# Python Advanced Programming

# Unit 1: Tokenization

CHAPTER 2: EXTENDED BACKUS–NAUR FORM (EBNF)

DR. ERIC CHOU IEEE SENIOR MEMBER

# Tokenization

LECTURE 1

## Tokenization

- In the previous chapter, we discussed the tokenization of plain text file (**.txt**) and comma separated values file (**.csv**).
- In this chapter and the next chapter, we will discuss the text files which has tokens which follows certain syntax.
- The syntax is described in Extended Backus-Naur Form (**EBNF**). And the **EBNF** is further formulated (coded) as regular expression for Python language for qualifying input **token stream**.

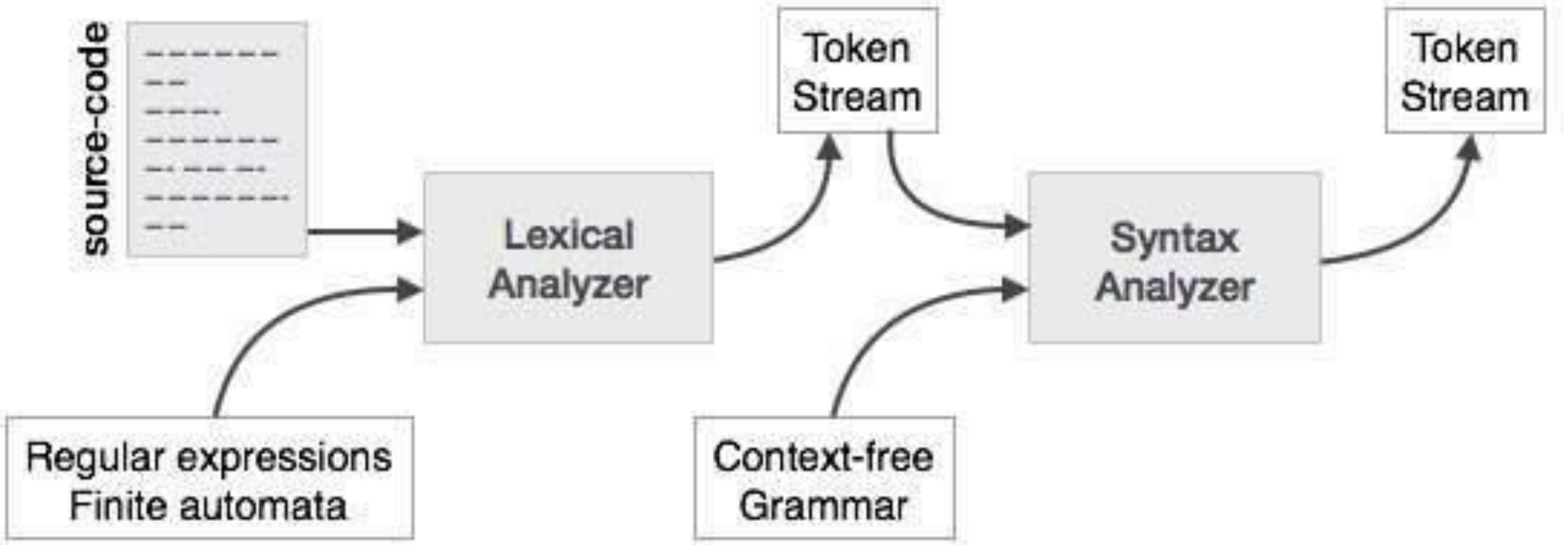
## Tokenization

- Tokenization does not involve the interpretation or compilation of the text file.
- The interpretation of a text file is called Parsing (Typing), generation of abstract syntax tree and the generation of target code.

## Tokenization

- The process of segmenting a string of characters into words is known as **tokenization**.
- Tokenization is important in text processing because it tells our processing software what our basic units are.
- ***Type*** vs. ***token***

## Tokenization    Parsing



# Introduction of Formal Language

LECTURE 2

## Formal Languages

### A Computer Term

formal language

noun

1. a language designed for use in situations in which natural language is unsuitable, as for example in mathematics, logic, or computer programming. The symbols and formulas of such languages stand in precisely specified syntactic and semantic relations to one another

**A Set of Rules**

2. (**logic**) a logistic system for which an interpretation is provided: distinguished from formal calculus in that the semantics enable it to be regarded as *about* some subject matter

**A Logistic System**

**R**ule (Grammar) – **I**nput (Language) – **M**achine (States)

## Grammar

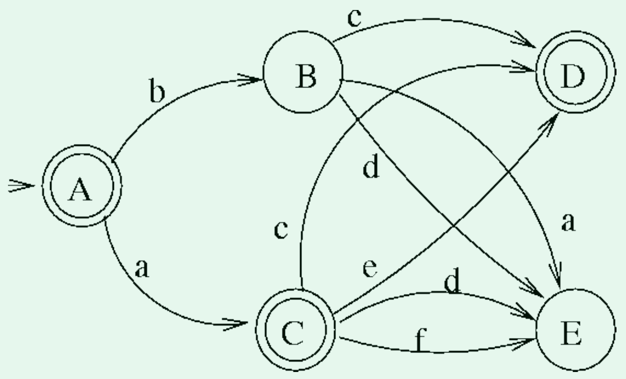
$$\begin{array}{lcl} S & \rightarrow & aS \mid bX \\ X & \rightarrow & aX \mid bY \\ Y & \rightarrow & aY \mid bZ \mid \Lambda \\ Z & \rightarrow & aZ \mid \Lambda \end{array}$$

