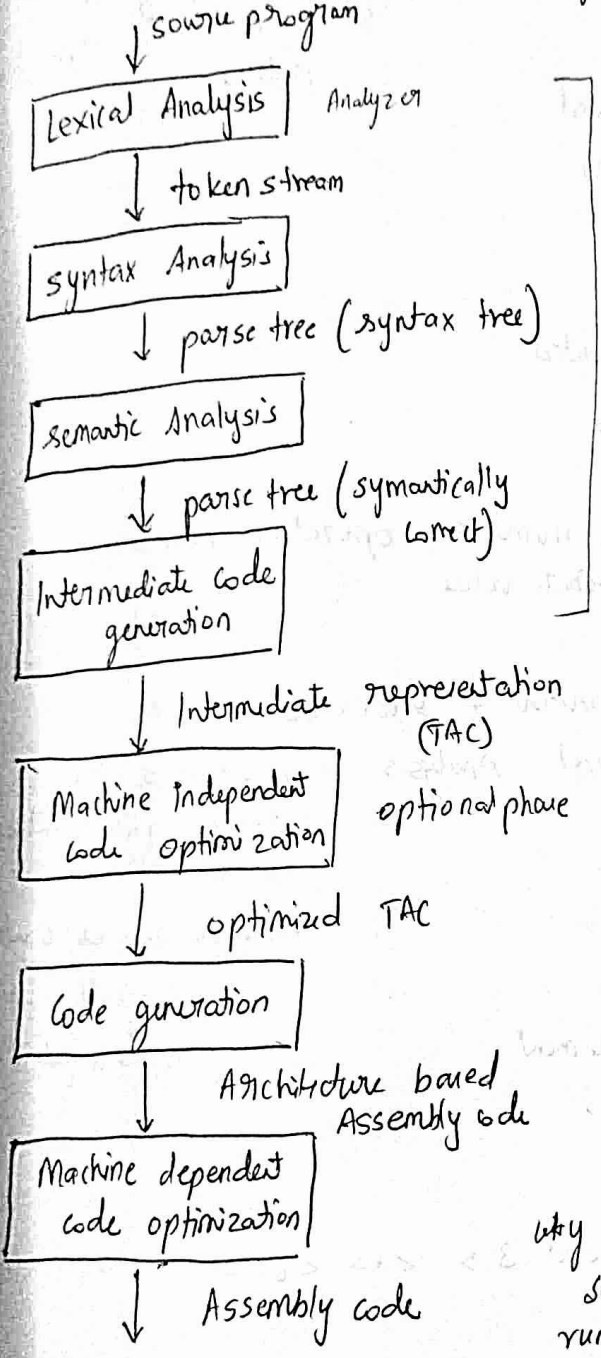
06/April/2022

Language Processing system

1) Preprocessor
2) Compiler
3) Assembler
4) Loader

Phases of compiler



TAC - Three Address Code

3 operands

2 operators - of which one is assignment operator by default

eg:
$c_1 = a + b$

why optimize?
so that time req to run the program is less

Introduction is overview of all phases

Why is it called Frontend?

LLVM

low level virtual Machine

Evaluating expressions

Compiler Reads from left to right

delimiter - space

lexene - set of characters from source program which matches a specific pattern

Rules of language - should be implemented ↓ in Lexical Analyzer

Eg: keyword can't be identifier

What is token? How is it represented

< tokenname , attribute value >

↑ optional

keywords, numbers, operators have no attribute value.

position

↑↑↑↑↑↑↑↑

↑ retrace

position = initial + rate x 60;

After lexical Analysis → position is also identifier, rate is also identifier

How do you differentiate attribute value

↓

Symbol table

Exam:

All phases of compilation Textbook ✓

i/p + o/p of each phase

Translation of an assignment statement

position = initial + rate * 60;

tokens ↓

<id, 1> <=> <id, 2> <+> <id, 3> <*> <60>

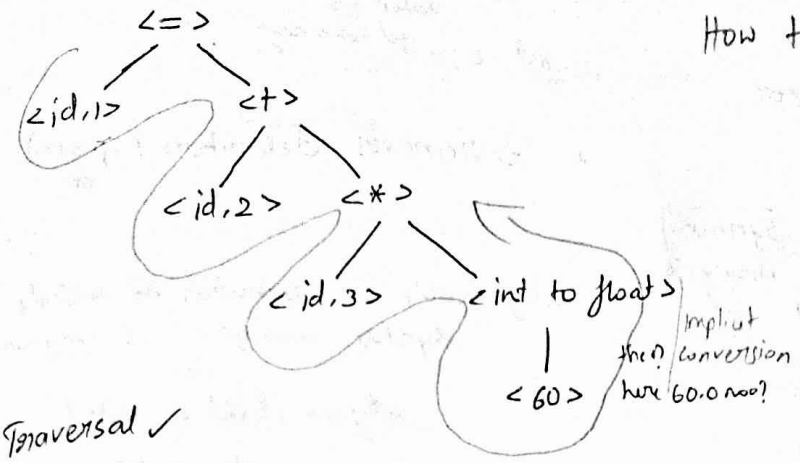
Symbol table

1 position

2 initial

3 rate

Write parse tree for above



How to write parse tree?
I forgot

Traversal ✓

for an operator, 2 operands should be there.

