S. No.	Date	Title	Page No.	Teacher's Sign/ Remarks

MODULE - 1

REASONS FOR STUDYING THE CONCEPTS OF PROGRAMMING LANGUAGES

- To improve your ability to develop effective algorithms
- To improve the use of your existing programming language
- To allow a better choice of programming language
- To make it easier to learn a new language
- To make it easier to design a new language

PROGRAMMING DOMAINS:

1. Scientifical Applications: precision of values.
   C, FORTRAN (Formula TRANslation).
2. Business Applications:
   COBOL (Common business oriented language)
   for providing a detailed report.
   ROBMS [Relational], Excels.
3. Artificial Intelligence:
   LISP [List Processing), PROLOG (Programming logic]

4. System Programming:
NOTE: System software - OS, Compiler, loader, linker

eg: C language - Linux.

5 Special Purpose Languages
(General purpose system simulation - GPSS)

LANGUAGE EVALUATION CRITERIA

1. Readability
2. Writability
3. Reliability
4. Cost

Readability:
The ease by which we can read and understand the program.
1970 - Booch - Software lifecycle
Factors Influencing readability
- Overall simplicity
  eg: A programming language having lesser no. of features is much easier than one having large no. of features

WRITABILITY
The ease by which we can write the program.
- factors affecting readability also affects writability. because ~~all the~~ codes we while writing a code, we read the code as well.

Simplicity & Orthogonality
language with less features has more writability. because error correction would be easier
- Support for abstraction.
  To get an outline
  2 types of abstraction.
  1. Process abstraction
  2. Data abstraction

Process abstraction:
eg: Sort process occurs at different places. Instead of repeating, use function. The function invoking

statement to process abstraction.

Data abstraction

data is stored as binary numbers. in computers. but user need not know about that to do all that.

3. Expressibility

How easily we can express the statement

count ++ ✓

count = count + 1 ✓

- RELIABILITY.

Under all conditions, program should work correctly.

1) Type checking

It checks a) whether proper arguments are provided

b) whether data types is provided

Checking that whether each operator receives proper no of arguments of proper data type

It can be done at either at compile time or at runtime time.

Compile time type checking - C, C++

Run time type checking - Python

2) Exception handling

Run time errors are handled here.

- e.g.; division by 0.

3. Aliasing

Same memory location is provided with more than one names.

4. Readability and writability

If it is easy to read and write, less chance of making errors. reliability ↑

COST:

How much cost involved at diff stages

- Cost involved in training a programer

- Cost of writing a program
- Cost involved during compilation.
- Cost involved in executing the program
- Maintenance cost
- Cost involved due to poor reliability
- In worst jobs its too high

Ambiguous Grammar

If expression is having 2 or more left most or right most

A grammar that produces two or more left most derivation or ~~2 or more~~ right most derivation or parse trees for the same program statement.

