Unit II

Parser

CFG – Derivation – CFG vs R.E. - Types Of Parser –Bottom UP: Shift Reduce Parsing - Operator Precedence Parsing, SLR parser- Top Down: Recursive Decent Parser - Non-Recursive Decent Parser.

SYNTAX ANALYSIS

Every programming language has rules that prescribe the syntactic structure of well-formed programs. In Pascal, for example, a program is made out of blocks, a block out of statements, a statement out of expressions, an expression out of tokens, and so on. The syntax of programming language constructs can be described by context-free grammars or BNF (Backus-Naur Form) notation. Grammars offer significant advantages to both language designers and compiler writers.

• A grammar gives a precise, yet easy-to-understand. Syntactic specification of a programming language.

• From certain classes of grammars we can automatically construct an efficient parser that determines if a source program is syntactically well formed. As an additional benefit, the parser construction process can reveal syntactic ambiguities and other difficult-to-parse constructs that might otherwise go undetected in the initial design phase of a language and its compiler.

• A properly designed grammar imparts a structure to a programming language that is useful for the translation of source programs into correct object code and for the detection of errors. Tools are available for converting grammar-based descriptions of translations into working pro-grams.

• Languages evolve over a period of time, acquiring new constructs and performing additional tasks. These new constructs can be added to a language more easily when there is an existing implementation based on a grammatical description of the language.

ROLE OF THE PARSER :

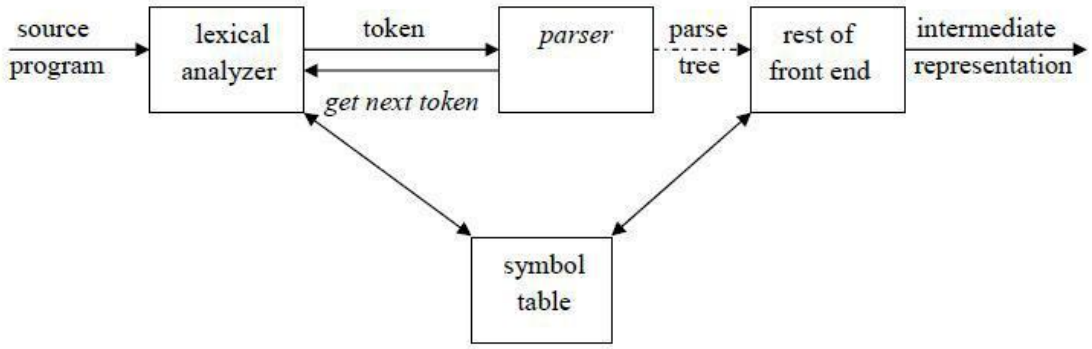
Parser for any grammar is program that takes as input string w (obtain set of strings tokens from the lexical analyzer) and produces as output either a parse tree for w , if w is a valid sentences of grammar or error message indicating that w is not a valid sentences of given grammar. The goal of the parser is to determine the syntactic validity of a source string is valid, a tree is built for use by the subsequent phases of the computer. The tree reflects the sequence of derivations or reduction used during the parser. Hence, it is called parse tree. If string is invalid, the parse has to issue diagnostic message identifying the nature and cause of the errors in string. Every elementary subtree in the parse tree corresponds to a production of the grammar.

There are two ways of identifying an elementry subtree:

By deriving a string from a non-terminal or
By reducing a string of symbol to a non-terminal.

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.



CONTEXT FREE GRAMMARS

A context-free grammar (grammar for short) consists of terminals, nonterminals, a start symbol, and productions.

1. Terminals are the basic symbols from which strings are formed. The word "token" is a synonym for "terminal" when we are talking about grammars for programming languages.
2. Non terminals are syntactic variables that denote sets of strings. They also impose a hierarchical structure on the language that is useful for both syntax analysis and translation.
3. In a grammar, one non terminal is distinguished as the start symbol, and the set of strings it denotes is the language defined by the grammar.
4. The productions of a grammar specify the manner in which the terminals and non terminals can be combined to form strings. Each production consists of a non terminal, followed by an arrow, followed by a string of non terminals and terminals.

Inherently recursive structures of a programming language are defined by a context-free Grammar. In a context-free grammar, we have four triples G(V,T,P,S). Here , V is finite set of terminals (in our case, this will be the set of tokens) T is a finite set of non-terminals (syntactic-variables).P is a finite set of productions rules in the following form A → α where A is a non-terminal and α is a string of terminals and non-terminals (including the empty string).S is a start symbol (one of the non-terminal symbol).

L(G) is the language of G (the language generated by G) which is a set of sentences.

A sentence of L(G) is a string of terminal symbols of G. If S is the start symbol of G then ω is a sentence of L(G) iff S ω where ω is a string of terminals of G. If G is a context-free grammar, L(G) is a context-free language. Two grammar G_1 and G_2 are equivalent, if they produce same grammar.

Consider the production of the form S α, If α contains non-terminals, it is called as a sentential form of G. If α does not contain non-terminals, it is called as a sentence of G.

Derivations

In general a derivation step is αAβ αγβ is sentential form and if there is a production rule A→γ in our grammar. where α and β are arbitrary strings of terminal and non-terminal symbols α1 α2 ... αn (αn derives from α1 or α1 derives αn). There are two types of derivation:

1.At each derivation step, we can choose any of the non-terminal in the sentential form of G for

the replacement.If we always choose the left-most non-terminal in each derivation step, this derivation is called as left-most derivation.

