

UNIT –II

Syntax Analysis:-The Role of a parser, Context free Grammars, Writing A grammar, top down parsing bottom up parsing, Introduction to Lr Parser.

UNIT -2

SYNTAX ANALYSIS

2.1 ROLE OF THE PARSER

Parser obtains a string of tokens from the lexical analyzer and verifies that it can be generated by the language for the source program. The parser should report any syntax errors in an intelligible fashion. The two types of parsers employed are:

- 1.Top down parser: which build parse trees from top(root) to bottom(leaves)
- 2.Bottom up parser: which build parse trees from leaves and work up the root.

Therefore there are two types of parsing methods– top-down parsing and bottom-up parsing

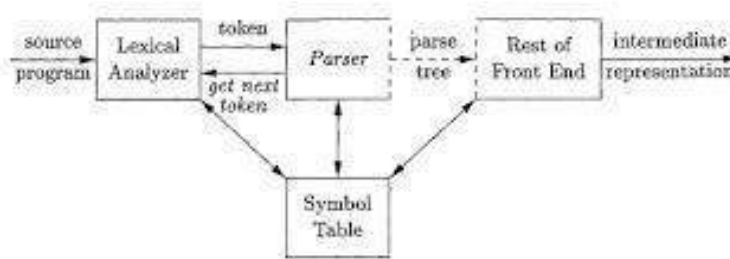


Figure 4.1: Position of parser in compiler model

2.2 Context free Grammars(CFG)

CFG is used to specify the syntax of a language. A grammar naturally describes the hierarchical structure of most programming language constructs.

Formal Definition of Grammars

A context-free grammar has four components:

1. A set of terminal symbols, sometimes referred to as "tokens." The terminals are the elementary symbols of the language defined by the grammar.
2. A set of nonterminals, sometimes called "syntactic variables." Each non-terminal represents a set of strings of terminals, in a manner we shall describe.
3. A set of productions, where each production consists of a nonterminal, called the head or left side of the production, an arrow, and a sequence of terminals and/or nonterminals, called the *body* or *right side* of the production. The intuitive intent of a production is to specify one of the written forms of a construct; if the head nonterminal represents a construct, then the body represents a written form of the construct.

4. A designation of one of the nonterminals as the *start* symbol.

Production is for a nonterminal if the nonterminal is the head of the production. A string of terminals is a sequence of zero or more terminals. The string of zero terminals, written as ϵ , is called the empty string.

Derivations

A grammar derives strings by beginning with the start symbol and repeatedly replacing a nonterminal by the body of a production for that nonterminal. The terminal strings that can be derived from the start symbol form the *language* defined by the grammar.

Leftmost and Rightmost Derivation of a String

- **Leftmost derivation** – A leftmost derivation is obtained by applying production to the leftmost variable in each step.
- **Rightmost derivation** – A rightmost derivation is obtained by applying production to the rightmost variable in each step.
- **Example**

Let any set of production rules in a CFG be

$$X \rightarrow X+X \mid X*X \mid X \mid a$$

over an alphabet $\{a\}$.

The leftmost derivation for the string "**a+a*a**" is

$$X \rightarrow X+X \rightarrow a+X \rightarrow a+X*X \rightarrow a+a*X \rightarrow a+a*a$$

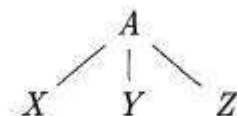
The rightmost derivation for the above string "**a+a*a**" is

$$X \rightarrow X*X \rightarrow X*a \rightarrow X+X*a \rightarrow X+a*a \rightarrow a+a*a$$

Derivation or Yield of a Tree

The derivation or the yield of a parse tree is the final string obtained by concatenating the labels of the leaves of the tree from left to right, ignoring the Nulls. However, if all the leaves are Null, derivation is Null.

parse tree pictorially shows how the start symbol of a grammar derives a string in the language. If nonterminal A has a production $A \rightarrow XYZ$, then a parse tree may have an interior node labeled A with three children labeled X, Y, and Z, from left to right:



Given a context-free grammar, a *parse tree* according to the grammar is a tree with the following properties:

1. The root is labeled by the start symbol.
2. Each leaf is labeled by a terminal or by ϵ .
3. Each interior node is labeled by a nonterminal

If A is the nonterminal labeling some interior node and X_1, X_2, \dots, X_n are the labels of the children of that node from left to right, then there must be a production $A \rightarrow X_1 X_2 \dots X_n$. Here, X_1, X_2, \dots, X_n , each stand for a symbol that is either a terminal or a nonterminal. As a special case, if $A \rightarrow c$ is a production, then a node labeled A may have a single child labeled c .

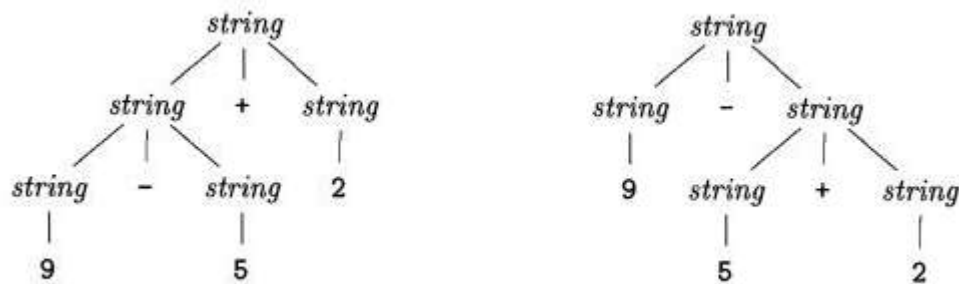
Ambiguity

A grammar can have more than one parse tree generating a given string of terminals. Such a grammar is said to be *ambiguous*. To show that a grammar is ambiguous, all we need to do is find a terminal string that is the yield of more than one parse tree. Since a string with more than one parse tree usually has more than one meaning, we need to design unambiguous grammars for compiling applications, or to use ambiguous grammars with additional rules to resolve the ambiguities.

