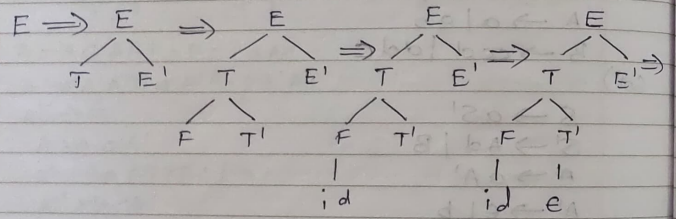
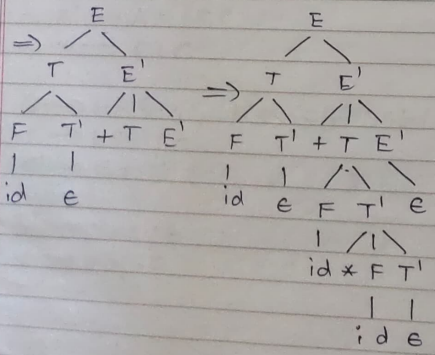
Top-Down Parsing

Apply By using top-down apply parse tree for the string id+id*id

Q $E \to TE'$
$E' \to +TE' \mid \epsilon$
$T \to FT'$
$T' \to *FT' \mid \epsilon$
$F \to (E) \mid id$

id+id*id





$E \underset{lm}{\Rightarrow} TE'$ ∵ $E \to TE'$
$E \underset{lm}{\Rightarrow} FT'E'$ ∵ $T \to FT'$
$E \underset{lm}{\Rightarrow} idT'E'$ $E \underset{lm}{\Rightarrow} idE'$ ∵ $F \to id$
$E \underset{lm}{\Rightarrow} id+TE'$ ∵ E
$E \underset{lm}{\Rightarrow} id+FT'E'$
$E \underset{lm}{\Rightarrow} id+idT'E'$
$E \underset{lm}{\Rightarrow} id+id*FT'E'$
$E \underset{lm}{\Rightarrow} id+id*idT'E'$
$E \underset{lm}{\Rightarrow} id+id*id\epsilon E'$
$E \underset{lm}{\Rightarrow} id+id*id$
$E \underset{lm}{\Rightarrow} id+id*id$

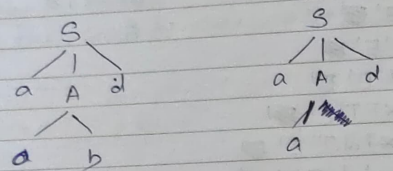
Recursive-Descent Parsing

A typical procedure for a non-terminal in a top-dow Parser

```
void A()
{
1)    choose an A-production, A → X1 X2 ..... Xk;
2)    for i=1 to k){
3)    if(Xi is a non-terminal)
4)        call procedure Xp();
5)    else if(Xi equals the current input symbol a)
6)        advance the input to the next symbol;
7)    else /* an error has occured */
    }
}
```

Q) $S \rightarrow cAd$
$A \rightarrow ab \mid a$

$w = cad$



★ FIRST AND FOLLOW functions

Define FIRST(α)
where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α.

Eg: $A \rightarrow ab \mid cd \mid ef$
FIRST(A) = {a, c, e}

if $\alpha \Rightarrow^{*} \epsilon$, then ϵ can also be in FIRST(α)

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

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

ϵ

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

③ If X is a non terminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production for some $k \geq 1$, then place 'a' in FIRST(X) if for some i, 'a' is in FIRST(Y_i) & ϵ is in all of FIRST(Y_1) ..... FIRST(Y_{i-1}) i.e $Y_1 \ldots Y_{i-1} \Rightarrow^{*} \epsilon$.

