CMSE520 Fall 2024 Problem session 01.11.2024

Topics covered: 1)  [Lecture 1. Software Evolution & Maintenance Concepts](https://staff.emu.edu.tr/alexanderchefranov/Documents/CMSE520/CMSE520%20Fall%202024/Introduction%2021092024.docx" \t "_blank), 2) [Lecture 2. Taxonomy of Software Maintenance and Evolution](https://staff.emu.edu.tr/alexanderchefranov/Documents/CMSE520/CMSE520%20Fall%202024/Lecture%202.%20Taxonomy%20of%20software%20maintenance%20and%20evolution%2022092024.docx), and 3) [Lecture 3. Reengineering](https://staff.emu.edu.tr/alexanderchefranov/Documents/CMSE520/CMSE520%20Fall%202024/Lecture%203.%20Reengineering%20.docx) (p. 1-22; Section 5 and the rest is not included)

# Reengineering

## Program metrics

**TABLE 4.3 Commonly Used Software Metrics**

Metric Description

Lines of code (LOC) The number of lines of executable code

Global variable (GV) The number of global variables

### Cyclomatic complexity (CC)

The number of linearly independent paths in a program unit is given by the cyclomatic complexity metric.

From <https://en.wikipedia.org/wiki/Cyclomatic_complexity>:

|  |  |
| --- | --- |
| It is proposed to use a graph in which each exit point is connected back to the entry point. In this case, the graph is [strongly connected](https://en.wikipedia.org/wiki/Strongly_connected). Here, the cyclomatic complexity of the program is equal to the [cyclomatic number](https://en.wikipedia.org/wiki/Cyclomatic_number" \o "Cyclomatic number) of its graph (also known as the [first Betti number](https://en.wikipedia.org/wiki/Betti_number#Example_2:_the_first_Betti_number_in_graph_theory)), which is defined as[[2]](https://en.wikipedia.org/wiki/Cyclomatic_complexity#cite_note-mccabe76-2)  M=E−N+P.  where *E* = the number of edges of the graph, *N* = the number of nodes of the graph, *P* = the number of [connected components](https://en.wikipedia.org/wiki/Component_(graph_theory)).  This may be seen as calculating the number of [linearly independent cycles](https://en.wikipedia.org/wiki/Linearly_independent_cycle) that exist in the graph: those cycles that do not contain other cycles within themselves. Because each exit point loops back to the entry point, there is at least one such cycle for each exit point. | undefined  Each exit point is connected back to the entry point. This graph has 10 edges, eight nodes and one [connected component](https://en.wikipedia.org/wiki/Connected_component_(graph_theory)), which also results in a cyclomatic complexity of 3 (10 − 8 + 1 = 3). |
| **if** (c1())  f1();  **else**  f2();  **if** (c2())  f3();  **else**  f4();  The cyclomatic complexity of the program is 3 (as the strongly connected graph for the program contains 9 edges, 7 nodes, and 1 connected component) (9 − 7 + 1). | undefined  F  E  D  C  B  A  S |

There are 4 cycles (paths) but only 3 of them are linearly independent. McCabe developed an algorithmic procedure (called the “baseline method”) to determine a set of basis paths. The method begins with the selection of a “baseline” path, which should correspond to some “normal case” program execution. This can be somewhat arbitrary; McCabe advises choosing a path with as many decision nodes as possible. Next the baseline path is retraced, and in turn each decision is “flipped”, that is when a node of outdegree ≥2 is reached, a different edge must be taken. Apply it:

P1=(S,A,C,D,F); P2=(S,B,C,D,F), and P3=(S,B,C,E,F).

Read coupling The number of global variables read by a program unit

Write coupling The number of global variables updated by a program unit

Address coupling The number of global variables whose addresses are extracted by a program unit but do not involve read/write coupling

Fan-in The number of other functions calling a given function in a module

Fan-out The number of other functions being called from a given function in a module

### Module coupling

<https://en.wikipedia.org/wiki/Coupling_%28computer_programming%29#:~:text=In%20software%20engineering%2C%20coupling%20is%20the%20degree%20of,between%20modules.%20Coupling%20is%20not%20binary%20but%20multi-dimensional>.