Example:
E → E + E | E – E | E * E | E / E | - E | (E) | id

Leftmost derivation :

E $\underset{lm}{}$ E + E

E * E+E $\underset{lm}{}$

id * E+E $\underset{lm}{}$

id * id+E $\underset{lm}{}$

id * id+id

The string is derive from the grammar ,w= id*id+id, which is consists of all terminal symbols

Rightmost derivation :

E $\underset{rm}{}$ E + E

$\underset{rm}{}$ E+E * E

$\underset{rm}{}$ E+ E*id

$\underset{rm}{}$ E+id*id

$\underset{rm}{}$ id+id*id

Given grammar G : E → E+E | E*E | (E) | - E | id Sentence to be derived : – (id+id)

LEFTMOST DERIVATION	RIGHTMOST DERIVATION
E $\underset{lm}{}$ - E	E $\underset{rm}{}$ - E
E $\underset{lm}{}$ - (E)	E $\underset{rm}{}$ - (E)
E $\underset{lm}{}$ - (E+E)	E $\underset{rm}{}$ - (E+E)
E $\underset{lm}{}$ - (id+E)	E $\underset{rm}{}$ - (E+id)
E $\underset{lm}{}$ - (id+id)	E $\underset{rm}{}$ - (id+id)

String that appear in leftmost derivation are called left sentinel forms.String that appear in rightmost derivation are called right sentinel forms.

Sentinels:

Given a grammar G with start symbol S, if S → α , where α may contain non-terminals or terminals, then α is called the sentinel form of G.

Yield or frontier of tree:

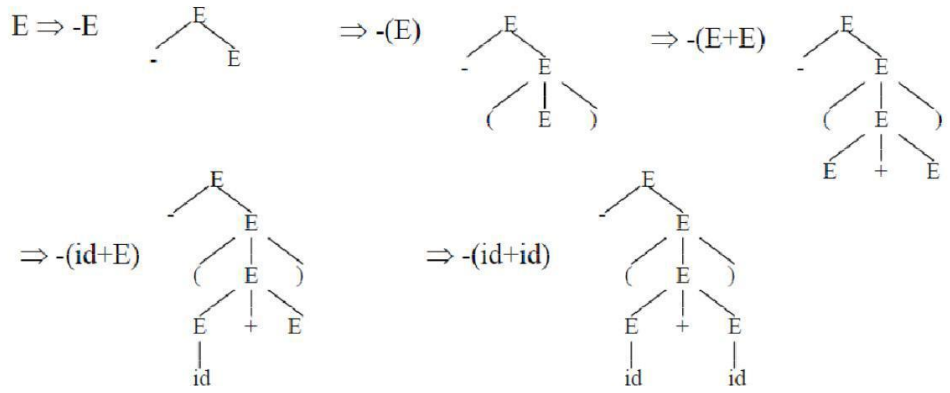
Each interior node of a parse tree is a non-terminal. The children of node can be a

terminal or non-terminal of the sentinel forms that are read from left to right. The sentinel form in the parse tree is called yield or frontier of the tree.

Parse Tree:

- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.
- A parse tree can be seen as a graphical representation of a derivation.

Example:



Ambiguity:

A grammar that produces more than one parse tree for some sentence is said to be ambiguous grammar.

Example : Given grammar G : E → E+E | E*E | (E) | - E | id
The sentence id+id*id has the following two distinct leftmost derivations:

E_{lm} E+ E	E_{lm} E* E
E_{lm} id + E	E_{lm} E + E * E
E_{lm} id + E * E	E_{lm} id + E * E
E_{lm} id + id * E	E_{lm} id + id * E
E_{lm} id + id * id	E_{lm} id + id * id

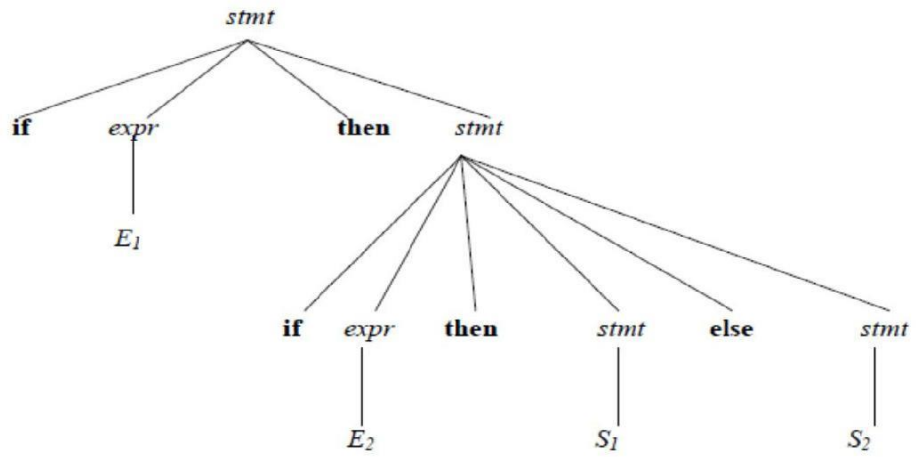
The two corresponding parse trees are :