If ϵ is in FIRST(Y_j) for all $j = 1, 2 \ldots k$ then add ϵ to first of X

★ Compute FIRST for the following grammar

For a production (ϵ does not belong) $X \rightarrow Y_1 Y_2 Y_3$
→ if $\epsilon \notin$ FIRST(Y_1), then FIRST(X) = FIRST(Y_1)
→ if $\epsilon \in$ (belongs) FIRST(Y_1), then FIRST(X) = {FIRST(Y_1) − ϵ} ∪ FIRST($Y_2 Y_3$)

1) $S \rightarrow aBDh$
$B \rightarrow cC$
$C \rightarrow bC \mid \epsilon$
$D \rightarrow EF$
$E \rightarrow g \mid \epsilon$
$F \rightarrow f \mid \epsilon$

Sol") FIRST(S) = {a}
FIRST(B) = {c}
FIRST(C) = {b}
FIRST(D) = {FIRST(E) − ϵ} ∪ FIRST(F)
= {g, f, ϵ}
FIRST(E) = {g, ϵ}
FIRST(F) = {f, ϵ}

2) $S \to A$
$A \to aB \mid Ad$
$B \to b$
$C \to g$

Soln) FIRST(S) Eliminate left recursion

$S \to A$
$A \to aBA'$
$A' \to dA' \mid \epsilon$
$B \to b$
$C \to g$

FIRST(S) = {a} FIRST(B) = {b}
FIRST(A) = {a} FIRST(C) = {g}
FIRST(A') = {d}

3) $S \to (L) \mid a$
$L \to SL'$
$L' \to ,SL' \mid \epsilon$

Soln)

FIRST(S) = {(, a}
FIRST(L) = {(, a}
FIRST(L') = {, , ϵ}

4) $S \to AaAb \mid BbBa$
$A \to \epsilon$
$B \to \epsilon$

Soln) FIRST(S) = {FIRST(A) − ϵ} ∪ FIRST(a) ∪ {FIRST(B) − ϵ} ∪ FIRST(b)
= {a, b}
FIRST(A) = {ϵ}
FIRST(B) = {ϵ}

5) $E \to TE'$
$E' \to +TE' \mid \epsilon$
$T \to FT'$
$T' \to *FT' \mid \epsilon$
$F \to (E) \mid id$

Soln) FIRST(E') = {+, ϵ}
FIRST(T') = {*, ϵ}
FIRST(F) = {(, id}
FIRST(T) = FIRST(F) = {(, id}
FIRST(E) = FIRST(T) = {(, id}

6) $S \to ACB \mid CbB \mid Ba$
$A \to da \mid BC$
$B \to g \mid \epsilon$
$C \to h \mid \epsilon$

Soln) FIRST(B) = {g, ϵ}
FIRST(C) = {h, ϵ}
FIRST(A) = FIRST(d) ∪ {FIRST(B) − ϵ} ∪ FIRST(C)
= {d, g, h, ϵ}
FIRST(S) = {FIRST(A) − ϵ} ∪ {FIRST(C) − ϵ} ∪ FIRST(B) ∪ FIRST(b)
∪ {FIRST(B) − ϵ} ∪ FIRST(a)
= {d, g, h, ϵ, b, a}

Follow Function

Define FOLLOW(A) for non-terminal A to be a set of terminals 'a' that can appear immediately to to right of A in some sentential form. i.e

$S \overset{*}{\Rightarrow} \alpha A a \beta$

To compute FOLLOW(A) for all non-terminals A apply the foll^{n} rules until nothing can be added to any follow set

1) place $ in FOLLOW(S) where S is the start symbol and $ is input right end marker.
2) If there is a production $A \to \alpha B \beta$, then everything in FIRST(β) except ε is in FOLLOW(B).
3) If there is a production $A \to \alpha B$ or $A \to \alpha B \beta$ where FIRST(β) contains ε then everything in FOLLOW(A) is in FOLLOW(B)

- Compute FOLLOW for the foll^{n} grammar
Note: FOLLOW is only for non-terminals

Q) $S \to aBDh$
$B \to cC$
$C \to bC \mid \epsilon$
$D \to EF$
$E \to g \mid \epsilon$
$F \to f \mid \epsilon$

Soln)

FIRST(S) = {a}
FIRST(B) = {c}
FIRST(C) = {b, ε}
FIRST(D) = {g, f, ε}
FIRST(E) = {g, ε}
FIRST(F) = {f, ε}

FOLLOW(S) = {$}
FOLLOW(D) = {h}
FOLLOW(B) = {FIRST(D) − ε} ∪ FIRST(h) = {g, f, h}
FOLLOW(C) = FOLLOW(B) = {g, f, h}
FOLLOW(E) = FOLLOW(D) = {h}
FOLLOW(F) = {FIRST(F) − ε} ∪ FOLLOW(D)
= {f, h}

