MODULE 2.

SYNTAX ANALYSIS:

Review of context free grammar, derivation trees and parse trees, ambiguity.

TOP DOWN PARSING.

Recursive decent parsing

Predictive parsing

LL(1) Grammars.

SYNTAX ANALYSIS.

creates the syntatic structure of the given source program.

syntatic structure - parse tree.

-checks whether the given source prgrm satisfies the rules implied by CFGs or not.

if satisfies creates parse tree else give error message.

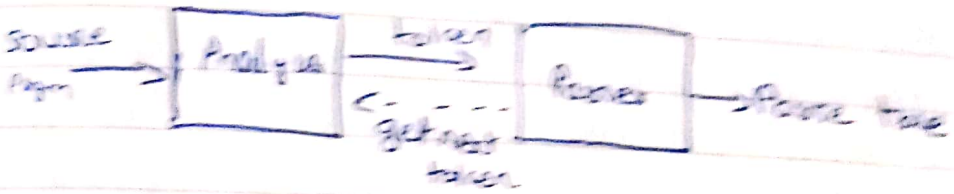
3 types.

Universal Parsers.

Top down : build trees from root to leaves

Bottom up : start from leaves and look up to root.

JULY 2014 16
Day 197 • 168 left WEDNESDAY



Representative Languages

Errors handling
- Lexical errors
- Syntatic errors
- Semantic errors - type mismatch
- Logical errors

Goals
- report the presence of error clearly and accurately
- recover from error quickly to detect subsequent errors
- add minimal overhead to the processing of correct programs

ERROR RECOVERING STRATEGIES
error msg

⇒ PANIC MODE RECOVERY.
on discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronizing tokens is found.
+ simplicity, no chance of further infinite loop
- no further checking.

17 JULY 2014

THURSDAY

Day 198 * 167 Left

Week 29

→ PHRASE LEVEL RECOVERY

replacing a prefix of remaining input by some string that allows the parser to continue.

→ difficulty is it has in coping with situations in which actual error occured before the point of detection.

→ ERROR PRODUCTIONS

Augment the grammar with production that generate the erroneous constructs.

→ GLOBAL CORRECTION

Choosing minimal sequence of changes to obtain a globally least-cost correction.

CONTEXT FREE GRAMMAR

$G = (N, T, P, S)$

Derivations: sequence of replacement of non terminal symbols.

Parse tree

graphical representation of a derivation

inner nodes - non terminals

leaves - terminals.

Ambiguity.

grammar that produces more than one parse tree for a sentence.

Left Recursion.

left recursive if it has a non terminal A such that $A \Rightarrow^{+} A\alpha$

top down parsing can't handle left recursion.

3 types

- direct : $A \to A\alpha$
- indirect : $A \to BC$ $B \to A$
- hidden : $A \to BA$, $B \to \varepsilon$.

To eliminate,

replace $A \to A\alpha_1 | A\alpha_2 | \cdots | A\alpha_m | \beta_1 | \beta_2 | \cdots | \beta_n$

with $A \to \beta_1 B | \beta_2 B | \cdots | \beta_n B$

$B \to \alpha_1 B | \alpha_2 B | \cdots | \alpha_m B | \varepsilon$.

eg: $A \to A\alpha | \beta$

$A \to \beta A'$

$A' \to \alpha A' | \varepsilon$

Begin to be now what you will be hereafter.

– William James

19 JULY 2014

SATURDAY

Day 200 • 165 Left

Edgar Degas' Birthday (France)

Week 29

eg: $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to id \mid (E)$

$E \to TE'$

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

$T \to FT'$

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

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

Eliminate indirect left Recursion.

for i from 1 to n do {

for j from 1 to i-1 do {

replace each production

$A_i \to A_j \gamma$

by

$A_i \to \alpha_1\gamma \mid \alpha_2\gamma \mid \cdots \mid \alpha_k\gamma$

$A_j \to \alpha_1 \mid \cdots \mid \alpha_k$.