<assign> → <id> = <expr>

<id> → A|B|C|D

<expr> → <expr> + <expr> |
<expr> * <expr> |
'('<expr>')' |
<id>

A = B + C * D

<assign> → <id> = <expr>

A = <expr>

A = <expr> + <expr>

A = <id> + <expr>

A = B + <expr>

A = B + <expr> * <expr>

A = B + <id> * <expr>

A = B + C * <expr>

A = B + C * <id>

A = B + C * D

⇒ A = <expr> +

A = <expr> * <expr>

~~A = <id> * <expr>~~

A = <expr> + <expr> * <expr>

A = <id> + <expr> * <expr>

A = B + <expr> * <expr>

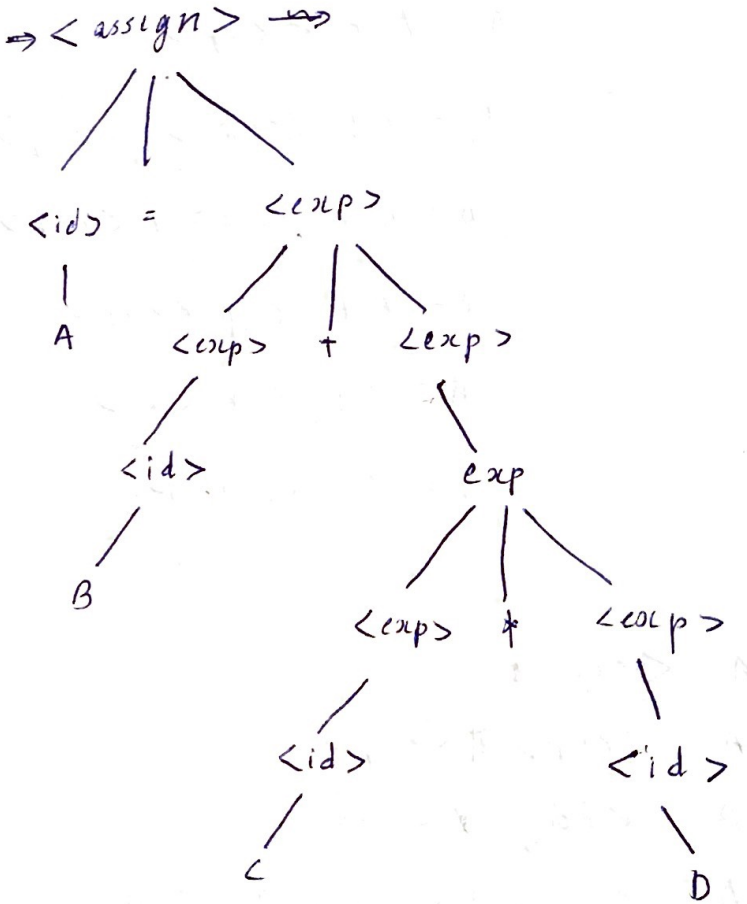
A = B + <id> * <exp>

A = B + C * <exp>

A = B + C * <id>

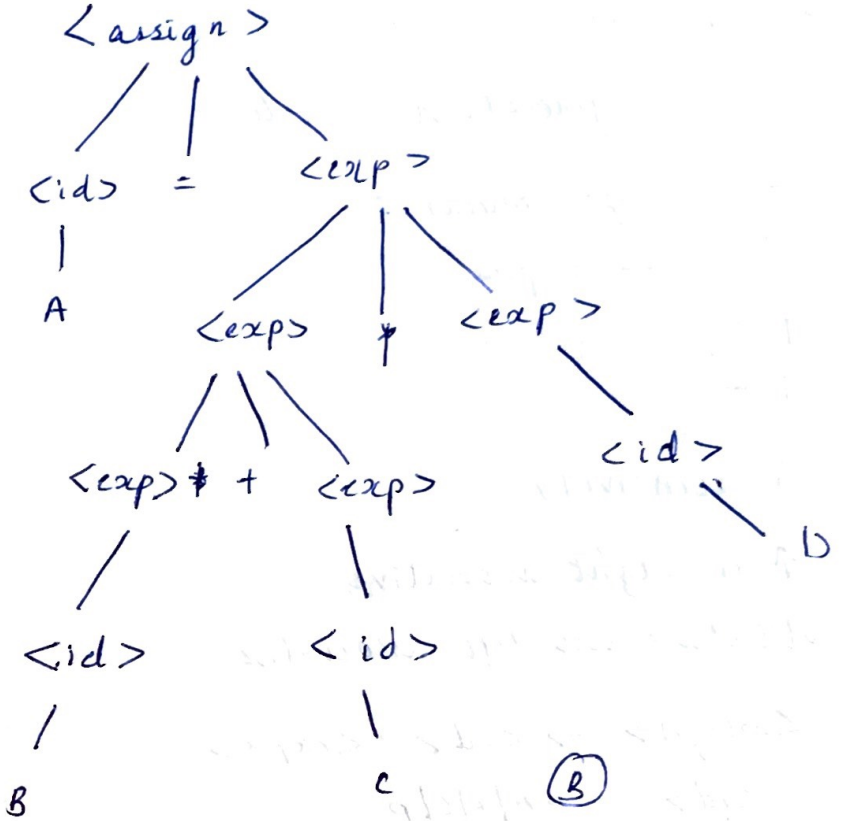
A = B + C * D

∴ It is ambiguous.



(a)

(A)



(B)

Tree valuation is from bottom to top

According to BODMAS concept, A is correct coz + come after * while evaluating from bottom to top.

DISAMBIGUATING RULES:

Unambiguous Grammars

① Operator precedence rule

- − unary minus
- ↑ Power exponent
- *, ÷
- + −

② Associativity

↑ is right associative

All other are left associative

<assign> → <id> = <expr>

<id> → A|B|C|D

<expr> → <expr> + <term> / <term>

<term> → <term> * <factor> / <factor>

<factor> → (<expr>) / <id>

Precedence is there.

