Module 2

1. CFG
2. Derivation of string
   a. leftmost
   b. Rightmost
3. Derivation tree and parse tree
4. Ambigous grammar with eg.

Elimination of left recursion

A grammar is left recursive if it has a non-terminal / variable A such that there is a derivation $A \overset{*}{\Rightarrow} A\alpha$ for some string α.

Top down parsing methods can't handle left recursive handle so, transformation that eliminates left recursion is needed.

A left recursion of production $A \rightarrow A\alpha / \beta$ can be replaced by the non-left recursive production.

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' / \varepsilon$$

eg. Consider the following grammar for arithmetic Expression.

E → E+T/T
T → T+F/F
F → (E)/id

left recursion is of the form $A \to A\alpha/\beta$

$A \to \beta A'$
$A' \to \alpha A'$

→ E → E+T/T
A A α β

$E \to TE'$
$E' \to TE'/\varepsilon$

→ T → T+F/F
A A α β

$T \to FT'$
$T' \to +FT'/\varepsilon$

→ F → (E)/id

Non-left recursion.

S → Aa/b —①
A → Ac/Sd/ε —②

There is an indirect left recursion ① can be written as, S → Sda/b.

Eliminate A → Sd with production of S

S → Aa/b —①
A → Ac/Aad/bd/ε —②

Now only direct recursion

② ⇒ A → Ac/Aad/bd/ε
A A α1 A α2 β1 β2

$A \to bdA'/A'$
$A' \to cA'/adA'/\varepsilon$

Syntax Analysis

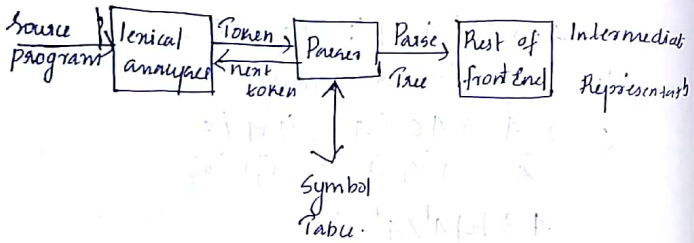
Syntax analysis / parsing is the second phase of a compiler. Syntax analyser receives tokens from lexical analyser and check the syntax of the given programming stmts with the help of CFG.

The syntax analyser generates parse tree as it's output. Every program language has rules that prescribe syntax structure of pgm link.

The structure of pgm language can be described by CFG / BNF (Backus - Naur form)

Parsing:

The syntax analyser follows production rule defined by CFG. The role of parser / syntax analyser is shown in below fig.

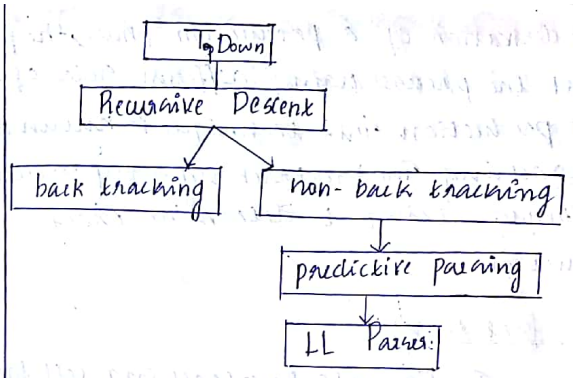


The way the production rules are applied to derive strings from grammar divides parsing into 2 types.

1) TOP-DOWN PARSING
2) BOTTOM-UP PARSING.

TOP-DOWN PARSING

It attempts to build the parse tree from root to leaf. Top-down parser will start from the start symbol and then tries to transform the start symbol into the input stream. It follows left most derivation. In left most derivation the left most terminal is always choosen. The types of top-down parsing are described below:



Recursive Descent parsing:

Recursive descent is a TD parsing technique that constructs from the top and the i/p is read from left to right it uses procedures for every terminal and non-terminal entity. This parsing technique recursively pass into make a parse tree, which may or may not require back tracking.

A form of recursive descent parsing that doesn't require any back tracking is known as predictive parsing

TD parses starts from the root node and match the i/p string against the production rule

If one derivation of a production phase, the parser restart the process using different rules of same production. This technique is known as backtracking. The backtracking may process the input string to determine exact production.

9/2/18

Left factoring:

To eliminate backtracking, left factoring is used. Left factoring is a process of factoring common prefixes of the grammar rule.

