**UNIT-2**

**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.



**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:**

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* → ↑

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.

**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.

**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

The two corresponding parse trees are :



**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.

**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 E1 then if E2 then S1 else S2** has the following

two parse trees for leftmost derivation :



**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 → FT’

T’ → \*FT’ | ε

F → (E) | id

**Algorithm to eliminate left recursion:**

1. Arrange the non-terminals in some order A1, A2 . . . An.

2. **for** *i* := 1 **to** *n* **do begin**

**for** *j* := 1 **to** *i*-1 **do begin**

replace each production of the form Ai → Aj γ

by the productions Ai → δ1 γ | δ2γ | . . . | δk γ

where Aj → δ1 | δ2 | . . . | δk are all the current Aj-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 → αβ1 | αβ2 , it can be rewritten as**

**A → αA’**

**A’ → β1 | β2**

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 nonterminals.

**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.

**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

**1. RECURSIVE DESCENT PARSING**

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.

**Example for backtracking :**

Consider the grammar G : S → cAd

A → ab | a

and the input string w=cad.

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

**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.

**Example for recursive decent parsing:**

A left-recursive grammar can cause a recursive-descent 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

T → FT’

T’ → \*FT’ | ε

F → (E) | id

Now we can write the procedure for grammar as follows:

**2. PREDICTIVE PARSING**

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**



 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 a non-terminal and **‘*a*’** is a terminal.

**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* ≠ $, 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* → *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*$;

**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* →*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* → *Y1 Y2 . . . Yk*

**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* → ε is a production, then add ε to FIRST(*X*).

3. If *X* is non-terminal and X → *a*α is a production then add *a* to FIRST(X).

4. If X is non-terminal and *X* → *Y1 Y2*…*Yk* is a production, then place *a* in FIRST(*X*) if for some

*i*, *a* is in FIRST(*Yi*), and ε is in all of FIRST(*Y1*),…,FIRST(*Yi-1*); that is, *Y1*,….*Yi-1* => ε. If ε is

in FIRST(*Yj*) 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* → α*B*β, then everything in FIRST(β) except ε is placed in

follow(*B*).

3. If there is a production *A* → α*B*, or a production *A* → α*B*β 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**.

**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) = {+, \* , $ , ) }



**Stack implementation:**



**LL(1) grammar:**

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:**



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

**Actions performed in predictive parsing:**

1. Shift

2. Reduce

3. Accept

4. Error

**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**.

**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.**

 **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 id1+id2\*id3

The rightmost derivation is :

E → **E**+**E**

→ E+**E\*E**

→ E+E**\*id3**

→ E+**id2\***id3

→ **id1**+id2**\***id3

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* = γ*n*, where γ*n* is the *n*th rightsentinel

form of some rightmost derivation.

**Stack implementation of shift-reduce parsing :**

****

**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 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.

**1. Shift-reduce conflict:**

**Example:**

Consider the grammar:

E→E+E | E\*E | id and input id+id\*id



**. Reduce-reduce conflict:**

Consider the grammar:

M → R+R | R+c | R

R → c

and input c+c

****

**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.

**OPERATOR-PRECEDENCE PARSING**

An efficient way of constructing shift-reduce parser is called operator-precedence parsing.

Operator precedence parser can be constructed from a grammar called Operator-grammar. These

grammars have the property that no production on right side is ɛ or has two adjacent nonterminals.

**Example:**

Consider the grammar:

E → EAE | (E) | -E | id

A → + | - | \* | / | ↑

Since the right side EAE has three consecutive non-terminals, the grammar can be written as

follows:

E → E+E | E-E | E\*E | E/E | E↑E | -E | id

**Operator precedence relations:**

There are three disjoint precedence relations namely

**< .** - less than

**=** - equal to**.**

 **>** - greater than

The relations give the following meaning:

a **< .** b – a yields precedence to b

a = b – a has the same precedence as b

a **. >** b – a takes precedence over b

**Rules for binary operations:**

1. If operator θ1 has higher precedence than operator θ2, then make

θ1**. >** θ2 and θ2 **< .** θ1

2. If operators θ1 and θ2, are of equal precedence, then make

θ1**. >** θ2 and θ2**. >** θ1 if operators are left associative

θ1 **< .** θ2 and θ2 **< .** θ1 if right associative

3. Make the following for all operators θ:

θ **< .** id , id **. >** θ

θ **< .** ( , ( **< .** θ

) **. >** θ , θ **. >** )

θ **. >** $ , $ **< .** θ

Also make

( = ) , ( **< .** ( , ) **. >** ) , ( **< .** id , id **. >** ) , $ **< .** id , id **. >** $ , $ **< .** ( , ) **. >** $

**Example:**

Operator-precedence relations for the grammar

E → E+E | E-E | E\*E | E/E | E↑E | (E) | -E | id is given in the following table assuming

1. ↑ is of highest precedence and right-associative

2. \* and / are of next higher precedence and left-associative, and

3. + and - are of lowest precedence and left-associative

Note that the **blanks** in the table denote error entries.

