Compiler Designing

6/9/22

Predictive Parsing

It is a type of top-down parsing as it builds the parse tree from top to bottom, i.e. from root to leaves.

Given a grammar, we can use predictive parsing to generate the input string.

STEPS FOR PREDICTIVE PARSING →

i) Elimination of Left Recursion

ii) Left Factoring (if two or more productions have common symbols from the left end).

iii) First and Follow functions

iv) Predictive Parsing table

v) Parsing the input string.

Example:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Step 1:

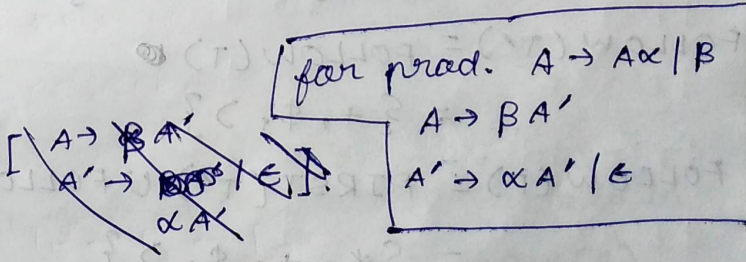
$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$



Step 2:

As no production in the subsequent grammar contains same symbols on the left end of the RHS, so there is no left factoring.

Step 3:

FIRST(E) = FIRST(T) = FIRST(F) = FIRST(C) / FIRST(id) = { (, id }.

FIRST(E') = FIRST(+) ∪ FIRST(∈)
= { +, ∈ }.

FIRST(T') = FIRST(*) ∪ FIRST(∈)
= { *, ∈ }.

FOLLOW(E) = FIRST(>) = { $, > }. (as E is start symbol, include $.)

FOLLOW(E') = FOLLOW(E) ∪ FOLLOW(E') = { $, > }.

FOLLOW(T) = FIRST(E') ∪ FOLLOW(E) ∪ FOLLOW(E')
= { +, $, > } (substituting ∈ in 2nd prod.)

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

FOLLOW(F) = FIRST(T') ∪ FOLLOW(T) ∪ FOLLOW(T')
= { *, +, $, > }.

Step-4

Rows are the non-terminals
columns are the terminals.

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

RULES:

if $A \to \alpha$, consider row for non-terminal A and columns for FIRST(α).

If α is ϵ, the row $\to A$, column = FOLLOW(A).

$A \to \alpha$

1) $M[A, a] = A \to \alpha$, a is in FIRST(α).

2) $M[A, b] = A \to \alpha$, if ϵ is in FIRST(α), b is in FOLLOW(A).

---

i) $E \to TE'$ ⇒ find FIRST(T) = {(, id}.
so in row E and columns (and id, insert $E \to TE'$.

ii) $E' \to +TE' \mid \epsilon$ ⇒ FIRST(+) ∪ FOLLOW(E').
+, $,) ⇒ columns.

Step-5:

RULES → • If NT on TOS, refer to PP table. ⇒ write production in output
• If terminal on TOS matches 1st string of input, replace and move to next.

$w = id * id + id.$

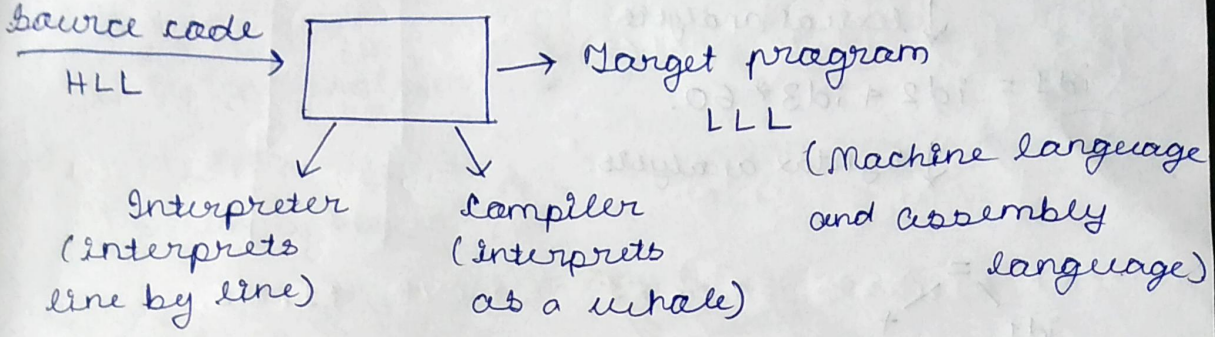
STACK	INPUT	OUTPUT
$E	id * id + id $	
$E'T	id * id + id $	$E \to TE'$
$E'T'F	id * id + id $	$T \to FT'$
$E'T'id	id * id + id $	$F \to id$
$E'T'	* id + id $	
$E'T'F*	* id + id $	$T' \to *FT'$
$E'T'F	id + id $	
$E'T'id	id + id $	$F \to id$
$E'T'	+ id $	
$E'	+ id $	$T' \to \epsilon$
$E'T+	+ id $	$E' \to +TE'$
$E'T	id $	
$E'T'F	id $	$T \to FT'$
$E'T'id	id $	$F \to id$
$E'T'	$	
$E'	$	$T' \to \epsilon$
$	$	$E' \to \epsilon$

(Pop existing TOS and push RHS of output production in reverse order.)

As both stack and input buffer are empty, so the input string has been successfully parsed by the predictive parsing technique.

COMPILER DESIGNING

16/9/22



Debugging is easy in interpreter.

Interpreter is very time consuming as it interprets the source code line by line.

Phases of a compiler →

Source program

↓

Lexical analysis (converts source program into a no. of tokens).

↓ (tokens)

Syntax analysis (checks syntax of tokens and creates a parse tree).

