



CS46K Programming Languages

Structure and Interpretation of Computer Programs

Chapter 8B Lexical Analysis – Scanner Design

LECTURE 11: SCANNER DESIGN

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

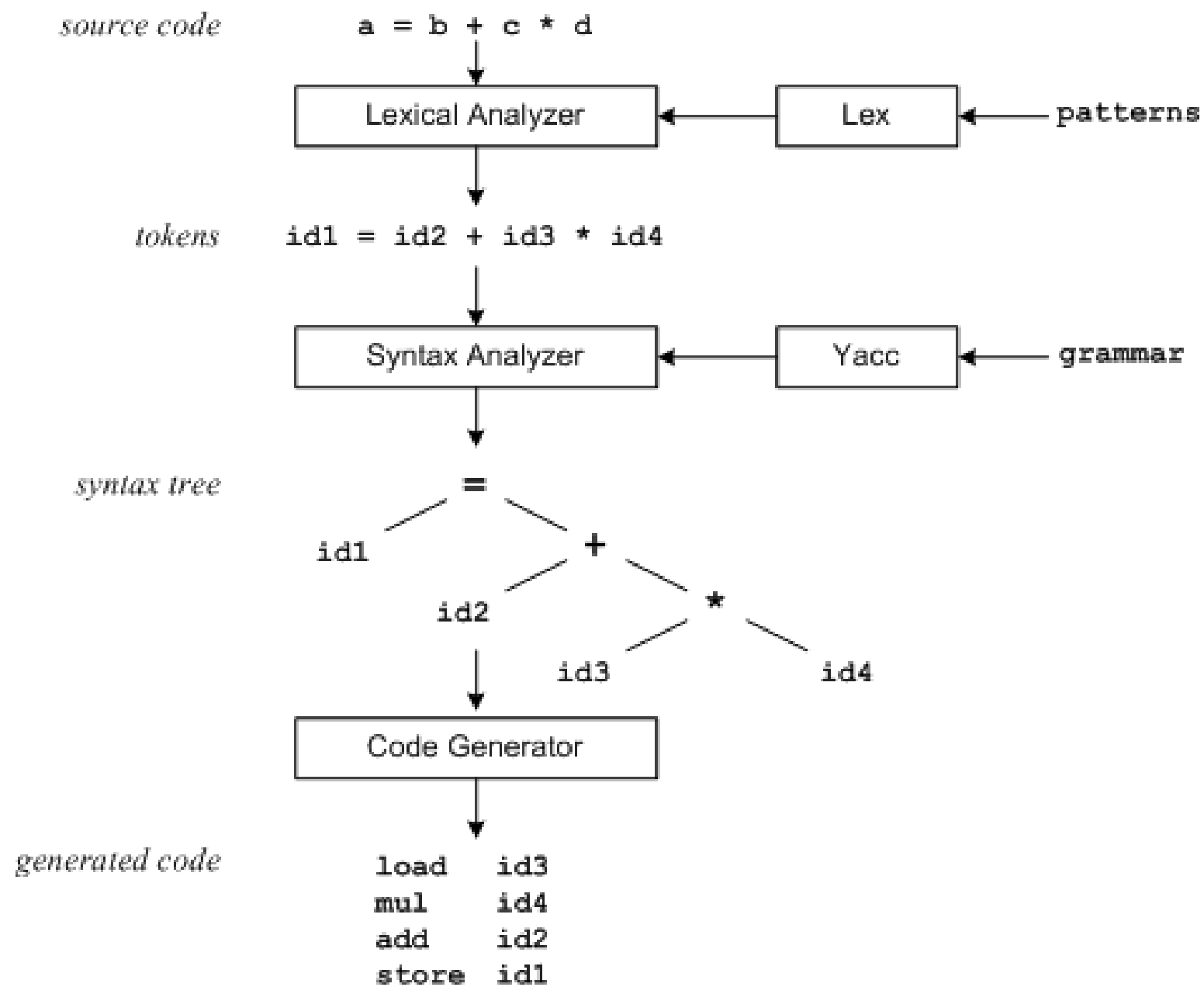
- What is Compiler-Compiler?
- Lex and Yacc, Flex and Bison, PLY
- What How to write Regular Expression
- Study the lexical analysis using Regular Expressions
- A Simple PLY Scanner design that match text to generate tokens.

Overview of Lex and Yacc

SECTION 1

Overview

- Before 1975 writing a compiler was a very time-consuming process. Then [Lesk \[1975\]](#) and [Johnson \[1975\]](#) published papers on lex and yacc. These utilities greatly simplify compiler writing. Implementation details for lex and yacc may be found in [Aho \[2006\]](#). Flex and bison, clones for lex and yacc, can be obtained for free from [GNU](#) and [Cygwin](#).
- Cygwin is a 32-bit Windows ports of GNU software. In fact, Cygwin is a port of the Unix operating system to Windows and comes with compilers gcc and g++. To install simply download and run the setup executable. Under **devel** install bison, flex, gcc-g++, gdb, and make. Under **editors** install vim. Lately
- Flex and Bison can also be used under the Cygwin (MinGW) environment.



Lexical Analysis (Flex)

With help from Bison

Patterns: Regular Expression (in .l Lex file)

- The lexical analyzer matches strings in the input, based on your patterns, and converts the strings to tokens.
- Tokens are numerical representations of strings, and simplify processing.
- When the lexical analyzer finds identifiers in the input stream it enters them in a symbol table. The symbol table may also contain other information such as data type (integer or real) and location of each variable in memory.
- All subsequent references to identifiers refer to the appropriate symbol table index.

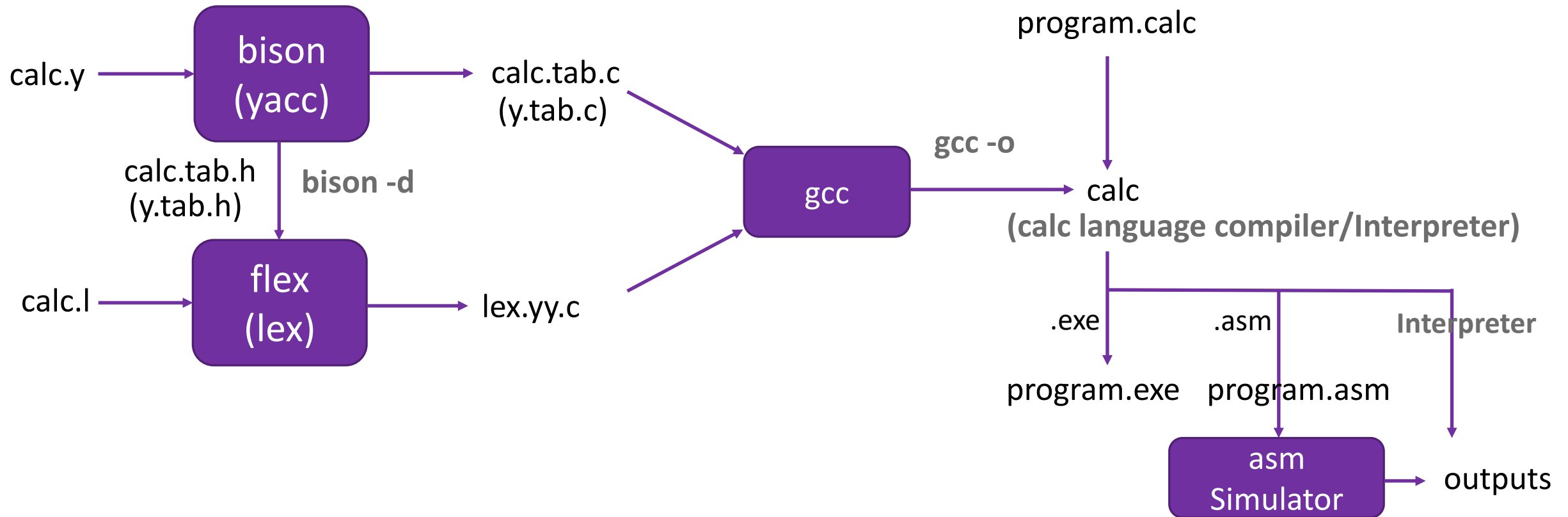
Syntax Analysis and Semantic Analysis

Grammar: Context Free Grammar (in .y Yacc file)

The syntax analyzer uses grammar rules that allow it to analyze tokens from the lexical analyzer and create a syntax tree. The syntax tree imposes a hierarchical structure on the tokens.

For example, operator precedence and associativity are apparent in the syntax tree. The next step, code generation, does a depth-first walk of the syntax tree to generate code. Some compilers produce machine code, while others, as shown above, output assembly language.

How Flex and Bison Works Together?



Project Building Process (I)

- First, we need to specify all pattern matching rules for flex (calc.l) and grammar rules for bison (calc.y). Commands to create our compiler, calc.exe, are listed below:

bison -d calc.y

create calc.tab.h (y.tab.h),

calc.tab.c (y.tab.c)

flex calc.l

create lex.yy.c,

need to include calc.tab.h (or y.tab.h)

gcc lex.yy.c calc.tab.c -o calc.exe

compile/link

Project Building Process (II)

- Bison reads the grammar descriptions in `calc.y` and generates a syntax analyzer (parser), that includes function `yyparse()`, in file `calc.tab.c` (`y.tab.c`). Included in file `calc.y` are token declarations.
- The `-d` option causes Bison to generate definitions for tokens and place them in file `calc.tab.h` (`y.tab.h`). Flex reads the pattern descriptions in `calc.l`, includes file `calc.tab.h` (`y.tab.h`), and generates a lexical analyzer, function `yylex()`, in file `lex.yy.c`.
- Finally, the lexical analyzer and parser are compiled and linked together to create executable `calc.exe`. From main we call `yyparse()` to run the compiler. Function `yyparse()` automatically calls `yylex()` to obtain each token.

Bibliography

- **Aho**, Alfred V., Ravi Sethi and Jeffrey D. Ullman [2006]. [Compilers, Principles, Techniques and Tools](#) (2nd edition). Addison-Wesley, Reading, Massachusetts.
- **Gardner**, Jim, Chris Retterath and Eric Gisin [1988]. [MKS Lex & Yacc](#). Mortice Kern Systems Inc., Waterloo, Ontario, Canada.
- **Johnson**, Stephen C. [1975]. [Yacc: Yet Another Compiler Compiler](#). Computing Science Technical Report No. 32, Bell Laboratories, Murray hill, New Jersey.
- **Lesk**, M. E. and E. Schmidt [1975]. [Lex - A Lexical Analyzer Generator](#). Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, New Jersey.
- **Levine**, John R., Tony Mason and Doug Brown [1992]. [Lex & Yacc](#). O'Reilly & Associates, Inc. Sebastopol, California.

Lex Theory

SECTION 2

Lex Theory

- During the first phase the compiler reads the input and converts strings in the source to tokens. With regular expressions we can specify patterns to **lex** so it can generate code that will allow it to scan and match strings in the input. Each pattern in the input to **lex** has an associated action. Typically an action returns a token that represents the matched string for subsequent use by the parser. Initially we will simply print the matched string rather than return a token value.
- The following represents a simple pattern, composed of a regular expression, that scans for identifiers (**ID**). **Lex** will read this pattern and produce C code for a lexical analyzer that scans for identifiers.

Lex Theory

letter(letter|digit) *

This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits. This example nicely illustrates operations allowed in regular expressions:

- **repetition**, expressed by the "*" operator
- **alternation**, expressed by the "|" operator
- **concatenation**

Finite State Automaton

Any regular expression expressions may be expressed as a finite state automaton (FSA). We can represent an FSA using states, and transitions between states. There is one start state and one or more final or accepting states.

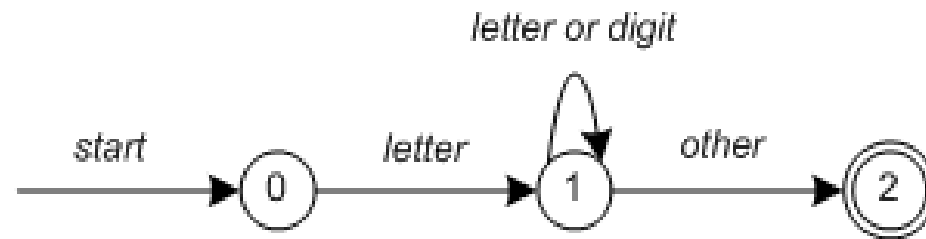
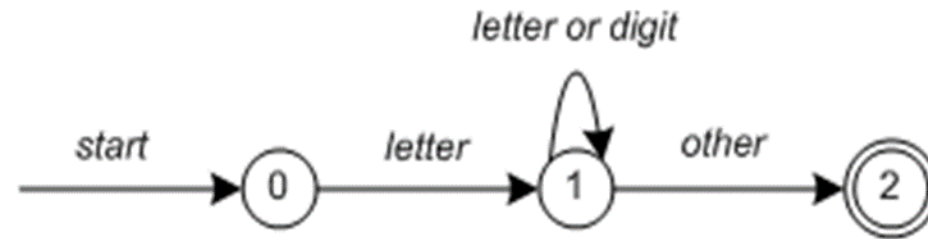


Figure A: Finite State Automaton

Ad hoc Lexical Analyzer

In Figure A state 0 is the start state and state 2 is the accepting state. As characters are read we make a transition from one state to another. When the first letter is read we transition to state 1. We remain in state 1 as more letters or digits are read. When we read a character other than a letter or digit we transition to accepting state 2. Any FSA may be expressed as a computer program. For example, our 3-state machine is easily programmed:

```
start: goto state0
state0: read c
        if c = letter goto state1
        goto state0
state1: read c
        if c = letter goto state1
        if c = digit goto state1
        goto state2
state2: accept string
```



This is the technique used by lex. Regular expressions are translated by lex to a computer program that mimics an FSA. Using the next input character and current state the next state is easily determined by indexing into a computer-generated state table.

Limitations

Now we can easily understand some of **lex**'s limitations.

- **lex** cannot be used to recognize nested structures such as parentheses.
- Nested structures are handled by incorporating a stack. Whenever we encounter a "(" we push it on the stack. When a ")" is encountered we match it with the top of the stack and pop the stack. However, **lex** only has states and transitions between states.
- Since it has no stack it is not well suited for parsing nested structures. **Yacc** augments an **FSA** with a stack and can process constructs such as parentheses with ease.
- The important thing is to use the right tool for the job. **Lex** is good at pattern matching. **Yacc** is appropriate for more challenging tasks.

Lex with Regular Expression

SECTION 3

Flex Regular Expression File

- Regular expressions in Flex are composed of metacharacters (Table 1).
- Pattern-matching examples are shown in Table 2.
- Within a character class normal operators lose their meaning. Two operators allowed in a character class are the hyphen ("-") and circumflex ("^"). When used between two characters the hyphen represents a range of characters. The circumflex, when used as the first character, negates the expression.
- **If two patterns match the same string the longest match wins. In case both matches are the same length, then the first pattern listed is used.**

Meta-symbols, Literal and Regex Rules

Metacharacter	Matches
.	any character except newline
\n	newline
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line
\$	end of line
a b	a or b
(ab) +	one or more copies of ab (grouping)
"a+b"	literal " a+b " (C escapes still work)
[]	character class

Table 1: Pattern Matching Primitives

Regular Expression

Expression	Matches
<code>abc</code>	<code>abc</code>
<code>abc*</code>	<code>ab abc abcc abccc ...</code>
<code>abc+</code>	<code>abc, abcc, abccc, abcccc, ...</code>
<code>a(bc)+</code>	<code>abc, abcbc, abcbcbc, ...</code>
<code>a(bc)?</code>	<code>a, abc</code>
<code>[abc]</code>	one of: <code>a, b, c</code>
<code>[a-z]</code>	any letter, a through z
<code>[a\-z]</code>	one of: <code>a, -, z</code>
<code>[-az]</code>	one of: <code>- a z</code>
<code>[A-Za-z0-9]+</code>	one or more alphanumeric characters
<code>[\t\n]+</code>	whitespace
<code>[^ab]</code>	anything except: <code>a, b</code>
<code>[a^b]</code>	<code>a, ^, b</code>
<code>[a b]</code>	<code>a, , b</code>
<code>a b</code>	<code>a, b</code>

Table 2: Pattern Matching Examples

Name	Function
<code>int yylex(void)</code>	call to invoke lexer, returns token
<code>char *yytext</code>	pointer to matched string
<code>yylen</code>	length of matched string
<code>yyval</code>	value associated with token
<code>int yywrap(void)</code>	wrapup, return 1 if done, 0 if not done
<code>FILE *yyout</code>	output file
<code>FILE *yyin</code>	input file
<code>INITIAL</code>	initial start condition
<code>BEGIN condition</code>	switch start condition
<code>ECHO</code>	write matched string

Table 3: Lex Predefined Variables

Flex Program Structure

... definitions ...

