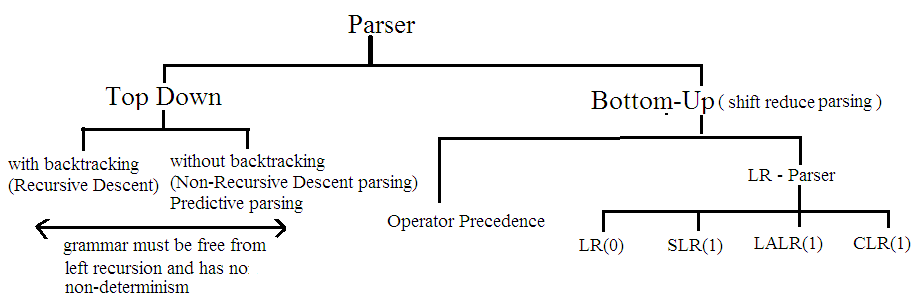
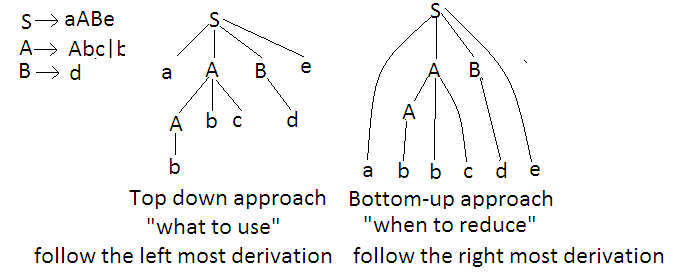
**Parser**



Example:-



### Top-down Parsing:-

When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing.

**Recursive Descent Parsing(Uses Backtracking):-**

The idea of recursive-descent parsing is extremely simple.We view the grammar rule for a non-terminal A as a definition for a procedure that will recognize an A. The right hand side of the grammar rule for A specifies the structure of the code for this procedure , the sequence of terminals and non-terminal in a choice correspond to matches of the input and calls other procedure ,while choices correspond to alternatives (switch case or if statement) within the code.

Ex:-



main()

{

E();

if(l==’$’)

printf(“parsing successful”);

}

E()

{

if(l==’i’)

{

match(‘i’);

();

}

}

( )

{

if(l==’+’)

{

match(‘+’);

match(‘i’);

( );

}

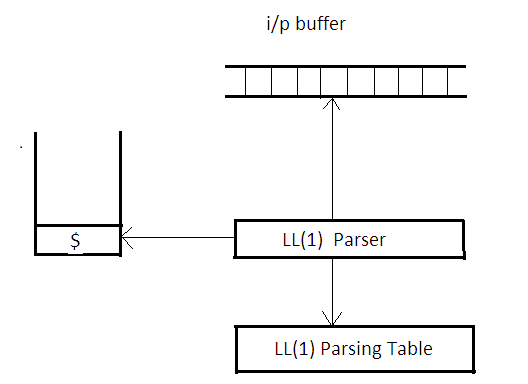
else

return;

}

**Disadvantage:-** Stack overflow

**Non-Recursive Descent Parsing(LL(1) Parsing):-**



**Algorithms:-**

repeat

let X be the top stack symbol and ‘**a’** the symbol pointed by pointer

if X is a terminal or $ then

if X=a then

pop X from the stack and move pointer forward

else

error()

else

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

else

error();

Until X=$

**Constructing LL(1) Parsing Table (Predective Parsing):-**

Method:-

1. For each production A→α of the grammar do step 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.

**To compute First and Follow of all grammar symbol:-**

Ex.

