UNIT II SYNTAX ANALYSIS

Role of Parser – Grammars – Error Handling – Context-free grammars – Writing a grammar – Top Down Parsing - General Strategies Recursive Descent Parser Predictive Parser-LL(1) Parser-Shift Reduce Parser-LR Parser-LR (0)Item Construction of SLR Parsing Table - Introduction to LALR Parser - Error Handling and Recovery in Syntax Analyzer-YACC.

SYNTAX ANALYSIS

Syntax analysis is the second phase of the compiler. It gets the input from the tokens and generates a syntax tree or parse tree.

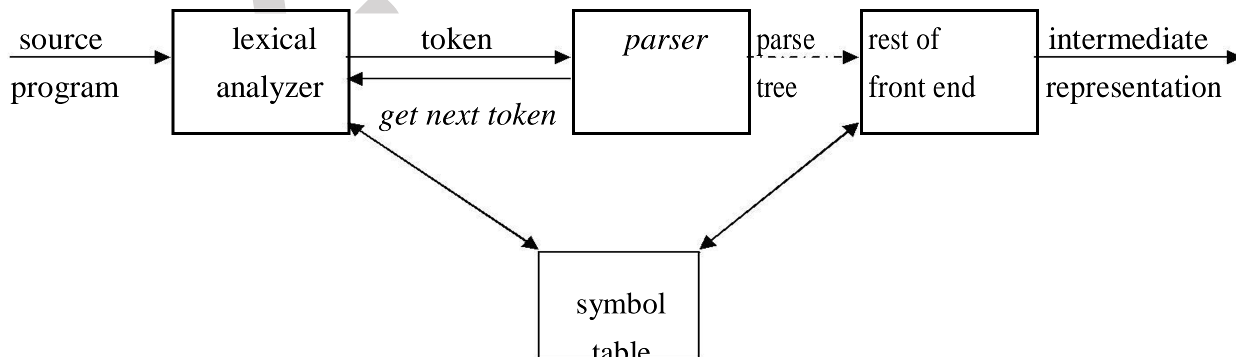
Advantages of grammar for syntactic specification:

1. A grammar gives a precise and easy-to-understand syntactic specification of a programming language.
2. An efficient parser can be constructed automatically from a properly designed grammar.
3. A grammar imparts a structure to a source program that is useful for its translation into object code and for the detection of errors.
4. New constructs can be added to a language more easily when there is a grammatical description of the language.

THE ROLE OF PARSER

The parser or syntactic analyzer obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language. It reports any syntax errors in the program. It also recovers from commonly occurring errors so that it can continue processing its input.

Position of parser in compiler model



Functions of the parser :

1. It verifies the structure generated by the tokens based on the grammar.
2. It constructs the parse tree.
3. It reports the errors.
4. It performs error recovery.

Issues :

Parser cannot detect errors such as:
1. Variable re-declaration
2. Variable initialization before use.
3. Data type mismatch for an operation.

The above issues are handled by Semantic Analysis phase.

Syntax error handling :

Programs can contain errors at many different levels. For example :
1. Lexical, such as misspelling a keyword.
2. Syntactic, such as an arithmetic expression with unbalanced parentheses.
3. Semantic, such as an operator applied to an incompatible operand.
4. Logical, such as an infinitely recursive call.

Functions of error handler :

1. It should report the presence of errors clearly and accurately.
2. It should recover from each error quickly enough to be able to detect subsequent errors.
3. It should not significantly slow down the processing of correct programs.

Error recovery strategies :

The different strategies that a parse uses to recover from a syntactic error are:

1. Panic mode
2. Phrase level
3. Error productions
4. Global correction

Panic mode recovery:

On discovering an error, the parser discards input symbols one at a time until a synchronizing token is found. The synchronizing tokens are usually delimiters, such as semicolon or **end**. It has the advantage of simplicity and does not go into an infinite loop. When multiple errors in the same statement are rare, this method is quite useful.

Phrase level recovery:

2106-JIT

On discovering an error, the parser performs local correction on the remaining input that allows it to continue. Example: Insert a missing semicolon or delete an extraneous semicolon etc.

Error productions:

The parser is constructed using augmented grammar with error productions. If an error production is used by the parser, appropriate error diagnostics can be generated to indicate the erroneous constructs recognized by the input.

Global correction:

Given an incorrect input string x and grammar G, certain algorithms can be used to find a parse tree for a string y, such that the number of insertions, deletions and changes of tokens is as small as possible. However, these methods are in general too costly in terms of time and space.

CONTEXT-FREE GRAMMARS

A Context-Free Grammar is a quadruple that consists of **terminals**, **non-terminals, start symbol** and **productions.**

Terminals : These are the basic symbols from which strings are formed.

Non-Terminals : These are the syntactic variables that denote a set of strings. These help to define the language generated by the grammar.

Start Symbol : One non-terminal in the grammar is denoted as the "Start-symbol" and the set of strings it denotes is the language defined by the grammar.

Productions : It specifies the manner in which 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.

Example of context-free grammar: The following grammar defines **simple** arithmetic expressions:

expr → *expr op expr*
expr → (*expr*)
expr → - *expr*
expr → **id**
op → +
op → -
op → *
op → /
op → ↑

2106-JIT

In this grammar,

- **id** + - * / ↑( ) are terminals.
- *expr* , *op* are non-terminals.
- *expr* is the start symbol.
- Each line is a production.