For data and control flow coupling:

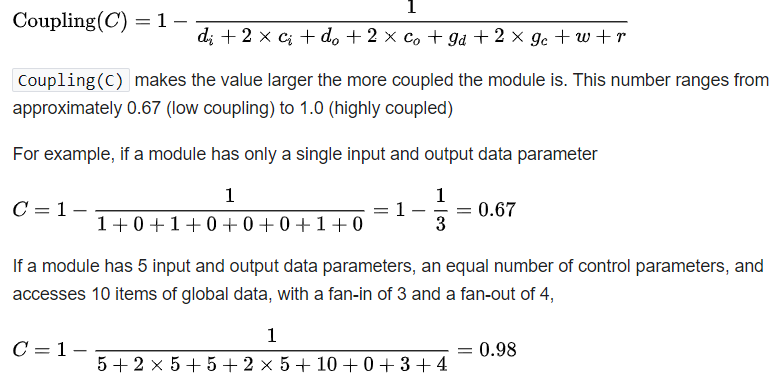
* **di**: number of input data parameters
* **ci**: number of input control parameters
* **do**: number of output data parameters
* **co**: number of output control parameters

For global coupling:

* **gd**: number of global variables used as data
* **gc**: number of global variables used as control

For environmental coupling:

* **w**: number of modules called (fan-out)
* **r**: number of modules calling the module under consideration (fan-in)



From H. Dhama, Quantitative Models of Cohesion and Coupling in Software, J. SYSTEMS SOFTWARE 1995; 29:65-74

MODULEl-Implementation #l-a single procedure implements a “selection sort.”

MODULE1

package sort1 is

type array-type is array (l..lOOO) of integer;

-- Specifications of the procedure

procedure sort1 (n : in integer; to-be-sorted: in out array-type; a--or-d: in character);

end sortl;

-- The code in the body is evaluated

package body sort1 is

procedure sort1 (n : in integer; to-be-sorted: in out array-type; a-or-d: in character) is

location, temp : integer;

begin

for start in l..n loop

location := start;

--loop to get min or max each time

for i in (start + 1X.n loop

if a-or-d = ‘d’ then

if to-be-sorted(i) > to-be-sorted(location) then

location := i;

end if;

else

if to-be-sorted(i) < to-be-sorted(location) then

location := i;

end if;

end loop;

--The exchange

temp := to-be-sortedcstart);

to-be-sortedcstart) := to-be-sorted(location);

to-be-sorted(location) := temp;

end loop;

end sortl;

end sortl;

MODULE2-Implementation #2-four procedures implement a “selection sort.”

MODULE2

package sort2 is

type array-type is array (l..lOOO) of integer;

--Specifications of the procedures

procedure sort2 (n : in integer; to-be-sorted: in out array-type; a-or-d: in character);

procedure find-max (n : in integer;to-be-sorted: in out array-type;location : in out integer);

procedure find-min (n, start : in integer;to-be-sorted: in out array-type;location : in out integer);

procedure exchange (start : in integer; to-be-sorted: in out array-type;location :in out integer);

end sort2;

package body sort2 is

--Only body code evaluated

--Procedure find-max body (‘Proc. B’ in Tables Al and A2)

procedure find-max (n, start : in integer;to-be-sorted: in array-type;location : in out integer); is

begin

location := start;

for i in start + l..n loop

if to-be-sorted(i) > to-be-sorted (location) then

location := i;

end if;

end loop;

end find-max;

--Procedure find-min body (‘Proc. c’ in Tables Al and A2)

procedure find-min (n, start : in integer;to-be-sorted: in array-type;location : in out integer) is

begin

location := start;

for i in start + l..n loop

if to-be-sorted(i) < to-be-sorted (location) then

location := i;

end if;

end loop;

end find-min;

--Procedure exchange body (‘Proc. D’ in Tables Al and A2)

procedure exchange (start : in integer; to-be-sorted: in out array-type; location : in integer) is

temp : integer;

begin

temp := to-be-sorted(start);

to-be-sorted(start):= to-be-sorted(location);

to-be-sorted(location) := temp;

end exchange;

--Procedure sort2 body CProc. A’ in Tables Al and A2)

procedure sort2 (n : in integer;to-be-sorted: in out array-type;a-or-d: in character) is

location : integer;

begin

for start in l..n loop

if a-or-d =‘d’ then

find-max(n, start, to-be-sorted, location);

else

find-min(n, start, to-be-sorted, location);

end if;

exchange (start, to-be-sorted, location);

end loop;

end sort2;

end sort2;

Calculate Coupling of the procedures sort1, sort2, find\_max, find\_min, and exchange

### Functional Cohesion.

Let

***F =*** functional cohesion of a module,

il = in data parameters,

***i2 =*** in control parameters,

ul = out data parameters,

u2 = out control parameters,

11 = number of local variables used as data,

12 = number of local variables used as control,

gl = number of global variables used as data,

g2 = number of global variables used as control,

*w =*number of modules called.

Now let

*p = il + q1i2 + ul + q2u2 + I1 + q312+ gl + q4g2 + q5w,*

where ql, q2, q3, q4, and q5 are constants, and, as a first heuristic estimate, are assumed

to be 2 in our calculations.