|  |  |  |
| --- | --- | --- |
|  | **First** | **Follow** |
| S→ABCD | {b,c} | {$} |
| A→b│ϵ | {b,ϵ} | {c} |
| B→c | {c} | {d} |
| C→d | {d} | {e} |
| D→e | {e} | {$} |
| Exa | | |
| S→ABCDE | {a,b,c} | {$} |
| A→a│ϵ | {a,ϵ} | {b,c} |
| B→ b│ϵ | {b,ϵ} | {c} |
| C→ c | {c} | {d,e,$} |
| D→ d│ϵ | {d,ϵ} | {e,$} |
| E→ e│ϵ | {e,ϵ} | {$} |
|  |  |  |
| S→Bb│Cd | {a,b,c,d} | {$} |
| B→aB│ϵ | {a,ϵ} | {b} |
| C→cC│ϵ | {c,ϵ} | {d} |
|  |  |  |
| E→ | {id,(} | {$,)} |
|  | {+,ϵ} | {$,)} |
| T→F | {id,(} | {+,$,)} |
|  | {\*,ϵ} | {+,$,)} |
| F→id│(E) | {id,(} | {\*,+,$,)} |
|  |  |  |
| S→ACB│CbB│Ba | {d,g,h,b,a,ϵ} | {$} |
| A→da│BC | {d,g,h,ϵ} | {h,g,$} |
| B→g│ϵ | {g,ϵ} | {$,a,h,g} |
| C→h│ϵ | {h,ϵ} | {g,b,h,$} |
|  |  |  |
| S→aABb | {a} | {$} |
| A→c│ϵ | {c, ϵ} | {d,b} |
| B→d│ϵ | {d,ϵ } | {b} |
|  |  |  |
| S→aBDh | {a} | {$} |
| B→cC | {c} | {g,f,h} |
| C→bC│ϵ | {b,ϵ} | {g,f,h} |
| D→EF | {g, f, ϵ } | {h} |
| E→g│ϵ | {g, ϵ } | {f,h} |
| F→f│ϵ | {f, ϵ } | {h} |

**Ex.**

Parse the string ***id+id\*id*** with the grammar below by using LL(1) parsing techniques or predictive parsing techniques.

E→

T→F

F→id│(E)

**Solution:-**

**LL(1) or predective parsing table**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **id** | **+** | **\*** | **(** | **)** | **$** |
| E | E→ |  |  | E→ |  |  |
|  |  |  |  |  |  |  |
| T | T→F |  |  | T→F |  |  |
|  |  |  |  |  |  |  |
| F | F→id |  |  | F→(E) |  |  |

**Working:-**

|  |  |  |
| --- | --- | --- |
| **Stack** | **Input** | **Output** |
| $E | id+id\*id$ |  |
| $ | id+id\*id$ | E→ |
| $ | id+id\*id$ | T→F |
| $ | id+id\*id$ | F→id |
| $ | +id\*id$ |  |
| $ | +id\*id$ |  |
| $ | +id\*id$ |  |
| $ | id\*id$ |  |
| $ | id\*id$ | T→F |
| $ | id\*id$ | F→id |
| $ | \*id$ |  |
| $ | \*id$ |  |
| $ | id$ |  |
| $ | id$ | F→id |
| $ | $ |  |
| $ | $ |  |
| $ | $ |  |

**Bottom-up Parsing(Shift -Reduce Parsing):-**

In Bottom-up parsing we start with sentence and try to apply the production rules in reverse order to finish up with the start symbol of the grammar.This corresponds to starting at leaves of the parse tree , and working back to root. Bottom –up parsing is also known as shift-reduce parsing.

**Ex.-**

Parse the string **abbcde** by following grammar using Shift -Reduce technique.

S→aABe

A→Abc│b

B→d

**Solution:-**

abbcde→aAbcde→aAde→aABe→S

**Handle:-**

A handle is a substring that matches the right hand side of a production and replacing RHS by LHS must be step in the reverse rightmost derivation that ultimately leads to the start symbol.If replacing a string does not ultimately lead to the start symbol it can not be handle.

|  |  |  |
| --- | --- | --- |
| **String** | **Handle** | **Production Rule** |
| abbcde | b | A→b |
| aAbcde | Abc | A→Abc |
| aAde | d | B→d |
| aABe | aABe | S→aABe |

**Operator Precedence Parsing:-**

**Operator grammar:-** In an operator grammar,no production rule can have

* ϵ at the right side
* two adjacent non-terminal at the right side

**Ex.**

E→E+E│E\*E│id (operator grammar)

E →EAE│id (not operator grammar)

A→ +│\*

S→SAS│a (not operator grammar)

A→bSb│b

S→SbSbS│SbS│a (operator grammar)

A→bSb│b

**Operator-precedence table:-**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | id | + | \* | $ |
| id |  | > | > | > |
| + | < | > | < | > |
| \* | < | > | > | > |
| $ | < | < | < |  |

**Algorithms:-**

set i/p pointer to the first symbol of w$

repeat forever

if $ is on top of the stack and i/p points to $ then

return

else

let a be the topmost terminal symbol on the stack and let b be the symbol

pointed to by i/p pointer

if a<b or a=b then

push b onto the stack advance i/p pointer to the next i/p symbol.

else if a>b then

pop the stack until the top stack terminal is related by < to the terminal

most recently popped

else

error

**Ex.**

Parse the string id+id\*id by the operator precedence parsing techniques.