Derivations:

Two basic requirements for a grammar are :
1. To generate a valid string.
2. To recognize a valid string.

Derivation is a process that generates a valid string with the help of grammar by replacing the non-terminals on the left with the string on the right side of the production.

Example : Consider the following grammar for arithmetic expressions :

E → E+E |E*E |(E) | - E | id

To generate a valid string - (id+id) from the grammar the steps are

1. E → - E
2. E → - (E)
3. E → - (E+E)
4. E → - (id+E)
5. E → - (id+id)

In the above derivation,

- E is the start symbol.
- - (id+id) is the required sentence (only terminals).
- Strings such as E, -E, -(E), . . . are called sentinel forms.

Types of derivations:

The two types of derivation are:

1. Left most derivation
2. Right most derivation.

- In leftmost derivations, the leftmost non-terminal in each sentinel is always chosen first for replacement.
- In rightmost derivations, the rightmost non-terminal in each sentinel is always chosen first for replacement.

2106-JIT

Example:

Given grammar G : E → E+E |E*E |(E) | - E |id

Sentence to be derived : – (id+id)

LEFTMOST DERIVATION	RIGHTMOST DERIVATION
E → - E	E → - E
E → - (E)	E → - (E)
E → - (E+E)	E → - (E+E)
E → - (id+E)	E → - (E+id)
E → - (id+id)	E → - (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.

Ambiguity:

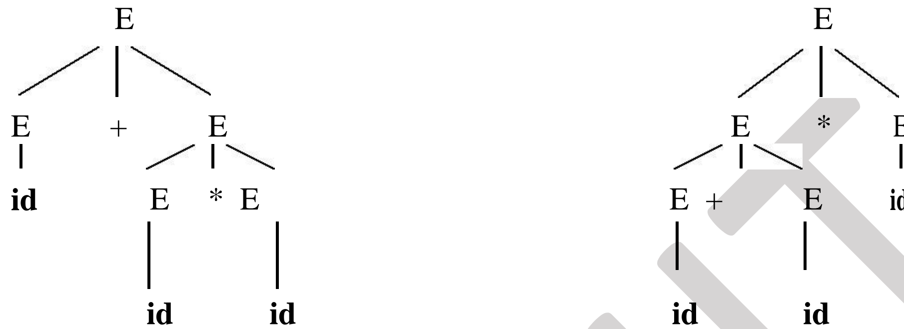
A grammar that produces more than one parse for some sentence is said to be **ambiguous grammar**.

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 → E+ E	E → E* E
E → id + E	E → E + E * E
E → id + E * E	E → id + E * E
E → id + id * E	E → id + id * E
E → id + id * id	E → id + id * id

2106-JIT



WRITING A GRAMMAR

There are four categories in writing a grammar:

1. Regular Expression Vs Context Free Grammar
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.

Regular Expressions vs. Context-Free Grammars:

REGULAR EXPRESSION	CONTEXT-FREE GRAMMAR
It is used to describe the tokens of programming languages.	It consists of a quadruple where S → start symbol, P → production, T → terminal, V → variable or non- terminal.
It is used to check whether the given input is valid or not using **transition diagram.**	It is used to check whether the given input is valid or not using **derivation**.
The transition diagram has set of states and edges.	The context-free grammar has set of productions.
It has no start symbol.	It has start symbol.
It is useful for describing the structure of lexical constructs such as identifiers, constants, keywords, and so forth.	It is useful in describing nested structures such as balanced parentheses, matching begin-end's and so on.

2106-JIT

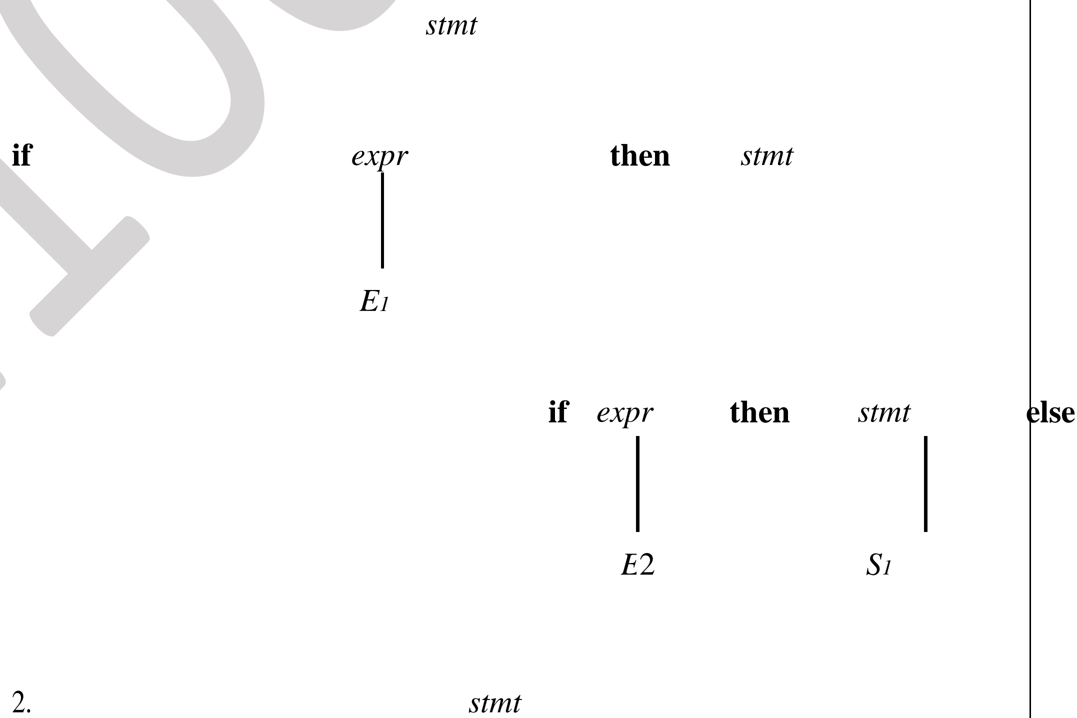
- The lexical rules of a language are simple and RE is used to describe them.
- Regular expressions provide a more concise and easier to understand notation for tokens than grammars.
- Efficient lexical analyzers can be constructed automatically from RE than from grammars.
- Separating the syntactic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