%%

... rules ...

%%

... user program ...
(or sub-routines)

Input to Lex is divided into three sections with %% dividing the sections. This is best illustrated by example. The first example is the shortest possible lex file:

%%

Input is copied to output one character at a time. The first %% is always required as there must always be a rules section. However if we don't specify any rules then the default action is to match everything and copy it to output.

Demo Program:

Defaults for input and output are **stdin** and **stdout**, respectively. Here is the same example with defaults explicitly coded:

```
%%  
\n      ;  
.  
%%
```

<<flex0.l Do nothing Flex program>>

```
%%  
    /* match everything except newline */  
    .      ECHO;  
    /* match newline */  
\n      ECHO;  
  
%%  
  
int yywrap(void) {  
    return 1;  
}  
  
int main(void) {  
    yylex();  
    return 0;  
}  
    <<flex1.l Repeating Flex program>>
```

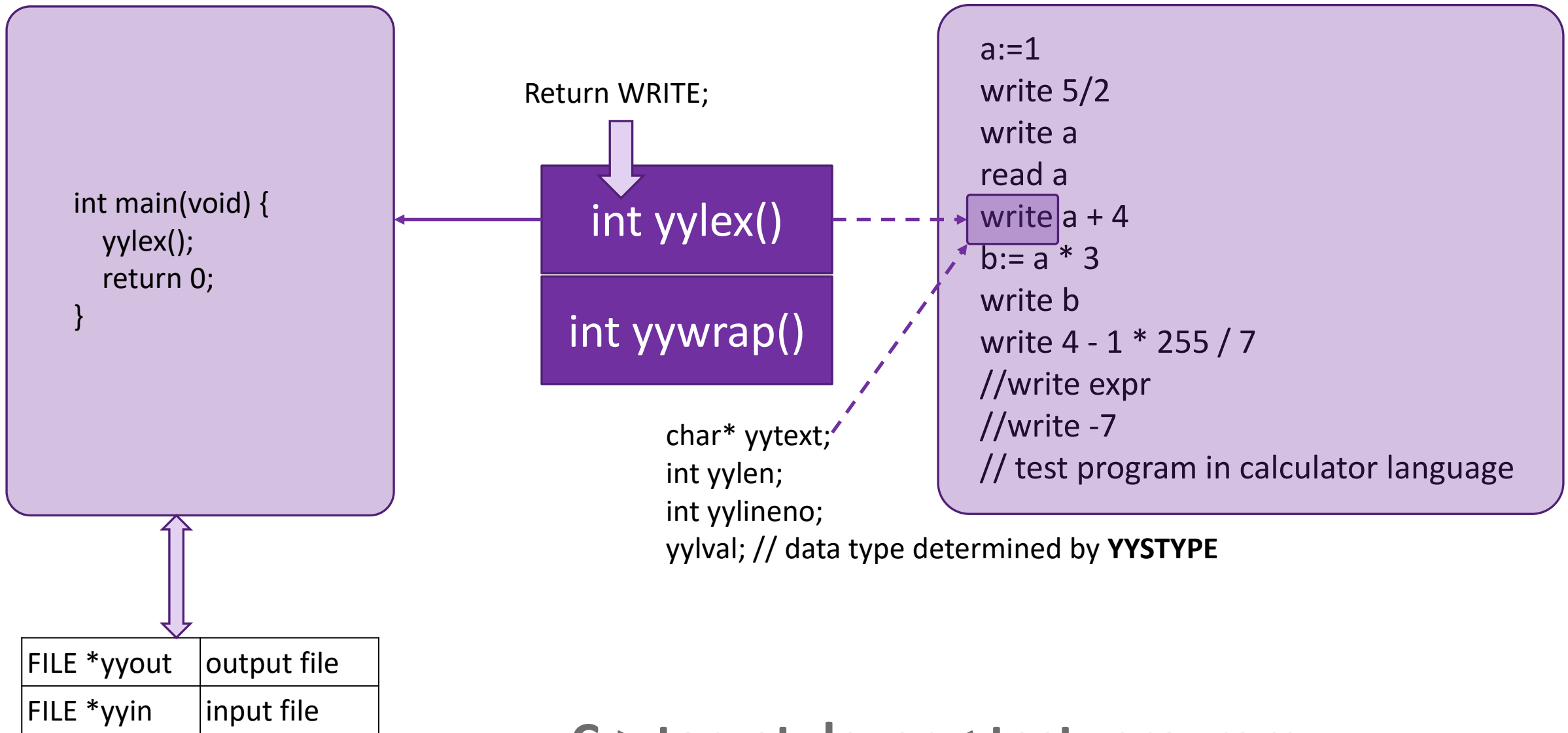

Demo Program:

Two patterns have been specified in the rules section. Each pattern must begin in column one. This is followed by whitespace (space, tab or newline) and an optional action associated with the pattern.

The action may be a single C statement, or multiple C statements, enclosed in braces. Anything not starting in column one is copied verbatim to the generated C file. We may take advantage of this behavior to specify comments in our lex file.

In this example there are two patterns, "." and "\n", with an **ECHO** action associated for each pattern. Several macros and variables are predefined by flex. **ECHO** is a macro that writes code matched by the pattern. This is the default action for any unmatched strings. Typically, **ECHO** is defined as:

```
#define ECHO fwrite(yytext, yyleng, 1, yyout)
```



C:> target_lexer < test_program

Demo Program:

- Variable **yytext** is a pointer to the matched string (**NULL**-terminated) and **yylen** is the length of the matched string. Variable **yyout** is the output file and defaults to **stdout**.
- Function **yywrap()** is called by **FLex** when input is exhausted. Return **1** if you are done or **0** if more processing is required.

Demo Program:

- Every **C** program requires a **main()** function.
- In this case we simply call **yylex()** that is the main entry-point for **Flex**. Some implementations of **Flex** include copies of **main()** and **yywrap()** in a library thus eliminating the need to code them explicitly.
- This is why our first example, the shortest **Flex** program, functioned properly.

Demo Program:

flex2.l

The following example prepends line numbers to each line in a file. Some implementations of lex predefine and calculate yylineno. The input file for flex is yyin and defaults to stdin.

Waive the requirement for yywrap()

```
%option noyywrap
%{
    int yylineno;
}%
%%
^(.*)\n    printf("%4d\t%s", ++yylineno, yytext);
%%
int main(int argc, char *argv[]) {
    yyin = fopen(argv[1], "r");
    yylex();
    fclose(yyin);
}
```

```
W:\Eric_Chou\FlexBison\CDev\Flex2>flex flex2.l
W:\Eric_Chou\FlexBison\CDev\Flex2>gcc lex.yy.c -o flex2
W:\Eric_Chou\FlexBison\CDev\Flex2>flex2 flex2.l
1    %option noyywrap
2    %{
3        int yylineno;
4    %}
5    %%
6    ^(.*)\n    printf("%4d\t%s", ++yylineno, yytext);
7    %%
8    int main(int argc, char *argv[]) {
9        yyin = fopen(argv[1], "r");
10       yylex();
11       fclose(yyin);
}
```

Print every line but no token release (no Return statements in action part.)

Demo Program:

flex3.l

- The definitions section is composed of substitutions, code, and start states.
- Code in the definitions section is simply copied as-is to the top of the generated C file and must be bracketed with "%{" and "%}" markers.
- Substitutions simplify pattern-matching rules. For example, we may define digits and letters: (Check flex5.l project)

digit [0-9]
letter [A-Za-z]

```
%option noyywrap
%{
    int count;
}%
%%
[a-zA-Z]([a-zA-Z]|[0-9])*      count++;
.
%%
int main(int argc, char *argv[]) {
    yyin = fopen(argv[1], "r");
    yylex();
    printf("number of identifiers = %d\n", count);
    fclose(yyin);
    return 0;
}
```

```
W:\Eric_Chou\FlexBison\CDev\Flex3>flex flex3.l

W:\Eric_Chou\FlexBison\CDev\Flex3>gcc lex.yy.c -o flex3

W:\Eric_Chou\FlexBison\CDev\Flex3>flex3 flex3.l

number of identifiers = 32
```

Demo Program:

flex4.l

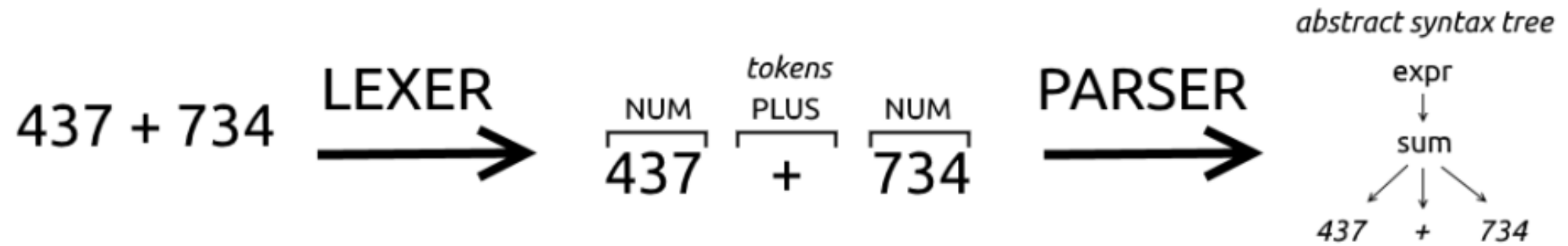
Whitespace must separate the defining term and the associated expression. References to substitutions in the rules section are surrounded by braces ([a-zA-Z]) to distinguish them from literals. When we have a match in the rules section the associated C code is executed. Here is a scanner that counts the number of characters, words, and lines in a file (similar to Unix wc):


```
%option noyywrap
%{
    int nchar, nword, nline;
}%
%%
\n          { nline++; nchar++; }
[^\t\n]+    { nword++, nchar += yyleng; }
.           { nchar++; }
%%
int main(int argc, char *argv[]) {
    yyin = fopen(argv[1], "r");
    yylex();
    printf("%d\t%d\t%d\n", nchar, nword, nline);
    fclose(yyin);
    return 0;
}
```

```
W:\Eric_Chou\FlexBison\CDev\Flex3>cd ..
W:\Eric_Chou\FlexBison\CDev>cd Flex4
W:\Eric_Chou\FlexBison\CDev\Flex4>flex flex4.l
W:\Eric_Chou\FlexBison\CDev\Flex4>gcc lex.yy.c -o flex4
W:\Eric_Chou\FlexBison\CDev\Flex4>flex4 flex4.l
316      46      16
```

Parsing of Tokens: Extended Backus-Naur Form

SECTION 4



	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

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.
 - $\langle \text{ident} \rangle \rightarrow \text{letter } \{ \text{letter} \mid \text{digit} \}^*$
 - $\langle \text{ident} \rangle \rightarrow \text{letter } \{ \text{letter} \mid \text{digit} \}$

(.*) 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 package, nltk: natural language toolkit)

Regular Expression Rules

SECTION 5

General Rule of Matching:

- **General Rule of Matching:** Regular expressions match the most number of characters possible (called a **greedy algorithm**; there are patterns that match the fewest number of characters possible; we will mention, but not discuss nor use those patterns).

Regular Expressions

Metacharacters

- . Matches any single character (except newline: \n)
- [] Matches one character specified inside []; e.g., [aeiou]
- [^] Matches one character NOT specified inside [] after ^; e.g., [^aeiouy]
- Matches one character in range inside []: e.g., [0-9] matches any digit

Anchors (these don't match characters)

- ^ matches beginning of line (when not used in [])
- \$ matches end of line

Regular Expressions

Patterns: R, Ra, Rb are regular expression patterns

RaRb Matches a sequence of Ra followed by Rb

Ra|Rb Matches either alternative Ra or Rb

R? Matches regular expression R 0/1 time: R is optional

R* Matches regular expression R 0 or more times

R+ Matches regular expression R 1 or more times

R{m} Matches regular expression R exactly m times: e.g., R{5} = RRRRR

R{m, n} Matches regular expression R at least m and at most n times:

R{3,5} = RRR|RRRR|RRRRR = RRRR?R?

R??,R*?,R+?,R{m,n}? The postfix ? means match as few characters possible (not the most: so not greedy).

Parentheses/Parenthesized Patterns

- Parentheses are used for grouping, but can also remember subpatterns (this is also called a "**Capturing Group**").
- By placing subpattern **R** in parentheses, the text matching **R** will be remembered (either by its number, starting at 1, or its name, if named) in a group, for use later in the pattern or when extracting information from the matched text.

Parentheses/Parenthesized Patterns

(R) Matches R and delimits a group (1...) (remembers/captures matched text)

(?P<name>R) Matches R and remembers/captures matched text in a group using name for the group (it is still numbered as well); see **(?P=name)** and groupdict method below for use of "name".

(?:R) Matches R but does not remember/capture matched text in a group So, there () are used only for grouping, not capturing groups; **?:** is useful when you want the minimum number (no redundant groups)

Parentheses/Parenthesized Patterns

(?P=name) Matches remembered text with name (for backreferencing which text)

(?=R) Matches **R**, doesn't remember matched text/consume text matched. For example `(?=abc)(.*)` matches `abcxyz` with group 1 `'abcxyz'`; it doesn't match `abxy` because this text doesn't start with `abc`

(?!R) Matches anything but **R** and does not consume input needed for match (hint: **!=** means "not equal", **?!R** means "not matching **R**")

Context

- matches itself if not in [], and if not between two characters
- Special characters are treated as themselves in []: e.g, [.] matches literal .
- Generally, if interpreting a character makes no sense one way, try to find another way to interpret it that fits the context

Escape Characters with Special Meanings

<code>\</code>	Used before <code>.</code> <code>[]-?*</code> <code>{ } () ^ \$ \</code> (and others) to specify a special character
<code>\#</code>	Backreferencing group # (numbered from 1, 2, ...): see (R) above
<code>\t</code>	tab
<code>\n</code>	newline
<code>\r</code>	carriage return
<code>\f</code>	form-feed
<code>\v</code>	vertical tab

Escape Characters with Special Meanings

<code>\d</code>	<code>[0-9]</code>	Digit
<code>\D</code>	<code>[^0-9]</code>	non-Digit
<code>\s</code>	<code>[\t\n\r\f\v]</code>	White space
<code>\S</code>	<code>[^ \t\n\r\f\v]</code>	non-White space
<code>\w</code>	<code>[a-zA-Z0-9_]</code>	alphanumeric (or underscore): Word character (id letters)
<code>\W</code>	<code>[^a-zA-Z0-9_]</code>	non-alphanumeric: non-Word character (non-id letters)

Interesting Equivalences

a^+ == aa^*

$a(b|c|d)e$ == $a[bcd]e$ only if b, c, and d are single characters

$R\{0,1\}$ == $R?$

Hints on Using |

(a low-precedence operator for Regular Expression)

- In Python, we know that writing $a*b+c*d$ performs $*$ before $+$: we say $*$ has higher precedence than $+$, so it is performed earlier.
- We could be explicit and write this as $(a*b)+(c*d)$. Think of REs as having sequence as an operation (it is implicit, with no operator).
- The sequence precedence is lower than the precedence of postfix operators (like $?$, $*$, and $+$): e.g., ab^* has the same meaning as $a(b^*)$.
- **All have higher precedence than $|$.** So writing $ab|cd$ is the equivalent of writing $(ab)|(cd)$, which matches either ab or cd only.