Also left associative

A ↑ B ↑ C

<assign> → <id> = <expr>

<id> = <expr> + <term>

<id> = <expr> + <term> * <factor>

<id> = <exp> + term > × <id>

<id> = <exp> + dem > ↑ b

<id> = <exp> + <factor> ↑ D

<id> = <exp> + <id> * D

<id> = <exp> + C * D

<id> = <exp term> + C * D

<id> = <factor> + C * D

<id> = <id> + C * D

<id> = B + C * D

A = B + C * D

Ambiguous Grammar for Conditional Statements.

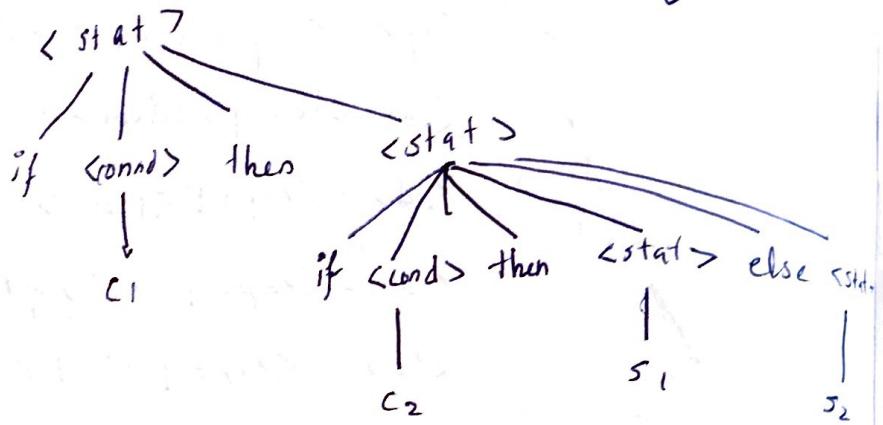
<stat> → if <cond> then <stat> /

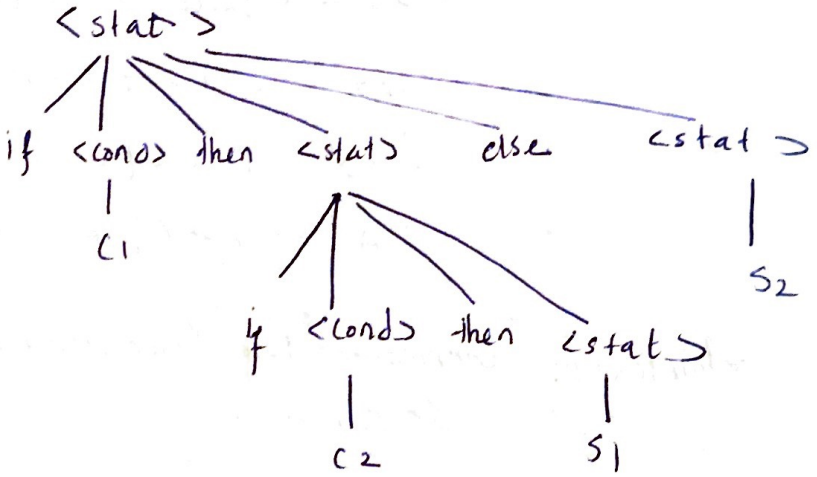
if <cond> then <stat> else <stat> /

S1 / S2

<cond> → $c_1 \mid c_2$

if c_1 then if c_2 then s_1 else s_2 ✓





⇒ Each else has to be matched with nearest if

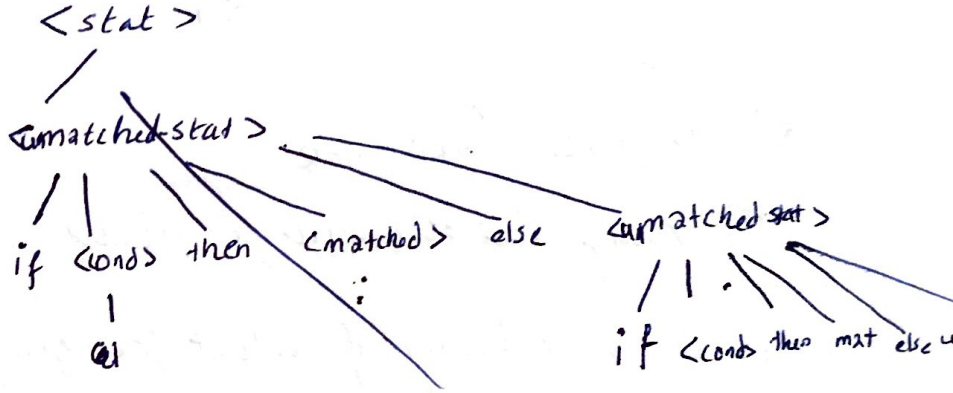
<stat> → <matched-stat> / <unmatched-stat>

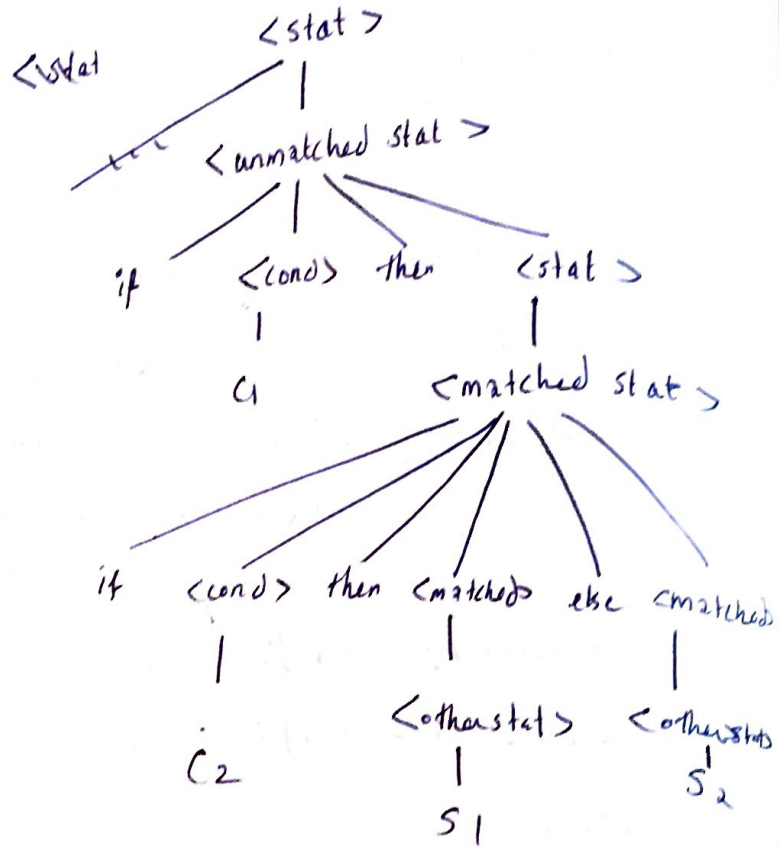
<matched-stat> → if <cond> then <matched-stat>
else <matched-stat> / <other-stat>

<unmatched-stat> → if <cond> then <stat> /
if <cond> then <matched-stat>
else <unmatched-stat>

<other-stat> → $s_1 \mid s_2$

<cond> → $c_1 \mid c_2$





ATTRIBUTE GRAMMAR.

To verify the semantic correctness

- It is an extension of context free grammar
- Used to specify semantic correctness of programming statements.

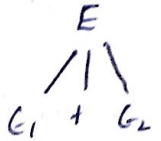
2 sections

→ Attribute computation function

→ Attribute predicate function.

Attribute: computed
- To compute att at node make use attributes of children.
- Synthesized attribute
- Inherited attribute

Syn: $E - E_1 + E_2$



Inherited attribute: making use of parent or sibling attribute

Synthesized attribute:

An attribute is said to be synthesized if its value at a parse tree node is determined by the attribute values at the child nodes.

Inherited attribute:

Inherited attributes are those whose

evaluated value at a node in parse tree is defined in terms of attributes of parent and/or siblings.

$X_0 \rightarrow X_1, X_2, X_3 \ldots X_n$

$S(X_0) \rightarrow f(A(X_1) \ldots A(X_n))$

$I(X_j) \rightarrow f(A(X_0) \ldots A(X_n))$ $j \rightarrow (1-n)$

$\rightarrow f(A(X_0) \ldots A(X_{j-1}))$

Attributes of X_j depends on X_j itself if X_j appears between X_0 and X_n.