Eliminating ambiguity:

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

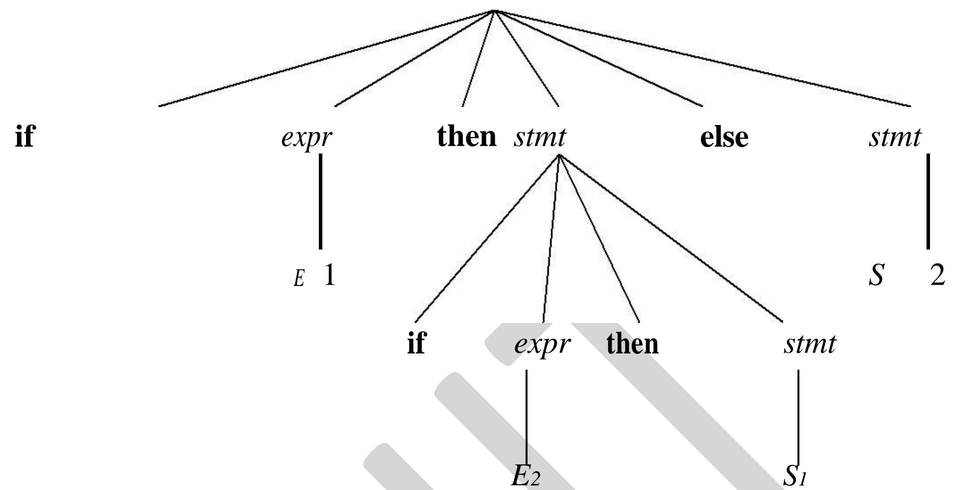
Consider this example, G: *stmt* → **if** *expr* **then** *stmt* | **if** *expr* **then** *stmt* **else** *stmt* | **other**

This grammar is ambiguous since the string **if E_1 then if E_2 then S_1 else S_2** has the following two parse trees for leftmost derivation:



2. *stmt*

2106-JIT



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*

2106-JIT

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 → FT'

T' → *FT' | ε

F → (E) |id

Algorithm to eliminate left recursion:

1. Arrange the non-terminals in some order $A_1, A_2 \ldots A_n$.

2106-JIT

2. **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 A_i-productions
end

2106-JIT

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 $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$, it can be rewritten as

$$A \rightarrow \alpha A'$$

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

Consider the grammar , G : S → iEtS | iEtSeS | a
E → b

Left factored, this grammar becomes

S → iEtSS' | a
S'→ eS |ε
E → b

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.

Parse tree:

Graphical representation of a derivation or deduction is called a parse tree. Each interior node of the parse tree is a non-terminal; the children of the node can be terminals or non-terminals.

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

2106-JIT

symbol.
Example : LR Parsers.

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.

2106-JIT

Types of top-down parsing :

1. Recursive descent parsing
2. Predictive parsing

1. <u>RECURSIVE DESCENT PARSING</u>

- Recursive descent parsing is one of the top-down parsing techniques that uses a set of recursive procedures to scan its input**.**
- This parsing method may involve **backtracking**, that is, making repeated scans of the input.

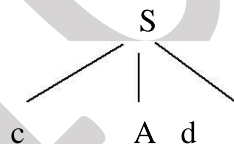
<u>Example for backtracking</u> :

Consider the grammar G : S → cAd
A → ab |a
and the input string w=cad.

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

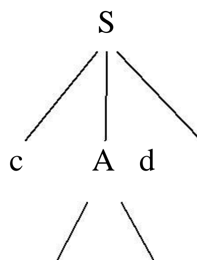
Step1:

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.



2106-JIT

a b

Step3:

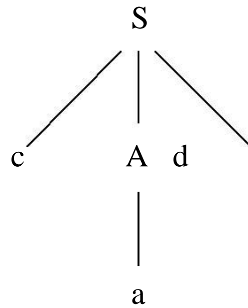
The second symbol 'a' of w also matches with second leaf of tree. So advance the input pointer to third symbol of w 'd'**.** But the third leaf of tree is b which does not match with the input symbol **d.**

2106-JIT

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.