Hints on Using |

(a low-precedence operator for Regular Expression)

- Now, given this understanding, look what **`^a|b$`** means. By above, it means the same as **`(^a)|(b$)`**. Type **`^a|b$`** into the online tool and read its Explanation).
- Note that the **`^`** anchor applies only to **a**, and the **`$`** anchor applies only to **b** (see its parenthesized equivalent). So **`^a|b$`** (which is equivalent to **`(^a)|(b$)`**) will match (using the online tool)
 - **any text starting with an a: a** or **aa** or **aaab** or **abcda** (**`$`** is not part of it)
 - **any text ending with a b: b** or **cb** or **ccb** or **abcd** (**`^`** is not part of it)

Hints on Using |

(a low-precedence operator for Regular Expression)

- To avoid confusion, I strongly recommend always writing all the alternatives in a regular expression as a group: `^(a|b)$` to ensure that the `|` applies only to the alternatives inside the `()`s.
- This regular expression is a sequence of 3: the `^` anchor, a choice of a or b, and a `$` anchor.



Regular Expression Syntax

[or-set] [^not-set], Range: [a-z], Union: [set1[set2]] Intersection: [set1&&set2]

Regular Expression	Matches	Example
x	a specified character x	Java matches Java
.	any single character	Java matches J..a
(ab cd)	a, b, or c	ten matches t(en im)
[abc]	a, b, or c	Java matches Ja[uvw]a
[^abc]	any character except a, b, or c	Java matches Ja[^ars]a
[a-z]	a through z	Java matches [A-M]av[a-d]
[^a-z]	any character except a through z	Java matches Jav[^b-d]
[a-e[m-p]]	a through e or m through p	Java matches [A-G[I-M]]av[a-d]
[a-e&&[c-p]]	intersection of a-e with c-p	Java matches [A-P&&[I-M]]av[a-d]



Regular Expression Syntax

`\`one-wildcard-type-letter (in string `\\d`, first `\` is escape)

Regular Expression	Matches	Example
<code>\d</code>	a digit, same as <code>[1-9]</code>	Java2 matches <code>"Java[\\d]"</code>
<code>\D</code>	a non-digit	\$Java matches <code>"[\\D][\\D]ava"</code>
<code>\w</code>	a word character	Java matches <code>"[\\w]ava"</code>
<code>\W</code>	a non-word character	\$Java matches <code>"[\\W][\\w]ava"</code>
<code>\s</code>	a whitespace character	"Java 2" matches <code>"Java\\s2"</code>
<code>\S</code>	a non-whitespace char	Java matches <code>"[\\S]ava"</code>
Quantifiers		
<code>p*</code>	zero or more occurrences of pattern <code>p</code>	Java matches <code>"[\\w]*"</code>
<code>p+</code>	one or more occurrences of pattern <code>p</code>	Java matches <code>"[\\w]+"</code>
<code>p?</code>	zero or one occurrence of pattern <code>p</code>	Java matches <code>"[\\w]?Java"</code> Java matches <code>"[\\w]?ava"</code>
<code>p{n}</code>	exactly <code>n</code> occurrences of pattern <code>p</code>	Java matches <code>"[\\w]{4}"</code>
<code>p{n,}</code>	at least <code>n</code> occurrences of pattern <code>p</code>	Java matches <code>"[\\w]{3,}"</code>
<code>p{n,m}</code>	between <code>n</code> and <code>m</code> occurrences (inclusive)	Java matches <code>"[\\w]{1,9}"</code>

Python re.match()

Demo Program: regex_match.py

re.match():

```
re.match(pattern, string[, flags])
```

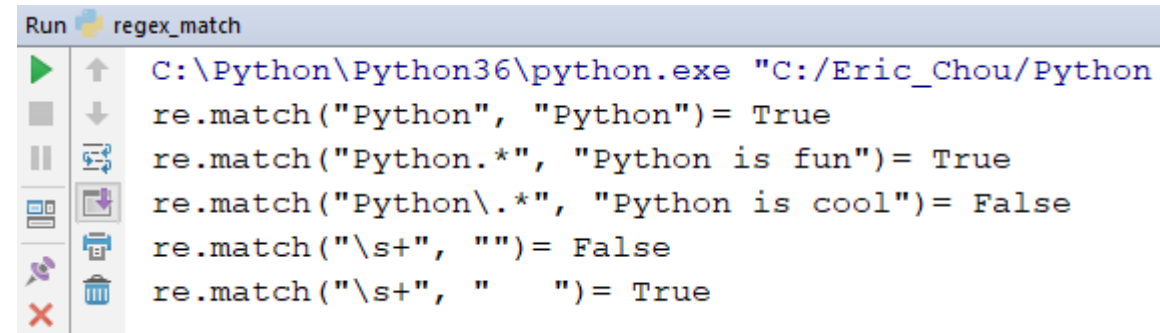
Example:

```
match-object = re.match(pattern, string_to_be_matched)
```

Return Value:

Return None if the string does not match the pattern; note that this is different from a zero-length match.


```
import re
# matches Python Strings
matchObj= re.match("Python", "Python")
if matchObj == None: matching = False
else: matching = True
print("re.match(\"Python\", \"Python\")=", matching)
# matches zero or more characters
matchObj= re.match("Python.*", "Python is fun")
if matchObj == None: matching = False
else: matching = True
print("re.match(\"Python.*\", \"Python is fun\")=", matching)
# not matching \.
matchObj= re.match("Python\.", "Python is cool")
if matchObj == None: matching = False
else: matching = True
print("re.match(\"Python\\.\", \"Python is cool\")=", matching)
# not matching one or more space
matchObj= re.match("\s+", "")
if matchObj == None: matching = False
else: matching = True
print("re.match(\"\\s+\", \"\")=", matching)
# matching one or more space
matchObj= re.match("\s+", " ")
if matchObj == None: matching = False
else: matching = True
print("re.match(\"\\s+\", \" \")=", matching)
```



```
Run regex_match
C:\Python\Python36\python.exe "C:/Eric_Chou/Python
re.match("Python", "Python")= True
re.match("Python.*", "Python is fun")= True
re.match("Python\\. ", "Python is cool")= False
re.match("\\s+", "")= False
re.match("\\s+", " ")= True
```

Regular Expression Designs (Part 1)

SECTION 6

Design Regular Expression for the Patterns

- Write the smallest pattern that matches the required characters.
- Check your patterns with the Regular Expression Tester (see the Sample Programs **link**) to ensure they match correct exemplars and don't match incorrect ones.
- Note that for a match, group #0 should include all the required characters.

Problem 1:

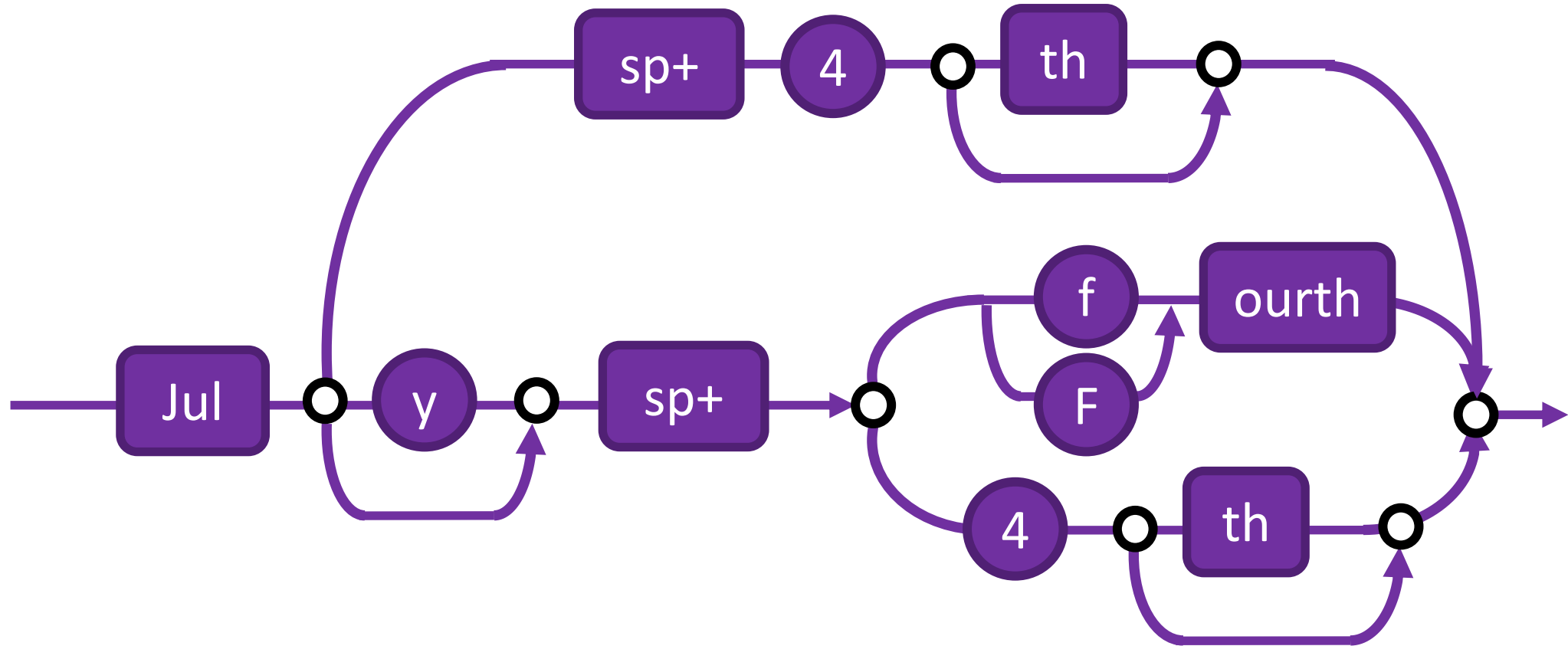
1. Write a regular expression pattern that matches the strings **Jul 4**, **July 4**, **Jul 4th**, **July 4th**, **July fourth**, and **July Fourth**.

Hint: my re pattern was 24 characters.

Problem 1 Solution: july_fourth.py

```
# july_fourth.py  
# problme 1:  
# regular expression for July Fourth  
import re  
# Sample strings.  
list = ["Jul 4", "Jul 4th", "July 4", "July 4th", "Jul Fourth",  
        "July Fourth", "jul 4", "July fourth", "j 4", "july Fourth"]  
# Loop.  
for element in list:  
    # Match if two words starting with letter d.  
    #12345678901234567890123  
    m = re.match("Jul (\s4(th)?|y\s(4(th)?|(f|F)ourth))", element)  
    # See if success.  
    if m:  
        print(m.group(0))
```

Syntax Diagram



Problem 2:

2. Write a regular expression pattern that matches strings representing **times on a 12 hour clock**.

An example time is 5:09am or 11:23pm. Allow only times that are legal (not 1:73pm nor 13:02pm)

Hint: my re pattern was 32 characters.

Problem 2 Solution: clock.py

clock.py

problem 2:

regular expression for clock

import re

Sample strings.

list = ["1:30pm", "23:18am", "32:00kk", "7:20pm", "10:32am"]

Loop.

for element **in** list:

Match if two words starting with letter d.

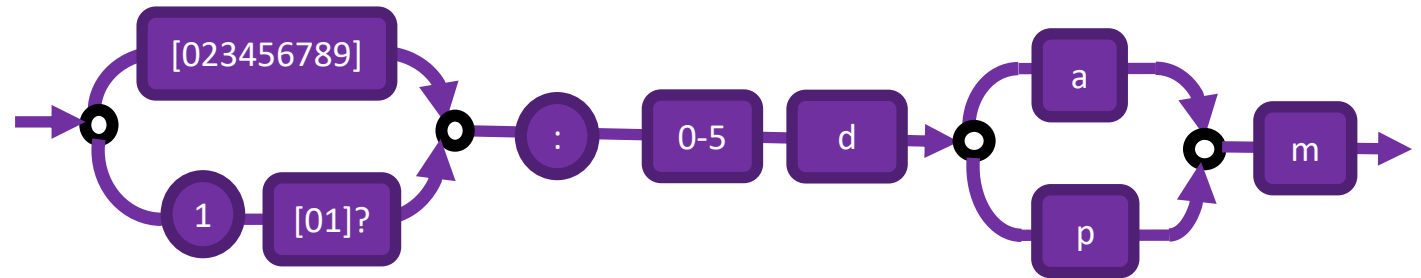
#12345678901234567890123456789012

m = re.match("([1-9]|1[0-2]):[0-5][0-9](a|p)m", element)

See if success.

if m:

 print(m.group(0))



```
Run clock
C:\Python\Python36\python.exe
1:30pm
7:20pm
10:32am
```

Problem 3:

3. Write a regular expression pattern that matches strings representing **phone numbers** of the following form.

- **Normal:** a three digit exchange, followed by a dash, followed by a four digit number: e.g., 555-1212
- **Long Distance:** a 1, followed by a dash, followed by a three digit area code enclosed in parentheses, followed by a three digit exchange, followed by a dash, followed by a four digit number: e.g.,

1-(800)555-1212

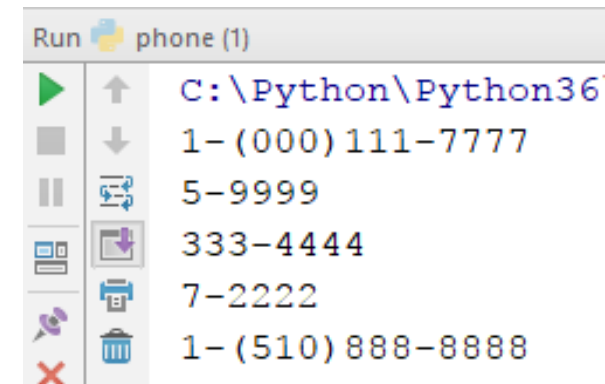
Problem 3:

- **Interoffice:** a single digit followed by a dash followed by a four digit number: e.g., 8-2404.

Hint: my re pattern was 30 characters; note that you must use `\(` and `\)` to match parentheses.