Consider this example,

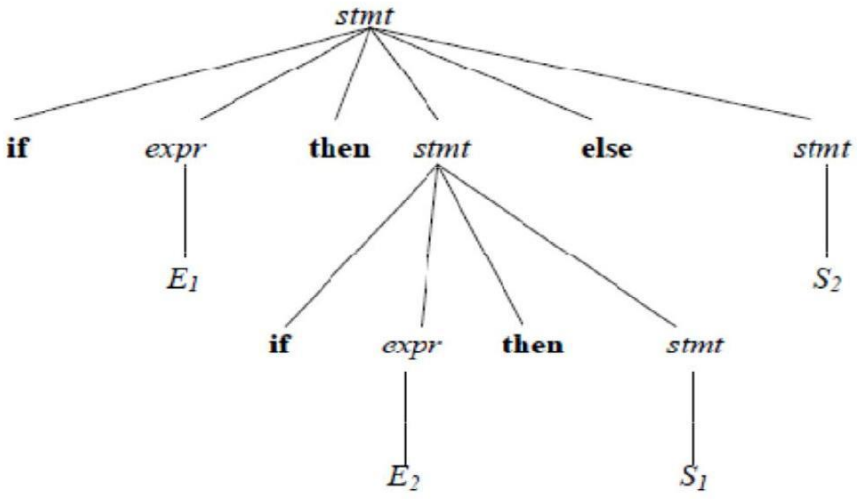
G: stmt → if expr then stmt | if expr then stmt else stmt | other

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 :

1.



2.



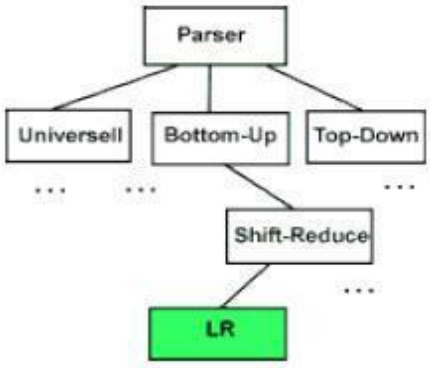
To eliminate ambiguity, the following grammar may be used: stmt → matched_stmt | unmatched_stmt

matched_stmt →if expr then matched_stmt else matched_stmt | other unmatched_stmt → if expr then stmt |

if expr then matched_stmt else unmatched_stmt

LR PARSING: INTRODUCTION

The "L" is for left-to-right scanning of the input and the "R" is for constructing a rightmost derivation in reverse.



WHY LR PARSING:

- LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written.
- The LR parsing method is the most general non-backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.
- The class of grammars that can be parsed using LR methods is a proper subset of the class of grammars that can be parsed with predictive parsers.

 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 disadvantage is that it takes too much work to construct an LR parser by hand for a typical programming-language grammar. But there are lots of LR parser generators available to make this task easy.

LL vs LR:

LL	**LR**
Does a leftmost derivation.	Does a rightmost derivation in reverse.
Starts with the root nonterminal on the stack.	Ends with the root nonterminal on the stack.

Ends when the stack is empty.	Starts with an empty stack.
Uses the stack for designating what is still to be expected.	Uses the stack for designating what is already seen.
Builds the parse tree top-down.	Builds the parse tree bottom-up.
Continuously pops a nonterminal off the stack, and pushes the corresponding right hand side.	Tries to recognize a right hand side on the stack, pops it, and pushes the corresponding nonterminal.
Expands the non-terminals.	Reduces the non-terminals.
Reads the terminals when it pops one off the stack.	Reads the terminals while it pushes them on the stack.
Pre-order traversal of the parse tree.	Post-order traversal of the parse tree.

Notational Conventions

To avoid always having to state that "these are the terminals." "these are the nonterminals," and so on. we shall employ the following notational conventions:

1. These symbols are terminals:
   i) Lowercase letters early in the alphabet such as a, b, c.
   ii) Operator symbols such as +, -,etc.
   iii) Punctuation symbols such as parentheses, comma. etc.
   iv) The digits 0, 1….. 9
   v) Boldface strings such as ***id*** or ***if***
2. These symbols are nonterminals:
   i) Upper-case letters early in the alphabet such as A,B,C.
   ii) The letter S, which, when it appears,is usually the start symbol.
   iii) Lower-case italic names such as *expr* or *stmt*.
3. Upper-case letters late in the alphabet. such as X. Y, Z, represent grammar symbols, that is, either nonterminals or terminals.
4. Lower-case letters late in the alphabet, namely u, v ……z represent strings of terminals.
5. Lower-case Greek letters, α,β, γ, for example, represent strings of grammar symbols. Thus, a generic production could be written as A→α, indicating that there is a single nonterminal A on the left of the arrow (the left side of the production) and a string of grammar symbols α to the right of the arrow (the right ride of the production).
6. If $A\rightarrow\alpha_1$ $A\rightarrow\alpha_2$ ……..$A\rightarrow\alpha_k$ as are all productions with A on the left (we call them A-productions), we may write $A\rightarrow\alpha_1|\ \alpha_2|\ldots\ldots\ldots|\ \alpha_k$. We call $\alpha_1,\alpha_2,\ldots\ldots\ \alpha_k$ the alternatives for A.

7. Unless otherwise stated, the left side of the first production is the start symbol.

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.

