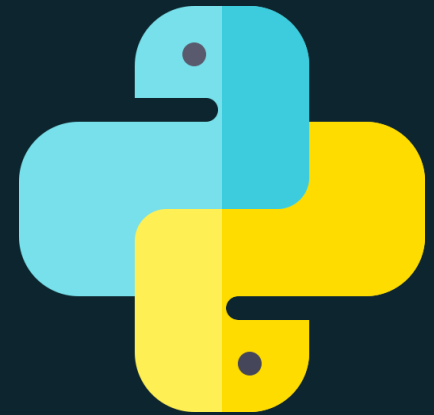


Brief Python

Python Course for Programmers



Learn Python Language for Data Science

CHAPTER 6: TOKENIZATION

DR. ERIC CHOU

IEEE SENIOR MEMBER



Objectives

- Data Cleansing
- Tokenization
- Counting Tokens
- Building of Symbol Table
- BENF and Syntax Diagram



Data Cleansing

Token Preprocessing

LECTURE 1



Step 1: Remove Non-Letter Characters

Demo Program: `file_remove1.py`

Goal: read all words from a file. Remove the white space characters, punctuation marks, numbers, and “’s” possessive specifier ‘s.

1. Read the whole file into a text string.
2. remove ‘s
3. remove non-letter characters.
4. trim white space characters.

```
f = open("usdeclar.txt", "r") # file_remove1.py
① text = f.read()
② text = text.replace("'s", " ") # remove all 's
tokens = text.split()
for i in range(len(tokens)):
    new_token = tokens[i]
    for j in range(len(new_token)):
        ch = new_token[j]
        if not (ch.isalpha()):
            # must assign the result back to new_token
            ③ new_token = new_token.replace(ch, " ")
            ④ tokens[i] = new_token.strip()
count = 0 # this part just to print out the words
for token in tokens:
    if (count % 20 == 0): print(token, end="")
    elif (count % 20 == 19): print(" "+token)
    else: print(" "+token, end=" ")
    count = count + 1
f.close()
```



Step 2: Remove all the Empty Tokens

Demo Program: `file_remove2.py`

Goal: remove all the tokens that is either " " or length == 0.

1. strip all whitespace again (there might be spaces created in step 1)
2. convert all words to lower case. [`str.lower()`]
3. append only non-empty and non-space strings to a new word list.

```
f = open("usdeclar.txt", "r")
text = f.read()
text = text.replace("'s", " ")    # remove all 's
tokens = text.split()
for i in range(len(tokens)):
    new_token = tokens[i]
    for j in range(len(new_token)):
        ch = new_token[j]
        if not (ch.isalpha()):
            new_token = new_token.replace(ch, " ")    # must assign the result back to new_token
    tokens[i] = new_token.strip()
# remove empty or meaningless tokens
tokenb = []
for token in tokens:
    token = token.strip()
    token = token.lower()
    if (token != " " and len(token) != 0): tokenb.append(token)
tokens = tokenb
# print out part
count = 0
for token in tokens:
    if (count % 20 == 0): print(token, end="")
    elif (count % 20 == 19): print(" "+token)
    else: print(" "+token, end=" ")
    count = count + 1
print("\n")
print("Word count in File "+ "usdeclar.txt is "+str(len(tokens)))
f.close()
```



Result at this Stage

- Each token is a lower-case string.
- Each token contains no symbol.
- Each token has no number.
- The length of the list tokens is the number of words in the source file.



Text Processing After File Read-in

LECTURE 1



Step 3: Calculate the count of individual words

Demo Program: [filecount.py](#)

- Convert the tokens list to a non-repeating word list.
 1. If a word has never shown up, add the word into the non-repeating word list.
 2. Sort the word list according to the alphabetical order

```
f = open("usdeclar.txt", "r")
text = f.read()
text = text.replace("'s", " ")    # remove all 's
tokens = text.split()           # using regular expression by one or more spaces
for i in range(len(tokens)):
    new_token = tokens[i]
    for j in range(len(new_token)):
        ch = new_token[j]
        if not (ch.isalpha()):
            new_token = new_token.replace(ch, " ")    # must assign the result back
to new_token
    tokens[i] = new_token.strip()

# remove empty or meaningless tokens
tokenb = []
for token in tokens:
    token = token.strip()
    token = token.lower()
    if (token != " " and len(token) != 0): tokenb.append(token)
tokens = tokenb
```

1 *# create non-repeating word list*

```
word_list = []  
for i in range(len(tokens)):  
    found = False  
    j=0  
    while not found and j<len(word_list):  
        if (tokens[i]==word_list[j]):  
            found = True  
            j = j + 1  
    if not found:  
        word_list.append(tokens[i])
```

*# sort the word_list***2**

```
word_list.sort()  
# print out part  
tokens = word_list  
count = 0  
for token in tokens:  
    print(token)  
    count = count + 1  
print("\n")  
print("Word count in File "+"usdeclar.txt is "+str(len(tokens)))  
f.close()
```



Create Occurrence Count List

LECTURE 1



Step 4: Generating the Occurrent Count List

Demo Program: [filecount2.py](#)

1. When a word is first found, add the word into the `word_list` and append a one into the occurrence count list.
2. When a word is found later, increase the occurrence count number for the token.

Creating the Occurrence List

```
# create non-repeating word list
word_list = []
occurrence = []
for i in range(len(tokens)) :
    found = False
    j=0
    while not found and j<len(word_list):
        if (tokens[i]==word_list[j]):
            ❷ occurrence[j] += 1
            found = True
        j = j + 1
    if not found:
        word_list.append(tokens[i])
        ❶ occurrence.append(1)
```



Step 4: Sorting by Occurrence

1. Repeat number of Tokens times, each time find the token with maximum occurrence time.
 - Each time, remove the token and its occurrence count with maximum occurrence from word_list
 - Append the token to wlist
 - Append the occurrence count to olist
2. Print out the token and occurrence pair

Sorting the Word List by Occurrence Count

```
leng = len(word_list)
wlist = []
olist = []
for i in range(leng):
    ❶ max = -1
    ind = 0
    for j in range(len(occurrence)):
        if (occurrence[j] > max):
            max = occurrence[j]
            ind = j
    wlist.append(word_list[ind])
    olist.append(occurrence[ind])
    del(word_list[ind])
    del(occurrence[ind])
```

Print out the Token and Occurrence Count

```
# print out part
tokens      = wlist
occurrence  = olist
for i in range(len(tokens)):
    2 t = "%-20s - %d" % (tokens[i], occurrence[i])
      print(t)
```



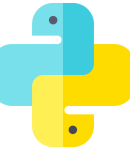
Simple and Space Efficient Code

LECTURE 1



Strip of Right-Hand-Side “\n”

- **file.readlines()** operation will read in a list of line strings. Each line has a “\n” newline mark.
- Therefore, we can remove the newline mark by
for line in open(file_name):
 process(line.rstrip('\n'))
or
for line in open(file_name):
 line = line.rstrip('\n') # re-bind line (also rebind on next loop iteration)
 process(line)



Python is Immutable

All Returned Result must be Assigned to a String Reference Variable

CORRECT

```
for line in open(file_name):  
    line = line.rstrip('\n')  
    process(line)
```

INCORRECT

```
for line in open(file_name):  
    line.rstrip('\n') # does NOTHING!  
    process(line)
```



List Element Creation by Iterator

```
line_list = [line.rstrip('\n') for line in open(file_name)]
```



Alias for list of a File

- With the open context manager, we would write the above code

- fragments as

```
with open(file_name) as open_file:
```

```
    for line in open_file:
```

```
        process(line.rstrip('\n'))
```

- or

```
with open(file_name) as open_file:
```

```
    line_list = [line.rstrip('\n') for line in open_file]
```

Alias for the `open(file_name)` as an iterable object





Simple and Space Efficient Code for `readlines()`

- If we wanted to call `.readlines()` and process every string in the file (without the `'\n'` characters at the end) we would write
`open(file_name).readlines()` is also iterable
`for line in open(file_name).readlines():`
 `process(line.rstrip('\n'))`
- or
 `for line in open(file_name).readlines():`
 `line = line.rstrip('\n') # re-bind line (also rebind on next loop iteration)`
 `process(line)`
- or (by list creator by iterator)
 `line_list = [line.rstrip('\n') for line in open(file_name).readlines()]`



List of Tokens Generation `read().split()`

- calling `open_file.read()` returns the string

`'Line 1\nLine 2\nLine 3\n'`

- We can split this string into a list of strings by calling the `.split` method.

