SYNTAX ANALYSIS

The Role of the Parser

The parser obtains a string of tokens from the lexical analyzer, as shown in Fig. 4.1, and verifies that the string of token names can be generated by the grammar for the source language. Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing. In fact, the parse tree need not be constructed explicitly, since checking and translation actions can be interspersed with parsing.

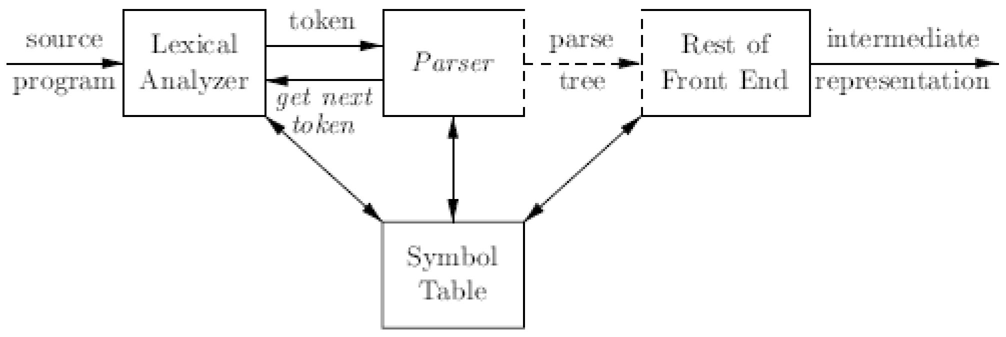


Figure 4.1: Position of parser in compiler model

There are three general types of parsers for grammars: universal, top-down, and bottom-up. Universal parsing methods such as the Cocke-Younger-Kasami algorithm and Earley's algorithm can parse any grammar. These general methods are, however, too inefficient to use in production compilers.

The methods commonly used in compilers can be classified as being either <u>top-down or bottom-up.</u> As implied by their names, <u>top-down methods build parse trees from the top (root) to the bottom (leaves),</u> while <u>bottom-up methods start from the leaves and work their way up to the root</u>. In either case, the input to the parser is scanned from left to right, one symbol at a time.

The most efficient top-down and bottom-up methods work only for sub- classes of grammars, but several of these classes, particularly, LL and LR grammars, are expressive enough to describe most of the syntactic constructs in modern programming languages. Parsers implemented by hand often use LL grammars; Parsers for the larger class of LR grammars are usually constructed using automated tools.

Representative Grammars

$$\left.\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid id \end{array}\right. \longrightarrow \quad (4.1)$$

Expression grammar (4.1) belongs to the class of LR grammars that are suitable for bottom-up parsing. This grammar can be adapted to handle additional operators and additional levels of precedence. However, it cannot be used for top-down parsing because it is left recursive. The following non-left-recursive variant of the expression grammar (4.1) will be used for top-down parsing:

E-> T E'

E'-> +T E' | Ɛ

T-> F T' $\longrightarrow$ (4.2)

T'-> *F T' | Ɛ

F-> (E) | id

Context-Free Grammars

Using a syntactic variable stmt to denote statements and variable expr to denote expressions, the production

stmt-> if (expr) stmt else stmt $\longrightarrow$ (4.4)

Specifies the structure of this form of conditional statement. Other productions then de ne precisely what an expr is and what else a stmt can be.

The Formal Definition of a Context-Free Grammar

1. Terminals are the basic symbols from which strings are formed. Terminals are the keywords **if** and **else** and the symbols "(" and ")."

2. Nonterminals are syntactic variables that denote sets of strings. In (4.4), stmt and expr are nonterminals. The sets of strings denoted by non terminals help define the language generated by the grammar. Nonterminals impose a hierarchical structure on the language that is key to syntax analysis and translation.

3. In a grammar, one nonterminal is distinguished as the start symbol, and the set of strings it denotes is the language generated by the grammar.

4. The productions of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings. Each production consists of:

(a) A nonterminal called the head or left side of the production; this production defines some of the strings denoted by the head.

(b) The symbol→. Sometimes ::= has been used in place of the arrow.

(c) A body or right side consisting of zero or more terminals and non- terminals. The components of the body describe one way in which strings of the nonterminal at the head can be constructed.

The grammar in Fig. 4.2 defines simple arithmetic expressions.

In this grammar, the terminal symbols are

id + - * / ( )

The nonterminal symbols are **expression, term** and **factor**, and expression is the start symbol.

```
expression -> expression + term
expression -> expression – term
expression -> term
term -> term * factor
term -> term / factor
term -> factor
factor -> (expression)
factor -> id
```

Figure 4.2: grammar for simple arithmetic expression

Notational Conventions

To avoid always having to state that “these are the terminals," “these are the nonterminals," and so on, the following notational conventions for grammars will be used:

1. These symbols are terminals:

(a) Lowercase letters early in the alphabet, such as a, b, c.

(b) Operator symbols such as +, _, and so on.

(c) Punctuation symbols such as parentheses, comma, and so on.

(d) The digits 0,1,……,9.

(e) Boldface strings such as id or if, each of which represents a single terminal symbol.

2. These symbols are nonterminals:

(a) Uppercase letters early in the alphabet, such as A, B, C.

(b) The letter S, which, when it appears, is usually the start symbol.

(c) Lowercase, italic names such as expr or stmt.

(d) When discussing programming constructs, uppercase letters may be used to represent nonterminals for the constructs. For example, non-terminals for expressions, terms, and factors are often represented by E, T, and F, respectively.

3. Uppercase letters late in the alphabet, such as X, Y, Z, represent grammar symbols; that is, either nonterminals or terminals.

4. Lowercase letters late in the alphabet, chiey u,v,……,z, represent (possibly empty) strings of terminals.

5. Lowercase Greek letters, α, β, γ for example, represent (possibly empty) strings of grammar symbols. Thus, a generic production can be written as A ! _, where A is the head and _ the body.

6. A set of productions A -> α_1, A -> α_2, …… A -> α_k with a common head A (call them A-productions), may be written A -> A -> α_1,| α_2 |……| α_k Call α_1, α_2, ……, α_k the alternatives for A.

7. Unless stated otherwise, the head of the _rst production is the start symbol.

Using these conventions, the grammar of Example 4.5 can be rewritten concisely as

E-> E + T | E – T | T

T-> T * F | T / F | F

F-> (E) | id

The notational conventions tell us that E, T, and F are nonterminals, with E the start symbol. The remaining symbols are terminals.

Derivations

The construction of a parse tree can be made precise by taking a derivational view, in which productions are treated as rewriting rules. Beginning with the start symbol, each rewriting step replaces a nonterminal by the body of one of its productions. This derivational view corresponds to the top-down construction of a parse tree.

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.

The arithmetic expression grammar (4.3) permits two distinctleftmost derivations for the sentence id + id * id:

```
E=> E + E              E=> E * E
 ⇨  id + E              => E + E * E
 ⇨  id + E * E          => id + E * E
 ⇨  id + id * E         => id + id * E
 ⇨  id + id * id        => id + id * id
```

It is convenient to use carefully chosen ambiguous grammars, together with disambiguating rules that “throw away" undesirable parse trees, leaving only one tree for each sentence.

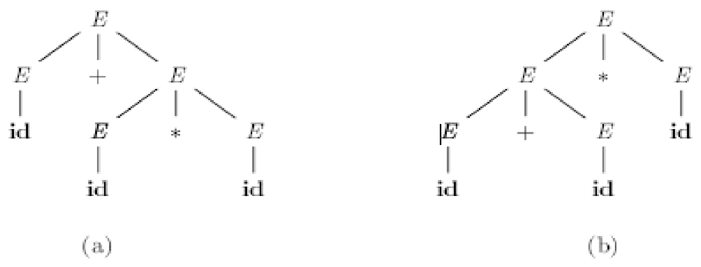


Figure 4.5: Two parse trees for **id+id*id**

Elimination of Left Recursion

A grammar is left recursive if it has a nonterminal A such that there is a derivation $A \Rightarrow^{+} A\alpha$ for some string α . Top-down parsing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion. The left-recursive pair of productions $A \rightarrow A\alpha|\beta$ could be replaced by the non-left-recursive productions:

$$A \rightarrow \beta A$$

$$A' \rightarrow \alpha A' | \varepsilon$$

without changing the strings derivable from A. This rule by itself suffices for many grammars.
could be replaced by the non-left-recursive productions:
The non-left-recursive expression grammar (4.2), repeated here,

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

is obtained by eliminating immediate left recursion from the expression grammar (4.1). The left-recursive pair of productions E →E + T | T are replaced by **E→TE'** and **E'→+TE'| ε.** The new productions for T and T' are obtained similarly by eliminating immediate left recursion.
Immediate left recursion can be eliminated by the following technique, which works for any number of A-productions. First, group the productions as:

$A \rightarrow A\alpha_1 | A\alpha_2 | \ldots\ldots\ldots\ldots\ldots\ldots | A\alpha_m | \beta_1 | \beta_2 | \ldots\ldots\ldots\ldots\ldots\ldots \beta_n$

where no β_i begins with an A. Then, replace the A-productions by

$A \rightarrow \beta_1 A^1 | \beta_2 A^1 | \ldots | \beta_n A^1$
$A^1 \rightarrow \alpha_1 A^1 | \alpha_2 A^1 | \ldots | \alpha_m A^1 | \varepsilon$

The nonterminal A generates the same strings as before but is no longer left recursive. For example, consider the grammar:

S->Aa | b
A-> Ac | Sd | Ɛ

The nonterminal S is left recursive because **S=>Aa =>Sda**, but it is not immediately left recursive.
Algorithm: Eliminating left recursion.
INPUT: Grammar G with no cycles or -productions.
OUTPUT: An equivalent grammar with no left recursion.
METHOD: Apply the algorithm to G.

```
1) arrange the nonterminals in some order $A_1, A_2, \ldots\ldots, A_n$.
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 $A_i \rightarrow \delta_1\gamma$
        $| \delta_2\gamma | \ldots | \delta_k\gamma$ , where $A_j \rightarrow \delta_1 | \delta_2 | \ldots | \delta_k$ are all current $A_j$-productions
5)     }
6)     eliminate the immediate left recursion among the $A_i$-productions
7) }
```

PRACTICE PROBLEMS BASED ON LEFT RECURSION ELIMINATION

1. A → ABd / Aa / a

B → Be / b

<u>Solution:</u> The grammar after eliminating left recursion is-

A → aA'

A' → BdA' / aA' / ∈

B → bB'

B' → eB' / ∈

2. E → E + E / E * E / a

<u>Solution:</u>The grammar after eliminating left recursion is-

E → aE'

E' → +EE' / *EE'/ ∈

3. E → E + T / T

T → T * F / F

F → id

Solution:- The grammar after eliminating left recursion is-

E → TE'

E' → +TE' / ∈

T → FT'

T' → *FT' / ∈

F → id

4. S → (L) / a

L → L , S / S

<u>Solution:-</u>The grammar after eliminating left recursion is-

S → (L) / a

L → SL'

L' → ,SL' / ∈

5. **S → S0S1S / 01**

<u>Solution-:</u>The grammar after eliminating left recursion is-

S → 01S'

S' → 0S1SS' / ∈

6. S → A

A → Ad / Ae / aB / ac

B → bBc / f

Solution:-**The grammar after eliminating left recursion is-**

S → A

A → aBA’ / acA’

A’ → dA’ / eA’ / ∈

B → bBc / f

7. A → AAα / β

Solution-: The grammar after eliminating left recursion is-

A → βA’

A’ → AαA’ / ∈

8. A → Ba / Aa / c

B → Bb / Ab / d

Solution-:This is a case of indirect left recursion.