<id,1> 2nd is not operand it is an operator so evaluate it first

<id, 2> again 2nd operator evaluate it is

<id,3>, operator?

then implicit conversion here 60.0 now? No, it is now operand, do the operation - int to float then it becomes 60.0

TAC

```
t1 = (int to float) 60
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

machine independent code optimization

```
t1 = id3 * 60.0
t2 = id2 + t1
id1 = t2
```

or

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
id1 = id2 + t1
```

Assembly code for the above ~~optimization~~ TAC ? How to write

Textbook →

```
LDF   R2, id3
MULF  R2, R2, #60.0
LDF   R1, id2
ADDF  R1, R1, R2
STF   id1, R1
```

LD load, F for float

I have forgotten all these :(

This whole ans representation in textbook is ♥

mces

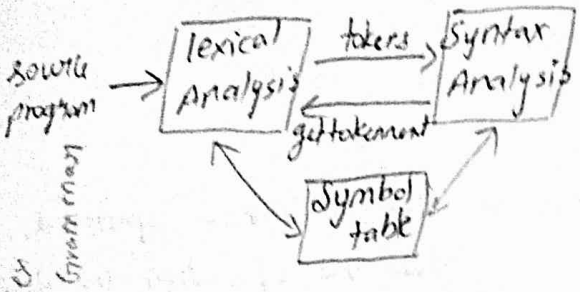
07/April/2

Lexical Analysis

Role of lexical Analyzer

int a; (token → get tokenized)

↓ → remove delimiters (spaces) or



can't impose structure using Regex
structure is imparted using Grammars

why there's a separation as lexical & syntax analysis (token generation)

why we divide a task? to complete it fast

when technology improves we can apply some techniques to lexical & syntax analysis to make it faster

for lexical Analysis - we can apply some buffering techniques.

std input

Syntax analysis -

1) Scanning - Remove extra spaces & comments (remove whatever is not required)

2) lexical analysis - tokenization

- Token is pair consisting of token name and an optional attribute value.
- Token name is an abstract symbol representing a kind of lexical unit
- Pattern - description of the form that the lexemes of the token may take.
- lexeme - is a sequence of characters in the source program that matches the pattern for a token & is identified by the lexical analyzer as an instance of that token

token
↑
pattern
↑
→ lexeme

lexeme	pattern	token
printf	letter followed by letters/digit	<id, 1>
(	what is the pattern for this? (only, fixed.	<(>
"	"	<">
token = %d	Anything between " "	literal
"	"	<">
,	,	<,>
score	letter followed by letters/digit	<id, 2>
)	)	<)>
;	;	<;>

" - this too counts as a token

printf("total %d", score);

what are printf & scanf? function names defined in std i/p o/p library

change the name in std files you can use the changed file name then

symbol table

1	printf	....
2	score	

id number lexeme blank for now detail - future chapters

For problems like this you must write symbol table as well.

How to write → see text book

Types of tokens

one type of token for: 1) keywords — differentiate b/w keywords using attribute values

2) operators

3) identifier

4) constants

5) punctuation symbols

Recovery method in Lexical Analysis

Panic mode error recovery - Handles simple things — look ahead methods

1) transpose fi → if 1/2 letters

fr → for

Why correction is needed here?

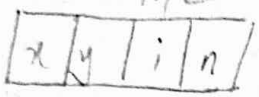
Input buffering - to read fast

int a

in one I/O operation.. it reads many characters & store it in buffer then reads from buffer.

Single buffer

problem ! :



next cycle

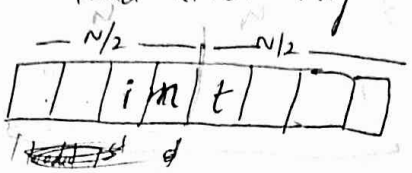
t | x | y | z

in is overwritten

so ->

Buffer pairs

load alternatively

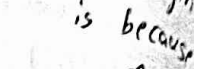


->2 pointers

1) lexene begin

2) forward

restriction on identifier length is because of



buffer length.

Sentinels -

inserted at the end of buffer

should not appear in source program / input stream

what could be the candidate character for sentinel

EOF

what if you read EOF not at the end?

=> end of the program :)

matches this with some pattern & forms as token

after this both pointers move to next lexeme

for each character read, it needs to check 2 things

1) character which is read

2) whether it has reached buffer end - solved using sentinels

Specifications of token

12/April/202

Regular expression

alphabet
string
language

operations on strings

1) prefix
2) suffix
3) substring
4) proper prefix, suffix, substring
   ↓
   excluding ε & original string

subsequence

operations on languages

1) Union of L and M
2) Concatenation of L & M
3) Kleene closure of L
4) Positive Closure

Notations

$L \cup M = \{ s \mid s \text{ is in } L \text{ or } M \}$

$LM = \{ st \mid s \text{ is in } L \text{ and } t \text{ is in } M \}$

$L^* = \bigcup_{i=0}^{\infty} L$

$L^+ = \bigcup_{i=1}^{\infty} L$

Law
r's are RE

1) $r|s = s|r$
2) $r|(s|t) = (r|s)|t$
3) $r(st) = (rs)t$
4) $r(s|t) = rs|st$
   $(s|t)r = sr|tr$
5) $\varepsilon r = r\varepsilon = r$
6) $r^* = (r|\varepsilon)^*$
   $r^{**} = r^*$

Definitions

| is commutative

| is associative

Concatenation is associative

Concatenation is distributive over |

Diff b/w REs & Regular Definitions

How do you define a digit in lex terms

[0-9]

can we have something like this

int _ ; allowed

Digit → [0-9] · Regular definition.

letter → [a-zA-Z_] underscore included.

