Syntax and Semantics

Motivation

Concise yet understandable description of a programming language is difficult

↓ ~~But it's essential to the~~

Language's Success

For instances, ALGOL 60 and ALGOL 68 were using concise formal descriptions.

↳ But descriptions were not easily understandable (Partly because each used a new notation).

↓

As a result, the languages have sufferd!

↳ not widely acceptable

There are other languages that have simple but informal and imprecise definition.

↳ Many closely similar dialects

Issues in describing a language:

Diversity of the people who must understand the description.

- Initial evaluators
- Implementors
- Users

] feedback cycles required for the design process.

Success of such feedback cycles heavily depends on the clarity of the description.

Programming language implementors must understand how the

expressions
statements and
program units

of a language are formed.

Also, intended effect when those expressions, statements and program units are executed.

Finally, users must be able to determine the ways to encode software solutions using reference manual.

LANGUAGE AND SYNTAX

Languages, be it natural language or artificial language, are a set of —

strings of characters from some alphabets.

Syntax rules of a language specify the strings of characters that are in the language.

↳ from alphabets

For example, in natural languages, such as in English, there are a large and complex collection of rules.

↳ For specifying the syntax of its sentences

In comparison,

Even the most complex programming language is not syntactically very simple.

SYNTAX

In natural language, the term Syntax means

- arrangement of words or phrases to produce well-formed sentences.

For instance, consider the below –

1. NSU is the ranked-one private university in Bangladesh
2. NSU in Bangladesh Private university is the ranked-one

In the above two examples, both have same number of words and they have same lexical information.

But we prefer structure in A.

Another example :

1. Students finished CSE425 happily
2. Students happily finished CSE425
3. Happily, students finished CSE425

By rearranging "happily", we obtain three alternative options.

↳ Three different syntax

Syntax in programming languages

For a programing language, Syntax refers to a set of roles and grammars

→ Generally, these set of rules are sufficient to generate valid codes

So, we can say that —

Syntax refers to specific ordering of words of specific word sets.

→ Different for different programming languages.

For instance,

"While" statement in C follows the following grammar rules :

<while-statement> ::= while (< >) <statement>;

|||

while (condition)
{
    statement;
}

} Syntax

A few definitions: needed to describe Syntax

Lexemes:

Basic element of a language

In natural languages such as in English

- Refers to a word with distinctive meaning
- Example: Man, student, Food

In programming language:

- Lexeme means a string of characters that are the lowest-level syntactic unit.
- In a practical programming language, there are infinite number of lexemes. (possibly)
- Examples:
  - Numeric literals
  - Operators

Given that a source code is:

$$x = 3;$$

then, lexemes are

'x', '=', '3', ';'

Tokens

Lexemes are generally partitioned into groups —

names of variables
methods
classes etc.

are all formed a group.

Token [Each lexeme group is represented by a name.

So, token is the category of lexemes.

- Tokens of a programming language are finite in numbers.
- Sometimes, token can have single lexeme
  - + arithmetic operator is the only lexeme for the corresponding token.
- So, Tokens have collective meaning.

EXAMPLE : TOKENS in C

There are Six different types of Tokens in C.

① • Keywords : can't be used as variable or constant — break, for, while, do if, else, long, int,

② • Identifiers:

Generally, sequence of numerical digits. It also includes sequence of alphabets.

name of variables, functions, etc. are examples.

In C,

User-defined names consisting of C-standard character set are

IDENTIFIERS

↳ identifies particular element in a program.

Some of the C-standard:

- No keyword can be an identifier
- First character must be an alphabet/undersore
- Each identifier should be unique
- No whitespace character.
- Name should be meaningful.

Discussion on

RESERVED WORDS/KEYWORDS VS. IDENTIFIERS

- This is often confusing to differentiate between these two.

Generally,

If Reserved words/key words are named, the same name cannot be used as Identifier.
↳ character string

So, for instance, the following Identifier name cannot be used in "C"

double if

↳ as if is a reserved word.

Predefined Identifiers:

↳ Given an initial meaning for all programs in the language

example: print, round

↳ in ython

With these sets of information —

How do we differentiate between

doif

do if

This ambiguity is eliminated using—

Principle of longest substring

↳ At each point, the longest possible string of non-blank characters is considered as a single token

↓ suggest

in do if, there are two reserved words—

do

if

and it is not an Identifier

CONCEPTS OF FREE-FORMAT LANGUAGE

CONSTANTS (3)

Constants don't change during the execution of the program.

Constants are of different types —

- Integer constants

  10, 15, 25

- Could be Octal / Hexadecimal

- Character constant
  - enclosed by single quote

    'A', 'B', 'C'

- String constant
  - sequence of characters enclosed using double quotes.

    "Hello", "World"

Real Constants

- Constants that contain decimal/ fraction value.
- For instance, PI = 3.14;

```
#include <stdio.h>
int main ( )
{
  const double PI = 3.14;
  printf ("%f", PI);
  PI++;
```

The code generates errors, as constant cannot be incremented.

VARIABLE (4)

- Gives name and allocate necessary memory space.
- May vary data during the execution of the program

- Variable generally stores value.

↳ it can change its value during execution; that is, value update is possible

In C, variable names must follow certain rules —

Valid names	Invalid names
age, _age	1age, a$ge

STRING (5)

- Sequence of characters.

represented by double quotes.

"Hello", "world"

Operators (6)

- Acts as connectors and decides type of operations to be carried out.
- Perform basic operations, comparison, manupulation etc.

Example of Lexemes and Tokenization

Let us consider the below statement —

```
index = 2 * count + 17;
```

For the above statement, the lexemes and tokens are as follows:

Lexemes	Tokens
index	identifier
=	equal_sign
2	int_literal
*	mult_op
count	identifier
+	plus_op
17	int_literal
;	semicolon

Java example

JAVA Tokens

- There are about six tokens in Java

1. Identifiers
   Names chosen by programmers
2. Keywords
   Names that are already present in the programming language

3. Separators
   - also, known as punctuators
   - Generally, punctuators

4. Operators
   - symbols that operate on argument

5. Literals:
   - Numeric → int and double
   - Logical → boolean
   - Textual → Char and string
   - Reference → Null

6. Comments.
   - Line, Block

* whitespace helps to decide tokens
  - end of a token
  - start of another token.

In many older languages, Identifiers length is fixed — That is, identifiers have maximum length.

In contrast,

Most modern-languages allow arbitrarily long name of identifiers.

↓ However

Often, the first six or eight characters are guaranteed to be significant.

Problem arises for variable-length Token such as an Identifier

→ How to decide the end ?

→ necessary to distinguish between identifiers and reserve words.

do if
doif

X12 → could be single identifier

X 12
↓ ↓ → constant
identifier
→ blank space

C-example: Lexemes and Tokens

Consider the "while" statement below:

```
while (x >= s)
    x = x- ;
```

Lexeme	Tokens
while	WHILE
(	LPAREN
x	IDENTIFIER
>=	COMPARISON
s	IDENTIFIER
)	RPAREN
x	IDENTIFIER
=	ASSIGNMENT
-	ARITHMETIC
6	INTEGER
;	SEMICOLON

Task of a Lexical analyzer is to create pairs of lexemes and tokens.

Consider a Java assignment statement

<assign> → <var> = <expression>

L.H.S — arrow — R.H.S

Here,

Text to the left of arrow is LHS

Text to the right of arrow is RHS

Represents abstraction being defined

example: <assign>

<var>

consists of

- mix of Tokens
- lexemes
- references to other abstractions

Altogether,

This definition is called rule or production

The above rule specifies

abstraction <assign>

↓ defined as

abstraction <var> —followed by→ lexeme = —followed by→

an instance of abstraction <expression>

Example:

sum = subtotal1 + subtotal2

var | lexeme | expression

FORMAL METHODS OF DESCRIBING SYNTAX

Human brain is biolgically programmed to learn language, so language faculty is innate.

↓ in addition

Mind works during the learning of a language.

According to Chomsky,

Transformational Generative grammar

Humans can learn a language for the linguistic faculty with which a human child is born with.

This is the innate part of human

↓

Universal grammar (UG)

And that the use of language for an adult is mostly a mental exercise

Precisely,

Noam Chomsky provided four classes of grammars.

↳ also known as generative devices

Two of those grammar classes —

a) Context-free

b) Regular

are useful for describing the syntax of programming language.

- Syntax of programming can be described, by context-free grammars. ↳ However, with minor exceptions

  CFG

- Shortly after Chomsky's work

  - John Backus, in one of his seminal paper, introduced

    a new formal notation for specifying the programming language syntax.

  - The new notation was later modified slightly by Peter Naur

So,

John Backus's new syntax notation

↓ modified by

Peter Naur

⇓

Backus-Naur Form (BNF)

↳ It is the natural notation for describing syntax.

↳ BNF is nearly identical to Chomsky's Context-free grammar

A few Fundamentals

Metalanguage :

A language that is used to describe another language.

For instance,

BNF is a metalanguage for programming language

- BNF uses abstraction for syntactic structures.

Terminal vs. non-terminal symbols

- Lexemes and Tokens of any rule are Terminal symbols
- Abstractions in BNF description are non-terminals.

BNF description:
- also, known as grammar
- Also, known as grammar
- It is a collection of rules

- Non-terminal symbols can have <u>two or more distinct definitions.</u>
  - Two or more syntactic form

- ✓ Multiple definitions can be written as a single rule.
  - Alternative definitions are separated by the symbol | (OR)
    - logical OR

Metasymbols: usefulness

* Sometimes formatting is not available
  - Text
  - handwritten text

Metasymbol is more useful

As arrow (used for "defined as") is not always available, it is often replaced by

metasymbol ::=

↳ it is also pure text

For instance,

sentence → noun-phrase verb-phase

↓ metasymbol

<sentence> ::= <noun-phrase> <verb-phrase>

Context-free grammar CFG

When all the syntax rules apply regardless of the symbols before or after the rules

↓

Context-Free

Let's consider the below example — [1]

① sentence ⇒ non-terminal noun-phrase non-terminal verb-phrase

② noun-phrase ⇒ article noun

③ article ⇒ a | the

④ noun ⇒ boy | girl | cat | dog

⑤ verb-phase ⇒ verb noun-phrase

⑥ verb ⇒ sees | pets | bites

Here, the terminal symbols :

'a', 'the', 'boy', 'girl', 'sees', 'pets', 'bites'

[1] Notes from Prof. Terry R. Payne University of Liverpool

Why is it context-free?