`line_list = open(file_name).read().split('\n')`

- and this code is simpler than what we have seen before, which is equivalent to

`line_list = [line.rstrip('\n') for line in open(file_name)]`



`.read().split('\n')`

Generating `line_list` not tokens

The `.read().split('\n')` code above is even less space efficient than the simpler comprehension, because it stores both the entire file string and a list of all the lines in the file at the same time; the comprehension stores the list of all the lines in the file, but not a string whose contents is the entire file itself.

```
for line in open(file_name).read().split('\n'):
    process(line)
```



Summary

There is little to be gained by reading files by calling the **.readlines()** or the **.read()** method. Iterate directly over the "open" file with a standard for loop or a for loop in a comprehension.



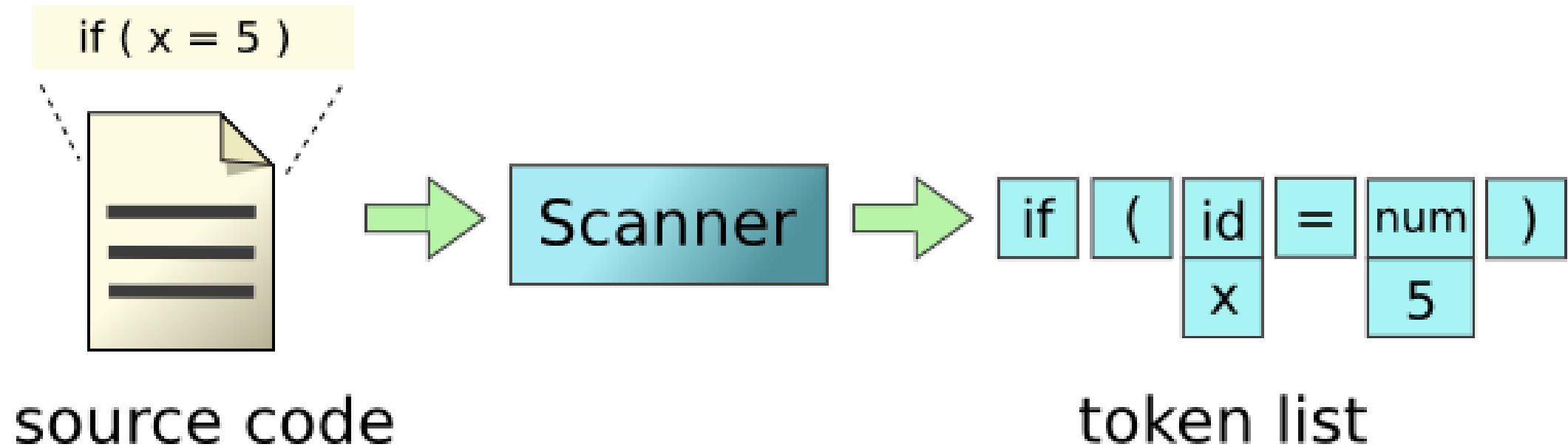
Parsing of Tokens

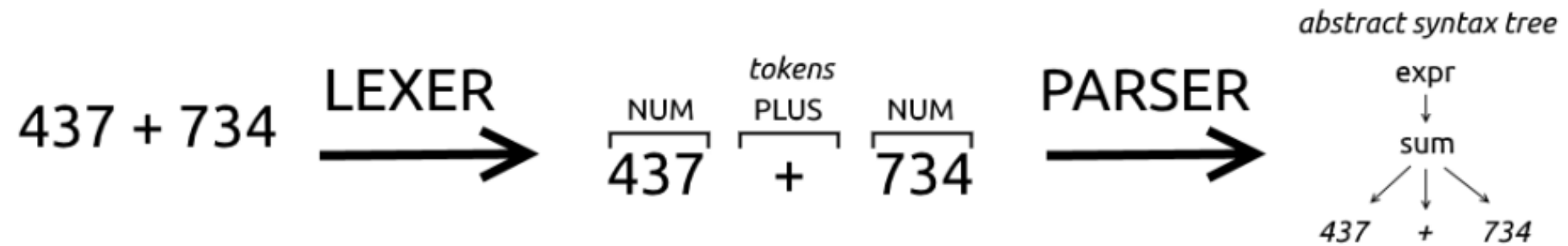
LECTURE 1




Lexical Analysis

From Source File to Token List





	Scanning	Parsing
Task	determining the structure of tokens	determining the syntax or structure of a program
Describing Tools	regular expression	context-free grammar 
Algorithmic Method	represent by DFA	top-down parsing bottom-up parsing
Result Data Structure	liner structure	parser tree or syntax tree, they are recursive



Reading Files and Parsing their Contents

Step 1:

```
for line in open(file_name):  
    process( int(line.rstrip('\n')) )
```

Some text files contain lines that store other types or mixed-types of information. Suppose that we wanted to read a text file that stored strings representing numbers (one number per line).



Parsing their Contents

Step 2: Simple Parsing (Pre-condition: Each line is a record.)

Here we are assuming process takes an integer value as an argument.

In some files each line is a "record": a fixed number of fields of values, with possibly different types, separated by some special character (often a space or punctuation character like a comma or colon). To process each record in a file, we must

1. read its line
2. separate its fields of values (still each value is a string)
3. call a conversion function for each string to get its value



Preprocessing for Parsing

LECTURE 1



zip Function

zip() Parameters

The zip() function takes zero or more iterables (lists)

Return Value from zip()

The zip() function returns an a of tuples based on the iterable object.

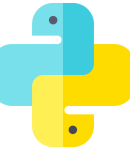


zip Function

Return Value from zip()

The zip() function returns an iterator of tuples based on the iterable object.

- If no parameters are passed, zip() returns an **empty iterator**
- If a single iterable is passed, zip() returns an **iterator of 1-tuples**.
Meaning, the number of elements in each tuple is 1.
- If multiple iterables are passed, ith tuple contains ith Suppose, two iterables are passed; one iterable containing 3 and other containing 5 elements. Then, the returned iterator has 3 tuples. It's because iterator stops when shortest iterable is exhausted.

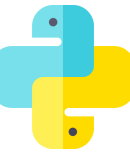


Demo Program: zip.py

- Show how to combine many **iterables** (data collections like lists) into iterables of tuples.
- Purpose: Combine list of same records into one list. Each tuple will represent one record.

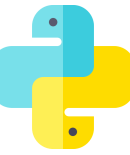
```
1 numberList = [1, 2, 3]
2 strList = ['one', 'two', 'three']
3
4 # No iterables are passed
5 result = zip()
6
7 # Converting itertor to list
8 resultList = list(result)
9 print(resultList)
10
11 # Two iterables are passed
12 result = zip(numberList, strList)
13
14 # Converting itertor to set
15 resultSet = set(result)
16 print(resultSet)
```

Run zip	
▶	C:\Python\Python36\python.exe "C:\Eric_Chou
■	[]
	{(1, 'one'), (3, 'three'), (2, 'two')}



Demo Program: filecount3.py

- Rewrite the filecount2.py to combine the word_list and occurrence list into one list of records (each word record has two tuples: one for word string and one for integer occurrence count of that word.)
- We didn't use the default **zip** function. We used a custom designed zip function.



Custom Designed zip Function

```
def zip(a, b):  
    zip_list = []  
    for i in range(len(a)):  
        zip_list.append([a[i], b[i]])  
    return zip_list
```



```
f = open("usdeclar.txt", "r") # filecount3.py (part 1: tokenization)
text = f.read()
text = text.replace("'s", " ") # remove all 's
tokens = text.split() # using regular expression by one or more spaces
for i in range(len(tokens)):
    new_token = tokens[i]
    for j in range(len(new_token)):
        ch = new_token[j]
        if not (ch.isalpha()):
            new_token = new_token.replace(ch, " ")
            # must assign the result back to new_token
    tokens[i] = new_token.strip()

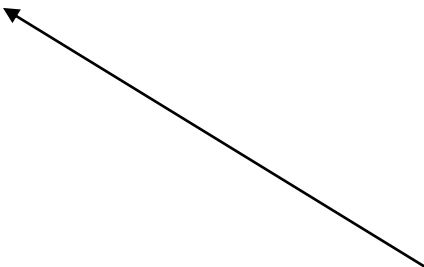
# remove empty or meaningless tokens
tokenb = []
for token in tokens:
    token = token.strip()
    token = token.lower()
    if (token != " " and len(token) != 0): tokenb.append(token)
tokens = tokenb
```

filecount3.py Part 2: create non-repeating word list

```
word_list = []
occurrence = []
for i in range(len(tokens)):
    found = False
    j=0
    while not found and j<len(word_list):
        if (tokens[i]==word_list[j]):
            occurrence[j] += 1
            found = True
        j = j + 1
    if not found:
        word_list.append(tokens[i])
        occurrence.append(1)
```

filecount3.py Part 3: selection sort

```
words = zip(word_list, occurrence)
print(words)
leng = len(words)
words2 = []
for i in range(leng):
    max = -1
    ind = 0
    for j in range(len(words)):
        if (words[j][1] > max):
            max = words[j][1]
            ind = j
    words2.append(words[ind])
    del (words[ind])
```



Zip function groups the word and its associated occurrence into a list (tuple). This makes sorting easier.

```
# filecount3.py Part 4: listing words in order
```

```
words = words2
```

```
# print out part
```

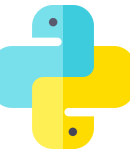
```
for i in range(len(words)):
```

```
    t = "%-20s - %d" % (words[i][0], words[i][1])
```

```
    print(t)
```

```
print("\n")
```

```
print("Word count in File "+ "usdeclar.txt is "+str(len(words)))
```



Summary for Pre-Processing

1. Retrieve proper information before parsing the data.
2. Remove un-wanted symbols.
3. Data File format to be processed: Plain Text file (TXT), CSV (Comma Separated Values) file, JSON file, JSONP file, XML file, XLS (Excel), XLSX (Excel new).

