) }

FOLLOW(E') = { $,) }

FOLLOW(T) = { +,), $ }

FOLLOW(T') = { +,), $ }

FOLLOW(F) = {+, *,), $ }

$S \rightarrow (L) \mid a$
$L \rightarrow L,S \mid S$

$\Downarrow$

$S \rightarrow (L) \mid a$
$L \rightarrow SL'$
$L' \rightarrow ,SL' \mid \varepsilon$

FIRST(S)={(,a}
FIRST(L)=FIRST(S)={(,a}

FIRST(L')**{,** , ε}

FOLLOW(S)={$,), **,**}
FOLLOW(L)={)}
FOLLOW(L')={)}

$E \rightarrow E+T \mid T$

$T \rightarrow TF \mid F$

$F \rightarrow F^* \mid a \mid b$

$S \rightarrow a \mid \hat{} \mid (T)$

$T \rightarrow T,S \mid S$

$S \rightarrow ABCDE$

$A \rightarrow a \mid \epsilon$

$B \rightarrow b \mid \epsilon$

$C \rightarrow C \mid \epsilon$

$D \rightarrow d \mid \epsilon$

$E \rightarrow e \mid \epsilon$

Constructing LL(1) Parsing Table -- Algorithm

- for each production rule $A \rightarrow \alpha$ of a grammar G
  - for each terminal a in FIRST(α)
    ➔ add $A \rightarrow \alpha$ to M[A,a]
  - If ε in FIRST(α)
    ➔ for each terminal a in FOLLOW(A) add $A \rightarrow \alpha$ to M[A,a]
  - If ε in FIRST(α) and \$ in FOLLOW(A)
    ➔ add $A \rightarrow \alpha$ to M[A,\$]

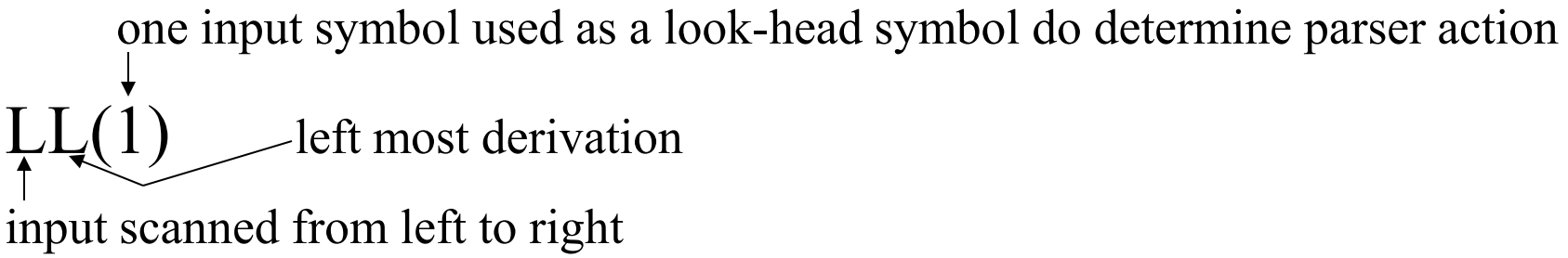
- All other undefined entries of the parsing table are error entries.

Constructing LL(1) Parsing Table -- Example

$E \rightarrow TE'$	FIRST(TE')={(,id}	➔ $E \rightarrow TE'$ into M[E,(] and M[E,id]
$E' \rightarrow +TE'$	FIRST($+TE'$)={+}	➔ $E' \rightarrow +TE'$ into M[E',+]
$E' \rightarrow \varepsilon$	FIRST(ε)={ε}	➔ none
	but since ε in FIRST(ε) and FOLLOW(E')={$,)}	➔ $E' \rightarrow \varepsilon$ into M[E',$] and M[E',)]
$T \rightarrow FT'$	FIRST(FT')={(,id}	➔ $T \rightarrow FT'$ into M[T,(] and M[T,id]
$T' \rightarrow *FT'$	FIRST($*FT'$)={*}	➔ $T' \rightarrow *FT'$ into M[T',*]
$T' \rightarrow \varepsilon$	FIRST(ε)={ε}	➔ none
	but since ε in FIRST(ε) and FOLLOW(T')={$,),+}	➔ $T' \rightarrow \varepsilon$ into M[T',$], M[T',)] and M[T',+]
$F \rightarrow (E)$	FIRST((E))={(}	➔ $F \rightarrow (E)$ into M[F,(]
$F \rightarrow id$	FIRST(id)={id}	➔ $F \rightarrow id$ into M[F,id]

LL(1) Grammars

- A grammar whose parsing table has no multiply-defined entries is said to be LL(1) grammar.



- The parsing table of a grammar may contain more than one production rule. In this case, we say that it is not a LL(1) grammar.

A Grammar which is not LL(1)

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

$E \rightarrow e\ S \mid \varepsilon$

$C \rightarrow b$

FOLLOW(S) = { $,e }

FOLLOW(E) = { $,e }

FOLLOW(C) = { t }

FIRST(iCtSE) = {i}

FIRST(a) = {a}

FIRST(eS) = {e}

FIRST(ε) = {ε}

FIRST(b) = {b}

	a	b	e	i	t	$
S	$S \rightarrow a$			$S \rightarrow iCtSE$		
E			$E \rightarrow eS$ $E \rightarrow \varepsilon$			$E \rightarrow \varepsilon$
C		$C \rightarrow b$				

two production rules for M[E,e]

Problem ➔ ambiguity

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

$E \rightarrow e\ S \mid \varepsilon$

$C \rightarrow b$

A Grammar which is not LL(1) (cont.)

- What do we have to do it if the resulting parsing table contains multiply defined entries?
  - If we didn't eliminate left recursion, eliminate the left recursion in the grammar.
  - If the grammar is not left factored, we have to left factor the grammar.
  - If its (new grammar's) parsing table still contains multiply defined entries, that grammar is ambiguous or it is inherently not a LL(1) grammar.
- A left recursive grammar cannot be a LL(1) grammar.
  - $A \rightarrow A\alpha \mid \beta$
    - ➔ any terminal that appears in FIRST(β) also appears FIRST($A\alpha$) because $A\alpha \Rightarrow \beta\alpha$.
    - ➔ If β is ε, any terminal that appears in FIRST(α) also appears in FIRST($A\alpha$) and FOLLOW(A).
- A grammar is not left factored, it cannot be a LL(1) grammar
  - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
    - ➔ any terminal that appears in FIRST($\alpha\beta_1$) also appears in FIRST($\alpha\beta_2$).
- An ambiguous grammar cannot be a LL(1) grammar.

Properties of LL(1) Grammars

- A grammar G is LL(1) if and only if the following conditions hold for two distinctive production rules $A \to \alpha$ and $A \to \beta$

  1. Both α and β cannot derive strings starting with same terminals.

  2. At most one of α and β can derive to ε.

  3. If β can derive to ε, then α cannot derive to any string starting with a terminal in FOLLOW(A).

Bottom-Up Parsing

- A **bottom-up parser** creates the parse tree of the given input starting from leaves towards the root.
- A bottom-up parser tries to find the right-most derivation of the given input in the reverse order.