- Non-terminals appear singly on the Left Hand Side (LHS) of productions
- Each non-terminal can be replaced by any right hand choice.

No matter where the non-terminal appears.

Essence of context-sesitivity:

In above example, we have two verbs —
i) sees ii) pets

So, according to the grammar defined, the verbs — sees and pets — will make sense in the sentence if they are used with the subject girl.

→ Could be considered as an instance of context sensitivity.

※ Context-sensitive grammar can be written adding the context-sensitive string to the LHS of the grammar rules.

↓

Context-sensitivity is also treated as syntactic issue

Example : Context sensitivity

Consider the case obtained in rule 4

⇒ the girl verb-phrase rule 4

↳ First letter should be capitalized.

↓

Context-sensitivity

We can define a context-sensitive rule as:

beginning article → The | A

The First rule is defined as:

sentence → beginning noun-phrase verb-phrase

↳ article noun

Finally, the derivation looks as follows:

sentence → begining noun-phrase verb-phrase

→ begining article noun verb-phrase

→ The noun verb-phrase] Context sensitive rule.

→ ⋮

Any sentence that matches the productions as listed in ① to ⑥ is a valid syntax.

a girl sees a boy
a girl sees a girl
a girl sees the dog

← All are valid syntax.

a boy bites the dog
the dog pets the girl
a dog pets the boy
⋮

W To eliminate unwanted sentences without imposing context sensitive grammar, we specify semantic rules —

"a boy may not bite a dog"

Context-free grammars and BNFs

① sentence → noun-phrase verb-phase .
② noun-phrase → article noun
③ article → a | The
④ noun → girl | dog
⑤ verb-phase → verb noun-phrase
⑥ verb → sees | pets

From the above rule, we obtain a number of alternative options:

A girl sees the dog
A girl sees A dog
The girl sees a dog
The dog pets the girl
... and so on

} Productions

Any sentence that matches the above productions is VALID

In general, any sentence which is derived using production rule is said to be syntactically correct

Derivation from BNF

sentence ⇒ noun-phrase verb-phrase rule 1

⇒ article noun verb-phrase rule 2

⇒ the noun verb-phrase rule 3

⇒ the girl verb-phrase rule 4

⇒ the girl verb noun-phrase rule 5

⇒ the girl sees noun-phrase rule 6

⇒ the girl sees article noun rule 2

⇒ the girl sees a noun rule 3

⇒ the girl sees a dog. rule 4

W Here,

Seven Terminals

Six non-terminals

Six Productions

As we see, there are finite number of sentences that could be made out of the grammar written above.

↓ However,

languages defined by CFG are not finite

Grammars and Derivations

A grammar is a generative device for defining language.

Sentences of any languages are generated through a sequence of applications of the rules.

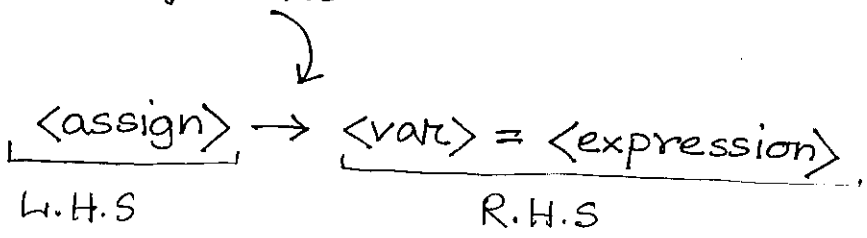
---

non-terminal
↳ Abstractions in a BNF description

begins with a special non-terminal of the grammar.
start-symbol

terminals
↳ Lexemes and tokens of the rules are called terminals

BNF description ≡ Grammar
↳ Collection of rules



Left-hand side (LHS): Abstraction being defined

Right-hand side (RHS): Text to the right of the arrow. Consists of
- tokens
- lexemes
- references to other abstractions

LHS and RHS, altogether, makes a rule. It is also known as production.

Consider a Grammar

<program> → begin <stmt_list> end

<stmt_list> → <stmt> | <stmt> ; <stmt_list>

<stmt> → <var> = <expression>

<var> → A | B | C

<expression> → <var> + <var> | <var> − <var>

Language described by above grammar

- has only one statement form
- Program consists of the special word
  begin
  ↓ followed by
  list of statements separated by semicolons
  ↓ followed by
  end

A Derivation :

<program> → begin <stmt_list> end
→ begin <stmt> ; <stmt_List>
→ begin <var> = <expression> ; <stmt_List> end
→ begin = <expression> ; <stmt_list> end
→ begin A = <var> + <var> ; <stmt_list> end
→ begin A = B + <var> ; <stmt_list> end
→ begin A = B + C ; <stmt> end
→ begin A = B + C ; <var> = <expression> end

→ begin A = B + c; B = <expression> end

→ begin A = B + c; B = <var> end

→ begin A = B + c; B = c end ..... (1)

All the derivations, such as this one, begins with a start symbol — in the above case, it is

<program>

How did we derive the eqn (1) ?

We replaced nonterminals one by one

→ Direction/ways to replace creates two alternative derivation types.

(i) Leftmost derivations

(ii) Rightmost derivations

In the previous derivation,

→ replaced nonterminal is always the leftmost nonterminal.

→ Derivations that use this order of replacement are called leftmost derivations

→ Derivations continue until the sentential form contains no nonterminals.

☐ Derivation may be rightmost or in an order that is neither rightmost nor leftmost

BNF example — if statement (Java)

<if-stmt> → if (<logic_expr>) <stmt>

<if-stmt> → if (<logic-expr>) <stmt>
else <stmt>

As we see, The if statement could be written in two different forms.

The two forms of if statement can be written using logical OR

<if-stmt> → if (<logic-expr>) <stmt> | OR

if (<logic_expr>) <Stmt>
else <stmt>

Summary: BNF

metasymbols:

::= defined as (also, → is used)

| OR

< > angle brackets surround non-terminal symbols

separates alternatives

<addop> ::= + | -

Another Example:

Consider a grammar for Simple Assignment statements

Grammar

<assign> → <id> = <expr>

<id> → A | B | C

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

Consider the statement

A = B * (A + C)

<assign> → <id> = <expr>
→ A = <expr>
→ A = <id> * <expr>
→ A = B * (<expr>)
→ A = B * (<id> + <expr>)
→ A = B * (A + <expr>)
→ A = B * (A + <id>)
→ A = B * (A + C)

Additional example:

Construct $A = A * (B + (C * A))$ using the grammar defined previously.

<assign> → <id> = <expr>

→ A = <id> * <expr>

→ A = A * (<expr>)

→ A = A * (<id> + <expr>)

→ A = A * (B + (<expr>))

→ A = A * (B + (<id> * <expr>))

→ A = A * (B + (C * <id>))

→ A = A * (B + (C * A))

PARSE TREE

A few points to be noted—

* Syntax establishes structure of a sentence or a program

  ↳ It does not say about the meaning

* Insted, meaning of a sentence or a program is dealt by the SEMANTICS CONCEPTS

For instance, in English senfences,

* Two parcts — subject and predicate-

  ↓

  Determine the meaning of a sentence.

Subject
actor
↓
Noun-phrase

predicate
action done by the actor
↓
Verb-phrase

So, a noun phrase is placed first in a sentence and is associated with a subject.

- Consider a grammar for expression —

  expr → expr + expr

  we expect → Adding two right-hand expression to obtain the left-hand expression.

So, we associate our desired semantics to this construct.

→ Defined as :

Syntax-directed semantics

- To make use of the syntactic structure of a program to determine its semantics

  we need a way to describe the syntax obtained by derivation

  ↓ A standard way.

  PARSE TREE

  → hierarchial syntactic structure

  → replacement of a derivation of syntax.

Example: Context-free grammar

Consider a simple integer arithmetic grammar

expr → expr + expr | exp * exp | (exp) | number

number → number digit | digit

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Observation:

- Recursive nature.
  - expression can be the sum or product of two expressions
    - each of which can be further sums of products.

Example of recursion:

- Defining an object in terms of itself is known as recursion
- Can be used to define sequence, fn^c sets.

Consider a sequence of powers of two:

$a_n = 2^n$ for $n = 0, 1, 2 \ldots \ldots$

$a_0 = 2^0 = 1$
$a_1 = 2^1 = 2$
$a_2 = 2^2 = 4$
$a_3 = 2^3 = 8$
⋮

∴ we see

$a_{n+1} = 2a_n$

Recursive Defⁿ

Recursion example :

$$f(0) = 3, \quad f(n+1) = 2f(n) + 3$$

Then, Fibonacci number : sequence of numbers

$$F_n = F_{n-1} + F_{n-2} \quad \text{Where, } F_1 = F_2 = 1$$

and conventionally, $F_0 = 0$

Coming back to the arithmetic example —

- The recursive process stops by selecting the alternative "number"
- Again, in this rule, 'number' is used to generate a sequence of digits —