E→E+E

E→E\*E

E→id

|  |  |  |
| --- | --- | --- |
| **Stack** | **Input** | **Production Rule** |
| $ | id+id\*id$ | - |
| $id | +id\*id$ | Shift |
| $ | +id\*id$ | Reduce(E→id) |
| $+ | id\*id$ | Shift |
| $+id | \*id$ | Shift |
| $+ | \*id$ | Reduce(E→id) |
| $+\* | id$ | Shift |
| $+\*id | $ | Shift |
| $+\* | $ | Reduce(E→id) |
| $+ | $ | Reduce(E→E\*E) |
| $ | $ | Accept |

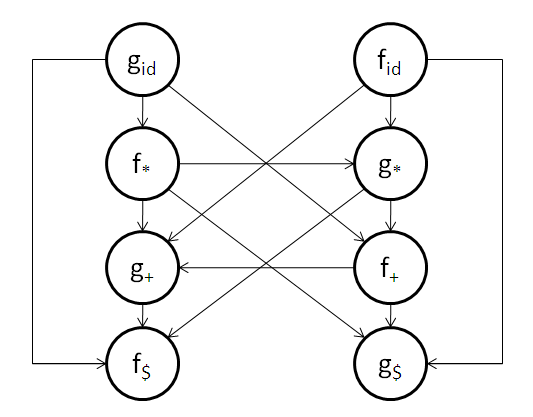
**Note :-** Disadvantages of operator precedence table is there are n2 entries in the precedence table, where n is no of operator.

**Time Complexity is O(n2)**

**Precedence function:-**

Compilers using operator-precedence parser need not store the table of precedence relation. The table can be encoded by two precedence function f and g that map terminal symbol to integers.

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



**Function Table:-**

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

**Note:-**

Function table represent the same information which operator precedence table representing with less storage.

**Disadvantages:-**

Error checking capability is less than operator precedence table.

**Leading and Trailing Method:-**

Algorithms:

begin

for each production A→B1B2…………Bn

for i=1 to n-1

if Bi and Bi+1 are both terminal then

Bi=Bi+1

if i<=n-2 and Bi and Bi+2 are both terminal and Bi+1 is non-terminal then

Bi=Bi+2

if Bi is terminal and Bi+1 is non-terminal then

for all a in LEADING(Bi+1)

set Bi<a

if Bi is non-terminal and Bi+1 is termnal then

for all a in TRAILING(Bi)