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 → aABe
A → Abc | b
B → d
The string to be recognized is abbcde. We want to reduce the string to S.

Steps of reduction:
abbcde (b,d can be reduced)
aAbcde (leftmost b is reduced,now Ab,b,d qualified for reduction)
aAcde (d can be reduced)
aAcBe
S
Each replacement of the right side of a production by the left side in the above example is called reduction, which is equivalent to rightmost derivation in reverse.

Handle:

A substring which is the right side of a production such that replacement of that substring by the production left side leads eventually to a reduction to the start symbol, by the reverse of a rightmost derivation is called a handle.

Definition: A handle of a right-sentential form γ is a production $A \rightarrow \beta$ and a position of γ where the string β may e found and replaced by A to produce the previous right-sentential form in the rightmost derivation of γ.That is,if $S \underset{rm}{\Rightarrow} \alpha Aw \underset{rm}{\Rightarrow} \alpha\beta w$,then $A \rightarrow \beta$ in the position following α is a handle of $\alpha\beta w$.The string w to the right of the handle contains only terminal symbols.

Stack Implementation of Shift-Reduce Parsing

There are two problems that must be solved if we are to parse by handle pruning.The first is to locate the substring to be reduced in a right-sentential form, and the second is to determine what production to choose in case there is more than one production with that substring on the right side.

A convenient way to implement a shift-reduce parser is to use a stack to hold grammar symbols and an input buffer to hold the string w 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
$	w$

The parser operates by shifting zero or more input symbols onto the stack until a handle is on top of the stack.. 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	$

Example: The actions a shift-reduce parser in parsing the input string $id_1+id_2*id_3$, according to the ambiguous grammar for arithmetic expression.

	STACK	INPUT	ACTION
(1)	$	$id_1 + id_2 * id_3$$	shift
(2)	$id_1	+ $id_2 * id_3$$	reduce by $E \rightarrow id$
(3)	$E	+ $id_2 * id_3$$	shift
(4)	$E +	$id_2 * id_3$$	shift
(5)	$E + id_2	* $id_3$$	reduce by $E \rightarrow id$
(6)	$E + E	* $id_3$$	shift
(7)	$E + E *	$id_3$$	shift
(8)	$E + E * id_3	$$	reduce by $E \rightarrow id$
(9)	$E + E * E	$$	reduce by $E \rightarrow E * E$
(10)	$E + E	$$	reduce by $E \rightarrow E + E$
(11)	$E	$$	accept

While the primary operations of the parser 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. In a *shift* action, the next input symbol is shifted unto the top of the stack.
2. In a *reduce* action, the parser knows the right end of the handle is at the top of the stack. It must then locate the left end of the handle within the stack and decide with what nonterminal to replace the handle.
3. In an *accept* action. the parser announces successful completion of parsing.
4. In an *error* action, the parser discovers that a syntax error has occurred and calls an error recovery routine.

Operator Precedence Parsing:

Operator grammars have the property that no production right side is ∈(empty) or has two adjacent non terminals. This property enables the implementation of efficient operator-precedence parsers.

Example:The following grammar for
expressions E→E A E | (E) | -E | id
A→+ | - | * | / | ^

Is not an operator grammar, because the right side EAE has two consecutive nonterminals.However, if we substitute for A each of its alternate, we obtain the following operator grammar:

E→E + E |E – E |E * E | E / E | (E) | E ^ E | - E | id

These parser rely on the following three precedence relations:

RELATION	MEANING
$a <\cdot b$	a "yields precedence to" b
$a \doteq b$	a "has the same precedence as" b
$a \cdot> b$	a "takes precedence over" b

These precedence relations guide the selection of handles. These operator precedence relations allow delimiting the handles in the right sentential forms: <· marks the left end, =· appears in the interior of the handle, and ·> marks the right end.

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	·>

Example: The input string: id1 + id2 * id3

after inserting precedence relations the string becomes:

$ <· id1 ·> + <· id2 ·> * <· id3 ·> $

Having precedence relations allows to identify handles as follows:
1. scan the string from left end until the leftmost ·> is encountered.
2. Then scan backwards over any =’s until a <· is encountered.
3. everything between the two relations <· and ·> forms the handle.

TABLE : Operator-precedence relations

	+	-	*	/	↑	id	(	)	$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	<	<	<	>	>
/	>	>	>	>	<	<	<	>	>
↑	>	>	>	>	<	<	<	>	>
id	>	>	>	>	>			>	>
(	<	<	<	<	<	<	<	=	
)	>	>	>	>	>			>	>
$	<	<	<	<	<	<	<		

STACK	INPUT	COMMENT
$	<· id+id*id $	shift id
$ id	·> +id*id $	pop the top of the stack id
$	<· +id*id $	shift +
$ +	<· id*id $	shift id
$ +id	·> *id $	pop id
$ +	<· *id $	shift *
$ + *	<· id $	shift id
$ + * id	·> $	pop id
$ + *	·> $	pop *
$ +	·> $	pop +
$	$	accept

OPERATOR PRECEDENCE PARSING ALGORITHM

Initialize: Set ip to point to the first symbol of w$

(1) ***repeat forever***
(2) ***if*** only $ is on the stack and only $ is on the input ***then***
accept and ***break***
else
begin
(3) Let a be the top terminal symbol on the stack,
and let b be the current input symbol pointed to by ip
(4) ***if*** a <· b or a =· b then push b onto the stack
(5) ***else if*** a ·> b ***then***
(6) ***repeat*** pop the stack
(7) ***until*** the top stack terminal is related by <· to the terminal most
recently popped.
(8) ***else*** call the error correcting routine
end

Precedence Functions

Compilers using operator-precedence parsers need not store the table of precedence relations. In most cases, the table can be encoded by two *precedence functions f* and *g* that map terminal symbols to integers. We attempt to select *l*and *g* so that, for symbols *a* and *b.*

1. f (a) < g(b) whenever a<·b.
2. f (a) = g(b) whenever a = b. and
3. f(a) > g(b) whenever a ·> b.