On-line Converter (CSV to JSON): <http://www.csvjson.com/csv2json>

Free File Format Converter: <https://www.lifewire.com/free-file-converter-software-and-online-services-2626121>



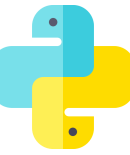
CSV (Comma Separated Values) Files

LECTURE 1



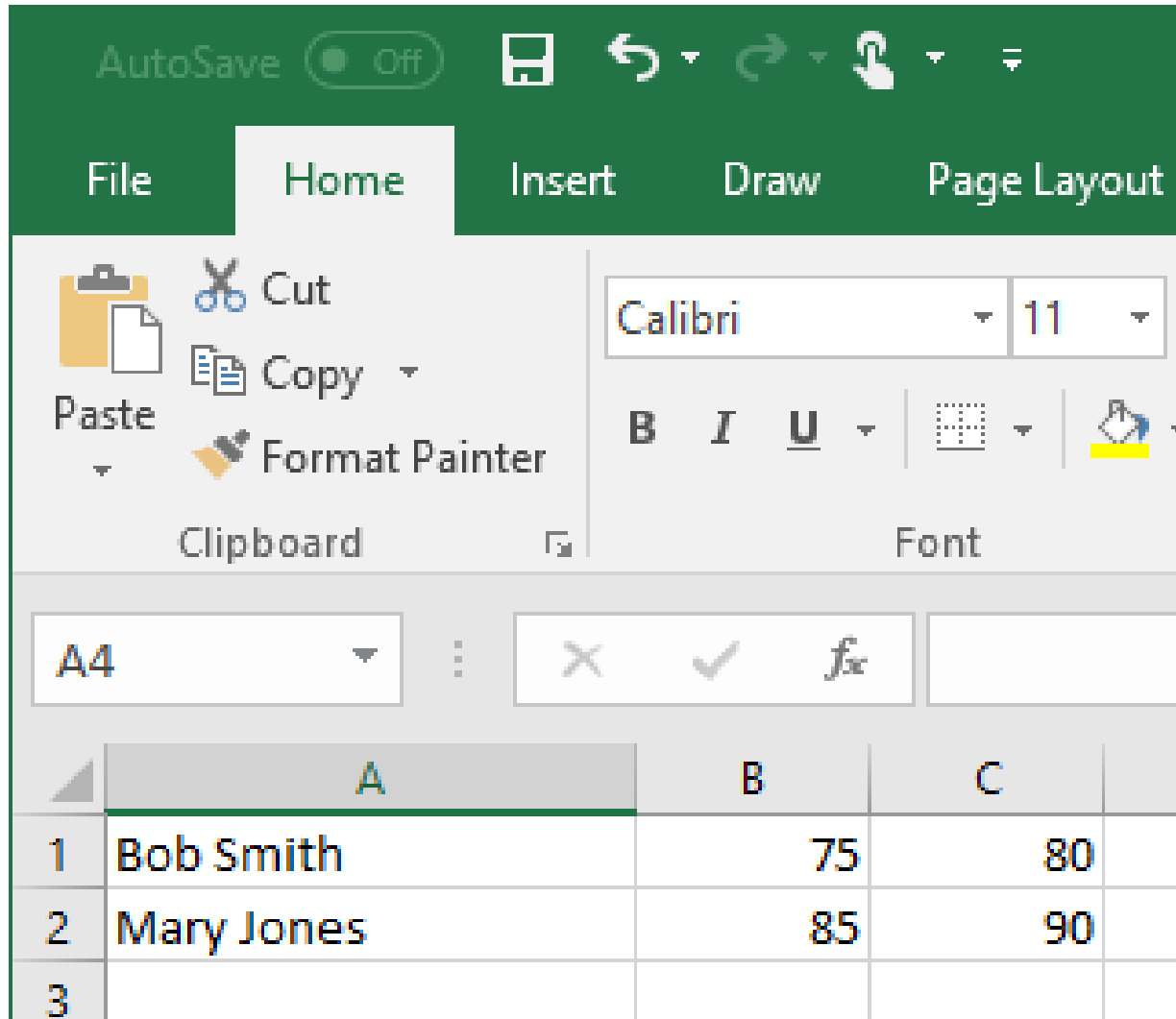
What is a CSV File?

- A file with the **CSV** file extension is a **Comma Separated Values** file. All CSV files are plain text, can contain numbers and letters only, and structure the data contained within them in a tabular, or table, form.
- Files of this format are generally used to exchange data, usually when there's a large amount, between different applications. Database programs, analytical software, and other applications that store massive amounts of information (like contacts and customer data), will usually support the CSV format.
- A Comma Separated Values file might sometimes be referred to as a Character Separated Values or Comma Delimited file but regardless of how someone says it, they're talking about the same CSV format.



How To Open a CSV File?

- **Spreadsheet** software (MS-Excel) is generally used to open and edit CSV files, such as the free **OpenOffice Calc** or **Kingsoft** Spreadsheets. Spreadsheet tools are great for CSV files because the data contained is usually going to be filtered or manipulated in some way after opening.
- **Text Editor**: [Notepad++](#) or [GenScriber](#) to open CSV files but large ones will be very difficult to work with in these types of programs. [Open Freely](#) is another alternative but with the same problem with larger CSVs.
- Considering the number of programs out there that support structured, text-based data like **CSV**, you may have more than one program installed that can open these types of files.



Generation of CSV by MS-Excel

Demo Program:
Student.XLSX

GO MS-EXCEL!!!