↓ (parse tree)

Semantic analysis (checks whether rules of lang. satisfied)

↓ (annotated parse tree)

Intermediate code generator (creates abstract code).

↓ (three address code)

Code optimization (removes unnecessary information).

↓

Code generator (generates target code)

↓

Target code.

Symbol table : A data structure that contains all info. about all variables used in each step.

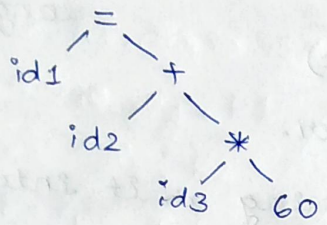
eg:

Position = initial + rate * 60.

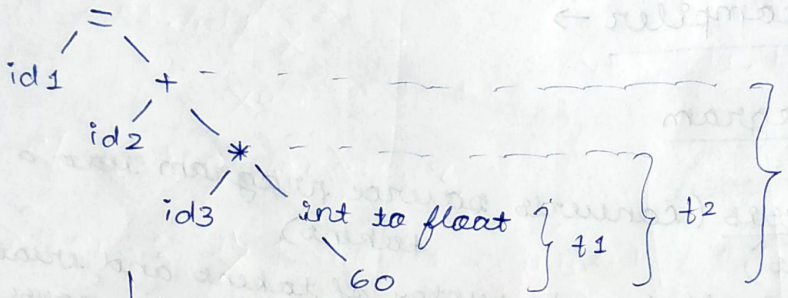
↓ lexical analysis

id1 = id2 + id3 * 60.

↓ syntax analysis.



↓ semantic analysis.



↓ intermediate code generation.

temp 1 = int to float (60)

temp 2 = id3 * temp 1

temp 3 = id2 + temp 2

id1 = temp 3

↓ code optimization.

temp 1 = id 3 * 60.0

id1 = id2 + temp 1

↓ code generator

```
MOVF  60.0, R1
MULF  id3, R1.
MOVF  id2, R2.
ADDF  R2, R1.
MOVF  R1, id1.
```

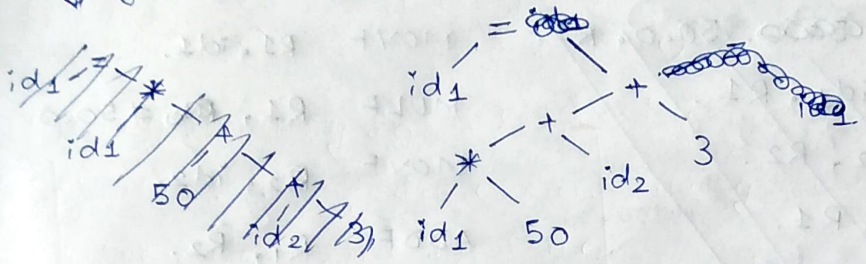
eg.

```
int i, j;
i = i * 50 + j + 3;
```

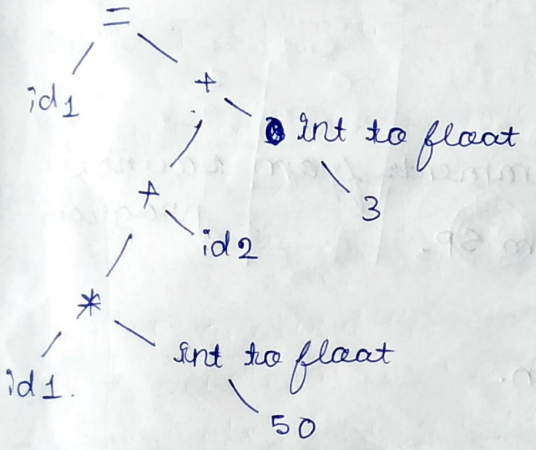
↓ lexical analyzer

$id_1 = id_1 * 50 + id_2 + 3;$ (* operation performed first).

↓ syntax analyzer



↓ semantic analyzer.



↓ intermediate code generator.

$t_1 = int\ to\ float(50)$

$t_2 = id_1 * t_1$

$t_3 = id_2 + t_2$

$t_4 = int\ to\ float(3)$

$t_5 = t_4 + t_3$

$id_1 = t_5$.

↓ code optimization.

$t_1 = id_1 * \text{int to float}(50)$

$t_2 = t_1 + id_2$

$t_3 = t_2 + \text{int to float}(3)$

$id_1 = t_3$

↓ code generator.

```
MOVF  50.0, R1.
MULF  id1, R1.
MOVF  id2, R2.
ADDF  R2, R1.
ADDF  3.0, R1.
MOVF  R1, id1.
```

id1 * 50 + id2

```
MOVF  R1, id1.
MULF  R1, #50.0
MOVF  R2, id2.
ADDF  R1, R2.
ADDF  R1, #3.0
MOVF  id1, R1.
```

—X—

Role of Lexical Analyzer

Functions: 1) Removes comments from source program.

2) Removes white spaces from SP.

3) Counts number of lines

4) Symbol table generation.

5) Handles errors.

6) If the language supports some macro preprocessor functions, then they can be implemented as lexical analysis takes place.

Lexical analyzer → recognizes regular exp.

Syntax analyzer → recognized by PDA, works with CFG.

Token → Any valid identifier, keyword, operator etc.

Lexeme: A sequence of characters in the source program that are matched by the predefined language rules to convert them to corresponding tokens.

Patterns: Used to define regular expressions.

$(l|\_)(l|d|\_)^*$

↳ general form of an identifier.