Formal

## Language Input

```
x := a + b;
y := a * b;
while (y > a) {
   a := a + 1;
   x := a + b
}
```

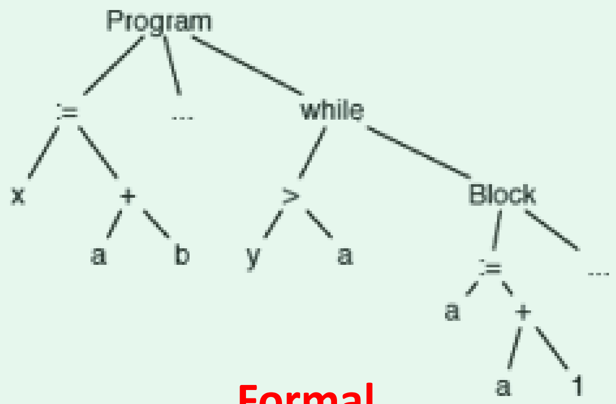
Informal



Formal

## State Machine

## Validated Tokens Syntax Tree



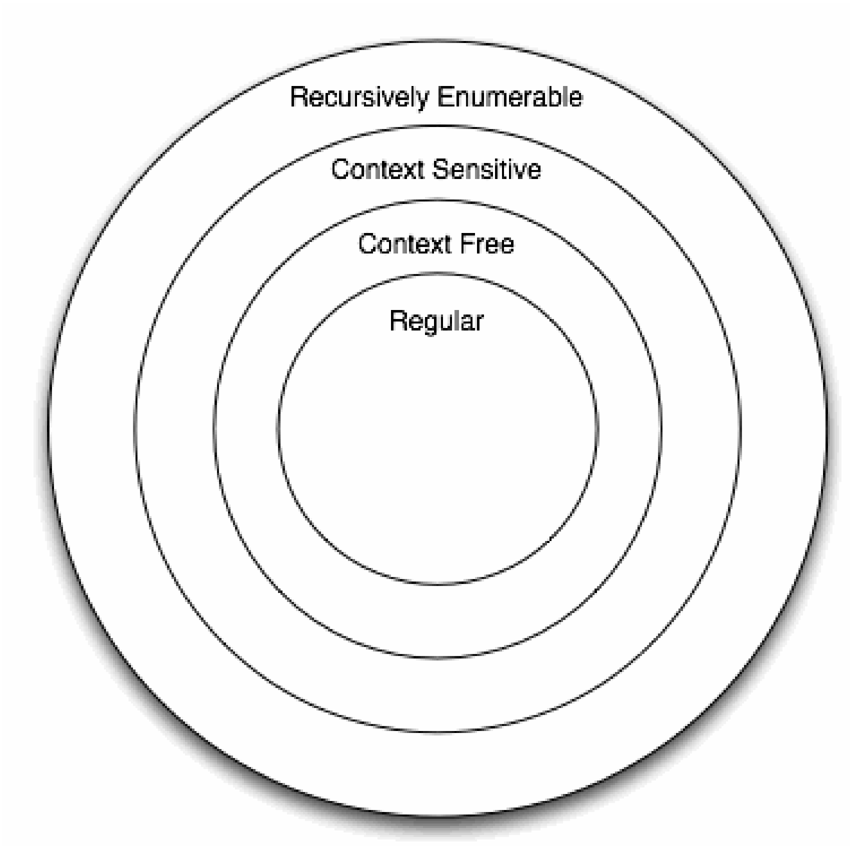
Formal

## Target Code

Machine Code
Assembly
Byte Code
Object Code

Informal

## Hierarchy of Grammars

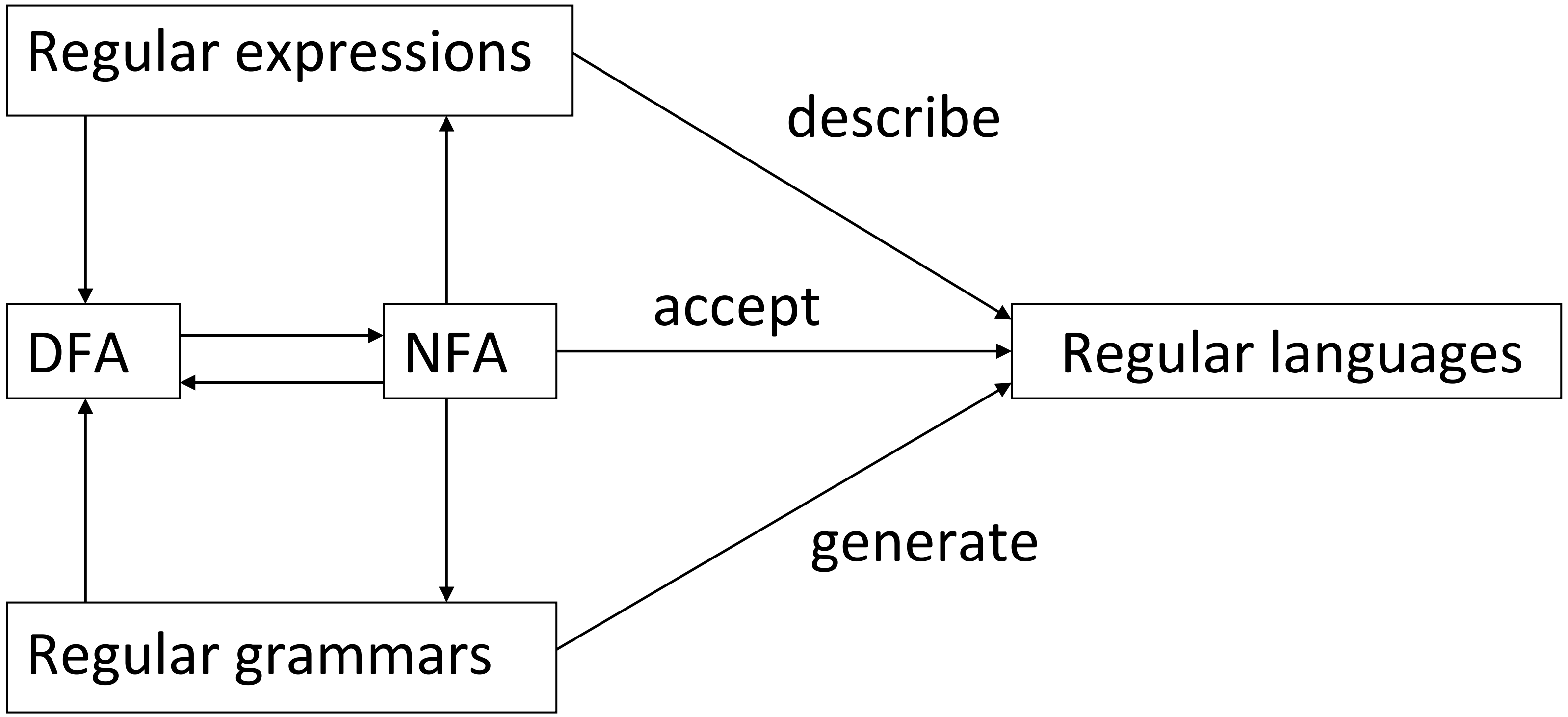


**Chomsky** defined four types of grammars in the hierarchy, of which, the regular grammars were classified as the most restricted. The other types are context free grammars, context sensitive grammars, and recursively enumerable grammars:

## Expression, Grammar, Language

- **Expression** is a special patterns (for the tokens of a language)
- **Grammar** is a set of rules.
- **Language** is all the composition of symbol sequences generated by the grammar.

# 3 ways of specifying regular languages



# Syntax of Arabic numerals

$$\textit{digit} \longrightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$
$$\textit{non\_zero\_digit} \longrightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$
$$\textit{natural\_number} \longrightarrow \textit{non\_zero\_digit}\ \textit{digit}^*$$

Alphabet = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
| : options
* : Kleene star *, zero or more repetition

Similar syntax rules and semantic interpretations can be devised for rational numbers, (limited-precision) real numbers, arithmetic, assignments, control flow, declarations, and indeed all of programming languages.

- A **regular language** over an alphabet $\Sigma$ is one that contains either a single string of length 0 or 1, or strings which can be obtained by using the operations of union, concatenation, or Kleene* on strings of length 0 or 1.

- **Operations on formal languages:**
  Let $L_1$ = {10} and $L_2$ = {011, 11}.
  - Union: $L_1 \cup L_2$ = {10, 011, 11}
  - Concatenation: $L_1 L_2$ = {10011, 1011}
  - Kleene Star: $L_1^*$ = {λ, 10, 1010, 101010, … }

  Other operations: intersection, complement, difference

## Why are we studying these?

- Formally, define a language using formal grammar. (avoidance of ambiguity.)
- Utilize the tools for syntax design in general programming. (**Regular Expression**, Lex, Yacc)
- Design a compiler. (Useful for software development including development tools, EDA, system automation)

# Regular Grammar

LECTURE 3

## Finite Automata

### 3-in-1  R, G, M

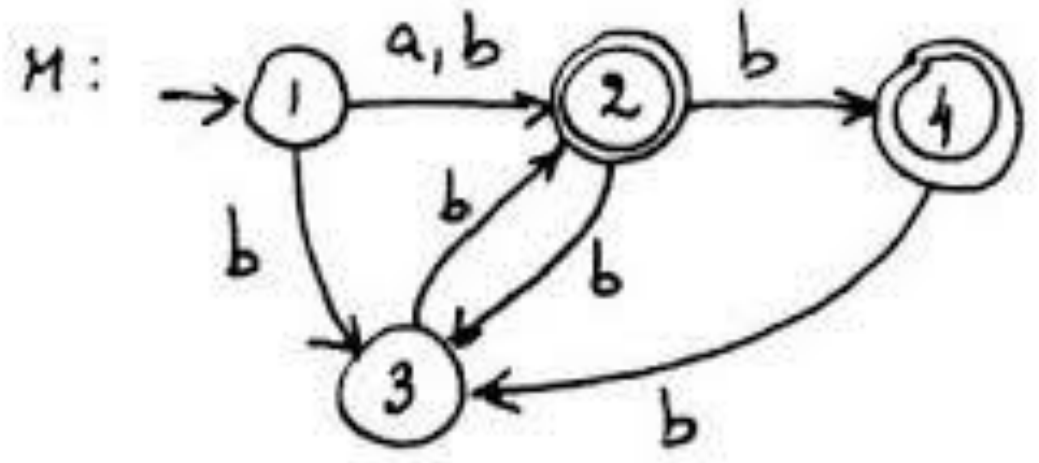
Regular expressions, regular grammars and finite automata are simply three different formalisms for the same thing. There are algorithms to convert from any of them to any other.