Step-01:

First let us eliminate left recursion from A → Ba / Aa / c

Eliminating left recursion from here, we get-

A → BaA’ / cA’

A’ → aA’ / ∈

Now, given grammar becomes-

A → BaA’ / cA’

A’ → aA’ / ∈

B → Bb / Ab / d

Step-02:

Substituting the productions of A in B → Ab, we get the following grammar-

A → BaA’ / cA’

A’ → aA’ / ∈

B → Bb / BaA’b / cA’b / d

Step-03:

Now, eliminating left recursion from the productions of B, we get the following grammar-

A → BaA’ / cA’

A’ → aA’ / ∈

B → cA’bB’ / dB’

B’ → bB’ / aA’bB’ / ∈

9. X → XSb / Sa / b

S → Sb / Xa / a

Solution-:This is a case of indirect left recursion.

Step-01:

First let us eliminate left recursion from

$X \rightarrow XSb$ / Sa / b

Eliminating left recursion from here, we get-

$X \rightarrow SaX'$ / bX'

$X' \rightarrow SbX'$ / $\in$

Now, given grammar becomes-

$X \rightarrow SaX'$ / bX'

$X' \rightarrow SbX'$ / $\in$

$S \rightarrow Sb$ / Xa / a

<u>Step-02:</u>

Substituting the productions of X in S → Xa, we get the following grammar-

$X \rightarrow SaX'$ / bX'

$X' \rightarrow SbX'$ / $\in$

$S \rightarrow Sb$ / $SaX'a$ / $bX'a$ / a

<u>Step-03:</u>

Now, eliminating left recursion from the productions of S, we get the following grammar-

$X \rightarrow SaX'$ / bX'

$X' \rightarrow SbX'$ / $\in$

$S \rightarrow bX'aS'$ / aS'

$S' \rightarrow bS'$ / $aX'aS'$ / $\in$

Left Factoring:

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing. For example, if we have the two productions

stmt -> if expr then stmt else stmt

| if expr then stmt

on seeing the input if, we cannot immediately tell which production to choose to expand stmt. In general,if A→**$\alpha\beta_1$| $\alpha\beta_2$** are two A-productions, and the input begins with a nonempty string derived from **α** we do not know whether to expand A to **$\alpha\beta_1$| $\alpha\beta_2$.** The decision by expanding A to **αA'.** Then, after seeing the input derived from ,we expand **A' to β_1 or β_2** to That is, left-factored, the original productions become

$A \rightarrow \alpha A^1$

$A^1 \rightarrow \beta_1 \mid \beta_2$

Algorithm: Left factoring a grammar

INPUT: Grammar G

OUTPUT: An equivalent left-factored grammar

METHOD: For each nonterminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \mathcal{E}$, i.e. there is a nontrivial common prefix, replace all of the A-productions $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid .. \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α by

$A \rightarrow \alpha A^1 \mid \gamma$

$A^1 \rightarrow \beta_1 \mid \beta_2 \mid .. \mid \beta_n$

Here A^1 is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

The following grammar abstracts the “dangling-else" problem:

S-> I E t S | I E t S e S | a

E-> b

Here, i, t, and e stand for if, then, and else; E and S stand for “conditional expression" and “statement." Left-factored, this grammar becomes

$S \rightarrow i\ E\ t\ S\ S^1 \mid a$
$S^1 \rightarrow e\ S \mid \varepsilon$
E-> b

PRACTICE PROBLEMS BASED ON LEFT FACTORING: Left Factoring-

Left factoring is a process by which the grammar with common prefixes is transformed to make it useful for Top down parsers.

How?

In left factoring,

- We make one production for each common prefixes.
- The common prefix may be a terminal or a non-terminal or a combination of both.
- Rest of the derivation is added by new productions.
- The grammar obtained after the process of left factoring is called as Left Factored Grammar.

1. A → aAB / aBc / aAc

Solution-

Step 1:

A → aA’

A’ → AB / Bc / Ac

Again, this is a grammar with common prefixes.

Step-02:

A → aA’

A’ → AD / Bc

D → B / c

This is a left factored grammar.

2. S → bSSaaS / bSSaSb / bSb / a

Solution:

Step 1:

S → bSS’ / a

S’ → SaaS / SaSb / b

Again, this is a grammar with common prefixes.

Step 2:

S → bSS’ / a

S’ → SaA / b

A → aS / Sb

3. S → aSSbS / aSaSb / abb / b

Solution:

Step 1:

S → aS’ / b

S’ → SSbS / SaSb / bb

Again, this is a grammar with common prefixes.

Step 2:

S → aS’ / b

S’ → SA / bb

A → SbS / aSb

4. S → a / ab / abc / abcd

Solution:

Step 1:

S → aS’

S’ → b / bc / bcd / ∈

Again, this is a grammar with common prefixes.

Step-02:

S → aS’

S’ → bA / ∈

A → c / cd / ∈

Again, this is a grammar with common prefixes.

Step-03:

S → aS’

S’ → bA / ∈

A → cB / ∈
B → d / ∈

5.S → aAd / aB
A → a / ab
B → ccd / ddc

Solution:
S → aS’
S’ → Ad / B
A → aA’
A’ → b / ∈
B → ccd / ddc

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. Consider input **id+id*id** is a top-down parse according to grammar

E→TE'

E'→+TE'| ε
T→+FT'
T'→*FT'| ε
F→(E)|id

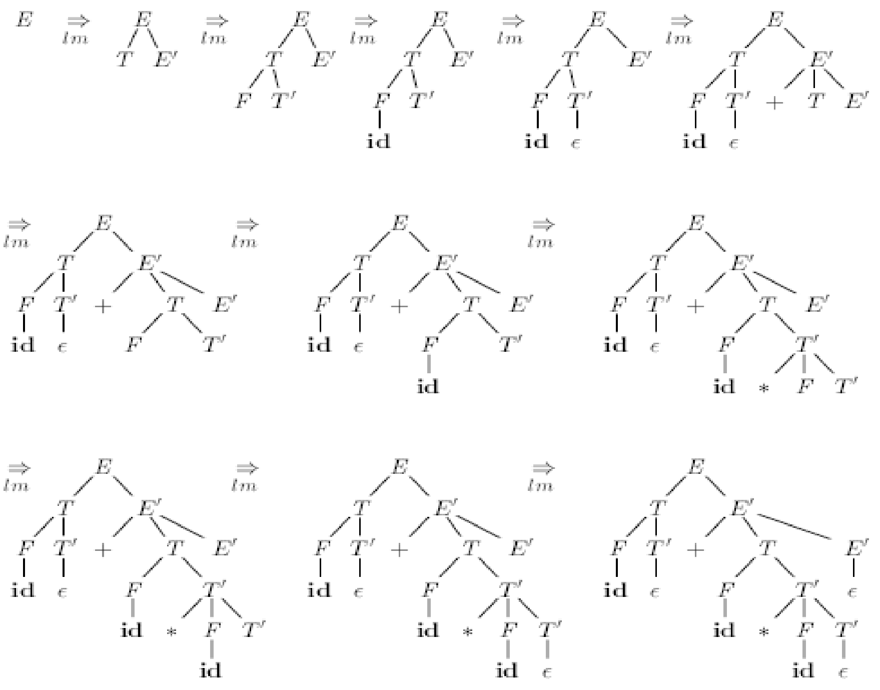


Figure 4.12: Top-down parse for **id** + **id** ∗ **id**

Recursive-Descent Parsing

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

A typical procedure for a nonterminal in a top-down parser

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. Pseudocode for a typical nonterminal is shown above.

General recursive-descent may require backtracking; that is, it may require repeated scans over the input. However, backtracking is rarely needed to parse programming language constructs, so backtracking parsers are not seen frequently.

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 FIRST(α), where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α If $\alpha \Rightarrow^* \varepsilon$ then ε is also in FIRST(α).

For a preview of how FIRST can be used during predictive parsing, consider two A-productions $A \rightarrow \alpha | \beta$ where FIRST(α) and FIRST(β) are disjoint sets.

1. If X is a terminal, then FIRST(X) = { X }

2. If X is a nonterminal and X -> Y_1Y_2…… Yk is a production for some k >=1, then place a in FIRST(X) if for some i, a is in FIRST(Yi), and Ɛ is in all of FIRST(Y_1), …… , FIRST(Y_{i-1}); that is, Y1…… Yi-1)=>Ɛ. If Ɛ is in FIRST(Yj) for all j = 1,2,……, k, then add Ɛ to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive Ɛ, then we add nothing more to FIRST(X), but if Y_1 =>Ɛ, then we add FIRST(Y2), and so on.

3. If X -> Ɛis a production, then add Ɛ to FIRST(X).

Now, we can compute FIRST for any string X_1X_2 …… Xn as follows. Add to FIRST(X_1X_2 …… X_n) all non-Ɛ symbols of FIRST(X_1). Also add the non-Ɛ symbols of FIRST(X_2), if Ɛ is in FIRST(X_1); the non- Ɛ symbols of FIRST(X_3), if Ɛ is in FIRST(X_1) and FIRST(X_2); and so on. Finally, add Ɛ to FIRST(X_1X_2 …… Xn) if, for all i, Ɛ is in FIRST(Xi).

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

1. Place $ in FOLLOW(S), where S is the start symbol, and $ is the input right endmarker.
2. If there is a production A-> αBβ, then everything in FIRST(β) except Ɛ is in FOLLOW(B).
3. If there is a production A->αB, or a production A-> αBβ, where FIRST(β) contains Ɛ, then everything in FOLLOW(A) is in FOLLOW(B).