Thus precedence relation between a and b can be determined by a numerical comparison between *f (a)* and *g(b).H*owever, the error entries in the precedence matrix are obscured, since one of (1), (2),or (3) holds no matter what f (a) and g(b) are. The loss of error detection capabil-

ity is generally not considered serious enough to prevent the using of precedence functions where possible; errors can still *be* caught when a reduction is called for and no handle can be found.

Not every table of precedence relations has precedence functions to encode it. but in practical cases the functions usually exist.
Example: The precedence table given above has the following pair of precedence functions

	+	−	*	/	↑	(	)	id	$
f	2	2	4	4	4	0	6	6	0
g	1	1	3	3	5	5	0	5	0

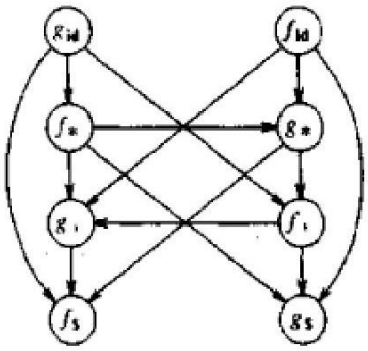
For example. * <· id, and *f(*)* < g(id). Note that *f* id) > g(id) suggests that id •> id; but. in fact, no precedence relation holds between id and id. Other error entries are similarly replaced by one or another precedence relation.

Algorithm: Constructing precedence functions.
Input: An operator precedence matrix.
Output: Precedence functions representing the input matrix, or an indication that none exist.
Method:

1. Create symbols *f, and* g, for each *a* that is a terminal or $.
2. Partition the created symbols into as many groups as possible, in such a way that if *a = b.* then f_a and g_b, are in the same group. Note that we may have to put symbols in the same group even if they are not related by =. For example, if *a = b* and c = *b,* then f_a, and f_c,. must be in the same group, since they are both in the same group as g_b. If,in addition, *c = d.* then f_a and g_d,, are in the same group even though *a = d* may not hold.
3. Create a directed graph whose nodes are the groups found in (2). For any *a* and *b,* if *a <. B,* place an edge from the group of g_b to the group of f_a. If *a .> b,* place an edge from the group of f_a, to that of g_b. Note that an edge or path from ,f_a to *gb* means that *f (a)* must exceed *g(b);* a path from g_b to f_a, means that *g(b)* must exceed *f (a).*
4. If the graph constructed in (3) has a cycle, then no precedence functions exist. If there are no cycles, let *f (a)* be the length of the longest path beginning at the group of f_a. let g *(a)* be the length of the longest path beginning at the group of g_a.

Example: Consider the below matrix ,there are no = relationship.The graph is shown below:

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	



There are no cycles,so precedence function exist.As $f_\$$ and $g_\$$ have no out edges,f($)=g($)=0.The longest path from g_+ has length 1,so g(+)=1.There is a path from g_{id} to f_* to g_* to f_+ to g_+ to $f_\$$,so g(id)=5.The resulting precedence functions are:

	+	*	id	$
f	2	4	4	0
g	1	3	5	0

Steps in Operator Precedence Parsing:

1. Computation of LEADING
2. Computation of TRAILING
3. Construction of Precedence Table
4. Parsing of the given input string

Problem 1:

Consider the following grammar, and construct the operator precedence parsing table and check whether the input string (i) *id=id (ii)id*id=id are successfully parsed or not?

S→L=R
S→R
L→*R
L→id
R→L

Solution:

1.Computation of LEADING:

LEADING(S) = {=, * , id}
LEADING(L) = {* , id}
LEADING(R) = {* , id}

2.Computation of TRAILING:

TRAILING(S) = {= , * , id}
TRAILING(L)= {* , id}
TRAILING(R)= {* , id}

3.Precedence Table:

	=	*	id	$
=		<·	<·	·>
*	·>	<·	<·	·>
id	·>			·>
$	<·	<·	<·	

All undefined entries are error.