$R = (a \mid b)(bb \mid b)^*$

$G: \quad S \rightarrow aAB \mid bAb$

$\quad\quad A \rightarrow bbA \mid b \mid \varepsilon$

$\quad\quad B \rightarrow bB \mid bAB \mid \varepsilon$



# A Formal Definition of Regular Grammars

## Terminal, Non-terminal and Rules

A *regular grammar* is a mathematical object, $G$, with four components, $G = (N, \Sigma, P, S)$, where

•$N$ is a nonempty, finite set of *nonterminal symbols*,

•$\Sigma$ is a finite set of *terminal symbols* , or *alphabet, symbols,*

•$P$ is a set of grammar rules, each of one having one of the forms

- $A \rightarrow aB$
- $A \rightarrow a$
- $A \rightarrow \varepsilon$, for $A, B \in N$, $a \in \Sigma$, and $\varepsilon$ the empty string, and

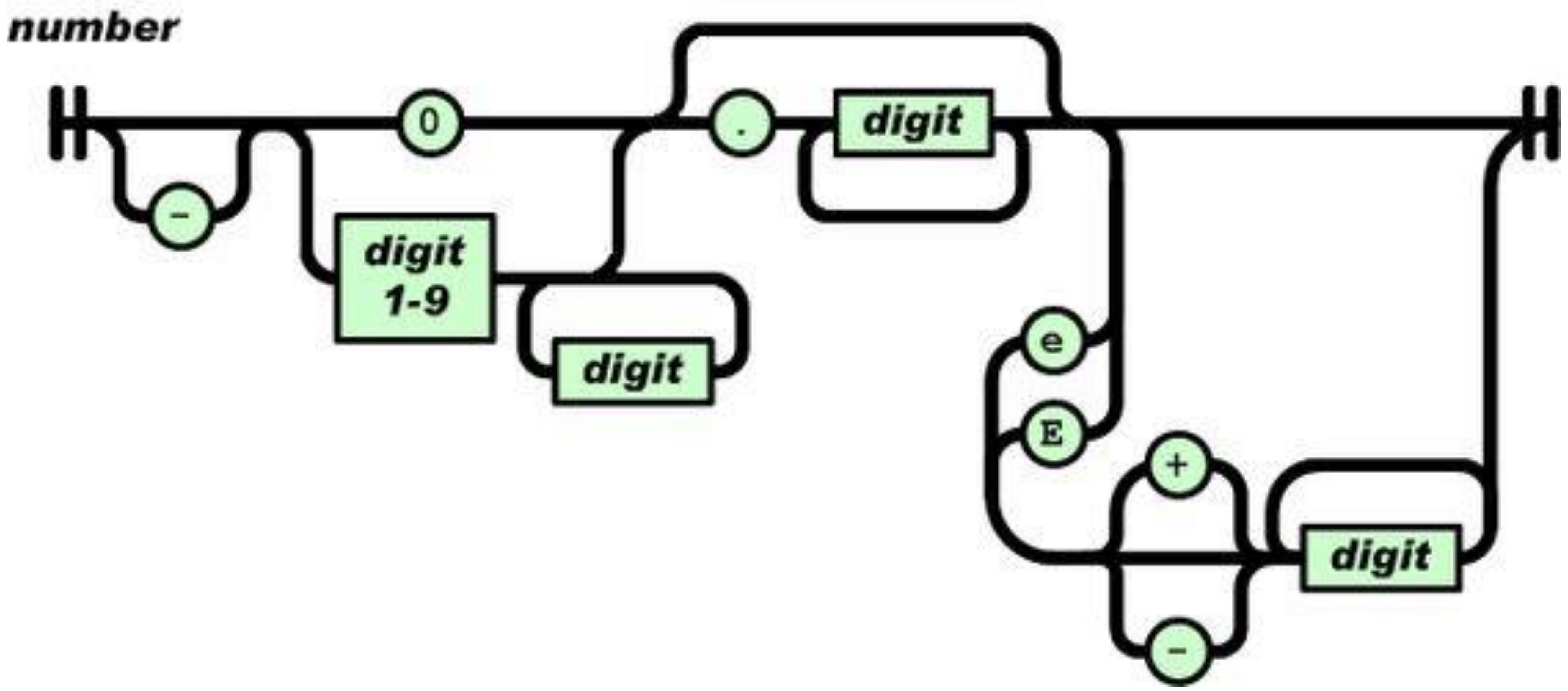
•$S \in N$ is the *start symbol*.

Notice that this definition captures all of the components of a regular grammar that we have heretofore identified (*heretofor*, how often does one get to use that word?).

## A regular expression is one of the following

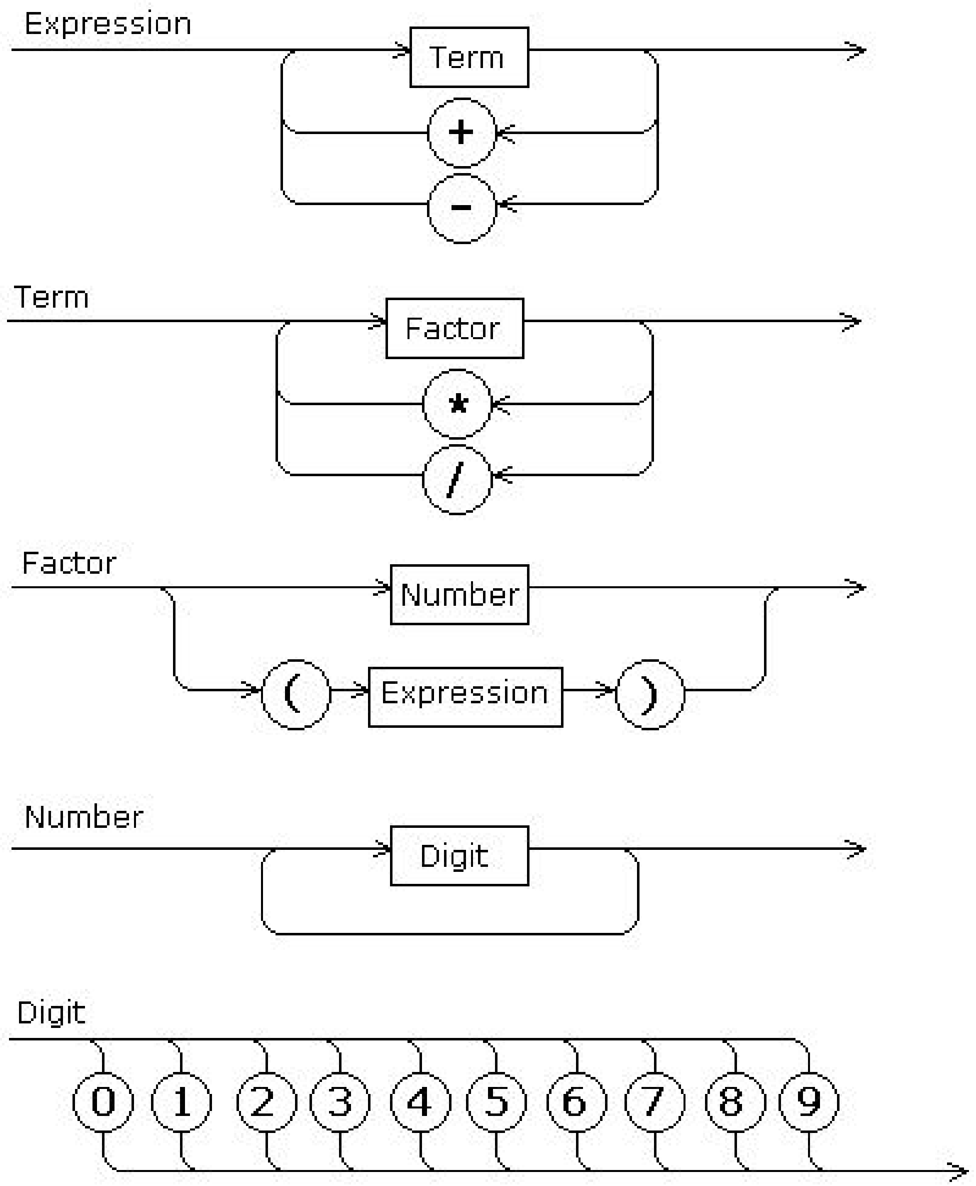
- A character
- The empty string, denoted by ε
- Two regular expressions concatenated
- Two regular expressions separated by | (i.e., or)
- A regular expression followed by the Kleene star * (concatenation of zero or more strings)