—X—

Q. Construct a DFA to accept the language over $\Sigma = \{0,1\}$, $L = \{w \mid w$ has even number of 0s and 1s$\}$.

$w = \{0110, 1100110, 1001, \ldots\}$

We define four states q_0, q_1, q_2, q_3 where

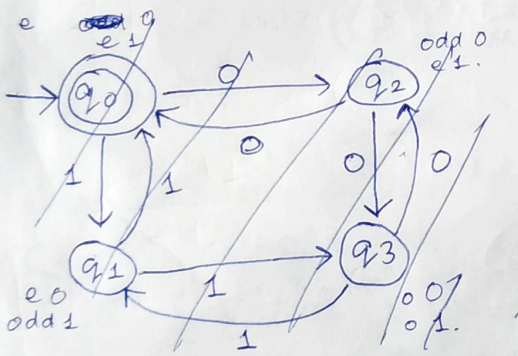
q_0 → even 0s and even 1s

q_1 → even 0s and odd 1s

q_2 → odd 0s and ~~odd~~ even 1s

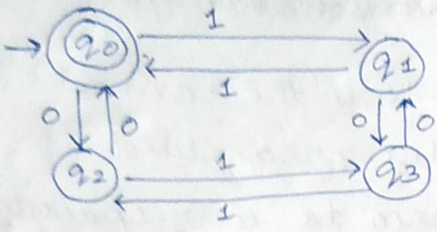
q_3 → odd 0s and odd 1s.

1100110.



Transition Table

State	Inputs	
	0	1
→ q_0	q_2	q_1
q_1		
q_2		
q_3		



Transition Table

State	Input 0	1
→ (q0)	q2	q1
q1	q3	q0
q2	q0	q3
q3	q1	q2

DFA, $D = (Q, \Sigma, \delta, q_0, F)$

where

$Q = \{q_0, q_1, q_2, q_3\}$.

$\Sigma = \{0, 1\}$.

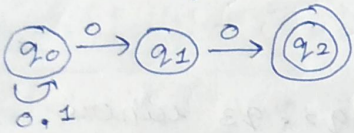
$q_0 = \{q_0\}$.

$F = \{q_0\}$.

—— X ——

Conversion from NFA to DFA using subset constr.

1)



States	Inputs 0	1
→ q0	{q0, q1}	{q0}
q1	{q2}	ϕ
(q2)	ϕ	ϕ

If NFA → n states

DFA → 2^n states.

$\{q_0\} = A$.

$\delta(A, 0) = \{q_0, q_1\} = B$

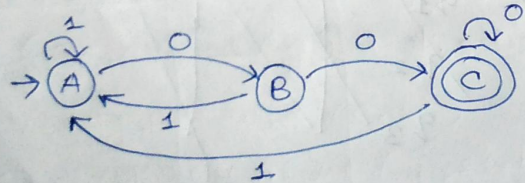
$\delta(A, 1) = \{q_0\} = A$.

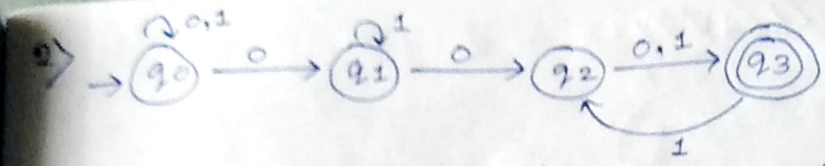
$\delta(B, 0) = \{q_0, q_1, q_2\} = C$

$\delta(B, 1) = \{q_0\} = A$.

$\delta(C, 0) = \{q_0, q_1, q_2\} = C$.

$\delta(C, 1) = \{q_0\} = A$.





do q_0 to start state of NFA, it will also be the start state of DFA.

$\{q_0\} = A$.

$\delta(A, 0) = \{q_0, q_1\} = B$.

$\delta(A, 1) = \{q_0\} = A$.

$\delta(B, 0) = \{q_0, q_1, q_2\} = C$

$\delta(B, 1) = \{q_0, q_1\} = B$.

$\delta(C, 0) = \{q_0, q_1, q_2, q_3\} = D$.

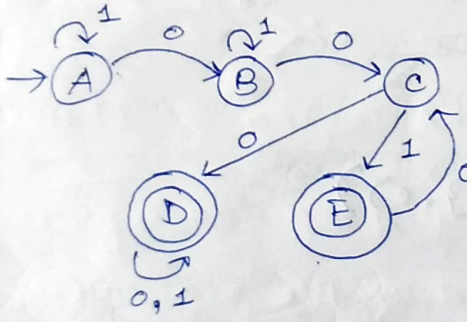
$\delta(C, 1) = \{q_0, q_1, q_3\} = E$

$\delta(D, 0) = \{q_0, q_1, q_2, q_3\} = D$

$\delta(D, 1) = \{q_0, q_1, q_3, q_2\} = D$.

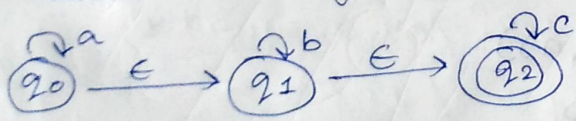
$\delta(E, 0) = \{q_0, q_1, q_2\} = C$

$\delta(E, 1) = \{q_0, q_1, q_2\} = C$.



—— X ——

Q. Convert the given ϵ-NFA to DFA.



~~ϵ-closure $(q_0) = \{q_0, q_1, q_2\} = A$.~~ ~~$q_0 = A$.~~

ϵ-closure (move(A, a))

$= \epsilon$-closure $(\{q_0\}) = \{q_0, q_1, q_2\} = B$

ϵ-closure (move (A, b))

= ϵ-closure (

ϵ-closure $(q_0) = \{q_0, q_1, q_2\} = A$.

ϵ-closure (move (A, a)) =

= ϵ-closure (q_0)

= $\{q_0, q_1, q_2\} = A$.

