COMPILER CONSTRUCTION

Page No.:
Date:
YOUVA

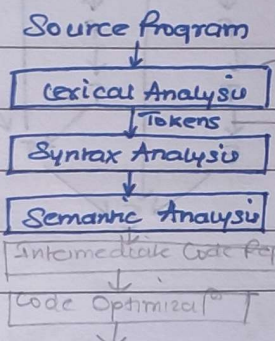
Thu 16/9/21

Compiler : tool to translate high level lang. to low level to be processed directly by the processor

Mon. 2(0) 20/9/21

- Basic Units
- Syntax / Rule

YACC tool
LEX tool



Wed. 3 22/9/21

o/p from semantic phase is very complex and has to be represented in a simpler form for the next phases, based on the processor

Gives the assembly code as an O/p

Tue. 1(P) 21/9/21

Write a Context free Grammar to identify expressions. a+10*C 10+5*6

$S \rightarrow idB$

$B \rightarrow +S \mid -S \mid *S \mid /S \mid +id \mid -id \mid *id \mid /id$

$S \rightarrow idB$
$\rightarrow 10+S$
$\rightarrow 10+5B$
$\rightarrow 10+5*S$
$\rightarrow 10+5*6B \rightarrow 10+5*6$

all operators will have same precedence and following BODMAS

OR

$S \rightarrow Exp$ $S \rightarrow SE \mid \lambda$ (in case there are multiple Expressions)

$Exp \rightarrow Exp+Exp \mid Exp-Exp \mid Exp*Exp \mid Exp/Exp \mid Num$

OR

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

Fri. 1(T) 24/9/21

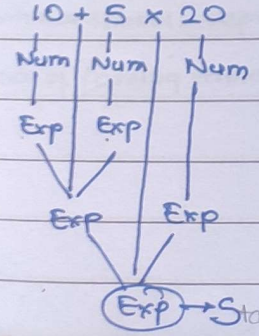
Define a Context free Grammar for assignment statement

$S \rightarrow id = E$

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

Mon. 4(0) 27/9/21

Bottom Up Parsing



yacc : .y or .yacc → generates c & header file (.c) (.h)

lex : filename.l or filename.lex

Page No.:
Date:

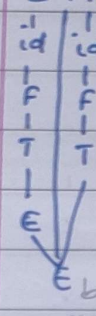
Expression with precedence

$E \rightarrow E+T \mid E-T \mid T$

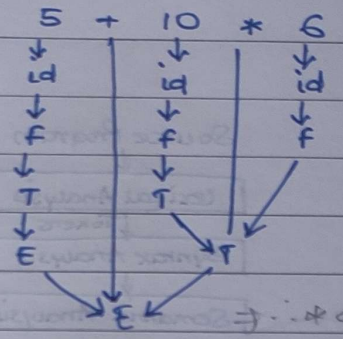
$T \rightarrow T*F \mid F$

$F \rightarrow id$

⇒ 5 + 10 * 6



E but now there is no option for E*. : .end ⇒ can't do + 1st



⇒ . : * done 1st to terminate successfully

Assignment Statement

$S \rightarrow id = E$ (id = Variable → not Num)

$E \rightarrow E+T \mid E-T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow Num \mid id$ (can be no. or variable)

if Statement

$S \rightarrow if\ (COND)\{EXP\}$ (terminal)

$COND \rightarrow id\ OP\ id$

$OP \rightarrow < \mid > \mid <= \mid >= \mid == \mid !=$

$EXP \rightarrow EXP+EXP \mid EXP*EXP \mid EXP-EXP \mid id$

if (i < 10) { temp * 10; }

for loop

$S \rightarrow for(INIT ; COND ; INC)\{STAT\}$ (STAT: statement)

$INIT \rightarrow id = EXP$

$COND \rightarrow id\ OP\ id$

$OP \rightarrow < \mid > \mid <= \mid >=$

$INC \rightarrow id++ \mid id--$

$STAT \rightarrow STAT\ EXP \mid E$ (STAT EXP: iteration)