For instance, number 234 can be constructed as follows:

number → number digit | digit

→ number digit

→ number digit digit

→ digit digit digit

→ 2 digit digit

→ 2 3 digit

→ 2 3 4

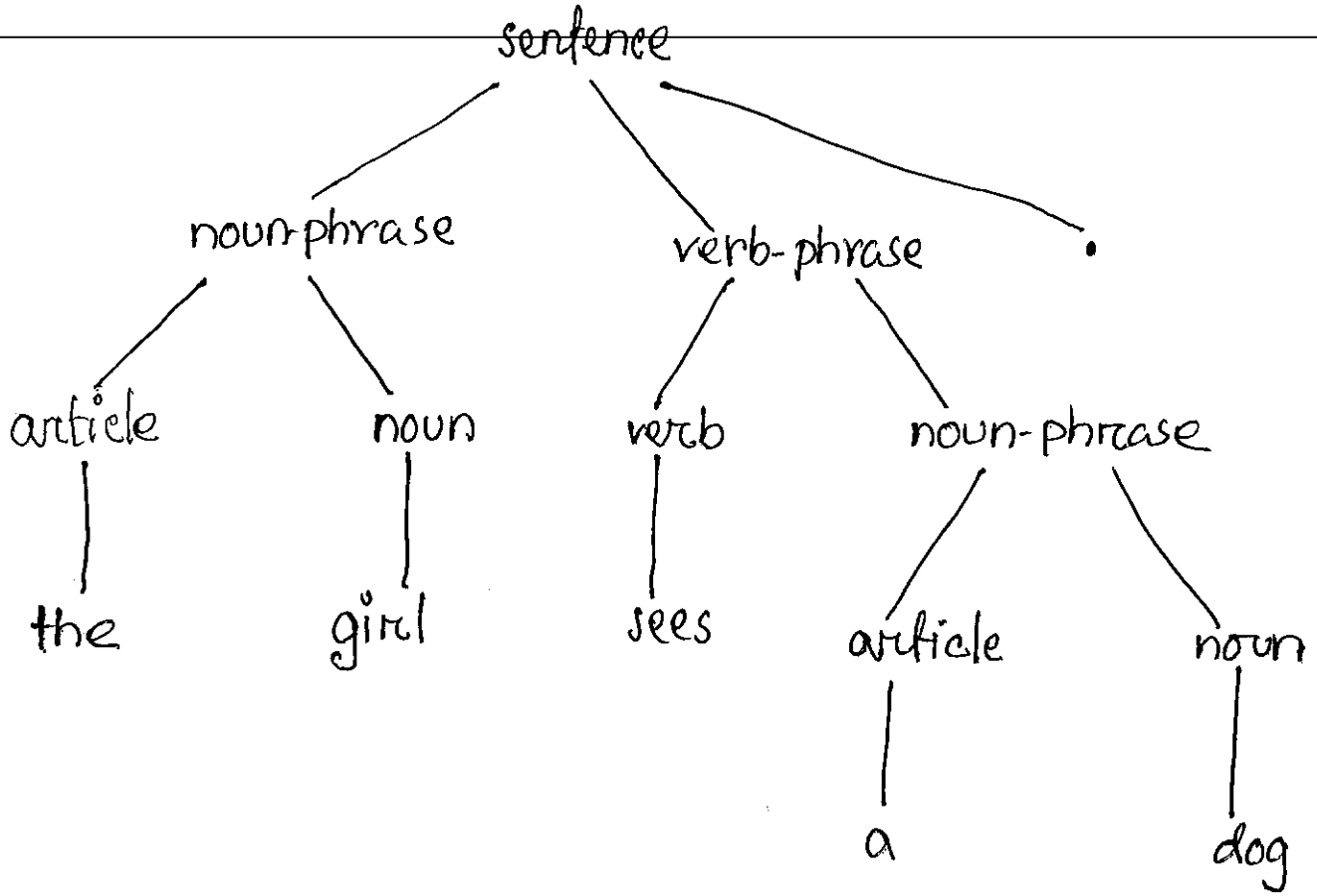
9. Try 2345

PARSE TREE

Example

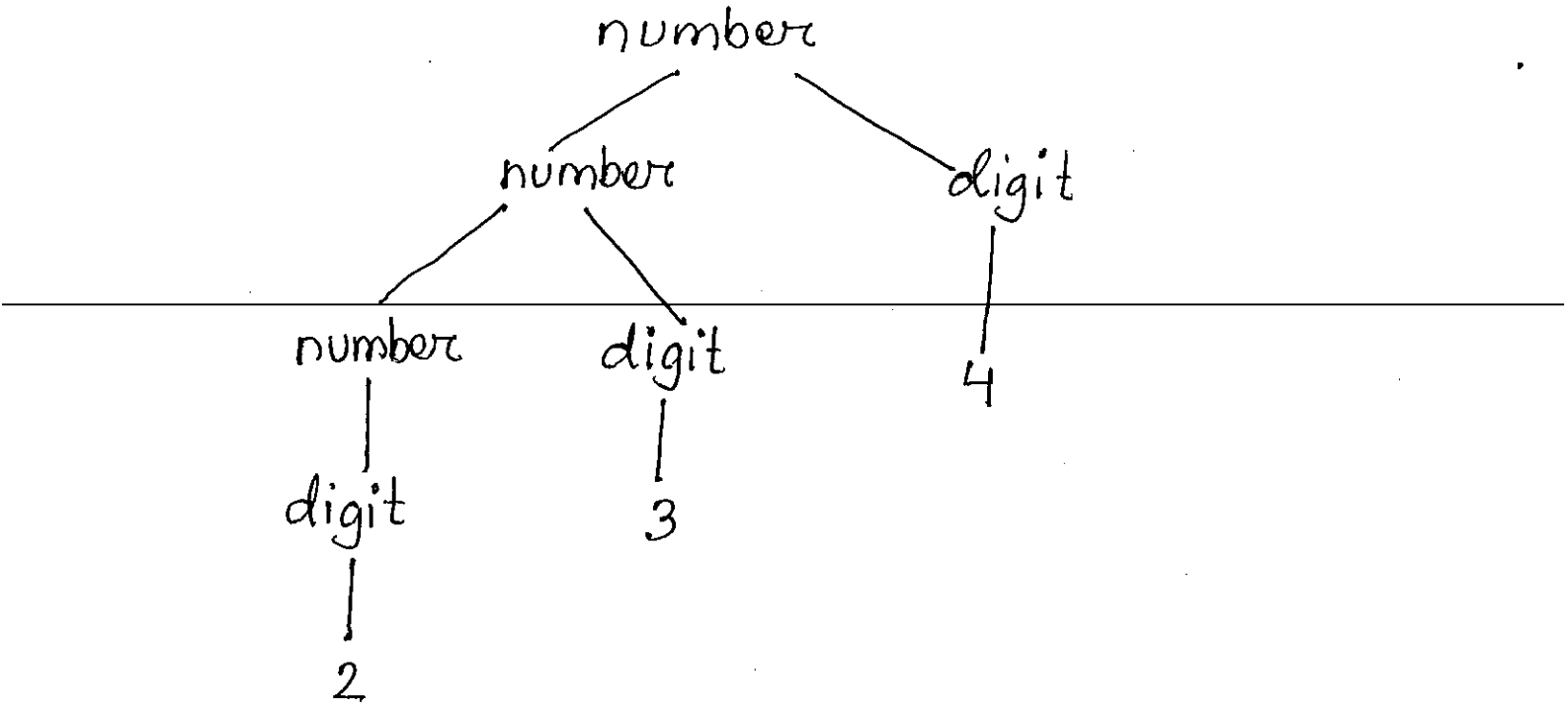
Let's consider the English sentence

the girl sees a dog.



- nonterminals are at interior nodes.
  - ↳ node with at least one child
- Terminals are at leaves.
  - ↳ nodes with no children.
- Every sub-tree of a parse tree describes one instance of abstraction.

Similarly, tree for the number 234 is as follows:



* Structure of a parse tree is completely specified by the grammar rules of the language and a derivation of a particular sequence of terminals.

→ Grammar rules specify the structure of each interior node.

| implies

steps in the derivation for 234 and number of internal nodes in the corresponding