→ A grammar with common prefixes

$A \to \alpha\beta_1 / \alpha\beta_2$ [Here α is the common prefix for β_1 and β_2]

~~A →~~ (left)

can be factored as

$A \to \alpha A'$

$A' \to \beta_1 / \beta_2$

eg:1 Eliminate leftfactor in the given below grammar.

$S \to icts / ictses / a$

$C \to d$

Soln:

common prefix present is icts

after left factoring grammar gives

$S \to icss' / a$

$S' \to \varepsilon / es$

$C \to d$

---

eg:2 $E \to E$ sub E sub E

$E \to E$ sub E

$E \to \{E\}$

$E \to d$

common prefix present in E is E sub E

$E \to E$ sub $E E'$

$E' \to$ sub E / ε

$E \to \{E\}$

$E \to d$.

First and follow function

First

First of a grammar symbol is the set of first terminal present in the strings derived from X. The rule to calculate the F(X) are given below.

1) If X is a terminal a then F(X) = a

2) If there is a production $X \to \varepsilon$ then $F(X) = \{\varepsilon\}$

3) if there is a production $X \to Y_1 Y_2 Y_3 \cdots Y_k$ then First of $X = F(Y_1) F(Y_2), F(Y_3) \ldots F(Y_k)$

eq: Find First of symbols present in grammar.

$S \to A\ B$
$A \to a / \varepsilon$
$B \to b$.

Soln

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

Follow(S)

A follow of a non-terminal 'A' is a set of terminals that follow or occur to the right of A. Follow(A) is calculated using following set of rules

1) if A is a start symbol then follow(A) has a dollar symbol ($)
2) If there is a grammar rule A derives $\alpha B \beta$. $A \to \alpha B \beta$ and $\beta \neq \varepsilon$ then follow of B is first of β
3) If there is a grammar rule A gives αB or $A \to \alpha B \beta$ and $\beta = \varepsilon$ then Follow(B) = Follow(A)

eq: Find follow of grammar

$S \to AbB$
$A \to a / c$
$B \to b$

Follow(S) = {$}.
Follow(A) = First(B) = {b}.
Follow(B) = Follow(S) = {$}.

eq:2 $S \to ABC$

$F(S) \to \{a, b\}$
$F(A) \to \{a, \varepsilon\}$
$F(B) \to \{b\}$
$F(C) \to \{c\}$

Follow(S) = {$}
Follow(A) = First(B) = {b}
Follow(B) = F(C) = {c}
Follow(C) = Follow(S) = {$}.

	FIRST	FOLLOW
S → ABC	{a,b}	{$}
A → a/ε	{a,ε}	{b}
B → b	{b}	{c}
C → c	{c}	{$}

Find first and follow of below grammar.

E → TE'
E' → +TE'/ε
T → FT'
T' → *FT'/ε
F → id / (E)

Soln

F(E) = {(, id, E}

Variables are, E, E', T, T', F

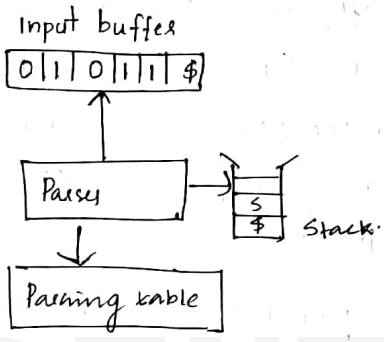
Terminals are : {+, *, (,)}

	First	Follow
E → TE'	{id, (}	{$,)}
E → +TE'/ε	{+, ε}	{$,)}
T → FT'	{id, (}	{*, +, $,)}
T' → *FT'/ε	{*, ε}	{+, $,)}
F → id / (E)	{id, (}	{*, +, $,)}

Follow(E') = {$,)}

Follow(T) = first(E') = {+}

Non - Recursive Predictive Parser.



Steps to construct predictive parser table for a grammar G all listed below

Step1: Eliminate left recursion in grammar G.

2: Perform left factor in grammar G

3: Find first & follow on the symbols in grammar G.

4: Construct the predictive class table.

5: Check if the given I/p string can be accepted by the parser.

Q Construct predictive parse table for the below grammar.

S -> a A B b
A -> c / ε
B -> d / ε

Soln: left recursion ~~of~~ A -> Aα / β.

1: No left recursion. ii Grammar is left ~~for~~ recursion free

2: No common prefixes no need of left factor.