set a>Bi+1

end

**Ex.**

Compute the leading and trailing for non-terminal E,T,F in following grammar

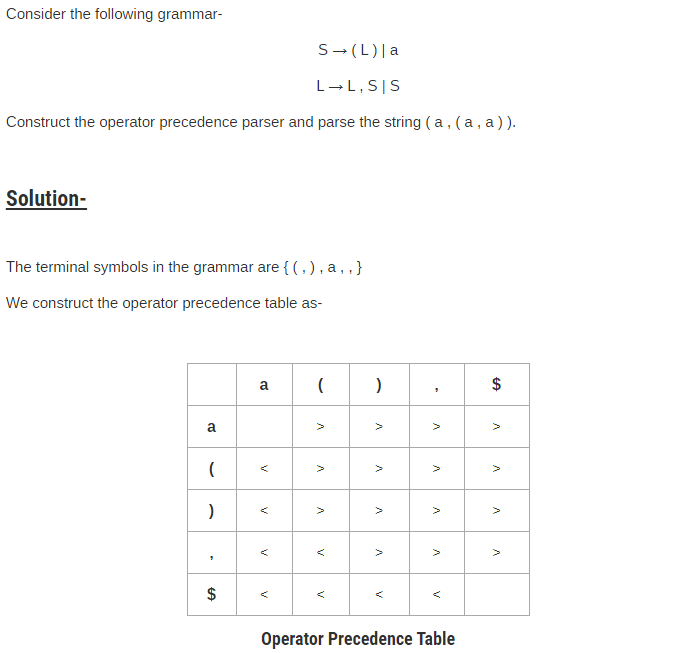
E→E+T│T

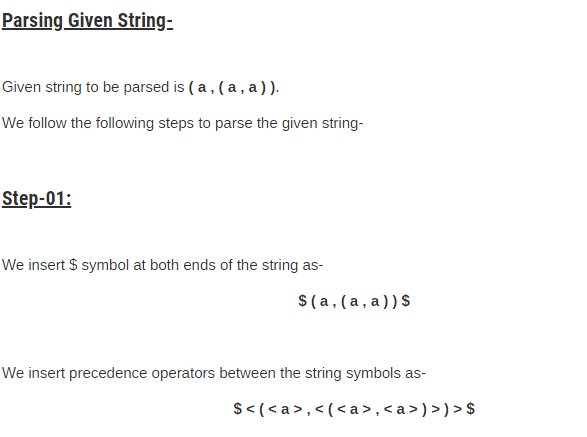
T→T\*F│F

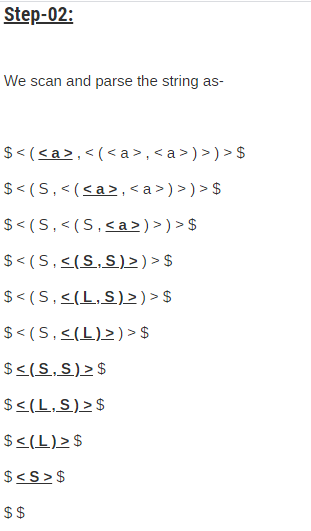
F→(E)│id

|  |  |  |
| --- | --- | --- |
|  | Leading | Trailing |
| E | {+ , \* , ( , id } | { + , \* , ) , id } |
| T | {\* , ( , id } | { \* , ) , id } |
| F | { ( , id } | { ) , id } |

Question:-

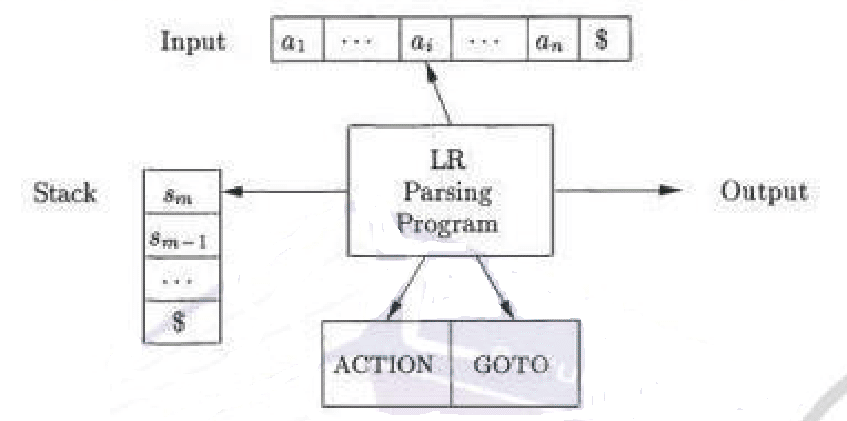
****

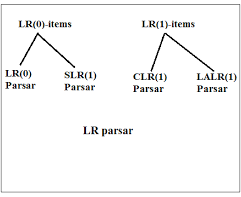
****

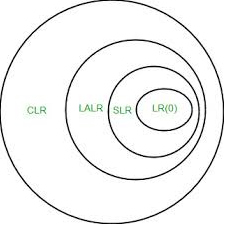


**LR Parsing:-**

A large class of grammar can be parsed using LR(k) parsers. Here L is for left to right scanning of the input string. R is for constructing the right most derivation in the reverse and k represents number of input symbols under look-ahead pointer used to making parsing decisions.