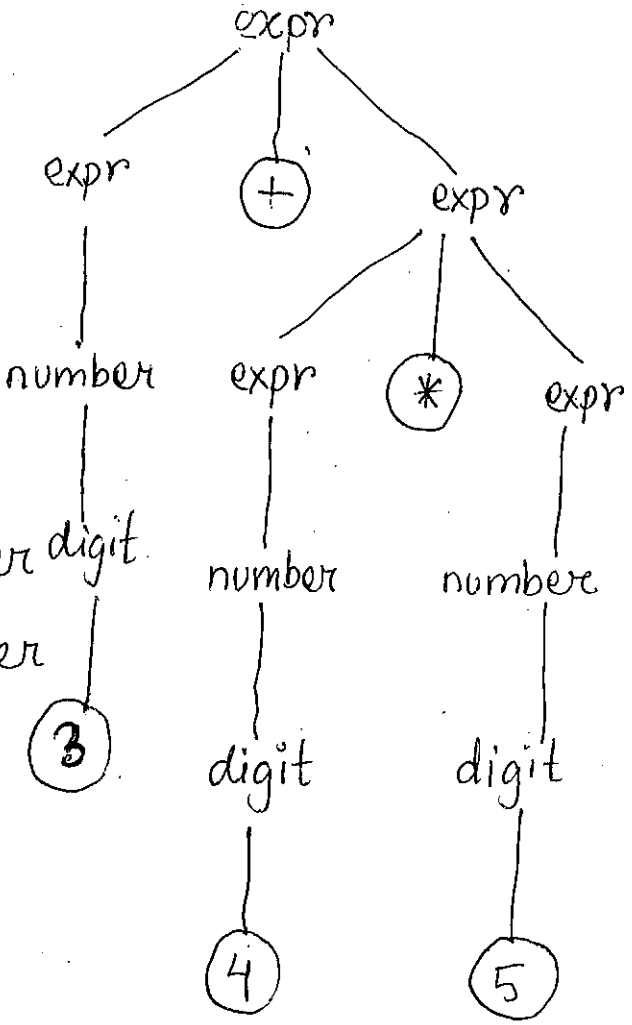
Parse Tree for 3+4*5

Given. grammar —

expr → expr + expr |
expr * expr |
(expr) |
number

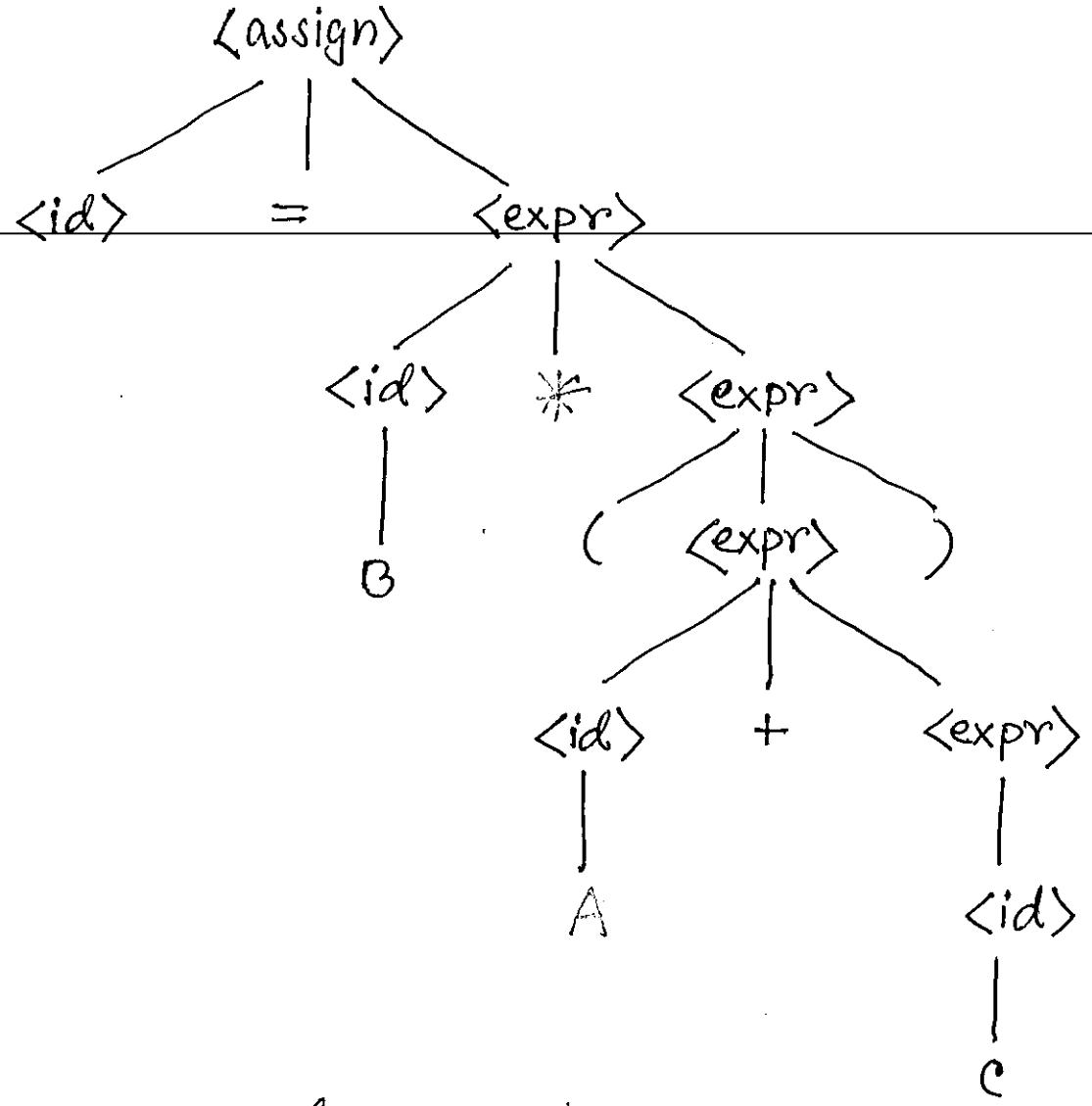
number → number digit |
digit

expr → expr * expr
→ expr + expr * expr
→ number + expr * expr
→ number + number * expr
→ number + number * number
→ digit + number * number
→ digit + digit * number
→ digit + digit * digit
→ 3 + digit * digit
→ 3 + 4 * digit
→ 3 + 4 * 5



Another Parse tree

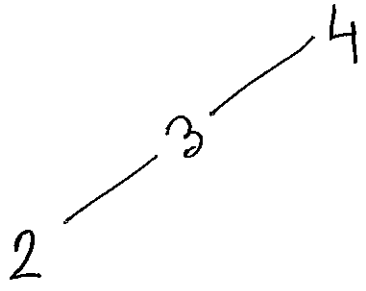
$$A = B * (A + C)$$



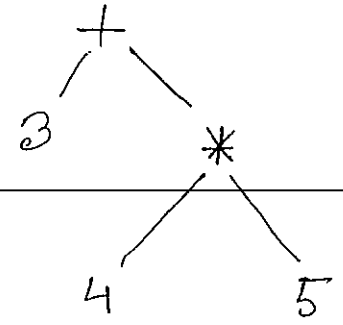
Condensed parse tree

↳ Abstract

- All terminals and non-terminals in a derivation are included in the parse tree
- However, not all terminals and nonterminals are necessary to determine the syntactic structure.



Structure of the expression 3+4*5 can be completely determined from the below tree



Condensed Parse Tree

All these trees are known as

Abstract Syntax Trees, or simply syntax trees.

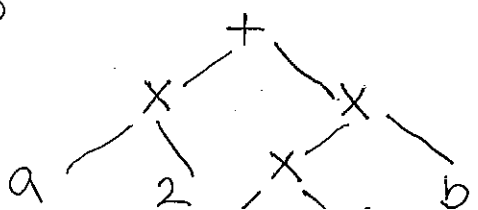
Abstract Syntax Trees (AST)

- Abstract the essential structure of the parse tree.
- Remove the terminals that are redundant
  - ↳ once the structure of the tree is already determined.

That is,

AST retains the essential structure of the parse tree but eliminates the extraneous node.

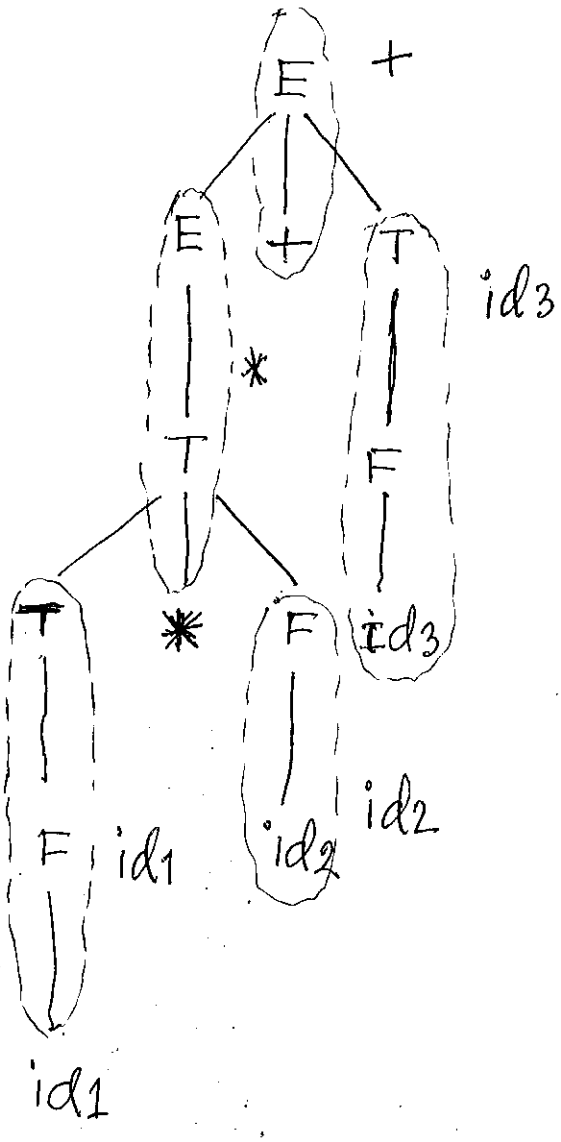
Example: a*2+a*2*b



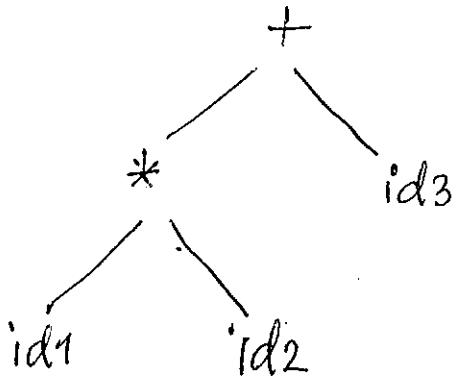
⊞ How to obtain AST from complete parse Tree?

※ Chain of single productions should collapse and the operators move up to the parent nodes.

Example: Consider a parse tree as below—



• Chain collapses



same { Condensed parse tree
Abstract syntax tree
Syntax tree

Discussion on AST

- Often confusing ——

  if-statement → expression statement statement

  this is not what is done in actual program.

- Important to the designer and translator writer

  - Because, it expresses language's essential structure.
  - Translator will often construct AST because it is concise.
    - ↳ Many compilers and interpreters use AST.

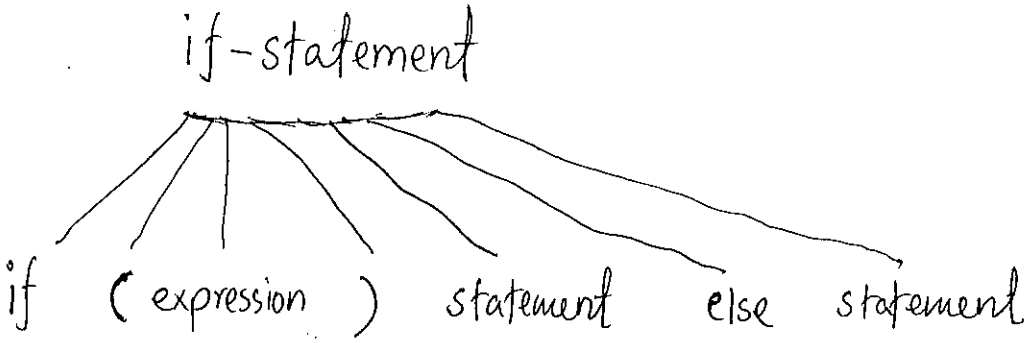
PARSE Tree Vs. Syntax Tree

• Parse tree: Replacement process in a derivation using graphical representation	Compact form of a parse tree
• Interior node represents a grammar rule.	Each interior node represents an operator.
Leaf node represents a terminal	Each leaf node is for an operand
• Provides every characteristic information from real syntax	Don't provide characteristic information

AST : Example if-statement

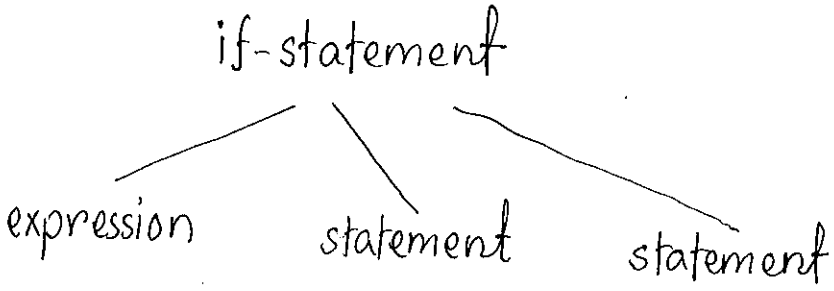
Consider the grammar rule :

if-statement → if (expression) statement else statement



* no need to record keywords or punctuation. so, else, if, '(,), are removed.

