Page No.
Date / /

ex: | Position | | = | | initial | | + | | rate | | x | | 60 | Tokens = 7
T1 T2 T3 T4 T5 T6 T7

Generally, we make entry of only identifiers in the symbol table.

| int | a | ; | Syntactic error but no lexical error.
1 2 3

So total 3 tokens.

25/1/18

Symbol Table:

SNo.	Vname	Vtype
1	position	float
2	initial	float
3	rate	float

Token = < Token name , Token no >
abstract

= < id , 1 >

So, now statement becomes -

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

OR

<id, 1> <assign , => <id, 2> <add , +> <id, 3> <mul , x>
<constant , 60>

DOMS | Page No. | Date / /

Lexical Analyser : *LAX* = Lexical Analyser Tool.

* Lexical analysis phase converts source program into tokens with the help of Regular expression.

RE for identifier = $l(l+d)^*$

* Meaningful character stream is called lexemes and for each lexeme, we generate tokens.

Token = < token name , token no. >
↓ abstract name ↓ serial no.

* Removes whitespaces & comments from source program

Syntax Analysis Phase : *YACC* = Famous Syntax Analyser Tool (Yet another compiler compiler)

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

↓

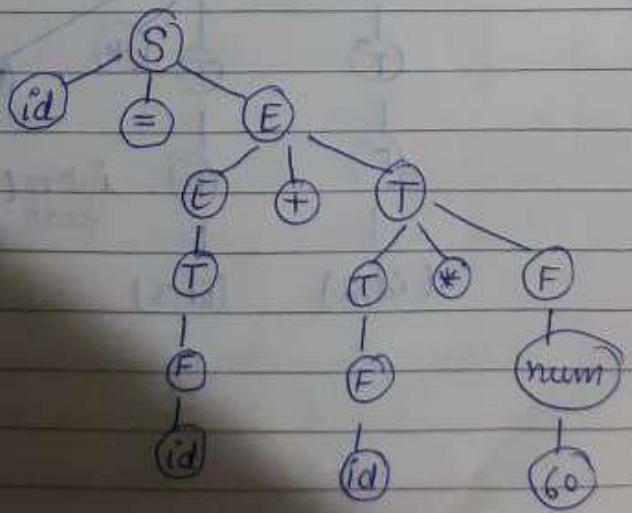
Syntax Analyser ← CFG ⇒

E → E+T / T
T → T*F / F
F → id / num
num → 0/.....9
S → id = E

↓

Parse Tree

New parse tree:



Makes parse tree with the help of CFG.

* It takes input as a stream of tokens with the help of the grammar.
* We generate parse tree for these tokens.
* We we generate 2 LMD or 2 RMD, then the grammar is ambiguous.

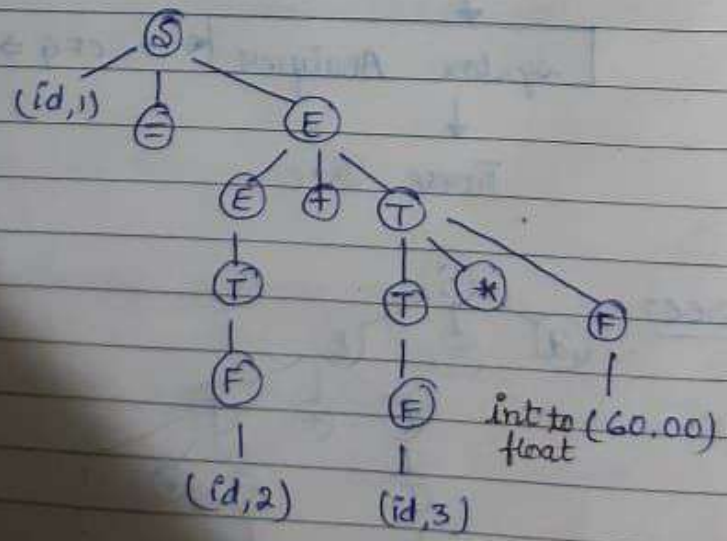
29 / Jan / 18

Semantic Analysis - Type checking
- Type conversion

Implicit : int a;
float b;
b = a;

Explicit: int a;
float b;
a = (int) b;

Semantically verified parse tree -



Date / /

Intermediate Code Generation: - 3 address Code

$t_1 = 60.00$
$t_2 = (id,3) * t_1$
$t_3 = (id,2) * t_2$
$(id,1) = t_3$.

: Immediate addressing mode @ : Indirect addressing mode

Code Optimisation: Optimising the code by putting constants directly using immediate addressing mode.

$t_2 = (id,3) * \#60.0$
$(id,1) = (id,2) + t_2$

We did this optimisation so that translator get less no of lines to translate.

Target code generation - Generating code.

```
LDF   R1 ,(id,3)          // R1 ← (id,3)
MULF  R2, R1 , #60.0      //R2 ← R1 * 60.0
LDF   R3 ,(id,2)          // R3 ←(id,2)
ADDF  R3, R3, R2          // R3 ← R3 +R2
STF   (id,1) , R3         // (id,1) ← R3
```

Theory

3. Syntactic Analysis :

eg: a ++ ++b

Tokens : a++ ++b 4 tokens { No lexical error

It has syntactic error.

Semantic Analysis - It uses the syntax tree and the information stored in the symbol table to check the source program for semantic consistency with the language definition.

It also gather type information & save it either in syntax tree or symbol table including the intermediate code representation.

Important part of semantic analysis is type checking

eg: 60 = a+b // Semantic error.

Intermediate Code Generation - We consider an intermediate form called 3 address code which consist of a sequence of assembly like instructions with 3 operands / instruction.

Each operand can act like a register.

Each 3 address assignment instruction has atmost one operand on the right hand side.

Code Optimisation - This phase attempts to improve intermediate code so that better target code is produced.

Code Generation - This phase takes intermediate representation of the source program as input and maps it into the target language.

Symbol Table - The essential function of a compiler is to record variable name used in the source program & collect information about various attributes of each name.

Q- If a > b then a = b; else b = 2; 14 tokens
Find out the no. of tokens in the given expression.

Q- int max (x, y).
{
int x, y;
// Calculate max
return (x > y ? x : y);
}

25 tokens

Q- int a[4][5]; 13 tokens
int *(*(a+4)(a+5));

Q- int @b; // Lexical error as @b is not any identifier

Q a+++b 4 tokens

Q- int a[4]; ⇒ int *(a+4); 8 tokens.
↓
*(a+4)