Then,

*F=1/p*

When a module is called, there is an implicit transfer of control to the module, that is, an implied system control parameter is being passed to the module. Therefore, when all the constituents of *p* are 0, then the minimum value of p is assumed to be 1. With this assumption, functional cohesion is an interval scale with values > 0 but < 1. Therefore,

*F =* Kl/maximum (p, 1)

where Kl is the proportionality constant. Assuming Kl = 1,

*F =* l/maximum( p, 1) (1)

Calculate Functional Cohesion (defined below) of the procedures sort1, sort2, find\_max, find\_min, and exchange

### Halstead complexity (HC)

It is defined as effort: *E* = *D* ∗ *V*, where: Difficulty: *D* =(*n*1/2)×(*N*2/*n*2); Volume: *V* =*N*× log2 *n* Program length: *N* = *N*1 + *N*2; Program vocabulary: *n* = *n*1 + *n*2 *n*1 = the number of distinct operators *n*2 = the number of distinct operands *N*1 = the total number of operators *N*2 = the total number of operands

<https://en.wikipedia.org/wiki/Halstead_complexity_measures>

For a given problem, let:

* η1 = the number of distinct operators
* η2 = the number of distinct operands
* N1 = the total number of operators
* N2 = the total number of operands

From these numbers, several measures can be calculated:

* Program vocabulary: η=η1+η2
* Program length: N=N1+N2
* Calculated estimated program length: N^=η1log2η1+η2log2η2
* Volume: V=N×log2η
* Difficulty : D=η1/2×N2/η2
* Effort: E=D×V

The difficulty measure is related to the difficulty of the program to write or understand, e.g. when doing [code review](https://en.wikipedia.org/wiki/Code_review).

The effort measure translates into actual coding time using the following relation,

* Time required to program: T=E/18 seconds

Halstead's delivered bugs (B) is an estimate for the number of errors in the implementation.

* Number of delivered bugs : B=E2/3/3000 or, more recently, B=V/3000 is accepted.

## Example

Consider the following [C](https://en.wikipedia.org/wiki/C_(programming_language)) program:

main()

{

int a, b, c, avg;

scanf("%d %d %d", &a, &b, &c);

avg = (a+b+c)/3;

printf("avg = %d", avg);

}

The distinct operators (η1) are: main, (), {}, int, scanf, &, =, +, /, printf, ,, ;

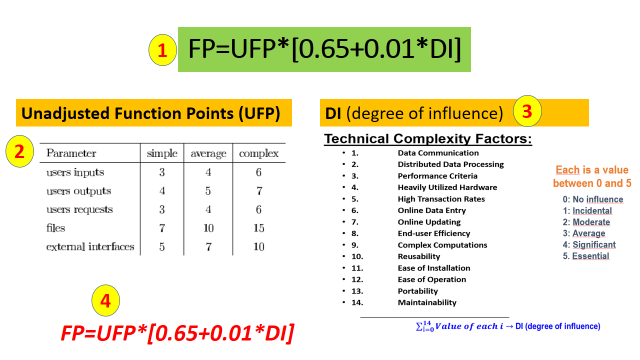
The distinct operands (η2) are: a, b, c, avg, "%d %d %d", 3, "avg = %d"

* η1=12, η2=7, η=19
* N1=27, N2=15, N=42
* Calculated Estimated Program Length: N^=12×log212+7×log27=62.67
* Volume: V=42×log219=178.4
* Difficulty: D=12/2×15/7=12.85
* Effort: E=12.85×178.4=2292.44
* Time required to program: T=2292.44/18=127.357 seconds
* Number of delivered bugs: B=2292.44233000=0.05

### Function points

It is a unit of measurement to express the amount of business functionality an information system provides to a user. Function points are a measure of the size of computer applications and the projects that build them

<https://www.geeksforgeeks.org/software-engineering-calculation-of-function-point-fp/>



**Example:**  
Given the following values, compute function point when all complexity adjustment factor (CAF) and weighting factors are average.