**LR(0)Item:-**

Item for a grammar G is a production of G with a dot(.) at some position of the right side. That is if we have a production A→ a BC in grammar G than items of G are

A→ .a BC

A→ a. BC

A→ a B.C

A→ a BC.

If G has null production (A→ϵ) than

A→**.** is an item of G

**Closure operation:-**

If I is a set of item 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→ұ is a production, then add the B→.ұ 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:-**

If I is a set of LR(0) items and X is a grammar symbol symbol(terminal or non-terminal) then goto (I,X) is defined as follows

If A→α.Xβ in I then every item in closure ({A→αX.β}) will be in goto(I,X).

**Parser Action:-**

1. if Action [sm , ai] =shift s , the parser executes a shift move, entering the configuration

(s0x1s1x2…………….xmsm ai s, ai+1………….an$)

1. if Action[sm,ai]=reduce A→β , then parser executes a reduce move ,entering the configuration

(s0x1s1x2…………….xm-1sm-1 As, aiai+1……an$)

Where r is the length of β and s=GOTO[sm-r , A].

Here the parser popped first 2r symbols(r state symbols and r grammar symbols)off the stack, exposing state sm-r . The parser then pushed s, entry for GOTO[sm-r, A], onto the stack. The current input symbol is not changed in a reduce move.

1. if Action[sm , ai]=accept parsing is completed.
2. if Action [sm,ai]=error , the parser has discovered and error and calls an error recovery routine.

**Algorithms:-**

Set i/p pointer to point the first symbol of w$

Repeat forever

begin

Let S be the state on top of the stack and a the symbol pointed to by i/p pointer

If action [s,a]=shift then

Push a then on top of the stack advance i/p pointer to the next input symbol.

else if action[s,a] =reduce A→β then

pop 2\*│β│ symbol off the stack. Let be the state now on top of the stack.

Push A then goto [,A] on top of the stack.

Output the production A→β

else if action[S,a]=accept then

return

else

error();

end

**Ex.** Parse the string “**aabb”**  by LR(0) parsing technique?

S→AA

A→aA│b

S

A

a

b

b

a

a

b

A

I0

I1

I2

I3

I4

I5

I6

**Canonical collection of LR(0) item**

**LR(0) Table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **State** | **Action** | | | **Goto** | |
| **a** | **b** | **$** | **A** | **S** |
| 0 | S3 | S4 |  | 2 | 1 |
| 1 |  |  | Accept |  |  |
| 2 | S3 | S4 |  | 5 |  |
| 3 | S3 | S4 |  | 6 |  |
| 4 | r3 | r3 | r3 |  |  |
| 5 | r1 | r1 | r1 |  |  |
| 6 | r2 | r2 | r2 |  |  |

**Working:-**

|  |  |  |  |
| --- | --- | --- | --- |
| **Stack** | **Input** | **Action** | **Output** |
| 0 | aabb$ | S3(shift) |  |
| 0a3 | abb$ | S3(shift) |  |
| 0a3a3 | bb$ | S4(shift) |  |
| 0a3a3b4 | b$ | r3(reduce) | A→b |
| 0a3a3A6 | b$ | r2(reduce) | A→aA |
| 0a3A6 | b$ | r2(reduce) | A→aA |
| 0A2 | b$ | S4(shift) |  |
| 0A2b4 | $ | r3(reduce) | A→b |
| 0A2A5 | $ | r1(reduce) | S→AA |
| 0S1 | $ | Accept |  |