3:

	First	Follow
S -> a A B b	{a}	{$}
A -> c / ε	{c, ε}	{d, b}
B -> d / ε	{d, ε}	{b}

4: Construct predictive class table.

	a	b	c	d	$
S	S->aABb				
A		A->ε	A->c	A->ε	
B		B->ε		B->d	

If first includes ε then ~~fo~~ consider follow.

St Consider an input string 'acbd' various steps in parsing is shown in below table

Input	Stack contents	moves
acdb$	S | $	derived using S->aABb
acdb$	aABb | $	Pop a
cdb$	ABb$	A->c

cdb$	cBb$	Pop c
db$	Bb$	B→d
db$	db$	Pop d
b$	b$	Pop b
$	$	String

Q Construct predictive parsing table (LL(1)) for below grammar S→(S)/ε

Soln: S→(S)/ε

Step 1: No left recursion

2. No common prefix so to already left factored.

3 First & follow

f(S) → first((S)) ⇓ first(ε)

{(} ∪ {ε} = {(, ε}

follow(S) = {$} ∪ first())

($) ∪ {)} = {$,)}

3

Step 4 Construct parsing table.

	(	)	$
S	S→(S)	S→ε	S→ε

Q Construct predictive table for below grammar.

Soln: E→TE'

E'→+TE'/ε

T→*FT'/ε

F→id/(E)

Soln: follow(E) = first(T') ∪ first(T')

= {*, ε}

follow(E) = first()) ∪ $

= {$,)}

~~follow(E)~~

follow(T) = first(E') ∪ first(E')

= (+, E) ∪ (

follow(E') = {$,)}

	First	Follow
E	{id, (}	{$,)}
E'	{+, ε}	{$,)}
T	{id, (}	{+, $,)}
T'	{*, ε}	{+, $,)}
F	{id, (}	{*, +, $,)}

	id	+	(	)	*	$
E	E->TE'		E->TE'			
E'		E->TE'		E'->ε		E'->ε
T	F->FT'		T->FT'			
T'		T'->ε		T'->ε	T->*FT'	T'->ε
F	F->id		F->(E)			

Q 15/2/18

S -> aB / ε

B -> bC / ε

C -> cS / ε

First(S) = a

Step1: Eliminate left recursion from this grammar

doesn't contain left recursion

Step2: perform left factoring on grammar G. Here already left factored grammar

Step3:

First(S) = {a, ε}

First(B) = {b, ε}

First(C) = {c, ε}

Follow(S) = $ ∪ follow(C) = {$}

follow(B) = follow(S) = {$}

follow(C) = follow(B) = {$}

	a	b	c	$
S	S->aB			S->ε
B		B->bC		B->ε
C			C->cS	C->ε

	First	follow
S	{a, ε}	{$}
B	{b, ε}	{$}
C	{c, ε}	{$}

To write parsing table

Column → Variables

rows → terminals

To fill column S we look at the first(S)

first(S) = {a, ε}

LL(1)

↓ look ahead

Predictive parsers: are free from backtracking

Top down parsers

Recursive.

→ Recursive Descent Parsing

Top down parsing technique

Parser construction for Recursive Descent Parsing

$E \rightarrow iE'$

$E' \rightarrow +iE'/\varepsilon$

Solution: Variables - $\{E, E'\}$

Terminals - $\{+, i\}$

To create RDP for a particular grammar.

Create a function for each and every variable.

Func^n

For Variable E

```
E()
{ if (l==i) // l is global look ahead
  { match(i)
} } E'()
```

For Variable E'

```
E'()
{
 if (l== +)
 { match(+)
   match(i)
 } E'();
else
  return();
}
```

DEFINITION OF MATCH

```
match (char t)
{
 if (l==t)
 l= getchar();
 else
}print
```

MAIN

Develop RD code for below grammar.

```
main()
{ E()
  if (l==$)
  printf("Successful Parsing");
else printf("error);
}
```

Q Develop Recursive Descent parsing for below gramm

$S \rightarrow aAb$

$S \rightarrow cd \mid c$

Soln:

Function for S

```
S()
{
  if (l == a)
  {match(a);
    A();
    match(b);
  }
}
```

Function for A

```
A()
{ if (l==c)
  {
    match(c)
    match(d)
  }
else if (l==c)
  {
    match(c);
  }
}
```