  $S \Rightarrow ... \Rightarrow \omega$ (the right-most derivation of ω)

  $\leftarrow$ (the bottom-up parser finds the right-most derivation in the reverse order)
- Bottom-up parsing is also known as **shift-reduce parsing** because its two main actions are shift and reduce.
  - At each shift action, the current symbol in the input string is pushed to a stack.
  - At each reduction step, the symbols at the top of the stack (this symbol sequence is the right side of a production) will replaced by the non-terminal at the left side of that production.
  - There are also two more actions: accept and error.

Shift-Reduce Parsing

- A shift-reduce parser tries to reduce the given input string into the starting symbol.

  a string ➔ the starting symbol

  reduced to

- At each reduction step, a substring of the input matching to the right side of a production rule is replaced by the non-terminal at the left side of that production rule.
- If the substring is chosen correctly, the right most derivation of that string is created in the reverse order.

Rightmost Derivation: $S \underset{rm}{\overset{*}{\Rightarrow}} \omega$

Shift-Reduce Parser finds: $\omega \underset{rm}{\Leftarrow} \ldots \underset{rm}{\Leftarrow} S$

Shift-Reduce Parsing -- Example

$S \rightarrow aABb$

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid b$

input string: aaabb

aaAbb

aAbb

aABb

S

$\Downarrow$ reduction

$S \underset{rm}{\Rightarrow} aABb \underset{rm}{\Rightarrow} aAbb \underset{rm}{\Rightarrow} aaAbb \underset{rm}{\Rightarrow} aaabb$

Right Sentential Forms

- How do we know which substring to be replaced at each reduction step?

Handle

- Informally, a **handle** of a string is a substring that matches the right side of a production rule.
  - But not every substring matches the right side of a production rule is handle

- A **handle** of a right sentential form $\gamma \ (\equiv \alpha\beta\omega)$ is

  a production rule $A \rightarrow \beta$ and a position of γ where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ.

$$S \underset{rm}{\overset{*}{\Rightarrow}} \alpha A \omega \underset{rm}{\Rightarrow} \alpha\beta\omega$$

- If the grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.
- We will see that ω is a string of terminals.

Handle Pruning

- A right-most derivation in reverse can be obtained by **handle-pruning**.

$$S=\gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = \omega$$

input string

- Start from γ_n, find a handle $A_n \rightarrow \beta_n$ in γ_n, and replace β_n in by A_n to get γ_{n-1}.
- Then find a handle $A_{n-1} \rightarrow \beta_{n-1}$ in γ_{n-1}, and replace β_{n-1} in by A_{n-1} to get γ_{n-2}.
- Repeat this, until we reach S.

A Shift-Reduce Parser

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Right-Most Derivation of `id+id*id`

$E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*id \Rightarrow E+F*id$

$\Rightarrow E+id*id \Rightarrow T+id*id \Rightarrow F+id*id \Rightarrow id+id*id$

Right-Most Sentential Form	Reducing Production
`id`+id*id	$F \rightarrow id$
`F`+id*id	$T \rightarrow F$
`T`+id*id	$E \rightarrow T$
E+`id`*id	$F \rightarrow id$
E+`F`*id	$T \rightarrow F$
E+T*`id`	$F \rightarrow id$
E+`T*F`	$T \rightarrow T*F$
`E+T`	$E \rightarrow E+T$
E	

`Handles` are red and underlined in the right-sentential forms.

A Stack Implementation of A Shift-Reduce Parser

- There are four possible actions of a shift-reduce parser:

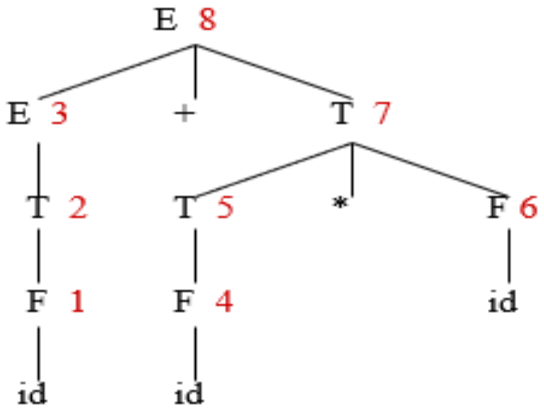
  1. **Shift** : The next input symbol is shifted onto the top of the stack.
  2. **Reduce**: Replace the handle on the top of the stack by the non-terminal.
  3. **Accept**: Successful completion of parsing.
  4. **Error**: Parser discovers a syntax error, and calls an error recovery routine.

- Initial stack just contains only the end-marker $.
- The end of the input string is marked by the end-marker $.

A Stack Implementation of A Shift-Reduce Parser

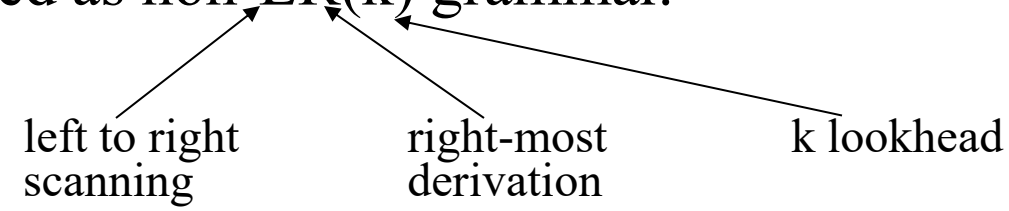
Stack	Input	Action
$	id+id*id$	shift
$id	+id*id$	reduce by F → id
$F	+id*id$	reduce by T → F
$T	+id*id$	reduce by E → T
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	reduce by F → id
$E+F	*id$	reduce by T → F
$E+T	*id$	shift
$E+T*	id$	shift
$E+T*id	$	reduce by F → id
$E+T*F	$	reduce by T → T*F
$E+T	$	reduce by E → E+T
$E	$	accept

Parse Tree



Conflicts During Shift-Reduce Parsing

- There are context-free grammars for which shift-reduce parsers cannot be used.
- Stack contents and the next input symbol may not decide action:
  - **shift/reduce conflict**: Whether make a shift operation or a reduction.
  - **reduce/reduce conflict**: The parser cannot decide which of several reductions to make.
- If a shift-reduce parser cannot be used for a grammar, that grammar is called as non-LR(k) grammar.