Info

New

Open

Save

Save As

Print

Share

Export

Publish

Close

Account 

Feedback

Options

Export



Create PDF/XPS Document



Change File Type

Change File Type

Workbook File Types



Workbook
Uses the Excel Spreadsheet format



Excel 97-2003 Workbook
Uses the Excel 97-2003 Spreadsheet format



OpenDocument Spreadsheet
Uses the OpenDocument Spreadsheet format



Template
Starting point for new spreadsheets



Macro-Enabled Workbook
Macro enabled spreadsheet



Binary Workbook
Optimized for fast loading and saving

Other File Types



Text (Tab delimited)
Text format separated by tabs



CSV (Comma delimited)
Text format separated by commas



Formatted Text (Space delimited)
Text format separated by spaces



Save as Another File Type

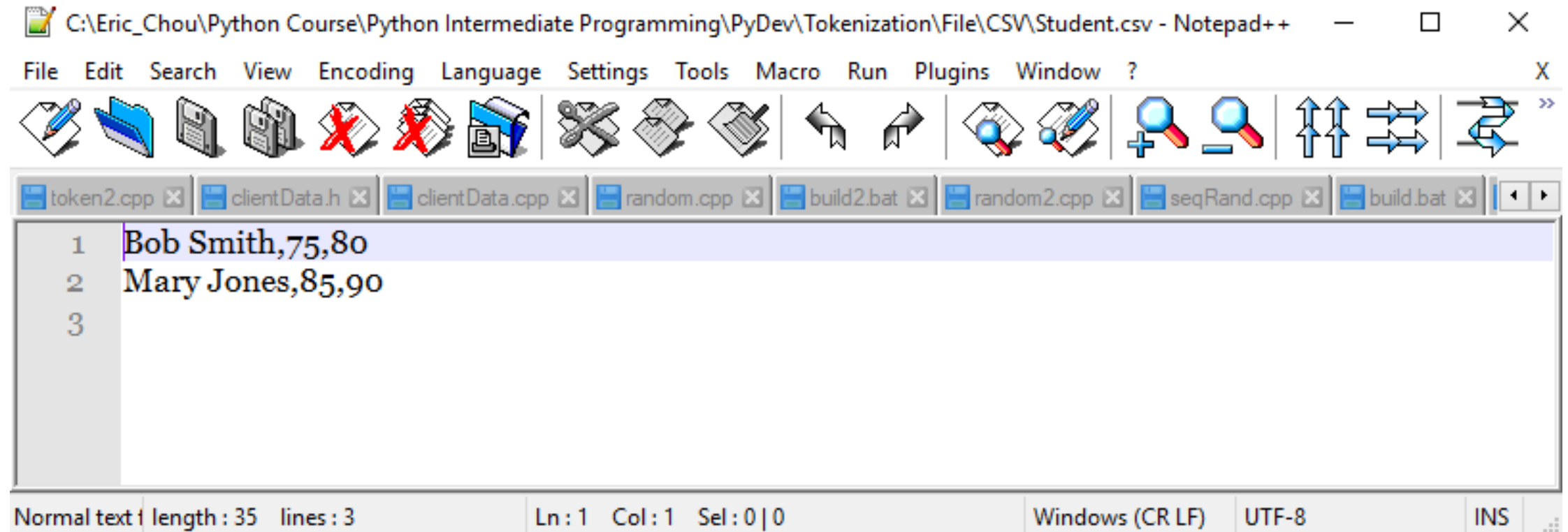


Save As



Pick This One

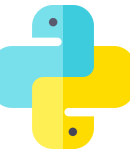
Open CVS File with Notepad++





Parsing CSV Files

LECTURE 1



parse_lines function

Demo Program: parse_lines.py

```
# parse_lines.py
# read a csv file and print the student record to screen.
#
def parse_lines(open_file, sep, conversions):
    for line in open_file:
        yield [conv(item) for conv, item in
               zip(conversions, line.rstrip('\n').split(sep))]

for fields in parse_lines(open("Student.csv"), ',', (str, int, int)):
    print(fields[0], (fields[1]+fields[2])/2)
```

```
Run parse_lines
C:\Python\Python36\python.exe
Bob Smith 77.5
Mary Jones 87.5
```



parse_lines function

Demo Program: `parse_lines.py`

- The module contains the **parse_lines** function that easily supports reading records from files (similarly to how lines are read from "open" files).
- **sep** is a special character used to separate the fields in the record;
- **conversions** is a tuple of function objects: they are applied in sequence to the string values extracted from the separated fields.
- When we iterate over a call to **parse_lines** (similar to iterating over a call to "open"), the index variable is bound to a list of the values of the fields in the record.



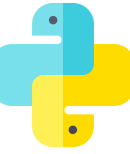
zip function

conversions: (str,int,int)

a line of data fields: (Bob Smith,75,80)

zip(conversions, ('Bob Smith', 75, 80) →
[(str, 'Bob Smith', (int, 75), (int, 80))] # zip object

Apply **str("Bob Smith"), int(75), int(80)**



Apply str("Bob Smith"), int(75), int(80)

```
yield [conv(item) for conv, item in  
       zip(conversions, line.rstrip('\n').split(sep))]
```

yield
return generator

generator is a kind of iterable which is only temporary iterable. It can only be used once (on the fly).



Iterables

- When you create a list, you can read its items one by one. Reading its items one by one is called **iteration**:

```
>>> mylist = [1, 2, 3]
>>> for i in mylist:
...     print(i)
1
2
3
```

- **mylist** is an iterable.



Iterables

- When you use a list comprehension, you create a list, and so an iterable:

```
>>> mylist = [x*x for x in range(3)]  
>>> for i in mylist:  
...     print(i)  
0  
1  
4
```

- Everything you can use "for... in..." on is an **iterable**; **lists**, **strings**, **files**...



Generators

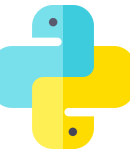
- Generators are iterators, a kind of iterable you can only iterate over once. Generators do not store all the values in memory, they generate the values **on the fly**:

```
>>> mygenerator = (x*x for x in range(3))
>>> for i in mygenerator:
...     print(i)
0
1
4
```



Generators

- It is just the same except you used `()` instead of `[]`.
- BUT, you cannot perform `for i in mygenerator` a second time since generators can only be used once: they calculate 0, then forget about it and calculate 1, and end calculating 4, one by one.



Yield

- yield is a keyword that is used like **return**, except the function will return a generator.

```
>>> def createGenerator():
...     mylist = range(3)
...     for i in mylist:
...         yield i*i ...
>>> mygenerator = createGenerator() # create a generator
>>> print(mygenerator) # mygenerator is an object!
<generator object createGenerator at 0xb7555c34>
>>> for i in mygenerator:
...     print(i)
0
1
4
```



Yield

Return as a whole, Yield on each iteration

- To master **yield**, you must understand that when you call the function, the code you have written in the function body does not run. The function only returns the generator object, this is a bit tricky.
- Then, your code will be run each time the for uses the generator.
- The first time the for calls the generator object created from your function, it will run the code in your function from the beginning until it hits yield, then it'll return the **first** value of the loop. Then, each other call will run the loop you have written in the function one more time, and return the next value, until there is no value to return.

```

# generator_examples.py
print("Iterable Example:")
mylist = [1, 2, 3]
for i in mylist:
    print(i)
print("\n")
print("List Creation by Iterator Example 2:")
mylist = [x*x for x in range(3)]
for i in mylist:
    print(i)
print("\n")
print("Generator Example:")
mygenerator = (x*x for x in range(3))
for i in mygenerator:
    print(i)
print("\n")
print("Yield Example:")
def createGenerator():
    mylist = range(3)
    for i in mylist:
        yield i*i
mygenerator = createGenerator() # create a generator
print(mygenerator) # mygenerator is an object!
for i in mygenerator:
    print(i)

```

Iterable Example:

```

1
2
3

```

List Creation by Iterator Example 2:

```

0
1
4

```

Generator Example:

```

0
1
4

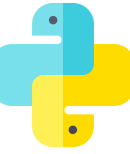
```

Yield Example:

```

<generator object createGenerator at 0x0000023B39B25BF8>
0
1
4

```



Understand parse_lines.py

- For example, the following file contains fields of a name (str) followed by two test scores (ints) all separated by commas.

Bob Smith,75,80

Mary Jones,85,90

- We could read this file and print out the names of each student and their average test score by

```
for fields in parse_lines( open(file_name), ',' , (str,int,int) ):  
    print(fields[0], (fields[1]+fields[2])/2)
```

- Here fields is repeatedly bound to a 3-list containing a name (str) followed by two test scores (ints). fields is first bound to ['Bob Smith', 75, 80] and then to ['Mary Jones', 85, 90].

Understand parse_lines.py

Note:

1. if we specified conversions as (str, int) it would return the 2-lists ['Bob Smith', 75] followed by ['Mary Jones', 85] (because looping over a zip stops when one of its arguments runs out of values: here each line contains more field values than conversion functions). Accessing fields[2] in the code above would raise an **IndexError** exception.
 2. Likewise (because looping over a zip stops when one of its arguments runs out of values), if the a line contains a name and 3 integer values, only the name and first two integers would be returned in the 3-list: the line
 Paul White,80,75,85
 returns only the 3-list ['Paul White', 80, 75]
- So parse_lines would not raise any exceptions in the code above; instead it incorrectly reads the file contents with no warning.
 - We could define a more complicated parse_lines function that **checked** and immediately **raised** an **exception** if the number of separated field values in a record was not equal to the length of the tuple of conversion functions.



Simplification of parse_lines.py

LECTURE 1



Simplification by Unpacking

- A simpler way to write such code is to use multiple index variables and unpacking (as we do when we write: `for k,v in adict.items()`).

```
for name,test1,test2 in parse_lines(open(file_name),',',(str,int,int)):  
    print(name, (test1+test2)/2)
```

- With this for loop, the first error noted above would also raise an exception because there would not be three values to **unpack** into `name`, `test1`, and `test2`; the second error would again go unnoticed.