ϵ-closure (move (A, b))

= ϵ-closure (q_1).

= $\{q_1, q_2\} = B$.

ϵ-closure (move (A, c))

= ϵ-closure $(\{q_2\})$.

= $\{q_2\} = C$.

ϵ-closure (move (B, a))

= ϵ-closure (move $(\{q_1, q_2\}, a)$)

= ϵ-closure $(\{\phi\})$. = ϕ. = D

ϵ-closure (move (B, b))

= ϵ-closure (move $(\{q_1, q_2\}, b)$)

= ϵ-closure $(\{q_1\})$.

= $\{q_1, q_2\} = B$.

ϵ-closure (move (B, c))

= ϵ-closure (move $(\{q_1, q_2\}, c)$)

= ϵ-closure $(\{q_2\})$.

= C.

ϵ-closure (move (C, a))

= ϵ-closure (move $(\{q_2\}, a)$).

= ϵ-closure (ϕ). = ϕ. = D

ϵ-closure (move (C, b))

= ϵ-closure (move $(\{q_2\}, b)$)

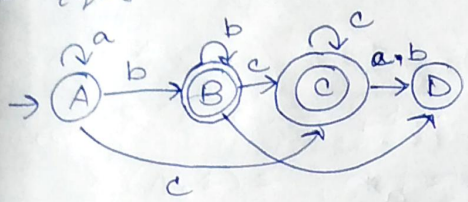
= ϵ-closure $(\phi) = \phi = D$.

ϵ- closure (move (C, c))

= ϵ- closure (move ({q_2}, c))

= ϵ- closure ({q_2})

= {q_2} = C.

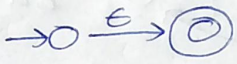


— X —

Thompson's Method (RE to ϵ-NFA)

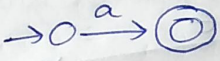
Rules:

1) ϵ :

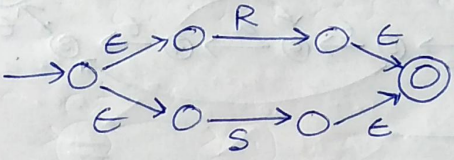


2) ϕ :

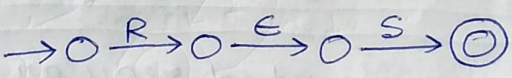
3) a :



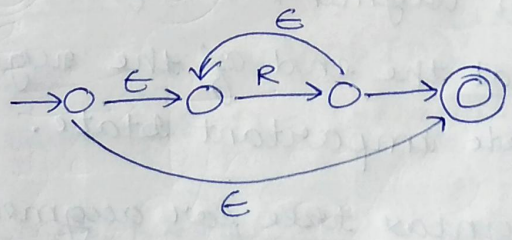
4) R+S :



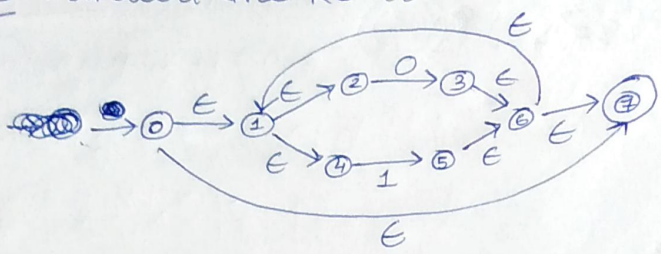
5) RS :



6) R* :

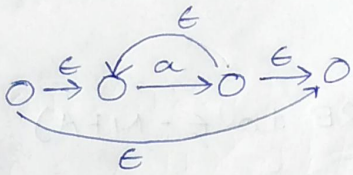


Q. Convert the RE $(0+1)^*$ to an ϵ-NFA.



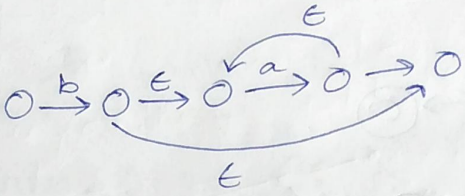
Q. Convert $b+ba^*$ to ϵ-NFA.

Step 1:



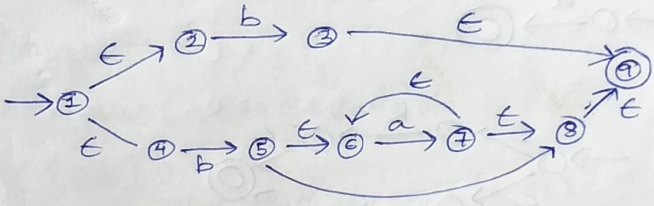
a^*

Step 2:



ba^*

Step 3:



— X —

Regular expression to DFA using direct method.

Step-1: Create an augmented regular expression. Apply end marker at the end of the regular expression to denote important state.

Step-2: Create syntax tree for augmented grammar, using three types of nodes –

i) Cat-node (·)

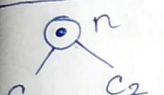
ii) Or-node (|)

iii) Star node (*).

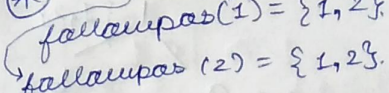
Step 3: Label every leaf node with unique number, known as its position.

Step 4: Find the nullable, firstpos, lastpos and followpos functions.