4.Parsing the given input string:

(i) *id =id

STACK	INPUT STRING	ACTION
$	*id=id$	$<·* Push
$*	id=id$	*<·id Push
$*id	=id$	id·>= Pop
$*	=id$	*·>= Pop
$	=id$	$<·= Push
$=	id$	=<·id Push
$=id	$	id·>$ Pop
$=	$	=·>$ Pop
$	$	Accept

(ii) Id*id=id

STACK	INPUT STRING	ACTION
$	id*id=id$	$<·idPush
$id	*id=id$	Error

Assignment:

Consider the following grammar:

1) S→(L)
   S→a
   L→L,S
   L→S

2) S→a
   S→↑
   S→(T)
   T→T,S
   T→S

Problem 2: Check whether the following Grammar is an operator precedence grammar or not. E→E+E

E→E*E

E→id

Solution:

1.Computation of LEADING:
LEADING(E) = {+, * , id}

2.Computation of TRAILING:
TRAILING(E) = {+, * , id}

3.Precedence Table:

	+	*	id	$
+	<·/·>	<·/·>	<·	·>
*	<·/·>	<·/·>	<·	·>
id	·>	·>		·>
$	<·	<·	<·	

All undefined entries are error. Since the precedence table has multiple defined entries ,the grammar is not an operator precedence grammar.

Top-Down Parsing:

Eliminating Left Recursion:

A grammar is said to be left recursive if it has a non-terminal A such that there is a derivation A=>Aα for some string α. Top-down parsing methods cannot handle left-recursive grammars. Hence, left recursion can be eliminated as follows:

If there is a production A → Aα | β it can be replaced with a sequence of two productions A → βA'
A' → αA' | ε
Without changing the set of strings derivable from A.

Example : Consider the following grammar for arithmetic expressions: E → E+T | T

T → T*F | F

F → (E) | id

First eliminate the left recursion for E as E → TE'

E' → +TE' | ε

Then eliminate for T as T → FT'

T'→ *FT' | ε

Thus the obtained grammar after eliminating left recursion is E → TE' E' → +TE' | ε

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Algorithm to eliminate left recursion:

2. Arrange the non-terminals in some order A1, A2 . . . An.
3. for i := 1 to n do begin

    for j := 1 to i-1 do begin

        replace each production of the form $A_i \rightarrow A_j \gamma$

        by the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k \gamma$.

        where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current A_j-productions;

end

eliminate the immediate left recursion among the Ai-productions end

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.

If there is any production $A \rightarrow \alpha\beta1 \mid \alpha\beta2$, it can be rewritten as $A \rightarrow \alpha A'$

$A' \rightarrow \beta1 \mid \beta2$

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

$E \rightarrow b$

Left factored, this grammar becomes $S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \varepsilon$ $E \rightarrow b$

Example for back tracking :

Consider the grammar G :

$S \rightarrow cAd$

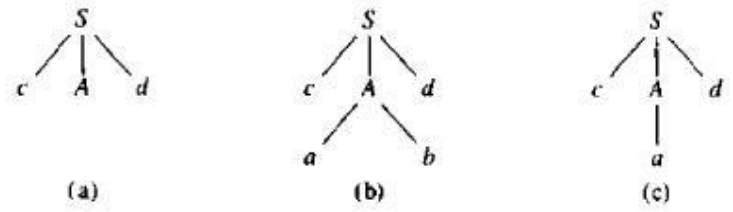
$A \rightarrow ab \mid a$

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 position. This is called backtracking.

Step4:

Now try the second alternative for A.Now we can halt and announce the successful completion of parsing.

Nonrecursive 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.

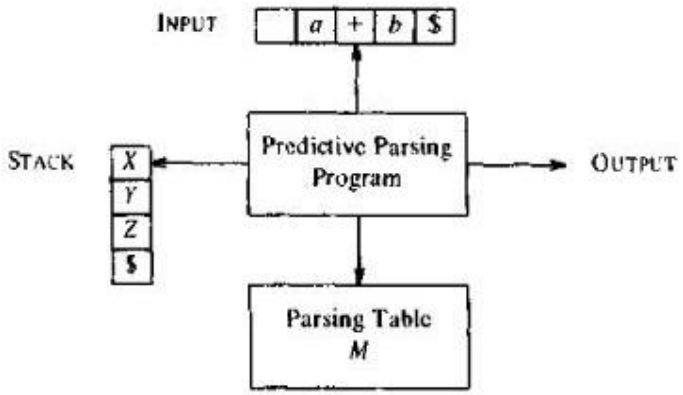


Fig. 4.13. Model of a nonrecursive predictive parser.

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 with $ on the bottom, indicating the bottom of the stack. Initially. the stack contains the start symbol of the grammar on top of S. 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 program considers X, the symbol on 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 =$,the parser halts and announces successful completion of parsing.
2. If X =a ≠$, 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 →UVW}, the parser replaces X on top of the stack by WVU (with U on top).

If M[X, a] = error, the parser calls an error recovery routine.

Predictive parsing table construction:

The construction of a predictive parser is aided by two functions associated with a grammar G :

FIRST()

FOLLOW()

Rules for FIRST():

1. If X is terminal, then FIRST(X) is {X}.
2. If X → ε is a production, then add ε to FIRST(X).
3. If X is non-terminal and X → aα is a production then add a to FIRST(X).
4. If X is non-terminal and X → $Y_1 Y_2 \ldots Y_k$ 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, $Y_1,\ldots Y_{i-1}$ => ε. If ε is in FIRST(Y_j) for all j=1,2,..,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 → αBβ, then everything in FIRST(β) except ε is placed in follow(B).

3. If there is a production A → αB, or a production A → αBβ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

Algorithm for construction of predictive parsing table: Input : Grammar G

Output : Parsing table

M Method :

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

Algorithm : 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 .