Demo Program: parse_lines2.py

```
# parse_lines2.py
```

```
# read a csv file and print the student record to screen.
```

```
#
```

```
def parse_lines(open_file, sep, conversions):
```

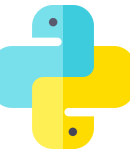
```
    for line in open_file:
```

```
        yield [conv(item) for conv,item in
```

```
                zip(conversions,line.rstrip('\n').split(sep))]
```

```
for name,test1,test2 in parse_lines(open("Student.csv"),',',(str,int,int)):
```

```
    print(name, (test1+test2)/2)
```

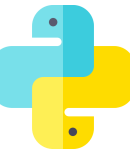


Colon Separated Data List

- Finally, note that besides using the standard conversion function(s) like `str` and `int`, we can define our own conversion function(s).
- For example, suppose that each record in the file specified a string name, some number of int quiz results separated by colons, and an int final exam, with these three fields (name, quizzes, final) separated by commas.
- Such a file might look like

Bob Smith,75:80,90

Mary Jones,85:90:77,85



Decoding Colon Separated Data_List

Here Bob took two quizzes but Mary took three. We could define

```
def quiz_list(scores):
```

```
    return [int(q) for q in scores.split(':')]
```

and then write

```
for name,quizzes,final in parse_lines(open(file_name),',',(str,quiz_list,int)):
```

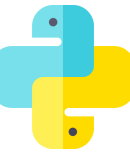
```
    print(name, sum(quizzes)/len(quizzes), final)
```

which would print

Bob Smith 77.5 90

Mary Jones 84.0 85

Note that 77.5 is $(75+90)/2$ and 84 is $(85+90+77)/3$.



Demo Program: parse_lines3.py

```
# parse_lines3.py
```

```
# read a csv file and print the student record to screen.
```

```
#
```

```
def parse_lines(open_file, sep, conversions):
```

```
    for line in open_file:
```

```
        yield [conv(item) for conv,item in
```

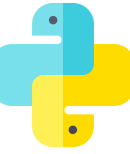
```
                zip(conversions,line.rstrip('\n').split(sep))]
```

```
def quiz_list(scores):
```

```
    return [int(q) for q in scores.split(':')]
```

```
for name,quizzes,final in parse_lines(open("Student.csv"),',',(str,quiz_list,int)):
```

```
    print(name, sum(quizzes)/len(quizzes), final)
```



Using Lambda Function

If we instead wrote

```
for fields in parse_lines(open(file_name),',',(str, quiz_list, int)):  
    print(fields)
```

it would print the 3-lists

```
['Bob Smith', [75, 80], 90]
```

```
['Mary Jones', [85, 90, 77], 85]
```

Of course, we can also use lambdas instead of named functions; below we have substituted a **lambda** for the **quiz_list** function.

```
for fields in parse_lines(open(file_name),',', (str, lambda scores : [int(q) for q in scores.split(':')], int)):  
    print(fields)
```




- The lambda expression is the function itself
 - Apply the expression to one or more parameters

`(λ (x) x * x * x) (4)`



```
# Python
(lambda x:x*x*x) (4)
```

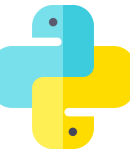
Formal Parameter
Returned Value
Actual Argument

`(λ (x,y) x * y) (8,7)`



```
# Python
(lambda x,y:x*y) (8,7)
```


Python Lambda Function



Demo Program: parse_lines4.py

```
# parse_lines4.py
# read a csv file and print the student record to screen.
#
def parse_lines(open_file, sep, conversions):
    for line in open_file:
        yield [conv(item) for conv,item in
               zip(conversions,line.rstrip('\n').split(sep))]

for fields in parse_lines(open("Student.csv"),",",(str, lambda scores : [int(q) for q in scores.split(":")],int)):
    print(fields)
```

```
Run  parse_lines4

▶ ↑ C:\Python\Python36\python.exe
■ ↓ ['Bob Smith', [75], 80]
|| ↕ ['Mary Jones', [85], 90]
```



File Read-in Styles

LECTURE 1



File Processing Styles

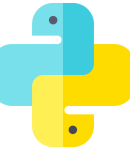
- Suppose that we want to process the lower case version of every word on every line (where the words on a line are separated by spaces) in a file named file.txt. Which of the following code fragments correctly does so? For those that don't, explain why they fail. For example, if the file contained the three lines:

See spot

See spot run

Run spot run

- it should process the following words in the following order:
'see', 'spot', 'see', 'spot', 'run', 'run', 'spot', 'run'.



Demo Program:

`spot.py` and `spot_error.py`

- Please check `spot_error.py` to see which examples will deliver the expected results and which examples are actually wrong.
- The `spot.py` is the corrected version.



Summary of the Chapter

1. Text files read-in before tokenization may need to remove white space symbol, punctuation marks, digits, and many other things.
2. Text can be read-in character by character, token by token, line by line or as a block.
3. Line of text or block of text can be split into a list of tokens.
4. Text file of .txt and .csv are used as example in this chapter.
5. Basic parsing by validating each data field in a record from a line of text has been demonstrated.



Token Classifications

LECTURE 1

Grammar

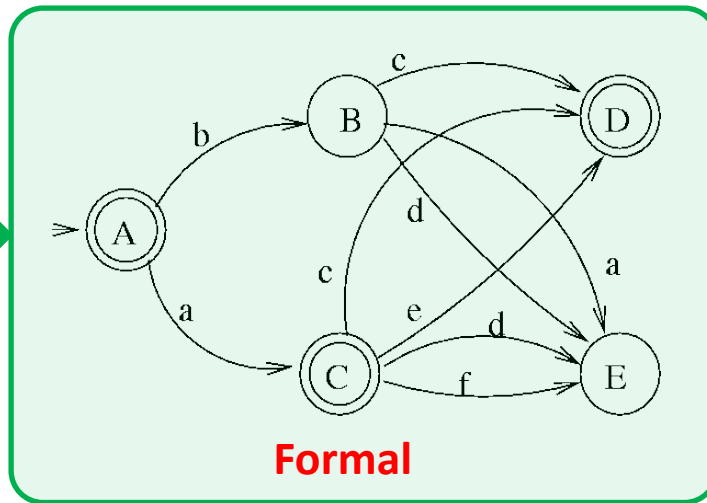
$$\begin{aligned} S &\rightarrow aS \mid bX \\ X &\rightarrow aX \mid bY \\ Y &\rightarrow aY \mid bZ \mid \Lambda \\ Z &\rightarrow aZ \mid \Lambda \end{aligned}$$

Formal

Language Input

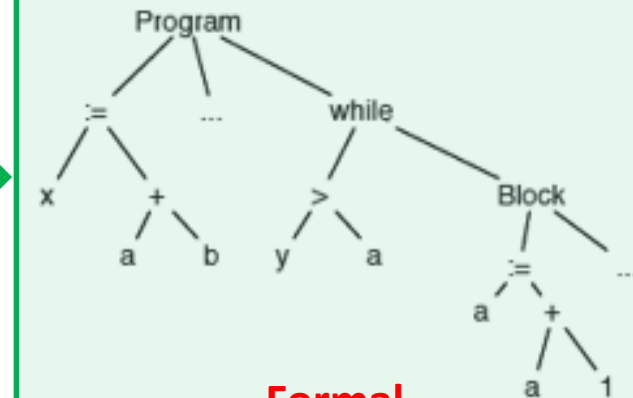
```
x := a + b;  
y := a * b;  
while (y > a) {  
  a := a + 1;  
  x := a + b  
}
```

Informal



State Machine

Validated Tokens Syntax Tree



Target Code

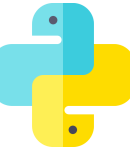
Machine Code
Assembly
Byte Code
Object Code