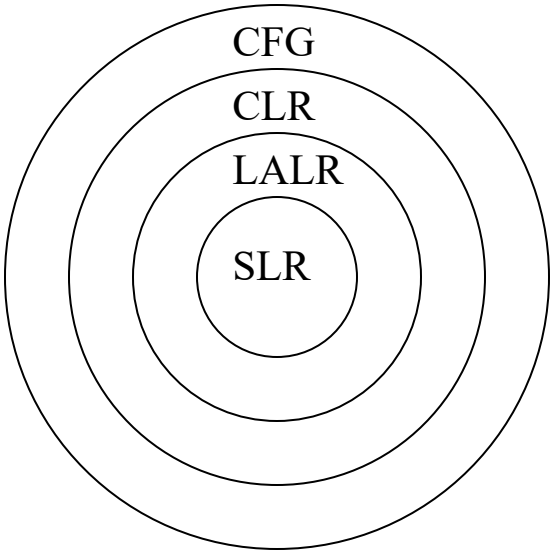


- An ambiguous grammar can never be a LR grammar.

Shift-Reduce Parsers

- There are two main categories of shift-reduce parsers

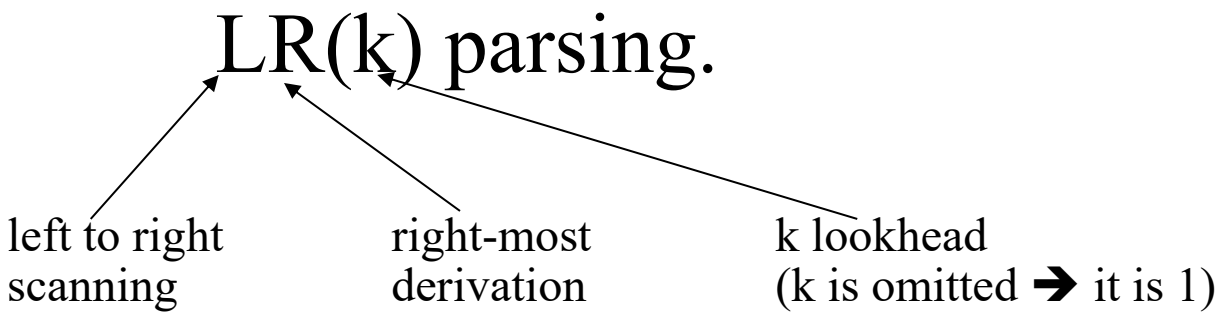
1. **Operator-Precedence Parser**
   - simple, but only a small class of grammars.



2. **LR-Parsers**
   - covers wide range of grammars.
     - SLR – simple LR parser
     - CLR – Canonical LR parser
     - LALR –lookahead LR parser
   - SLR, LR and LALR working principle is same, only their parsing tables are different.

LR Parsers

- The most powerful shift-reduce parsing is:



- LR parsing is attractive because:
  - LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
  - The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

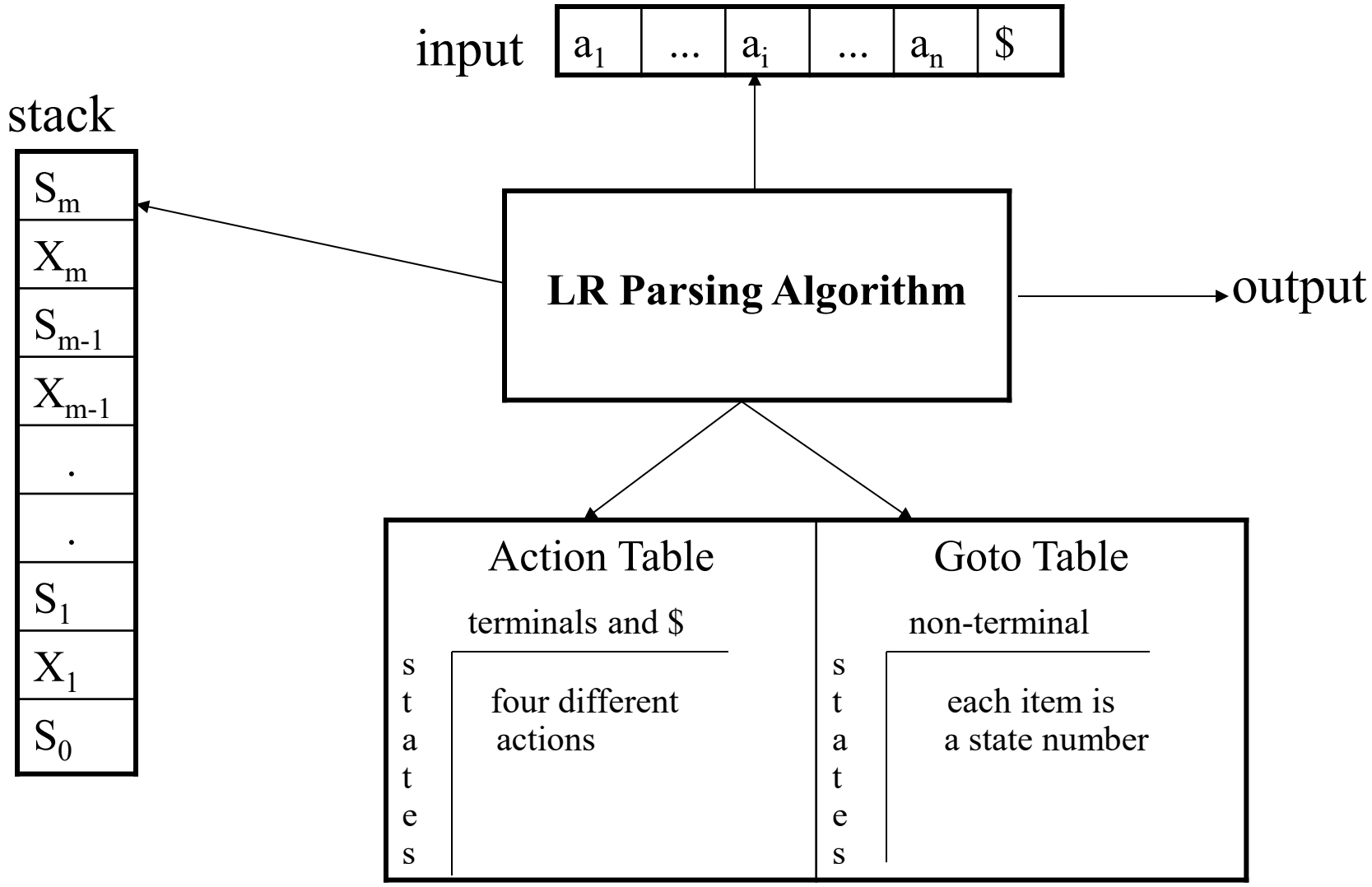
$$\text{LL(1)-Grammars} \subset \text{LR(1)-Grammars}$$

  - An LR-parser can detect a syntactic error as soon as it is possible to do so a left-to-right scan of the input.

LR Parsers

- **LR-Parsers**
  - covers wide range of grammars.
  - SLR – simple LR parser
  - CLR – Canoical LR parser
  - LALR – intermediate LR parser (look-head LR parser)
  - SLR, CLR and LALR work same (they used the same algorithm), only their parsing tables are different.