$EXP \rightarrow EXP+EXP \mid EXP-EXP \mid EXP*EXP \mid id$

Mon. 2(P)
27/9/21

%token INTEGER → to state that INTEGER is a terminal variable

%{ ..... %} → Declaral"

%% ..... %% → Rules

Thu. 5
30/9/21

Predichive parsel Top Down Parsel

Predichive Parsing Algorithm :

1. Eliminate left Recursion
2. Algo. to Compute first
3. Algo. to Compute follow → terminal symbol immediately after the non terminal symbol. (follow of only nonterminal symbol)
4. Algo. to Construct Parsing Table
5. Parsing of I/P string Validation

1. Elimination of Left Recursion

→ if written in function we will keep calling the same function again & again & won't reach the terminal symbol resulting in Stack Overflow ∴ have to be eliminated

→ Activation Record per call

⇒ Top Down Parsel goes into ∞ loop

not required to eliminate for Bottom Up parsers → rather left or right recursion

$A \to A\alpha \mid \beta$ where α, β are strings & A is non terminal

⇒ After elimination:

$A \to \beta A'$

$A' \to \alpha A' \mid \varepsilon$

$S \to Ab \mid a \Rightarrow S \xrightarrow{*} Sa$

$A \to Ab \mid Sa$

$\therefore A \xrightarrow{*} A\alpha$

↓ Substitutes this in then eliminate for A

eg. $E \to E+E \mid id$

$\Rightarrow E \to idE'$

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

eg. $V \to V, id \mid id$

$\Rightarrow V \to idV'$

$V' \to ,idV' \mid \varepsilon$

declaral" stm

Mon. 6
4/10/21

Parsing Table for Top Down Parsing

Context free Grammar

$S \to DV;$

$D \to int \mid float \mid char$

$V \to id, V \mid id$

Parsing Table

- Consists of rows & columns
- Non terminal symbol is represented row wise
- Terminal symbol is represented column wise.
- A cell of parsing table is representing RHS string of a production.

	int	char	float	id	,	;	$
S	S→DV;	S→DV;	S→DV;				
D	D→int	D→char	D→float				
V				V→id,V / id			

Top Down parsing using parsing Table

eg.

Stack	Stack
$S	top down parser starts with pushing S
$;VD	
$;Vint	= ; pop & incr. ptr
$;V	
$;V,id	can choose either id,V or id but I/P buffer has more than 1 terminal, we chose id,V
$;V,	
$;V	
$;id	here we 'id' chose, same terminal left
$;	done automatically by parser acc. to algorithm
$	

I/P pointer

I/P string	only terminal symbols
int a,b; $	→ how to validate this given string.
int id,id; $	
int id,id; $	
id,id; $	→ ptr incremented
id,id; $	
,id; $	
id; $	
id; $	
; $	
$	

Hence the I/P string is verified

- If top of stack is terminal and not matches with the terminal in I/P buffer, pop the stack & increment the I/P buffer pointer by 1
- If top of stack is non terminal, check for the cell in the parsing table corresponding to that non terminal symbol and the terminal pointed by I/P buffer and reverse that cell string & push onto the stack and I/P buffer ptr remains same.

eg.

Stack	I/P String
$S	float f1,f2$ missing ;
$;VD	float id,id $
$;Vfloat	float id,id $
$;V	id,id $
$;V,id	id,id $
$;V	id $
$;id	id $
$;	$

Hence the string is invalid

left factoring removed so that incase we have → common string to come back if one of branch doesn't give proper result then we will have to process that & again ∴ left factoring removed to increase performance

Page No.: Date: YOUVA

Left Recursion & Left factoring

in top down parsing, not removing left factoring is fine ∵ it doesn't affect the final result only the performance but removing left recursion is necessary to get results.