Informal

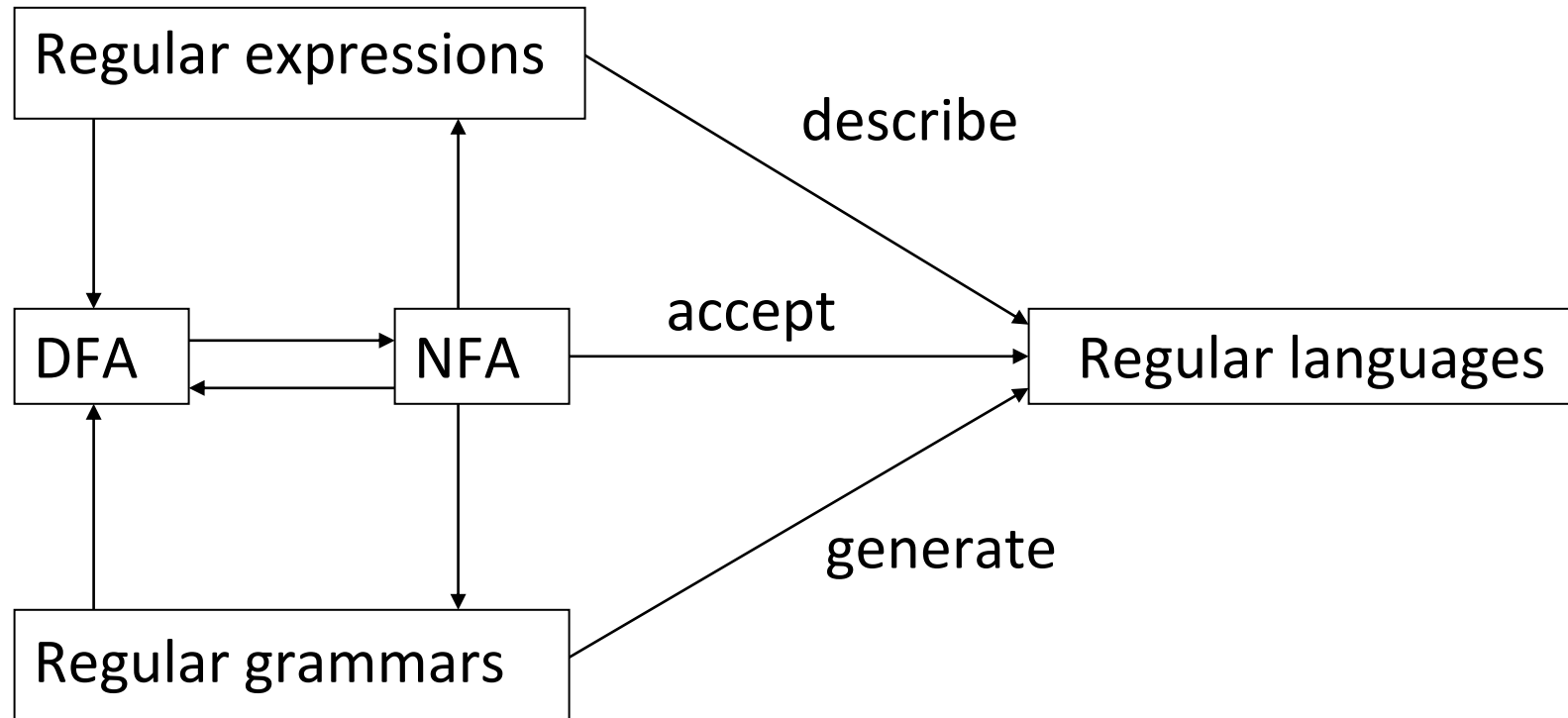


Expression, Grammar, Language

- **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

$digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
 $non_zero_digit \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
 $natural_number \rightarrow non_zero_digit\ digit^*$

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 Σ 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: $L_1^* = \{\lambda, 10, 1010, 101010, \dots\}$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)





Numerical constants accepted by a simple hand-held calculator

$number \longrightarrow integer \mid real$

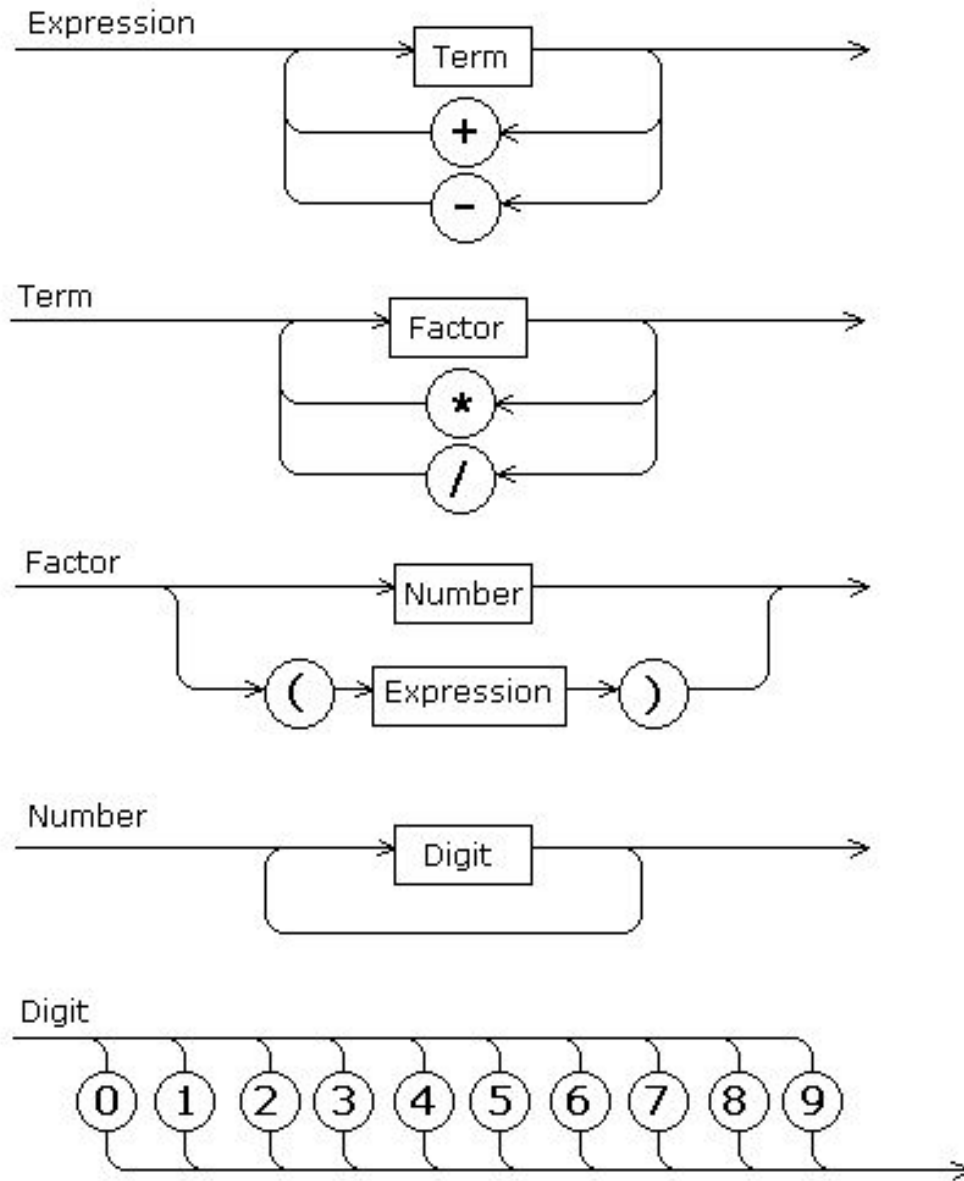
$integer \longrightarrow digit \, digit^*$

$real \longrightarrow integer \, exponent \mid decimal \, (\, exponent \mid \epsilon \,)$

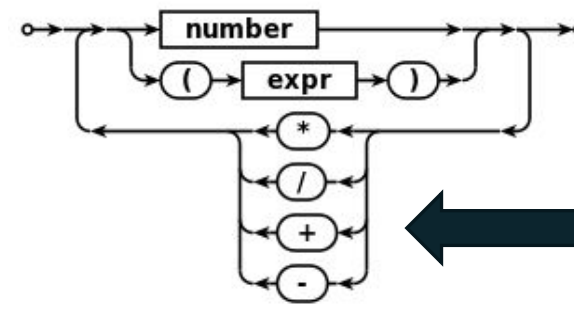
$decimal \longrightarrow digit^* \, (\, . \, digit \mid digit \, . \,) \, digit^*$

$exponent \longrightarrow (\, e \mid E \,) \, (\, + \mid - \mid \epsilon \,) \, integer$

$digit \longrightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

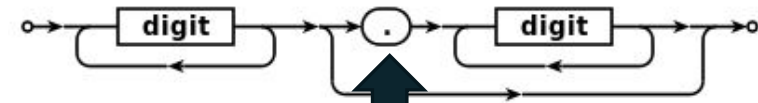


expr:

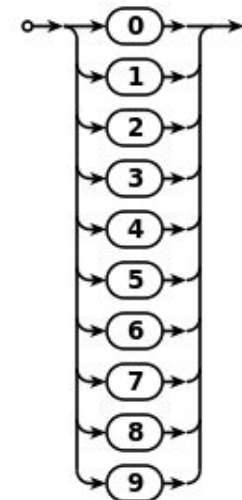


No precedence

number:



digit:



With decimal point for floating point and integer



Backus-Naur Form

LECTURE 1



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, $\langle \rangle$, 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 \rightarrow **name** ::= **expansion**

name \rightarrow < **identifier** >

expansion \rightarrow expansion expansion ; concatenation

expansion \rightarrow expansion | expansion ; option

expansion \rightarrow name ; non-terminals

expansion \rightarrow 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

$$\begin{aligned}\langle \text{expr} \rangle &\rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \\ &\quad | \langle \text{expr} \rangle - \langle \text{term} \rangle \\ &\quad | \langle \text{term} \rangle\end{aligned}$$
$$\begin{aligned}\langle \text{term} \rangle &\rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle \\ &\quad | \langle \text{term} \rangle / \langle \text{factor} \rangle \\ &\quad | \langle \text{factor} \rangle\end{aligned}$$