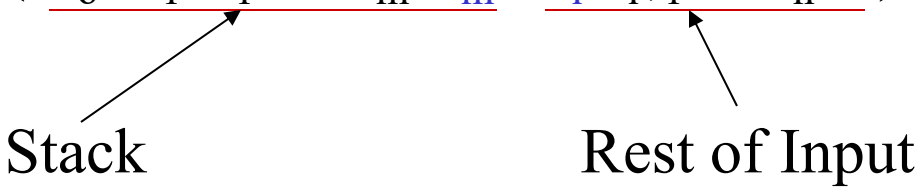
LR Parsing Algorithm



A Configuration of LR Parsing Algorithm

- A configuration of a LR parsing is:

$$(S_o\ X_1\ S_1 \ldots X_m\ S_m,\ \ a_i\ a_{i+1} \ldots a_n\ \$)$$



- S_m and a_i decides the parser action by consulting the parsing action table. (*Initial Stack* contains just S_o)

- A configuration of a LR parsing represents the right sentential form:

$$X_1 \ldots X_m\ a_i\ a_{i+1} \ldots a_n\ \$$$

Actions of A LR-Parser

1. **shift s** -- shifts the next input symbol and the state **s** onto the stack

   ($S_o X_1 S_1 \ldots X_m S_m, a_i a_{i+1} \ldots a_n \$$) ➔ ($S_o X_1 S_1 \ldots X_m S_m a_i s, a_{i+1} \ldots a_n \$$)

2. **reduce A→β** (or **`rn`** where n is a production number)
   - pop 2|**β**| (=r) items from the stack;
   - then push **A** and **s** where **s=goto[s_{m-r},A]**

   ($S_o X_1 S_1 \ldots X_m S_m, a_i a_{i+1} \ldots a_n \$$) ➔ ($S_o X_1 S_1 \ldots X_{m-r} S_{m-r} A s, a_i \ldots a_n \$$)

   - Output is the reducing production reduce A→β

3. **Accept** – Parsing successfully completed

4. **Error** -- Parser detected an error (an empty entry in the action table)

(SLR) Parsing Tables for Expression Grammar

1) $E \rightarrow E+T$
2) $E \rightarrow T$
3) $T \rightarrow T*F$
4) $T \rightarrow F$
5) $F \rightarrow (E)$
6) $F \rightarrow id$

	Action Table							Goto Table		
state	**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				

Actions of A (S)LR-Parser -- Example

stack	input	action	output
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3	*id+id$	reduce by T→F	T→F
0T2	*id+id$	shift 7	
0T2*7	id+id$	shift 5	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10	+id$	reduce by T→T*F	T→T*F
0T2	+id$	reduce by E→T	E→T
0E1	+id$	shift 6	
0E1+6	id$	shift 5	
0E1+6id5	$	reduce by F→id	F→id
0E1+6F3	$	reduce by T→F	T→F
0E1+6T9	$	reduce by E→E+T	E→E+T
0E1	$	accept	

Constructing SLR Parsing Tables – LR(0) Item

- An **LR(0) item** of a grammar G is a production of G and dot at some position of the right side.
- Ex: $A \rightarrow aBb$ Possible LR(0) Items: (four different possibility)

  $A \rightarrow \bullet aBb$
  $A \rightarrow a \bullet Bb$
  $A \rightarrow aB \bullet b$
  $A \rightarrow aBb \bullet$
- Sets of LR(0) items will be the states of action and goto table of the SLR parser.
- A collection of sets of LR(0) items (**the canonical LR(0) collection**) is the basis for constructing SLR parsers.
- *Augmented Grammar*:

  Augmented Grammar G' is G with a new production rule $S' \rightarrow S$ where S' is the new starting symbol.

The Closure Operation

- If ***I*** is a set of LR(0) items for a grammar G, then ***closure(I)*** is the set of LR(0) items constructed from I by the two rules:

  1. Initially, every LR(0) item in I is added to closure(I).
  2. If $A \rightarrow \alpha \bullet B\beta$ is in closure(I) and $B \rightarrow \gamma$ is a production rule of G; then $B \rightarrow \bullet \gamma$ will be in the closure(I).
     We will apply this rule until no more new LR(0) items can be added to closure(I).

The Closure Operation -- Example

E' → E

E → E+T

E → T

T → T*F

T → F

F → (E)

F → id

closure({E' → •E}) =

{ E' → •E ← kernel items

E → •E+T

E → •T

T → •T*F

T → •F

F → •(E)

F → •id }

Goto Operation

- If I is a set of LR(0) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:
  - If $A \rightarrow \alpha \bullet X\beta$ in I
    then every item in **closure({$A \rightarrow \alpha X \bullet \beta$})** will be in goto(I,X).

Example:

I ={ $E' \rightarrow \bullet E$, $E \rightarrow \bullet E+T$, $E \rightarrow \bullet T$,
$T \rightarrow \bullet T*F$, $T \rightarrow \bullet F$,
$F \rightarrow \bullet (E)$, $F \rightarrow \bullet id$ }

goto(I,E) = { $E' \rightarrow E \bullet$, $E \rightarrow E \bullet +T$ }

goto(I,T) = { $E \rightarrow T \bullet$, $T \rightarrow T \bullet *F$ }

goto(I,F) = {$T \rightarrow F \bullet$ }

goto(I,() = { $F \rightarrow (\bullet E)$, $E \rightarrow \bullet E+T$, $E \rightarrow \bullet T$, $T \rightarrow \bullet T*F$, $T \rightarrow \bullet F$,
$F \rightarrow \bullet (E)$, $F \rightarrow \bullet id$ }

goto(I,id) = { $F \rightarrow id \bullet$ }

Construction of The Canonical LR(0) Collection

- To create the SLR parsing tables for a grammar G, we will create the canonical LR(0) collection of the grammar G'.

- ***Algorithm***:

  C is { closure({S'$\rightarrow \bullet$ S}) }

  repeat the followings until no more set of LR(0) items can be added to ***C***.

  for each I in ***C*** and each grammar symbol X

  if goto(I,X) is not empty and not in ***C***

  add goto(I,X) to ***C***

- goto function is a DFA on the sets in C.

The Canonical LR(0) Collection -- Example

I_0: E' → .E
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I_1: E' → E.
E → E.+T

I_2: E → T.
T → T.*F

I_3: T → F.

I_4: F → (.E)
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I_5: F → id.