Condense parse tree
Abstract syntax tree



Rules for abstract syntax

- Can be written in BNF style

if-statement → expression statement statement

More examples on AST

Consider the grammar:

$E \rightarrow E + T \mid T$

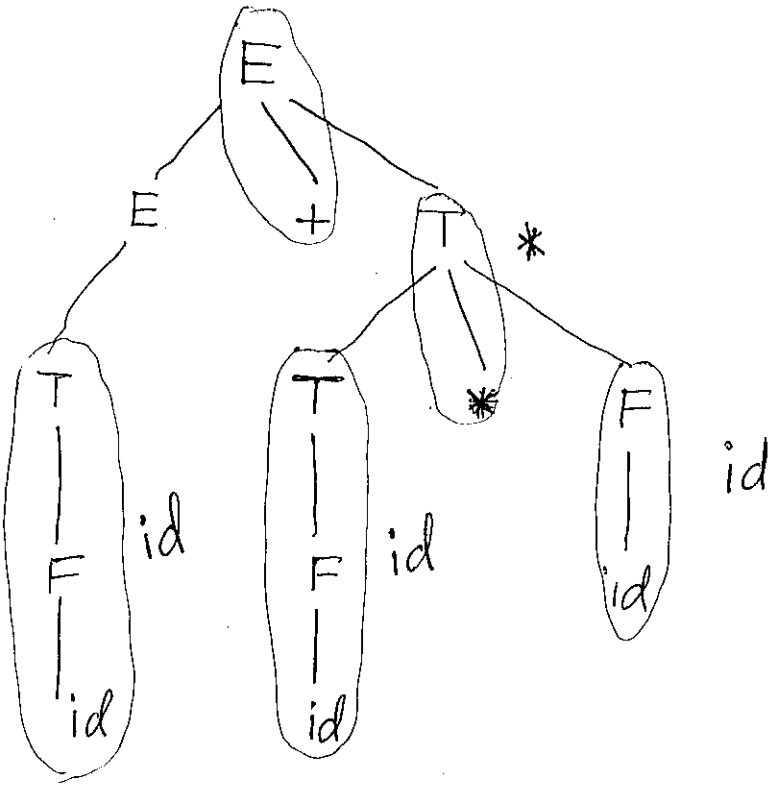
$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

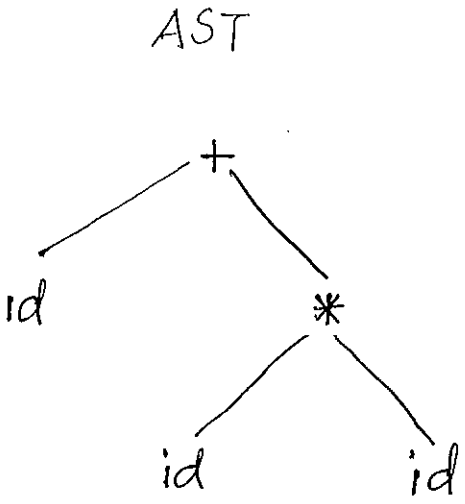
T: Tokens
F: Factors

Construct $id + id * id$

Parse Tree



AST



Ambiguity

Two different derivations can have same parse tree or syntax tree.

Grammar expr → expr + expr | expr * expr | (expr) |

number → number digit | digit

digit → 0|1|2|3|4|5|6|7|8|9

Construct 234

(1)

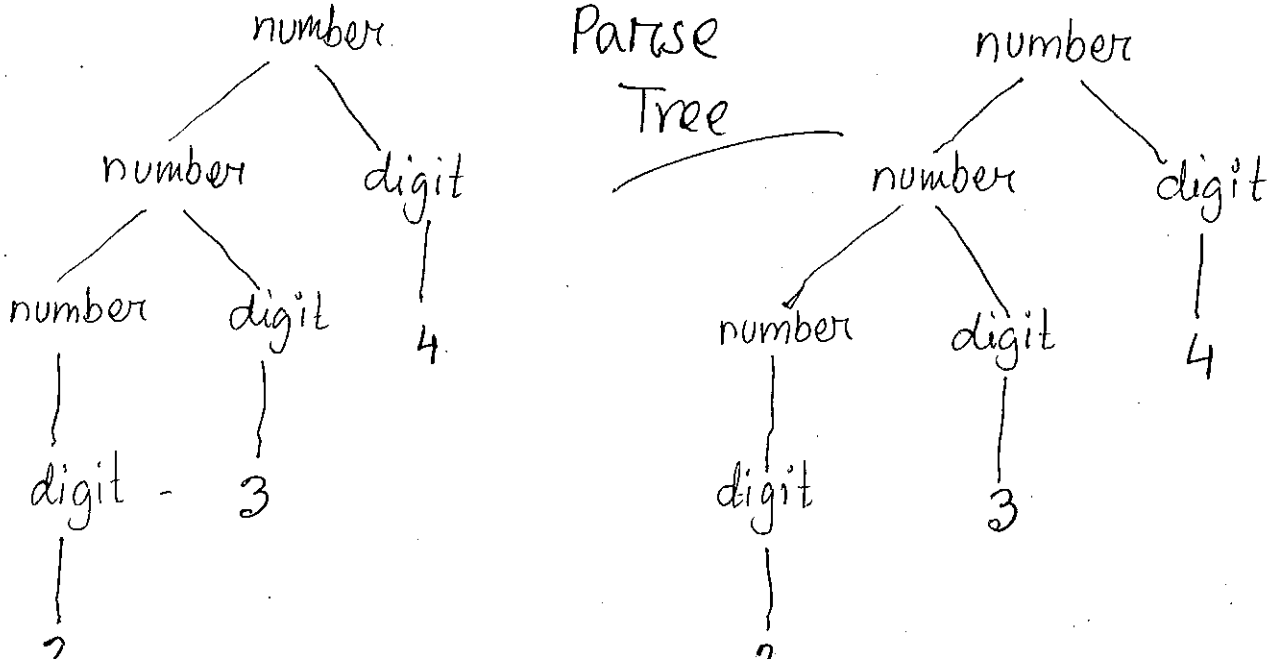
number → number digit
→ number digit digit
→ digit digit digit
→ 2 digit digit
→ 2 3 digit
→ 2 3 4

(2)

number → number digit
→ number 4
→ number digit 4
→ number 3 4
→ digit 3 4
→ 2 3 4

Here, digit nonterminal is replaced first.

Similar Parse Tree



☐ However, different derivations can also lead to different parse trees.

Consider 3+4*5 for the grammar given below:

expr → expr + expr | expr * expr | (expr) | number

number → number digit | digit

digit → 0|1|2|3|4|5|6|7|8|9

Derivation: (1)

expr → expr + expr

→ expr + expr * expr

↳ replacing the 2nd expr

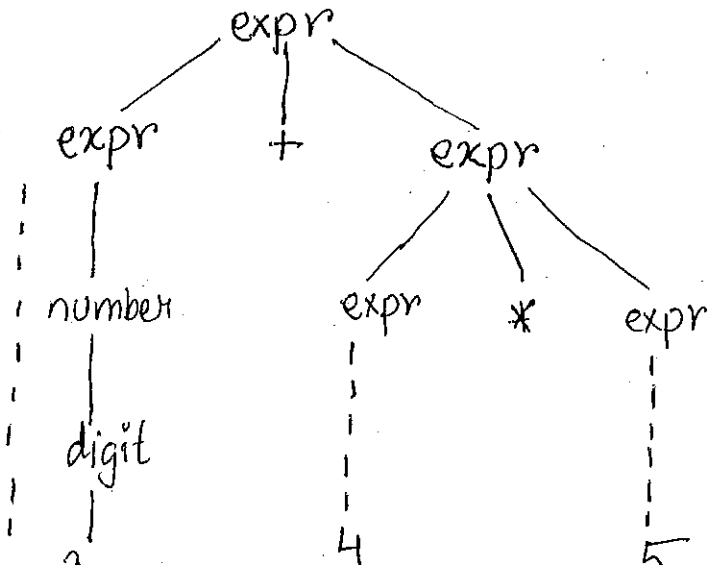
→ number + expr * expr

→ digit + number * expr

→ 3 + digit * number

→ 3 + 4 * digit

→ 3 + 4 * 5



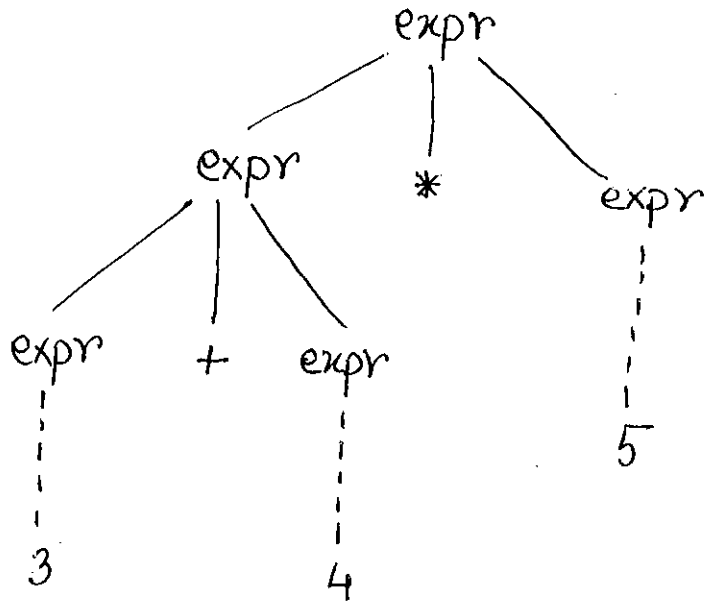
Derivation (2)

expr → expr * expr
(expr = 1st expr)