Rules for firstpos(n) and lastpos(n) →

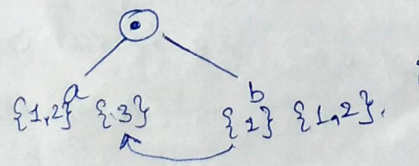
Node n	nullable(n)	firstpos(n)	lastpos(n)
leaf labelled ∈	true	ϕ	ϕ
leaf with position i	false	{i}	{i}
	nullable(c_1) or nullable(c_2)	firstpos(c_1) ∪ firstpos(c_2)	lastpos(c_1) ∪ lastpos(c_2)
	nullable(c_1) and nullable(c_2)	if(nullable(c_1)) firstpos(c_1) ∪ firstpos(c_2) else firstpos(c_1)	if(nullable(c_2)) lastpos(c_1) ∪ lastpos(c_2) else lastpos(c_2).
	true	firstpos(c_1)	lastpos(c_1).

Rules for followpos →



i) If (n is a star-node)
   for (each i in lastpos(n))
      followpos(i) = followpos(i) ∪ firstpos(n);

ii) If (n is a cat-node)
   for (each i in lastpos(c1))
      followpos(i) = followpos(i) ∪ firstpos(c2);



followpos(3) = {1}.

Step 5: Calculate the DFA by choosing followpos nodes pertaining to each input and finding the followpos, same as with subset construction.

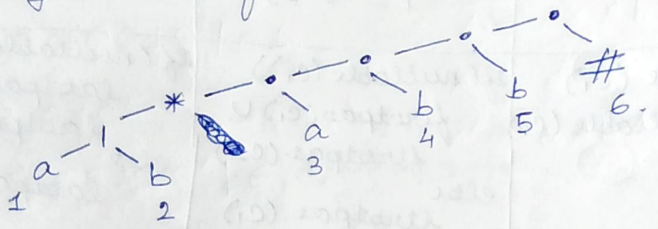
Q: Construct the corresponding DFA for the regular expression r = (a|b)*abb.

Step 1: r = (a|b)*abb.

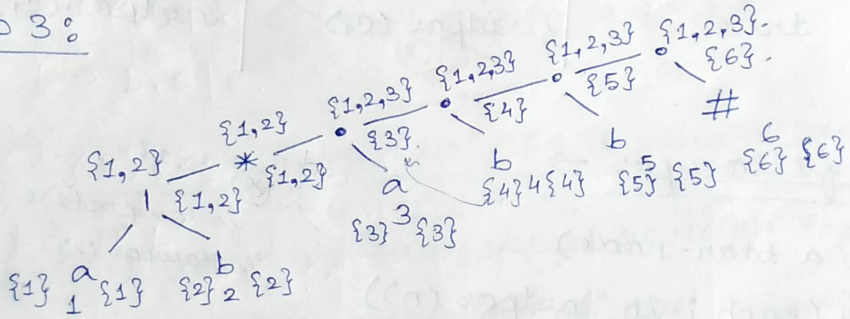
Augmented r = (a|b)*abb#

Step 2:

The syntax tree for the given RE is:



Step 3:



Step 4:

	Node	Followpos
a	1	{1, 2, 3}
b	2	{1, 2, 3}
a	3	{4}
b	4	{5}
b	5	{6}
#	6	-

Step-5: firstpos (root) = {1, 2, 3} = A (start state).

(A, a) = followpos (1) ∪ followpos (3)
= {1, 2, 3, 4} = B.

(A, b) = followpos (2) = {1, 2, 3} = A

(B, a) = followpos (1) ∪ followpos (3)
= B.

(B, b) = followpos (2) ∪ followpos (4).
= {1, 2, 3, 5} = C.

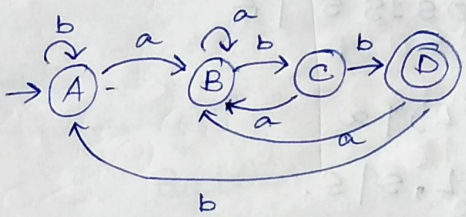
(C, a) = followpos (1) ∪ followpos (3) = B.

(C, b) = followpos (2) ∪ followpos (5)
= {1, 2, 3, 6} = D.

(D, a) = B.

(D, b) = followpos (2) = A.

State	Input	
	a	b
→ A	B	A
B	B	C
C	B	D
(D)	B	A



— × —

Left Recursion of CFG

For a production of the form $A \to A\alpha \mid \beta$, the grammar is said to have left recursion.
The variable in the left side occurs at the first position on the right side of the production, due to which left recursion occurs.

(*) If we have left recursion in our grammar, it leads to infinite recursion, (in top-down parsers) due to which we cannot generate the given string.

Rule: For production $A \to A\alpha \mid \beta$, we replace the productions as $A \to \beta A'$, $A' \to \alpha A' \mid \epsilon$

eg. $A \to A\alpha \mid \beta_1 \mid \beta_2 \ldots \beta_n$

$$\begin{cases} A \to \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A' \\ A' \to \alpha A' \mid \epsilon. \end{cases}$$

eg. $A \to A\alpha_1 \mid A\alpha_2 \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$.

$$\begin{cases} A \to \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A' \\ A' \to \alpha_1 A' \mid \alpha_2 A' \ldots \mid \alpha_n A' \mid \epsilon. \end{cases}$$

eg. $S \to SOS1S \mid 01.$

$S \to 01S'$

$S' \to OS1SS' \mid \epsilon$

eg. $A \to A\alpha A b \mid \epsilon$

$A \to \epsilon . A'$

$A' \to \alpha A b A' \mid \epsilon.$

eg. $S \to (L) \mid x$

$L \to L, S \mid S.$

$S \to (L) \mid x$

$L \to SL'$

$L' \to , SL' \mid \epsilon.$

— x —

Left Factoring

Consider a grammar, $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \ldots \mid \gamma_1 \mid \gamma_2$

If more than one production on the RHS contains the same prefix, it is said to have left factoring.

(*) Left factoring in grammars creates a problem for top-down parsers as they cannot decide which production must be chosen to parse

the given string, after scanning only the first element of the RHS of the productions having left factoring.