Problem 3 Solution: phonep3.py

```
# phonep3.py  
# problme 3:  
# regular expression for phone number  
import re  
# Sample strings.  
list = ["1-(000)111-7777", "0000", "5-9999", "333-4444",  
        "9999999", "7-2222", "1-(510)888-8888"]  
  
# Loop.  
for element in list:  
    # Match if two words starting with letter d.  
        #12345678901234567890123456789012  
    m = re.match("(1-(\d{3}\d)?\d\d)?\d-\d{4}", element)  
    # See if success.  
    if m:  
        print(m.group(0))
```



Run phone (1)

↑	C:\Python\Python36
↓	1-(000)111-7777
↺	5-9999
↻	333-4444
🖨	7-2222
🗑	1-(510)888-8888

Regex for Phone Call Formats

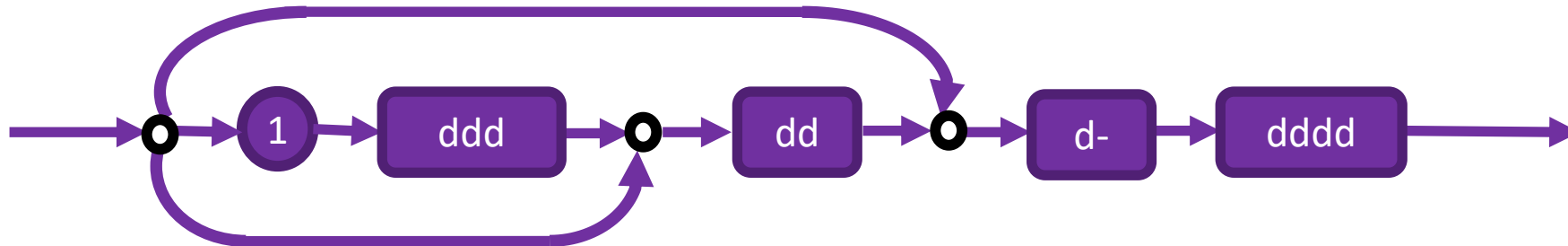
5-1212 Interoffice = $[0-9]-[0-9]\{4\}$

555-1212 Normal = $[0-9]\{2\}<\text{Interoffice}>$

1-(800)555-1212 International = $1-\backslash([0-9]\{3\}\backslash)<\text{Normal}>$

Three-in-one format:

$((1-\backslash(<\text{Area}>\backslash)?) [0-9]\{2\})? <\text{Interoffice}>$

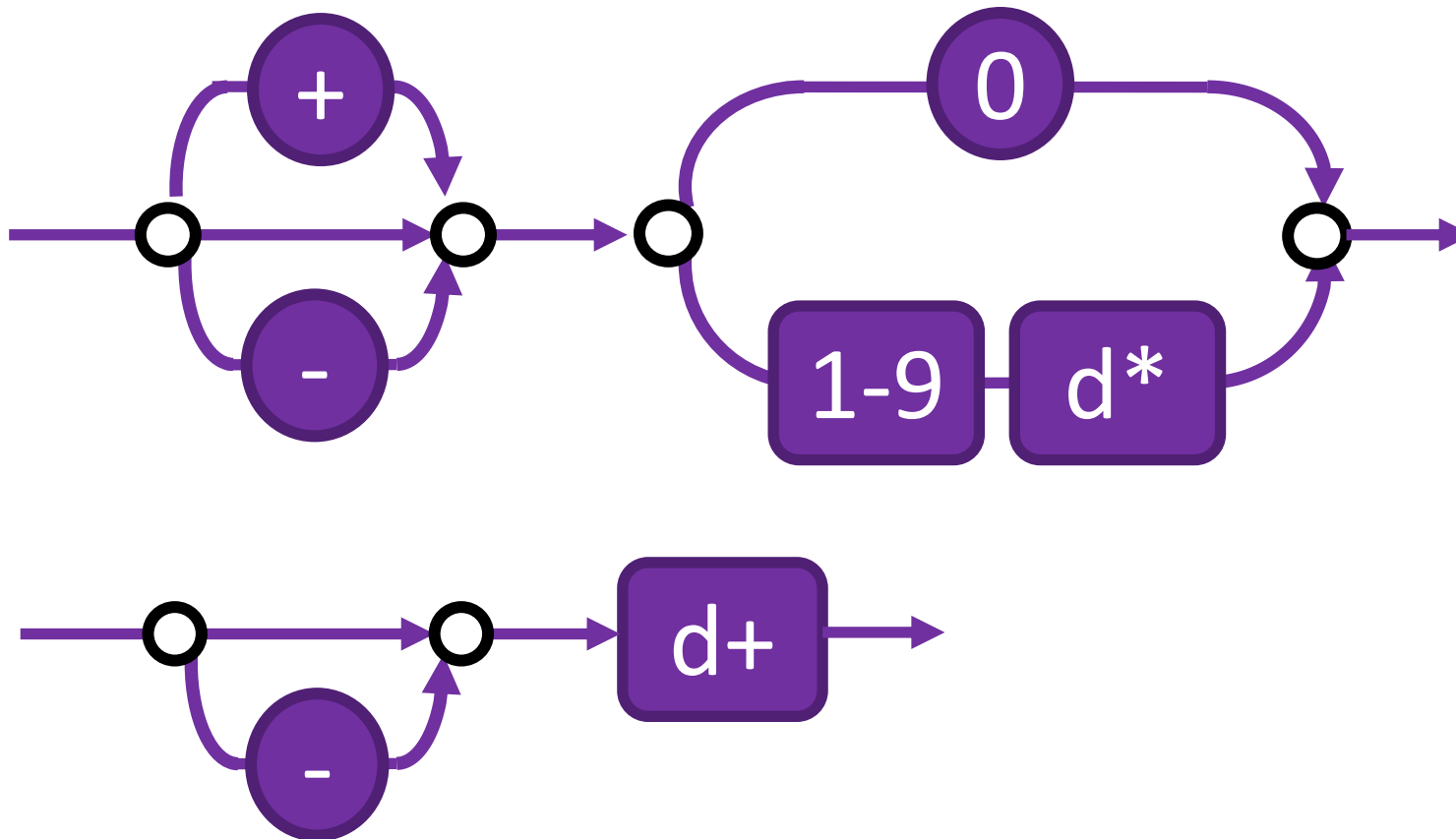


Problem 4:

4. Write a regular expression pattern that matches strings representing **simple integers**: optional + or - signs followed by one or more digits.

Hint: my re pattern was 7 characters.

Regex for Integer



Problem 4 Solution: integer.py

```
# integer.py  
# problme 4:  
# regular expression normal integer  
import re  
# Sample strings.  
list = ["00", "000", "33", "-102", "3.55", "0", "-32", "+255", "4300"]  
# Loop.  
for element in list:  
    # Match if two words starting with letter d.  
        #123456789012345678901  
    m = re.match("^(\+|-)?(0|([1-9]\d*))$", element)  
    # See if success.  
    if m:  
        print(m.group(0))
```

Problem 4 Solution: integer_short.py

```
# integer_short.py  
# problme 4:  
# regular expression for short integer format  
import re  
# Sample strings.  
list = ["00", "000", "33", "-102", "3.55", "0", "-32", "+255",  
"4300"]  
# Loop.  
for element in list:  
    # Match if two words starting with letter d.  
        #123456789012345678901  
    m = re.match("^-?\d+$", element)  
    # See if success.  
    if m:  
        print(m.group(0))
```

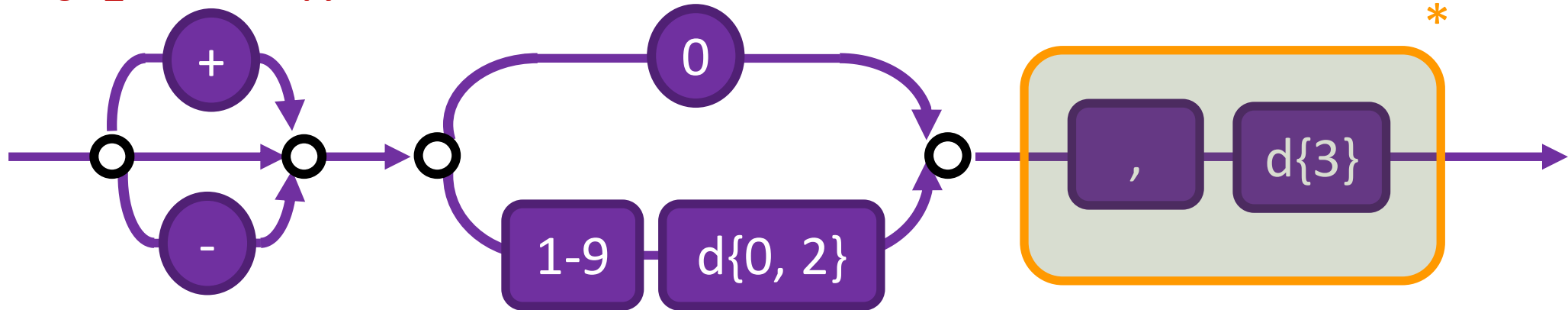
Problem 5:

5. Write a regular expression pattern that matches strings representing **normalized integers** (each number is either an unsigned 0 or is unsigned or signed and starts with a non-0 digit) with commas in only the correct positions

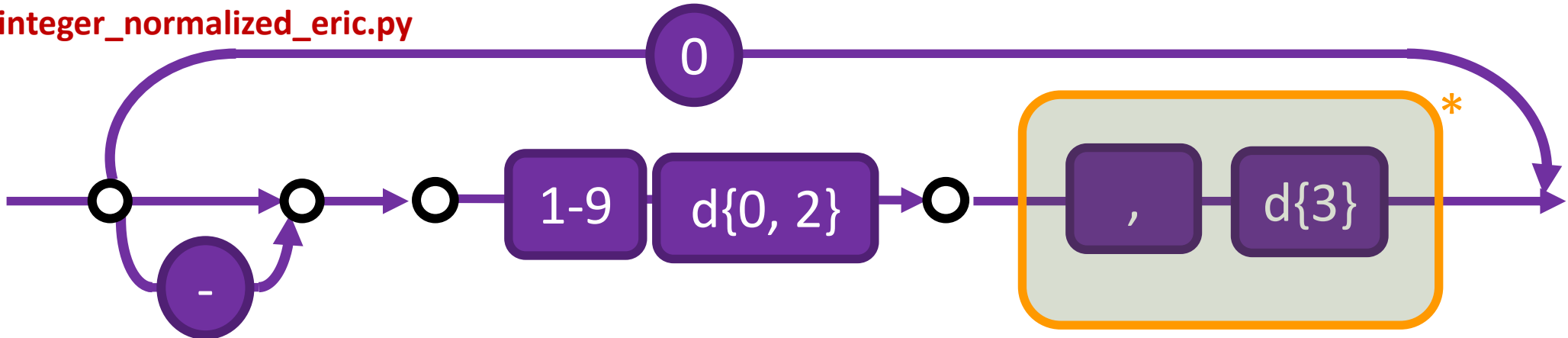
Hint: my re pattern was 30 characters.

Regex for Normalized Integer

integer_normalized.py



integer_normalized_eric.py



Demo Program: integer_normalized.py

```
# integer_normalized.py
# problme 5:
# regular expression normalized integer
import re
# Sample strings.
list = ["00", "1,000", "33", "-102", "3.55", "0", "-32", "+255",
        "4,300", "1,000,000", "22,999,444"]
# Loop.
for element in list:
    # Match if two words starting with letter d.
    #123456789012345678901
    m = re.match("^(\\+|-)?(0|([1-9]\\d*))\\,(\\d{3})*$", element)
    # See if success.
    if m:
        print(m.group(0))
```

Problem 6:

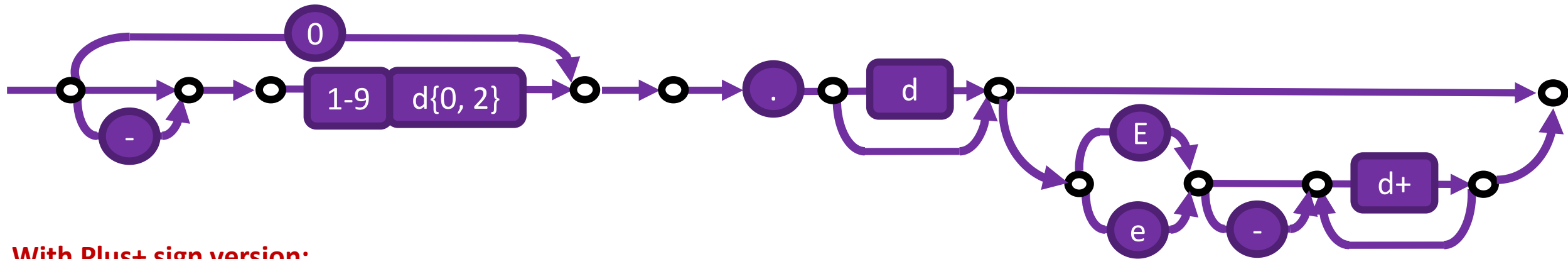
6. Write a regular expression pattern that matches strings representing **float values**.

They are unsigned or signed (but not normalized: see 5) and any number of digits before or after a decimal point (but there must be at least one digit either before or after a decimal point: e.g., just . is not allowed) followed by an optional e or E followed by an unsigned or signed integer (again not normalized).

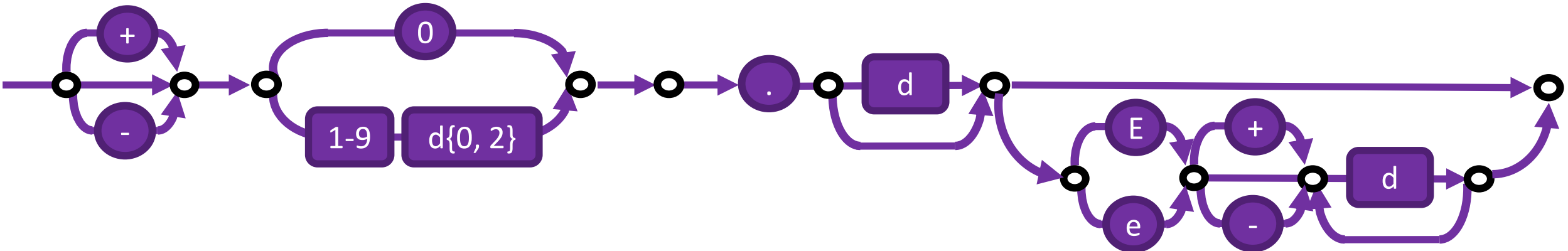
Hint: my re pattern was 36 characters.

Regex for Floating Point Number

Without Plus+ sign version:



With Plus+ sign version:



Problem 6 Solution: floatingpoint.py

```
# floatingpoint.py  
# problme 6:  
# regular expression floating point number  
import re  
# Sample strings.  
list = ["0", "0.00", "100.0", "0.33", "-102.5", "3.55", "0.0", "33",  
        "-32.7423897423", "+2.55", "43.0e33", "3.22E+7843"]  
# Loop.  
for element in list:  
    # Match if two words starting with letter d.  
    #123456789012345678901234567890123456789012345678901234  
    m = re.match("^(\+|-)?(0|([1-9]\d*))\.\d+((E|e)(\+|-)?\d+)?$", element)  
    # See if success.  
    if m:  
        print(m.group(0))
```

Problem 7:

7. Write a regular expression pattern that matches strings representing **trains**.

A single letter stands for each kind of car in a train: Engine, Caboose, Boxcar, Passenger car, and Dining car. There are four rules specifying how to form trains.

1. One or more Engines appear at the front; one Caboose at the end.
2. Boxcars always come in pairs: BB, BBBB, etc.
3. There cannot be more than four Passenger cars in a series.
4. One dining car must follow each series of passenger cars.

These cars cannot appear anywhere other than these locations. Here are some legal and illegal exemplars.

Problem 7:

EC

Legal: the smallest train

EEEEPPDBBPDBBBBC

Legal: simple train showing all the cars

EEBB

Illegal: no caboose (everything else OK)

EBBBC

Illegal: three boxcars in a row

EEPPPPPPDBBC

Illegal: more than four passenger cars in a row

EEPPBBC

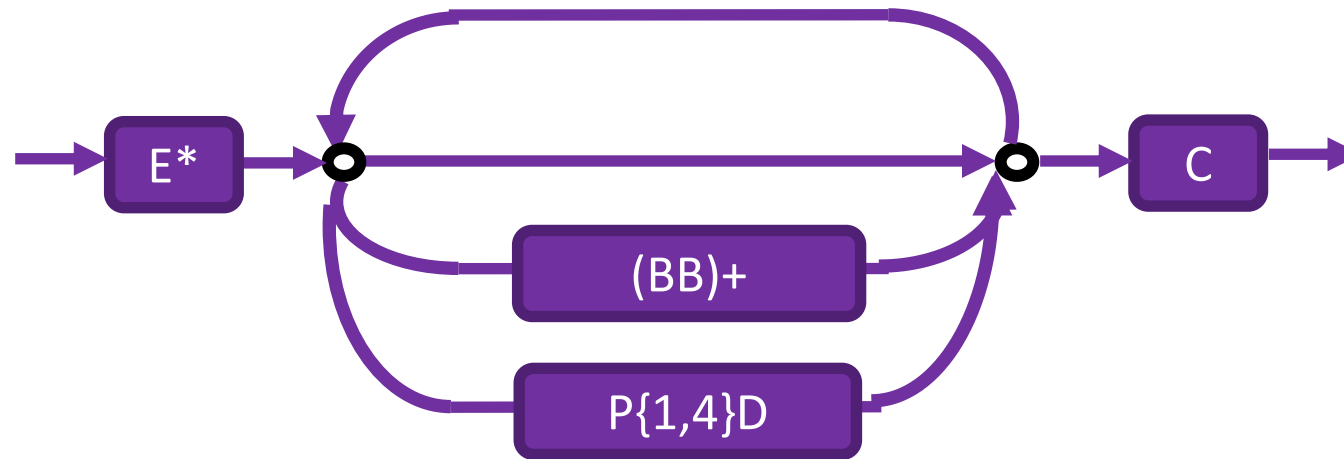
Illegal: no dining car after passenger cars

EEBBDC

Illegal: dining car after box car

Hint: my re pattern was 15 characters.