Example :: Suppose we used a single nonterminal *string* and did not distinguish between digits and lists,

string \rightarrow *string* + *string* | *string* - *string* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Fig. shows that an expression like 9-5+2 has more than one parse tree with this grammar. The two trees for 9-5+2 correspond to the two ways of parenthesizing the expression: (9-5) + 2 and 9- (5+2). This second parenthesization gives the expression the unexpected value 2 rather than the customary value 6.



Two parse trees for 9-5+2

Verifying the language generated by a grammar

The set of all strings that can be derived from a grammar is said to be the language generated from that grammar. A language generated by a grammar G is a subset formally defined by

$$L(G) = \{ W | W \in \Sigma^*, S \Rightarrow_G W \}$$

If $L(G_1) = L(G_2)$, the Grammar G_1 is equivalent to the Grammar G_2 .

Example

If there is a grammar

$G: N = \{S, A, B\} \quad T = \{a, b\} \quad P = \{S \rightarrow AB, A \rightarrow a, B \rightarrow b\}$

Here S produces AB , and we can replace A by a , and B by b . Here, the only accepted string is ab , i.e.,

$L(G) = \{ab\}$

2.3 Writing a grammar

A *grammar* consists of a number of *productions*. Each production has an abstract symbol called a *nonterminal* as its *left-hand side*, and a sequence of one or more nonterminal and *terminal* symbols as its *right-hand side*. For each grammar, the terminal symbols are drawn from a specified *alphabet*.

Starting from a sentence consisting of a single distinguished nonterminal, called the *goal symbol*, a given context-free grammar specifies a *language*, namely, the set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.

There are **four categories** in writing a grammar :

1. Lexical Vs Syntax Analysis
2. Eliminating ambiguous grammar.
3. Eliminating left-recursion
4. Left-factoring.

Each parsing method can handle grammars only of a certain form hence, the initial grammar may have to be rewritten to make it parsable

1. Lexical Vs Syntax Analysis

Reasons for using the regular expression to define the lexical syntax of a language

- a) Regular expressions provide a more concise and easier to understand notation for tokens than grammars.
- b) The lexical rules of a language are simple and to describe them, we donot need notation as powerful as grammars.
- c) Efficient lexical analyzers can be constructed automatically from RE than from grammars.
- d) Separating the syntactic structure of a language into lexical and nonlexical parts provides a convenient way of modularizing the front end into two manageable-sized components.

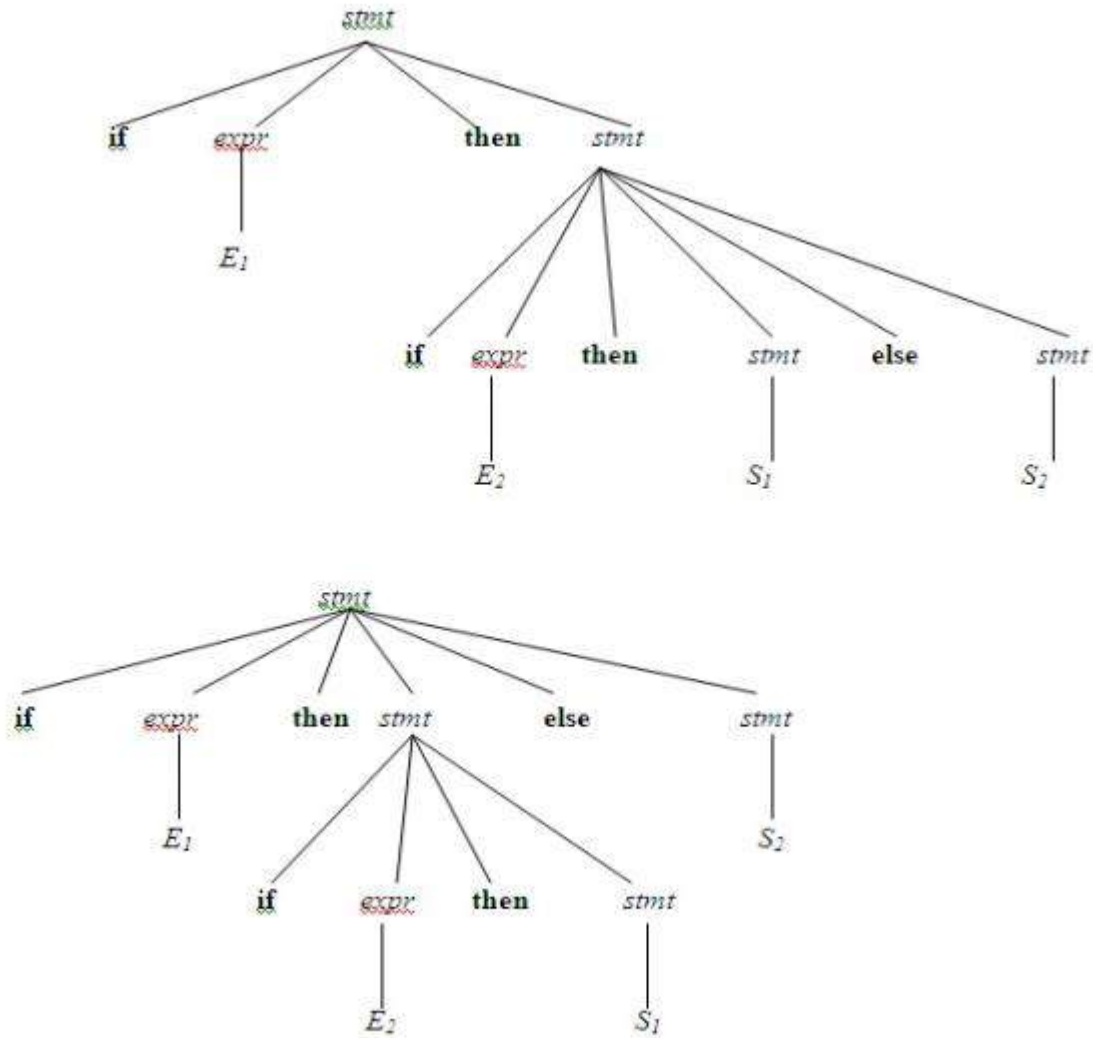
2. Eliminating ambiguous grammar.