$A \to \alpha A' \mid \delta_1 \mid \delta_2$.

$A' \to \beta_1 \mid \beta_2 \mid \ldots$

eg. $S \to bSSaaS \mid bSSaSb \mid bSb \mid a$

common prefix → bS.

$S \to bSS' \mid a$.

$S' \to SaaS \mid SaSb \mid b$

$S' \to \bullet\, SaS'' \mid b$.

$S'' \to \bullet\, aS \mid Sb$

w = baaaaa.

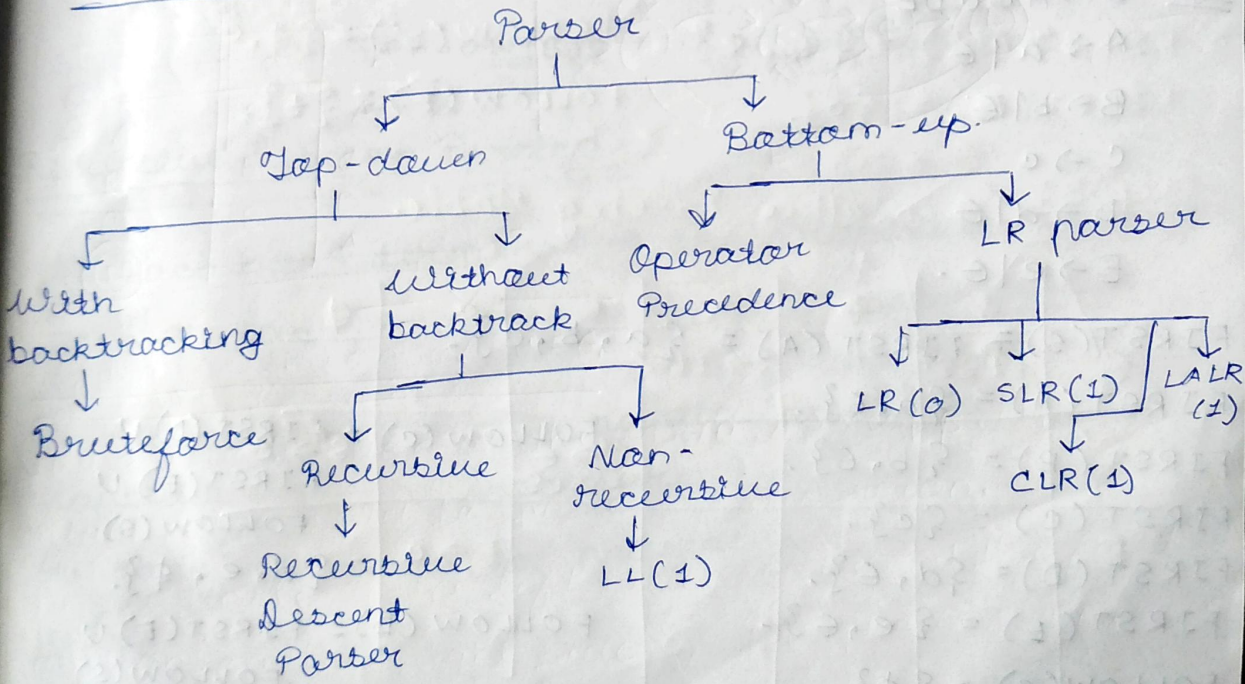
$S \Rightarrow bSS' \Rightarrow baS'$

$\Rightarrow baSaS'' \Rightarrow baaaS''$

$\Rightarrow baaaaS \Rightarrow baaaaa$

—x—

PARSERS



FIRST and FOLLOW :

$S \to ABCD$
$A \to b \mid \epsilon$
$B \to c$
$C \to d$
$D \to e$

FIRST(S) = {b, c}.
FIRST(A) = {b, ∈}.
FIRST(B) = {c}
FIRST(C) = {d}
FIRST(D) = {e}.

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

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

eg. $S \to ABCDE$
$A \to a \mid \epsilon$
$B \to b \mid \epsilon$
$C \to c$
$D \to d \mid \epsilon$
$E \to e \mid \epsilon$.

FIRST(S) = FIRST(A) = {a, b, c}.
FIRST(A) = {a, ∈}.
FIRST(B) = {b, ∈}.
FIRST(C) = {c}.
FIRST(D) = {d, ∈}.
FIRST(E) = {e, ∈}.
FOLLOW(S) = {$}.
FOLLOW(A) = FIRST(B) ∪ FIRST(C).
= {b, c}.
FOLLOW(B) = FIRST(C) = {c}.
FOLLOW(C) = FIRST(D) ∪ FIRST(E) ∪ FOLLOW(S).
= {d, e, $}.
FOLLOW(D) = FIRST(E) ∪ FOLLOW(S)
= {e, $}.
FOLLOW(E) = {$}.

RULES FOR FIRST:

1) $A \to a\alpha$, $\alpha \in (V \cup T)^*$, FIRST(A) = {a}.

2) $A \to \epsilon$, FIRST(A) = {ϵ}.

3) $A \to BC$, $B, C \in V$. ⇒ go to production of B.

   i) If $B \to a \mid b$, calculate FIRST(B). $\notin \epsilon$.

   ii) If $B \to a \mid b \mid \epsilon$, FIRST(A) = FIRST(B) ∪ FIRST(C).

RULES FOR FOLLOW: (always check for those prod. that contain the NT on RHS)

1) If S is start symbol, FOLLOW(S) = {$}.