}

eliminate immediate left recursions among A_i

}

SETF

eg: S → E
E → E+T
→ E, E → T
T → F T → E-T
T → F
F → E*F
F → id.

i = S. no left recursion.
i = E
S → E
E → TE'
E' → +TE' | ε
T → E-T
T → F
F → E*F
F → id.

i = T j = E [Ai → Aj α
T → E-T]

replace T → TE'-T

Then remove T left recursion

S → E
E → TE'
E' → +TE' | ε
T → TE'-T } T → FT'
T → F } T' → E'-TT' | ε
F → E*F
F → id

21 JULY 2014

MONDAY Day 202 • 163 Left

Week 30

$i=F \quad j=E$

$F \to E*F$ replace with $F \to iE'*F$

$S \to E$

$E \to iE'$

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

$i \to Fi'$

$i' \to E'-ii' \mid \varepsilon$

$F \to iE'*F$

$F \to id.$

$i=F \quad j=i$

replace $F \to iE'*F$ with $F \to Fi'E'*F$

$\left.\begin{array}{l} F \to Fi'E'*F \\ F \to id \end{array}\right\}$ $\left.\begin{array}{l} F \to id\,F' \\ F' \to i'E'*FF' \mid \varepsilon \end{array}\right\}$

Left factoring.

a grammar transformation useful for producing a grammar suitable for top down parser

$A \to \alpha\beta_1 \mid \alpha\beta_2$ ⇒ then $A \to \alpha A'$

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

A predictive parser insist that the grammar must be left factored.

Left

eg: $A \to abB \mid aB \mid cdg \mid cdeB \mid cdfB$.

$A \to aA' \mid cdg \mid cdeB \mid cdfB$

$A' \to bB \mid B$

$A \to aA' \mid cdA''$

$A' \to bB \mid B$

$A'' \to g \mid eB \mid fB$

} Left factored.

Non Context Free Language Constraints.

Some language constructions in Programming lang which are not context free.

→ semantic analyzer

eg: $\{wcw \mid w \text{ is } (a|b)^*\}$.

$\{a^n b^m c^n d^m \mid n \geq 1, m \geq 1\}$.

TOP DOWN PARSING

LL Parsers for top down parsing

23 JULY 2014

WEDNESDAY Day 204 • 161 Left

- Scanning input left to right.
  - viewed as a leftmost derivation.
  - creates nodes of parse tree in preorder (depth first)
- top down parsers which, mostly, requires backtracking is recursive decent parser.
- where no backtracking is required is called predictive parsing.

RECURSIVE DECENT PARSING

consist of set of procedures, one for each non terminal.

```
Void A() {
    chose an A production A -> X1 X2 ... Xn;
    for (i = 1 to k)
    {   if (Xi is non terminal)
            call Xi();
        else if (Xi is current input symbol a)
            advance input to next input symbol
        else
            error;
    } }
```

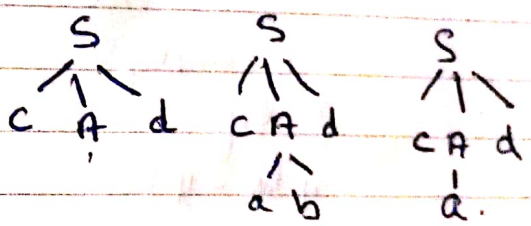
- may require backtracking (repeated scans over the input).
- cant be used for left recursive grammar.

The else part error is not ultimate failure, return to line 1 and try another A production
input error if there are no more A productions to try.

In order to try another A production, reset input pointer, store pointer in a local variable.

S → cAd
A → ab | a

e Input. cad




```
Procedure S ()
begin
  if ip = 'c'
     Advance();
  end if
  A();
  if ip = 'd'
     Advance();
     return true;
  endif
  end
```

25 JULY 2014

FRIDAY

Day 206 ★ 159 Left

Week 30

```
A -> ab|a
Procedure A()
begin
    iptr = ip
    if ip = 'a'
        Advance();
        if ip = 'b'
            Advance();
            return true;
        endif
    endif
    ip = iptr
        if ip = 'a'
            Advance();
            return true;
        endif
end
```

PREDICTIVE PARSERS

for each nonterminal, intial & final state.
draw transition diagrams
for $A \rightarrow X_1 X_2 \ldots X_n$