Method. Initially, the parser is in a configuration in which it has *$$* 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.

```
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 or $ then
           If X = a then
              pop X from the stack and
            advance ip else error()
       else          /* X is a nomerminal */
          if M[X,a] =X→Y1Y2…….Yk, then
              begin pop X from the stack:
               push Yk , Yk - 1 ……..Y1 , onto the stack, with Y1 on top;
              output the production X→Y1Y2…….Yk
          end
         else error()
until  X≠$  /* stack is empty*/
```

Example:

Consider the following grammar :

E → E+T | T

T → T*F | F

F → (E) | id

After eliminating left-recursion the grammar is

E → TE’

E’ → +TE’ | ε

T → FT’

T’ → *FT’ | ε

F → (E) | id

First( ) :

FIRST(E) = { (, id}

FIRST(E') ={+ , ε }

FIRST(T) = { (, id}

FIRST(T') = {*, ε }

FIRST(F) = { (, id }

Follow( ):

FOLLOW(E) = { $,) }

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

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

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

Predictive parsing table :

NON-TERMINAL	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E'→ ε
T	T → FT'			T → FT'		
T'		T'→ ε	T'→ *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

Stack implementation:

stack	Input	Output
$E	id+id*id $	
$E'T	id+id*id $	E → TE'
$E'T'F	id+id*id $	T → FT'
$E'T'id	id+id*id $	F → id
$E'T'	+id*id $	
$E'	+id*id $	T' → ε
$E'T+	+id*id $	E' → +TE'
$E'T	id*id $	
$E'T'F	id*id $	T → FT'
$E'T'id	id*id $	F → id
$E'T'	*id $	
$E'T'F*	*id $	T' → *FT'
$E'T'F	id $	
$E'T'id	id $	F → id
$E'T'	$	
$E'	$	T' → ε
$	$	E' → ε

Table:Moves made by the predictive parser on the input id+id*id$

LL(1) grammars:

The parsing table entries are single entries. So each location has not more than one entry. This type of grammar is called LL(1) grammar.

Consider this following grammar: S
→ iEtS | iEtSeS | a
E → b

After eliminating left factoring, we have S
→ iEtSS' | a
S'→ eS | ε E → b

To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals. FIRST(S) = { i, a }
FIRST(S') = {e, ε } FIRST(E) = { b} FOLLOW(S) = { $,e } FOLLOW(S') = { $,e } FOLLOW(E) = {t}

Parsing table:

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

Since there are more than one production, the grammar is not LL(1) grammar.

LR PARSERS

An efficient bottom-up syntax analysis technique that can be used to parse a large class of 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:

- ✓ It recognizes virtually all programming language constructs for which CFG can be written.
- ✓ It is an efficient non-backtracking shift-reduce parsing method.
- ✓ A grammar that can be parsed using LR method is a proper superset of a grammar that can be parsed with predictive parser.
- ✓ It detects asyntactic 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.

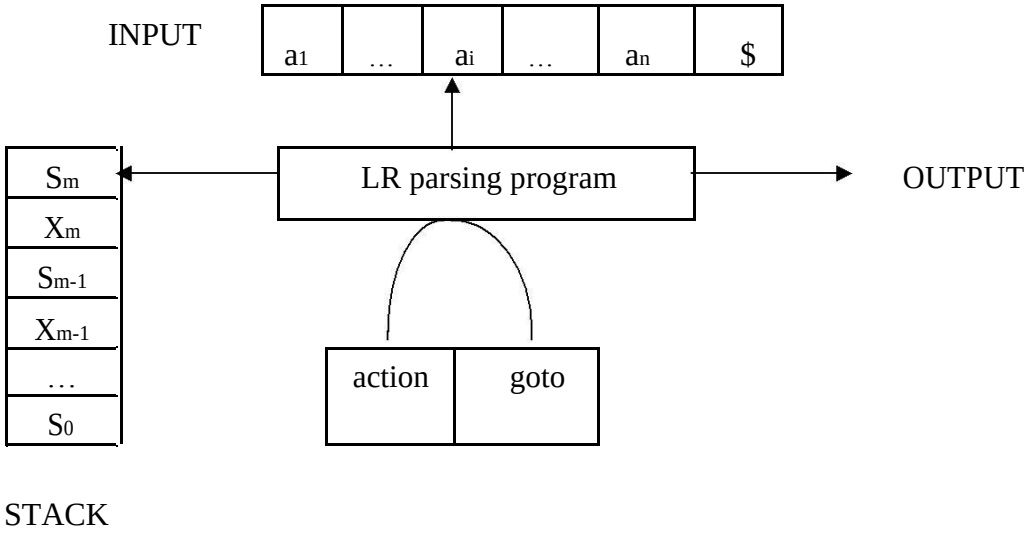
Types of LR parsing method:

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:



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 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 *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, and
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 :

```
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→β then begin
          pop 2* |β |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→ β
      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 → XYZ yields the four items :

A → **.** XYZ
A → X **.** YZ
A → XY **.** Z
A → 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 → α . Bβ is in closure(I) and B → γ is a production, then add the item B → . γ 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→ αX . β] such that [A→ α . Xβ] 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.

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 ={$I_0, I_1, \ldots I_n$}, 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→α·*a*β] is in I_i and goto(I_i,*a*) = I_j, then set *action*[*i*,*a*] to "shift j". Here *a* must be terminal.
   (b) If[A→α·] is in I_i, then set *action*[*i*,*a*] to "reduce A→α" for all *a* in FOLLOW(A).
   (c) If [S'→S.] is in I_i, then set *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-terminals A using the rule: If *goto*(I_i,A)= I_j, then *goto*[i,A] = *j*.
4. All entries not defined by rules (2) and (3) are made "error"
5. The initial state of the parser is the one constructed from the set of items containing [S'→.S].

Example for SLR parsing:

Construct SLR parsing for the following grammar :

G : E → E + T | T
T → T * F | F
F→ (E) | id

The given grammar is :

G : E → E + T ------ (1)
E →T ------ (2)
T → T * F ------ (3)
T → F ------ (4)
F→ (E) ------ (5)
F→ id ------ (6)

Step 1 : Convert given grammar into augmented grammar.
Augmented grammar :

E' → E
E → E + T
E → T
T → T * F
T → F
F→ (E)

F→ id

Step 2 : Find LR (0) items.

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

GOTO (I_0 , E)
I_1 : E’ → E **.**
E → E **.** + T

GOTO (I_4 , id)
I_5 : F→ id **.**

GOTO (I_0 , T)
I_2 : E → T .
T → T . * F

GOTO (I_0 , F)
I_3 : T → F .

GOTO (I_0 , ( )
I_4 : F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I_0 , id)
I_5 : F→ id .

GOTO (I_1 , +)
I_6 : E → E + . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I_2 , *)
I_7 : T → T * . F
F → . (E)
F → . id

GOTO (I_4 , E)
I_8 : F→ (E .)
E → E . + T

GOTO (I_4 , T)
I_2 : E →T .
T → T . * F

GOTO (I_4 , F)
I_3 : T → F .

GOTO (I_6 , T)
I_9 : E → E + T .
T → T . * F

GOTO (I_6 , F)
I_3 : T → F .

GOTO (I_6 , ( )
I_4 : F→ (. E)

GOTO (I_6 , id)
I_5 : F→ id .

GOTO (I_7 , F)
I_{10} : T → T * F .

GOTO (I_7 , ( )
I_4 : F→ (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I_7 , id)
I_5 : F → id .

GOTO (I_8 ,))
I_{11} : F→ (E) .

GOTO (I_8 , +)
I_6 : E → E + . T
T → . T * F
T → . F
F→ . (E)
F→ . id

GOTO (I_9 , *)
I_7 : T → T * . F
F→ . (E)
F→ . id

GOTO (I_4 , ())

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

E → E + T | T
T → T * F | F
F→ (E) | id

FOLLOW (E) = { $,) , +}
FOLLOW (T) = { $, + ,) , * }
FOOLOW (F) = { * , + ,) , $ }

SLR parsing table:

	ACTION						GOTO		
	id	+	*	(	)	$	**E**	**T**	**F**
I_0	s5			s4			1	2	3
I_1		s6				ACC			
I_2		r2	s7		r2	r2			
I_3		r4	r4		r4	r4			
I_4	s5			s4			8	2	3
I_5		r6	r6		r6	r6			
I_6	s5			s4				9	3
I_7	s5			s4					10
I_8		s6			s11				
I_9		r1	s7		r1	r1			
I_{10}		r3	r3		r3	r3			
I_{11}		r5	r5		r5	r5			

Blank entries are error entries.

Stack implementation:

Check whether the input **id + id * id** is valid or not.

STACK	INPUT	ACTION
0	id + id * id $	GOTO (I_0 , id) = s5 ; **shift**
0 id 5	+ id * id $	GOTO (I_5 , +) = r6 ; **reduce** by F→id
0 F 3	+ id * id $	GOTO (I_0 , F) = 3 GOTO (I_3 , +) = r4 ; **reduce** by T → F
0 T 2	+ id * id $	GOTO (I_0 , T) = 2 GOTO (I_2 , +) = r2 ; **reduce** by E → T
0 E 1	+ id * id $	GOTO (I_0 , E) = 1 GOTO (I_1 , +) = s6 ; **shift**
0 E 1 + 6	id * id $	GOTO (I_6 , id) = s5 ; **shift**
0 E 1 + 6 id 5	* id $	GOTO (I_5 , *) = r6 ; **reduce** by F→ id
0 E 1 + 6 F 3	* id $	GOTO (I_6 , F) = 3 GOTO (I_3 , *) = r4 ; **reduce** by T → F

0 E 1 + 6 T 9	* id $	GOTO (I_6 , T) = 9 GOTO (I_9 , *) = s7 ; **shift**
0 E 1 + 6 T 9 * 7	id $	GOTO (I_7 , id) = s5 ; **shift**
0 E + 6 T 9 * 7 id 1 5	$	GOTO (I_5 , $) = r6 ; **reduce** by F→ id
0 E + 6 T 9 * 7 F 1 10	$	GOTO (I_7 , F) = 10 GOTO (I_{10} , $) = r3 ; **reduce** by T → T * F
0 E 1 + 6 T 9	$	GOTO (I_6 , T) = 9 GOTO (I_9 , $) = r1 ; **reduce** by E → E + T
0 E 1	$	GOTO (I_0 , E) = 1 GOTO (I_1 , $) = **accept**