Syntax Diagram for Problem 7



Problem 7 Solution: train.py

```
# train.py  
# problem 7:  
# regular expression train  
import re  
# Sample strings.  
list = ["EC", "EEPPDBBPDBBBC", "EEBB", "EBBBC",  
        "EEPPPPDBBC", "EEPPBBC", "EEBBDC"]  
# Loop.  
for element in list:  
    # Match if two words starting with letter d.  
    #123456789012345678901  
    m = re.match("^E+(P{1,4}D|BB)*C$", element)  
    # See if success.  
    if m:  
        print(m.group(0))
```

Regular Expression Methods

Regex String Operations

SECTION 7

Regular Expression Methods and Regex Pattern Object

Pattern Matching Methods:

- Generally, the functions discussed in this lecture operate on a **regular expression pattern** and **text**.
- These functions produce information related to attempting to match the pattern and text: which parts of the text matched which parts of the pattern.

Pattern Object (Re-usable Regex):

We can use the compile function to compile a pattern (producing a regex), and then call methods on that regex directly, as an object to perform the same operations as the functions, but more efficiently if the **pattern** is to be used repeatedly.

Note: We will omit discussing/using the [,flags] option in this discussion, but see section 6.2 of the Python Library Documentation for a discussion of A/ASCII, DEBUG, I/IGNORECASE, L/LOCALE, M/MULTILINE, S/DOTALL, and X/VERBOSE.

Regular Expression Functions

```
import re
```

re.match():

```
re.match(pattern, string[, flags])
```

Example:

```
match-object = re.match(pattern, string_to_be_matched)
```

Return Value:

- Returns a **match object**, consisting of tuple of groups (0, 1, ...)
- Return None if the string does not match the pattern; note that this is different from a zero-length match.
- Matches start at the text's beginning

Python program that uses match

Demo Program: `re_match1.py`

```
import re
# Sample strings.
list = ["dog dot", "do don't", "dumb-dumb", "no match"]
# Loop.
for element in list:
    # Match if two words starting with letter d.
    m = re.match("(d\\w+)\\W(d\\w+)", element)
    # See if success.
    if m:
        print(m.groups())
```

Output

('dog', 'dot') ('do', 'don') ('dumb', 'dumb')

Pattern details

- **Pattern:** `(d\w+)\W(d\w+)`
- `d` Lowercase letter d.
- `\w+` One or more word characters.
- `\W` A non-word character.

Python program that tests starts, ends

Demo Program: `re_match2.py`

```
import re
list = ["123", "4cat", "dog5", "6mouse"]
for element in list:
    # See if string starts in digit.
    m = re.match("^\d", element)
    if m: print("START:", element)
    # See if string ends in digit.
    m = re.match(".*\d$", element)
    if m: print(" END:", element)
```

Results

Output

START: 123

END: 123

START: 4cat

END: dog5

START: 6mouse

Pattern details

`^\d` Match at the start, check for single digit.

`.*\d$` Check for zero or more of any char.

Check for single digit.

Match at the end.

Python program that uses re, expressions, repeats, or

Demo Program: `re_match3.py`

```
import re
values = ["cat100", "---200", "xxxyyy", "jjj", "box4000", "tent500"]
for v in values:
    # Require 3 letters OR 3 dashes.
    # ... Also require 3 digits.
    m = re.match("(?:?:\w{3})|(?:\-{3}))\d\d\d$", v)
    if m: print(" OK:", v)
    else: print("FAIL:", v)
```

Python program that uses re, expressions, repeats, or

Output

OK: cat100

OK: ---200

FAIL: xxxyyy

FAIL: jjj

FAIL: box4000

FAIL: tent500

Pattern details

(?:

The start of a non-capturing group.

\w{3}

Three word characters.

|

Logical or: a group within the chain must match.

\-

An escaped hyphen.

\d

A digit.

\$

The end of the string.

Regular Expression Functions

```
import re
```

re.search():

```
re.search(pattern, text [,flags])
```

Example:

```
match_object = re.search(pattern, string_to_be_searched)
```

Return Value:

- Returns a **match object**, consisting of tuple of only the first occurrence.
- Returns None if no match
- Matches can start anywhere in the text (different from re.match())

Python program that uses search

Demo Program: re_search1.py

```
import re
# Input.
value = "voorheesville"
m = re.search("(vi.*)", value)
if m: # This is reached.
    print("search:", m.group(1))
m = re.match("(vi.*)", value)
if m: # This is not reached.
    print("match:", m.group(1))
```

Output

search: ville

Pattern details

Pattern: (vi.*)

- vi** The lowercase letters v and i together.
- .*** Zero or more characters of any type.

Comparison between match() and search()

re.match():

`re.match("(a+)b","aaab")` matches;

`re.match("(a+)b","xaaab")` doesn't match

re.search():

`re.search("(a+)b","aaab")` matches;

`re.search("(a+)b","xaaab")` matches

by using patterns like `^...$`, these functions produce the same results

Regular Expression Functions

import re

re.findall():

re.findall(pattern, text [,flags])

Example:

- re.findall('a*b','**abaabc**bdabc') returns ['ab', 'aab', 'b', 'ab']
- re.findall('((a*)(b))','abaabc**bdabc**') returns [('ab','a','b'), ('aab','aa','b'), ('b','','b'), ('ab','a','b')]

Return Value:

- Returns a **list of string/of tuples of string** (the groups), specifying matches
- Matches can start anywhere in the text;
- The next attempted match starts one character after the previous match terminates.
- If the pattern has groups, then the string matching each group is included in the resulting list too: use **?:** to avoid these groups

Python program that uses findall

Demo Program: re_findall.py

```
import re
# Input.
value = "abc 123 def 456 dot map pat"
# Find all words starting with d or p.
list = re.findall("[dp]\w+", value)
# Print result.
print(list)
```

Output

['def', 'dot', 'pat']

Pattern details

Pattern: [dp]\w+

[dp] A lowercase d, or a lowercase p.

\w+ One or more word characters.

Regular Expression Functions

```
import re
```

re.finditer():

```
re.finditer(pattern, text [,flags])
```

Example:

- `iterable_object = re.finditer(pattern, string_to_be_operates)`

Return Value:

- Returns iterable equivalent of `findall`
- Returns an iterator **yielding** `match_object` instances over all non-overlapping matches for the **re** pattern in the string.
- Need to be operated with **`tuple(iterable_object.groups())`** to have same result as **`findall`**

Python program that uses finditer

Demo Program: `re_finditer.py`

```
import re
value = "123 456 7890"
# Loop over all matches found.
for m in re.finditer("\d+", value):
    print(m.group(0))
    print("start index:", m.start())
```

Output

```
123
start index: 0
456
start index: 4
7890
start index: 8
```

Example for: re.finditer() Demo Program:

```
>>> import re
>>> re.finditer(r'\w','http://www.hackerrank.com/')
<callable-iterator object at 0x0266C790>
>>> map(lambda x:
x.group(),re.finditer(r'\w','http://www.hackerrank.com/'))
['h', 't', 't', 'p', 'w', 'w', 'w', 'h', 'a', 'c', 'k', 'e', 'r', 'r', 'a', 'n', 'k', 'c', 'o', 'm']
```

Regular Expression Functions

```
import re
```

re.split():

```
re.split(pattern, text [,maxsplit, flags])
```

Example:

```
list_of_strings = re.split(pattern, string_to_be_split)
```

Return Value:

- Returns a **list of strings**: much like calling **text.split(...)**

Example for re.split()

- **re.split(pattern, text [,maxsplit, flags])** like the `text.split(...)` method, but using a regular expressions pattern to determine how to split the text:
- **re.split('\.|-', 'a.b-c')** returns `['a','b','c']`, splitting on either
 - . (which must be written here as `\.`) or
 - which standard string split function, **text.split(...)** can't do;

Note that `'a.b-c'.split(".-")` splits only on `'.-'` both a `.` followed by a `-`, so in this case it fails to split anywhere, since `'.-'` is not anywhere in the text at all.

- If the pattern has groups, then the text matching each group is included in the resulting list too: use `?:` to avoid these groups.

Example for re.split()

Examples:

`re.split(';+', 'abc;d;;e')` returns `['abc', 'd', 'e']`

`re.split('(;+)', 'abc;d;;e')` returns `['abc', ';', 'd', ';;', 'e']`

Python program that uses split

Demo Program: re_split1.py

```
import re
```

```
# Input string.
```

```
value = "one 1 two 2 three 3"
```

```
# Separate on one or more non-digit characters.
```

```
result = re.split("\\D+", value)
```

```
# Print results.
```

```
for element in result:
```

```
    print(element)
```

Output

1

2

3

Pattern details

Pattern: \\D+

\\D+ One or more non-digit characters.

Regular Expression Functions

import re

re.sub():

re.sub(pattern, repl, text, [,count, flags])

Example:

- re.sub('(a+)', '"**as**"', 'aabcaaadaf') returns "as"bc"as"d"as"f
- re.sub('(a+)', '(\g<1>)', 'aabcaaadaf') returns (aa)bc(aaa)d(a)f

Return Value:

- Returns a string

Explanation: re.sub()

re.sub(pattern, repl, text, [,count, flags])

- in text, replace pattern by **repl** (which may refer to matched groups via `\#` (e.g. `\1`) or `\g<#>`, (e.g., `\g<1>`), or `\g<name>`)
- **repl**: a replacement string or a **callable** (function, lambda) return string
- (where name comes from `?P<name>`) or a function that is passed a match object);
- if there is no match, then it just returns the text parameter's value, unchanged

Python program that uses string replacement

Demo Program: re_sub1.py

```
import re
# An example string.
v = "running eating reading"
# Replace words starting with "r" and ending in "ing"
# ... with a new string. *? Means .* using lazy algorithm
v = re.sub(r"r.*?ing", "ring", v)
print(v)
```

Output

ring eating ring

Python program that uses re.sub

Demo Program: re_sub2.py

```
import re
def multiply(m):
    # Convert group 0 to an integer.
    v = int(m.group(0))
    # Multiply integer by 2.
    # ... Convert back into string and return it.
    return str(v * 2)
# Use pattern of 1 or more digits.
# ... Use multiply method as second argument.
result = re.sub("\\d+", multiply, "10 20 30 40 50")
print(result)
```

Output

20 40 60 80 100

Python program that uses re.sub, lambda

Demo Program: re_sub3.py

```
import re
# The input string.
input = "laugh eat sleep think"
# Use lambda to add "ing" to all words.
result = re.sub("\\w+", lambda m: m.group(0) + "ing", input)
# Display result.
print(result)
```

Output

laughing eating sleeping thinking

Python program that uses re.sub with dictionary

Demo Program: re_sub4.py

```
import re
plants = {"flower": 1, "tree": 1, "grass": 1}
def modify(m):
    v = m.group(0)
    # If string is in dictionary, return different string.
    if v in plants:
        return "PLANT"
    # Do not change anything.
    return v
# Modify to remove all strings within the dictionary.
result = re.sub("\\w+", modify, "bird flower dog fish tree")
print(result)
```

Output

bird PLANT dog fish PLANT

Regular Expression Functions

```
import re
```

re.subn():

```
re.subn(pattern, repl, text, [,count, flags])
```

Example:

- re.subn

Return Value:

- same as sub but returns a **tuple**: (new string, number of subs made)

Python program that calls re.subn

Demo Program: re_subn.py

```
import re
def add(m):
    # Convert.
    v = int(m.group(0))
    # Add 2.
    return str(v + 1)
# Call re.subn.
result = re.subn("\\d+", add, "1 2 3 4 5")
print("Result string:", result[0])
print("Number of substitutions:", result[1])
```

Output

Result string: 2 3 4 5 6

Number of substitutions: 5

Regular Expression Functions

```
import re
```

re.escape():

```
re.escape(string)
```

Example:

- `re.escape('^a.*$')`

Return Value:

- Returns a string
- Return string with all non-alphanumerics backslashed; this is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it.

Non-Overlapping Matching

- In **findall** and **sub/subn**, only non-overlapping patterns are found/replaced:
- in text **aaaa** there are two non-overlapping occurrence of the pattern **aa**: starting in index 0 and 2 (not in index 1, which overlaps with the previous match in indexes **0-1**).

Regular Expression Methods

Regex Pattern Object and Operations

SECTION 8

Regular Expression Functions

```
import re
```

re.compile():

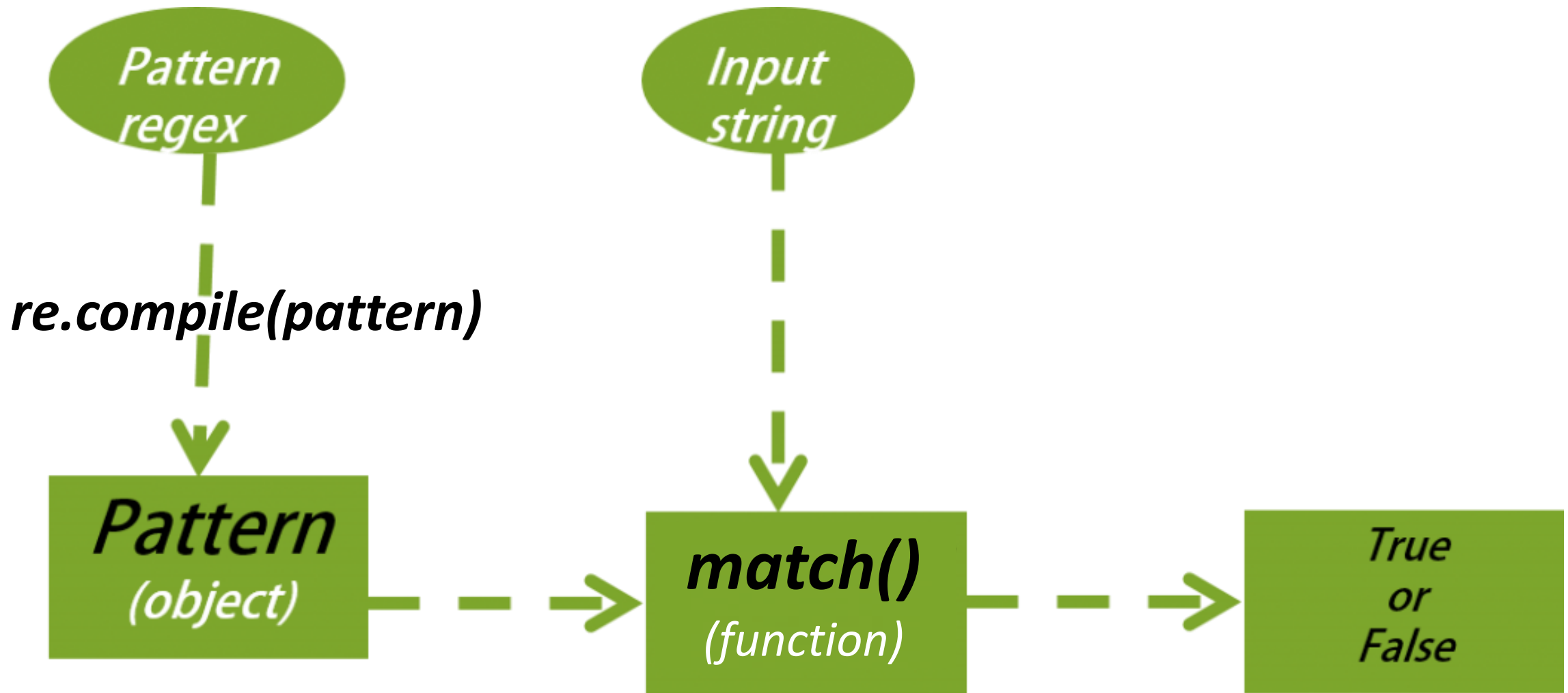
```
compile(pattern, [,flags])
```

Example:

```
compiled_regex_pattern_object = re.compile(pattern)
```

Return Value:

- Returns a regex (compiled pattern) object.