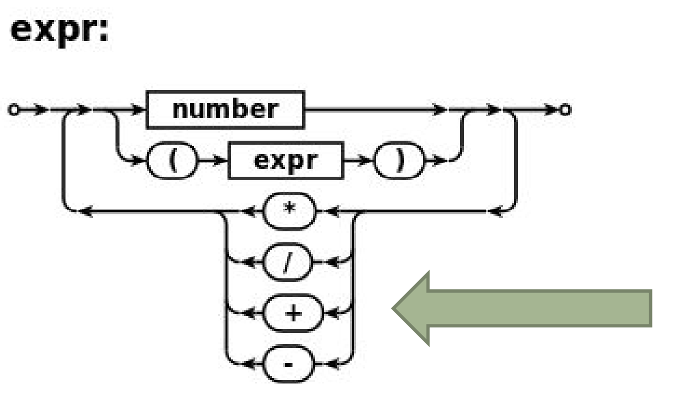
# Backus-Naur Form for Number



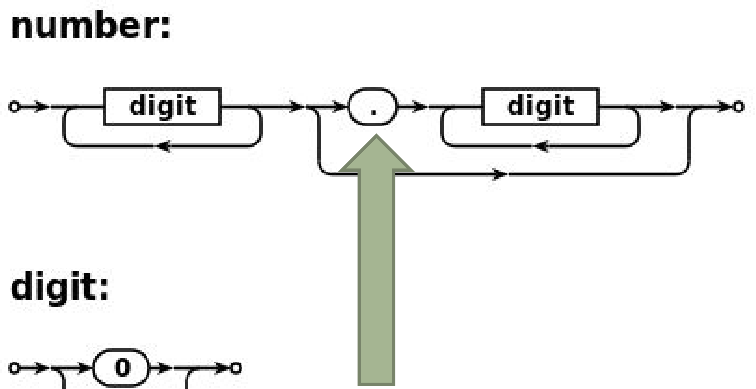
## Numerical constants accepted by a simple hand-held calculator

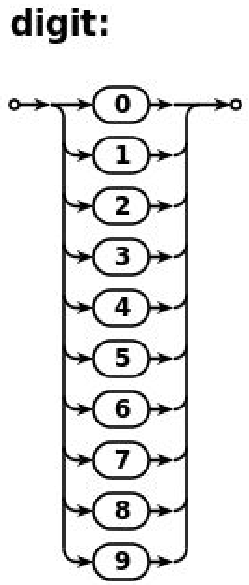
$$
\begin{aligned}
\textit{number} &\longrightarrow \textit{integer} \mid \textit{real} \\
\textit{integer} &\longrightarrow \textit{digit}\ \textit{digit}^{*} \\
\textit{real} &\longrightarrow \textit{integer}\ \textit{exponent} \mid \textit{decimal}\ (\,\textit{exponent} \mid \epsilon\,) \\
\textit{decimal} &\longrightarrow \textit{digit}^{*}\ (\ .\ \ \textit{digit} \mid \textit{digit}\ \ .\ )\ \textit{digit}^{*} \\
\textit{exponent} &\longrightarrow (\,\texttt{e} \mid \texttt{E}\,)\,(\,\texttt{+} \mid \texttt{-} \mid \epsilon\,)\ \textit{integer} \\
\textit{digit} &\longrightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}
\end{aligned}
$$





No precedence





With decimal point for floating point and integer

## Regular vs. context-free

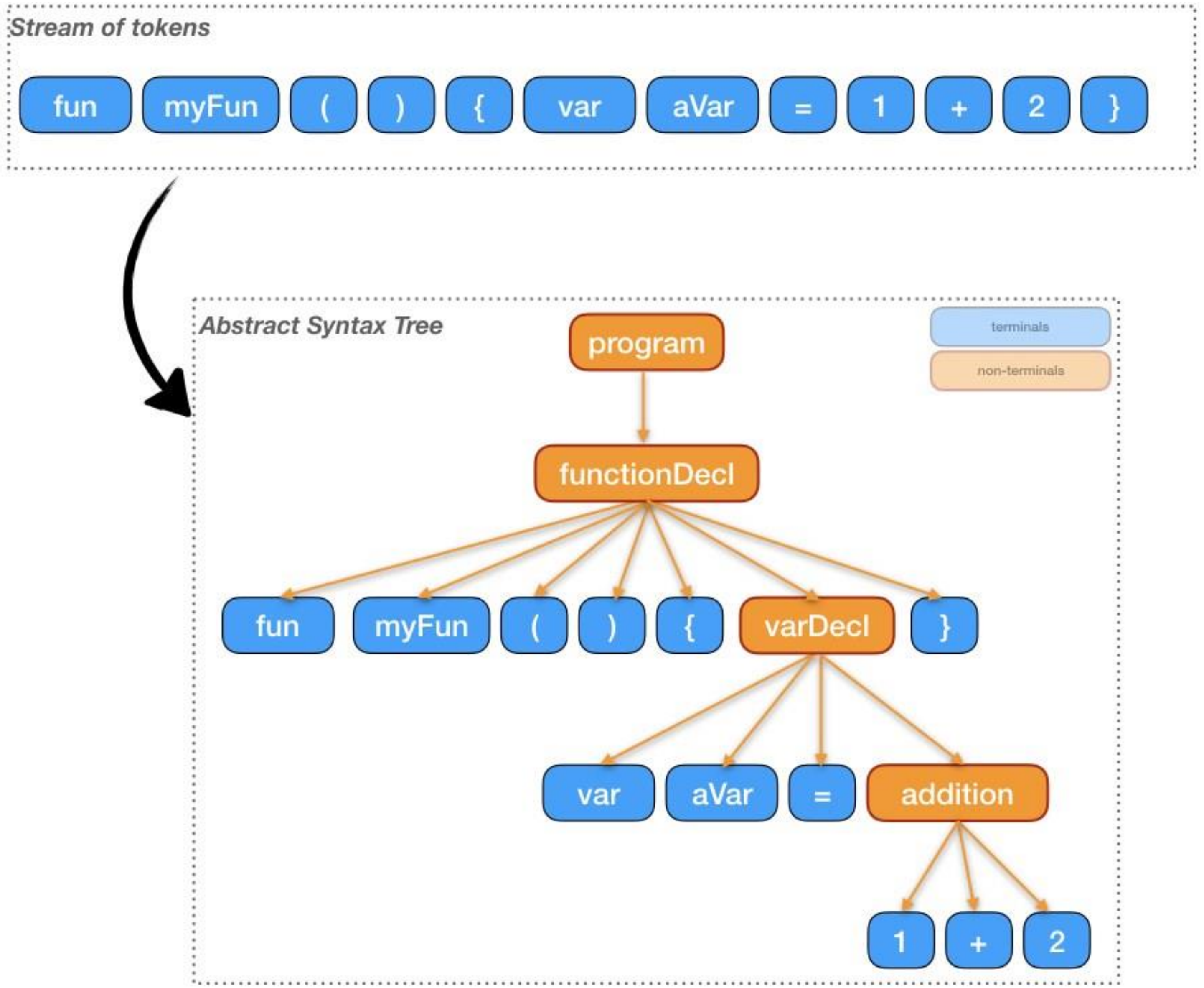
Are regular languages context-free?