2) If $A \to \alpha B \beta$, FOLLOW(B) = FIRST(β). [FOLLOW NEVER CONTAINS ϵ]

   i) If $\beta \in (V \cup T)^+$, FOLLOW(B) = FIRST(β)

   ii) If $\beta = \epsilon$, FOLLOW(B) = FOLLOW(A).

   iii) If $\beta = \gamma a$, $\gamma \in (V \cup T)^*$, FIRST(γ) $\in (V \cup T)^*$,

   FOLLOW(B) = FIRST(γ) ∪ FIRST(a).

— X —

Recursive Descent Parsing

1. Each non-terminal → one procedure.
2. For each terminal → check if match → incr. input pointer.
3. All productions of a particular NT to be considered in one procedure.
4. no. of NT = no. of procedure.

$E \to TE'$

$E' \to +TE' \mid \epsilon$

$T \to FT'$

$T' \to *FT' \mid \epsilon$

$F \to (E) \mid id.$

(*) Not LL(1) Grammar

$S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow c$

Step 1: This grammar does not contain left recursion.

But it contains left factoring, as the prefix iEtS is same in multiple productions.

Removing left factoring,

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \epsilon$

$E \rightarrow c$

Step 2: Find FIRST and FOLLOW for all non-terminals.

FIRST(S) = {i, a}.

FIRST(S') = {ϵ, e}.

FIRST(E) = {c}.

FOLLOW(S) = {$, e}

FOLLOW(S') = {$, e}.

FOLLOW(E) = {t}.

Step 3: Predictive Parsing Table.

	i	t	a	e	c	$
S	S → iEtSS'		S → iEtSS'			
S'				S' → ϵ S' → eS		S' → ϵ
E					E → c	

Here, M[S', e] = 2.

So this grammar is not an LL(1) grammar as it has multiple entries in one block of the parsing table.

Handle

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.

eg. $S \to aABC$ $\quad w = abbcdc$

$A \to Abc \mid b$

$B \to d$

Right sentential form	Handle	Reducing production
abbcdc	b	$A \to b$.
aAbcdc	Abc	$A \to Abc$.
aAdc	d	$B \to d$
aABC	aABC	$S \to aABC$
S		

Shift-Reduce Parsing (Bottom up parsing).

Initial config. of stack →

Stack	Input
$	w$
⋮	⋮
$S	$

Actions :-

i) Shift → Parser shifts zero or more input symbols until handle β found on TOP of stack.

ii) Reduce → β is reduced to LHS of the prod.

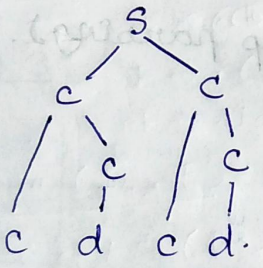
iii) Accept iv) Error.

eg. $S \to cc$

$C \to cC \mid d$

$w = cdcd$.

Stack	Input	Action
$	cdcd $	shift
$c	dcd $	shift.
$cd	cd $	reduce $C \to d$
$cC	cd $	reduce $C \to cC$.
$c	cd $	shift.
$cc	d $	shift.
$ccd	$	reduce $C \to d$.
$ccC	$	reduce $C \to cC$.
$cC	$	reduce $S \to cC$.
$S	$	accept.



Simple LR (SLR) Parsing

1. $E \to E+T$
2. $E \to T$
3. $T \to T*F$
4. $T \to F$
5. $F \to (E)$
6. $F \to id$.

Step 1: Construct augmented grammar.

1. $E' \to E$
2. $E \to E+T$
3. $E \to T$
   $T \to T*F$
   $T \to F$
   $F \to (E)$
   $F \to id$.

Canonical LR(0) collection:

I_0: $E' \to \cdot E$ (if $\cdot$ is followed by NT, write all productions for the NT). (closure operation).
$E \to \cdot E+T$
$E \to \cdot T$.
$T \to \cdot T*F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

goto (I_0, E)

I_1: $E' \to E \cdot$
$E \to E \cdot + T$

goto (I_0, T)

I_2: $E \to T \cdot$
$T \to T \cdot *F$

goto (I_0, F)

I_3 : $T \to F \cdot$

goto (I_0, ( ).

I_4 : $F \to (\cdot E)$
$E \to \cdot E+T$
$E \to \cdot T$
$T \to \cdot T*F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

goto (I_0, id)

I_5 : $F \to id \cdot$

goto (I_1, +)

I_6 : $E \to E+\cdot T$
$T \to \cdot T*F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

goto (I_2, *).

I_7 : $T \to T*\cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

goto (I_4, E).

I_8 : $F \to (E \cdot)$
$E \to E \cdot +T$

goto (I_6, T)

I_9 : $E \to E+T \cdot$
$T \to T \cdot *F$

goto (I_7, F)

I_{10} : $T \to T*F \cdot$

goto (I_8,))

I_{11} : $F \to (E) \cdot$

SLR Parsing Table

(All cases when • followed by terminal, perform shift).

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

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

FOLLOW(T) = FOLLOW(E) ∪ {*}.
= {$, +, *,)}.

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

S → aSSbS | aSaSb | abb | b.

S → aS' | abb | b.

S' → SSbS | SaSb | SbSS | aSbS | bbS | b.

S'' → SbS | aSb | bb | b.

S → SbS | aSb | bb | b.

S'' → SbS | aSb | b.

S''' → b | ∈.

Fill goto table when
- followed by NT.

Fill action table whenever
- at end of production
  ⇓ by reduce action

E → E+T.

In row = item no
and column = FOLLOW(E)
fill with reduce action
alongwith production
number where it is
reduced.

↓

E → E+T.
i.e. prod. 1 here.