I_6: E → E+.T
T → .T*F
T → .F
F → .(E)
F → .id

I_7: T → T*.F
F → .(E)
F → .id

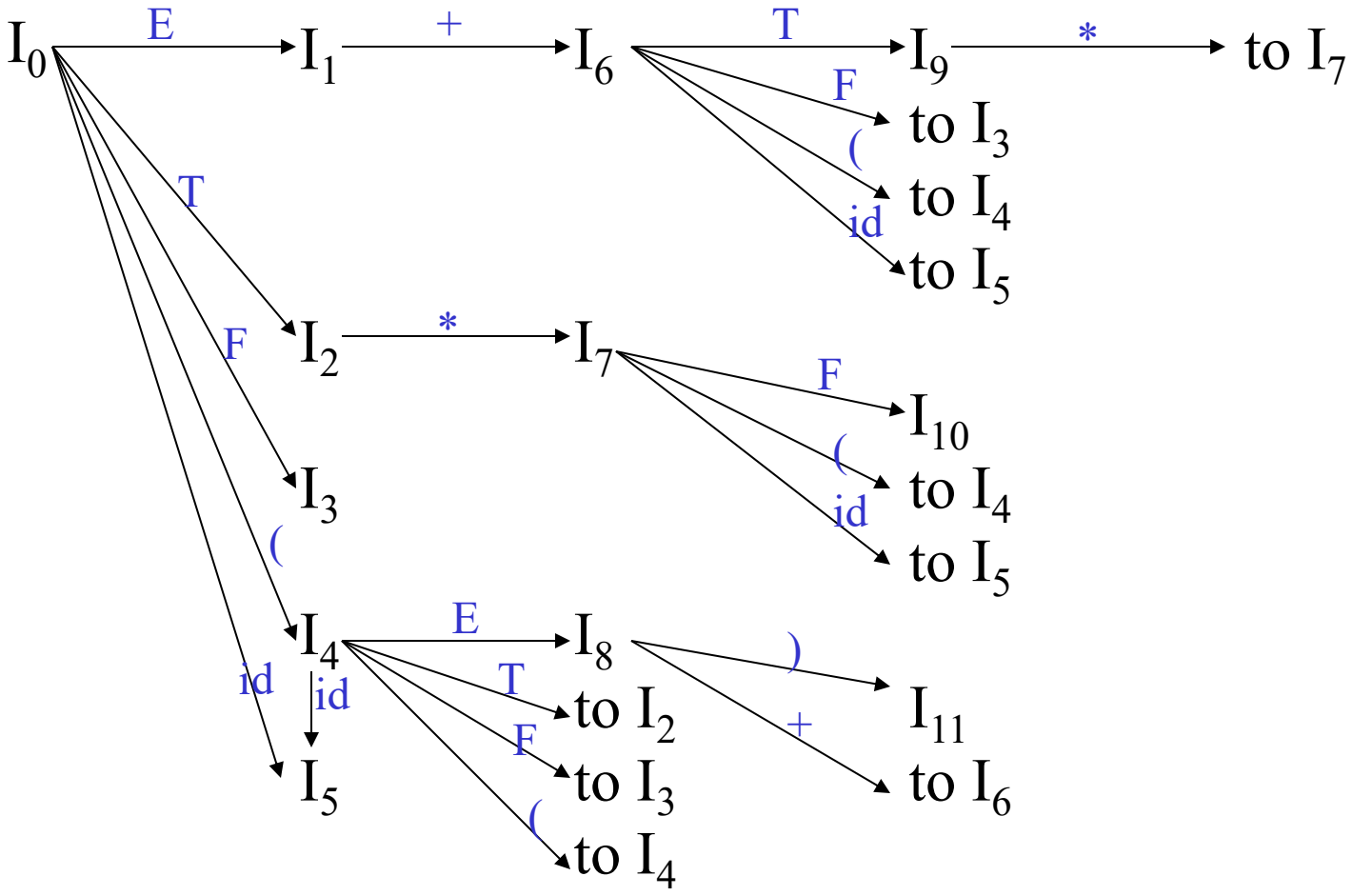
I_8: F → (E.)
E → E.+T

I_9: E → E+T.
T → T.*F

I_{10}: T → T*F.

I_{11}: F → (E).

Transition Diagram (DFA) of Goto Function



Constructing SLR Parsing Table

(of an augumented grammar G')

1. Construct the canonical collection of sets of LR(0) items for G'.
   $C \leftarrow \{I_0,...,I_n\}$
2. Create the parsing action table as follows
   - If a is a terminal, $A \rightarrow \alpha.a\beta$ in I_i and $goto(I_i,a)=I_j$ then action[i,a] is ***shift j.***
   - If $A \rightarrow \alpha.$ is in I_i, then action[i,a] is ***reduce*** $A \rightarrow \alpha$ for all a in FOLLOW(A) where $A \neq S'$.
   - If $S' \rightarrow S.$ is in I_i, then action[i,$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not SLR(1).
3. Create the parsing goto table
   - for all non-terminals A, if $goto(I_i,A)=I_j$ then goto[i,A]=j
4. All entries not defined by (2) and (3) are errors.
5. Initial state of the parser contains $S' \rightarrow .S$

Parsing Tables of Expression Grammar

	Action Table							Goto Table		
state	**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				

SLR(1) Grammar

- An LR parser using SLR(1) parsing tables for a grammar G is called as the SLR(1) parser for G.
- If a grammar G has an SLR(1) parsing table, it is called SLR(1) grammar (or SLR grammar in short).
- Every SLR grammar is unambiguous, but every unambiguous grammar is not a SLR grammar.

shift/reduce and reduce/reduce conflicts

- If a state does not know whether it will make a shift operation or reduction for a terminal, we say that there is a **shift/reduce conflict**.

- If a state does not know whether it will make a reduction operation using the production rule `i` or `j` for a terminal, we say that there is a **reduce/reduce conflict**.

- If the SLR parsing table of a grammar G has a conflict, we say that that grammar is not SLR grammar.

Conflict Example

$S \rightarrow L=R$
$S \rightarrow R$
$L \rightarrow *R$
$L \rightarrow id$
$R \rightarrow L$

I_0: $S' \rightarrow .S$
$S \rightarrow .L=R$
$S \rightarrow .R$
$L \rightarrow .*R$
$L \rightarrow .id$
$R \rightarrow .L$

I_1: $S' \rightarrow S.$

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

I_3: $S \rightarrow R.$

I_4: $L \rightarrow *.R$
$R \rightarrow .L$
$L \rightarrow .*R$
$L \rightarrow .id$

I_5: $L \rightarrow id.$

I_6: $S \rightarrow L=.R$
$R \rightarrow .L$
$L \rightarrow .*R$
$L \rightarrow .id$