Consider again the non-left-recursive grammar

E→TE'

E'→+TE'| ε

T→+FT'

T'→*FT'| ε

F→(E)|id

1. FIRST(F)=FIRST(T)=FIRST(E)={ (, 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 Ɛ, FIRST(T) must be the same as FIRST(F). the same argument covers FIRST(E).
2. FIRST(E^1)= { +, Ɛ}. The reason is that one of the two productions for E^1 has a body that begins with terminal +, and the others body is Ɛ. Whenever a nonterminal derives Ɛ, we place Ɛ in FIRST for that nonterminal.
3. FIRST(T^1)={ *, Ɛ }. The reasoning is analogous to that for FIRST(E^1).
4. FOLLOW(E)=FOLLOE(E^1)={), $ }. Since E is the start symbol, FOLLOW€ must contain $. The production body (E) explains why the right parenthesis is in FOLLOW(E). for E^1, note that this nonterminal appears only at the ends of bodies of E-productions. Thus, FOLLOW(E^1) must be the same as FOLLOW(E).
5. FOLLOW(T)=FOLLOW(T^1)={ +,), $ }. Notice that T appears in bodies only followed by E^1. Thus, everything except Ɛ that is in FIRST(E^1) must be in FOLLOW(T); that explains the symbol +. However, since FIRST(E^1) contains Ɛ and E^1 is the entire string following T in the bodies of the E-productions, everything in FOLLOW(E) must also be in FOLLOW(T). that explains the symbols $ and the right parenthesis. As for T^1, since it appears only at the ends of the T's production, it must be that FOLLOW(T^1)=FOLLOW(T).
6. FOLLOW(F)={ +, *,), $ }. Reason is same as point 5.

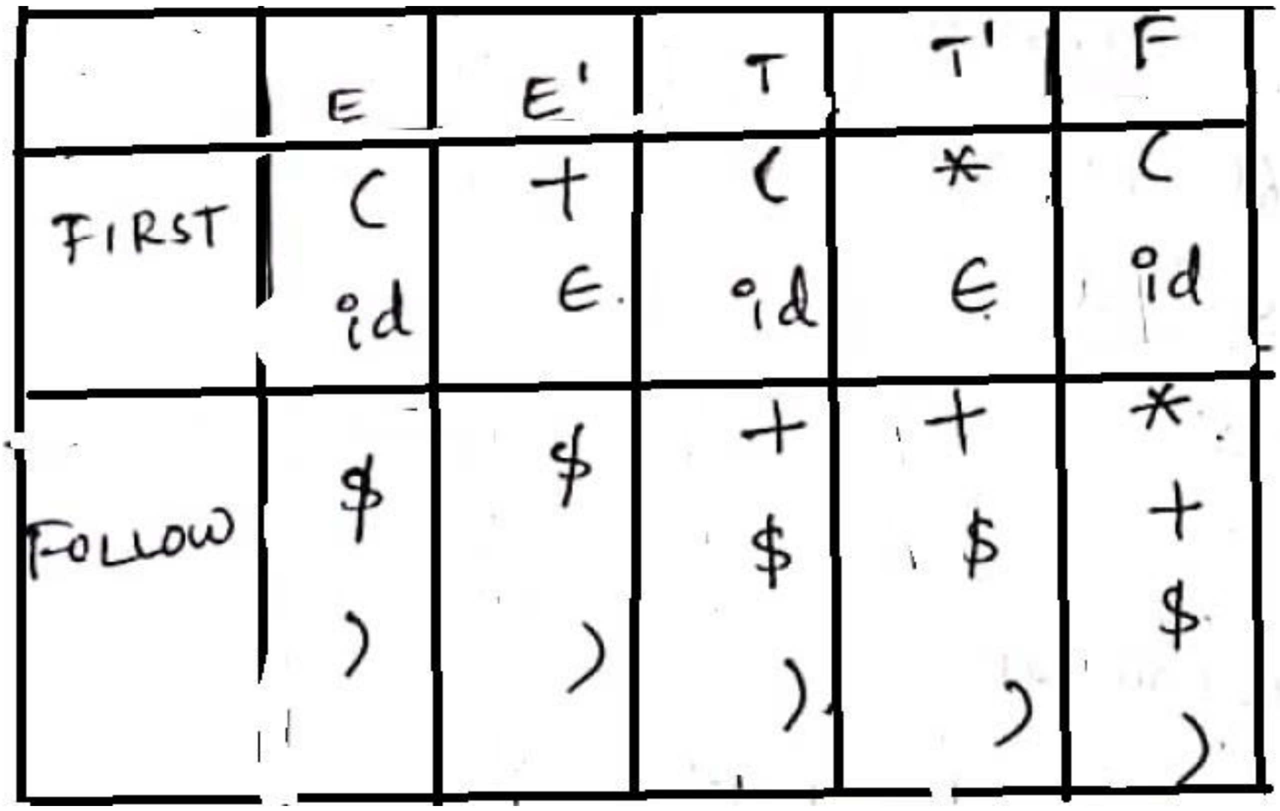
Predictive parsers can be constructed for LL(1) grammars since the proper production to apply for a nonterminal can be selected by looking only at the current input symbol.
The next algorithm collects the information from FIRST and FOLLOW sets into a predictive parsing table M[A,a], a two-dimensional array, where A is a nonterminal, and **a** is a terminal or the symbol $, the input endmarker. The algorithm is based on the following idea: the production $A\rightarrow\alpha$ is chosen if the next input symbol a is in FIRST(α). The only complication occurs when $\alpha=\varepsilon$ or, more generally, $\alpha \Rightarrow^* \varepsilon$. In this case, we should again choose $A\rightarrow\alpha$ if the current input symbol is in FOLLOW(A), or if the $ on the input has been reached and $ is in FOLLOW(A).

Algorithm: Construction of a predictive parsing table
INPUT: Grammar G
OUTPUT: Parsing table M
METHOD: For each production A-> α of the grammar, do the following:

1. For each terminal α in FIRST(α), add A-> α to M[A,a]
2. If Ɛ is in FIRST(α), then for each terminal b in FOLLOW(A), add A-> α to M[A,b]. if α is in FIRST(α) and $ is in FOLLOW(A), add A-> α to M[A,$] as well.

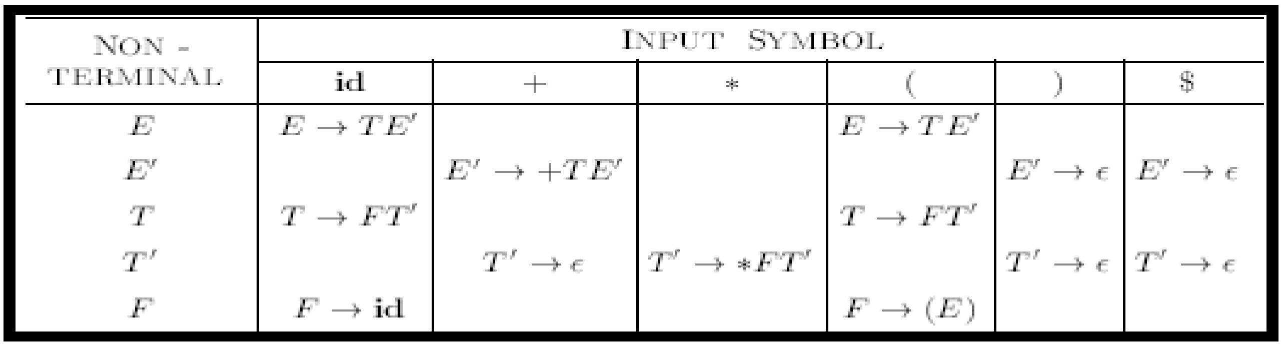
Example:

FIRST and FOLLOW :

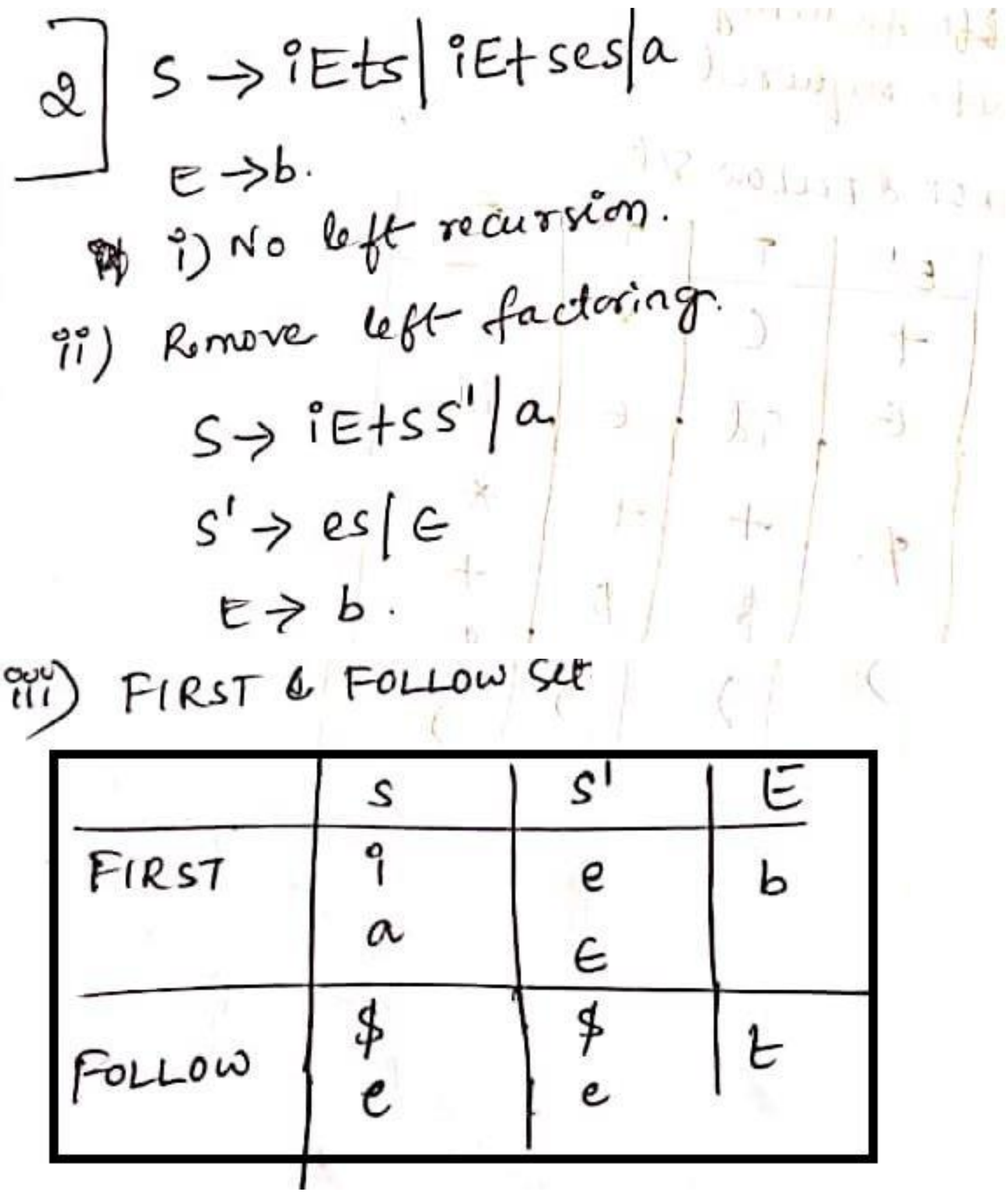


	E	E'	T	T'	F
FIRST	(id	+ ∈	(id	* ∈	(id
FOLLOW	$)	$)	+ $)	+ $)	* + $)

Parsing Table:



NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		



2] $S \rightarrow iEts \mid iEtses \mid a$

$E \rightarrow b.$

i) No left recursion.

ii) Remove left factoring.