a

a

b

b

**Ex.** Parse the string “**aabb”**  by SLR(1) parsing technique?

S→AA

A→aA│b

S

A

a

b

b

a

a

b

A

I0

I1

I2

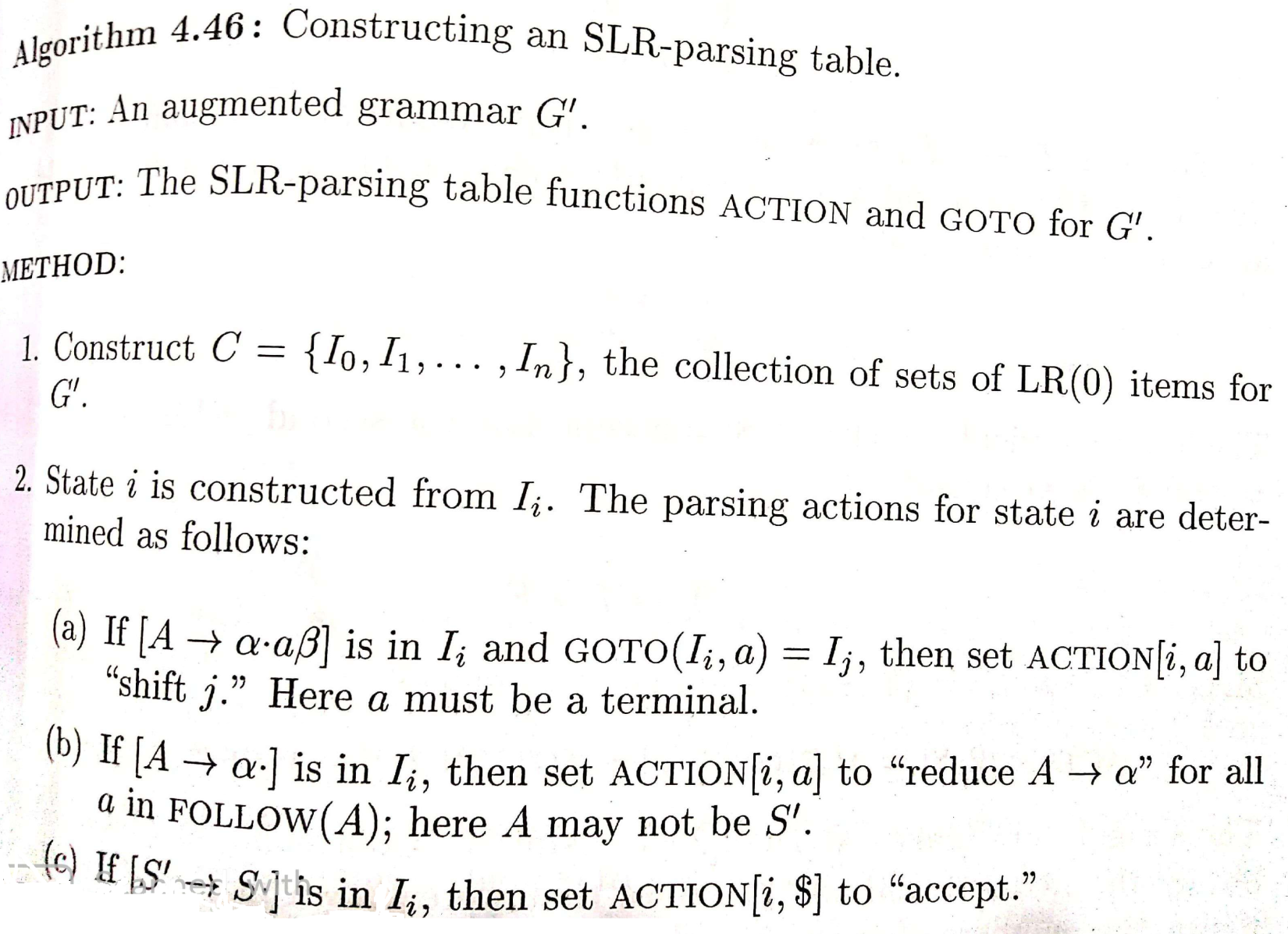
I3

I4

I5

I6

**Canonical collection of LR(0) item**



**SLR(1) Parsing Table:-**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Action** | | | **Goto** | |
|  | A | B | $ | A | S |
| 0 | S3 | S4 |  | 2 | 1 |
| 1 |  |  | Accept |  |  |
| 2 | S3 | S4 |  | 5 |  |
| 3 | S3 | S4 |  | 6 |  |
| 4 | r3 | r3 | r3 |  |  |
| 5 |  |  | r1 |  |  |
| 6 | r2 | r2 | r2 |  |  |