→ expr + expr * expr
(expr + expr replaced by)

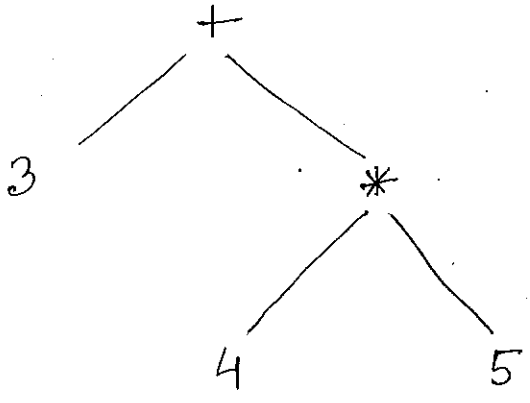
→ number + expr * expr

Parse Tree

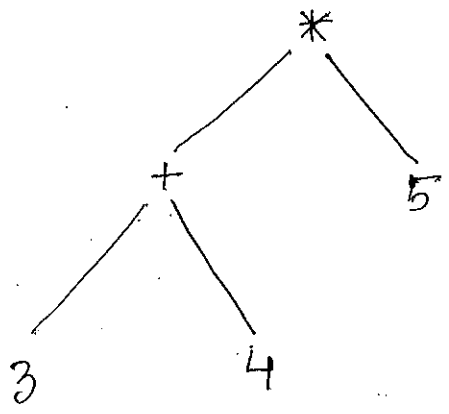


AST version :-

Derivation 1



Derivation 2



So, for a grammar if two distinct parse or AST are possible for a given string, the grammar is known as

AMBIGUOUS

Alternative realization —

Ambiguous grammar can also be explained using derivations

↓

Some derivations that are constructed in a special order do correspond to unique parse trees

↓

leftmost derivation

→ leftmost remaining non-terminal is singled out for replacement at each step.

Fact :

Each parse tree has a unique leftmost derivation.

↳ Can be used to differentiate between an ambiguous and unambiguous grammar.

So, a grammar for which two trees are possible for the same string / construct is known as

Ambiguous Grammar

↳ Ambiguity

- → Can be described in terms of derivations
- → many derivations may correspond to the same parse tree, but certain special derivations that follow special order correspond to unique parse trees.

Leftmost derivation → Leftmost remaining non-terminal is singled out for replacement at each step.

example 1

expr → expr + expr
→ number + expr
→ digit + expr
→ 3 + expr
→ 3 + expr * expr
→ 3 + number * expr
⋮

example 2

expr → expr * expr
→ expr + expr * expr
→ number + expr * expr
→ digit + expr * expr
⋮

So, for the construct of the desired string, if two different leftmost derivations exist, then the grammar must be ambiguous.

Leftmost 1

expr → expr + expr

→ number + expr

→ digit + expr

→ 3 + expr

→ 3 + expr * expr

→ 3 + number * expr

→ 3 + digit * expr

→ 3 + 4 * expr

→ 3 + 4 * number

→ 3 + 4 * digit

→ 3 + 4 * 5

Leftmost 2

expr → expr * expr

→ expr + expr * expr

→ number + expr * expr

→ digit + expr * expr

→ 3 + expr * expr

→ 3 + number * expr

→ 3 + digit * expr

→ 3 + 4 * expr

→ 3 + 4 * number

→ 3 + 4 * digit

→ 3 + 4 * 5

Obtained using leftmost derivation

They have different parse tree

but

for the same string

3+4*5

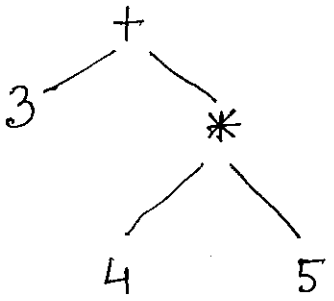
Problems with ambiguous grammar

Since an ambiguous grammar doesn't clearly express any structure, it presents difficulties.

↓ to make it useful

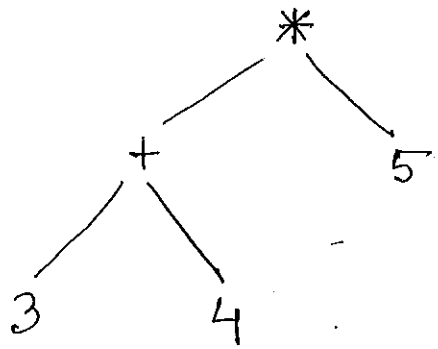
Remove Ambiguity or,
we need Disambiguating rule
to determine the structure meant.

If semantics is considered—



multiply 4 and 5, and then add with 3.

$20+3=23$



Add 3 and 4, and then multiply with 5

$7*5=35$

operations are applied in different orders

Resulted semantics are quite different

Solution

- Usual precedence of operators
- Precedence over addition

How do we obtain it ?

We could —

* State a disambiguating rule separately from the grammar

* Revise the grammar

Write new grammar rule (known as term) that establishes a precedence cascade.

↳ it forces matching of '*' at a lower point

So, expr → expr + expr | term

term → term * term | (expr) | number

new grammar rule

'*' is at a lower point.

However, The ambiguity problem is not completely solved.

For instance, the associativity problem

3 + 4 + 5

(3+4) + 5 OR 3 + (4+5)

Why is it a problem ?

In case of addition, this does not make any difference

But, For subtraction, this can create problem —

For instance, $\overline{8-4-2}$

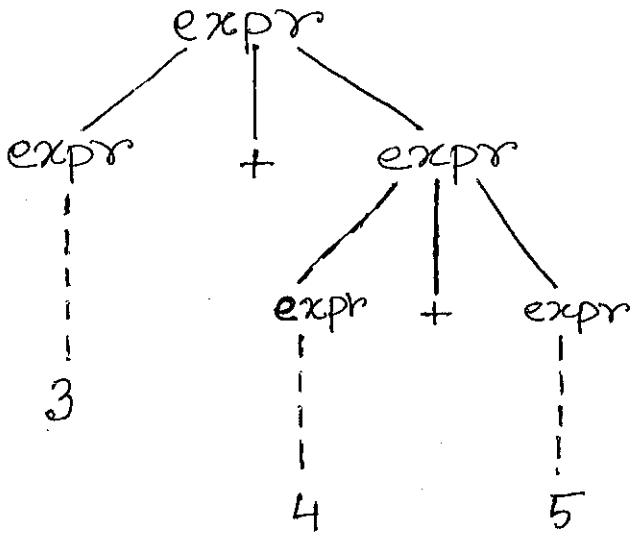
left associative

→ could be $4-2=2$

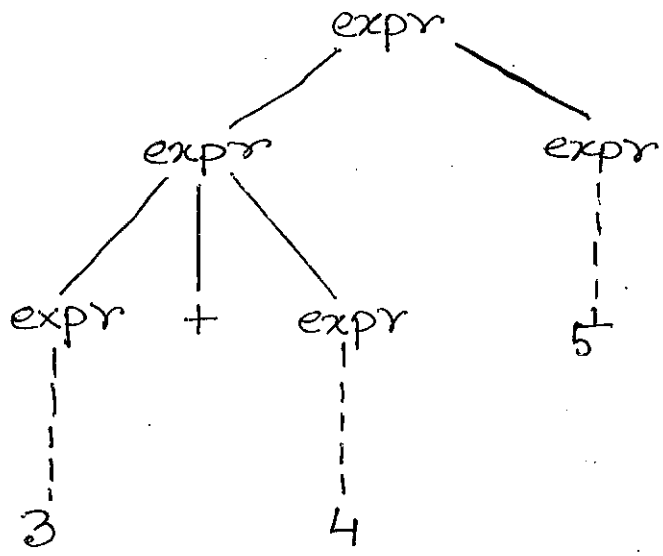
→ could be $8-2=6$

right associative

So, the problem makes a concern and needs to be addressed. For $3+4+5$ associativity provides



Right-associate



Left-associative

☐ We replace the rule —

$$expr \longrightarrow expr + expr$$

$$expr \longrightarrow expr + term \quad | \quad expr \longrightarrow term + expr$$

left-recursive | right-recursive

- Left-recursive → causes left-associate
- Right-recursive → causes right-associate

So, the modified version of the grammar that incorporates

Precedence
Associativity

Modified grammar

expr → expr + term | term

term → term * factor | factor

factor → (expr) | number

number → number digit | digit

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Comments :

- Sometimes, grammar becomes complex in the process of ambiguity elimination → In such case, we use disambiguity rule.

Ambiguity : Additional example.