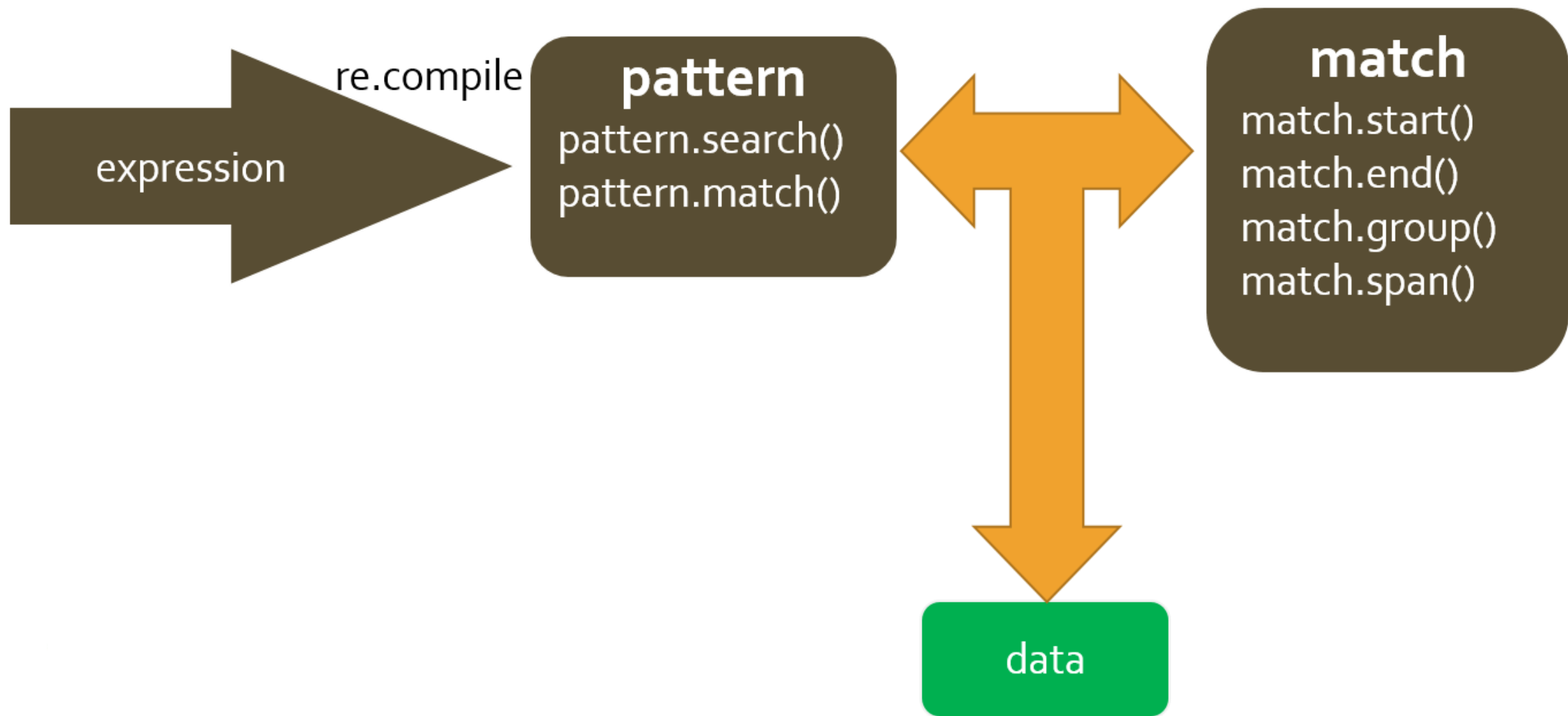


Regex

- regex (compiled pattern) object methods (see the compile method above, which produces regexes) are called like
c = re.compile(p)
- It is then efficient to call **c.match(...)** many times.
- Calling **re.match(p,...)** many times with the same pattern recompiles and matches the pattern each time **re.match** is called; whereas **c = re.compile(p)** compiles the pattern once and **c.match(...)** matches it each time it is called.

Regex

- Using this feature allows us to compile a pattern and reuse it for all the operations above: **`re.match(p,s)`** is just like **`re.compile(p).match(s)`**
- if we are doing many matches with the same pattern, compile the pattern once and use it with the match method below many times.
- **`pos/endpos`** are options that specify where in text the match starts and ends (from **`pos`** to **`endpos-1`**).
- **`pos`** defaults to **`0`** (the beginning of the text) and
- **`endpos`** defaults to the length of the text so **`endpos-1`** is its last character).



Instance Methods for Pattern Objects

Each of the re functions above has an equivalent method using a compiled pattern to call the method, but omitting the pattern from its argument list.

- **match(text [,pos][,endpos])** See match above, with pos/endpos
- **search(text [,pos][,endpos])** See search above, with pos/endpos
- **findall(text [,pos][,endpos])** See findall above, with pos/endpos
- **finditer(text [,pos][,endpos])** See finditer above, with pos/endpos
- **split (text [,maxsplit])** See split above, with pos/endpos
- **sub(repl, text [,count])** See sub above, with pos/endpos
- **subn(repl, text [,count])** See subn above, with pos/endpos

Re-writing of Python Programs with Regex

- So, for example, instead of writing
for line in open_file:
 ...re.match(pattern_string,line)
- which implicitly compiles the same **pattern_string** during each loop iteration (whenever **re.match** executes) we can write
for line in open_file:
 pattern = re.compile(pattern_string)
 ...pattern.match(line)

Re-writing of Python Programs with Regex

- which explicitly compiles the **pattern_string** during each loop and uses the compiled version (instead of the function `re.match`) to call "**match**" (just two ways of doing the same thing). If we know the **pattern_string** stays the same,

- we can also write

```
pattern = re.compile(pattern_string)
```

```
for line in open_file:
```

```
    ...pattern.match(line)
```

Re-writing of Python Programs with Regex

- which explicitly compiles the **pattern_string** ONCE, before the loop executes, and calls "**match**" on it during each loop iteration.
- See the **grep.py** module in the **remethods** download that accompanies this lecture for code that calls **re.compile**.

Match Objects and Groups

SECTION 9

Match Objects

Match objects record information about which parts of a pattern match the text. **Each group** (referred to by either its number or an optional name) can be used as an argument to a function that specifies information about the **start**, **end**, **span**, and **characters** in the matching text.

- Calling **match/search** produces **None** or a **match object**
- Calling **findall** produces **None**, a **list of strings** (if there are no groups) or a **list of tuples of strings** (if there are groups, with the tuple index representing the each group #)
- Calling **finditer** produces **None** or an **iterable of groups** (not used in the course)

Groups

1. Each **group** is indexed by a **number** or **name** (a name only when the group was delimited by (?P<name>)); **group 0** is all the character in the match, **groups 1-n** are for delimited matches inside.
 - For example, in the pattern (a)(b(c)(d)) the a is in group 1, the b is in group 2, c is in group 3, and d in is group 4: groups are numbered by in what order we reach their OPENING parenthesis. Technically, group 2 includes all of b(c)(d), the characters in groups 2-4.
2. Note that if a parenthesized expression looks like (?:...) it is NOT numbered as a group. So in (a)(?:b(c)(d)) the a is in group 1, the b is in NO group, c is in group 2, and d in is group 3.

Groups

- If a group is followed by a **?** and the pattern in the group is skipped, its group will be **None**.
- In the result of `re.match('a(b)?c','ac')` group 1 will be **None**. If the group itself is not optional, but the text inside the group is, the group will show as matching an empty string.
- So the result of `re.match('a(b?)c','ac')` group 1 will be **"" (Empty String)**.
- The same is true for a repetition that matches 0 times. Compare `re.match('a(b)*c','ac')` group 1 and `re.match('a(b*)c','ac')` group 1.

Groups

- If a group matches multiple times (e.g., `a(.)*c`), only its **last match** is available, so for `axyzc` group 1, the `(.)` group, is bound to the character `z`. If we wrote this as `a(.*)c` the `(.*)` group is bound to the characters **xyz**. If we wrote it as `a((.)*)c` **group 1** is **xyz** and **groups 2** is just **z**.
- Printing the groups of match object prints a tuple of the matching characters for each group 1-n (not group #0)

Match Objects' Methods

SECTION 10

Match Object's Methods

- We can look at each resulting group by its number (including group #0), using any of the following methods that operate on match objects
 - **group(g)** text of group with specified name or number
 - **group(g1,g2, ...)** tuple of text of groups with specified name or number
 - **groups()** tuple of text of all groups (can iterate over tuple)
 - **groupdict()** text of all groups as dict (see ?P<name>)
 - **start([group])** starting index of group (or entire matched string)
 - **end([group])** ending index of group (or entire matched string)
 - **span([group])** tuple: (start([group]), end([group]))
- Try doing some matches and calling .groups() on the result.

Group is a way of iterating through the string

Demo Program: [compile.py](#)

```
import re
t="Demoadmin, demo_ms1, demo_ms2, my_clustr1"
p=re.compile('(d\\w+)', re.I)
pos=0
while 1:
    m=p.search(t, pos)
    if m:
        print("Now search start position :", pos)
        print(m.group())
        pos = m.end()
    else:
        break
```

Diagram illustrating the use of the `pos` variable and the `m.end()` method to iterate through the string `t` using the `re.compile` pattern `(d\\w+)`.

- `pos` is the current search start position.
- `m.start()` returns the start position of the match.
- `m.end()` returns the end position of the match.
- `m.span()` returns a tuple containing the start and end positions of the match.

Regular expression FLAGS

re.I == re.IGNORECASE Ignore case

re.L == re.LOCALE Make \w, \b, and \s locale dependent

re.M == re.MULTILINE Multiline

re.S == re.DOTALL Dot matches all (including newline)

re.U == re.UNICODE Make \w, \b, \d, and \s unicode dependent

re.X == re.VERBOSE Verbose (unescaped whitespace in pattern is ignored, and '#' marks comment lines)

Packages for Example

Unzip **remethods.zip** and examine the `phonecall.py` and `readingtest.py` modules for examples of Python programs that use regular expressions (and groups) to perform useful computations.

Python program that uses groupdict

Demo Program: `groupdict.py`

```
import re
name = "Roberta Alden"
# Match names.
m = re.match("(?P<first>\w+)\W+(?P<last>\w+)", name)
if m: # Get dict. d is a list for dict {'first': 'Roberta', 'last': 'Alden'}
    d = m.groupdict()
    # Loop over dictionary with for-loop.
    for t in d:
        print(" key:", t)
        print("value:", d[t])
```

Output

key: last

value: Alden

key: first

value: Roberta

Python program that uses Regex comments (?#)

Demo Program: group1.py

```
import re
data = "bird frog"
# Use comments inside a regular expression.
m = re.match("(?#Before part).+?(?#Separator)\W(?#End part)(.+)", data)
if m: print(m.group(1))
```

Output

frog

Pattern details

(?#Before part)	Comment, ignored
.+?	As few characters as possible
(?#Separator)	Comment, ignored
\W	Non-word character
(?#End part)	Comment, ignored
(.+)	One or more characters, captured

Python program that uses not-followed-by pattern (?!)

Demo Program: `pattern1.py`

```
import re
data = "100cat 200cat 300dog 400cat 500car"
# Find all 3-digit strings except those followed by "dog" string.
# ... Dogs are not allowed.
m = re.findall("(?!\\d\\d\\ddog)(\\d\\d\\d)", data)
print(m)
```

Output

```
['100', '200', '400', '500']
```

Pattern details

<code>(?!\\d\\d\\ddog)</code>	Not followed by 3 digits and "dog"
<code>(\\d\\d\\d)</code>	3 digit value

A Simple but Illustrative Example

SECTION 11

Demo Program: phone.py

```
import re

phone = r'^(?:\((\d{3})\))?(\\d{3})[-.](\\d{4})$'

m = re.match(phone, '(949)824-2704')

assert m != None, 'No match'

print(m.groups())

area, exchange, number = [int(i) if i != None else None for i in m.group(1,2,3)]

print(area, exchange, number)
```

Regular Expression Example:

1) Here, phone is a pattern anchored at both ends.

(a) It starts with `^(?:\((\d{3})\))?`...

- controlling an optional area code.
- The `?:` means that the parentheses are not used to create a group, but are used with the `?` (option) symbol.
- Inside it is `\((\d{3})\)`: a left parenthesis, group 1 which consists of any 3 digits, and a right parenthesis.

(b) Next is `\d{3}` group 2, which consists of any 3 digits.

(c) Next is `[-.]` that is one symbol, either a - or . (not in a group).

(d) Next is `\d{4}` group 3, which consists of any 4 digits.

Regular Expression Example:

- 2) Calling the `re.match` function matches the pattern against some text, it returns a match object that is bound to `m`.
- 3) If the match `m` is **None**, there is no match (raises **AssertionError** exception).
- 4) Converts every non-None string from groups 1, 2, and 3 into an int.

Regular Expression Example:

5) Prints the the groups

Try also replacing line 2 by

<code>m = re.match(phone, '824-2704')</code>	<code># area is None</code>
<code>m = re.match(phone, '(949)824.2704')</code>	<code># . instead of -; no match</code>
<code>m = re.match(phone, '(94)824-2704')</code>	<code># only 2 in area code; no match</code>

Also, we can replace the first two lines by the following equivalent lines

```
phone_pat = re.compile(r'^(?:\((\d{3})\))?(?!\d{3})[-.](\d{4})$')  
m = phone_pat.match('(949)824-2704')
```

Extra Topics

SECTION 12

Raw Strings

Make Sure Escape Sequence Don't Happen for Regular Expression Pattern Strings

When writing regular expression pattern strings as arguments in Python it is best to use raw strings: they are written in the form `r'...'` or `r"..."`.

These should be used because of an issue dealing with using the backslash character in patterns, which is sometimes necessary.

For example, in regular strings when you write `'\n'` Python turns that into a **1** character string with the newline character: `len('\n')` is **1**. But with raw strings, writing `r'\n'` specifies a string with a backslash followed by an n: `len(r'\n')` is **2**. Normally this isn't a big issue because writing `'\d'` or `'*'` in regular strings doesn't generate an escape character, since there is no escape character for **d** or **(** so `len('\d')` and `len('*')` is **2**.

`**d` in function/method calls

(where `d` is a dict, variable parameter list)

- If we call a function we can specify `**d` as one or more of its arguments. For each `**d`, Python uses all its keys as parameter names and all its values as default arguments for these parameter names. For example
`f(**{'b':2, 'a':1, 'c':3})` is translated by Python into `f(b=2,a=1,c=3)`
- Note that this is useful in regular expressions if we use the `(?P<name> ...)` option and then the `groupdict()` method for the match it produces.
- There is also a version that works the other way. Suppose we have a functions
- whose header is

```
def f(x,y,**kargs): # The typical name is **kargs
```

`**d` in function/method calls (where `d` is a dict)

- if we call it by `f(1,2,a=3,b=4,c=5)` then
 - `x` is bound to 1
 - `y` is bound to 2
 - `kargs` is bound to a dictionary `{'b':4, 'a':3, 'c':5}`
- See the argument/parameter matching rules from the review lecturer for a complete description of what happens.
- So (in reverse of the order explained above) `**` as a parameter creates a dictionary of "extra" named-arguments supplied when the function is called, and `**` as an argument supplies lots of named-arguments to the function call. We will cover this information again when we examine inheritance
- The `parse_phone_named` method (in `phoncecall.py`) uses this language feature.