- Left recursion is said to be present in given production if left hand side non terminal is the 1st symbol in RHS of string.

eg. $A \to A\alpha$

$B \to BaB \mid C$

- Left factoring is nothing but the replacement or substitution of common string present in RHS of productions.

eg. $A \to \alpha A1 \mid \alpha B1$

$\Rightarrow A \to \alpha A'$

$A' \to A1 \mid B1$

eg. $A \to abAB \mid abC$

$\Rightarrow A \to abA'$

$A' \to AB \mid C$

Thu. 7 개미지옥

2. Algorithm to Compute first → first terminal symbol in the prod searching for

Step 1: If 'a' is a terminal then 1st of 'a' is {a}

Means if $A \to a$, then first(A) = {a}, where 'a' is a terminal symbol.

Step 2: If there is a production of the form

$A \to A_1 A_2 A_3 \ldots A_n$, where $A_1, A_2, \ldots, A_n$ are terminal or non terminal symbol, then

first(A) = {first(A_1) ∪ first(A_2) ∪ ... ∪ first(A_n)} {∪ε if $A_1, A_2, \ldots, A_n$ derives to ε. ↳ not really required ∵ all derives ε, already will be included then}

Cases:-

- If A_1 is not deriving to ε, then first(A) = {First(A_1)} → not deriving to ε then stop at mar first else move to next one
- If A_1 derives to ε, then first(A) = {first(A_1) ∪ first(A_2)}
- If A_1 & A_2 derive to ε, then first(A) = {first(A_1) ∪ first(A_2) ∪ first(A_3)}

eg. $S \to DV;$

$D \to int \mid float \mid char$

$V \to idV'$

$V' \to ,idV' \mid \varepsilon$ considered as terminal symbol

⇒ first(S) = {int, float, char}

first(D) = {int, float, char} → from Step 1

first(V) = {id} V → multiple ∴ step 2 ⇒ id ↛ ε ∴ we stop at id

first(V') = {, , ε} for 1st prod^n V → ,idV' ⇒ , ↛ ε ∴ stop ; for 2nd prod^n V → ε ⇒ ε ↛ ε ∴ stop

like a union so no repetitions

ε ∴ stop if don't go to

first(S) = {firs(D)}

S → D ⇒ D → int ⇒ int ↛ ε ∴ stop at int

D ⇒ float ⇒ float ↛ ε ∴ stop at float

eg. $S \to DV;$

$D \to int \mid float \mid char \mid \varepsilon$

$V \to id\ V'$

$V' \to , id V' \mid \varepsilon$

$\Rightarrow first(S) = \{first(D), first(V)\} = \{int, float, char, \varepsilon, id\}$ (ε included: $D \to \varepsilon$)

$first(D) = \{int, float, char, \varepsilon\}$

$first(V) = \{id\}$

$first(V') = \{, , \varepsilon\}$

Mon. 8(0) 11/10/21

3. follow → search terminal symbol followed immediately by that variable

1. Add '$' to follow(S), where S is start symbol.
2. for a given production of the form $A \to \alpha B \beta$ (variable whose follow is to be found; string (terminal & non-terminal)), add first(β) except 'ε' to follow(B)
3. for a given prod^n of the form $A \to \alpha B$ or $A \to \alpha B \beta$, where $\beta \overset{*}{\Rightarrow} \varepsilon$, then copy follow(A) to follow(B)

eg. $S \to abBcDEg$

∴ follow(B) = {c}

follow(D) = {g}

follow(E) = {g}

follow(S) = {$}

$S \to \underset{\alpha}{\varepsilon} \underset{B}{D} \underset{\beta}{V};$ $\quad S \to \underset{\alpha}{D} \underset{\beta}{V};$

eg. $S \to DV;$

$D \to int \mid float \mid char$

$V \to V, id \mid id$ ⇒ $V \to idV' ; V' \to , idV' \mid \varepsilon$ (but this example is w/o removing left recursion but have to eliminate otherwise)

apply step 2 to all prod's first then step 3 at the end - it only copy operation (directly copy the follow)

follow(S) = {$}

follow(D) = {id}

follow(V) = {$, ;,)}

step 2 (first(V)) is first(B)

$V \to \underset{\alpha}{\varepsilon} \underset{B}{V}, \underset{\beta}{id}$ (step 2)

step 3 does not apply

eg. $S \to abBDE$

$D \to id \mid \varepsilon$

$E \to id \mid \varepsilon$