Q) $S \to A$
$A \to aBA'$
$A' \to dA' \mid \epsilon$
$B \to b$
$C \to g$

Soln) FIRST(S) = {a}
FIRST(A) = {a}
FIRST(B) = {b}
FIRST(A') = {d}
FIRST(C) = {g}

FOLLOW(S) = {$}
FOLLOW(A) = {$}
FOLLOW(A') = {$}
FOLLOW(B) = {d, $}
FOLLOW(C) = NA

Q) $S \rightarrow (L) \mid a$
$L \rightarrow SL'$
$L' \rightarrow ,SL' \mid \epsilon$

Sol^n) FIRST(S) = {(, a}
FIRST(L) = {(, a}
FIRST(L') = {,, ε}

FOLLOW(L) = {)}
FOLLOW(L') = FOLLOW(L) = {)}
FOLLOW(S) = {FIRST(L') − ε} ∪ FOLLOW(L)
∪ {FIRST(L') − ε} ∪ FOLLOW(L') ∪ $
= {,,), $}

Q) $S \rightarrow AaAb \mid BbBa$
$A \rightarrow \epsilon$
$B \rightarrow \epsilon$

Sol^n) FOLLOW(S) = {$}
FOLLOW(A) = {a, b}
FOLLOW(B) = {b, a}

Q) $E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \epsilon$
$F \rightarrow (E) \mid id$

Sol^n) FIRST(F) = {(, id}
FIRST(T') = {*, ε}
FIRST(T) = {(, id, *, ε}
FIRST(E') = {+, ε}
FIRST(E) = {(, id, *, +, ε}

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

Q) $S \rightarrow ACB \mid CbB \mid Ba$
$A \rightarrow da \mid BC$
$B \rightarrow g \mid \epsilon$
$C \rightarrow h \mid \epsilon$

Sol^n) FIRST(C) = {h, ε}
FIRST(B) = {g, ε}
FIRST(A) = {d, g, h, ε}
FIRST(S) = {b, d, g, h, a}

FOLLOW(A) = {g, h, $}
FOLLOW(B) = {$, a, g, h}
FOLLOW(C) = {$, b, g, h}
FOLLOW(S) = {$}

LL(1) Grammar:

↓ ↓

↓ Producing left most derivation

Scanning the input from left to right

A grammar G is LL1 if and only if whenever

$A \to \alpha \mid \beta$ are two disjoint productions of 'G' the following conditions hold

① For no terminal 'a' do both α & β derived strings beginning with 'a'

② Atmost one of α & β can derive the empty string (or) epsilon(e)

③ If β derives e then α does not derive any string beginning with terminal in follow of A (FOLLOW(A))
Likewise if α derives e then β does not derive any string beginning with terminal in FOLLOW(A)

Construction of Predictive Parsing Table

→INPUT: Grammar G
→OUTPUT: Parsing table 'M'
→METHOD: For each production $A \to \alpha$ of the grammar do the following

i) For each terminal 'a' in FIRST(α) add $A \to \alpha$ to M[A, a]

ii) If e is in FIRST(α) then for each terminal 'b' in FOLLOW(A) add $A \to \alpha$ to M[A, b].

iii) If e is in FIRST(α) & $ is in FOLLOW(A) add $A \to \alpha$ to M[A, $]

iv) If after performing the above there is no production at all in M[A, a] then set

10M

• Construct PPT for the following grammar

$E \to TE'$
$E' \to +TE' \mid e$
$T \to FT'$
$T' \to *FT' \mid e$
$F \to (E) \mid id$

Sol")

FIRST(F) = {(, id}
FIRST(T) = {(, id, *, e}
FIRST(T)

FIRST(T') = {*, e}
FIRST(E') = {+, e}

Table Structure:

Non Terminal	Input Symbol					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ϵ	E'→ϵ
T	T→FT'			T→FT'		
T'		T'→ϵ	T'→*FT'		T'→ϵ	T'→ϵ
F	F→id			F→(E)		