Ambiguity of the grammar that produces more than one parse tree for leftmost or rightmost derivation can be eliminated by re-writing the grammar.

Consider this example,

G: *stmt* → **if** *expr* **then** *stmt*
 if *expr* **then** *stmt* **else** *stmt*
 other

This grammar is ambiguous since the string **if E1 then if E2 then S1 else S2** has the following two parse trees for leftmost derivation



Two parse trees for an ambiguous sentence

The general rule is “Match each else with the closest unmatched then. This disambiguating rule can be used directly in the grammar,

To eliminate ambiguity, the following grammar may be used:

$stmt \rightarrow matched \mid unmatchedstmt$
 $matched \rightarrow \text{if } expr \text{ then } matched \text{ else } matchedstmt \mid \text{other}$
 $unmatched \rightarrow \text{if } expr \text{ then } stmt \mid \text{if } expr \text{ then } matched \text{ else } unmatchedstmt$

3. Eliminating left-recursion

CSE Dept., Sir CRR COE.

Because we try to generate a leftmost derivation by scanning the input from left to right, grammars of the form $A \rightarrow A x$ may cause endless recursion. Such grammars are called left-recursive and they must be transformed if we want to use a top-down parser.

■ A grammar is left recursive if for a non-terminal A, there is a derivation $A \Rightarrow^+ A\alpha$

■ To eliminate direct left recursion replace

$$1) \quad A \rightarrow A\alpha | \beta \quad \text{with} \quad A' \rightarrow \alpha A' | \epsilon$$

$$2) \quad A \rightarrow A\alpha_1 | A\alpha_2 | \dots | A\alpha_m | \beta_1 | \beta_2 | \dots | \beta_n$$

with

$$A \rightarrow \beta_1 B | \beta_2 B | \dots | \beta_n B$$
$$B \rightarrow \alpha_1 B | \alpha_2 B | \dots | \alpha_m B | \epsilon$$

4. Left-factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. When it is not clear which of two alternative productions to use to expand a non-terminal A, we can rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

■ Consider $S \rightarrow \text{if } E \text{ then } S \text{ else } S \mid \text{if } E \text{ then } S$

■ Which of the two productions should we use to expand non-terminal S when the next token is if?

We can solve this problem by factoring out the common part in these rules.

$$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \dots | \alpha\beta_n | \gamma$$

becomes

$$A \rightarrow \alpha B | \gamma$$
$$B \rightarrow \beta_1 | \beta_2 | \dots | \beta_n$$

Consider the grammar, $G : S \rightarrow iEtS \mid iEtSeS \mid a$

$$E \rightarrow b$$

Left factored, this grammar becomes

$$S \rightarrow iEtSS' \mid a$$

$$S' \rightarrow eS \mid \epsilon$$

$$E \rightarrow b$$

2.4 PARSING

It is the process of analyzing a continuous stream of input in order to determine its grammatical structure with respect to a given formal grammar.

Types of parsing:

1. Top down parsing
2. Bottom up parsing

Top-down parsing : A parser can start with the start symbol and try to transform it to the input string.

Example : LL Parsers.

Bottom-up parsing : A parser can start with input and attempt to rewrite it into the start symbol.

Example : LR Parsers.

2.5 TOP-DOWN PARSING

It can be viewed as an attempt to find a left-most derivation for an input string or an attempt to construct a parse tree for the input starting from the root to the leaves.

Types of TOP-DOWN PARSING

1. Recursive descent parsing
2. Predictive parsing

2.5.1 RECURSIVE DESCENT PARSING

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as **predictive parsing**.

This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.

This parsing method may involve backtracking.

Example for :backtracking

Consider the grammar $G : S \rightarrow cAd$

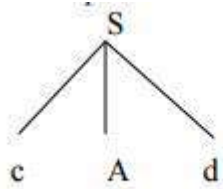
$A \rightarrow ab|a$

and the input string $w=cad$.

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

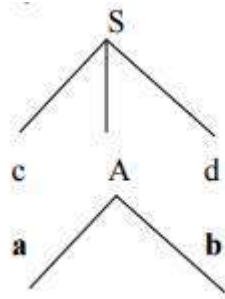
Step1:

Initially create a tree with single node labeled S. An input pointer points to 'c', the first symbol of w. Expand the tree with the production of S.



Step2:

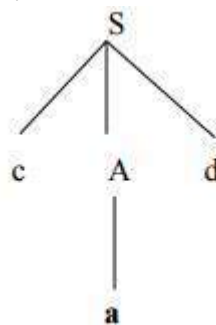
The leftmost leaf 'c' matches the first symbol of w, so advance the input pointer to the second symbol of w 'a' and consider the next leaf 'A'. Expand A using the first alternative.



Step3:

The second symbol 'a' of w also matches with second leaf of tree. So advance the input pointer to third symbol of w 'd'. But the third leaf of tree is b which does not match with the input symbol **d**. Hence discard the chosen production and reset the pointer to second **backtracking**.