$S \rightarrow iEtsS' \mid a$

$S' \rightarrow es \mid \epsilon$

$E \rightarrow b.$

iii) FIRST & FOLLOW set

	S	S'	E
FIRST	i a	e ϵ	b
FOLLOW	\$ e	\$ e	t

Parse table:

	$	i	e	b	a	t
S		S → iEtSS'			S → a	
S'	S' → ε		S' → eS S' → ε			
E				E → b		

3] $S \rightarrow SS+ \mid SS* \mid a$

i) Remove left recursion.

$S \rightarrow aS'$

$S' \rightarrow S+S' \mid S*S' \mid \epsilon$

ii) Remove left-factoring

$S \rightarrow aS'$

$S' \rightarrow SS'' \mid \epsilon$

$S'' \rightarrow +S' \mid *S'$.

iii) Find FIRST and FOLLOW set.

	S	S'	S''
FIRST	a	a ε	+ *
FOLLOW	$ + *	$ + *	$ + *

parsing table

	$	a	+	*
S		$S \rightarrow aS'$		
S'	$S' \rightarrow \epsilon$	$S' \rightarrow SS''$	$S' \rightarrow \epsilon$	$S' \rightarrow \epsilon$
S''			$S'' \rightarrow +S'$	$S'' \rightarrow *S'$

2. $S \to SS+ \mid SS* \mid a$

$S \to aS'$

$S' \to \underline{S+}\,S' \mid S*S' \mid \underline{\epsilon}$

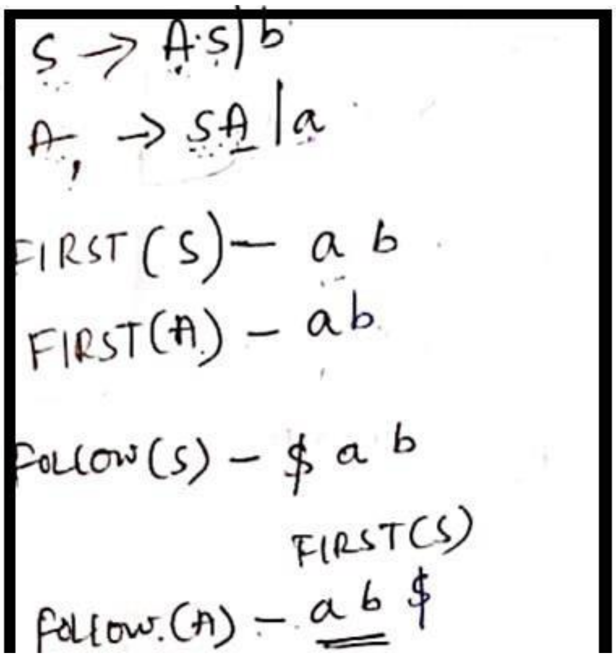
$\Downarrow$ left factoring

final production $S \to a\underline{S'}$

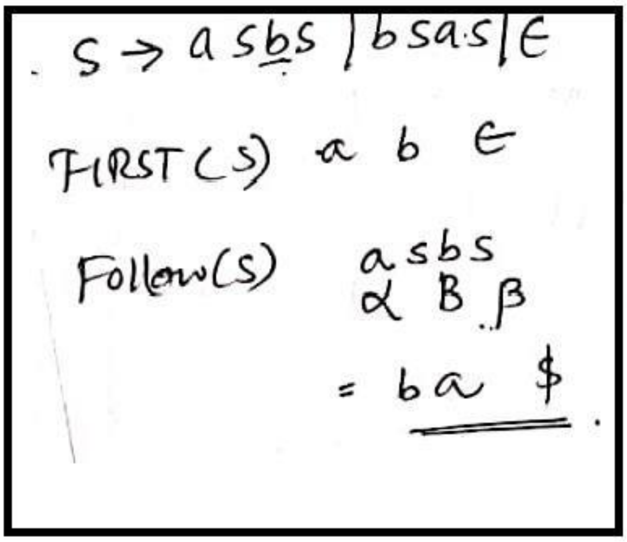
$S' \to SS'' \mid \epsilon$

$S'' \to +S' \mid *S'$

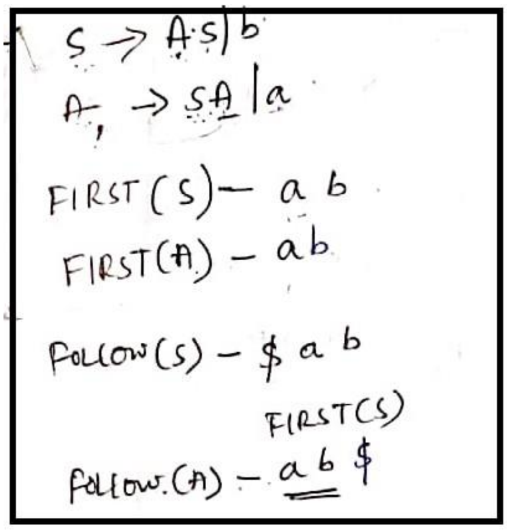
~~FIRST~~	S	S'	S''
FIRST	a	a ϵ	+ *
FOLLOW	\$ + *	\$ + *	\$ + *



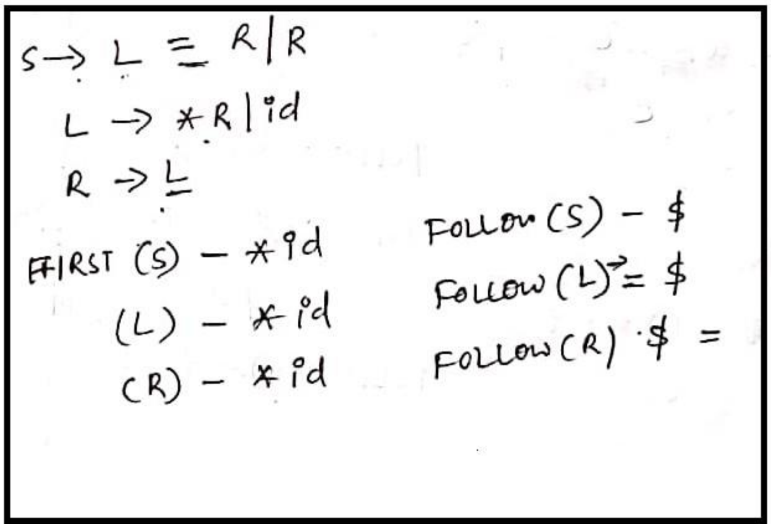
S → A S | b
A → S A | a
FIRST(S) — a b
FIRST(A) — a b
FOLLOW(S) — $ a b
FIRST(S)
follow(A) — a b $



S → a S b S | b S a S | ε
FIRST(S) a b ε
Follow(S) a s b s
α B β
= b a $



S → A S | b
A → S A | a
FIRST(S) — a b
FIRST(A) — a b
FOLLOW(S) — $ a b
FIRST(S)
follow(A) — a b $



S → L = R | R
L → * R | id
R → L
FIRST (S) — * id
(L) — * id
(R) — * id
FOLLOW(S) — $
FOLLOW(L) = $
FOLLOW(R) $ =

Non recursive Predictive Parsing

A nonrecursive predictive parser can be built by maintaining a stack explicitly, rather than implicitly via recursive calls. The parser mimics a leftmost derivation. If w is the input that has been matched so far, then the stack holds a sequence of grammar symbols such that

$$S \underset{lm}{\overset{*}{\Rightarrow}} w\alpha$$

The table-driven parser in Fig. 4.19 has an input buffer, a stack containing a sequence of grammar symbols, a parsing table constructed by Algorithm 4.31, and an output stream. The input buffer contains the string to be parsed, followed by the end marker $. We reuse the symbol $ to mark the bottom of the stack, which initially contains the start symbol of the grammar on top of $.

The parser is controlled by a program that considers X, the symbol on top of the stack, and a, the current input symbol. If X is a nonterminal, the parser chooses an X-production by consulting entry M[X,a] of the parsing table M. Otherwise, it checks for a match between the terminal X and current input symbol a.

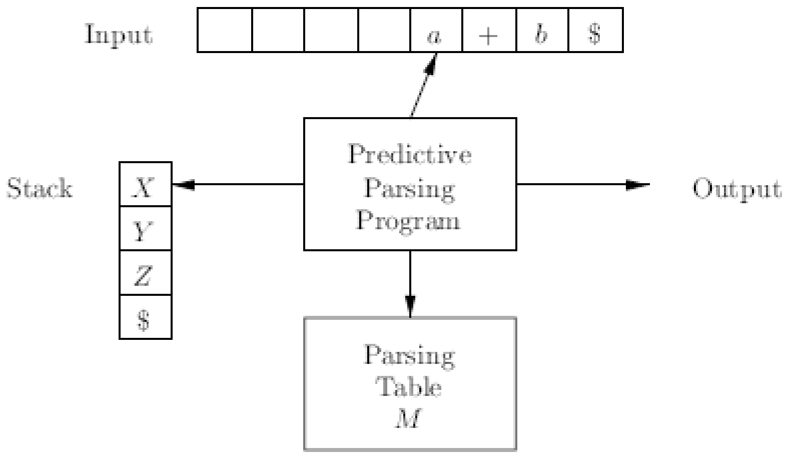


Figure 4.19: Model of a table-driven predictive parser

Algorithm 4.34 : Table-driven 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.