► Construct predictive parsing table for the following grammar

1) S → (L) | a
L → SL'
L' → ,SL' | ϵ

Sol")

Non terminal	Input Symbol				
	(	)	a	,	$
S	S→(L)		S→a		
L	L→SL'		L→SL'		
L'		L'→ϵ	L'→,SL'		

S → (L)
FIRST((L)) = {(}

S → a
FIRST(a) = a

FOLLOW(L') = {)}

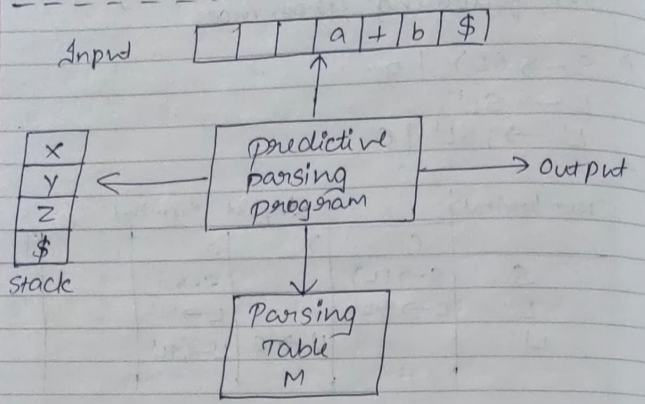
2) S → ACB | CbB | Ba
A → da | BC
B → g | ϵ
C → h | ϵ

Sol")

Non terminal	Input Symbol					
	a	b	d	g	h	$
S	S→Ba	S→CbB	S→ACB	S→ACB	S→ACB	
				S→Ba	S→CbB	
A			A→da	A→BC	A→BC	A→BC
B	B→ϵ			B→g	B→ϵ	B→ϵ
C	C→ϵ			C→ϵ	C→h C→ϵ	C→ϵ

↳ L(G) - Language of

* Non-Recursive Predictive Parsing



* Predictive Parsing Algorithm (Table driven Predictive Parsing)

Input: A string w & a parsing table M for a Grammar

Output: If w is in L(G), a leftmost derivation of w otherwise an error indication

Method: Intially the parser is in a configuration with w $ in i/p buffer & the start symbol S of G(S(G) on top of the stack above $.

```
Let 'a' be the first symbol of w;
Let 'X' be the top stack symbol;
while (X ≠ $)
{
    if (X = a)
        pop the stack & let 'a' be the next
        symbol of w;
    else if (X is a terminal)
        error();
    else if (M[X,a] is an error entry)
        error();
    else if (M[X,a] = X → Y1, Y2 ... Yk)
    {
        output the production X → Y1, Y2 ... Yk;
        pop the stack;
        push Yk, Yk-1, ... Y1 onto the stack
        with Y1 on top;
    }
    Let X be the top stack Symbol;
}
```

Matched	Stack	Input	Action
	E$	id+id*id$	
	TE'$	id+id*id$	Output E→TE'
	FT'E'$	id+id*id$	Output T→FT'
	idT'E'$	id+id*id$	Output F→id
id	T'E'$	+id*id$	Match id
id	E'$	+id*id$	Output T→ϵ
id	+TE'$	+id*id$	Output E'→+TE'
id+	TE'$	id*id$	Match +
id+	FT'E'$	id*id$	Output T→FT'
id+	idT'E'$	id*id$	Output F→id
id+id	T'E'$	*id$	Match id
id+id	*FT'E'$	*id$	Output T'→*FT'

id+id*	FT'E'$	id$	~~Out~~Match *
id+id*	idT'E'$	id$	Output F→id
id+id*id	T'E'$	$	Match id
id+id*id	E'$	$	Output T'→ϵ
id+id*id	$	$	Output E'→ϵ

* Error Recovery in predictive parsing.

1) Panic mode Error Recovery
→ If M[A,a] is blank, then the input symbol A is skipped
→ If the entry is skipped then the non-terminal on the top of the stack is pop.