Yes, because context-free means that there is a single variable on the left side of each grammar rule. All regular languages are generated by grammars that have a single variable on the left side of each grammar rule.

But, as we have seen, not all context-free grammars are regular. So regular languages are a proper subset of the class of context-free languages.

## Major Difference Between Regular Expression and Context-free Grammar

- The rules of context-free grammar are recursive
- The power of representation of context-free grammar is increased significantly over that of regular expression


Stream of tokens
fun
myFun
(
)
{
var
aVar
=
1
+
2
}
Abstract Syntax Tree
program
functionDecl
fun
myFun
(
)
{
varDecl
}
var
aVar
=
addition
1
+
2
terminals
non-terminals

# Backus-Naur Form

LECTURE 4

## Backus-Naur Form (BNF) notation

When describing languages, Backus-Naur form (**BNF**) is a formal notation for encoding grammars intended for human consumption.

Many programming languages, protocols or formats have a BNF description in their specification.

Every rule in Backus-Naur form has the following structure:

```
name ::= expansion
```

The symbol ::= means "may expand into" and "may be replaced with."

## Backus-Naur Form (BNF) notation

- In some texts, a **name** is also called a **non-terminal** symbol.
- Every **name** in Backus-Naur form is surrounded by angle brackets, **< >**, whether it appears on the left- or right-hand side of the rule.
- An **expansion** is an expression containing terminal symbols and non-terminal symbols, joined together by sequencing and choice.
- A **terminal symbol** is a **literal** like ("+" or "function") or a class of literals (like integer).
- Simply juxtaposing expressions indicates sequencing.
- A vertical bar **|** indicates choice.

# Backus-Naur Form (BNF) notation

For example, in BNF, the classic expression grammar is:

```
<expr> ::= <term> "+" <expr>
        |  <term>
<term> ::= <factor> "*" <term>
        |  <factor>
<factor> ::= "(" <expr> ")"
          |  <const>
<const> ::= integer
```

## BNF

Naturally, we can define a grammar for rules in BNF:

```
rule → name ::= expansion
name → < identifier >
expansion → expansion expansion            ; concatenation
expansion → expansion | expansion          ; option
expansion → name                           ; non-terminals
expansion → terminal                       ; terminals
```

## Regular Expression is Another Way to Define Regular Grammar

- We might define identifiers as using the regular expression **[-A-Za-z_0-9]+**.
- A terminal could be a quoted literal (like "+", "switch" or "<<=") or the name of a class of literals (like integer).
- The name of a class of literals is usually defined by other means, such as a regular expression or even prose.

## BNF

```
<expr> → <expr> + <term>
       | <expr> - <term>
       | <term>

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

## EBNF

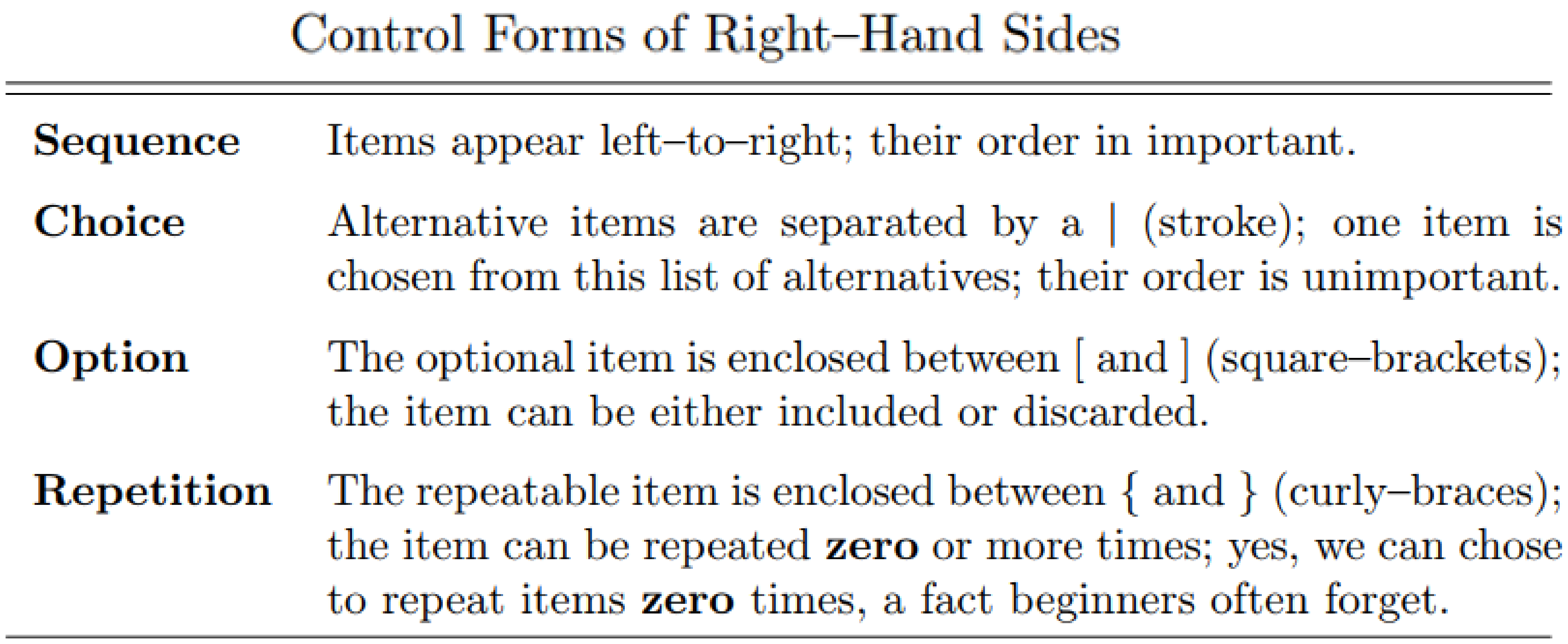
```
<expr> → <term> {(+ | -) <term>}