METHOD: Initially, the parser is in a configuration with w$ in the input buffer and the start symbol S of G on top of the stack, above $. The program in Fig. 4.20 uses the predictive parsing table M to produce a predictive parse for the input.

```
let a be the first symbol of w;
let X be the top stack symbol;
while ( X ≠ $ ) { /* stack is not empty */
        if ( X = a ) pop the stack and let a be the next symbol of w;
        else if ( X is a terminal ) error();
        else if ( M[X, a] is an error entry ) error();
        else if ( M[X, a] = X -> Y1Y2 …… Yk ) {
                output the production X -> Y1Y2 …… Yk;
                pop the stack;
                push Yk, Yk-1,……, Y1 onto the stack, with Y1 on top;
        }
        let X be the top stack symbol;
}
```

Figure 4.20: Predictive parsing algorithm

Consider grammar

E→TE'

E'→+TE'| ε

T→+FT'

T'→*FT'| ε

F→(E)|id

On input id + id * id, the nonrecursive predictive parser of Algorithm makes the following sequence of moves. These moves correspond to a leftmost derivation

Matched	Stack	Input	Action
	E$	id+id*id$	
	TE’$	id+id*id$	Output E->TE’
	FT’E’$	id+id*id$	Output T->FT’
	idT’E’$	id+id*id$	Output F->id
id	T’E’$	+id*id$	Match id
id	E’$	+id*id$	Output T->Ɛ
id	+TE’$	+id*id$	Output E’->+TE’
id+	TE$	id*id$	Match +
id+	FT’E’$	id*id$	Output T->FT’
id+	idT’E’$	id*id$	Output F-> id
id+id	T’E’$	*id$	Match id
id+id	*FT’E’$	*id$	Output T’->*FT’
id+id*	FT’E’$	*id$	Match *
id+id*	idT’E’$	id$	Output F->id
id+id*id	T’E’$	$	Match id
id+id*id	E’$	$	Output T’->Ɛ
id+id*id	$	$	Output E’->Ɛ

Note that the sentential forms in this derivation correspond to the input that has already been matched (in column MATCHED) followed by the stack contents.

Error Recovery in Predictive Parsing

An error is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol or when nonterminal A is on top of the stack, a is the next input symbol, and M[A, a] is error.

Panic Mode

Panic-mode error recovery is based on the idea of skipping over symbols on the input until a token in a selected set of synchronizing tokens appears. Its effectiveness depends on the choice of synchronizing set. Some heuristics are as follows:

1. As a starting point, place all symbols in FOLLOW(A) into the synchronizing set for nonterminal A. If we skip tokens until an element of FOLLOW(A) is seen and pop A from the stack, it is likely that parsing can continue.
2. It is not enough to use FOLLOW(A) as the synchronizing set for A. For example, if semicolons terminate statements, as in C, then keywords that begin statements may not appear in the FOLLOW set of the nonterminal representing expressions. A missing semicolon after an assignment may therefore result in the keyword beginning the next statement being skipped.
3. If we add symbols in FIRST(A) to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if a symbol in FIRST(A) appears in the input.
4. If a nonterminal can generate the empty string, then the production deriving ε can be used as a default. Doing so may postpone some error detection, but cannot cause an error to be missed. This approach reduces the number of nonterminals that have to be considered during error recovery.
5. If a terminal on top of the stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted, and continue parsing. In effect, this approach takes the synchronizing set of a token to consist of all other tokens.

If the parser looks up entry M[A,a] and finds that it is blank, then the input symbol a is skipped. If the entry is “synch," then the nonterminal on top of the stack is popped in an attempt to resume parsing. If a token on top of the stack does not match the input symbol, then we pop the token from the stack, as mentioned above.

Non terminal	Input Symbols					
	id	+	*	(	)	$
E	E->TE’			E->TE’	synch	Synch
E’		E->+TE’				E->Ɛ
T	T->FT’	synch		T->FT’	synch	Synch
T’		T’->Ɛ	T’->*FT’		T’->Ɛ	T’->Ɛ
F	F->id	synch	synch	F->(E)	synch	synch

Figure 4.22: Synchronizing tokens added to the paring table for the input id+id*id

On the erroneous input) id* +id, the parser and error recovery mechanism of Figure. 4.22 behave as in Figure. 4.23.

Stack	Input	Remarks
E$	)id*+id$	Error, skip)
E$	id*+id$	id is in FIRST(E)
TE’$	id*+id$	
FT’E’$	id*+id$	
idT’E’$	id*+id$	
T’E’$	*+id$	
*FT’E’$	*+id$	
FT’E’$	+id$	Error, M[F,+]=synch
T’E’$	+id$	F has been popped
E’$	+id$	
+TE’$	+id$	
TE’$	id$	
FT’E’$	id$	
idT’E’$	id$	
T’E’$	$	
E’$	$	
$	$	

Figure 4.23: parsing and error recovery moves made by a predictive parser

Phrase-level Recovery

Phrase-level error recovery is implemented by filling in the blank entries in the predictive parsing table with pointers to error routines. These routines may change, insert, or delete symbols on the input and issue appropriate error messages. They may also pop from the stack. Alteration of stack symbols or the pushing of new symbols onto the stack is questionable for several reasons. First, the steps carried out by the parser might then not correspond to the derivation of any word in the language at all. Second, we must ensure that there is no possibility of an infinite loop. Checking that any recovery action eventually results in an input symbol being consumed (or the stack being shortened if the end of the input has been reached) is a good way to protect against such loops.

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). It is convenient to describe parsing as the process of building parse trees, although a front end may in fact carry out a translation directly without building an explicit tree. The sequence of tree snapshots in Fig. 4.25 illustrates

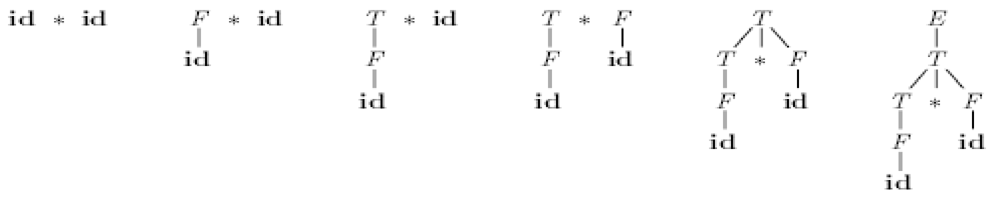


Figure 4.25: A bottom-up parse for **id * id**

a bottom-up parse of the token stream id*id, with respect to the expression grammar (4.1).

Reductions

Bottom-up parsing as the process of “reducing" a string w to the start symbol of the grammar. At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.

The snapshots in Fig. 4.25 illustrate a sequence of reductions; the grammar is the expression grammar (4.1). The reductions will be discussed in terms of the sequence of strings

id*id,F *id,T *id,T *F,T,E

The strings in this sequence are formed from the roots of all the subtrees in the snapshots. The sequence starts with the input string **id*id**. The first reduction produces **F*id** by reducing the leftmost id to F, using the production **F→ id**. The second reduction produces **T*id** by reducing F to T. Now, a choice between reducing the string **T**, which is the body of E ***T**, and the string consisting of the second **id**, which is the body of F→ id. Rather than reduce **T** to **E**, the second id is reduced to **F**, resulting in the string **T *F**. This string then reduces to T. The parse completes with the reduction of **T** to the start symbol **E**.

The goal of bottom-up parsing is therefore to construct a derivation in reverse. The following corresponds to the parse in Fig. 4.25:

E =>T =>T *F =>T *id => F * id => id * id

This derivation is in fact a rightmost derivation.

Handle Pruning

Bottom-up parsing during a left-to-right scan of the input constructs a right- most derivation in reverse. Informally, a “handle" is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.

For example, adding subscripts to the tokens id for clarity, the handles during the parse of **id1* id2** according to the expression grammar (4.1) are as in Fig. 4.26. Although T is the body of the production **E →T**, the symbol **T** is not a handle in the sentential form **T *id2**. If **T** were indeed replaced by **E**, the string **E* id2**, which cannot be derived from the start symbol E. Thus, the leftmost substring that matches the body of some production need not be a handle.

Right Sentential Form	Handle	Production
id*id	id	F->id
F*id	F	T->F
T*id	id	F->id
T*F	T*F	T->T*F
T	T	E->T
E		

Figure 4.26: Handles during a parse of id1*id2

Shift-Reduce Parsing

Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed. We use $ to mark the bottom of the stack and also the right end of the input. Initially, the stack is empty, and the string w is on the input, as follows:

STACK	**INPUT**
$	**ω $**

During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string **β** of grammar symbols on top of the stack. It then reduces **β** to the head of the appropriate production. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty:

STACK	**INPUT**
$ S	**$**

Figure 4.28 steps through the actions a shift-reduce parser might take in parsing the input string **id1*id2** according to the expression grammar (4.1).

Stack	Input Buffer	Parsing Action
$	Id1*id2$	Shift
$id1	*id2$	Reduce F->id
$F	*id2$	Reduce T->F
$T	*id2$	Shift
$T*	id2$	Shift
$T* id2	$	Reduce by F->id
$T*F	$	Reduce by T->T*F
$T	$	Reduce by E->T
$E	$	Accept

Figure 4.28: Configurations of a shift-reduce parser on input id1*id2

While the primary operations are shift and reduce, there are actually four possible actions a shift-reduce parser can make: (1) shift, (2) reduce, (3) accept, and (4) error.

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.

The use of a stack in shift-reduce parsing is justified by an important fact: the handle will always eventually appear on top of the stack, never inside. This fact can be shown by considering the possible forms of two successive steps in any rightmost derivation.

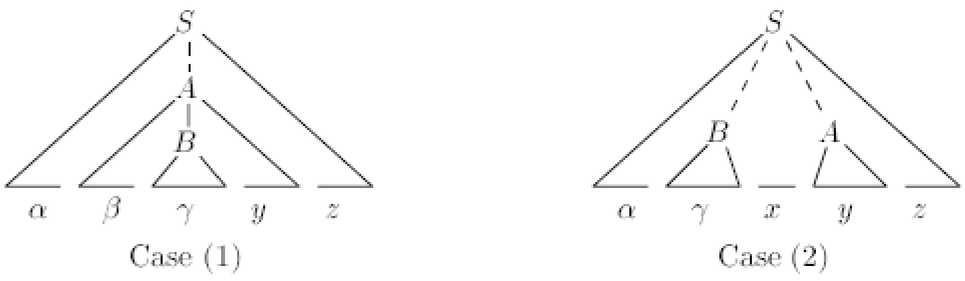


Figure 4.29: Cases for two successive steps of a rightmost derivation

Consider case (1) in reverse, where a shift-reduce parser has just reached the configuration

STACK	INPUT
$\$\alpha\beta\gamma$	$yz\$$

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

$\$\alpha\beta By$	$z\$$

with the handle βBy on top of the stack, and it gets reduced to A.