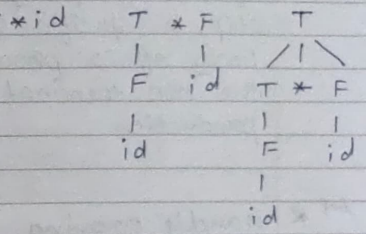
→ If a token on top of the stack does not match the input, then we pop token from stack

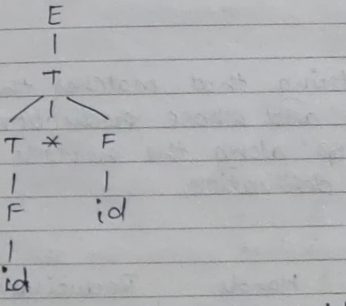
Bottom-up Parsing

UNIT - 3

* Bottom
- E → E+T | T
  T → T*F | F
  F → (E) | id
  construct id*id using bottom up approach

Sol^n) id*id F*id T*id T*F T





id*id, F*id, T*id, T*F, T, E

$E \underset{rm}{\Rightarrow} T \underset{rm}{\Rightarrow} T*F \underset{rm}{\Rightarrow} T*id \underset{rm}{\Rightarrow} F*id \underset{rm}{\Rightarrow} id*id$

→ Bottom up Approach or Bottom up parsing is the reverse of right most derivation

→ Bottom up Parse corresponds to the construction of parse tree for an i/p string beginning

at the leaves & working up towards the root. This process is called Reduction

* Reduction

Bottom up parsing is the process of reducing a string w to the start symbol of the ~~graph~~ grammar. At each reduction step, a specific substring matching the body of a production is replaced by the non terminal at the head of that production

* Handle proving

Handle is a substring that matches the body of a production and whose reduction represents one step along the reverse of a right most derivation

Right Sentential Form	Handle	Reducing production
$id_1 * id_2$	id_1	$F \to id$
$F * id_2$	F	$T \to F$
$T * id_2$	id_2	$F \to id$
$T * F$	$T * F$	$T \to T * F$
T	T	$E \to T$

→ The right most derivation in reverse is achieved by handle proving

* Shift-Reduce Parsing

It is a form of bottom up parsing in which a stack holds grammar symbols and input buffer holds the rest of the string to be parsed

Initial configuration

1)

Stack	Input
$	w$

Final configuration

2)

Stack	Input
$S	$

3)

Stack	Input	Action
$ ~~$ id, * id $~~	$id_1 * id_2$ $	shift
$ id_1	$* id_2$ $	reduce by $F \to id$
$ ~~$ E~~	$* id_2$ $	reduce by $T \to F$
$ T	$* id_2$ $	shift
$ $T *$	id_2 $	shift
$ $T * id_2$	$	reduce by $E \to id$
$ $T * ~~id2~~ F$	$	reduce by $T \to T * F$
$ T	$	reduce by $E \to T$
$ E	$	Accept

Shift, Reduce, Accept, Error

Introduction to LR Parsing (Simple LR)

LR(k) parsing
↓
left to right scanning of the input.
→ constructive right most derivation in reverse.
→ No. of input signals of look-ahead.

Why LR Parsers?

imp
10mks

Items & LR(0) automation

LR(0) automaton = LR(0) grammar + DFA

Item

States represent sets of items. An LR(0) item of a grammar G is a production of G with a '.' at some position of the body.

Eg: $A \to XYZ$ production yields the foll^n items
$A \to .XYZ$
$A \to X.YZ$
$A \to XY.Z$
$A \to XYZ.$

One collection of sets of LR(0) items called canonical LR(0) collection provides the basis for constructing a DFA. Such an automaton is called LR(0) automaton.

imp

To construct the cannonical LR(0) collection for a grammar we define an augmented grammar and two functions CLOSURE & GOTO

If G is a grammar with start symbol S then G' the augmented grammar for G is a G with a new start symbol S' and production $S' \to S$

CLOSURE of item sets

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:
→ Initially add energy item in I to CLOSURE(I)
→ If $A \to \alpha.B\beta$ is in CLOSURE(I) & $B \to \gamma$ is a production then add the item $B \to .\gamma$ to CLOSURE(I) if it is not already there. Apply this rule until no more new items can be added to closure(I)

→ Ex: Compute the closure of the foll^n items
$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$
⇒ augmented G' ⇒ $E' \to E$ $E \to E.$