Regex for identifier [a-zA-Z_][a-zA-Z0-9_]*

will simply this using Regular Defn.

letter (letter/digit)*

1) Write Regular Expression for Unsigned number.

definition

start with digits

digit → [0-9] →

allowed Unsigned no.

52
80
0.0123
6.66E4
1.89E-4
1.89E4
6.02E-5
.25

Ans:

digit → 0/1/2/.../9

digits → digit digit*

optionalFraction → .digits/ε (.digits)/ε

? → 0 or 1 occurance

optionalExponent → (E(+/-/ε))digits/ε

Unsigned number → digits.optinalFraction.optionalExponent

while using ?

digits(.digits)? (E(+/-)? digits)? ✗

digits((.digits)? (E(+/-)? digits)?)? ✗ No need

52E64 shouldn't be accepted without • E shouldn't come.

digits((.digits)(E(+/-)? digits)?)?

Recognition of Tokens

digit → [0-9]

digits → digit+

number → digits (.digit)? (E(+/-)? digits)?

letter → a|b|...z

id → letter(letter|digit)*

if → if

then → then

else → else

relop → < | > | == | <= | >= | != (or <>)

Acc to some books == can be =, != can be <>

13-04-2022

Grammar

stmt → if expr then stmt

| if expr then stmt else stmt

| ∈

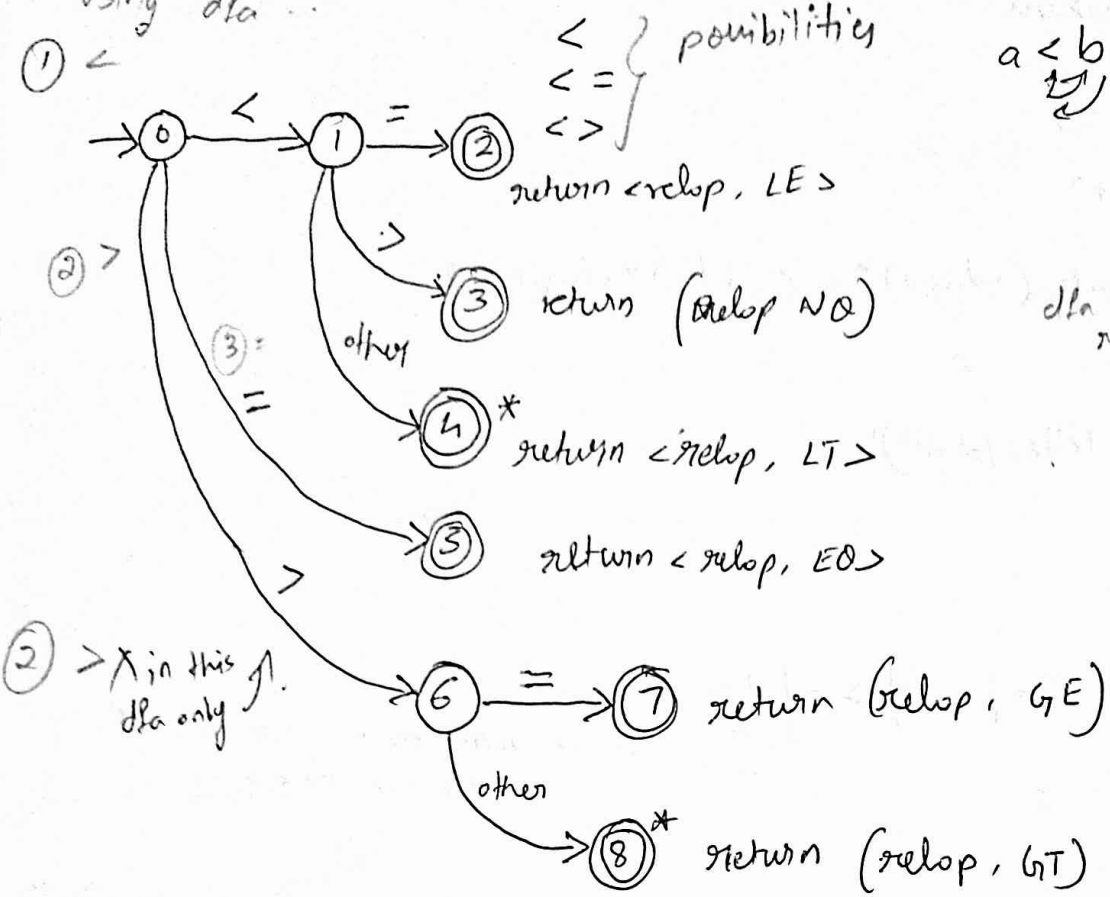
expr → term relop term

| term

term → id | number

lexeme	pattern	token
if	if	keyword token
else	else	—
then	then	—
any id	letter(letter|digit)*	entry to the symbol table
<	[< <= > ..]	LT (<relop, LE>)
>	— —	GT
<=		LE

using dfa ..