Syntax rule:

<Proc-def> → Procedure <Proc-name[1]>

<Proc-body>

ed <Proc-name[2]>

Semantic rule <Proc name>[1].string

= <Proc-name>[2].string

Procedure add

end sum

Syntactically correct but semantically wrong

<assign> → <var> = <expr>

<expr> → <var> + <var> | <var>

<var> → A | B | C

Type check: Type compatibility

① Syntatic rule: <assign> → <var> = <expr>

Semantic rule: <expr>.expected type ←

<var>.actual type

2) Syntatic rule: <expr> → <var>[2] + <var>[3]

Semantic rule: <expr>.actual type ← if (<var>

[2].actual type = int)

and (<var>[3].actual type = int)

then int

else real

end if

Predicate: <expr>.actual type = <expr>.expected ty...

3) Syntax rule: <exp> ⟶ <var>

<expr>.actual type

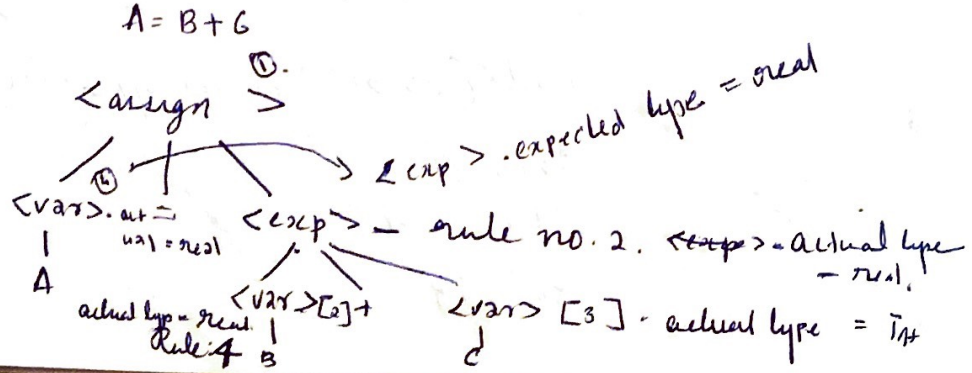
Semantic rule: <expr>.actual type ← <var>.
actual type

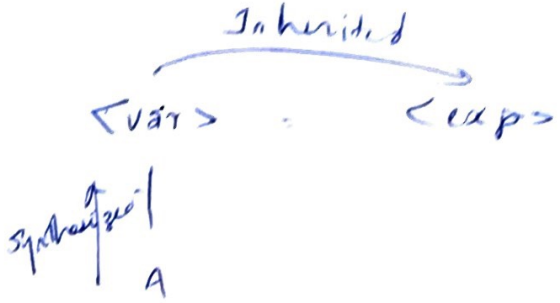
Predicate: <expr>.actual type = <exp>.expected
type

4) Syntax rule: <var> → A|B|C

Semantic rule: <var>.actual type ← lookup
(<var>.string)

A.B+C
int real





COMPUTING ATTRIBUTE VALUES

1. <var>.actual type ← lookup (A) Rule 4
2. <exp>.expected type ← <var>.actual type Rule
3. <var>[2].actual type ← lookup (B) Rule 4
4. <var>[3].actual type ← lookup (C) Rule
5. <exp>.actual type ← either int or real.
6. <exp>.expected type = <exp>.actual type is either true or false
Type combatible. Rule 2.

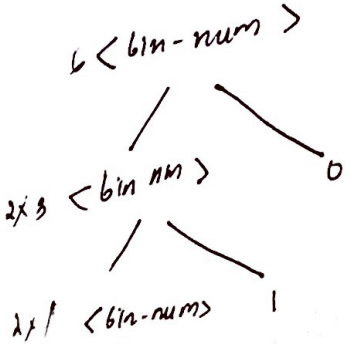
Rule 2

DENOTATIONAL SEMANTICS

Meaning of pgm statement is explained using a mathematical function.