• Compute closure of $E' \to E$

Sol^n)

I ⇒ I : $E' \to .E$
CLOSURE (I)
({$E' \to E$})

$E' \rightarrow .E$
$E \rightarrow .E+T$
$E \rightarrow .T$
$T \rightarrow .T+F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

g) CLOSURE of $E \rightarrow .T$

Sol" CLOSURE($E \rightarrow .T$)

$T \rightarrow .T+F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

* CLOSURE Computation:

```
Set of items CLOSURE(I){
    J = I;
    repeat
        for(each item A → α.Bβ in J)
            for(each production B → γ of G)
                if(B → .γ is not in J)
                    add B → .γ to J;
        until no more items are added
        on one round;
    repeat Jj
}
```

- Compute the closure of the set $\{E \rightarrow E.+T\}$

$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$

Sol") CLOSURE of $E \rightarrow E+.T$ is

$E \rightarrow E+.T$ → Kernel
$T \rightarrow .T*F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$
→ Non-kernel

Kernel items: The initial item is $S \rightarrow .S$ and all items whose dots are not at the left end

Non-kernel items:
All items with their dots at the left end except for $S \rightarrow .S$

* Function GOTO

GOTO(I, X) where I is a set of items & X is a grammar symbol is defined to be the closure of the set of all items $[A \rightarrow \alpha X.\beta]$ such that $[A \rightarrow \alpha.X\beta]$ is in I

GOTO function is used to define the transitions in LR(0) automata.

GOTO(I, x) specifies the transition from state for I under input x

- Compute GOTO(I, +) ~~I, +~~ for the set I : {E → E.+T}

Soln)

GOTO(I, +) is

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

~~Compute the GOTO~~

- Find closure of {E' → .E}

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

I_0

Soln) E' → .E
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

- Compute GOTO(I_0, E)

Soln) E' → E. → I_1
E → E.+T

$ → Accept

- Compute GOTO(I_0, T)

Soln) E → T. → I_2
T → T.*F

- Compute GOTO(I_0, F)

Soln) T → F. → I_3

- Compute GOTO(I_0, id)

Soln) F → id. → I_4 Is

- Compute GOTO(I_0, ( )

Soln) F → (.E)
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id
→ Is I_4

GOTO(I_1, +)

E → E+.T
T → .T*F
T → .F
F → .(E)
F → .id
→ I_6

GOTO(I_2, *)

T → T*.F
F → .(E)
F → .id
→ I_7

GOTO (I_5, E)

F → (E.)
E → E.+T
} → I_8

GOTO(I_5, T) → I_2

GOTO(I_5, F) → I_3

GOTO(I_5, () → I_5

GOTO(I_5, id) → I_4

GOTO(I_8,))

F → (E) → I_9

GOTO(I_8, +) → I_6

GOTO(I_6, T)

E → E+T.
T → T.*F
} → I_{10}

GOTO(I_6, F) → I_3

GOTO(I_6, () → I_5

GOTO(I_6, id) → I_4

GOTO(I_{10}, *) → I_7

GOTO(I_7, F)