Translation of a Regular Expression Pattern into a NDFA

- How do the functions/methods in re compile a regular expression and match it against text? It translates every regular expression into a non-deterministic finite automaton (see Programming Assignment #1), and then matches against the text (ibid) to see if the match succeeds (reaches the special last state).
- The general algorithm (known as **Thompson's Algorithm**) is a bit beyond the scope of this course and uses a concept we haven't discussed (epsilon transitions), but you can look up the details if you are interested. Here is an example for the regular expression pattern **((a* | b)cd)+**. It produces an **NDFA** described by

Translation of a Regular Expression Pattern into a NDFA

start;a;1;a;2;b;2;c;3

1;a;1;a;2

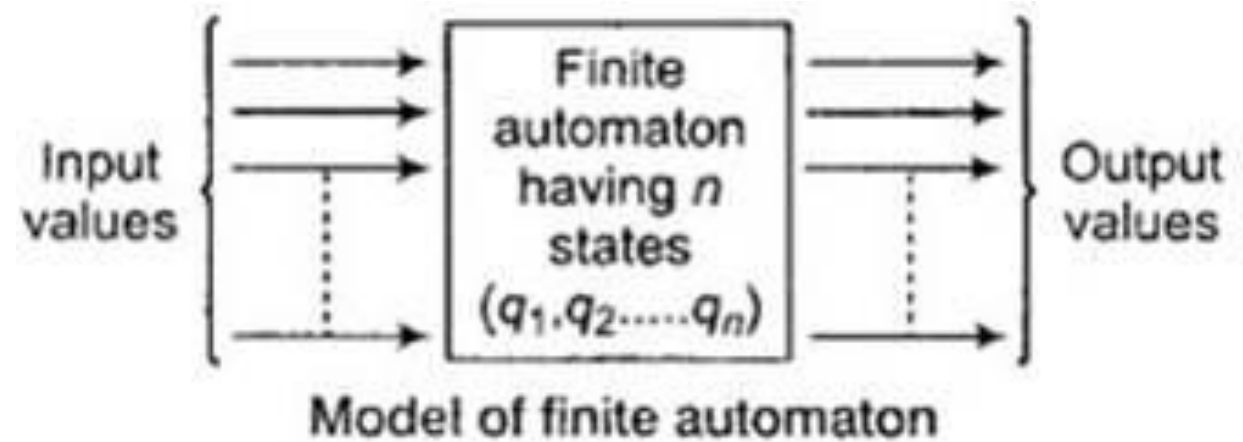
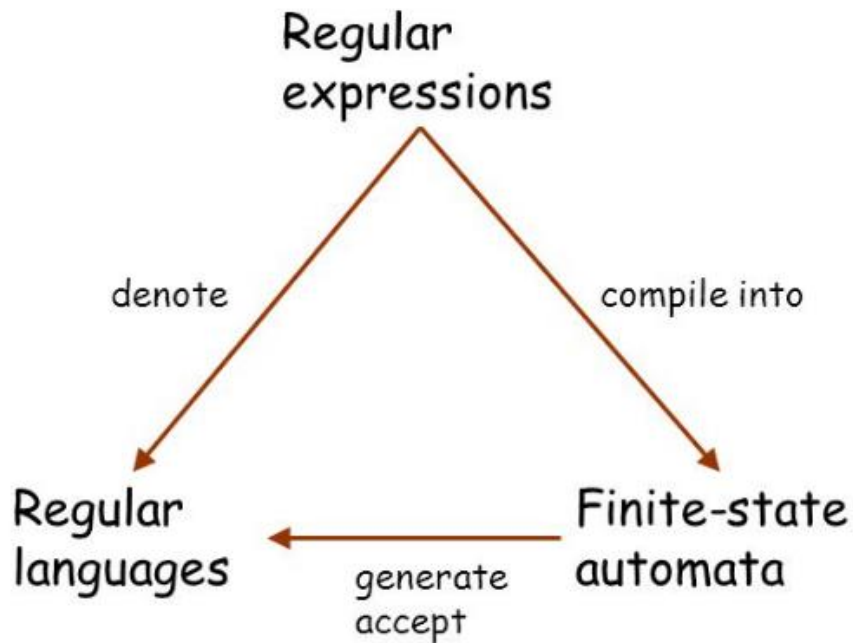
2;c;3

3;d;start;d;last

last

- This pattern matches a text string by starting in state '**start**' and exhausting all the characters and having '**last**' in its possible states.

Regular Expression, Regular Languages, Finite State Automata, and Finite State Machine



Conversion of Regular Expression to Finite State Automaton

▲ Starting state

Regular Expression

Finite Automaton

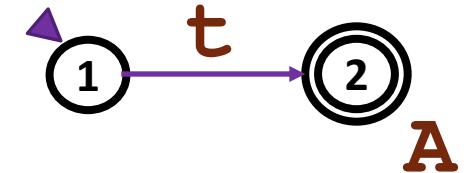
Regular Grammar Rules

Finite Automaton

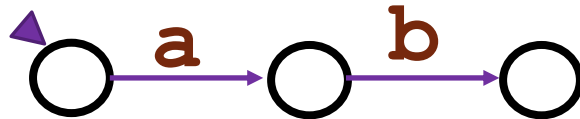
a



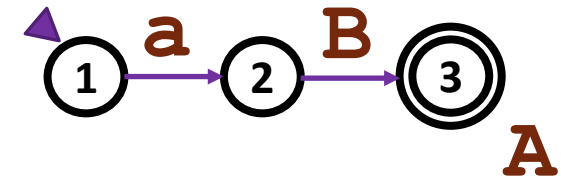
$A \rightarrow t$



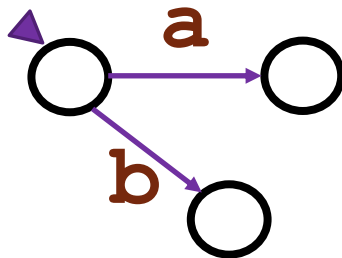
ab



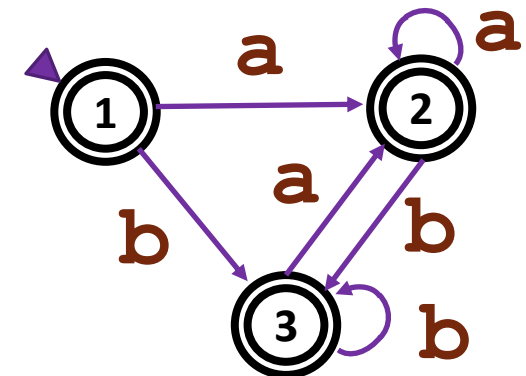
$A \rightarrow aB$



$a | b$



$A \rightarrow (a | b)^*$



a^*



Regular Expression Designs (Part 2)

SECTION 13

Problems:

Write functions using regular expression patterns

8. Write a function named **contract** that takes a string as a parameter.

It substitutes the word 'goal' to replace any occurrences of variants of this word written with any number of o's, e.g., 'goooooal') in its argument. So calling `contract('It is a gooooooal! A gooal.')` returns 'It is a goal! A goal.'.

Problem 8 Solution: goal.py

```
# goal.py  
# string substitute (replace)  
import re  
  
def contract(v):  
    v = re.sub(r"go+al", "goal", v)  
    return v  
  
v = contract("goal goal gooooooal goooooooal goooooooooal"+  
             " goal!! gooooooal!! goal!!!")  
print(v)
```

Problem 8 Solution: goal2.py

```
# goal2.py  
# string substitute (replace)  
import re  
  
def contract(v):  
    p = re.compile(r"goal")  
    v = p.sub("goal", v)  
    return v  
  
v = contract("goal goal gooooooal gooooooal gooooooooooal"+  
            " goal!! gooooooal!! goal!!!")  
  
print(v)
```

Problems:

Write functions using regular expression patterns

9. Write a function named **grep** that takes a regular expression pattern string and a file name as parameters.

It returns a list of 3-tuples consisting of the file-name, line number, and line of the file, for each line whose text matches the pattern.

Hint: Using `enumerate` and a comprehension, this is a 3 line function, but you can use explicit looping in a longer function.

Problem 9 Solution: grep.py

```
# grep.py
import re
def grep(filename, pattern):
    list=[]
    i = 1
    for line in open(filename, 'r').readlines():
        line = line.rstrip()
        m = pattern.match(line)
        if m: list.append((filename, i, line))
        i += 1
    return list
def main():
    filename = "data.txt"
    pattern = re.compile("(0|-?[1-9]\d*)")
    list = grep(filename, pattern)
    for f, lineno, st in list:
        print("%-10s Line %d %s" % (f, lineno, st))
if __name__ == "__main__":
    main()
```

Problems:

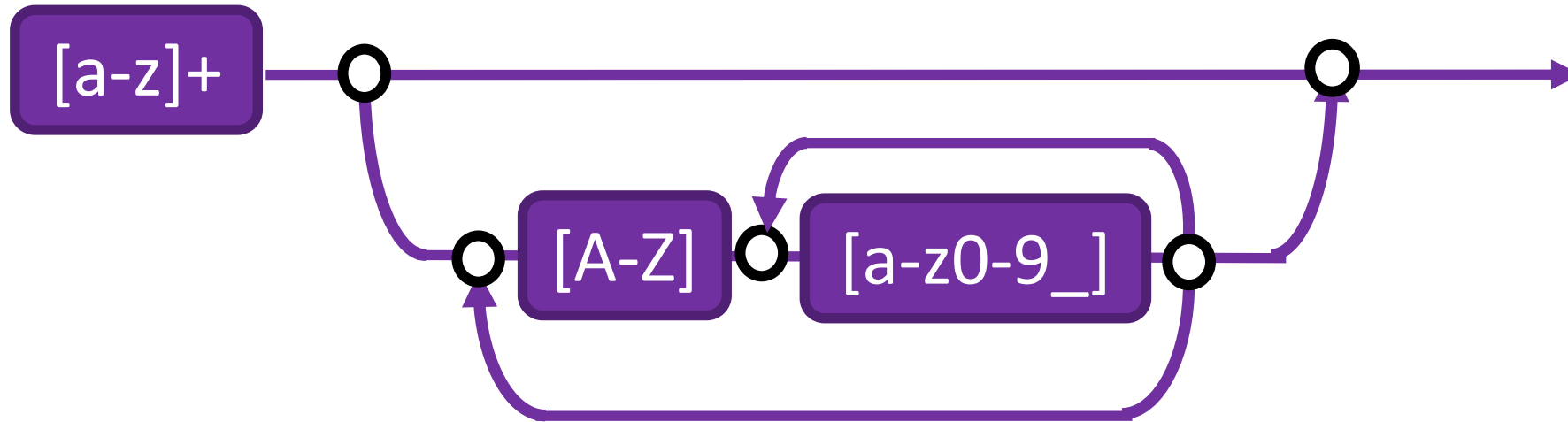
Write functions using regular expression patterns

10. Write a function named **name_convert** that takes two file names as parameter.

It reads the first file (which should be a Python program) and writes each line into the second file, but with identifiers originally written in camel notation converted to underscore notation:

e.g. **aCamelName** converts to **a_camel_name**. **Camel** identifiers start with a lower-case letter followed by upper/lower-case letters and digits: each upper-case letter is preceded by an underscore and turned into a lower-case letter.

Java ID pattern



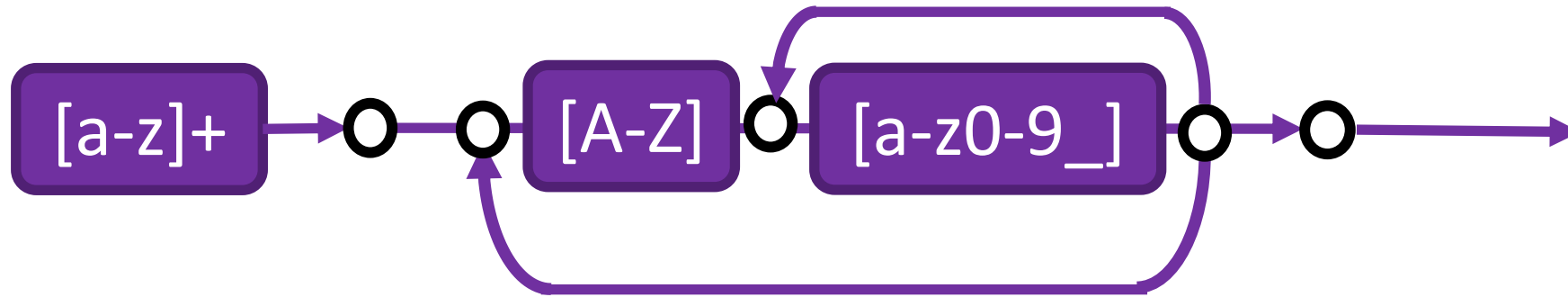
Java Variable (id) Convention:

Lower case for learning word, Uppercase for other words.

variableForStudentGrade

variable

Java ID pattern (Words need Replacement)



Java Variables (id) those Need Conversion:

Lower case for learning word, Uppercase for other words.
variableForStudentGrade

Pseudo Code

1. Open a text file
2. For each line in the text file.
 - replace the tokens that matches the Java ID Regular
Expression Pattern
 - add the line to a list.
3. close the text file.
4. Write all of the lines in the list back to the text file.

Demo Program 1: java_match.py

Design the Java id matching regular expression:

```
# java_match.py
import re
list = ["javaVariable", "computer
science", "variableA", "floatingNumber",
"initialVariableValue"]
for element in list:
    m = re.match("[a-z]+([A-Z][a-z0-9_]*)*",
element)
    if m:
        print(m.group(0))
```

Demo Program 2: java_iteration.py

Use Regular Expression to find the Characters to be replaced:

```
def main():  
    name_convert("Loan.java", "Loan2.java")  
  
if __name__ == "__main__":  
    main()
```

Demo Program 2: java_iteration.py

```
# java_replacement.py
import re
def calculate_new_string(token): # replacement of C by _c
    ch_list = []
    for j in range(len(token)):
        if (token[j].isupper() and j!=0):
            ch_list.append(token[j])
    for ch in ch_list:
        token = re.sub(ch, "_" + ch.lower(), token)
    return token
```

```

def name_convert(filename1, filename2):
    f1 = open(filename1, "r")
    p = re.compile("[a-z]+([A-Z][a-z0-9_]*)+")
    newlines = []
    for line in f1.readlines():
        replacement_list = []
        new_list = []
        pos = 0
        m = p.search(line, pos)
        while m:  # find all id's which need to be modified
            pos = m.end()
            replacement_list.append(m.group())
            new_list.append(calculate_new_string(m.group()))
            m = p.search(line, pos)
        for pair in list(zip(replacement_list, new_list)):
            line = re.sub(pair[0], pair[1], line)
        newlines.append(line)
    f1.close()
    f2 = open(filename2, "w")
    for line in newlines:
        f2.write(line)
    f2.close()

```

Java Style Variable Name Regex Pattern

List to hold lines after conversion

Search for a new match

While the line has new matches

The id to be replaced

The id after conversion

Zip the old and new string together and make it iterable as list

Convert the line

Problem 10: Conversion of Variable Names from Java Style to C/C++ (Python) Style

```
1 public class Loan { // Loan.java (part)
2     private double annualInterestRate;
3     private int numberOfYears;
4     private double loanAmount;
5     private java.util.Date loanDate;
6
7     /** Default constructor */
8     public Loan() {
9         this(2.5, 1, 1000);
10    }
11
12    /** Construct a loan with specified annual interest rate,
13        number of years, and loan amount
14    */
15    public Loan(double annualInterestRate, int numberOfYears,
16        double loanAmount) {
17        this.annualInterestRate = annualInterestRate;
18        this.numberOfYears = numberOfYears;
19        this.loanAmount = loanAmount;
20        loanDate = new java.util.Date();
21    }
22 }
```

```
1 public class Loan { // Loan2.java (part)
2     private double annual_interest_rate;
3     private int number_of_years;
4     private double loan_amount;
5     private java.util.Date loan_date;
6
7     /** Default constructor */
8     public Loan() {
9         this(2.5, 1, 1000);
10    }
11
12    /** Construct a loan with specified annual interest rate,
13        number of years, and loan amount
14    */
15    public Loan(double annual_interest_rate, int number_of_years,
16        double loan_amount) {
17        this.annual_interest_rate = annual_interest_rate;
18        this.number_of_years = number_of_years;
19        this.loan_amount = loan_amount;
20        loan_date = new java.util.Date();
21    }
22 }
```


Overview of Python Lex and Yacc (PLY)