<bin-num> → 0|1|<bin-num>0|<bin-num>1

110



M bin ['0'] = 0

M bin ('1') = 1

M bin (<bin-num> '0') = 2 * M bin (<bin-num>)

M bin (<bin-num> '1') = 2 * M bin (<bin-num>) + 1

binary can be converted to decimal

- Overall Simplicity

2. Feature Multiplicity
   - the same operation can be represented in different ways.
   eg: count = count + 1 ↓ readability
   count++ = 1
   count ++

3. Operator Overloading
   Same operator - different purposes
   eg: + can be used for int addition, float addition, string concatanation,
   ↓ readability.

4. Orthoganality:
   simplicity
   2 m/c architecture : IBM, VAX
   IBM: A Reg', memory cell
   AR Reg', Reg 2
   Reg 1 ← Content (Reg 1) + Content (memory cell)
   Reg 2 ← Content (Reg 1) + Content (Reg 2).

VAX:

ADDL operand 1, operand 2

Operand 2 ← Content(operand 1) + Content(operand 2)

4 combinations: Reg+Reg, Reg+mem, mem+Reg, mem+mem

More orthogonal VAX

5 CONTROL STATEMENTS

control statement ↓ readability

a) less no. of go to's.

b) should not be too far. - destination

c) As far as possible destination should precede the goto statement.

6 DATA TYPES & STRUCTURES

- Appropriate data types should be there to give better readability

Sum = 1 (1 or true)

- Structures ↑ readability.

7 Syntax Considerations

a) Identifier forms

Max no. of characters: 31 (in C)

b) Reserved words

- Key words.

Ada

{ if

} endif

Structure to programming

In C, keywords cannot be used as an identifier but in FORTRAN it can be used

Do } FORTRAN

REAL Do

readability ↓ if keyword is used as identifier.

c) FORM & MEANING

different meaning for a form ↓ readability

eg static function - local and global.

Describing Syntax & Semantics

→ Syntax

The form of a programming statement

→ Semantics

Meaning of a programming statement

Lexeme: Lowest level of a programming language

a = b + c * d

a, =, b, +, c, *, d all are lexemes

identifiers, punctuation marks, operators etc

Token: Lexemes come under certain category that category is called Token.

In above example =, +, * etc come under token operators

GRAMMAR:

4 categories - Type 0, Type 1, Type 2, Type 3

Classification done by Chomsky

Type 2, Type 3 commonly used

Type 2 → Context free grammar

Type 3 - Regular grammar

- Type 2: Identify syntactical errors
- Type 3: Identify whether the given token a valid one

eg: Int 5sum;

CONTEXT FREE GRAMMAR (CFG) or BACKUS NAUR FORM (BNF)

Set of rules that can be used for the verification of syntactic correctness of program statements

4 quantities:

1. Terminal 2. Non terminal 3. Start symbol

4. Production

1. TERMINAL:

Basic unit of a program statement that cannot be expanded further.

eg: ~~identifier~~ lexemes

2. Non-terminal
Denotes set of strings which can be further expanded.

3. Start symbol: One of the terminal is considered as start symbol.

4. Production Rule.

eg: <assign> → <id> = <expr> — Production
<id> → A|B|C|b — Production
<expr> → <id> + <expr> |
<id> * <expr> |
(<expr>) |
<id>

→ Assigning.

Terminal : =, A, B, C, D, +, *, (,)

Non terminal : <id>, <expr>, assign

Start symbol: assign

Production :

A = B * (C + b)

Start symbol : Assign

<assign> → <id> = <expr>

A = <expr>

A = <id> * <expr>

A = B * <expression>

A = B * (<expression>)

A = B * (<id> + <exp>)

A = B * (C + <exp>

A = B * (C + <id> + <exp>)

A = B * (C + D)

<Program> → Begin <stmt-list> end

<stmt-list> → <stmt> | <stmt>; <stmt-list>

<stmt> → <var> = <expr>

<var> → A|B|C

<exp> → <var> + <expr> | <var-<expr>

|<var>

begin A = B + C ; B = C end

<Program> → begin <stmt-list> end

<stmt> → begin <stmt> end <stmt-list>

begin <var> = <exp> end <stmt list>

begin <var> = <exp> <stmt> end

begin <var> = <exp> ; <var> = <exp> end

begin A = <exp> ; <var> = <exp> end

begin A = <var> + <exp> ; <var> = <exp>

begin A = B + <exp> ; end

begin A = B + <VAR> ; <var> = <exp> end

begin A = B + C ; <var> = <exp> end

begin A = B + C ; B = <exp> end

begin A = B + C ; B = <var> end

begin A = B + C ; B = C end .