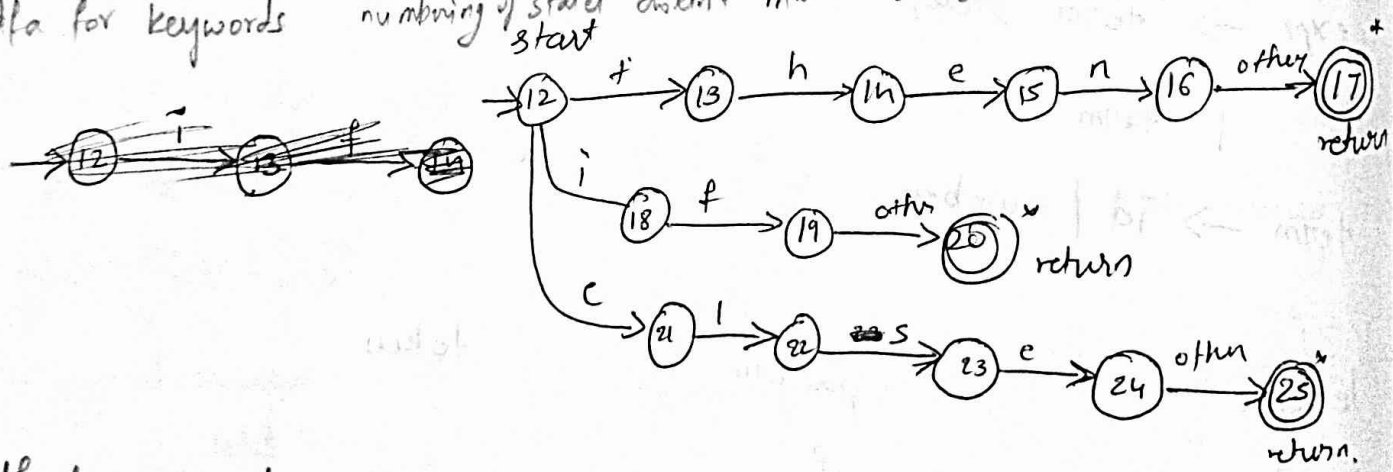


a < b
retrack -> denoted by *
** -> double retrack while

dfa to recognize relop

dfa for id.

dfa for keywords numbering of states doesn't matter .. switch case it is



dfa for unsigned numbers

```
TOKEN getRelop()
{
    TOKEN setToken = new (RELOP);
    while(1)
    {
      switch (state)
      {
        case 0 : c = nextToken();
                 if (c == '<') state = 5;
                 elseif (c == '>') state = 6;
                 else fail();
                 break;
        case 1 : ...
        case 8 : retract();
                 setToken.attribute = GT;
                 return setToken;
                 break;
      }
    }
}
```

Chapter 2

Role of parser

1) Universal parser — can't handle Ambiguity grammars & grammars having loop.
2) Top down parser - LL grammar — also ε must be removed; no left factoring; no left recursion
3) Bottom up - LR grammar. — can handle Ambiguous grammar; easy; YACC implements LR grammar

Error Recovery Strategy

1) Panic Mode error recovery
2) Phrase level — ——
3) Error Production
4) Global correction

$S \rightarrow SS+ \mid SS* \mid a$

input string : aa+a*

Derive this string using LMD and RMD.

LMD: → is for production, ⇒ is for derivation

$S \xrightarrow{lmd} \underline{S}S* \Rightarrow SS+S* \Rightarrow aS+S* \Rightarrow aa+S* \Rightarrow aa+a*$

(don't skip any step)

RMD

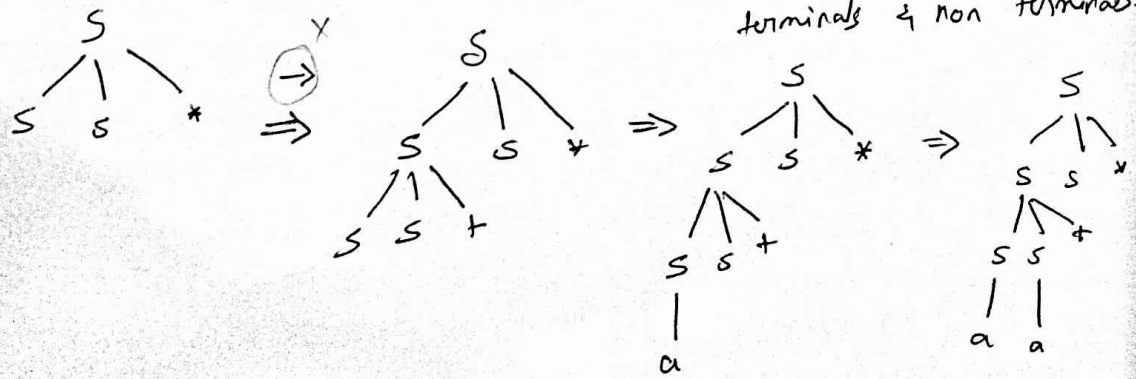
$S \xrightarrow{rmd} S\underline{S}* \Rightarrow S\underline{a}* \Rightarrow S\underline{S}+a* \Rightarrow \underline{S}a+a* \Rightarrow aa+a*$

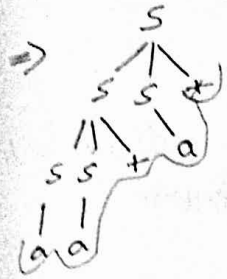
Sentence - has only terminals, final derivation

Sentential form - intermediate form in the derivation which contains both terminals & non terminals.