PREDICTIVE PARSER:

It is implemented by using non backtracking concept. It is categorized into two types,

1. recursive Predictive parser
2. non recursive Predictive parser

RECURSIVE PREDICTIVE PARSER

It is implemented by using recursion mechanism. A left-recursive grammar can cause a recursive Predictive parser to go into an infinite loop. Hence, **elimination of left-recursion** must be done before parsing.

Consider the grammar for arithmetic expressions

E → E+T |T

T → T*F |F

F→ (E) |id

After eliminating the left-recursion the grammar

becomes, E → TE'

E' → +TE' | ε

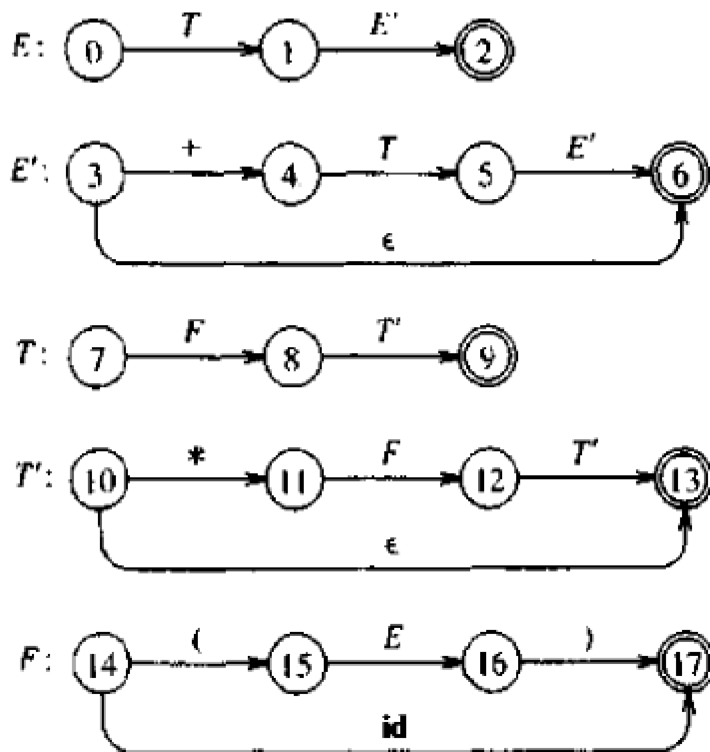
2106-JIT

T → FT’