Sequence of replacement : derivation

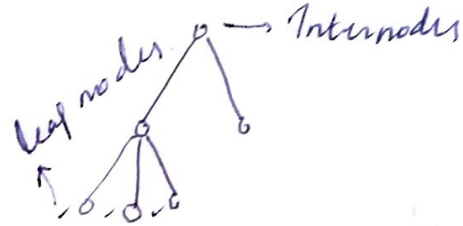
This is called left most derivation.

Each of step, non terminal is replaced by other from left in each step.

Similarly Rightmost

PARSE TREE

graphical representation of derivation.

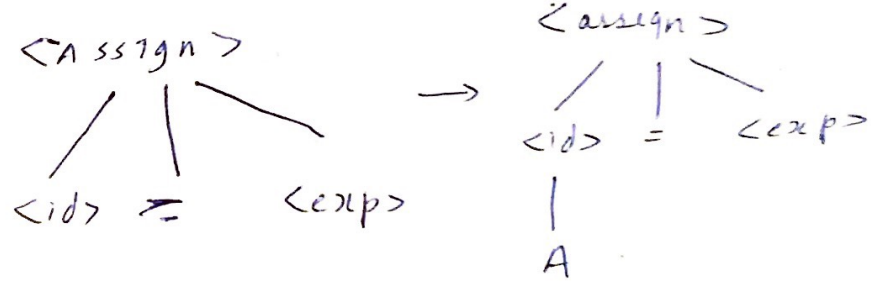


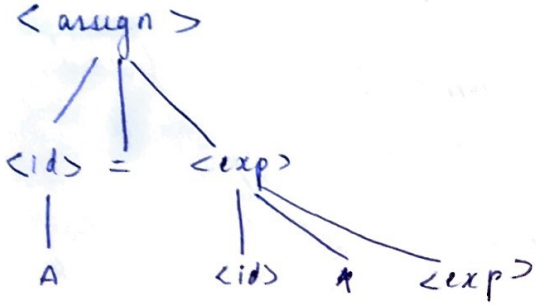
Corresponding to each derivation step, there is a parse tree.

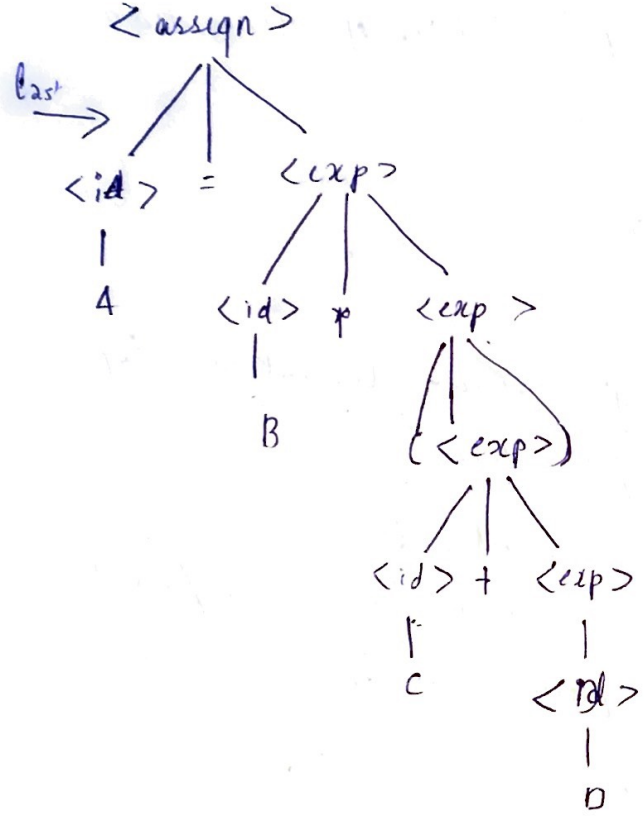
Internal nodes - Non terminal

Leaf nodes - Either terminal / non terminal

For final parse tree, leaf node - terminal




<assign>
<id> = <exp>
A
<id> * <exp>


<assign>
last
<id> = <exp>
A
<id> * <exp>
B
(<exp>)
<id> + <exp>
C
<id>
D