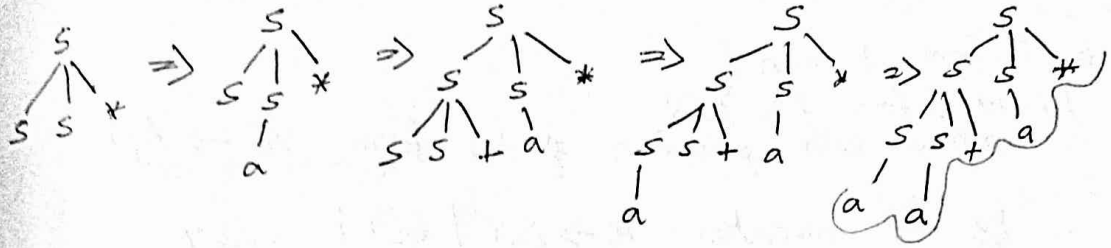
Tree also step wise

lmd





RMD Tree



Ambiguity in Grammars

$E \rightarrow E+E \mid E*E \mid id$

$id + id * id$

Elimination of Left Recursion

$A \rightarrow A\alpha \mid \beta$

then $A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

$A \rightarrow A\alpha_1 \mid \beta_1 \quad A \rightarrow \beta_1 A'$

$A' \rightarrow \alpha_1 A' \mid \varepsilon$

~~$A' \rightarrow \alpha_1 A$~~

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Left Factoring.

$S \rightarrow Aab \mid Aabd \mid Aabde$

$A \rightarrow t$

rewrite as →

$S \rightarrow AabS'$

$S' \rightarrow \epsilon \mid d \mid de$

$A \rightarrow t$

$S \rightarrow AabS'$

$S' \rightarrow \varepsilon \mid dS''$

$S'' \rightarrow \varepsilon \mid e$

$A \rightarrow t$

20/09/2022

Algorithm for eliminating left Recursion

Input to the algo. is grammar with no cycles or null productions
output → an equivalent grammar with no left recursion.

1. Arrange the non terminals in some order $A_1, A_2 .. A_n$
2. For each i from 1 to n {
3. for each j from 1 to $i-1$ {
   replace each production of the form $A_i \rightarrow A_j \gamma$
4. by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid .. \delta_k\gamma$
   where $A_j \rightarrow \delta_1 \mid \delta_2 \mid .. \mid \delta_k$ are all current A_j productions
5. }
6. eliminate immediate left recursion among the A_i productions.

1) $S \rightarrow abcd \mid e \mid A$

$A \rightarrow \underbrace{Aab}_{\alpha_1} \mid \underbrace{Ac}_{\alpha_2} \mid \underbrace{b}_{\beta_1} \mid \underbrace{B}_{\beta_2}$

$B \rightarrow d \mid e$

$A \rightarrow bA' \mid BA'$

$A' \rightarrow abA' \mid cA' \mid \varepsilon$

$\Rightarrow abA' \mid cA' \mid ab \mid c$

$S \rightarrow abcd \mid e \mid A$ } as it is

$B \rightarrow d \mid e$

After removing ~~recn~~ left recursion don't forget to write whole grammar

2) $B \rightarrow \underbrace{B \text{ or } T}_{\alpha_1} \mid \underbrace{T}_{\beta_1}$

$T \rightarrow \underbrace{T \text{ and } F}_{\alpha_1} \mid \underbrace{a}_{\beta_1}$

$F \rightarrow true \mid false \mid (B)$

$B \rightarrow ~~BT~~ TB'$

$B' \rightarrow orTB' \mid \varepsilon$

$T \rightarrow aT'$

$T' \rightarrow andFT' \mid \varepsilon$

$F \rightarrow true \mid false \mid (B)$

3) $S \to a \mid (L)$

$L \to L, S \mid S$

$S \to a \mid (L)$

$L \to SL'$

$L' \to ,SL' \mid \varepsilon$

4) $S \to S+S \mid SS \mid (S) \mid S* \mid a$

~~$S \to +SS' \mid SS' \mid$~~

$S \to (S)S' \mid aS'$

$S' \to +SS' \mid SS' \mid *S' \mid \varepsilon$

5) $S \to SS+ \mid SS* \mid a$ ✱

$S \to aS'$

$S' \to S+S' \mid S*S' \mid \varepsilon$

ambiguous : both starting from same non terminal

left factoring.

$S' \to SS'' \mid \varepsilon$

$S'' \to +S' \mid *S'$

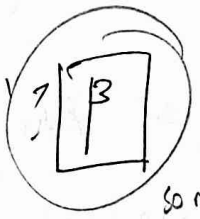
First write the rule then the Grammar

Best

β with X'

$X' \to \beta X'$ not α

α or β with X ? No!! ang so much of confusion



Algorithm for left factoring

Input : Grammar G

output : An equivalent left factored grammar.

Method:

For each non terminal A, find the longest prefix α common to two or more of its alternatives

if $\alpha \neq \varepsilon$ ~~there~~ there is no non trivial common prefix

Replace all of the A productions by

$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \alpha\beta_k \mid \gamma$ where γ represents all alternatives that donot begin with α

$A \to \alpha A' \mid \gamma$

$A' \to \beta_1 \mid \beta_2 \cdots \mid \beta_k$

1) $S \to iEtS \mid iEtSeS \mid a$

$E \to b$

2) $S \to 0S' \mid 1$

$S' \to S1 \mid 1$

$S \to iEtSS' \mid a$

$S' \to \varepsilon \mid eS$

$E \to b$

Top Down Parser $LL(1)$ (looking 1 symbol ahead.)