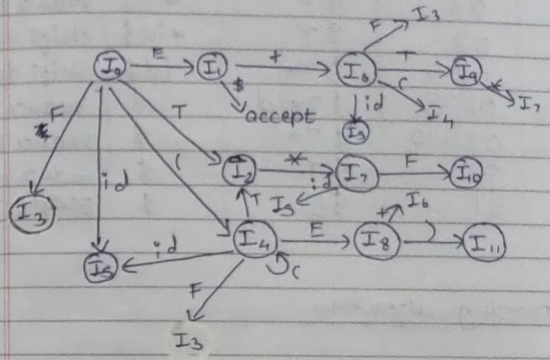
T → T*F. → I_{11}

G' → Augmented Grammar

GOTO(I_7, () → I_5

GOTO(I_7, id) → I_4

I_0 is the start state



Imp

Algorithm

Computation of cannonical collection of LR(0) items

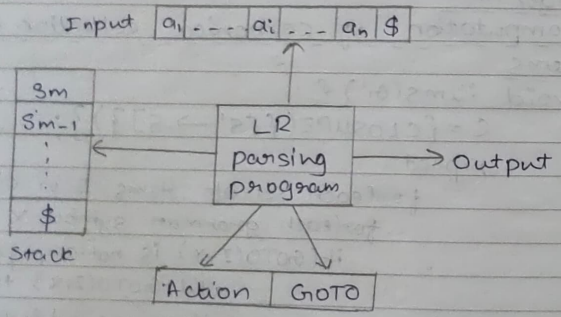
```
void items(G') {
    C = {CLOSURE({[S' → .S]})};
    repeat
        for(each set of items I in C)
            for(each grammar symbol X)
                if(GOTO(I,X) is not empty and not in C)
                    add GOTO(I,X) to C;
    until no new sets of items are added to C on a round;
}
```

* Shift-Reduce parser on input id*id

LINE	Stack	Symbols	Input	Action
1	0	$	id*id$	shift to 5
2	05	$id	*id$	reduce by F→id
3	03	$F	*id$	reduce by T→F
4	02	$T	*id$	shift to 7
5	027	$T*	id$	shift to 5
6	0275	$T*id	$	reduce by F→id
7	02710	$T*F	$	reduce by T→T*
8	02	$T	$	reduce by E→T
9	01	$E	$	accept

* L-R parsing algorithm

Model of an L-R parser



The Parsing table consists of 2 parts. ACTION & GOTO

The ACTION Function takes as arguments a state 'i' & a terminal 'a' or '$'. The value of ~~action~~ ACTION[i,a] can have one of 4 forms

i) Shift j, where j is a state
ii) Reduce A → β
iii) Accept
iv) Error

We extend the ~~The~~ GOTO Function to states. If GOTO[Ii,A] = Ij, then GOTO also maps a state 'i' and a non terminal 'A' to state 'j'

~~Imp~~

Algorithm

Input: A string w & L-R parsing table with functions action & goto for a grammar G

Output: If w is L(G) i.e (language of that grammar), the reduction steps of a ~~boo~~ to bottom up parse of w, otherwise an error indication

Method: Initially, the parser has S_0, on its stack, where S_0 is the initial state & w$ in the input buffer. The parser then executes, the foll" program.

```
Let 'a' be the first symbol of w$;
while(1){
    Let 's' be the state on top of the stack;
    if ~~action of~~(ACTION[s,a]) = shift t)
    {
        PUSH 't' onto the stack;
```

Let 'a' be the next input symbol;
}
else if (ACTION[S,a] = Reduce A→β)
{
POP |β| symbols of the stack;
Let state 't' now be on top of stack;
PUSH GOTO[t,A] onto the stack;
OUTPUT the production A→β;
}
else if(ACTION[S,a] = accept)
{
break;
}
else
call error recovery routine;
}

* Constructing SLR parsing table

State	ACTION id	+	*	(	)	$	GOTO E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r4			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

4) ~~E→E+~~

The codes for the actions are
1) Si → shift for stack state i
2) rj → reduce by the production number j
3) acc → accept
4) blank means error

* Constructing SLR parsing table

Input: An augmented grammar G'

output: The SLR parsing table function ACTION & 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 'Ii'. The parsing actions for state 'i' are determined as follows
i) If [A→α.aβ] is in 'Ii' and GOTO[Ii,a]=Ij then set ACTION[i,a] to 'shift j'
ii) If [A→α.] is in 'Ii', then set ACTION[i,a] to reduce A→α" for all 'a' in FOLLOW(A)
Here 'A' may not be 'S'
iii) If [S'→S.] is in Ii, then set ~~ACTION~~ ACTION[i,$].

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 & 3 are made error.