$S \rightarrow aSSbS \mid aSaSb \mid abb \mid b.$

$S \rightarrow aS' \mid b.$

$S' \rightarrow SSbS \mid SaSb \mid bb.$ ✗

$S' \rightarrow SS'' \mid bb.$

$S'' \rightarrow SbS \mid aSb.$

Q. $S \rightarrow a \mid ab \mid abc \mid abcd.$

$S \rightarrow aS'$

$S' \rightarrow \epsilon \mid b \mid bc \mid bcd.$

$S' \rightarrow \epsilon \mid bA.$

$A \rightarrow \epsilon \mid c \mid cd.$

$A \rightarrow \epsilon \mid cB.$

$B \rightarrow \epsilon \mid d.$

—— X ——

SOFTWARE ENGG.

Waterfall Model → Each phase must be completed before the next phase can begin and there is no overlapping in the phases.

Illustrates software development process in linear sequential flow. Sequential phases in waterfall model are:

SLR Parsing (using SLR parsing table)

w = id * id + id

1) E → E+T
2) E → T
3) T → T*F
4) T → F
5) F → (E)
6) F → id.

r ⇒ number of elements in RHS of reducing production.

in shift operation, push one input symbol to TOS and write the state to which it is shifted.

→ TOS should always be a state.

STACK	INPUT	ACTION
0	id * id + id $	shift
0 id 5	* id + id $	reduce by (6) # pop 2×r = 2 items from stack.
0 F 3	* id + id $	reduce by (4).
0 T 2	* id + id $	shift
0 T 2 * 7	id + id $	shift
0 T 2 * 7 id 5	+ id $	reduce by (6).
0 T 2 * 7 F 10	+ id $	reduce by (3).
0 T 2	+ id $	reduce by (2)
0 E 1	+ id $	shift
0 E 1 + 6	id $	shift.
0 E 1 + 6 id 5	$	reduce by (6).
0 E 1 + 6 F 3	$	reduce by (4).
0 E 1 + 6 T 9	$	reduce by (1).
0 E 1	$	accept.

Not SLR(1) Grammar

(1) $S \to L = R$

(2) $S \to R$

(3) $L \to *R$

(4) $L \to id$

(5) $R \to L$

Augmented Grammar is —

$S' \to S$

$S \to L = R$

$S \to R$

$L \to *R$

$L \to id$

$R \to L$

Constructing LR(0) items —

I_0: $S' \to \cdot S$
$S \to \cdot L = R$
$S \to \cdot R$
$R \to \cdot L$
$L \to \cdot *R$
$L \to \cdot id$

goto (I_0, S)

I_1: $S' \to S \cdot$

goto (I_0, L)

I_2: $S \to L \cdot = R$
$R \to L \cdot$

goto (I_0, R)

I_3: $S \to R \cdot$

goto $(I_0, *)$

I_4: $L \to * \cdot R$
$R \to \cdot L$
$L \to \cdot *R$
$L \to \cdot id$

goto (I_0, id)

I_5: $L \to id \cdot$

goto $(I_2, =)$

I_6: $S \to L = \cdot R$
$R \to \cdot L$
$L \to \cdot *R$
$L \to \cdot id$

goto (I_4, R)

I_7: $L \to * R \cdot$

goto (I_4, L)

I_8: $R \to L \cdot$

goto $(I_4, *)$

I_4

goto (I_4, id)

I_5

goto (I_6, R)

I_9: $S \to L = R \cdot$

goto $(I_6, L) = I_8$

goto $(I_6, *) = I_4$

goto $(I_6, id) = I_5$

FOLLOW(S) = {$}.

FOLLOW(L) = {=, $}.

FOLLOW(R) = {$, =}

Constructing the SLR Parsing Table. →

State	Action				Goto		
	:id	=	*	$	S	L	R
0	s5		s4		1	2	3
1				acc			
2		s6/r5		r5			
3				r2			
4	s5		s4			8	7
5		r4		r4			
6	s5		s4			8	9
7		r3		r3			
8		r5		r5			
9				r1			

In action table (I_2, =) we see that there are two entries, shift to state 6 and reduce by production 5. This creates shift-reduce conflict in this position as the parser would not know whether to perform shift or reduce operation.

CLR Parsing Table

LR(1) items →

i) closure: $A \to \alpha . B\beta, a$

$B \to .\gamma, FIRST(\beta a)$

ii) goto operation is same as SLR.

iii) Reduce: $A \to \alpha., a$.

1) $S \to CC$

2) $C \to cC$

3) $C \to d$

Augmented grammar →

$S' \to S$

$S \to CC$

$C \to cC$

$C \to d$

LR(1) items

I_0: $S' \to .S, \$$

$S \to .CC, \$$

$C \to .cC, c/d$ ⇒ $FIRST(C\$) = \{c/d\}$.

$C \to .d, c/d$

goto (I_0, S)

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

goto (I_0, C)

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

$C \to .cC, \$$

$C \to .d, \$$ (calculated from $S \to C.C$)

goto (I_0, c)

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

$C \to .cC, c/d$ [$FIRST(\epsilon . c/d) = c/d$].

$C \to .d, c/d$.

goto (I_0, d)

I_4: $C \to d., c/d$.

goto (I_2, C)

I_5: $S \to CC \cdot, \$$

goto (I_2, c) ⇒ already formed in I_3 but lookahead is diff-

I_6: $C \to c \cdot C, \$$

$C \to \cdot cC, \$$

$C \to \cdot d, \$$

goto (I_2, d)

I_7: $C \to d \cdot, \$$

goto (I_3, C)

I_8: $C \to cC \cdot, c|d$.

goto (I_3, c)

I_6

goto (I_3, d)

I_6

goto (I_6, C)

I_8

goto (I_6, c)

I_6

goto (I_6, d)

I_7

goto (I_6, C)

I_9: $C \to cC \cdot, \$$.

State	Action			Goto	
	c	d	$	S	C
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		

For reduce operation, columns are the lookahead symbols.