$LL(k)$ — Read from left 1nd; looking k symbols ahead

1) Recursive Decent Parser - no lookahead.

no definit actions

Recursively try all productions

1) FIRST set Rules

1. $FIRST(a) = \{a\}$, a is terminal
2. $FIRST(A) = \{a\}$ if $A \to a\alpha$ (↳ terminal)
3. If $A \to B\alpha$, $B \to \varepsilon$? $FIRST(A) = FIRST(B)$
4. if $A \to B\alpha$, $B \to \varepsilon$ $FIRST(A) = \{FIRST(B) - \varepsilon\} \cup FIRST(\alpha)$
5. if $A \to \varepsilon$ then $FIRST(A) = \{\varepsilon\}$
6. $A \to B\alpha$, $B \to \varepsilon$, $\alpha \to \varepsilon$ then $FIRST(A) = FIRST(B) \cup FIRST(\alpha) \cup \{\varepsilon\}$

1) $E \to TE'$

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

$T \to FT'$

$T' \to *ET' \mid \varepsilon$

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

$FIRST(E) = FIRST(T) = FIRST(F)$ (3rd rule →) (3rd rule)

$FIRST(F) = \{ (, id \}$ (2nd rule) (1st rule)

E ✓ T ✓ F ✓

remaining E' & T'

consider all alternating — not just ε

$E' \to +TE'$ & ε both do not just stop for $+TE'$

$FIRST(E') = \{ +, \varepsilon \}$ (2nd rule) (5th rule)

$FIRST(T') = \{ *, \varepsilon \}$ (2nd rule) (5th rule.)

21/04/2022

2) $S \to \underline{A}BcD \mid \underline{A} \mid B$

$A \to a \mid \varepsilon$

$B \to b \mid \epsilon$

$D \to \epsilon$

~~$FIRST(S) = FIRST(A)$~~

remove left factoring

$S \to AS' \mid \underline{B}$ doesn't come in LF only in remove LR while

$S' \to BcD \mid \varepsilon$

$A \to a \mid \varepsilon$

$B \to b \mid \varepsilon$

$D \to \epsilon$

$FIRST(S) = FIRST(A) \cup FIRST(B) \cup FIRST(S')$

$= \{ \epsilon, a, b, c \}$

$FIRST(S') = FIRST(B) \cup \{\varepsilon\}$

$= \{ \varepsilon, b, c \}$

Be careful if it is (capital letters) non terminal

4th rule. $\{FIRST(B) - \varepsilon\} \cup$ + 5th rule $FIRST(cD) \cup$ ~~FIRST~~ $\{\varepsilon\}$

3) $T \to aBcD \mid D$

$D \to d$

$B \to d.$

$FIRST(T) = \{a\} \cup FIRST(D)$

$= \{a, d\}$

$FIRST(D) = \{d\}$

$FIRST(B) = \{b\}$

3) $S \to \underbrace{SaSb}_{\alpha_1} \mid \underbrace{SbSa}_{\alpha_2} \mid \underbrace{\varepsilon}_{\beta}$

Remove L recursion

$S \to S'$

$S' \to aSbS' \mid bSaS' \mid \varepsilon$

$F(S) = F(S') = \{a, b, \varepsilon\}$

$F(aBCS)$

if terminal $\Rightarrow$ only a terminal will come is there

if non terminal is there,

$F(\text{non terminal})$

if that non terminal is having ε production, we to consider variable next to it

4) $S \to L=R \mid R$

$R \to L$

$L \to *R \mid id$

$F(S) = F(L) \cup F(R)$

$F(L) = \{*, id\}$

$F(R) = F(L) = \{*, id\}$

$F(S) = \{*, id\}$.

* in first set, $ will not come.

* in follow set, you'll never get ε

Follow set Rules

1) $FOLLOW(S) = \{\$\}$ where S is start symbol

2) $A \to \alpha B \beta$ where $\beta \not\Rightarrow \varepsilon$, $FOLLOW(B) = FIRST(\beta)$

3) $A \to \alpha B$ then $FOLLOW(B) = FOLLOW(A)$

4) $A \to \alpha B \beta$ where $\beta \Rightarrow \varepsilon$, $FOLLOW(B) = \{FIRST(\beta) - \varepsilon\} \cup FOLLOW$

i) $E \to \underline{TE'}$

$E' \to \underline{+TE'} \mid \varepsilon$

$T \to FT'$

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

$F \to (E) \mid id$

in Firstset we considered head of production

for follow set see body of production.

1) $FOLLOW(E) = \{), \$\}$

2nd rule, 1st rule

↓

see where E comes in body of production

2) $FOLLOW(E') = FOLLOW(E) \cup FOLLOW(E')$

$= \{), \$\}$

FOLLOW(T) = {FIRST(E') − ε} ∪ FOLLOW(E) ∪ FOLLOW(E') ?

4th rule

= {+,), $}

E → T E'

∩ FIRST(E') = ε But!!!