I_7: $L \rightarrow *R.$

I_8: $R \rightarrow L.$

I_9: $S \rightarrow L=R.$

Problem

FOLLOW(R)={=,$}

= → shift 6

→ reduce by $R \rightarrow L$

shift/reduce conflict

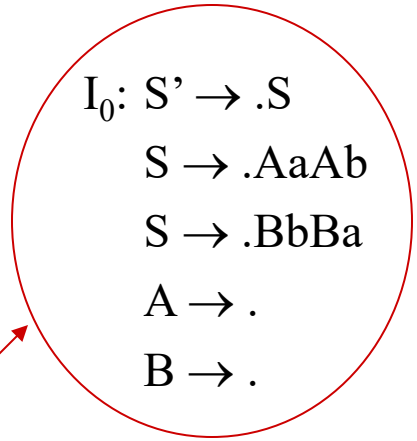
Conflict Example2

$S \rightarrow AaAb$

$S \rightarrow BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$



Problem

FOLLOW(A)={a,b}

FOLLOW(B)={a,b}

a → reduce by $A \rightarrow \varepsilon$

a → reduce by $B \rightarrow \varepsilon$

reduce/reduce conflict

b → reduce by $A \rightarrow \varepsilon$

b → reduce by $B \rightarrow \varepsilon$

reduce/reduce conflict

Construct the SLR parsing table for the following grammar

1. E → E + T / T
   T → T * T / F
   F → (E) / id

2. S → (L)/a
   L → L,S/S

3. S → a / ^ / (T)
   T → T,S / S

4. S → L=R
   S → R
   L → *R
   L → id
   R → L

5. S → aABe
   A → Abc/b
   B → d

Constructing Canonical LR(1) Parsing Tables

- In SLR method, the state i makes a reduction by $A \rightarrow \alpha$ when the current token is a:
  - if the $A \rightarrow \alpha\bullet$ in the I_i and a is FOLLOW(A)
- In some situations, βA cannot be followed by the terminal a in a right-sentential form when $\beta\alpha$ and the state i are on the top stack. This means that making reduction in this case is not correct.

$S \rightarrow AaAb$	$S \Rightarrow AaAb \Rightarrow Aab \Rightarrow ab$	$S \Rightarrow BbBa \Rightarrow Bba \Rightarrow ba$
$S \rightarrow BbBa$		
$A \rightarrow \varepsilon$	$Aab \Rightarrow \varepsilon ab$	$Bba \Rightarrow \varepsilon ba$
$B \rightarrow \varepsilon$	$AaAb \Rightarrow Aa \varepsilon b$	$BbBa \Rightarrow Bb \varepsilon a$

LR(1) Item

- To avoid some of invalid reductions, the states need to carry more information.
- Extra information is put into a state by including a terminal symbol as a second component in an item.
- A LR(1) item is:

$A \rightarrow \alpha . \beta, a$ where **a** is the look-head of the LR(1) item

(**a** is a terminal or end-marker.)

LR(1) Item (cont.)

- When β (in the LR(1) item $A \rightarrow \alpha \bullet \beta, a$) is not empty, the look-head does not have any affect.
- When β is empty ($A \rightarrow \alpha \bullet, a$), we do the reduction by $A \rightarrow \alpha$ only if the next input symbol is **a** (not for any terminal in FOLLOW(A)).
- A state will contain $A \rightarrow \alpha \bullet, a_1$ where $\{a_1, \ldots, a_n\} \subseteq$ FOLLOW(A)

  ...

  $A \rightarrow \alpha \bullet, a_n$

Canonical Collection of Sets of LR(1) Items

- The construction of the canonical collection of the sets of LR(1) items are similar to the construction of the canonical collection of the sets of LR(0) items, except that *closure* and *goto* operations work a little bit different.

closure(I) is: (where I is a set of LR(1) items)

- every LR(1) item in I is in closure(I)
- if $A \rightarrow \alpha \bullet B\beta, a$ in closure(I) and $B \rightarrow \gamma$ is a production rule of G; then $B \rightarrow .\gamma, b$ will be in the closure(I) for each terminal b in FIRST(βa) .

goto operation

- If I is a set of LR(1) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:
  - If $A \rightarrow \alpha.X\beta,a$ in I
    then every item in **closure({$A \rightarrow \alpha X.\beta,a$})** will be in goto(I,X).

Construction of The Canonical LR(1) Collection

- ***Algorithm***:

  C is { closure({S'→.S,$}) }

  repeat the followings until no more set of LR(1) items can be added to ***C***.

  for each I in ***C*** and each grammar symbol X

  if goto(I,X) is not empty and not in ***C***

  add goto(I,X) to ***C***

- goto function is a DFA on the sets in C.

A Short Notation for The Sets of LR(1) Items

- A set of LR(1) items containing the following items

$$A \rightarrow \alpha \bullet \beta, a_1$$

$$\ldots$$

$$A \rightarrow \alpha \bullet \beta, a_n$$

can be written as

$$A \rightarrow \alpha \bullet \beta, a_1/a_2/\ldots/a_n$$

Canonical LR(1) Collection -- Example

$S \rightarrow AaAb$

$S \rightarrow BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

I_0: $S' \rightarrow .S$,\$
$S \rightarrow .AaAb$,\$
$S \rightarrow .BbBa$,\$
$A \rightarrow .$,a
$B \rightarrow .$,b

S → I_1: $S' \rightarrow S.$,\$

A → I_2: $S \rightarrow A.aAb$,\$ —a→ to I_4

B → I_3: $S \rightarrow B.bBa$,\$ —b→ to I_5

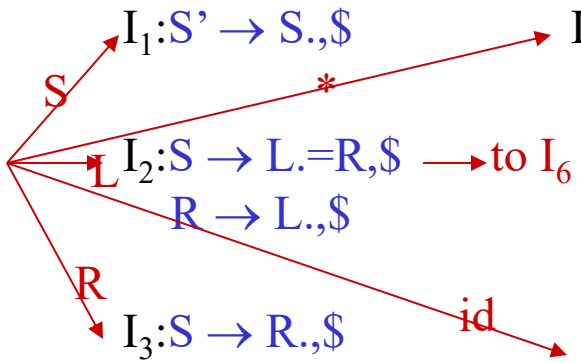
I_4: $S \rightarrow Aa.Ab$,\$ —A→ I_6: $S \rightarrow AaA.b$,\$ —a→ I_8: $S \rightarrow AaAb.$,\$
$A \rightarrow .$,b

I_5: $S \rightarrow Bb.Ba$,\$ —B→ I_7: $S \rightarrow BbB.a$,\$ —b→ I_9: $S \rightarrow BbBa.$,\$
$B \rightarrow .$,a