**Working:-**

|  |  |  |  |
| --- | --- | --- | --- |
| **Stack** | **Input** | **Action** | **Production Used** |
| 0 | aabb$ | S3(shift) | - |
| 0a3 | abb$ | S3(shift) | - |
| 0a3a3 | bb$ | S4(shift) | - |
| 0a3a3b4 | b$ | r3(reduce) | A→b |
| 0a3a3A6 | **b**$ | r2(reduce) | A→aA |
| 0a3A6 | b$ | r2(reduce) | A→aA |
| 0A2 | b$ | S4(shift) | - |
| 0A2b4 | $ | r3(reduce) | A→b |
| 0A2A5 | $ | r1(reduce) | S→AA |
| 0S1 | $ | Accept |  |

a

a

b

b

**Parse Tree**

**Ex. Draw the canonical collection of LR(0) item of the following grammar**

E→E+T

E→T

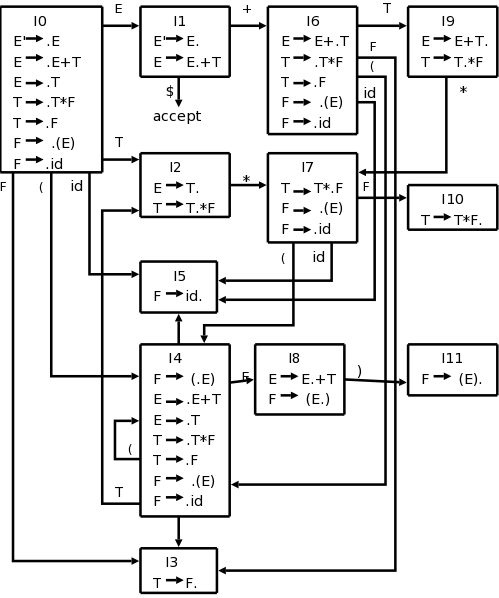
T→T\*F

T→F

F→ (E)

F→ id

**Solution:-**

****

**LR (1) item:-**

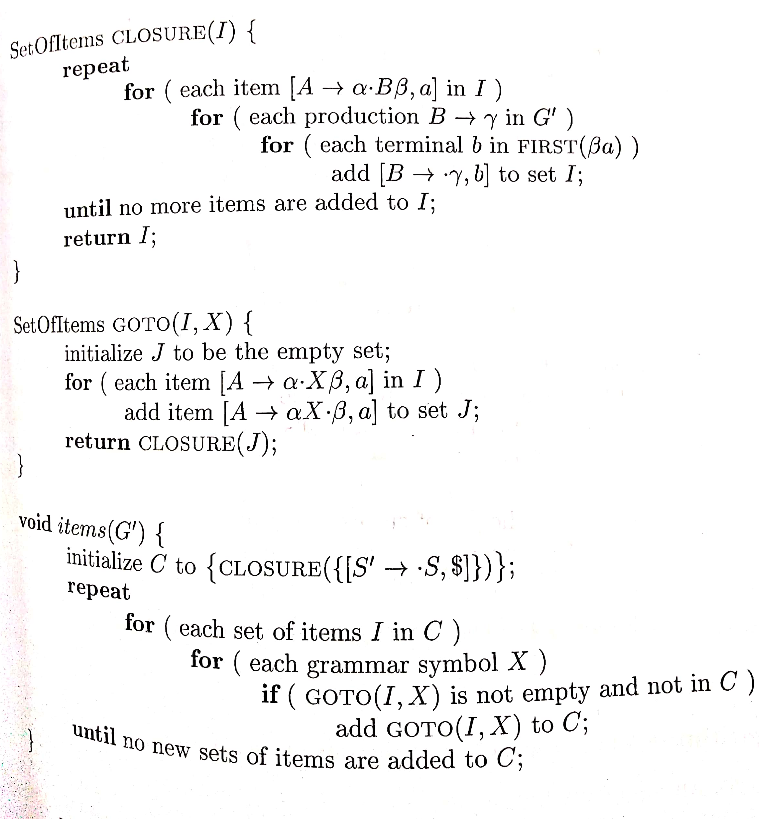
LR (1) item is a collection of LR (0) items and a look ahead symbol.

**LR (1) item = LR (0) item + look ahead**

The look ahead is used to determine that where we place the final item.

The look ahead always add $ symbol for the argument production.