Now consider case (2). In configuration

$αγ	xyz$

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:

$αBxy	z$

In both cases, after making a reduction the parser had to shift zero or more symbols to get the next handle onto the stack. It never had to go into the stack to find the handle.

Conflicts During Shift-Reduce Parsing

There are context-free grammars for which shift-reduce parsing cannot be used. Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack and also the next k input symbols, cannot decide whether to shift or to reduce (a shift/reduce conflict), or cannot decide, which of several reductions to make (a reduce/reduce conflict).

An ambiguous grammar can never be LR. For example, consider the dangling-else grammar

stmt-> if expr the stmt

| if expr then stmt else stmt

| other

If we have a shift-reduce parser in configuration

STACK	INPUT
…… if expr then stmt	else……$

Introduction to LR Parsing: Simple LR

The most prevalent type of bottom-up parser today is based on a concept called LR(k) parsing; the “L" is for left-to-right scanning of the input, the “R" for constructing a rightmost derivation in reverse, and the k for the number of input symbols of lookahead that are used in making parsing decisions.

Why LR Parsers?

A grammar for which we can construct a parsing table using one of the methods in this section and the next is said to be an LR grammar. Intuitively, for a grammar to be LR it is sufficient that a left-to-right shift-reduce parser be able to recognize handles of right-sentential forms when they appear on top of the stack.

LR parsing is attractive for a variety of reasons:

- LR parsers can be constructed to recognize virtually all programming- language constructs for which context-free grammars can be written. Non- LR context-free grammars exist, but these can generally be avoided for typical programming-language constructs.
- The LR-parsing method is the most general nonbacktracking shift-reduce parsing method known, yet it can be implemented as efficiently as other, more primitive shift-reduce
- An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.
- 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 or LL methods. For a grammar to be LR(k), we must be able to recognize the occurrence of the right side of a production in a right-sentential form, with k input symbols of lookahead.

The principal drawback of the LR method is that it is too much work to construct an LR parser by hand for a typical programming-language grammar. If the grammar contains ambiguities or other constructs that are difficult to parse in a left-to-right scan of the input, then the parser generator locates these constructs and provides detailed diagnostic messages.

An LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in a parse. States represent sets of “items." An LR(0) item (item for short) of a grammar G is a production of G with a dot at some position of the body. Thus, production A→XYZ yields the four items

A→•XYZ

A→X•YZ

A→XY•Z

A→XYZ•

The production **A→ε** generates only one item, A→•.

For example, the item A→•XY Z indicates that a string derivable from **XY Z** next on the input. Item **A→X•YZ** indicates that on the input a string derivable from X and that to see a string derivable from **Y Z**. Item **A →XY Z•** indicates that the body **XY Z** and that it may be time to reduce **XY Z** to **A.**

One collection of sets of LR(0) items, called the ***canonical LR(0) collection***, provides the basis for constructing a deterministic finite automaton that is used to make parsing decisions.

Closure of Item Sets

If I is a set of items for a grammar G, then CLOSURE(I) is the set of items constructed from I by the two rules:

1. Initially, add every item in I to CLOSURE(I).
2. If A-> α **.** Bβ is in CLOSURE(I) and B-> γ is aproduction, then add the item B-> . γ to CLOSURE(I), if it is not already there. Apply this rule until no more new items can be added to CLOSURE(I).

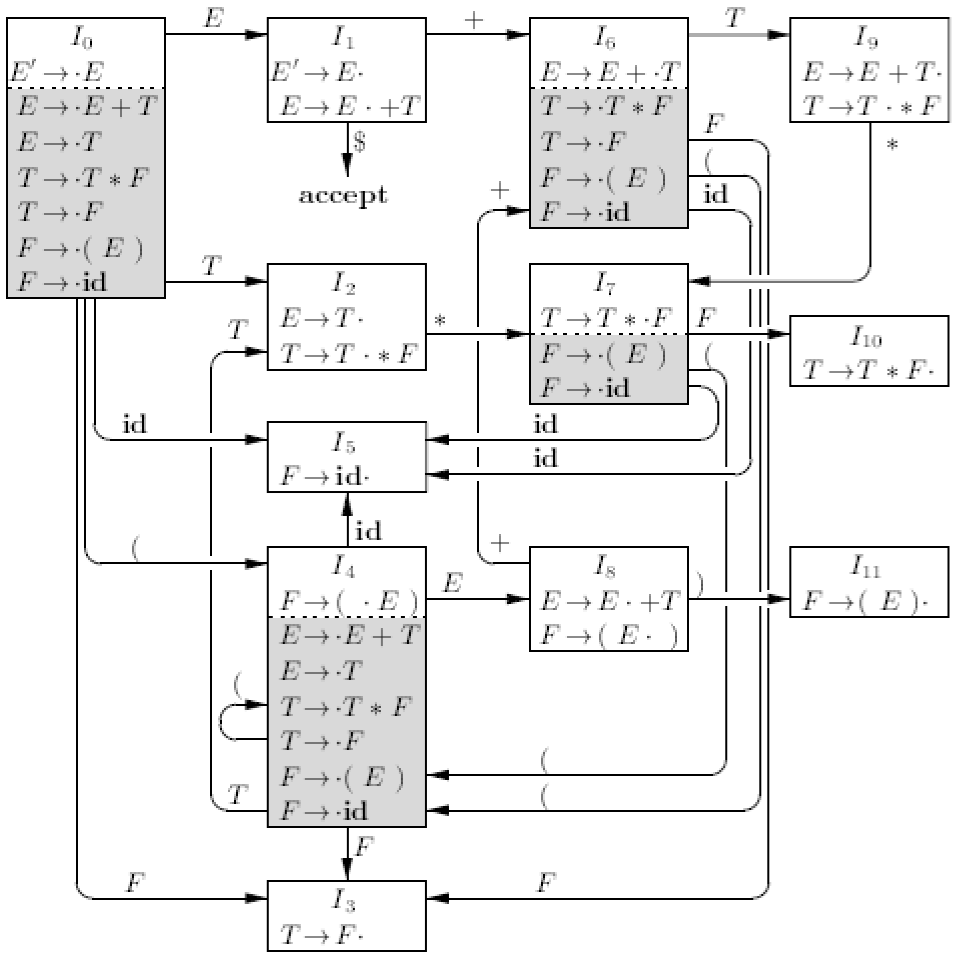


Figure 4.31: LR(0) automaton for the expression grammar (4.1)

Example 4.40: Consider the augmented expression grammar:

$$\begin{array}{lcl} E' & \to & E \\ E & \to & E + T \ | \ T \\ T & \to & T * F \ | \ F \\ F & \to & (E) \ | \ \mathbf{id} \end{array}$$

If I is the set of one item $\{[E' \to \cdot E]\}$, then CLOSURE(I) contains the set of items I_0 in Fig. 4.31.

```
SetOfItems CLOSURE(I) {
        J = I;
        repeat
                for ( each item A → α·Bβ in J )
                        for ( each production B → γ of G )
                                if ( B → ·γ is not in J )
                                        add B → ·γ to J;
        until no more items are added to J on one round;
        return J;
}
```

Figure 4.32: Computation of CLOSURE

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

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

The Function GOTO

The second useful function is GOTO(I, X) where I is a set of items and X is a grammar symbol. GOTO(I, X) is defined to be the closure of the set of all items $[A \to \alpha X \cdot \beta]$ such that $[A \to \alpha \cdot X\beta]$ is in I. Intuitively, the GOTO function is used to define the transitions in the LR(0) automaton for a grammar. The states of the automaton correspond to sets of items, and GOTO(I, X) specifies the transition from the state for I under input X.

Example 4.41: If I is the set of two items $\{[E' \to E\cdot],\ [E \to E\cdot + T]\}$, then GOTO$(I, +)$ contains the items

$$\begin{array}{l} E \to E + \cdot T \\ T \to \cdot T * F \\ T \to \cdot F \\ F \to \cdot(E) \\ F \to \cdot\mathbf{id} \end{array}$$