Step4: Now try the second alternative for A.



Now we can halt and announce the successful completion of parsing.

2.5.2 Predictive parsing

It is possible to build a nonrecursive predictive parser by maintaining a stack explicitly, rather than implicitly via recursive calls. The key problem during predictive parsing is that of determining the production to be applied for a nonterminal. The nonrecursive parser in figure looks up the production to be applied in parsing table. In what follows, we shall see how the table can be constructed directly from certain grammars.

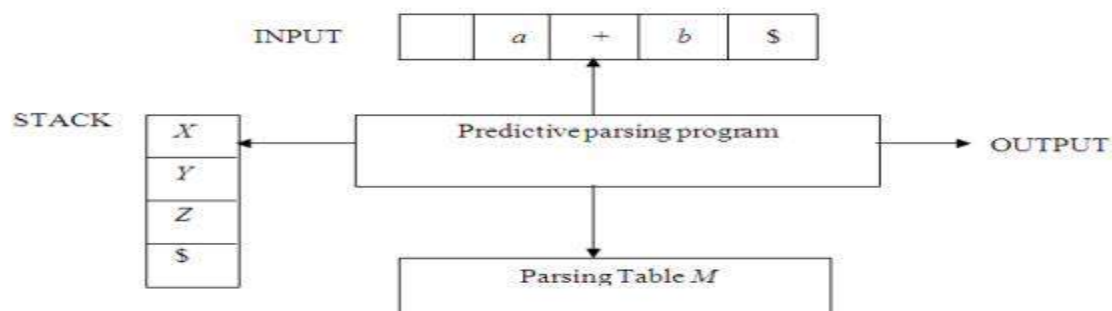


Fig. 2.4 Model of a nonrecursive predictive parser

A table-driven predictive parser has an input buffer, a stack, a parsing table, and an output stream. The input buffer contains the string to be parsed, followed by \$, a symbol used as a right endmarker to indicate the end of the input string. The stack contains a sequence of grammar symbols

CSE Dept.,Sir CRR COE.

with \$ on the bottom, indicating the bottom of the stack. Initially, the stack contains the start symbol of the grammar on top of \$. The parsing table is a two dimensional array $M[A,a]$ where A is a nonterminal, and a is a terminal or the symbol \$. The parser is controlled by a program that behaves as follows. The program considers X, the symbol on the top of the stack, and a, the current input symbol. These two symbols determine the action of the parser. There are three possibilities.

- 1 If $X = a = \$$, the parser halts and announces successful completion of parsing.
- 2 If $X = a \neq \$$, the parser pops X off the stack and advances the input pointer to the next input symbol.
- 3 If X is a nonterminal, the program consults entry $M[X,a]$ of the parsing table M. This entry will be either an X-production of the grammar or an error entry. If, for example, $M[X,a] = \{X \rightarrow UVW\}$, the parser replaces X on top of the stack by WVU(with U on top). As output, we shall assume that the parser just prints the production used; any other code could be executed here. If $M[X,a] = \text{error}$, the parser calls an error recovery routine

Implementation of predictive parser:

1. Elimination of left recursion, left factoring and ambiguous grammar.
2. Construct FIRST() and FOLLOW() for all non-terminals.
3. Construct predictive parsing table.
4. Parse the given input string using stack and parsing table

Algorithm for Nonrecursive 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 in which it has $\$S$ on the stack with S, the start symbol of G on top, and $w\$$ in the input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input is shown in Fig.

set ip to point to the first symbol of $w\$$. repeat

```
    let X be the top stack symbol and a the symbol pointed to by ip. if X is a terminal of $ then
        if  $X = a$  then
            pop X from the stack and advance ip else error()
        else
            if  $M[X,a] = X \rightarrow Y_1 Y_2 \dots Y_k$  then begin pop X from the stack;
                push  $Y_k, Y_{k-1} \dots Y_1$  onto the stack, with  $Y_1$  on top; output the production  $X \rightarrow Y_1 Y_2 \dots Y_k$ 
            end
            else error()
    until  $X = \$$  /* stack is empty */
```

2.5.3 FIRST AND FOLLOW

The construction of a predictive parsing table is aided by two functions associated with a grammar:

1. FIRST
2. FOLLOW

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

Rules for FIRST ():

1. If X is terminal, then FIRST(X) is {X}.
2. If $X \rightarrow \epsilon$ 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 $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1), ..., FIRST(Y_{i-1}); that is, $Y_1, \dots, Y_{i-1} \Rightarrow \epsilon$. If ϵ is in FIRST(Y_j) for all $j=1, 2, \dots, k$, then add ϵ to FIRST(X).

Rules for FOLLOW ():

1. If S is a start symbol, then FOLLOW(S) contains \$.
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ϵ is placed in follow(B).
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ϵ , then everything in FOLLOW(A) is in FOLLOW(B).

2.5.4 Algorithm for construction of predictive parsing table:

Input : Grammar G

Output : Parsing table M

Method :

1. For each production $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.
2. For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to $M[A, a]$.
3. If ϵ is in FIRST(α), add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal b in FOLLOW(A). If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, \$]$.
4. Make each undefined entry of M be error.