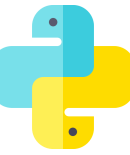
EBNF

$$\langle \text{expr} \rangle \rightarrow \langle \text{term} \rangle \{ (+ \mid -) \langle \text{term} \rangle \}$$
$$\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \{ (* \mid /) \langle \text{factor} \rangle \}$$



Extended Backus-Naur Form

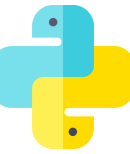
LECTURE 1



EBNF Rules and Descriptions

Control Forms of Right-Hand Sides

Sequence	Items appear left-to-right; their order is 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 choose 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	=
<u>concatenation</u>	,
termination	;
<u>alternation</u>	
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.
 - $\langle \text{proc_call} \rangle \rightarrow \text{ident } (\langle \text{expr_list} \rangle) \text{opt}$
 - $\langle \text{proc_call} \rangle \rightarrow \text{ident } [(\langle \text{expr_list} \rangle)]$
 - *Alternative parts:*
 - Pipe (|) indicates either-or choice
 - Grouping of the choices is done with square brackets or brackets.
 - $\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle [+ \mid -] \text{const}$
 - $\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle (+ \mid -) \text{const}$
 - *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.
 - $\langle \text{ident} \rangle \rightarrow \text{letter} \{ \text{letter} \mid \text{digit} \}^*$
 - $\langle \text{ident} \rangle \rightarrow \text{letter} \{ \text{letter} \mid \text{digit} \}$



Syntax Diagram

LECTURE 1



Syntax Diagram

- Graphical representation of EBNF rules

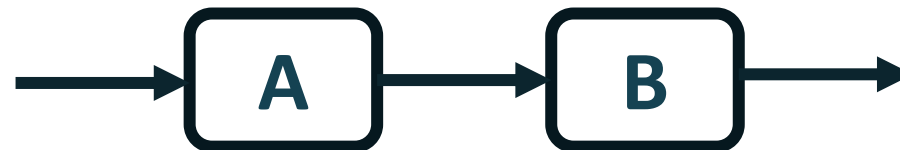
- Non-terminals:



- Terminals:



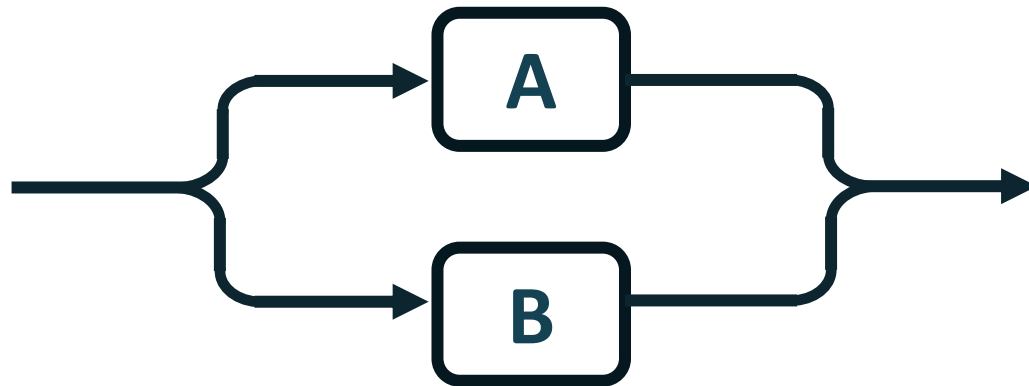
- Sequences AB





Syntax Diagram

- Graphical representation of EBNF rules
 - Sequences $A \mid B$





Syntax Diagram

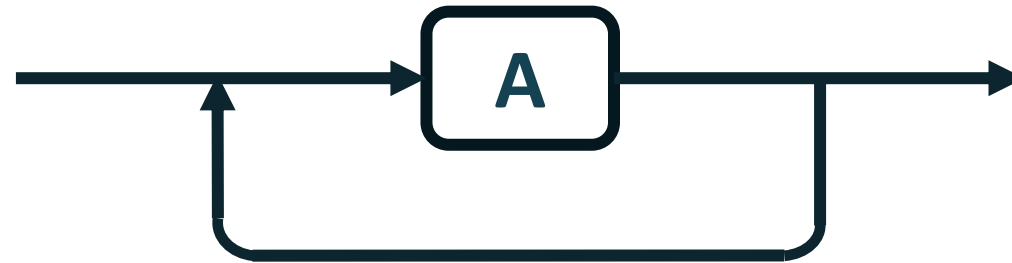
- Graphical representation of EBNF rules
 - Sequences A^*





Syntax Diagram

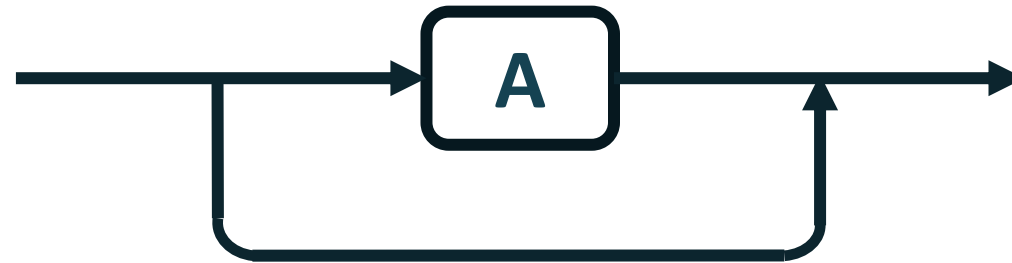
- Graphical representation of EBNF rules
 - Sequences A^+