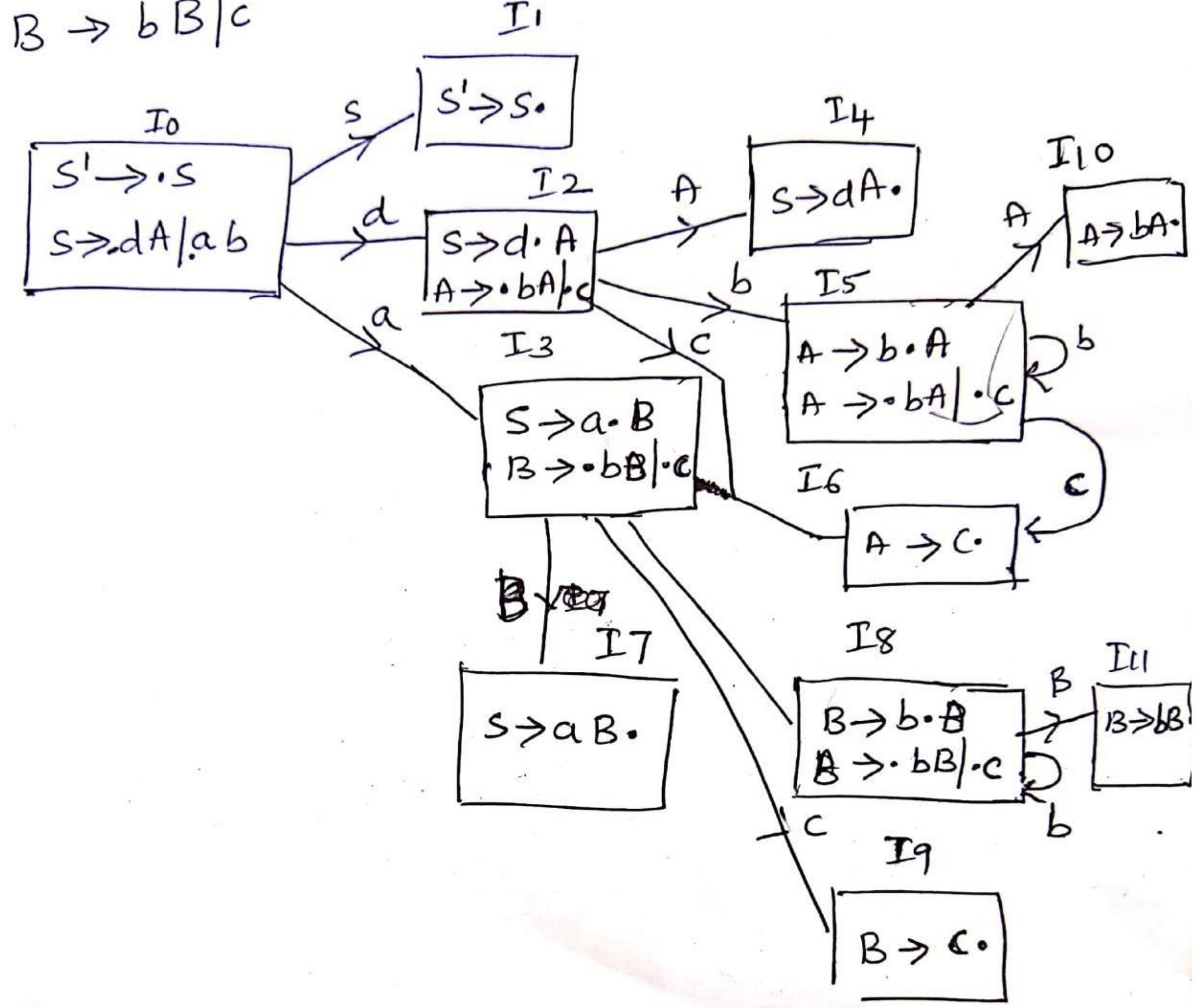
```
void items(G') {
        C = {CLOSURE({[S' → ·S]})};
        repeat
                for ( each set of items I in C )
                        for ( each grammar symbol X )
                                if ( GOTO(I, X) is not empty and not in C )
                                        add GOTO(I,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

Figure 4.33: Computation of the canonical collection of sets of LR(0) items

$S \rightarrow dA \mid aB$

$A \rightarrow bA \mid c$

$B \rightarrow bB \mid c$



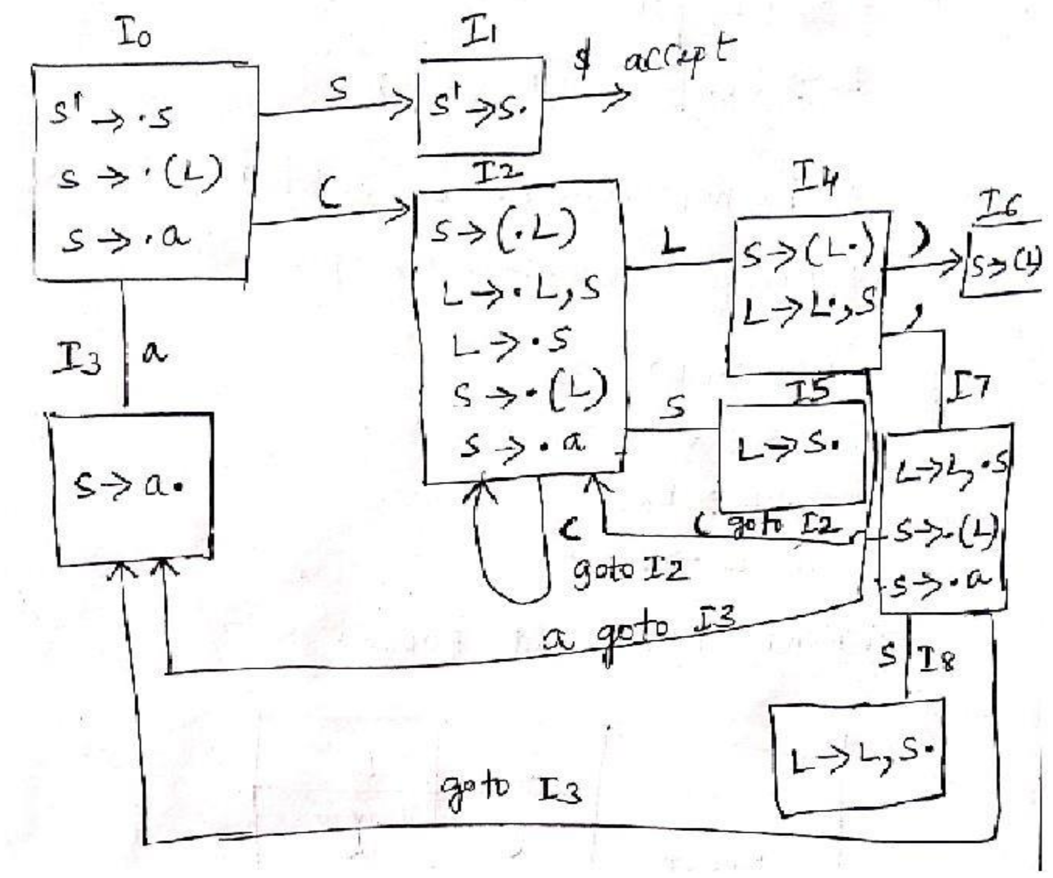
$S' \rightarrow S$

$S \rightarrow (L)$

$L \rightarrow L, S$

$L \rightarrow S$

Construct closure items set:



The LR-Parsing Algorithm

A schematic of an LR parser is shown in Fig. 4.35. It consists of an input, an output, a stack, a driver program, and a parsing table that has two parts (ACTION and GOTO). The driver program is the same for all LR parsers; only the parsing table changes from one parser to another. The parsing program reads characters from an input buffer one at a time. Where a shift-reduce parser would shift a symbol, an LR parser shifts a state.

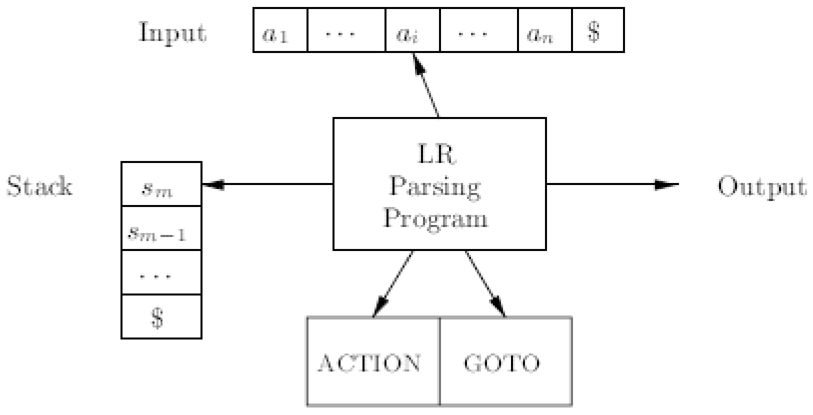


Figure 4.35: Model of an LR parser

The stack holds a sequence of states, s0,s1 .... s_m, where s_m is on top. In the SLR method, the stack holds states from the LR(0) automaton; the canonical- LR and LALR methods are similar. By construction, each state has a corresponding grammar symbol. Recall that states correspond to sets of items, and that there is a transition from state i to state j if GOTO(I_i,X) = I_j . All transitions to state j must be for the same grammar symbol X.

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 ->β. The action of the parser effectively reduces β on the top of the stack to head A.

(c) Accept. The parser accepts the input and finishes parsing.

(d) Error. The parser discovers an error in its input and takes some corrective action2. We extend the GOTO function, de_ned on sets of items, to states: if

GOTO[Ii,A] = Ij , then GOTO also maps a state i and a nonterminal A to state j.

Behavior of the LR Parser

1. If ACTION[sm, ai] = shift s, the parser executes a shift move; it shifts the next state s onto the stack, entering the configuration

(s0s1…… sms, ai+1 …… an$)

The symbol ai need not be held on the stack, since it can be recovered from s, if needed (which in practice it never is). The current input symbol is now a_{i+1}.

2. If ACTION[sm; ai] = reduce A ! _, then the parser executes a reduce move, entering the configuration

$(s_0s_1\ldots\ldots s_{m-r}s; a_ia_{i+1}\ldots\ldots a_n\$)$

where r is the length of β, and s = GOTO[s_{m-r},A]. Here the parser first popped r state symbols off the stack, exposing state s_{m-r}. The parser then pushed s, the entry for GOTO[s_{m-r},A], onto the stack. The current input symbol is not changed in a reduce move. For the LR parsers we shall construct, X_{m-r+1} ......X_m, the sequence of grammar symbols corresponding to the states popped off the stack, will always match β, the right side of the reducing production. The output of an LR parser is generated after a reduce move by executing the semantic action associated with the reducing production. For the time being, we shall assume the output consists of just printing the reducing production.

3. If ACTION[sm; ai] = accept, parsing is completed.

4. If ACTION[sm; ai] = error, the parser has discovered an error and calls an error recovery routine.

Algorithm 4.44 : LR-parsing algorithm.

INPUT: An input string w and an LR-parsing table with functions ACTION and GOTO for a grammar G.

OUTPUT: If w is in L(G), the reduction steps of a bottom-up parse for w; otherwise, an error indication.

METHOD: Initially, the parser has s0 on its stack, where s0 is the initial state, and w$ in the input bu_er. The parser then executes the program in Fig. 4.36.

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

Figure 4.36: LR-parsing program

Figure 4.37 shows the ACTION and GOTO functions of an LR-parsing table for the expression grammar (4.1), repeated here with the productions numbered:

1. E-> E + T	4. T->F
2. E-> T	5. F->(E)
3. T->T*F	6. F->id

The codes for the actions are:

1. si means shift and stack state i,
2. rj means reduce by the production numbered j,
3. acc means accept,
4. blank means error.

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			

Figure 4.37: parsing table for expression grammar

On input id *id + id, the sequence of stack and input contents is shown in Fig. 4.38. Also shown for clarity, are the sequences of grammar symbols corresponding to the states held on the stack.

	Stack	Symbol	Input	Action
1	0		Id*id+id$	Shift
2	0 5	id	*id+id$	Reduce by F->id
3	0 3	F	*id+id$	Reduce by T->F
4	0 2	T	*id+id$	Shift
5	0 2 7	T*	id+id$	Shift

6	0 2 7 5	T*id	+id$	Reduce by F->id
7	0 2 7 10	T*F	+id$	Reduce by T->T*F
8	0 2	T	+id$	Reduce by E->T
9	0 1	E	+id$	Shift
10	0 1 6	E+	id$	Shift
11	0 1 6 5	E+id	$	Reduce by F->id
12	0 1 6 3	E+F	$	Reduce by T->F
13	0 1 6 9	E+T	$	Reduce by E->E+T
14	0 1	E	$	Accept

Figure 4.38: Moves of an LR parser on the input id*id+id

Constructing SLR-Parsing Tables

The SLR method for constructing parsing tables is a good starting point for studying LR parsing. The ACTION and GOTO entries in the parsing table are then constructed using the following algorithm. It requires us to know FOLLOW(A) for each nonterminal A of a grammar.

Algorithm 4.46 : Constructing an SLR-parsing table.

INPUT: An augmented grammar G^1.

OUTPUT: The SLR-parsing table functions ACTION and GOTO for G^1.

METHOD:

1. Construct C = {I_0, $I_{1,......}$ I_n}, the collection of sets of LR(0) items for G^1.

2. State i is constructed from Ii . The parsing actions for state i are determined as follows:

(a) If [A ->α̥aβ] is in Ii and GOTO(Ii, a) = Ij , then set ACTION[I, a] to “shift j." Here a must be a terminal.

(b) If [A ->α̥] is in Ii, then set ACTION[I, a] to “reduce A ->α̥" for all a in FOLLOW(A); here A may not be S0.

(c) If [S^1 -> S.] is in Ii, then set 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.

The goto transitions for state i are constructed for all nonterminals A using the rule:

If GOTO(Ii;A) = Ij , then GOTO[i;A] = j.

4. All entries not de_ned 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^1 -> .S].

Let us construct the SLR table for the augmented expression grammar. The canonical collection of sets of LR(0) items for the grammar was shown in Fig. 4.31. First consider the set of items I0:

E^1->. E

E -> . E + T

E -> . T

T -> . T * F

T -> . F

F -> . (E)

F -> . id

The item F -> . (E) gives rise to the entry ACTION[0, (] = shift 4, and the item F -> . id to the entry ACTION[0, id] = shift 5. Other items in I0 yield no actions. Now consider I1:

E^1 -> E .

E -> E. + T

The first item yields ACTION[1, $] = accept, and the second yields ACTION[1, +]
= shift 6. Next consider I2:

E -> T.

T -> T.* F

Since FOLLOW(E) = {$,+,)}, the first item makes
ACTION[2, $] = ACTION[2, +] = ACTION[2,)] = reduce E -> T

The second item makes ACTION[2,*]] = shift 7. Continuing in this fashion we obtain the ACTION and GOTO tables that were shown in Fig. 4.31. In that figure, the numbers of productions in reduce actions are the same as the order in which they appear in the original grammar (4.1). That is, E -> E + T is number 1, E -> T is 2, and so on.

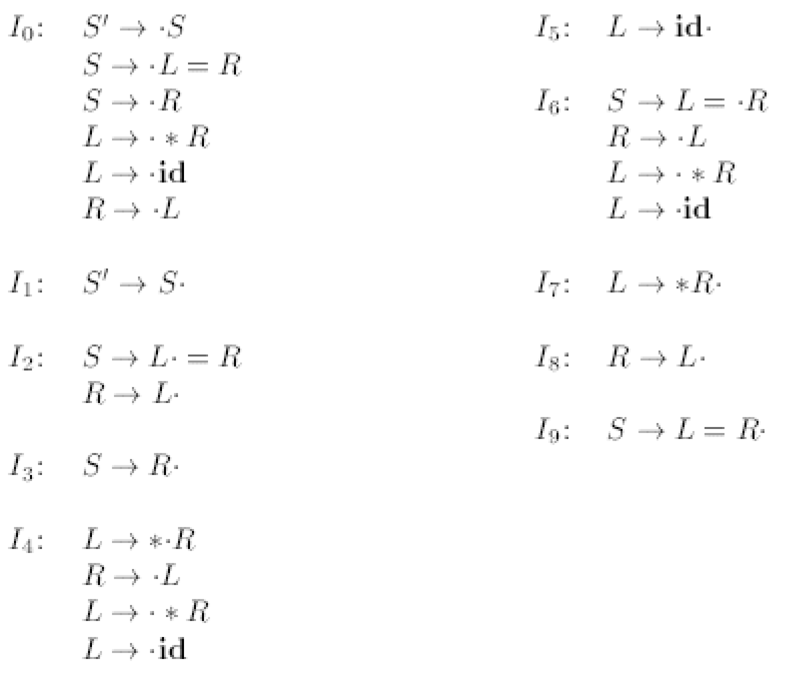
Every SLR(1) grammar is unambiguous, but there are many unambiguous grammars that are not SLR(1). Consider the grammar with productions

S -> L = R | R

L -> *R | id $\longrightarrow$ (4.49)

R -> L

Think of L and R as standing for l-value and r-value, respectively, and * as an operator indicating “contents of."5 The canonical collection of sets of LR(0) items for grammar (4.49) is shown in Fig. 4.39.



I_0: $S' \to \cdot S$
$S \to \cdot L = R$
$S \to \cdot R$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$
$R \to \cdot L$

I_1: $S' \to S \cdot$

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

I_3: $S \to R \cdot$

I_4: $L \to * \cdot R$
$R \to \cdot L$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$

I_5: $L \to \mathbf{id} \cdot$

I_6: $S \to L = \cdot R$
$R \to \cdot L$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$

I_7: $L \to * R \cdot$

I_8: $R \to L \cdot$

I_9: $S \to L = R \cdot$

Figure 4.39: Canonical LR(0) collection for grammar (4.49)

1. Consider the grammar:

A->(A) | a

1) Find LR(0) items
2) Construct LR(0) automata
3) Construct SLR parsing table
4) Parse the input string : (((a))).

ANSWER:

Augmented grammar is:

A'-> A

A->(A)

A-> a

LR(0) items:

I_0:

A'->**.**A

A->.(A)

A->.a

I_1: GOTO(I_0,A)

A'->A.

I2: GOTO(I_0,( )

A->(.A)

A->.(A)

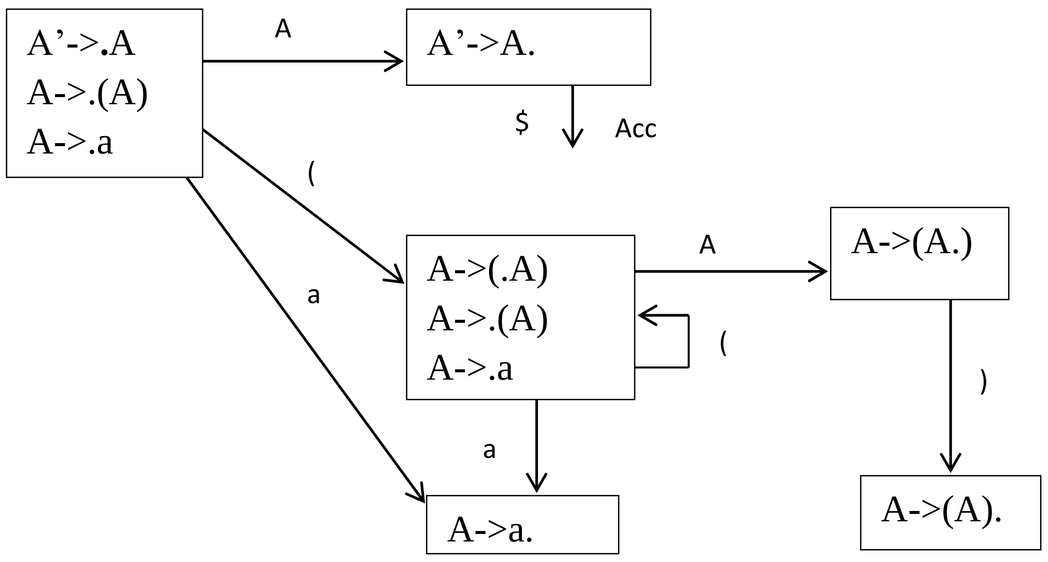
A->.a

I_3: GOTO(I_0,a)
A->a.

I_4: GOTO(I_2,A)
A->(A.)

I_5: GOTO(I_4,))
A->(A).

LR(0) automata:



FIRST(A')={(,a}   FOLLOW(A')={$}

FIRST(A)={(,a}   FOLLOW(A)={),$}

1. A->(A)
2. A->a

SLR parsing table:

State	Action				GOTO
	(	)	a	$	A
0	s2		s3		1
1				acc	
2	s2		s3		4
3		r2		r2	
4		s5			
5		r1		r1	

The grammar is SLR.

	Stack	Symbol	Input	Action
1	0		(((a)))$	Shift
2	0 2	(	((a)))$	Shift
3	0 2 2	((	(a)))$	Shift
4	0 2 2 2	(((	a)))$	Shift
5	0 2 2 2 3	(((a	)))$	Reduce by A->a
6	0 2 2 2 4	(((A	)))$	Shift
7	0 2 2 2 4 5	(((A)	))$	Reduce by A->(A)
8	0 2 2 4	((A	))$	Shift
9	0 2 2 4 5	((A)	)$	Reduce by A->(A)
10	0 2 4	(A	)$	Shift
11	0 2 4 5	(A)	$	Reduce by A->(A)
12	0 1	A	$	Accept

The string (((a))) is accepted.

Operator Precedence Grammar

A grammar that satisfies the following 2 conditions is called as **Operator Precedence Grammar**–

- There exists no production rule which contains ε on its RHS.
- There exists no production rule which contains two non-terminals adjacent to each other on its RHS.
- It represents a small class of grammar.
- But it is an important class because of its widespread applications.

Operator Precedence Parser-

A parser that reads and understand an operator precedence grammar is called as **Operator Precedence Parser**. In operator precedence parsing,we define precedence relations between every pair of terminal symbols. Second , we construct an operator precedence table.

Defining Precedence Relations

The precedence relations are defined using the following rules-

Rule-01:

- If precedence of b is higher than precedence of a, then we define a < b
- If precedence of b is same as precedence of a, then we define a = b
- If precedence of b is lower than precedence of a, then we define a > b

Rule-02:

- An identifier is always given the higher precedence than any other symbol.
- $ symbol is always given the lowest precedence.

Rule-03:

If two operators have the same precedence, then we go by checking their associativity.

Parsing A Given String-

The given input string is parsed using the following steps-

Step-01:

Insert the following-

- $ symbol at the beginning and ending of the input string.
- Precedence operator between every two symbols of the string by referring the operator precedence table.

Step-02:

- Start scanning the string from LHS in the forward direction until > symbol is encountered.
- Keep a pointer on that location.

Step-03:

- Start scanning the string from RHS in the backward direction until < symbol is encountered.
- Keep a pointer on that location.

Step-04:

- Everything that lies in the middle of < and > forms the handle.
- Replace the handle with the head of the respective production.

Step-05:

Keep repeating the cycle from Step-02 to Step-04 until the start symbol is reached.

Advantages **of operator precedence parsing are**

- The implementation is very easy and simple.
- The parser is quite powerful for expressions in programming languages.

Disadvantages-

- The handling of tokens known to have two different precedence becomes difficult.
- Only small class of grammars can be parsed using this parser.

Important Note-

In practice, operator precedence table is not stored by the operator precedence parsers. it occupies the large space. Instead, operator precedence parsers are implemented in a very unique style. They are implemented using operator precedence functions.

Operator Precedence Functions- Precedence functions perform the mapping of terminal symbols to the integers. To decide the precedence relation between symbols, a numerical comparison is performed. It reduces the space complexity to a large extent.