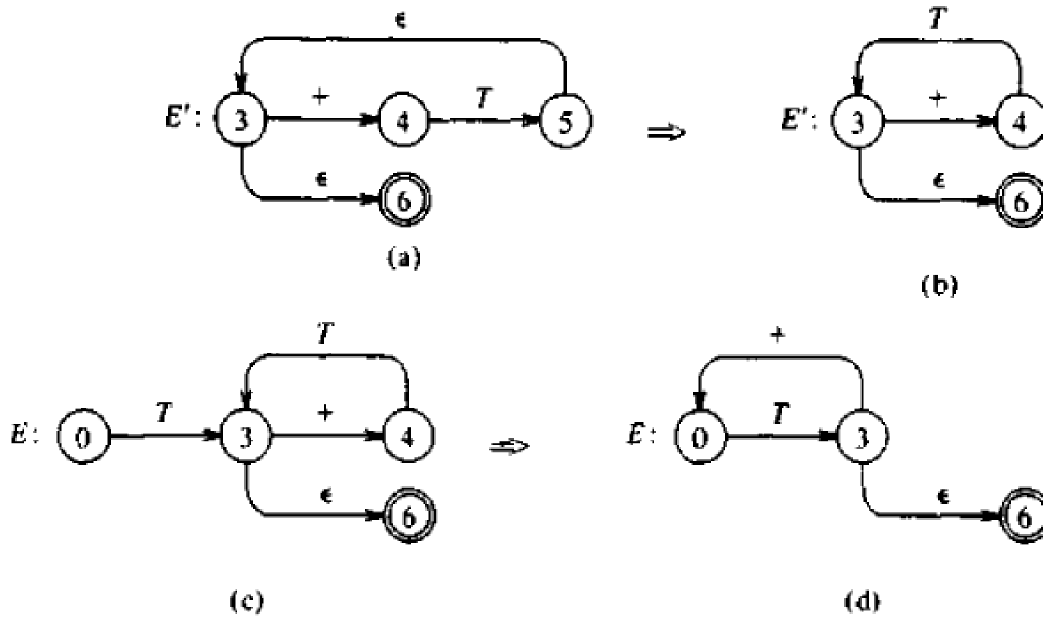
T’ → *FT’ | ε

F → (E) |id

Now we can draw transition diagram for each of the non terminal,

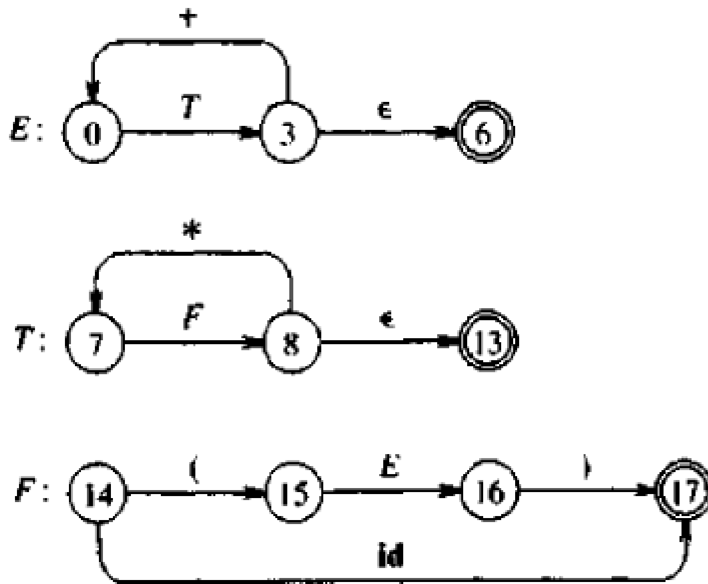


From the above diagram we can reduce the number of states in E’ diagram and substitute it in Nonterminal E diagram as follows,



Similarly reducing T' Diagram, the resultant transition diagram are,

2106



write the procedure based on above grammar as follows:

<u>Recursive procedure</u>:

Procedure E()
begin
T( );
EPRIME( );
end

Procedure EPRIME( )

begin
If input_symbol='+' then ADVANCE( );
T( ); EPRIME( );

End
Procedure T( )
begin
F( ); TPRIME( );
end

2106-JIT

Procedure TPRIME( )
begin
If input _symbol=’*’ then ADVANCE( );
F( ); TPRIME( );
end

Procedure F( )
begin
If input -symbol=’id’ then ADVANCE( );
else if input-symbol=’(‘ then ADVANCE( );
E( );
else if input-symbol=’)’ then ADVANCE( );
end

else ERROR( );

2. <u>PREDICTIVE PARSING (Non Recursive Predictive Parser)</u>

- Predictive parsing is a special case of recursive descent parsing where no backtracking is required.
- The key problem of predictive parsing is to determine the production to be applied for a non- terminal in case of alternatives.

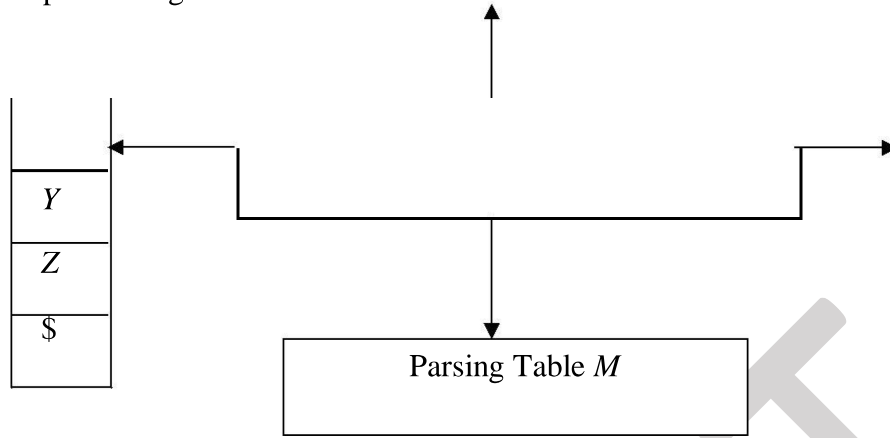
Non-recursive predictive parser

INPUT

	a	+	*b*	$

STACK *X* Predictive parsing program OUTPUT

2106-JIT



2106-JIT

The table-driven predictive parser has an input buffer, stack, a parsing table and an output stream.

Input buffer:

It consists of strings to be parsed, followed by $ to indicate the end of the input string.

Stack:

It contains a sequence of grammar symbols preceded by $ to indicate the bottom of the stack. Initially, the stack contains the start symbol on top of $.

Parsing table:

It is a two-dimensional array $M[A, a]$, where **'*A*'** is anon-terminal and **'*a*'** is aterminal.

Predictive parsing program:

The parser is controlled by a program that considers X, the symbol on top of stack, and a, the current input symbol. These two symbols determine the parser action. 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 non-terminal , the program consults entry $M[X, a]$ of the parsing table M. This entry will either be an X-production of the grammar or an error entry.
   If $M[X , a] = \{X \rightarrow UVW\}$,the parser replaces X on top of the stack by
   WVU. If $M[X , a]$ = **error**, the parser calls an error recovery routine.

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 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 as follows:

set *ip* to point to the first symbol of *w*$;

2106-JIT

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 e*rror*()

else /* X is a non-terminal */

if $M[X, a] = X \rightarrow Y_1 Y_2 \dots Y_k$ **then begin**

2106-JIT

pop X from the stack;
push Y_k, Y_{k-1}, … ,Y_1 onto the stack, with Y_1 on top;
output the production $X \rightarrow Y_1 Y_2 \ldots Y_k$
end
else *error*()
until $X = \$$ /* stack is empty */

Predictive parsing table construction:

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

1. FIRST

2. FOLLOW

Rules for first( ):

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

2. If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X).

3. If X is non- terminal and $X \rightarrow a\alpha$ is a production then add a to FIRST(X).

4. If X is non- terminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ 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, $Y_1,\ldots Y_{i-1} \Rightarrow \varepsilon$. 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 $A \rightarrow \alpha B\beta$, 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).