<term> → <factor> {(* | /) <factor>}
```

# Extended Backus-Naur Form

LECTURE 5

# EBNF Rules and Descriptions



Control Forms of Right–Hand Sides

| | |
|---|---|
| **Sequence** | Items appear left–to–right; their order in important. |
| **Choice** | Alternative items are separated by a \| (stroke); one item is chosen from this list of alternatives; their order is unimportant. |
| **Option** | The optional item is enclosed between [ and ] (square–brackets); the item can be either included or discarded. |
| **Repetition** | The repeatable item is enclosed between { and } (curly–braces); the item can be repeated **zero** or more times; yes, we can chose to repeat items **zero** times, a fact beginners often forget. |

## Basic Symbols for EBNF

:= derivation

| option

[] optional set ; () sometimes

{} repetition ; Kleene’s star *

; a a* is equivalent to a+

## History of EBNF

- The earliest EBNF was originally developed by Niklaus
- Wirth incorporating some of the concepts (with a different syntax and notation) from Wirth syntax notation.
- However, many variants of EBNF are in use. The International Organization for Standardization has adopted an EBNF standard (ISO/IEC 14977).
- This article uses EBNF as specified by the ISO for examples applying to all EBNFs. Other EBNF variants use somewhat different syntactic conventions.

## EBNF

- In computer science, extended Backus-Naur form (**EBNF**) is a family of metasyntax notations, any of which can be used to express a context-free grammar.
- **EBNF** is used to make a formal description of a formal language which can be a computer programming language.
- They are extensions of the basic Backus–Naur form (**BNF**) metasyntax notation.

# ISO Standard

## ISO/IEC 14977 standard

| Usage | Notation |
|---|---|
| definition | = |
| concatenation | , |
| termination | ; |
| alternation | \| |
| optional | [ ... ] |
| repetition | { ... } |
| grouping | ( ... ) |
| terminal string | " ... " |
| terminal string | ' ... ' |
| comment | (* ... *) |
| special sequence | ? ... ? |
| exception | - |

## Grammar Types

- **Extended BNF (EBNF):**
  - BNF's notation + regular expressions
  - Different notations persist:
    - *Optional parts:* Denoted with a subscript as opt or used within a square bracket.
      - <proc_call> → ident ( <expr_list>)opt
      - <proc_call> → ident [ ( <expr_list>)]
  - *Alternative parts:*
    - Pipe ( | ) indicates either-or choice
    - Grouping of the choices is done with square brackets or brackets.
      - <term> → <term> [+ | -] const
      - <term> → <term> (+ | -) const
  - *Put repetitions* (0 or more) in braces ({ })
    - Asterisk indicates zero or more occurrence of the item.
    - Presence or absence of asterisk means the same here, as the presence of curly brackets itself indicates zero or more occurrence of the item.
      - <ident> → letter {letter | digit}*
      - <ident> → letter {letter | digit}

# EBNF: Lecture vs. Python Documentation

LECTURE 6

# EBNF Format in this course

There are a few different typographic conventions used in the **EBNF** lecture, compared to the **EBNF** used in actual Python Documentation. Here is a short summary of the six differences, and a short and large example.

1. Write <= (separating LHS from RHS) as **::=**
2. Italicized *names* (of rules) are just written as **names**
3. Boxed characters (which stand for themselves) are written within **quotes**
4. () do not stand for themselves; they are used for grouping (see rule 5) write "(" and ")" for parentheses in the EBNF used for Python Documentation
5. **{item}** is written as **item***; **{item1 ... itemN}** is written **(item1 ... itemN)***
6. Writing **+** superscript means repeat 1 or more times

## EBNF in this course

::= derivation

| option

() optional set ; () sometimes

a* repetition ; Kleene's star *

a+ ; a a* is equivalent to a+

"51" "(" ; literals

# EBNF Example in Class Note

**Short Example:**

```
digit   ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
integer ::= ["+"|"-"] digit digit*
```

the last rule can be written as either:

```
integer ::= ["+"|"-"] digit (digit)*
```

or

```
integer ::= ["+"|"-"] (digit)+
```

Online:

See https://docs.python.org/3/reference/simple_stmts.html

Questions: is it legal in Python to write a = b = 0

Also, see the import statement (and all its forms)

# EBNF

Large Example:

This is from Section 6.1.3.1 **Format Specification Mini-Language** of the Python Library. Format strings contain “replacement fields” surrounded by curly braces {}. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. If you need to include a brace character in the literal text, it can be escaped by doubling: {{ and }}.

## Grammar for Replacement Field

**The grammar for a replacement field is as follows:**