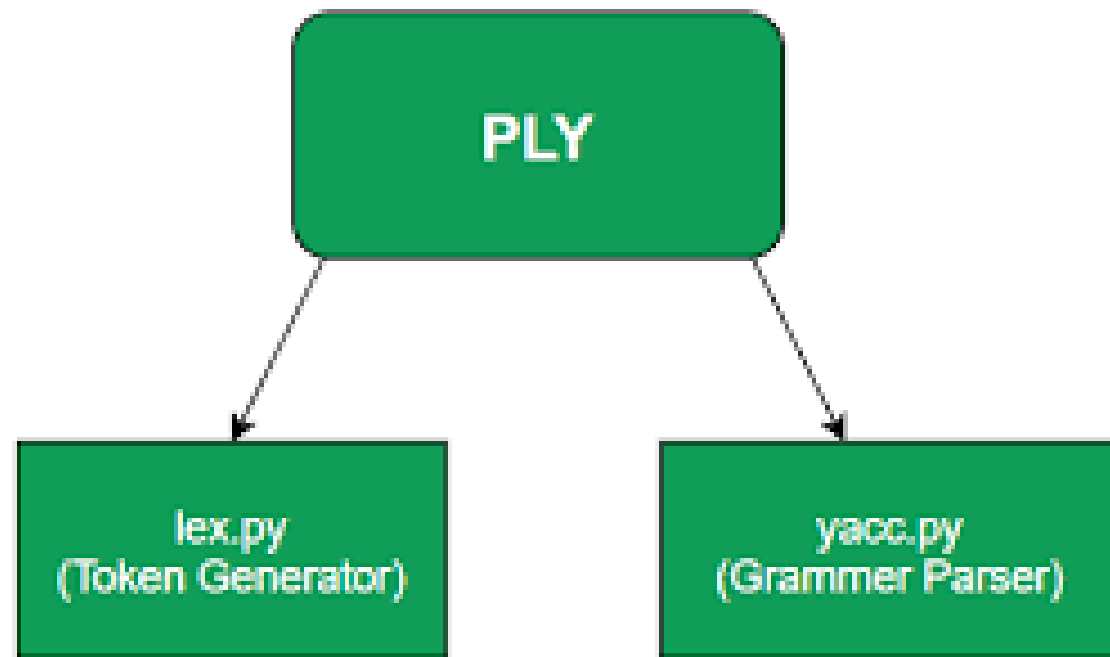
SECTION 14

PLY (Python Lex-Yacc)

<https://www.dabeaz.com/ply/>

In a nutshell, PLY is nothing more than a straightforward lex/yacc implementation. Here is a list of its essential features:


- It's implemented entirely in **Python**.
- It uses **LR-parsing** which is reasonably efficient and well suited for larger grammars.
- PLY provides most of the standard **lex/yacc** features including support for empty productions, precedence rules, error recovery, and support for ambiguous grammars.
- PLY is straightforward to use and provides *very* extensive error checking.
- PLY doesn't try to do anything more or less than provide the basic **lex/yacc** functionality. In other words, it's not a large parsing framework or a component of some larger system.



Installation of PLY

- <https://pypi.org/project/ply/>

Python PLY Installation



[Help](#) [Sponsors](#) [Log in](#) [Register](#)

ply 3.11

✓

[Latest version](#)

`pip install ply` 

Released: Feb 15, 2018

Python Lex & Yacc

Navigation

 Project description

 [Release history](#)

 [Download files](#)

Project description

PLY is yet another implementation of lex and yacc for Python. Some notable features include the fact that its implemented entirely in Python and it uses LALR(1) parsing which is efficient and well suited for larger grammars.

PLY provides most of the standard lex/yacc features including support for empty productions, precedence rules, error recovery, and support for ambiguous grammars.

PLY is extremely easy to use and provides very extensive error checking. It is compatible with both Python 2 and Python 3.

Project links

 [Homepage](#)

```
c:\Python\Python36>pip install ply
```

```
Collecting ply
```

```
  Downloading https://files.pythonhosted.org/packages/a3/58/35da89ee790598a0700ea49b2a665  
94140f44dec458c07e8e3d4979137fc/ply-3.11-py2.py3-none-any.whl (49kB)
```

```
    100% |████████████████████████████████████████| 51kB 1.6MB/s
```

```
Installing collected packages: ply
```

```
Successfully installed ply-3.11
```

```
You are using pip version 18.1, however version 21.3.1 is available.
```

```
You should consider upgrading via the 'python -m pip install --upgrade pip' command.
```

```
c:\Python\Python36>_
```

A Simple Example

- Calculator Language

Lex

SECTION 2

Lex

- lex.py is used to tokenize an input string. For example, suppose you're writing a programming language and a user supplied the following input string:

```
x = 3 + 42 * (s - t)
```

- A tokenizer splits the string into individual tokens:

```
'x', '=', '3', '+', '42', '*', '(', 's', '-', 't', ')'
```

- Tokens are usually given names to indicate what they are. For example:

```
'ID', 'EQUALS', 'NUMBER', 'PLUS', 'NUMBER', 'TIMES',  
'LPAREN', 'ID', 'MINUS', 'ID', 'RPAREN'
```

Lex

- More specifically, the input is broken into pairs of token types and values. For example:

```
('ID', 'x'), ('EQUALS', '='), ('NUMBER', '3'),  
('PLUS', '+'), ('NUMBER', '42'), ('TIMES', '*'),  
('LPAREN', '('), ('ID', 's'), ('MINUS', '-'),  
('ID', 't'), ('RPAREN', ')'
```

- The specification of tokens is done by writing a series of regular expression rules. The next section shows how this is done using `lex.py`.

Lex Example

- The following example shows how lex.py is used to write a simple tokenizer:

```
import ply.lex as lex

# List of token names.      This is always required
tokens = (
    'NUMBER',
    'PLUS',
    'MINUS',
    'TIMES',
    'DIVIDE',
    'LPAREN',
    'RPAREN',
)

# Regular expression rules for simple tokens
t_PLUS = r'\+'
t_MINUS = r'\-'
t_TIMES = r'\*'
t_DIVIDE = r'\/'
t_LPAREN = r'\('
t_RPAREN = r'\)'
```

```
# A regular expression rule with some action code
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Define a rule so we can track line numbers
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# A string containing ignored characters (spaces and tabs)
t_ignore = ' \t'

# Error handling rule
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)

# Build the lexer
lexer = lex.lex()
```

Testing It

- To use the lexer, you first need to feed it some input text using its `input()` method. After that, repeated calls to `token()` produce tokens. The following code shows how this works:

Output

calclex.txt

```
LexToken (NUMBER, 3, 2, 1)
LexToken (PLUS, '+', 2, 3)
LexToken (NUMBER, 4, 2, 5)
LexToken (TIMES, '*', 2, 7)
LexToken (NUMBER, 10, 2, 9)
LexToken (PLUS, '+', 3, 14)
LexToken (MINUS, '-', 3, 16)
LexToken (NUMBER, 20, 3, 17)
LexToken (TIMES, '*', 3, 20)
LexToken (NUMBER, 2, 3, 21)
```


Lexer

- Lexers also support the iteration protocol. So, you can write the above loop as follows:

```
for tok in lexer:  
    print(tok)
```

Tokens

- The tokens returned by `lexer.token()` are instances of `LexToken`. This object has attributes `type`, `value`, `lineno`, and `lexpos`. The following code shows an example of accessing these attributes:

```
# Tokenize
while True:
    tok = lexer.token()
    if not tok:
        break          # No more input
    print(tok.type, tok.value, tok.lineno, tok.lexpos)
```

- The `type` and `value` attributes contain the type and value of the token itself. `lineno` and `lexpos` contain information about the location of the token. `lexpos` is the index of the token relative to the start of the input text.

Small Calculator

SECTION 15



EXPLORER



calc.py ×



OPEN EDITORS

× calc.py

LECTURE 11

> _pycache_

calc.py

parser.out

parsetab.py

calc.py > ...

```
1  # -----
2  # calc.py
3  #
4  # A simple calculator with variables -- all in one file.
5  # -----
6
7  tokens = (
8      'NAME', 'NUMBER',
9      'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'EQUALS',
10     'LPAREN', 'RPAREN',
11     )
12
13 # Tokens
14
```

PROBLEMS

OUTPUT

TERMINAL

python + ▾ □ ✕ ^ ×

```
C:\Eric_Chou\Lewis University\CS 46K SIPC\PyDev\Lecture 11>python calc.py
calc > (3+5)
8
calc > (4+6+(2+7))
19
calc > 
```

> OUTLINE

```
tokens = (  
    'NAME', 'NUMBER',  
    'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'EQUALS',  
    'LPAREN', 'RPAREN',  
)
```

```
# Tokens
```

```
t_PLUS      = r'\+'
```

```
t_MINUS     = r'\-'
```

```
t_TIMES     = r'\*'
```

```
t_DIVIDE    = r'\/'
```

```
t_EQUALS    = r'='
```

```
t_LPAREN    = r'\('
```

```
t_RPAREN    = r'\)'
```

```
t_NAME      = r'[a-zA-Z_][a-zA-Z0-9_]*'
```

```
def t_NUMBER(t):  
    r'\d+'  
    try:  
        t.value = int(t.value)  
    except ValueError:  
        print("Integer value too large %d", t.value)  
        t.value = 0  
    return t
```

Ignored characters

```
t_ignore = " \t"
```

```
def t_newline(t):  
    r'\n+'  
    t.lexer.lineno += t.value.count("\n")
```

```
def t_error(t):  
    print("Illegal character '%s'" % t.value[0])  
    t.lexer.skip(1)
```

Build the Lexer

calc.py

```
# Build the lexer  
import ply.lex as lex  
lexer = lex.lex()
```

```
# Parsing rules

precedence = (
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE'),
    ('right', 'UMINUS'),
)

# dictionary of names
names = { }

def p_statement_assign(t):
    'statement : NAME EQUALS expression'
    names[t[1]] = t[3]

def p_statement_expr(t):
    'statement : expression'
    print(t[1])
```

```
def p_expression_binop(t):  
    '''expression : expression PLUS expression  
                  | expression MINUS expression  
                  | expression TIMES expression  
                  | expression DIVIDE expression'''  
    if t[2] == '+': t[0] = t[1] + t[3]  
    elif t[2] == '-': t[0] = t[1] - t[3]  
    elif t[2] == '*': t[0] = t[1] * t[3]  
    elif t[2] == '/': t[0] = t[1] / t[3]  
  
def p_expression_uminus(t):  
    'expression : MINUS expression %prec UMINUS'  
    t[0] = -t[2]  
  
def p_expression_group(t):  
    'expression : LPAREN expression RPAREN'  
    t[0] = t[2]  
  
def p_expression_number(t):  
    'expression : NUMBER'  
    t[0] = t[1]
```

```
def p_expression_name(t):  
    'expression : NAME'  
    try:  
        t[0] = names[t[1]]  
    except LookupError:  
        print("Undefined name '%s'" % t[1])  
        t[0] = 0  
  
def p_error(t):  
    print("Syntax error at '%s'" % t.value)
```

Build the Parser

calc.py

```
import ply.yacc as yacc
parser = yacc.yacc()

while True:                                # Loop
    try:
        s = input('calc > ')              # Use raw_input on Python 2    # Read
    except EOFError:
        break
    parser.parse(s)                         # Evaluate and Print
```

Symbol Table

[parsertab.py](#)

- Similar to `calc.tab.c` and `calc.tab.h`
- Symbol Table
- Action Routines (See Lecture 13 Parser)
- Production Rules

```

# parsetab.py
# This file is automatically generated. Do not edit.
# pylint: disable=W,C,R
_tabversion = '3.10'

_lr_method = 'LALR'

_lr_signature = 'leftPLUSMINUSleftTIMESDIVIDERightUMINUSDIVIDE EQUALS LPAREN MINUS NAME NUMBER PLUS RPAREN TIMESstatement : NAME
EQUALS expressionstatement : expressionexpression : expression PLUS expression\n                | expression MINUS
expression\n                | expression TIMES expression\n                | expression DIVIDE expressionexpression : MINUS
expression %prec UMINUSexpression : LPAREN expression RPARENexpression : NUMBERexpression : NAME'

_lr_action_items =
{'NAME':([0,4,5,7,8,9,10,11,],[2,13,13,13,13,13,13,13,]),'MINUS':([0,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,],[4,-
10,9,4,4,-9,4,4,4,4,4,-7,-10,9,9,-3,-4,-5,-6,-
8,]),'LPAREN':([0,4,5,7,8,9,10,11,],[5,5,5,5,5,5,5,5,]),'NUMBER':([0,4,5,7,8,9,10,11,],[6,6,6,6,6,6,6,6,]),'$end':([1,2,3,6,12,13,
15,16,17,18,19,20,],[0,-10,-2,-9,-7,-10,-1,-3,-4,-5,-6,-8,]),'EQUALS':([2,],[7,]),'PLUS':([2,3,6,12,13,14,15,16,17,18,19,20,],[ -
10,8,-9,-7,-10,8,8,-3,-4,-5,-6,-8,]),'TIMES':([2,3,6,12,13,14,15,16,17,18,19,20,],[ -10,10,-9,-7,-10,10,10,10,10,-5,-6,-
8,]),'DIVIDE':([2,3,6,12,13,14,15,16,17,18,19,20,],[ -10,11,-9,-7,-10,11,11,11,11,-5,-6,-
8,]),'RPAREN':([6,12,13,14,16,17,18,19,20,],[ -9,-7,-10,20,-3,-4,-5,-6,-8,]),}

_lr_action = {}
for _k, _v in _lr_action_items.items():
    for _x, _y in zip(_v[0],_v[1]):
        if not _x in _lr_action: _lr_action[_x] = {}
        _lr_action[_x][_k] = _y
del _lr_action_items

_lr_goto_items = {'statement':([0,],[1,]),'expression':([0,4,5,7,8,9,10,11,],[3,12,14,15,16,17,18,19,]),}

```

```
_lr_goto = {}
for _k, _v in _lr_goto_items.items():
    for _x, _y in zip(_v[0], _v[1]):
        if not _x in _lr_goto: _lr_goto[_x] = {}
        _lr_goto[_x][_k] = _y
del _lr_goto_items
_lr_productions = [
    ("S' -> statement", "S'", 1, None, None, None),
    ('statement -> NAME EQUALS expression', 'statement', 3, 'p_statement_assign', 'calc.py', 60),
    ('statement -> expression', 'statement', 1, 'p_statement_expr', 'calc.py', 64),
    ('expression -> expression PLUS expression', 'expression', 3, 'p_expression_binop', 'calc.py', 68),
    ('expression -> expression MINUS expression', 'expression', 3, 'p_expression_binop', 'calc.py', 69),
    ('expression -> expression TIMES expression', 'expression', 3, 'p_expression_binop', 'calc.py', 70),
    ('expression -> expression DIVIDE expression', 'expression', 3, 'p_expression_binop', 'calc.py', 71),
    ('expression -> MINUS expression', 'expression', 2, 'p_expression_uminus', 'calc.py', 78),
    ('expression -> LPAREN expression RPAREN', 'expression', 3, 'p_expression_group', 'calc.py', 82),
    ('expression -> NUMBER', 'expression', 1, 'p_expression_number', 'calc.py', 86),
    ('expression -> NAME', 'expression', 1, 'p_expression_name', 'calc.py', 90),
]
```

Parser Output

parser.out

- Summary of the execution of the parser

Grammar

- Rule 0 S' -> statement
 - Rule 1 statement -> NAME EQUALS expression
 - Rule 2 statement -> expression
 - Rule 3 expression -> expression PLUS expression
 - Rule 4 expression -> expression MINUS expression
 - Rule 5 expression -> expression TIMES expression
 - Rule 6 expression -> expression DIVIDE expression
 - Rule 7 expression -> MINUS expression
 - Rule 8 expression -> LPAREN expression RPAREN
 - Rule 9 expression -> NUMBER
 - Rule 10 expression -> NAME
-

Terminals, with rules where they appear

DIVIDE	: 6
EQUALS	: 1
LPAREN	: 8
MINUS	: 4 7
NAME	: 1 10
NUMBER	: 9
PLUS	: 3
RPAREN	: 8
TIMES	: 5
error	:

Nonterminals, with rules where they appear

expression	: 1 2 3 3 4 4 5 5 6 6 7 8
statement	: 0

(... omitted)



End of Chapter 8B