Example:

Consider the following grammar :

$$E \rightarrow E+T \mid T$$

$$T \rightarrow T*F \mid F$$

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

After eliminating left-recursion the grammar is

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

First() :

$$\text{FIRST}(E) = \{ (, id \}$$

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

$$\text{FIRST}(T) = \{ (, id \}$$

$$\text{FIRST}(T') = \{ *, \varepsilon \}$$

$$\text{FIRST}(F) = \{ (, id \}$$

Follow():

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

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

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

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

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

Predictive parsing Table

NON- TERMINAL	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	$F \rightarrow id$			$F \rightarrow (E)$		

Stack Implementation

<u>stack</u>	Input	Output
SE	<u>id+id*id</u> S	
SE'T	<u>id+id*id</u> S	$E \rightarrow TE'$
SE'T'F	<u>id+id*id</u> S	$T \rightarrow FT'$
<u>SE'T'id</u>	<u>id+id*id</u> S	<u>$F \rightarrow id$</u>
SE'T'	+id*id S	
SE'	+id*id S	$T' \rightarrow \varepsilon$
SE'T+	+id*id S	$E' \rightarrow +TE'$
SE'T	id*id S	
SE'T'F	id*id S	$T \rightarrow FT'$
<u>SE'T'id</u>	id*id S	<u>$F \rightarrow id$</u>
SE'T'	*id S	
SE'T'F*	*id S	$T' \rightarrow *FT'$
SE'T'F	id S	
<u>SE'T'id</u>	id S	<u>$F \rightarrow id$</u>
SE'T'	S	
SE'	S	$T' \rightarrow \varepsilon$
S	S	$E' \rightarrow \varepsilon$

- 24 -

2.5.5.LL(1) GRAMMAR

The parsing table algorithm can be applied to any grammar G to produce a parsing table M . For some Grammars, for example if G is left recursive or ambiguous, then M will have at least one multiply-defined entry. A grammar whose parsing table has no multiply defined entries is said to be LL(1). It can be shown that the above algorithm can be used to produce for every LL(1) grammar G , a parsing table M that parses all and only the sentences of G . LL(1) grammars have several distinctive properties. No ambiguous or left recursive grammar can be LL(1). There remains a question of what should be done in case of multiply defined entries. One easy solution is to eliminate all left recursion and left factoring, hoping to produce a grammar which will produce no multiply defined entries in the parse tables. Unfortunately there are some grammars which will give an LL(1) grammar after any kind of alteration. In general, there are no universal rules to convert multiply defined entries into single valued entries without affecting the language recognized by the parser.

The main difficulty in using predictive parsing is in writing a grammar for the source language such that a predictive parser can be constructed from the grammar. Although left recursion elimination and left factoring are easy to do, they make the resulting grammar hard to read and difficult to use the translation purposes. To alleviate some of this difficulty, a common organization for a parser in a compiler is to use a predictive parser for control constructs and to use operator precedence for expressions. However, if an LR parser generator is available, one can get all the benefits of predictive parsing and operator precedence automatically.

2.5.6.ERROR RECOVERY IN PREDICTIVE PARSING

The stack of a nonrecursive predictive parser makes explicit the terminals and nonterminals that the parser hopes to match with the remainder of the input. We shall therefore refer to symbols on the parser stack in the following discussion. 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 the parsing table entry $M[A,a]$ is empty.

Consider error recovery predictive parsing using the following two methods

Panic-Mode recovery

Phrase Level recovery

Panic-mode error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appears. Its effectiveness depends on the choice of synchronizing set. The sets should be chosen so that the parser recovers quickly from errors that are likely to occur in practice.

As a starting point, we can 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.

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 generating expressions. A missing semicolon after an assignment may therefore result in the keyword beginning the next statement being skipped. Often, there is a hierarchical structure on constructs in a language; e.g., expressions appear within statement, which appear within blocks, and so on. We can add to the

CSE Dept.,Sir CRR COE.

synchronizing set of a lower construct the symbols that begin higher constructs. For example, we might add keywords that begin statements to the synchronizing sets for the nonterminals generating expressions.

If we add symbols in $\text{FIRST}(A)$ to the synchronizing set for nonterminal A , then it may be possible to resume parsing according to A if a symbol in $\text{FIRST}(A)$ appears in the input.

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.

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.

Phrase Level recovery

This involves, defining the blank entries in the table with pointers to some error routines which may

Change, delete or insert symbols in the input or

May also pop symbols from the stack

2.6 BOTTOM-UP PARSING

Constructing a parse tree for an input string beginning at the leaves and going towards the root is called bottom-up parsing. A general type of bottom-up parser is a shift-reduce parser.

2.6.1 SHIFT-REDUCE PARSING

Shift-reduce parsing is a type of bottom-up parsing that attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

Example:

Consider the grammar: $S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

The sentence to be recognized is $abbcde$.

REDUCTION (LEFTMOST)

$abbcde (A \rightarrow b)$

$aAbcde (A \rightarrow Abc)$

$aAde (B \rightarrow d)$

$aABe (S \rightarrow aABe)$

S

RIGHTMOST DERIVATION

$S \rightarrow aABe$

$\rightarrow aAde$

$\rightarrow aAbcde$

$\rightarrow abbcde$

The reductions trace out the right-most derivation in reverse.

Handles: A handle of a string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a rightmost derivation.

Example:

Consider the grammar:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow id$

And the input string $id1 + id2 * id3$

The rightmost derivation is :

$E \rightarrow E + E$

$\rightarrow E + \underline{E * E}$

$\rightarrow E + E * \underline{id3}$

$\rightarrow E + id2 * id3$

$\rightarrow id1 + id2 *$

In the above derivation the underlined substrings are called handles.

Handle pruning:

A rightmost derivation in reverse can be obtained by “handle pruning”. (i.e.) if w is a sentence or string of the grammar at hand, then $w = \gamma_n$, where γ_n is the n th right sentential form of some rightmost derivation.

Actions in shift-reduce parser:

- shift - The next input symbol is shifted onto the top of the stack.
- reduce - The parser replaces the handle within a stack with a non-terminal.
- accept - The parser announces successful completion of parsing.
- error - The parser discovers that a syntax error has occurred and calls an error recovery routine.