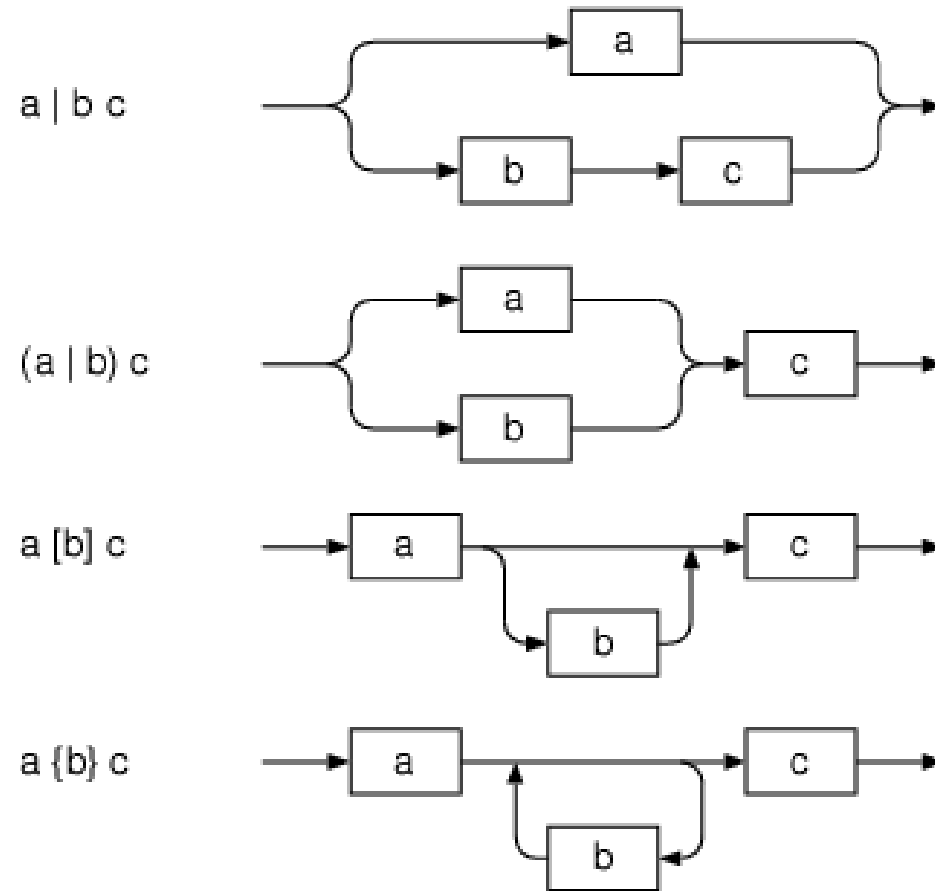
Syntax Diagram

- Graphical representation of EBNF rules
 - Sequences $A^?$



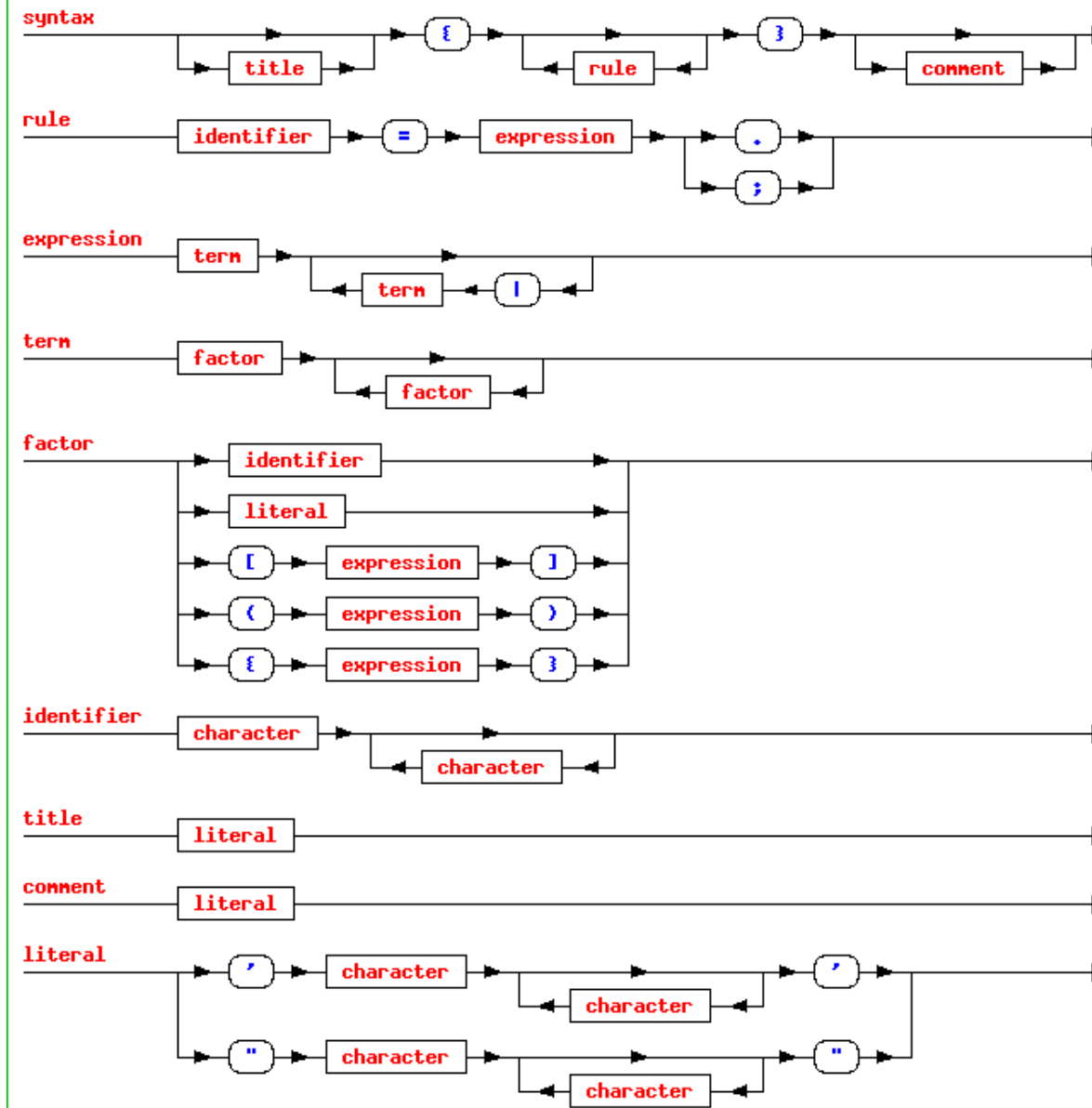


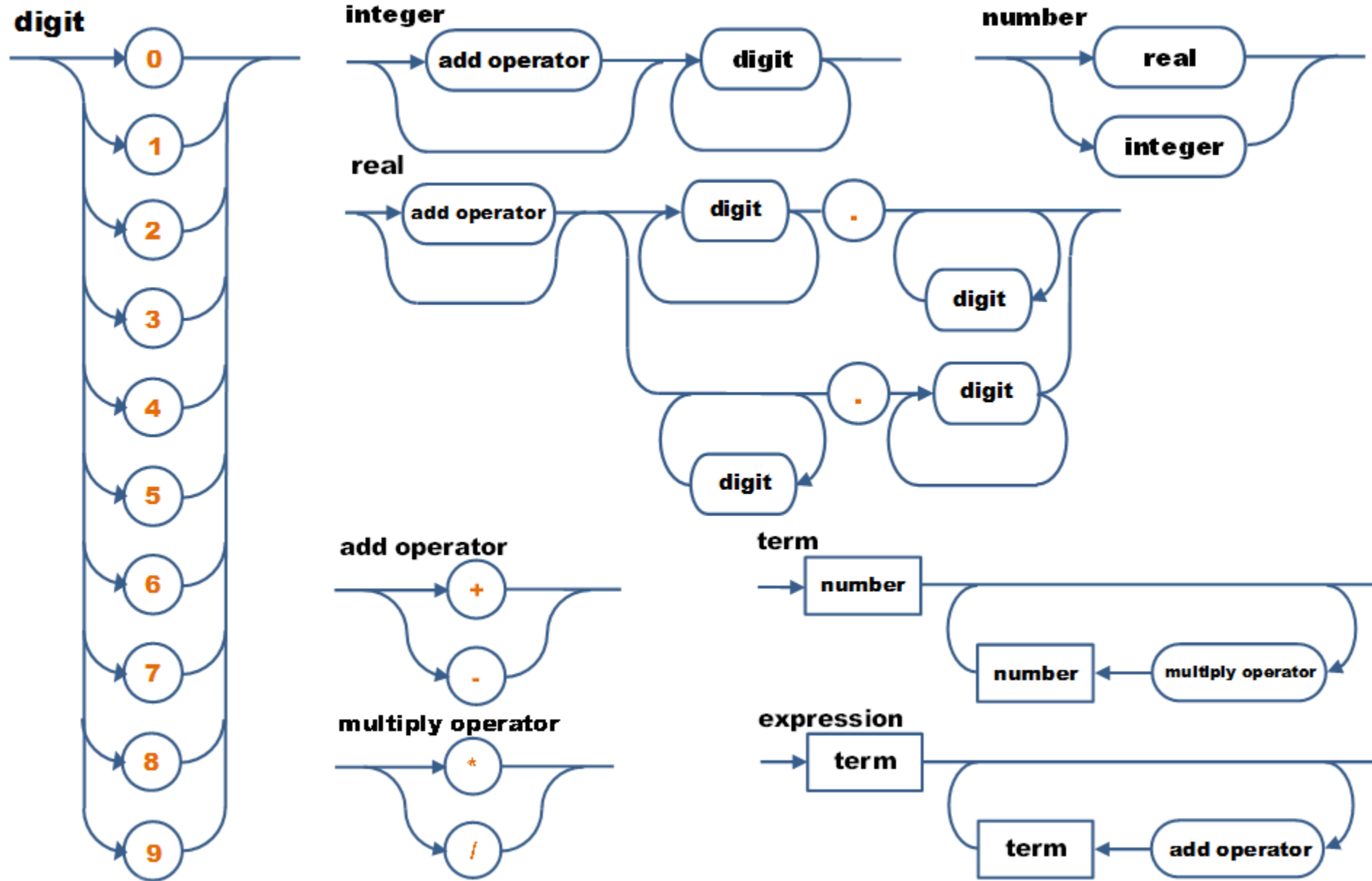
Syntax Diagram



Interpretation	BNF and EBNF example	Railroad Diagram example
Terminal Symbol for a reserved word. BEGIN is a reserved word.	BEGIN 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 (abc "("	
Non-terminal symbol. Item is defined elsewhere.	<Item> <Item>	
"or" a choice between two alternatives. Either Item1 or Item2.	<Item1> <Item2> <Item1> <Item2>	
"is defined as". Item can take the value a or b	Item ::= a b Item = a b	
Optional part. Item followed optionally by a Thing.	<Item> <Item><Thing> <Item>[<Thing>]	
Possible repetition. This is an Item repeated zero or more times.	This ::= <This><Item> <Item> "" This = {<Item>}	
Repetition. That is an Item repeated one or more times.	That ::= <That><Item> <Item> 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 Foogle ::= <Foo> BOO Foogle = (<Item>FOO) BOO	

EBNF defined in itself







EBNF and Regular Expression

LECTURE 1

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 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.

Regex 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).
{ <i>m</i> }	Matches exactly <i>m</i> repetitions of the previous sub-regexp
{ <i>m</i> , <i>n</i> }	Matches from <i>m</i> to <i>n</i> repetitions of the preceding sub-regexp (greedy).

Regex Syntax

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