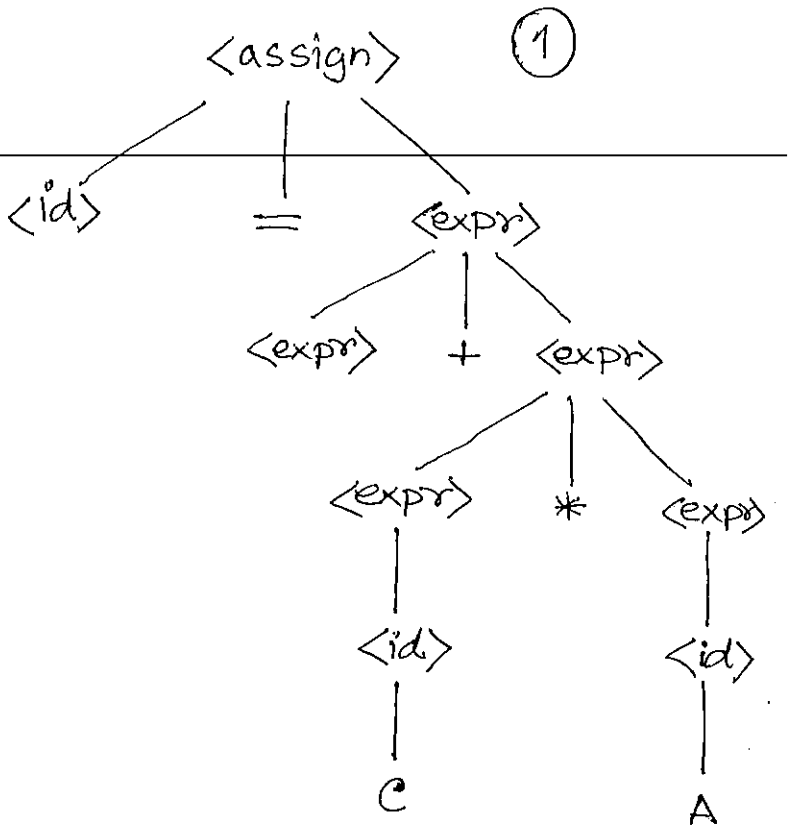
Ambiguous grammar

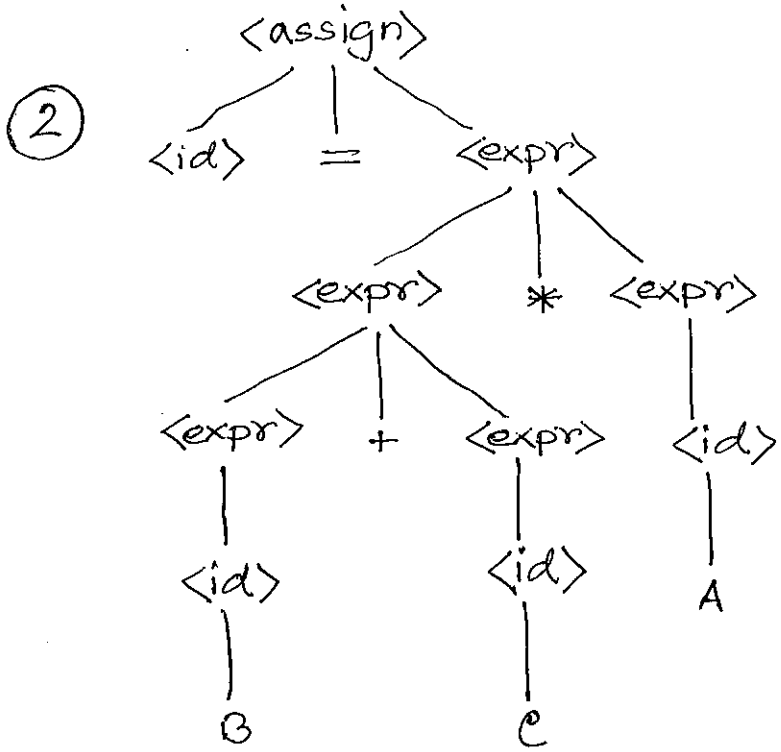
<assign> → <id> = <expr>

<id> → A | B | C

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

A = B + C *





Ambiguous grammar
- Same expression to be constructed but different parse trees

A = B+C *A

Given grammar:

<assign> → <id> = <expr>

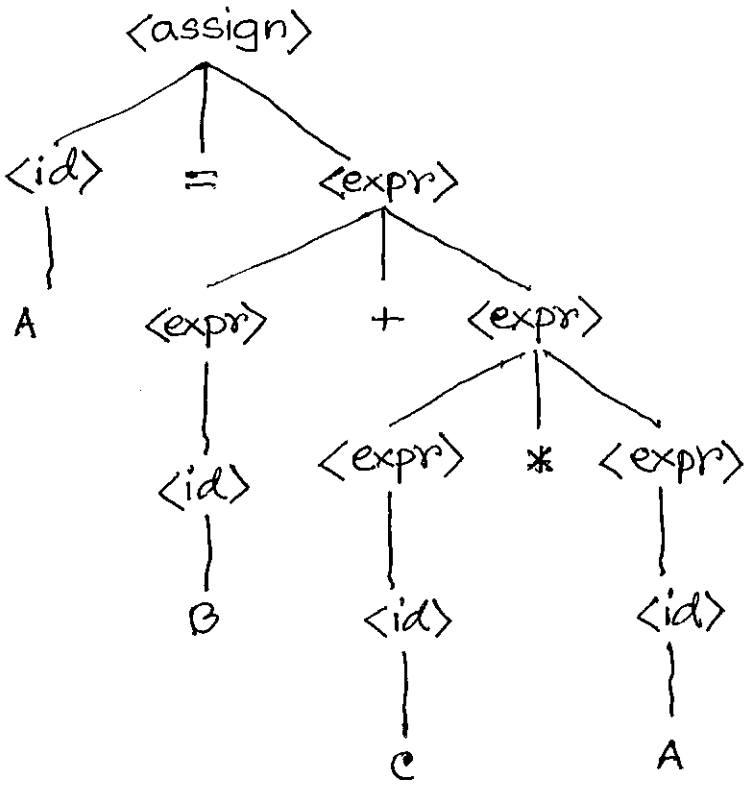
<id> → A | B | C

<expr> → <expr> + <expr> | <expr> * <expr>

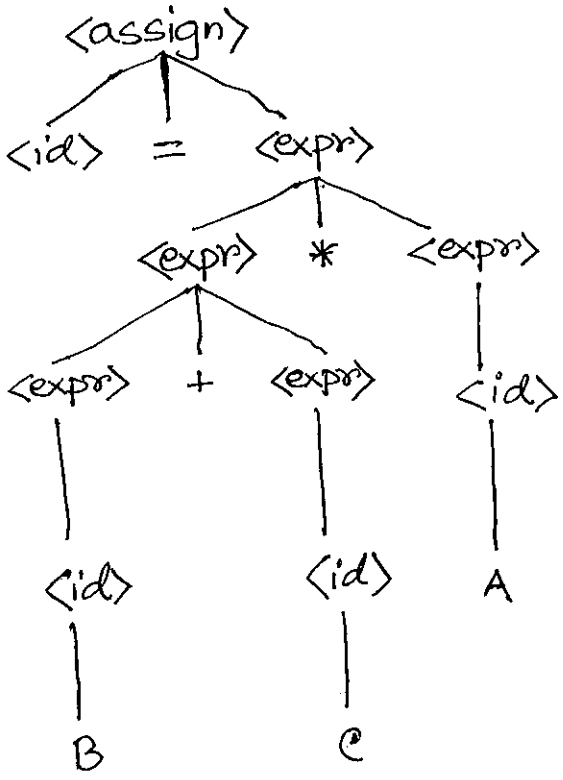
| (<expr>) | <id>

Syntactic representation of A = B+C *A has two distinct parse tree:

Parse Tree 1



Parse Tree 2



** The above grammar allows the parse tree of an expression to grow on both the left and right

→ Syntactic Ambiguity

Syntactic Ambiguity

Syntactic ambiguity of language structures is a problem.

Because,

- Compilers often base the semantics of those structures on their syntactic form.
- Compiler choses the code to be generated for a statement by examining its parse tree.

As a result, if a language structure has more than one parse tree, then the meaning of the structure cannot be determined uniquely.

Alternative, yet equivalent, notion of Ambiguity

1. If a grammar generates a sentence with more than one left-most derivation
2. If a grammar generates a sentence with more than one rightmost derivation.

Mathematically it is impossible to determine whether an arbitrary grammar is ambiguous.

Example: Operator Precedence

In the previous grammar, if '*' has been assigned higher precedence than "+", multiplication will be done first.

→ Regardless of the order of appearance of the two operators.

- Higher precedence of "*"
  - ↳ will be provided by the language designer.

- Position/appearance of an operator at a lower level of the parse tree indicates that the operator has higher precedence.

Modified grammar:

<assign> → <id> = <expr>

<id> → A | B | C

<expr> → <expr> + <term> | <term>

<term> → <term> * <factor> | <factor>

<factor> → (<expr>) | <id>

Leftmost

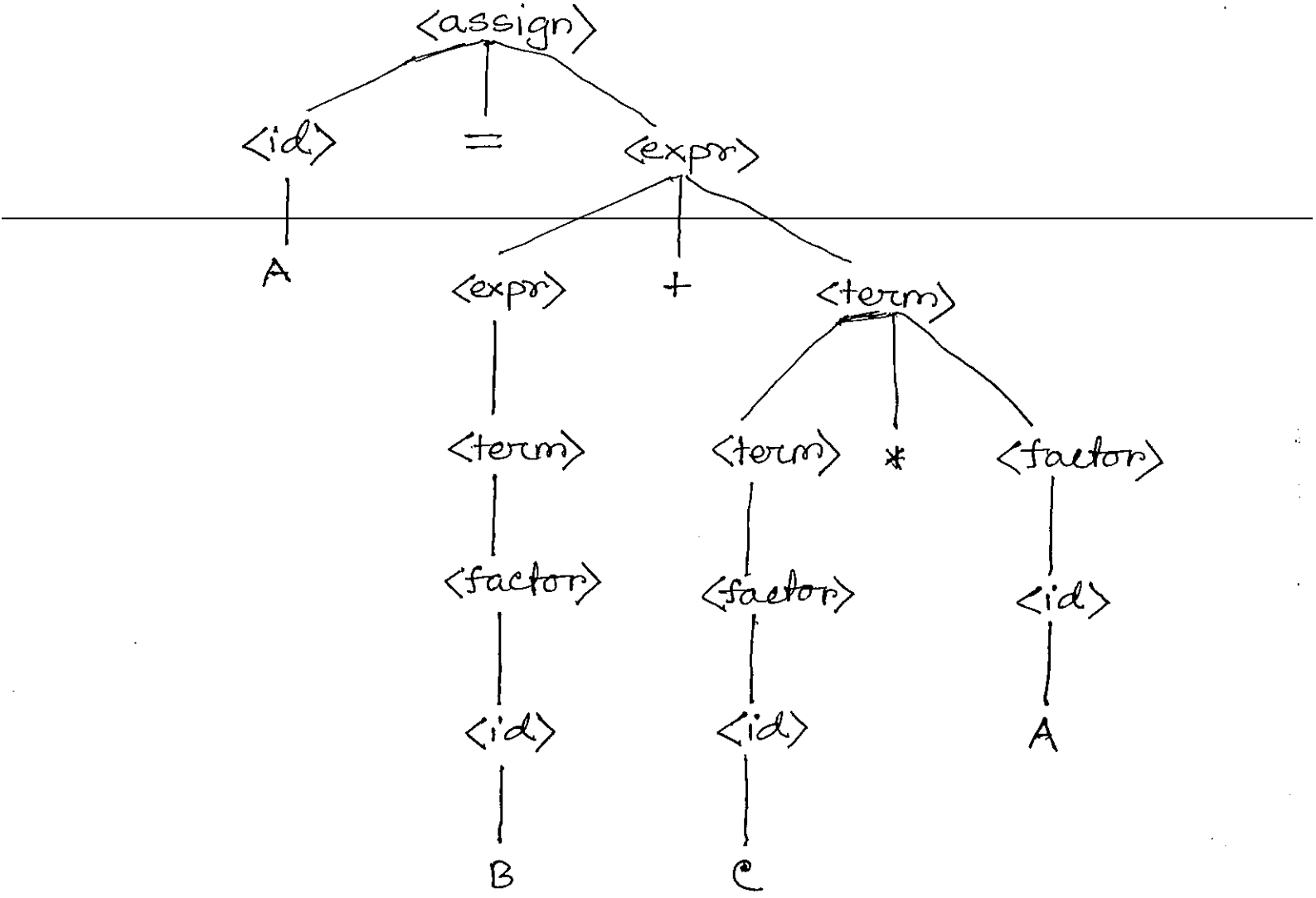
<assign> → <id> = <expr>
→ A = <expr>
→ A = <expr> + <term>
→ A = <term> + <term>
→ A = <factor> + <term>