FOLLOW(T') = FOLLOW(T) ∪ FOLLOW(T')

mark the production first

= {+,), $}

closure - follow

FIRST

if giving ε

FIRST = ε

FOLLOW(F) = {FIRST(T') − ε} ∪ FOLLOW(T) ∪ FOLLOW(T') ?

x . ε x

mark the production

= {*, +,), $}

T → F T'

T' → * F T'

FIRST(α)

↓

see if α gives ε

include FOLLOWS OR FIRSTs

when nothing is after α

↓

if something is there after α

Aα & we are finding FOLLOW(A)

1) S → ABcD | A | B

A → a | ε

B → b | ε

D → ε

remove left factoring

S → AS' | B

S' → BcD | ε

A → a | ε

B → b | ε

D → ε

FOLLOW(S) = {$}

FOLLOW(S') = FOLLOW(S) = {$}

FOLLOW(A) = FIRST(S') − ε ∪ FOLLOW(S)

= {b, c, ε, $}

FOLLOW(B) = FOLLOW(S) ∪ FIRST(CD)

= {$, c}

FOLLOW(D) = FOLLOW(S') − ε ?

= {$}

or only with FIRST

Yes!

2) $T \to aBcD \mid D$

$D \to d$

$B \to d$

$FOLLOW(T) = \{\$\}$

$FOLLOW(D) = FOLLOW(T) = \{\$\}$

$FOLLOW(B) = FIRST(D)$

$= \{c\}$

3) $S \to S'$

$S' \to aSbS' \mid bSaS' \mid \varepsilon$

aS'

$FOLLOW(S) = \{\$\} \cup FIRST(b) \cup FIRST(a)$ — we are Assuming is it correct actually it is $FIRST(aS')$ = a only

$= \{\$, b, a\}$

$FOLLOW(S') = FOLLOW(S) \cup FOLLOW(S')$

$= \{\$, b, a\}$

4) $S \to L = R \mid R$

$R \to L$

$L \to *R \mid id$

$FOLLOW(S) = \{\$\}$

$FOLLOW(L) = FOLLOW(R) \cup FIRST(=)$

$\{=, \$\}$ $= R.$

$FOLLOW(R) = FOLLOW(S) \cup$ ~~FOLLOW(R)~~ $\cup$ $FOLLOW(L)$

$= \{=, \$\}$

substitute FOLLOW(R) in FOLLOW(L) expression.

in case of something like above comes we ignore. RHS FOLLOW

eg $FOLLOW(R) = \ldots \cup FOLLOW(R)$ — ignore this.

★ $A \to BCD$

Follow(B)? thought it's only FIRST C seems it is FIRST(CD)

got it.

Q. Find FIRST set of each Non terminals of the following grammar which is required for Top down parsing

Combined doubt.

$S \rightarrow Aa \mid b$

$A \rightarrow \underbrace{Ac}_{\alpha} \mid \underbrace{Sd}_{\beta} \mid \underbrace{\varepsilon}_{\beta}$

How to remove left recursion practically it is coming should I remove ε first? Eg.

$FIRST(S) = FIRST(A) \cup \{b\} = \{a, b, c\}$

$\cup FIRST(a)$

$FIRST(A) = (FIRST(A) - \varepsilon) \cup FIRST(S) \cup FIRST(c)$

$= \{a, b, c\}$

$FOLLOW(S) = FIRST(d) = \{d, \$\}$

$Follow(A) = FIRST(a) \cup FIRST(c) = \{a, c\}$

Top down parsing

first you need to remove left recursion

$S \rightarrow Aa \mid b$

$A \rightarrow \underbrace{Sd}_{\alpha_1} A \mid \underbrace{bd}_{\beta_1} \mid \underbrace{Ac}_{\alpha_2} \mid \underbrace{\varepsilon}_{\beta_2}$

$A \rightarrow adA' \mid cA'$

$A' \rightarrow bdA' \mid$

$S \rightarrow Aa \mid b$

$A \rightarrow \underbrace{Ac}_{\alpha_1} \mid \underbrace{Sd}_{\beta_1} \mid \underbrace{\varepsilon}_{\beta_2}$

$A \rightarrow Sd A' \mid \varepsilon A'$

$A' \rightarrow cA' \mid \varepsilon$

$FIRST(S) = FIRST(A) \cup \{b\}$

$FIRST(A) = FIRST(S) \cup FIRST(A')$

$= FIRST(A) \cup \{b\} \cup \{c, \varepsilon\}$

$FIRST(A') = FIRST(c) \cup \varepsilon$

$= \{c, \varepsilon\}.$

$FIRST(A) = \{b, c, \varepsilon\}.$

$FIRST(S) =$

What I learnt from this? →

When you're given a grammar & asked to find out First set & Follow set

First 1) remove left recursion

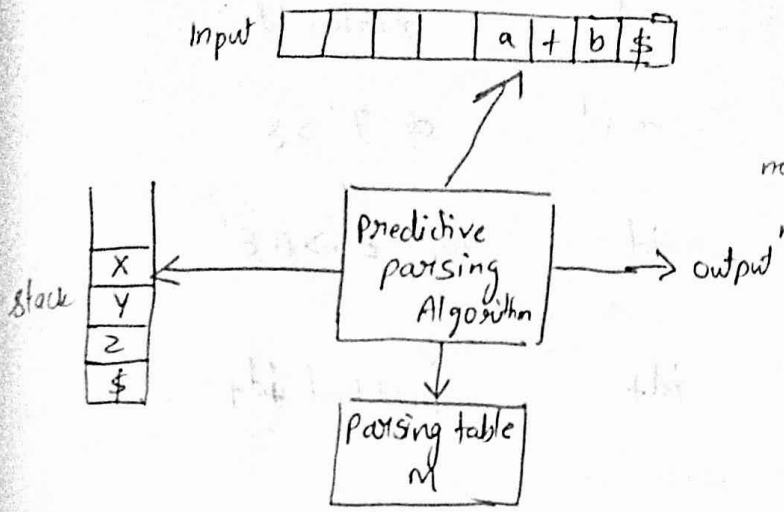
2) Remove left-factoring

3) Find FIRST set

4) then find follow set.