Canonical LR(1) Collection – Example2

S' → S
1) S → L=R
2) S → R
3) L→ *R
4) L → id
5) R → L

I_0: S' → .S,$
S → .L=R,$
S → .R,$
L → .*R,$/=
L → .id,$/=
R → .L,$



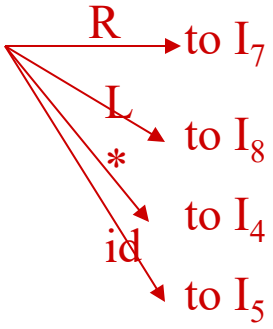
I_1: S' → S.,$

I_2: S → L.=R,$ → to I_6
R → L.,$

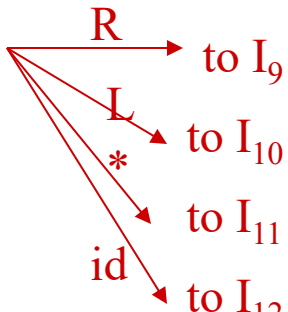
I_3: S → R.,$

I_4: L → *.R,$/=
R → .L,$/=
L→ .*R,$/=
L → .id,$/=

I_5: L → id.,$/=



I_6: S → L=.R,$
R → .L,$
L → .*R,$
L → .id,$



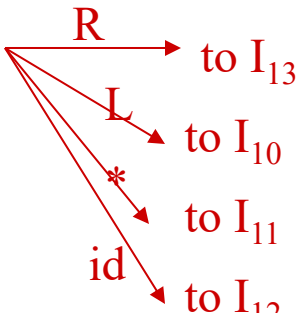
I_7: L → *R.,$/=

I_8: R → L.,$/=

I_9: S → L=R.,$

I_{10}: R → L.,$

I_{11}: L → *.R,$
R → .L,$
L→ .*R,$
L → .id,$



I_{12}: L → id.,$

I_{13}: L → *R.,$

I_4 and I_{11}

I_5 and I_{12}

I_7 and I_{13}

I_8 and I_{10}

Construction of LR(1) Parsing Tables

1. Construct the canonical collection of sets of LR(1) items for G'.
   $C \leftarrow \{I_0,...,I_n\}$

2. Create the parsing action table as follows
   - If a is a terminal, $A \rightarrow \alpha \bullet a\beta,b$ in I_i and $goto(I_i,a)=I_j$ then action[i,a] is ***shift j.***
   - If $A \rightarrow \alpha \bullet,a$ is in I_i, then action[i,a] is ***reduce*** $A \rightarrow \alpha$ where $A \neq S'$.
   - If $S' \rightarrow S \bullet,\$$ is in I_i, then action[i,$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not LR(1).

3. Create the parsing goto table
   - for all non-terminals A, if $goto(I_i,A)=I_j$ then goto[i,A]=j

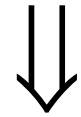
4. All entries not defined by (2) and (3) are errors.

5. Initial state of the parser contains $S' \rightarrow .S,\$$

LR(1) Parsing Tables – (for Example2)

	id	*	=	$	S	L	R
0	s5	s4			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s5	s4				8	7
5			r4	r4			
6	s12	s11				10	9
7			r3	r3			
8			r5	r5			
9				r1			
10				r5			
11	s12	s11				10	13
12				r4			
13				r3			

no shift/reduce or
no reduce/reduce conflict



so, it is a LR(1) grammar

LALR Parsing Tables

- **LALR** stands for **LookAhead LR.**

- LALR parsers are often used in practice because LALR parsing tables are smaller than LR(1) parsing tables.
- The number of states in SLR and LALR parsing tables for a grammar G are equal.
- But LALR parsers recognize more grammars than SLR parsers.
- ***yacc*** creates a LALR parser for the given grammar.
- A state of LALR parser will be again a set of LR(1) items.

Creating LALR Parsing Tables

Canonical LR(1) Parser ➔ LALR Parser

shrink # of states

- This shrink process may introduce a **reduce/reduce** conflict in the resulting LALR parser (so the grammar is NOT LALR)
- But, this shrink process does not produce a **shift/reduce** conflict.

The Core of A Set of LR(1) Items

- The core of a set of LR(1) items is the set of its first component.

Ex: $S \rightarrow L \bullet =R,\$$ ➔ $S \rightarrow L \bullet =R$ ← Core

$R \rightarrow L \bullet ,\$$ $R \rightarrow L \bullet$

- We will find the states (sets of LR(1) items) in a canonical LR(1) parser with same cores. Then we will merge them as a single state.

I_1: $L \rightarrow id \bullet ,=$ A new state: I_{12}: $L \rightarrow id \bullet ,=$

➔ $L \rightarrow id \bullet ,\$$

I_2: $L \rightarrow id \bullet ,\$$ have same core, merge them

- We will do this for all states of a canonical LR(1) parser to get the states of the LALR parser.
- In fact, the number of the states of the LALR parser for a grammar will be equal to the number of states of the SLR parser for that grammar.

Creation of LALR Parsing Tables

- Create the canonical LR(1) collection of the sets of LR(1) items for the given grammar.
- Find each core; find all sets having that same core; replace those sets having same cores with a single set which is their union.

  $C=\{I_0,...,I_n\}$ ➔ $C'=\{J_1,...,J_m\}$ where $m \leq n$
- Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser.
  - Note that: If $J=I_1 \cup ... \cup I_k$ since $I_1,...,I_k$ have same cores

    ➔ cores of $goto(I_1,X),...,goto(I_2,X)$ must be same.
  - So, $goto(J,X)=K$ where K is the union of all sets of items having same cores as $goto(I_1,X)$.

- If no conflict is introduced, the grammar is LALR(1) grammar. (We may only introduce reduce/reduce conflicts; we cannot introduce a shift/reduce conflict)

Shift/Reduce Conflict

- We say that we cannot introduce a shift/reduce conflict during the shrink process for the creation of the states of a LALR parser.
- Assume that we can introduce a shift/reduce conflict. In this case, a state of LALR parser must have:

$$A \rightarrow \alpha \bullet, a \qquad \text{and} \qquad B \rightarrow \beta \bullet a\gamma, b$$

- This means that a state of the canonical LR(1) parser must have:

$$A \rightarrow \alpha \bullet, a \qquad \text{and} \qquad B \rightarrow \beta \bullet a\gamma, c$$

But, this state has also a shift/reduce conflict. i.e. The original canonical LR(1) parser has a conflict.

(Reason for this, the shift operation does not depend on lookaheads)

Reduce/Reduce Conflict

- But, we may introduce a reduce/reduce conflict during the shrink process for the creation of the states of a LALR parser.

$I_1 : A \rightarrow \alpha \bullet, a$

$B \rightarrow \beta \bullet, b$

$I_2 : A \rightarrow \alpha \bullet, b$

$B \rightarrow \beta \bullet, c$

$I_{12} : A \rightarrow \alpha \bullet, a/b$

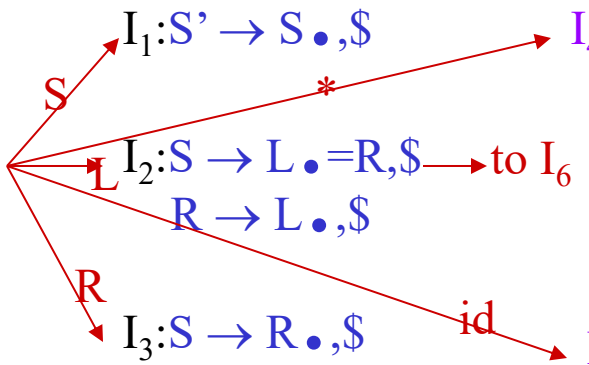
$B \rightarrow \beta \bullet, b/c$

➔ reduce/reduce conflict

Canonical LALR(1) Collection – Example2

S' → S
1) S → L=R
2) S → R
3) L→ *R
4) L → id
5) R → L

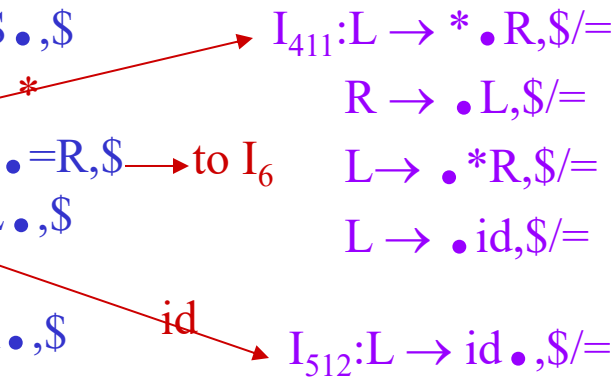
I_0: S' → •S,$
S → •L=R,$
S → •R,$
L → •*R,$/=
L → •id,$/=
R → •L,$