→ A = <id> + <term>
→ A = B + <term>
→ A = B + <term> * <factor>
→ A = B + <factor> * <factor>
→ A = B + <id> * <factor>
→ A = B + C * <factor>
→ A = B + C * <id>
→ A = B + C * A

Rightmost

<assign> → <id> = <expr>
→ <id> = <expr> + <term>
→ <id> = <expr> + <term> * <factor>
→ <id> = <expr> + <term> * <id>
→ <id> = <expr> + <term> * A
→ <id> = <expr> + <factor> * A
→ <id> = <expr> + <id> * A
→ <id> = <expr> + C * A
→ <id> = <term> + C * A
→ <id> = <factor> + C * A
→ <id> = <id> + C * A
→ <id> = B + C * A
→ A = B + C * A

Both the leftmost and rightmost derivation are represented using same parse tree —



Example

Consider the grammar

$S \rightarrow AS \mid \varepsilon$

$A \rightarrow A1 \mid 0A1 \mid 01$

Identify/Show if the grammar is ambiguous.

Answer: Let's consider the string

00111

Leftmost 1

$S \rightarrow \underline{A}S$

$\rightarrow \underline{0A1}S$

$\rightarrow 0\underline{A1}1S$

$\rightarrow 00\underline{A111}S$

$\rightarrow 00111$

Leftmost 2

$S \rightarrow AS$

$\rightarrow A1S$

$\rightarrow 0A11S$

$\rightarrow 00111S$

$\rightarrow 00111$

Unambiguous version of the Ambiguous grammar

<assign> → <id> = <expr>

<id> → A | B | C

<expr> → <expr> + <term> | term

<term> → <term> * <factor> | <factor>

<factor> → (<expr>) | <id>

Another example : Ambiguous Grammar

Given,

$S \rightarrow AB \mid C$
$A \rightarrow aAb \mid ab$
$B \rightarrow cBd \mid cd$
$C \rightarrow aCd \mid aDd$
$D \rightarrow bDc \mid bc$

Check if the grammar is ambiguous.

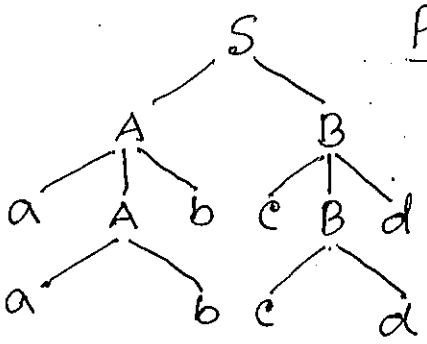
To identify if a grammar is ambiguous we can use leftmost derivation.

If two distinct leftmost derivation exist for any string, the grammar is ambiguous.

Consider the string aabbccdd

$S \rightarrow AB$
$\rightarrow aAbB$ [A=Ab]
$\rightarrow aabbB$ [A=ab]
$\rightarrow aabbcBd$ [B=cBd]
$\rightarrow aabbccdd$ [B=cd]

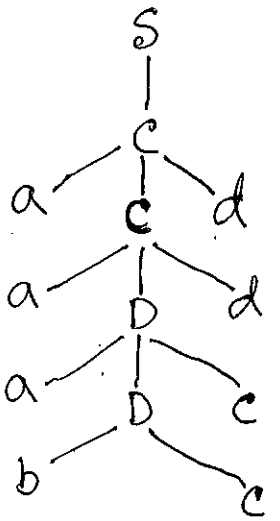
Parse Tree



Another leftmost derivation :

$S \rightarrow C$
$\rightarrow aCd$
$\rightarrow aaDdd$ [C=aDd]
$\rightarrow aabDcdd$ [D=bDc]
$\rightarrow aabbccdd$ [D=bc]

Parse Tree



source : iitg.ac.in

Example : CFG

For given alphabet $\Sigma = \{a, b\}$, let's identify the context-free-grammar —

1. All nonempty strings starting and ending with same symbol.

$$S \rightarrow aXa \mid bXb \mid a \mid b$$
$$X \rightarrow aX \mid bX \mid \varepsilon$$

For instance,

$S \rightarrow aXa$	$S \rightarrow aXa$
$\rightarrow abXa$	$\rightarrow aaXa$
$\rightarrow aba$	$\rightarrow aabXa$
	$\rightarrow aaba$

2. Very similar to number 1 — all nonempty strings (of length greater than one) with same starting and ending symbol.

$$S \rightarrow aXa \mid bXb$$
$$X \rightarrow aX \mid bX \mid \varepsilon$$

3. All palindromes

$$S \rightarrow aSa \mid bSb \mid a \mid b \mid \varepsilon$$

source: 'stanford.edu

Given the grammar — G

$A \rightarrow Ba \mid bC$
$B \rightarrow d \mid eBf$
$C \rightarrow gC \mid g$

L(G) → language with grammar G

Determine if the following strings are in L(G)

bg, bffd, bggg, edfa, eedffa faae defa

Answer:

From the given grammar, it is clear that L(G) must end in either "a" or "g".

So, the strings **bffd, faae** cannot be in the language.

Also,

the only string starting with d is da.

$A \rightarrow Ba$
$\rightarrow da$

So, defa is not in the language.

Extended Backus-Naur Form

Consider the rule —

(1)

number → number digit
→ number digit digit
→ number digit digit digit
→ digit ... ... digit

Another rule done previously —

generates

expr → expr + term | term

(2)

expr → expr + term
→ expr + term + term
→ expr + term + term + term
...
→ expr + term ... + term
→ term + ... + term

Such repetitive structures occur frequently and alternative notion becomes more useful —

(1) number → digit {digit}

(2) expr → term {+ term}

Here, the curly brackets { } stand for

"zero or more repetitions of"

So,

number → digit {digit}

This rule expresses that number is a sequence of one or more digits.

expr → term {+ term}

expression is a term followed by zero or more repetitions of a "+" and another term.

So, here the curly brackets { } are acting as new metasymbols.

Notations shown above for number, expr are known as

Extended Backus-Naur Form (EBNF)

Often provides simpler version of the grammar.

Example : EBNF

- Another situation could be the optional part in many structure —

For instance, if-statement in C

BNF { if-statement → if (expression) statement |
if (expression) statement else statement

↓ more simply and expressively by EBNF

if-statement → if (expression) statement [else statement]

here, [ ] is the new metasymbols that indicate the optional part of the structure.

- EBNF for multiple choice option

<term> → <term> * <factor> |
<term> / <factor> |
<term> % <factor>

↓ EBNF

<term> → <term> (* | / | %) <factor>

Some confusions — EBNF

$\langle expr \rangle \rightarrow \langle expr \rangle + \langle term \rangle$

↓ EBNF

$\langle expr \rangle \rightarrow \langle expr \rangle \{ + term \}$

Here, direction of associativity is missing.

↳ A syntax analyzer based on EBNF.

↓

by enforcing the correct associativity.

↳ say, by assuming that any operator involved in a curly bracket repetition is left-associative.

Example :

BNF

$\langle expr \rangle \rightarrow \langle expr \rangle + \langle term \rangle \mid$
$\rightarrow \langle expr \rangle - \langle term \rangle \mid$
$\rightarrow \langle term \rangle$

$\langle term \rangle \rightarrow \langle term \rangle * \langle factor \rangle \mid$
$\langle term \rangle / \langle factor \rangle \mid$
$\langle factor \rangle$

$\langle factor \rangle \rightarrow \langle exp \rangle ** \langle factor \rangle$
$\langle exp \rangle$

$\langle exp \rangle \rightarrow (\langle expr \rangle) \mid id$

EBNF

$\langle expr \rangle \rightarrow \langle term \rangle \{ (+ \mid -) \langle term \rangle \}$

$\langle term \rangle \rightarrow \langle factor \rangle \{ (* \mid /) \langle factor \rangle \}$

$\langle factor \rangle \rightarrow \langle exp \rangle \{ ** \langle exp \rangle \}$

$\langle exp \rangle \rightarrow (\langle expr \rangle) \mid id$

BNF and EBNF

BNF	EBNF
expr → expr + term | term	expr → term { + term }
term → term * factor | factor	term → factor { * factor }
factor → (expr) | number	factor → (expr) | number
~~number → number digit | digit~~	number → digit { digit }
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9	digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

BNF and EBNF rules for simple integer arithmetic expressions.