```
replacement_field ::=  "{" [field_name] ["!" conversion]
                                        [":" format_spec] "}"
field_name           ::=  arg_name ("." attribute_name |
                                               "[" element_index "]")*
arg_name             ::=  [identifier | integer]
attribute_name    ::=  identifier
element_index     ::=  integer | index_string
index_string        ::=  <any source character except "]“ > +
conversion          ::=  "r" | "s" | "a"
```

## Grammar for Replacement Field

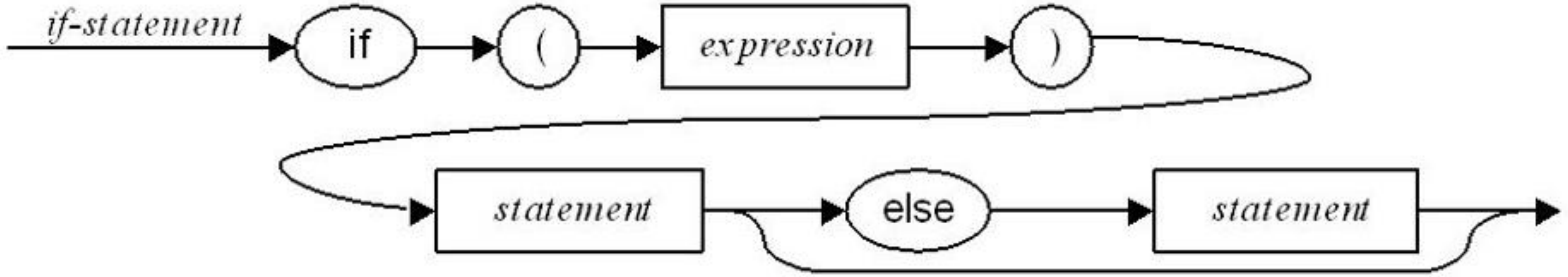
```
format_spec ::=  [[fill]align][sign][#][0][width][,][.precision][type]
fill        ::=  <a character other than '{' or '}'>
align       ::=  "<" | ">" | "=" | "^"
sign        ::=  "+" | "-" | " "
width       ::=  integer
precision   ::=  integer
type        ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" |
                 "n" | "o" | "s" | "x" | "X" | "%"
```

# Syntax Diagram

LECTURE 7

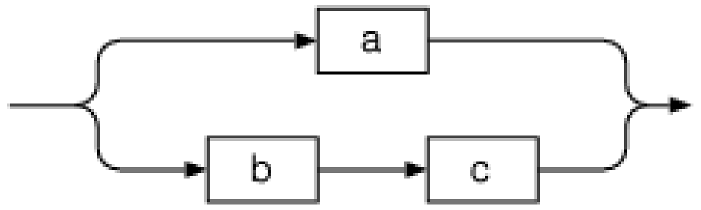
# Syntax Diagrams

- An alternative to EBNF.
- Rarely seen any more: EBNF is much more compact.
- Example (if-statement, p. 101):

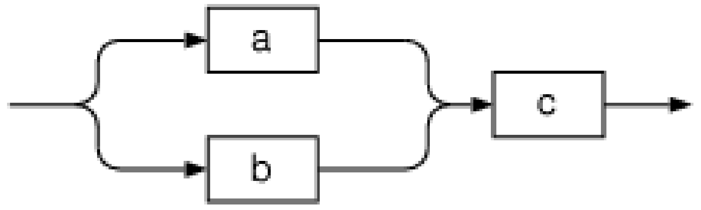


# Syntax Diagram

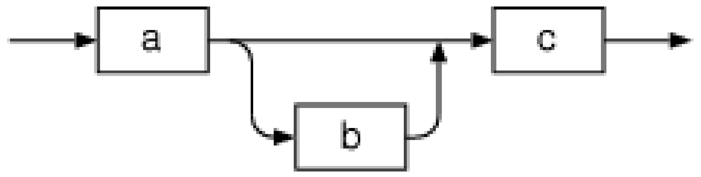
a | b c



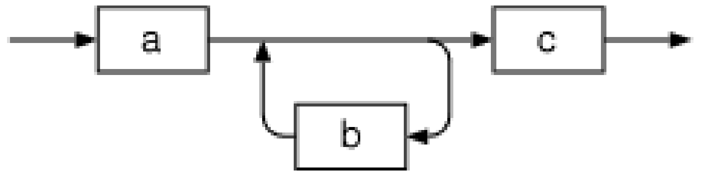
(a | b) c

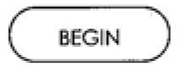
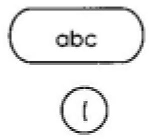
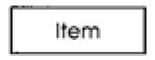
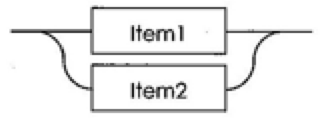
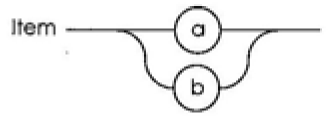
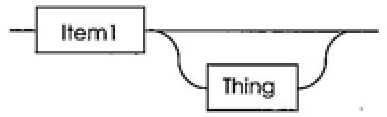
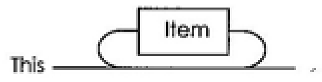
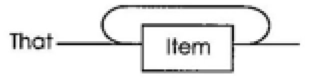
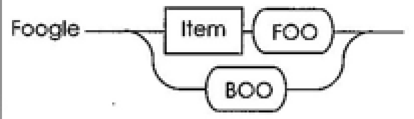


a [b] c

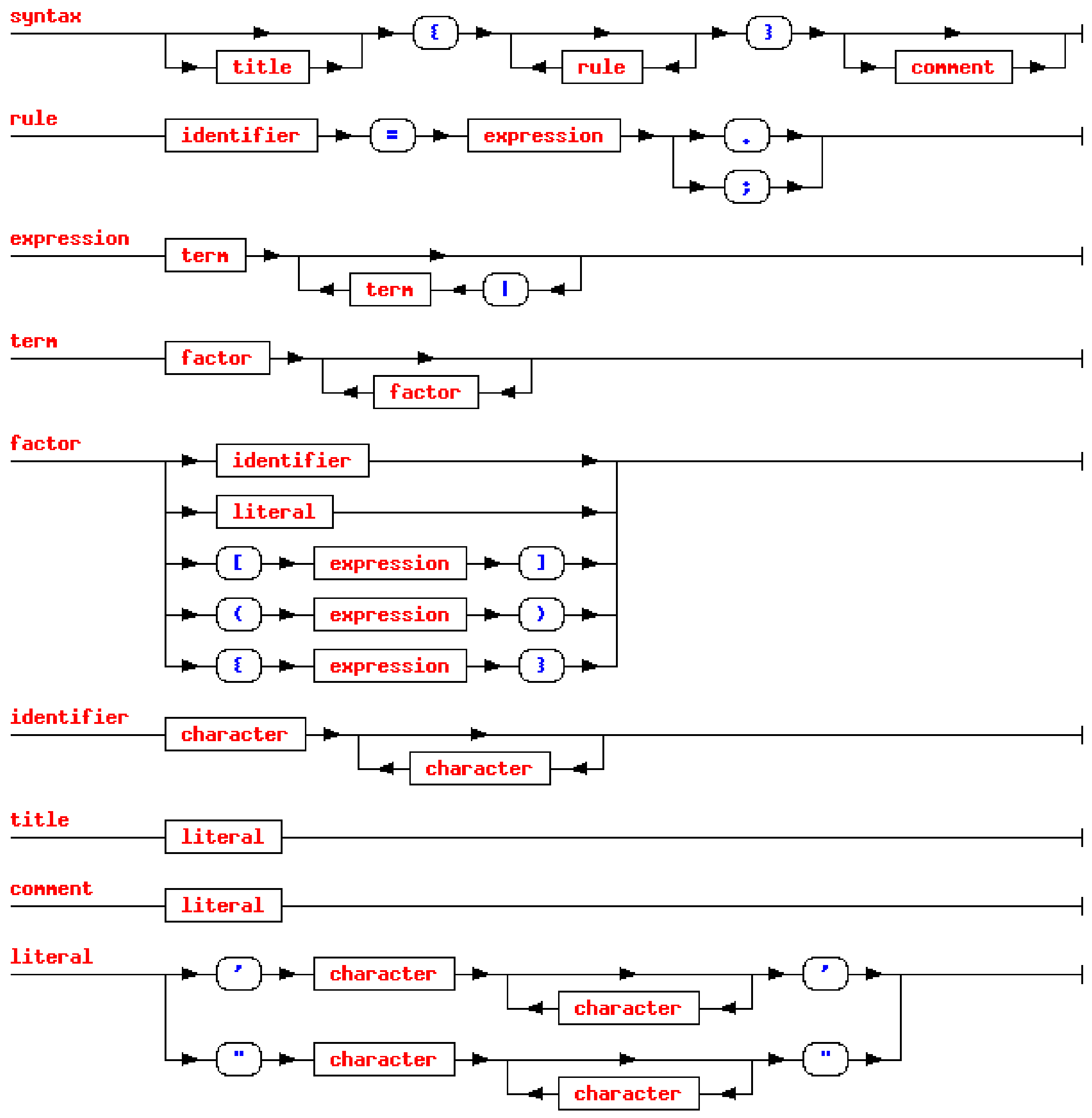


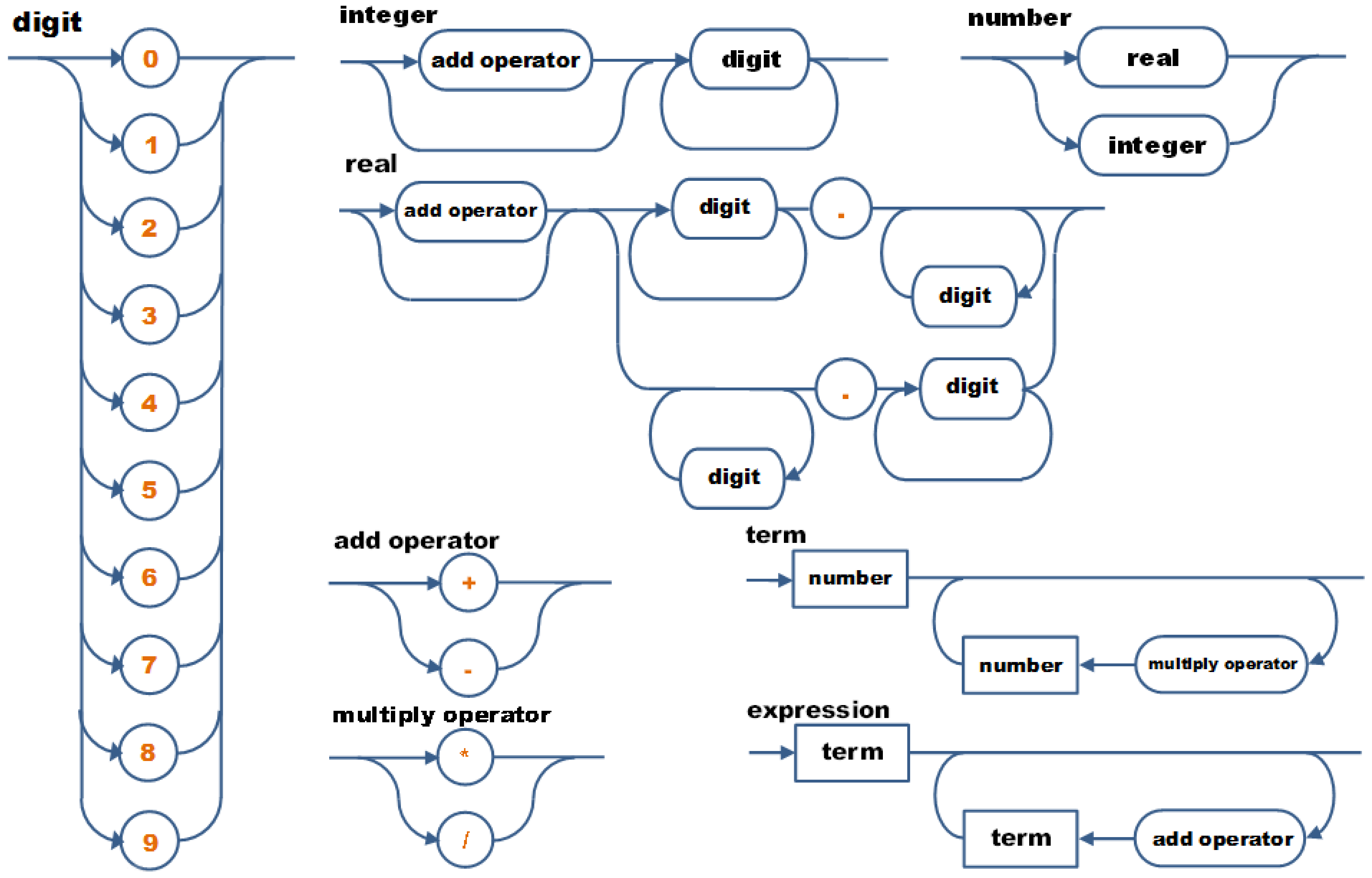
a {b} c



| Interpretation | BNF and EBNF example | Railroad Diagram example |
|---|---|---|
| Terminal Symbol for a reserved word. BEGIN is a reserved word. | BEGIN<br>BEGIN |  |
| Terminal symbol for a literal. The characters abc are written as they appear. Quotes are used to enclose symbols used by the metalanguage. | abc<br>(<br>abc<br>"(" |  |
| Non-terminal symbol. Item is defined elsewhere. | <Item><br><Item> |  |
| "or" a choice between two alternatives. Either Item1 or Item2. | <Item1> \| <Item2><br><Item1> \| <Item2> |  |
| "is defined as".Item can take the value a or b | Item::=a \| b<br>Item=a \| b |  |
| Optional part. Item followed optionally by a Thing. | <Item> \| <Item><Thing><br><Item>[<Thing>] |  |
| Possible repetition. This is an Item repeated zero or more times. | This::=<This><item> \| <Item> \| ""<br>This={<Item>} |  |
| Repetition. That is an Item repeated one or more times. | That::=<That><item> \| <Item><br>That=<Item>{<Item>} |  |
| Grouping. A Foogle is an Item followed by the reserved word FOO or it is the reserved word BOO. | Foo::=<Item>FOO<br>Foogle::=<Foo> \| BOO<br>Foogle=(<Item>FOO) \| BOO |  |

EBNF defined in itself




digit
0
1
2
3
4
5
6
7
8
9
integer
add operator
digit
number
real
integer
real
add operator
digit
.
digit
.
digit
digit
add operator
+
-
multiply operator
*
/
term
number
number
multiply operator
expression
term
term
add operator

# EBNF and Regular Expression

LECTURE 8

## Regular Expressions

- A sequence of characters that forms a search pattern
- Mainly used in pattern matching with strings
- Some PLs have built-in support for regular expressions and some use a standard library

- Implementations of regular expression functionality is often called a regular expression engine

## Regular Expressions

- Regular expression is used to represent the information required by the lexical analyzer
- ***Regular Expression Definitions:*** The rules of a language **L(E)** defined over the alphabet of the language is expressed using regular expression **E**.
  - ***Alternation:*** If **a** and **b** are regular expressions, then **(a+b)** is also a regular expression.
  - ***Concatenation (or Sequencing):*** If **a** and **b** are regular expressions, then **(a.b)** is also a regular expression.
  - ***Kleene Closure:*** If **a** is a regular expression, then **$a^*$** means zero or more representation of **a**.
  - ***Positive Closure:*** If **a** is a regular expression, then **$a^+$** means one or more of the representation of **a.**
  - ***Empty:*** Empty expressions are those with no strings.
  - ***Atom:*** Atoms indicate that there is only one string in the expression.

## Regexp Syntax

| | |
|---|---|
| `.` | Matches any character (except a newline, usually). |
| `*` | Matches 0 or more repetitions of the preceding sub-regexp (greedy). |
| `+` | Matches 1 or more repetitions of the preceding sub-regexp (greedy). |
| `?` | Matches 0 or 1 repetitions of the preceding sub-regexp (greedy). |
| `{m}` | Matches exactly *m* repetitions of the previous sub-regexp |
| `{m,n}` | Matches from *m* to *n* repetitions of the preceding sub-regexp (greedy). |

# Regexp Syntax

| | |
|---|---|
| \ | Escape character. |
| [] | Character class. Used to indicate a set of characters. `[sc]` will match ‘s’ or ’c’ `[a-z]` will match all characters between a and z `[^sc]` will match any character except ‘s’ and ‘c’ |
| \| | Alternation (or) |
| (...) | Match group. Allows recalling whatever was matched inside the parentheses later. |
| ^ | Start of line. |
| $ | End of line. |

## Extract emails

Given the following myfile.txt, extract things that look sort-of like email addresses:

```
<html>
...
<a href="mailto:bruce@gmail.com">send mail to bruce</a>
...
<a href="mailto:lee@yahoo.com">send mail to lee</a>
...
</html>
```

## Solution: extract email addresses

Extract all email addresses:

```
$ grep -E -o '[a-zA-Z0-9]+@[a-zA-Z0-9]+\.[a-zA-Z0-9]+' myfile.txt
bruce@gmail.com
lee@yahoo.com
```

Extract all domains of the email addresses:

```
$ grep -E -o '[a-zA-Z0-9]+@[a-zA-Z0-9]+\.[a-zA-Z0-9]+' myfile.txt \
 | sed -re 's/.*@([a-zA-Z0-9]+\.[a-zA-Z0-9]+)/\1/'
gmail.com
yahoo.com
```

## Example: extract phone numbers

```
$ cat phone_numbers.txt
Tal: 04-8294342, room 198
Dan: 04 8298888 room 745
Chen: 0523682930, room 002
Eugene: 97243453455, room 789

$ grep -E -o "(972[0-9]|[0-9]{2,3})[- ]?[0-9]{7}"
phone_numbers.txt
04-8294342
04 8298888
0523682930
97243453455
```

## Regex CSV Tokenizer

```
String [] tokens = lineBuffer.split ("\\s*,\\s*");

for (int i = 0; i < tokens.length; i++) {
     System.out.println ("" + i + ": " + tokens [i]);
}
```

# Qualified Tokens by BNF (BNF, EBNR, SDD, Regex)

LECTURE 9

(.*) Tokenization with Regular Expressions

- We want to tokenize some non-words like USA, $22.50
- This can be done by using Regular Expression patterns

## Regular Expression

- BNF, EBNF, SDD are for the design of the language.
- Regular Expression (Regex) is used for Tokenization, Parsing and many stages in compilation and natural language processing.
- Python libraries: (re: regex page, nltk: natural language tool-kit)