****

**Operator precedence parsing algorithm:**

**Input :** An input string *w* and a table of precedence relations.

**Output :** If *w* is well formed, a *skeletal* parse tree **,**with a placeholder non-terminal E labeling all

interior nodes; otherwise, an error indication.

**Method :** Initially the stack contains $ and the input buffer the string *w* $. To parse, we execute

the following program :

(1) Set *ip* to point to the first symbol of *w*$;

(2) **repeat forever**

(3) **if** $ is on top of the stack and *ip* points to $ **then**

(4) **return**

**else begin**

(5) let *a* be the topmost terminal symbol on the stack

and let *b* be the symbol pointed to by *ip;*

(6) **if** *a* <. *b* or *a* = *b* **then begin**

(7) push *b* onto the stack;

(8) advance *ip* to the next input symbol;

**end;**

(9) **else if** *a* . > *b* **then /\***reduce**\*/**

(10) **repeat**

(11) pop the stack

(12) **until** the top stack terminal is related by <.

to the terminal most recently popped

(13) **else** error( )

**end**

**Stack implementation of operator precedence parsing:**

Operator precedence parsing uses a stack and precedence relation table for its

implementation of above algorithm. It is a shift-reduce parsing containing all four actions shift,

reduce, accept and error.

The initial configuration of an operator precedence parsing is

STACK INPUT

$ w $

where w is the input string to be parsed.

**Example:**

Consider the grammar E → E+E | E-E | E\*E | E/E | E↑E | (E) | id. Input string is **id+id\*id** .The

implementation is as follows:

**Advantages of operator precedence parsing:**

1. It is easy to implement.

2. Once an operator precedence relation is made between all pairs of terminals of a grammar ,

the grammar can be ignored. The grammar is not referred anymore during implementation.

**Disadvantages of operator precedence parsing:**

1. It is hard to handle tokens like the minus sign (-) which has two different precedence.

2. Only a small class of grammar can be parsed using operator-precedence parser.

**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 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:



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 s0X1s1X2s2…Xmsm, where sm is on

top. Each Xi is a grammar symbol and each si is a state.

The parsing table consists of two parts : *action* and *goto* functions.

**Action** : The parsing program determines sm, the state currently on top of stack, and ai, the

current input symbol. It then consults *action*[sm,ai] 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 s0 on its stack, where s0 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 = {I0, I1, …. In}, 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 Ii and goto(Ii,*a*) = Ij, then set *action*[*i*,*a*] to “shift j”. Here *a* must be

terminal.

(b) If [A→α∙] is in Ii , then set *action*[*i*,*a*] to “reduce A→α” for all *a* in FOLLOW(A).

(c) If [S’→S.] is in Ii, 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*(Ii,A) = Ij, 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.

I0 : E’ → **.** E

E → **.** E + T

E → **.** T

T → **.** T \* F

T → **.** F

F → **.** (E)

F → **.** id

GOTO ( I0 , E) GOTO ( I4 , id )

I1 : E’ → E **.** I5 : F → id **.**

E → E **.** + T

GOTO ( I6 , T )

GOTO ( I0 , T) I9 : E → E + T **.**

I2 : E → T **.** T → T **.** \* F

T → T **.** \* F

GOTO ( I6 , F )

GOTO ( I0 , F) I3 : T → F **.**

I3 : T → F **.**

GOTO ( I6 , ( )

I4 : F → ( **.** E )

GOTO ( I0 , ( ) GOTO ( I6 , id)

I4 : F → ( **.** E) I5 : F → id **.**

E → **.** E + T

E → **.** T GOTO ( I7 , F )

T → **.** T \* F I10 : T → T \* F **.**

T → **.** F

F → **.** (E) GOTO ( I7 , ( )

F → **.** id I4 : F → ( **.** E )

E → **.** E + T

GOTO ( I0 , id ) E → **.** T

I5 : F → id **.** T → **.** T \* F

T → **.** F

GOTO ( I1 , + ) F → **.** (E)

I6 : E → E + **.** T F → **.** id

T → **.** T \* F

T → **.** F GOTO ( I7 , id )

F → **.** (E) I5 : F → id **.**

F → **.** id

GOTO ( I8 , ) )

GOTO ( I2 , \* )

I7 : T → T \* **.** F

F → **.** (E) GOTO ( I8 , + )

F → **.** id

I6 : E → E + **.** T

T → **.** T \* F

GOTO ( I4 , E ) T → **.** F

I8 : F → ( E **.** ) F → **.** ( E )

E → E **.** + T F → **.** id

GOTO ( I4 , T) GOTO ( I9 , \*)

I11 : F → ( E ) **.**

I2 : E →T **.** I7 : T → T \* **.** F

T → T **.** \* F F → **.** ( E )

F → **.** id

GOTO ( I4 , F)

I3 : T → F **.**

GOTO ( I4 , ( )

I4 : F → ( **.** E)

E → **.** E + T

E → **.** T

T → **.** T \* F

T → **.** F

F → **.** (E)

F → id

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

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

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

****

Blank entries are error entries.

**Stack implementation:**

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