**Algorithms of Constructing Canonical Collection of LR(1) item:-**

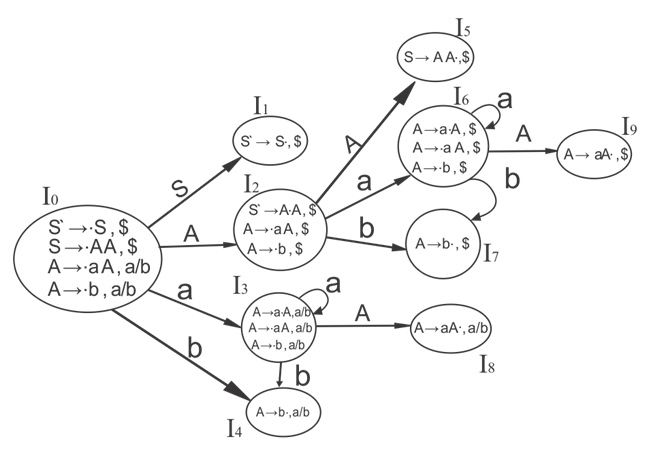


**Ex.**  Parse the string “aabb” by CLR(1) parsing techniques considering the following grammar.

S→AA

A→aA│b

**Canonical Collection of LR(1) item:-**



**CLR(1) Table:-**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **State** | **Action** | | | **Goto** | |
| **a** | **b** | **$** | **S** | **A** |
| **0** | S3 | S4 |  | 1 | 2 |
| **1** |  |  | Accept |  |  |
| **2** | S6 | S7 |  |  | 5 |
| **3** | S3 | S4 |  |  | 8 |
| **4** | r3 | r3 |  |  |  |
| **5** |  |  | r1 |  |  |
| **6** | S6 | S7 |  |  | 9 |
| **7** |  |  | r3 |  |  |
| **8** | r2 | r2 |  |  |  |
| **9** |  |  | r2 |  |  |

**Working:-**

|  |  |  |  |
| --- | --- | --- | --- |
| **Stack** | **Input** | **Action** | **Production Used** |
| 0 | aabb$ | S3(shift) | - |
| 0a3 | abb$ | S3(shift) | - |
| 0a3a3 | bb$ | S4(shift) | - |
| 0a3a3b4 | b$ | r3(reduce) | A→b |
| 0a3a3A8 | b$ | r2(reduce) | A→aA |
| 0a3A8 | b$ | r2(reduce) | A→aA |
| 0A2 | b$ | S7(shift) | - |
| 0A2b7 | $ | r3(reduce) | A→b |
| 0A2A5 | $ | r1(reduce) | S→AA |
| 0S1 | $ | Accept | - |

a

a

b

b

**Parse Tree**

**Ex.**  Parse the string “aabb” by LALR(1) parsing techniques considering the following grammar.

S→AA

A→aA│b

**LALR(1) table:-**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **State** | **Action** | | | **Goto** | |
| **A** | **b** | **$** | **S** | **A** |
| **0** | S36 | S47 |  | 1 | 2 |
| **1** |  |  | Accept |  |  |
| **2** | S36 | S47 |  |  | 5 |
| **36** | S36 | S47 |  |  | 89 |
| **47** | r3 | r3 | r3 |  |  |
| **5** |  |  | r1 |  |  |
| **89** | r2 | r2 | r2 |  |  |

**Working:-**

|  |  |  |  |
| --- | --- | --- | --- |
| **Stack** | **Input** | **Action** | **Production Used** |
| 0 | aabb$ | S36(shift) | - |
| 0a36 | abb$ | S36(shift) | - |
| 0a36a36 | bb$ | S47(shift) | - |
| 0a36a36b47 | b$ | r3(reduce) | A→b |
| 0a36a36A89 | b$ | r2(reduce) | A→aA |
| 0a36A89 | b$ | r2(reduce) | A→aA |
| 0A2 | b$ | S47(shift) | - |
| 0A2b47 | $ | r3(reduce) | A→b |
| 0A2A5 | $ | r1(reduce) | S→AA |
| 0S1 | $ | Accept | - |

**SR Conflict:-** In a parsing table, if a cell has both shift move as well as reduce move then shift reduce conflict arises

**RR Conflict:-**In a parsing table , if a cell has 2 different reduce moves then reduce-reduce conflict occurs.

**Ex.**

S→AaAb

S→BbBa

A→ϵ

B→ϵ