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**.

2106-JIT

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 :

2106-JIT

TERMINAL							
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)		

2106-JIT

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

<u>LL(1) grammar</u> :

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

2106-JIT

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 }

2106-JIT

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.

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.

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**. Other type is called LR 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 sentence to be recognized is **abbcde.**

2106-JIT

Reduction (leftmost)		**Rightmost derivation**
a**b**bcde	(A → b)	**S** → aA**B**e
a**Abc**de	(A → Abc)	→ a**A**de
aA**d**e	(B → d)	→ a**A**bcde
aABe	(S → aABe)	→ abbcde
S		

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 → E+E
E → E*E
E → (E)
E → id

And the input string $id_1+id_2*id_3$

The rightmost derivation is :

E → **E+E**
→ E+**E*E**
→ E+E***id_3**
→ E+**id_2***id_3
→ **id_1**+id_2*id_3

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 nth right-

2106-JIT

sentinel form of some rightmost derivation.

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→id
$ E	$+id_2*id_3$ $	shift
$ E+	id_2*id_3 $	shift
$ E+id_2	$*id_3$ $	reduce by E→id
$ E+E	$*id_3$ $	shift
$ E+E*	id3 $	shift
$ E+E*id3	$	reduce by E→id
$ E+E*E	$	reduce by E→ E *E
$ E+E	$	reduce by E→ E+E
$ E	$	accept

Actions in shift -reduce parser:

2106-JIT

- 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 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.

2106-JIT

Viable prefixes:

- α is a viable prefix of the grammar if there is *w* such that α*w* is a right sentinel form.
- The set of prefixes of right sentinel forms that can appear on the stack of a shift-reduce parser are called viable prefixes.
- The set of viable prefixes is a regular language.

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.

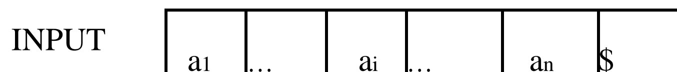
Types of LR parsing method:

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.

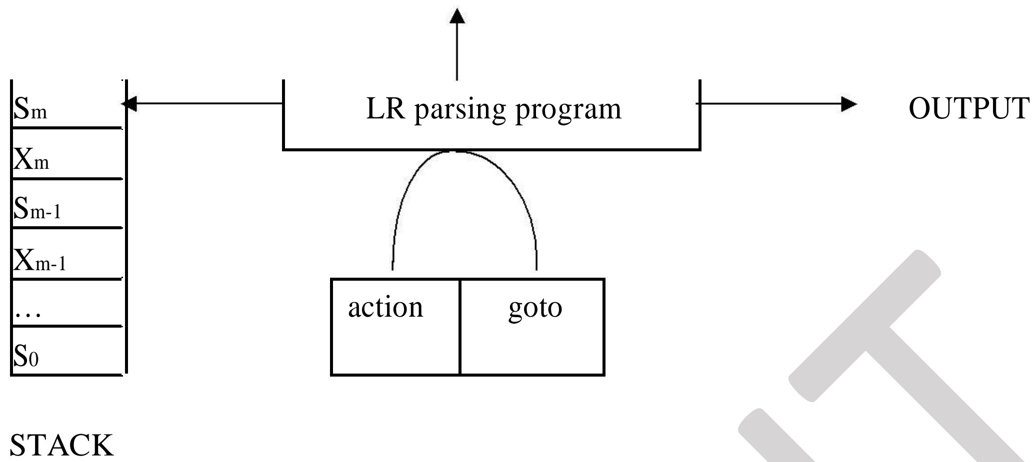
The LR parsing algorithm:

The schematic form of an LR parser is as follows:

INPUT



2106-JIT



STACK

2106-JIT

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 → β,
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;
```


2106-JIT

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

2106-JIT

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.

Algorithmfor construction of SLR parsing table:

Input : An augmented grammar G'
Output : The SLR parsing table functions *action* and *goto* for G'
Method :

2106-JIT

1. Construct C ={I_0, I_1, ... 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).

2106-JIT

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

2106-JIT

F → . (E)
F → . id

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

GOTO (I_4 , id)
I_5 : F→ id .

2106-JIT

I_8 : F→ (E .) E → E . + T

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

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

GOTO (I_0 , F)
I_3 : T → F .

GOTO (I_4 , 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)

2106-JIT

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

2106-JIT

GOTO (I_9 , *)

I_7 : T → T * . F
F→ . (E)

F → . id

2106-JIT

GOTO (I_4 , ()

I_4 : F → (. E)

E → . E + T

E → . T

T → . T * F

T → . F

F → . (E)

F → id

FOLLOW (E) = { $,) , +)

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

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

SLR parsing table:

	ACTION						GOTO		
	id	+	*	(	)	$	**E**	**T**	**F**
I0	s5			s4			1	2	3
I1		s6				ACC			
I2		r2	s7		r2	r2			
I3		r4	r4		r4	r4			
I4	s5			s4			8	2	3
I5		r6	r6		r6	r6			
I6	s5			s4				9	3
I7	s5			s4					10
I8		s6			s11				
I9		r1	s7		r1	r1			
I10		r3	r3		r3	r3			

2106-JIT

I_{11}		r5	r5		r5	r5			

Blank entries are error entries.

Stack implementation:

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

2106-JIT

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

2106-JIT

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 1 + 6 T 9 * 7 id 5	$	GOTO (I_5 , $) = r6 ; **reduce** by F→ id
0 E 1 + 6 T 9 * 7 F 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**

2106-JIT

CANONICAL LR PARSING:

Example:

S → CC
C →cC|d.

1. Number the grammar productions:

1. S →CC
2. C →cC
3. C →d

2. The Augmented grammar is:

S^{I} →S

S →CC
C →cC
C →d.

3.Constructing the sets of LR(1) items:

We begin with:

S^{1} →.S,$ begin with look-a-head (LAH) as $. Here after with the help of closure other items are added.

Closure():

For the production A->α.Bβ,a , Function closure tells us to add [B→.r,b] for each production B→r and terminal b in FIRST (βa). Now B→r must be S→CC, and since β is ∈ and a is $, b may only be $. Thus,

S→.CC,$

We continue to compute the closure by adding all items [C→.r,b] for b in FIRST [C$] i.e., matching [S→.CC,$] against [A→α.Bβ,a] we have, A=S, α=∈, B=C and a=$. FIRST (C$) = FIRST ©

FIRST© = {c,d} We add items:

C→.cC,C

C→cC,d

C→.d,c

2106-JIT

$C \rightarrow .d,d$

None of the new items have a non-terminal immediately to the right of the dot, so we have completed our first set of LR(1) items. The initial I_0 items are:

I_0 : $S^I \rightarrow .S,\$$ $S \rightarrow .CC,\$$ $C \rightarrow .CC,c/d$ $C \rightarrow .d.c/d$

Now we start computing goto (I_0,X) for various non-terminals i.e.,

Goto (I_0,S):

I_1 : $S^I \rightarrow S.,\$$ → reduced item.

Goto (I_0,C)

I_2 : $S \rightarrow C.C, \$$

$C \rightarrow .cC,\$$

$C \rightarrow .d,\$$

Goto (I_0,c) :

I_3 : $C \rightarrow c.C,c/d$

$C \rightarrow .cC,c/d$

$C \rightarrow .d,c/d$

Goto (I_0,d)

I_4 $C \rightarrow d., c/d$ → reduced item.

Goto (I_2,C)

I_5 $S \rightarrow CC.,\$$ → reduced item.

Goto (I_2,c)

2106-JIT

I_6 C→c.C,$

C→.cC,$

C→.d,$

Goto (I_2,d)

I_7 C→d.,$ → reduced item.

Goto (I_3,C)

I_8 C→cC.,c/d → reduced item.

Goto ($I_{3,c}$) I_3

C→c.C, c/d

C→.cC,c/d

C→.d,c/d

Goto (I_3,d) I_4

C→d.,c/d. → reduced item.

Goto (I_6,C)

I_9 C→cC.,$ → reduced item.

Goto (I_6,c) I_6

C→c.C,$

C→,cC,$

C→.d,$

Goto (I_6,d) I_7

C→d.,$ → reduced item.

All are completely reduced. So now we construct the canonical LR (1) parsing table –

Here there is no need to find FOLLOW ( ) set, as we have already taken look-a-head for each set of productions while constructing the states.

Constructing LR(1) Parsing table:

	Action			goto	
State	C	D	$	S	C
0	S3	S4		1	2

2106-JIT

1			Accept		
2	S6	S7			5
3	S3	S4			8
4	R4	R4			
5			R1		
6	S6	S7			9
7			R4		
8	R3	R3			
9			R3		

1. Consider I0 items:

The item S→.S.$ gives rise to goto [I0,S] = I1 so goto [0,s] = 1.

The item S→.CC, $ gives rise to goto [I0,C] = I2 so goto [0,C] = 2.

The item C→.cC, c/d gives rise to goto [I0,c] = I3 so goto [0,c] = shift 3

The item C→.d, c/d gives rise to goto [I0,d] = I4 so goto [0,d] = shift 4

2. Consider I0 items:

The item S^{I}→S.,$ is in I1, then set action [1,$] = accept

3. Consider I2 items:

The item S→C.C,$ gives rise to goto [I2,C] = I5. so goto [2,C] = 5

The item C→.cC, $ gives rise to goto [I2,c] = I6. so action [0,c] = shift 6. The item C→.d,$ gives rise to goto [I2,d] = I7. so action [2,d] = shift 7

4. Consider I3 items:

The item C→.cC, c/d gives rise to goto [I3,C] = I8. so goto [3,C] = 8

The item C→.cC, c/d gives rise to goto [I3,c] = I3. so action [3,c] = shift 3. The item C→.d, c/d gives rise to goto [I3,d] = I4. so action [3,d] = shift 4.

5. Consider I4 items:

The item C→.d, c/d is the reduced item, it is in I4 so set action [4,c/d] to reduce c→d. (production rule no.3)

6. Consider I5 items:

2106-JIT

The item S→CC.,$ is the reduced item, it is in I5 so set action [5,$] to S→CC (production rule no.1)

7. Consider I6 items:

The item C→c.C,$ gives rise to goto [I6 ,C] = I9. so goto [6,C] = 9

The item C→.cC,$ gives rise to goto [I6 ,c] = I6. so action [6,c] = shift 6

The item C→.d,$ gives rise to goto [I6 ,d] = I7. so action [6,d] = shift 7

8. Consider I7 items:
The item C→d., $ is the reduced item, it is in I7.

So set action [7,$] to reduce C→d (production no.3)

9. Consider I8 items:

The item C→cC.,c/d in the reduced item, It is in I_8, so set action[8,c/d] to reduce C→cC (production rule no .2)

10. Consider I9 items:

The item C →cC, $ is the reduced item, It is in I9, so set action [9,$] to reduce C→cC (Production rule no.3)

If the Parsing action table has no multiply –defined entries, then the given grammar is called as LR(1) grammar

<u>LALR PARSING:</u>

Example:

1. Construct C={I0,I1,… .... ,In} The collection of sets of LR(1) items

2. For each core present among the set of LR (1) items, find all sets having that core, and replace there sets by their Union# (group them into a single term)

I_0 →same as previous

I_1 → “

I_2 → “

I_{36} –> Clubbing item I3 and I6 into one I36 item.

2106-JIT

C →cC,c/d/$

C→cC,c/d/$
C→d,c/d/$

I_5 →same as previous

I_{47}-> Clubbing item I4 and I7 into one I47 item

C→d,c/d/$

I_{89} → Clubbing item I8 and I9 into one I89 item

C→cC, c/d/$

LALR parsing table construction:

State	Action			Goto	
	c	d			C

The Parser Generator Yacc

A translator can be constructed using Yacc in the manner illustrated in Fig. 4.55. First, a file, say `translate.y`, containing a Yacc specification of the translator is prepared. The UNIX system command