Non recursive Predictive Parsing

05 May 202



no left recursion. no left factoring ← proper Grammar

LMD always

↑ First & Follow set

↑ Table

↑ LMD

8 + 8

16 marks

~~Rotate~~

which set do we consider to write the table? First set?

empty slot - panic mode recovery

M[A, a]	Input symbol					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$		

$ represents end marker in stack → bottom of stack

w = id + id * id → do lmd grammar? E → .. that one by seeing the table :)

we always replace what's there on top of stack

input	stack	matched	Action
id + id * id $ ↑	E $ (start symbol; we always start with start symbol)	–	
id + id * id $ ↑	TE' $ (replaced using table)	–	o/p E → TE' E replaced by TE'
id + id * id $ ↑	FT'E' $	–	o/p T → FT'
id + id * id $ ↑	idT'E' $ (pop)	–	o/p F → id
+ id * id $	T'E' $	id	matched id.

input	stack	matched	action
+id*id$	T E' $	id	matched id
+id*id$	E' $	* id ? ✓	o/p T' → ε
+id*id$	+ T E' $	id ? ✓	o/p E' → +TE'
id*id$	T E' $	id+	matched +, X +

pop

how much/what is matched till now

how long this process continues →
till the input is not null
successful match → both in i/p & stack we should get $.

id*id$	F T' E' $	id+	o/p T → FT'
id*id$	id T' E' $	id+	o/p F → id
*id$	T' E' $	id+id	matched id+id, X id
*id$	* F T' E' $	id+id	o/p T' → *FT'
id$	F T' E' $	id+id*	matched *
id$	id T' E' $	id+id*	o/p F → *id
$	T' E' $	id+id*id	matched id
$	E' $	id+id*id	o/p T' → ε
$	$	id+id*id	o/p E' → ε

pop & put it in matched

But way to remember - Actually that is what is happening

pop & put it in matched

don't stop here, both i/p & $tack should have $ at the end

$S \to a \mid (L)$

$L \to L,S \mid S$ $\quad w = (a, (a, a))$

write LMD first - it will become easy for you to track back if you have missed somewhere

1) Eliminating left recursion

$L \to \underline{L,S} \mid \underline{S}$
$\quad\quad \alpha \quad\quad \beta$

$L \to \beta L'$
$L' \to \alpha L' \mid \varepsilon$ ✓

$S \to a \mid (L)$

$L \to SL'$

$L' \to ,SL' \mid \varepsilon$

2) No left factoring

3) First sets

$FIRST(S) = FIRST(a) \cup FIRST((L))$
$= \{a, (\}$

$FIRST(L) = FIRST(S) = \{a, (\}$

$FIRST(L') = FIRST(,SL') \cup \{\varepsilon\}$
$= \{, , \varepsilon\}$

$FOLLOW(S) = FIRST(L') - \{\varepsilon\} \cup FOLLOW(L) \cup \$$
$= \{,\} \cup \{)\} = \{, ,), \$\}$

$FOLLOW(L) = FIRST()) = \{)\}$

$FOLLOW(L') = FOLLOW(L) \cup FOLLOW(L')$
$= \{)\}$

M[A, a]	Input symbol				
	(	)	,	a	$
S	$S \to (L)$			$S \to a$	
L	$L \to SL'$			$L \to SL'$	
L'		$L' \to \varepsilon$	$L' \to ,SL'$		

$\omega = (a, (a, a))$

input	stack (don't directly write (L), first write the start symbol)	matched	Action
(a, (a, a))$ ↑	S$	–	–
(a, (a, a))$ ↑	(L)$ pop	–	o/p S -> (L)
a, (a, a))$ ↑	L)$	(	matched (
a, (a, a))$ ↑	SL')$	(	L -> SL'
a, (a, a))$ ↑	aL')$ pop	(	S -> a
, (a, a))$ ↑	L')$	(a	matched a
, (a, a))$ ↑	,SL')$	(a	L' -> , SL'
(a, a))$ ↑	SL')$	(a,	matched ,
(a, a))$ ↑	(L)L')$	(a,	S -> (L)
a, a))$ ↑	L)L')$	(a, (	matched (
a, a))$ ↑	SL')L')$	(a, (	L -> SL'
a, a))$ ↑	aL')L')$	(a, (	S -> a
, a))$ ↑	L')L')$	(a, (a	matched a
, a))$ ↑	,SL')L')$	(a, (a	L' -> , SL'
a))$ ↑	SL')L')$	(a, (a,	matched ,
a))$ ↑	aL')L')$ pop	(a, (a,	S -> a

))$	L'))$	(a,(a,a	matched a
))$	L'))$	(a,(a,a	L' -> ε
)$	L'))$	(a,(a,a)	matched)
)$	L'))$	(a,(a,a)	L' -> ε
$	$	(a,(a,a))	matched)

2) S -> 0S1 | 01

S -> +SS | *SS | a

w = 000111

w = +*aaa