Conflicts in shift-reduce parsing:

There are two conflicts that occur in shift-reduce parsing:

1. Shift-reduce conflict: The parser cannot decide whether to shift or to reduce.
2. Reduce-reduce conflict: The parser cannot decide which of several reductions to make.

Stack implementation of shift-reduce parsing :

Stack	Input	Action
\$	$id_1 + id_2 * id_3$ \$	shift
\$ id_1	$+ id_2 * id_3$ \$	reduce by $E \rightarrow id$
\$ E	$+ id_2 * id_3$ \$	shift
\$ E +	$id_2 * id_3$ \$	shift
\$ E + id_2	$* id_3$ \$	reduce by $E \rightarrow id$
\$ E + E	$* id_3$ \$	shift
\$ E + E *	id_3 \$	shift
\$ E + E * id_3	\$	reduce by $E \rightarrow id$
\$ E + E * E	\$	reduce by $E \rightarrow E * E$
\$ E + E	\$	reduce by $E \rightarrow E + E$
\$ E	\$	accept

1. Shift-reduce conflict:

Example:

Consider the grammar:

 $E \rightarrow E + E \mid E * E \mid id$ and input $id + id * id$

Stack	Input	Action	Stack	Input	
\$ E + E	*id \$	Reduce by $E \rightarrow E + E$	SE + E	*id \$	Shift
SE	*id \$	Shift	SE + E *	id \$	Shift
SE *	id \$	Shift	SE + E * id	\$	Reduce by $E \rightarrow id$
SE * id	\$	Reduce by $E \rightarrow id$	SE + E * E	\$	Reduce by $E \rightarrow E * E$
SE * E	\$	Reduce by $E \rightarrow E * E$	SE + E	\$	Reduce by $E \rightarrow E * E$
SE			SE		

2. Reduce-reduce conflict:

Consider the grammar: $M \rightarrow R+R | R+c | R$

$R \rightarrow c$

input $c+c$

Stack	Input	Action	Stack	Input	Action
\$	<u>c+c</u> \$	Shift	\$	<u>c+c</u> \$	Shift
Sc	+c \$	Reduce by <u>$R \rightarrow c$</u>	Sc	+c \$	Reduce by <u>$R \rightarrow c$</u>
SR	+c \$	Shift	SR	+c \$	Shift
\$R+	c \$	Shift	\$R+	c \$	Shift
<u>\$R+c</u>	\$	Reduce by <u>$R \rightarrow c$</u>	<u>\$R+c</u>	\$	Reduce by <u>$M \rightarrow R+c$</u>
\$R+R	\$	Reduce by <u>$M \rightarrow R+R$</u>	\$M	\$	
\$M	\$				

INTRODUCTION TO LR PARSERS

An efficient bottom-up syntax analysis technique that can be used CFG is 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. When 'k' is omitted, it is assumed to be 1.

Advantages of LR parsing:

1. It recognizes virtually all programming language constructs for which CFG can be written.
2. It is an efficient non-backtracking shift-reduce parsing method.
3. A grammar that can be parsed using LR method is a proper superset of a grammar that can be parsed with predictive parser
4. It detects a syntactic error as soon as possible.

Drawbacks of LR method:

It is too much of work to construct a LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator, is needed. Example: YACC.

Types of LR parsing method:

1. SLR- Simple LR

Easiest to implement, least powerful.

2. CLR- Canonical LR

Most powerful, most expensive.

3. LALR- Look-Ahead LR

Intermediate in size and cost between the other two methods.

The LR parsing algorithm:

The schematic form of an LR parser is as follows:

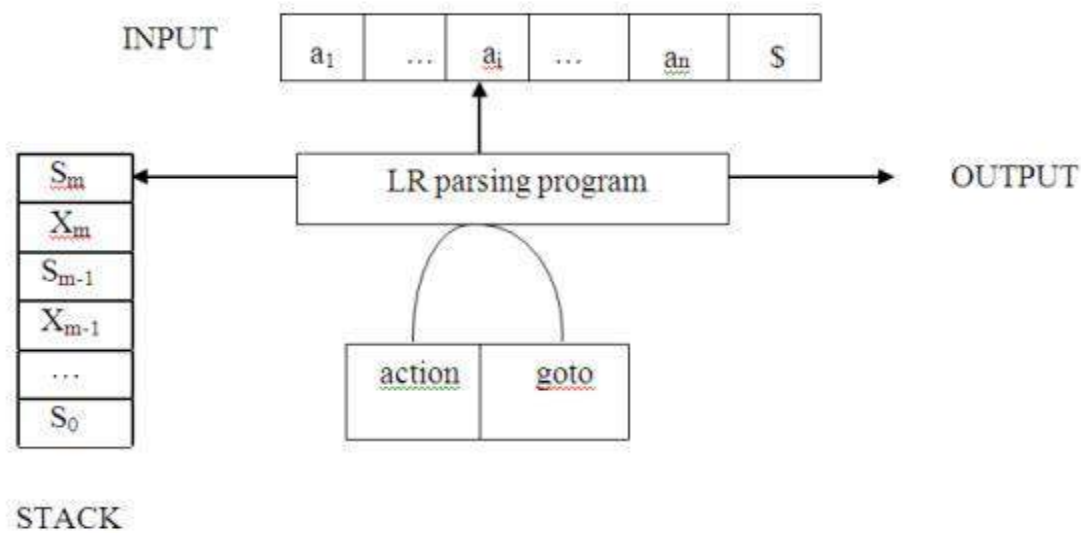


Fig. 2.5 Model of an LR parser

It consists of an input, an output, a stack, a driver program, and a pa parts (action and goto).