```
yacc translate.y
```

transforms the file `translate.y` into a C program called `y.tab.c` using the LALR method outlined in Algorithm 4.13. The program `y.tab.c` is a representation of an LALR parser written in C, along with other C routines that the user may have prepared. The LALR parsing table is compacted as described in Section 4.7. By compiling `y.tab.c` along with the `ly` library,

The Parser Generator Yacc

A translator can be constructed using Yacc in the manner illustrated in Fig. 4.55. First, a file, say `translate.y`, containing a Yacc specification of the translator is prepared. The UNIX system command

```
yacc translate.y
```

transforms the file `translate.y` into a C program called `y.tab.c` using the LALR method outlined in Algorithm 4.13. The program `y.tab.c` is a representation of an LALR parser written in C, along with other C routines that the user may have prepared. The LALR parsing table is compacted as described in Section 4.7. By compiling `y.tab.c` along with the `ly` library,

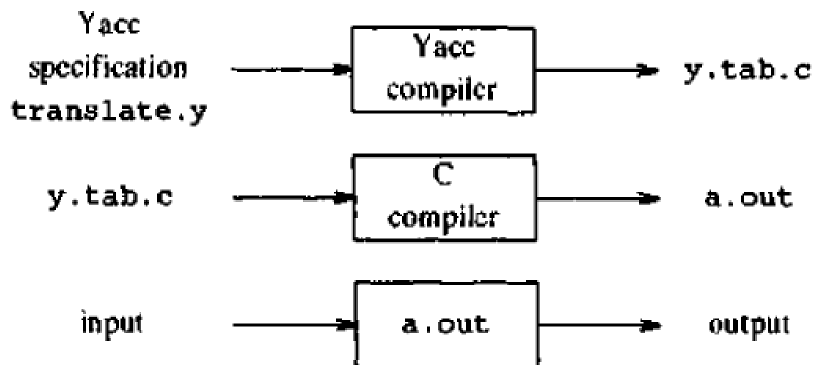


Fig. 4.55. Creating an input/output translator with Yacc.

that contains the LR parsing program using the command

```
cc y.tab.c -ly
```

we obtain the desired object program `a.out` that performs the translation specified by the original Yacc program.[6] If other procedures are needed, they can be compiled or loaded with `y.tab.c`, just as with any C program.

A Yacc source program has three parts:

```
declarations
%%
translation rules
%%
supporting C-routines
```

The declarations part. There are two optional sections in the declarations part of a Yacc program. In the first section, we put ordinary C declarations, delimited by %{ and %}. Here we place declarations of any temporaries used by the translation rules or procedures of the second and third sections. In

production and the associated semantic action. A set of productions that we have been writing

<left side> → <alt 1> | <alt 2> | · · · | <alt n>

would be written in Yacc as

```
<left side>    :   <alt 1>    { semantic action 1 }
               |   <alt 2>    { semantic action 2 }
            . . .
               |   <alt n>    { semantic action n }
               ;
```

The supporting C-routines part. The third part of a Yacc specification consists of supporting C-routines. A lexical analyzer by the name `yylex()` must be provided. Other procedures such as error recovery routines may be added as necessary.

The lexical analyzer `yylex()` produces pairs consisting of a token and its associated attribute value. If a token such as `DIGIT` is returned, the token must be declared in the first section of the Yacc specification. The attribute value associated with a token is communicated to the parser through a Yacc-defined variable `yylval`.