eg $E \to TE'$

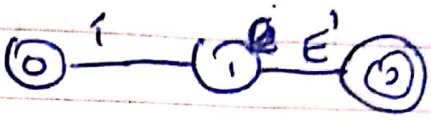
$E' \to +TE' \mid \varepsilon$

$T \to FT'$

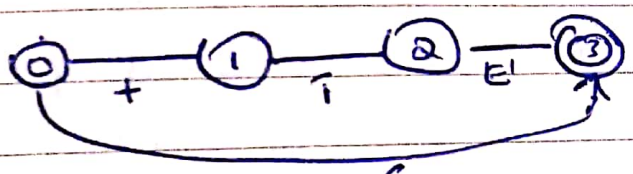
$T' \to *FT' \mid \varepsilon$

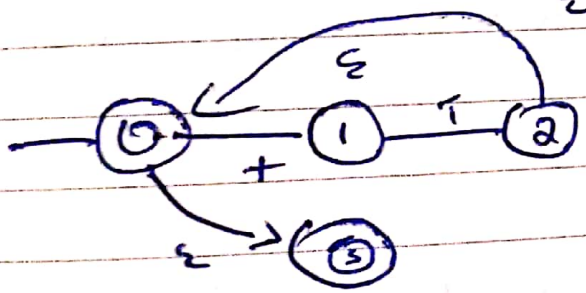
$F \to (E) \mid id.$

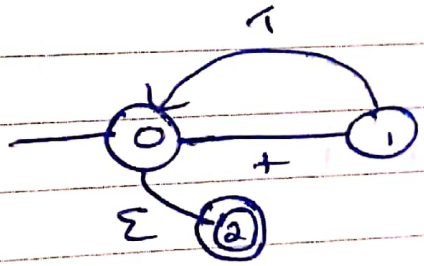
E :



E' :

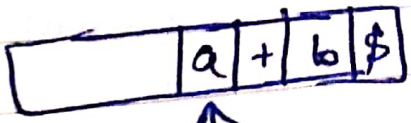




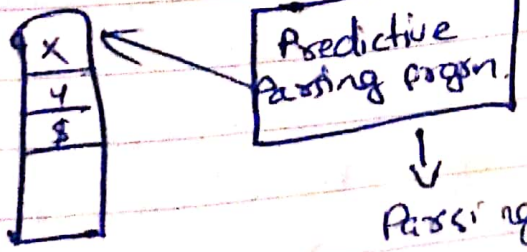


NON RECURSIVE PREDICTIVE PARSING.

maintaining a task.



input buffer



Stack

Initially parser contain w$ in input buffer.
Start symbol on top of stack above $.
3 possibilities.
-> X = a = $, halts ,succesfull
-> X = a ≠ $, pops X off stack and advance input ptr to next symbol
-> X = non terminal, pop x and push production from passing table.
M[X,a] = X → {UVW} pop X push UVW.

FIRST

-> If X is a terminal F(X) = {X}
-> If X → ε , F(X) = {ε}
-> X → $Y_1 Y_2 \ldots Y_k$.
Y_1 is terminal
$F(X) = \{Y_1\}$
$Y_1 \to \varepsilon$ and Y_2 terminal
$F(X) = \{Y_2\}$.
If all $Y_i = \varepsilon$, $F(X) = \varepsilon$
a in F(X) if a in $F(Y_i)$ and ε in all $F(Y_1)$ to $F(Y_{i-1})$

FOLLOW

→ If S is start symbol FOLLOW(S) = { $ }
→ If A → αBβ first of β will be in Follow(A)
→ If A → αB or A → αBβ and F(β) contain ε
then FOLLOW(B) = FOLLOW(A).

CONSTRUCTION OF PREDICTIVE PARSING TABLE.

For each A → α.

for each a in FIRST(α) add A → α to M[A, a].

If ε in FIRST(α), A → α M[A, b] for each b in FOLLOW(A)

If $ in FOLLOW(A) and ε in FIRST(α)
A → α to M[A, $]

Q.
E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → id | (E)

	FIRST	FOLLOW
E	id, (	$,)
E'	+, ε	$,)
T	id, (	+, $,)
T'	*, ε	*, +, $,)
F	id, (	+, $,)

29 JULY 2014

TUESDAY

Day 210 ★ 155 Left

JULY 2014

M	T	W	T	F	S	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

Ramadan Eid

Week 31

	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)		

E→TE'

FIRST(TE') = FIRST(T) = id, (

E'→+TE' E'→e.

Follow(E') = {$,)}

T→FT'

T'→*FT' T'→ε Follow(T') = +, $,)

LL(1) GRAMMAR.

L - scanning input from left to right.

L - producing a leftmost derivation.

1 - using one symbol of lookahead at each step to make parsing action decision

- no multiple defined entries
- leftmost parse tree is generated.

→ should be unambiguous.

→ operate in linear time & linear space.