- The driver program is the same for all LR parser.
- The parsing program reads characters from an input buffer one at a time.
- The program uses a stack to store a string of the form $s_0X_1s_1X_2s_2\ldots X_ms_m$, where s_m is on top. Each X_i is a grammar symbol and each s_i is a state.
- The parsing table consists of two parts : action and goto functions.

Action : The parsing program determines s_m , the state currently on top of stack, and a_i , the current input symbol. It then consults $\text{action}[s_m, a_i]$ in the action table which can have one of four values:

1. shift s , where s is a state,
2. reduce by a grammar production $A \rightarrow \beta$,
3. accept,
4. Error.

Goto : The function goto takes a state and grammar symbol as arguments and produces a state.

LR Parsing algorithm:

Input: An input string w and an LR parsing table with functions action and goto for grammar G .

Output: If w is in $L(G)$, a bottom-up-parse for w ; otherwise, an error indication.

Method: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer.

The parser then executes the following program:

JSVG Krishna, Associate Professor.

CSE Dept.,Sir CRR COE.

set ip to point to the first input symbol of w\$; repeat forever begin

let s be the state on top of the stack and a the symbol pointed to by ip;

if action[s, a] = shift s' then begin

push a then s' on top of the stack; advance ip to the next input symbol end

else if action[s, a] = reduce $A \rightarrow \beta$ then begin pop $2 * |\beta|$ symbols off the stack;

let s' be the state now on top of the stack; push A then goto[s', A] on top of the stack; output the production $A \rightarrow \beta$

end

else if action[s, a] = accept then

return

else error()

end

CONSTRUCTING SLR(1) PARSING TABLE

To perform SLR parsing, take grammar as input and do the following:

1. Find LR(0) items.
2. Completing the closure.
3. Compute goto(I,X), where, I is set of items and X is grammar symbol.

LR(0) items:

An LR(0) item of a grammar G is a production of G with a dot at some position of the right side. For example, production $A \rightarrow XYZ$ yields the four items :

$A \rightarrow .XYZ$

$A \rightarrow X .YZ$

$A \rightarrow XY .Z$

$A \rightarrow XYZ .$

Closure operation:

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, every item in I is added to closure(I).
2. If $A \rightarrow \alpha . B\beta$ is in closure(I) and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow . \gamma$ to I, if it is not already there. We apply this rule until no more new items can be added to closure(I).

Goto operation:

Goto(I, X) is defined to be the closure of the set of all items $[A \rightarrow \alpha X . \beta]$ such that $[A \rightarrow \alpha . X\beta]$ is in I.

Steps to construct SLR parsing table for grammar G are:

1. Augment G and produce G'
2. Construct the canonical collection of set of items C for G'
3. Construct the parsing action function action and goto using the following algorithm that requires FOLLOW(A) for each non-terminal of grammar.

CSE Dept., Sir CRR COE.

Algorithm for construction of SLR parsing table:

Input : An augmented grammar G'

Output : The SLR parsing table functions action and goto for G'

Method :

1. Construct $C = \{I_0, I_1, \dots, I_n\}$, the collection of sets of LR(0) items for G' .

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

(a) If $[A \rightarrow \alpha \cdot a \beta]$ is in I_i and $\text{goto}(I_i, a) = I_j$, then set $\text{action}[i, a]$ to "shift j ". Here a must be terminal.

(b) If $[A \rightarrow \alpha \cdot]$ is in I_i , then set $\text{action}[i, a]$ to "reduce $A \rightarrow \alpha$ " for all a in $\text{FOLLOW}(A)$.

(c) If $[S' \rightarrow \cdot S]$ is in I_i , then set $\text{action}[i, \$]$ to "accept".

If any conflicting actions are generated by the above rules, we say grammar is not SLR(1).

3. The goto transitions for state i are constructed for all non-term

If $\text{goto}(I_i, A) = I_j$, then $\text{goto}[i, A] = j$.

4. All entries not defined by rules (2) and (3) are made "error"

5. The initial state of the parser is the one constructed from the $[S' \rightarrow \cdot S]$.

Example on SLR (1) Grammar

$S \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow \text{id}$

Add Augment Production and insert ' \cdot ' symbol at the first position for every production in G

$S' \rightarrow \cdot E$

$E \rightarrow \cdot E + T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

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

I0 State:

Add Augment production to the I0 State and Compute the Closure

I0 = Closure ($S' \rightarrow \cdot E$)

Add all productions starting with E in to I0 State because "." is followed by the non-terminal. So, the I0 State becomes

I0 = $S' \rightarrow \cdot E$

$E \rightarrow \cdot E + T$

$E \rightarrow \cdot T$

Add all productions starting with T and F in modified I0 State because "." is followed by the non-terminal. So, the I0 State becomes.

CSE Dept., Sir CRR COE.

I0 = $S' \rightarrow \bullet E$

$E \rightarrow \bullet E + T$

$E \rightarrow \bullet T$

$T \rightarrow \bullet T * F$

$T \rightarrow \bullet F$

$F \rightarrow \bullet id$

I1 = Go to (I0, E) = closure ($S' \rightarrow E\bullet$, $E \rightarrow E\bullet + T$)

I2 = Go to (I0, T) = closure ($E \rightarrow T\bullet T$, $T\bullet \rightarrow * F$)

I3 = Go to (I0, F) = Closure ($T \rightarrow F\bullet$) = $T \rightarrow F\bullet$

I4 = Go to (I0, id) = closure ($F \rightarrow id\bullet$) = $F \rightarrow id\bullet$

I5 = Go to (I1, +) = Closure ($E \rightarrow E + \bullet T$)

Add all productions starting with T and F in I5 State because "." is followed by the non-terminal. So, the I5 State becomes

I5 = $E \rightarrow E + \bullet T$

$T \rightarrow \bullet T * F$

$T \rightarrow \bullet F$

$F \rightarrow \bullet id$

Go to (I5, F) = Closure ($T \rightarrow F\bullet$) = (same as I3)