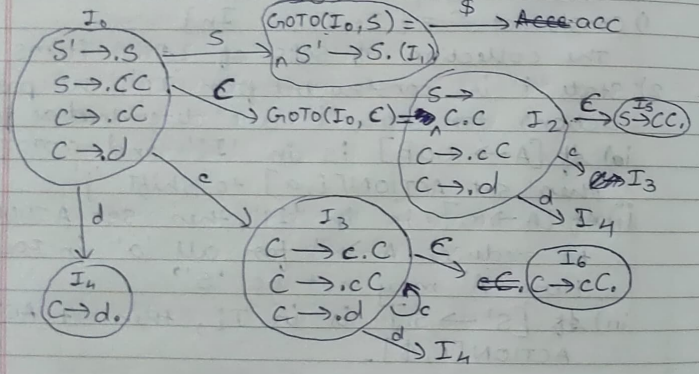
5) The initial state of the parser is the one constructed from the set of items containing $S' \to .S$

• 1) Construct LR(0) Automata [SLR Parser], SLR parsing table for the foll^n grammar

1) $S \to CC$
2) $C \to cC$
3) $C \to d$

Sol^n) $S' \to S$ (First Augment)

CLOSURE({ $S' \to .S$ })



STATE	ACTION			GOTO	
	c	d	$	S	C
0	S_3	S_4		1	2
1			acc		
2	S_3	S_4			5
3	S_3	S_4			6
4	r_3	r_3	r_3		
5			r_1		
6	r_2	r_2	r_2		

• 2) Parse the string ccdd for the above grammar

LINE	STACK	SYMBOLS	INPUT	ACTION
1	0	$	ccdd$	shift to 3
2	03	$c	cdd$	shift to 3
3	033	$cc	dd$	shift to 4
4	0334	$ccd	d$	Reduce by C→d
5	0336	$ccC	d$	Reduce by C→cC
6	036	$cC	d$	Reduce by C→cC
7	02	$C	d$	shift to 4
8	024	$Cd	$	Reduce by C→d
9	025	$CC	$	Reduce by S→CC
10	01	$S	$	accept

Viable prefix

The prefixes of right sentential forms that can appear on the stack of a shift reduce parser are called viable prefix

$E \underset{rm}{\overset{*}{\Rightarrow}} F*id \underset{rm}{\Rightarrow} (E)*id \underset{rm}{\Rightarrow} \ldots$

More powerful LR parsers

① "Cannonical LR" or LR

② "Lookahead LR" or LALR

① Cannonical LR(1) items

The general form of LR(1) item is $[A \to \alpha.B, a]$ where $A \to \alpha B$ is the production and 'a' is a terminal or '$'

'1' refers to length of the second component called lookahead of the item.

- Constructing LR(1) sets of items

▸ CLOSURE (Algorithm)

```
set of Items CLOSURE(I){
  repeat
    for(each item [A →α.Bβ, a] in I)
      for(each production B→γ in G')
        for(each terminal b in FIRST(βa))
          add[B→.γ, b] to set I;
  repeat until no more items are added to I;
  return I;
}
```

- Find the closure of the items

1) $S \to CC$
2) $C \to cC$
3) $C \to d$

Sol") $S' \to S$

CLOSURE({ $S' \to .S, \$$ })

$A \to \alpha.B\beta, a$

$A = S' \quad B = S$
$\alpha = \epsilon \quad \beta = \epsilon$
$a = \$$
FIRST(βa) = FIRST($\epsilon \$$) = $\$$
FIRST(βa) = FIRST($\epsilon \$$) = $\{c, d\}$

I_0

$S \to .S, \$$
$S \to .CC, \$$
$C \to .cC, c$
$C \to .d, c$
$C \to cC, d$
$C \to .d, d$

- Find the CLOSURE of $S' \to S., \$$

$\$ = S' \to S., \$ \to I_1$

- Find the CLOSURE of $S \to C.C, \$$

I_2

$S \to x.C, \$$
$C \to .cC, \$$
$C \to .d, \$$

GOTO (Algorithm)

```
Set ofItems GOTO(I, X){
    Initialize J to the empty set;
    for(each item [A → α.Xβ, a] in I)
        add item[A → αX.β, a] to set J;
    return CLOSURE(J);
}
```

- Find GOTO(I_0, S) from previous Sum

  S → .C, $
  C → .cC, $
  C → .d, $

GOTO (I_0, C)

  C → c.C, c/d — I_3
  C → .cC, c/d
  C → .d, c/d

GOTO(I_2, C)

  S → CC., $ — I_5

GOTO(I_2, c)

  C → c.C, $ — I_6
  C → .cC, c/d
  C → .d, c/d