I_1: S' → S•,$

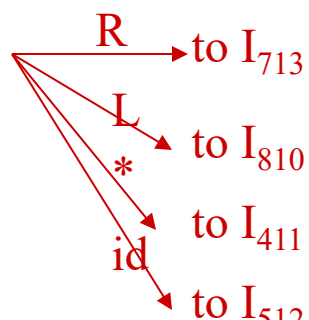
I_2: S → L•=R,$ to I_6
R → L•,$

I_3: S → R•,$

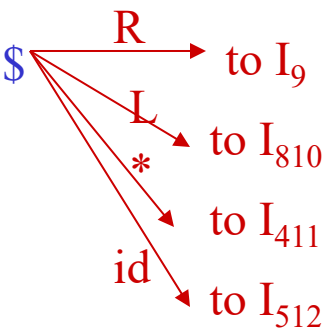


I_{411}: L → *•R,$/=
R → •L,$/=
L→ •*R,$/=
L → •id,$/=

I_{512}: L → id•,$/=



I_6: S → L=•R,$
R → •L,$
L → •*R,$
L → •id,$



I_9: S → L=R•,$

I_{713}: L → *R•,$/=

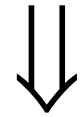
I_{810}: R → L•,$/=

Same Cores
I_4 and I_{11}
I_5 and I_{12}
I_7 and I_{13}
I_8 and I_{10}

LALR(1) Parsing Tables – (for Example2)

	id	*	=	$	S	L	R
0	s5	s4			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s5	s4				8	7
5			r4	r4			
6	s12	s11				10	9
7			r3	r3			
8			r5	r5			
9				r1			

no shift/reduce or
no reduce/reduce conflict



so, it is a LALR(1) grammar

INTERMEDIATE CODE GENERATION

Benefit

Intermediate Code Generation

Benefits of machine independent intermediate code

- Easy to change the source or the target language by adapting only the front-end or back-end.
- It makes optimization easier.
- The intermediate representation can be directly interpreted.

Intermediate languages

Commonly used intermediate code representations

1. Linear representation - Postfix notation (POSIX)
2. Graphical representation
3. Three-address code
4. Static Single Assignment form (SSA)

The choice of intermediate code depends on two important criteria

- Easy to convert the source program into intermediate code
- Easy for the subsequent processing to obtain the object program

Three Address Code

- It s one of the commonly used Intermediate Code Representations.
- In the three address code form, at the most, three addresses are used to represent any statement.
- The general form of three address code representation is **a = b op c** where a, b, c are operands and op is an operator.

Example :

a = b + c

c = a * b

Example :

Consider the expression x = a + b + c.

Its three address code will be

$t_0 = a + b$

$t_1 = t_0 + c$

$x = t_1$

where t_0 and t_1 are temporary names generated by the compiler.

Implementation of three address statements

Three address code is an abstract form of intermediate code that can be implemented as records with fields for operator and operands.

The three representations are

1. Quadruple representation
2. Triple representation
3. Indirect triple representation

1. Quadruple

- ✓ The quadruple is a structure with at the most four fields such as op, arg1, arg2 and result.
- ✓ The op field is used to represent the internal node for operator. The arg1 and arg2 represent the two operands. And result field is used to store the result of an expression.
- ✓ The three address statements with unary operator like a = -y or a = y do not use agr2.

x= -a*b + -a*b

t_1= uminus a

$t_2 = t_1 * b$

t_3= uminus a

$t_4 = t_3 * b$

$t_5 = t_2 + t_4$

x= t_5

Quadruple representation

	Operator	Arg1	Arg2	Result
(0)	uminus	a		t_1
(1)	*	t_1	b	t_2
(2)	uminus	a		t_3
(3)	*	t_3	b	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		x

2. Triple

- ✓ In triple, temporaries are not used instead of that pointers in the symbol table are used directly
- ✓ If we do so, three address statements can be represented by records with only three fields: op, arg1 and arg2.
- ✓ Numbers in the round bracket ( ) are used to represent pointers into the triple structure.

Quadruple representation

	Operator	Arg1	Arg2	Result
(0)	uminus	a		t_1
(1)	*	t_1	b	t_2
(2)	uminus	a		t_3
(3)	*	t_3	b	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		x

Triple representation

	Operator	Arg1	Arg2
(0)	uminus	a	
(1)	*	(0)	b
(2)	uminus	a	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	x	(4)

3. Indirect Triples

- ✓ In the indirect triple representation the listing of triples has been done. And listing pointers are used instead of using statement.
- ✓ This implementation is called indirect triples.

Triple representation

No.	Operator	Arg1	Arg2
(0)	uminus	a	
(1)	*	(0)	b
(2)	uminus	a	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	x	(4)

	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

No.	Operator	Arg1	Arg2
(0)	uminus	a	
(1)	*	(14)	b
(2)	uminus	a	
(3)	*	(16)	b
(4)	+	(15)	

Translate the following expression to quadruple, triple and indirect triple-

$$a + b \times c / e \uparrow f + b \times c$$

Solution-

Three Address Code for the given expression is-

$T1 = e \uparrow f$
$T2 = b \times c$
$T3 = T2 / T1$
$T4 = b \times a$
$T5 = a + T3$
$T6 = T5 + T4$

Quadruple Representation-

Location	Op	Arg1	Arg2	Result
(0)	↑	e	f	T1
(1)	x	b	c	T2
(2)	/	T2	T1	T3
(3)	x	b	a	T4
(4)	+	a	T3	T5
(5)	+	T5	T4	T6

Triple Representation

Location	Op	Arg1	Arg2
(0)	↑	e	f
(1)	x	b	c
(2)	/	(1)	(0)
(3)	x	b	a
(4)	+	a	(2)
(5)	+	(4)	(3)

Indirect Triple Representation-

	Statement
(0)	35
(1)	36
(2)	37
(3)	38
(4)	39
(5)	40

Location	Op	Arg1	Arg2
(0)	↑	E	f
(1)	x	B	e
(2)	/	(36)	(35)
(3)	X	b	a
(4)	+	a	(37)
(5)	+	(39)	(38)