Go to (I5, id) = Closure ($F \rightarrow id\bullet$) = (same as I4)

I6 = Go to (I2, *) = Closure ($T \rightarrow T * \bullet F$)

Add all productions starting with F in I6 State because "." is followed by the non-terminal. So, the I6 State becomes

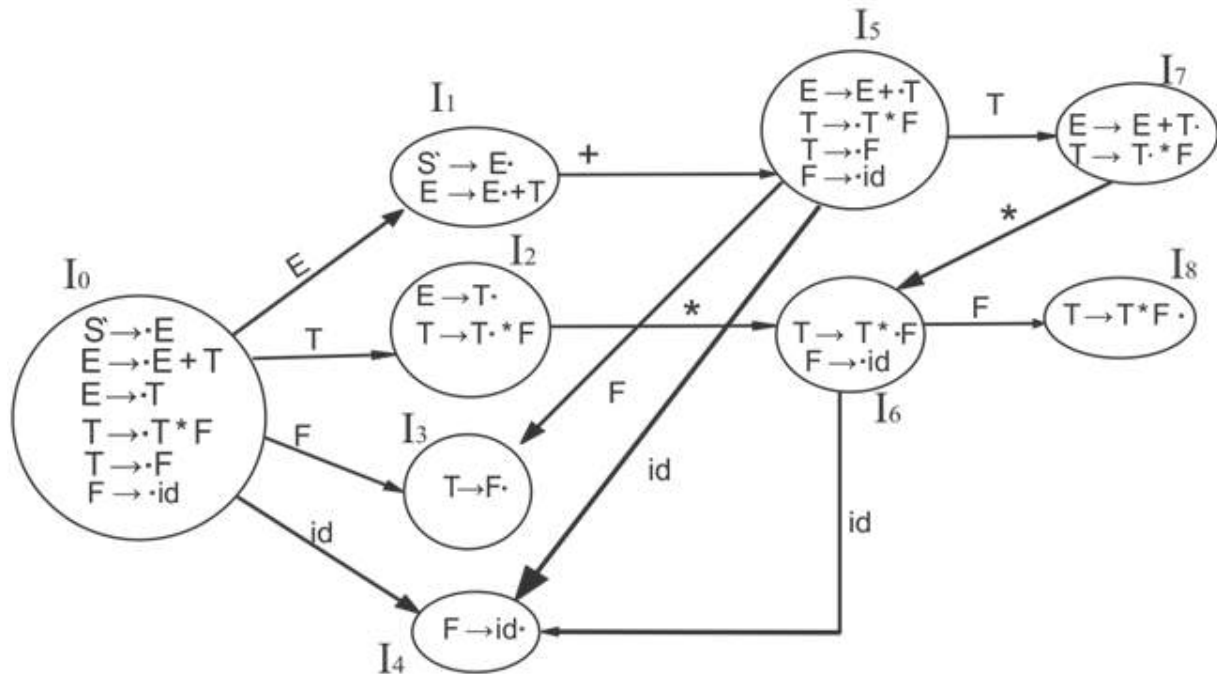
I6 = $T \rightarrow T * \bullet F$

$F \rightarrow \bullet id$

Go to (I6, id) = Closure ($F \rightarrow id\bullet$) = (same as I4)

I7 = Go to (I5, T) = Closure ($E \rightarrow E + T\bullet$) = $E \rightarrow E + T\bullet$

I8 = Go to (I6, F) = Closure ($T \rightarrow T * F\bullet$) = $T \rightarrow T * F\bullet$

Drawing DFA

SLR (1) Table

States	Action				Go to		
	id	+	*	\$	E	T	F
I ₀	S ₄				1	2	3
I ₁		S ₅		Accept			
I ₂		R ₂	S ₆	R ₂			
I ₃		R ₄	R ₄	R ₄			
I ₄		R ₅	R ₅	R ₅			
I ₅	S ₄					7	3
I ₆	S ₄						8
I ₇		R ₁	S ₆	R ₁			
I ₈		R ₃	R ₃	R ₃			

Explanation:

First (E) = First (E + T) \cup First (T)

First (T) = First (T * F) \cup First (F)

First (F) = {id}

First (T) = {id}

First (E) = {id}

Follow (E) = First (+T) \cup {\$} = {+, \$}

Follow (T) = First (*F) \cup First (F)

= {*, +, \$}

Follow (F) = {*, +, \$}

CSE Dept.,Sir CRR COE.

- 1) I1 contains the final item which drives $S \rightarrow E\bullet$ and follow (S) = { $\$$ }, so action {I1, $\$$ } = Accept
- 2) I2 contains the final item which drives $E \rightarrow T\bullet$ and follow (E) = {+, $\$$ }, so action {I2, +} = R2, action {I2, $\$$ } = R2
- 3) I3 contains the final item which drives $T \rightarrow F\bullet$ and follow (T) = {+, *, $\$$ }, so action {I3, +} = R4, action {I3, *} = R4, action {I3, $\$$ } = R4
- 4) I4 contains the final item which drives $F \rightarrow id\bullet$ and follow (F) = {+, *, $\$$ }, so action {I4, +} = R5, action {I4, *} = R5, action {I4, $\$$ } = R5
- 5) I7 contains the final item which drives $E \rightarrow E + T\bullet$ and follow (E) = {+, $\$$ }, so action {I7, +} = R1, action {I7, $\$$ } = R1
- 6) I8 contains the final item which drives $T \rightarrow T * F\bullet$ and follow (T) = {+, *, $\$$ }, so action {I8, +} = R3, action {I8, *} = R3, action {I8, $\$$ } = R3.