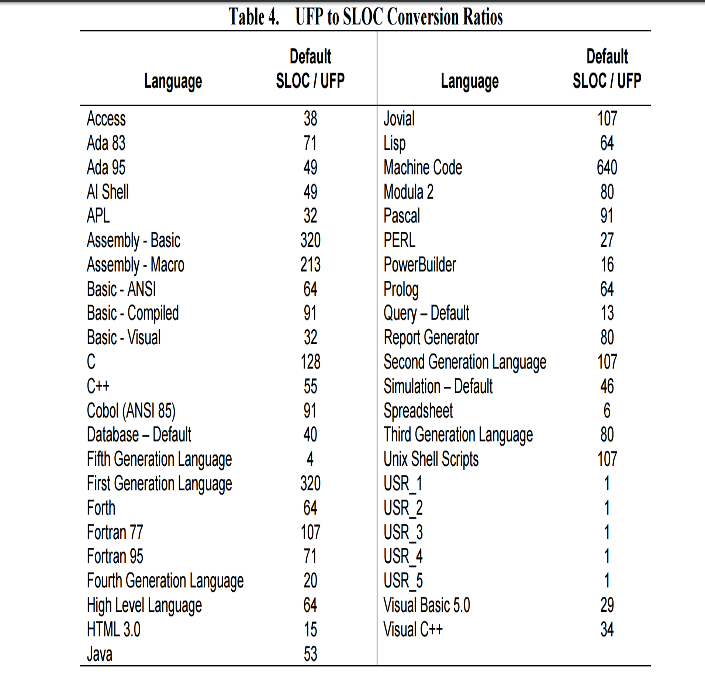
User Input = 50

User Output = 40

User Inquiries = 35

User Files = 6

External Interface = 4



Calculate KLOC if the project used VisualC++

### OO metrics

In the 1990s, large-scale adoption of object-oriented (OO) programming techniques gave rise to some OO design metrics known as Chidamber and Kemerer (CK) metric suite. Six performance metrics are found in the CK metric suite as follows:

* Weighted methods per class (WMC)—This is the number of methods implemented within a given class.
* Response for a class (RFC)—This is the number of methods that can potentially be executed in response to a message being received by an object of a given class. It is the number of methods implemented within a class plus the number of methods accessible to an object class due to inheritance.
* Lack of cohesion in methods (LCOM)—For each attribute in a given class, calculate the percentage of the methods in the class using that attributes. Next, compute the average of all those percentages, and subtract the average from 100%.
* LCOM from S.R. Chidamber, C.F. Kemerer, A metrics suite for object-oriented design, IEEE Trans on Software Engineering, 1994, vol. 30, no. 6, p. 476-493.
* 
* Coupling between object class (CBO)—This is the number of distinct non-inheritance-related classes on which a given class is coupled. A class is said to be coupled to another class if it uses methods or attributes of the other class.
* Depth of inheritance tree (DIT)—This is the length of the longest path from a given class to the root in the inheritance hierarchy.
* Number of children (NOC)—This is the number of classes that directly inherit from a given class.

## Program slicing

## <https://en.wikipedia.org/wiki/Program_slicing>

int i;

int sum = 0;

int product = 1;

int w = 7;

**for**(i = 1; i < N; ++i) {

sum = sum + i + w;

product = product \* i;

}

write(sum);

write(product);

The static executable slice for criteria (write(sum), sum) is the new program shown below.

int i;

int sum = 0;

int w = 7;

**for**(i = 1; i < N; ++i) {

sum = sum + i + w;

}

write(sum);

## The [reaching definition](https://en.wikipedia.org/wiki/Reaching_definition) analysis <https://en.wikipedia.org/wiki/Data-flow_analysis>

It calculates for each program point the set of definitions that may potentially reach this program point.

1. if b == 4 then
2. a = 5;
3. else
4. a = 3;
5. endif
6. if a < 4 then

...

The reaching definition of variable a at line 6 is the set of assignments a = 5 at line 2 and a = 3 at line 4.

## Dead code elimination

<https://en.wikipedia.org/wiki/Dead-code_elimination>

int foo(void) {

int a = 24;

int b = 25; */\* Assignment to dead variable \*/*

int c;

c = a \* 4;

**return** c;

b = 24; */\* Unreachable code \*/*

**return** 0;

}

Simple analysis of the uses of values would show that the value of b after the first assignment is not used inside foo. Furthermore, b is declared as a local variable inside foo, so its value cannot be used outside foo. Thus, the variable b is *dead* and an optimizer can reclaim its storage space and eliminate its initialization.

Furthermore, because the first return statement is executed unconditionally and there is no label after it which a "goto" could reach, no feasible execution path reaches the second assignment to b. Thus, the assignment is *unreachable* and can be removed. If the procedure had a more complex [control flow](https://en.wikipedia.org/wiki/Control_flow), such as a label after the return statement and a goto elsewhere in the procedure, then a feasible execution path might exist to the assignment to b.

## Backward analysis <https://en.wikipedia.org/wiki/Data-flow_analysis>

### Live variables

Consider the following program:

1. b = 3
2. c = 5
3. a = f(b \* c)

The set of live variables between lines 2 and 3 is {b, c} because both are used in the multiplication on line 3. But the set of live variables after line 1 is only {b}, since variable c is updated later, on line 2. The value of variable a is not used in this code.

### Backward analysis

The analysis is done in a [backwards](https://en.wikipedia.org/wiki/Data-flow_analysis#Backward_Analysis) order, and the dataflow [confluence operator](https://en.wikipedia.org/w/index.php?title=Confluence_operator&action=edit&redlink=1) is [set union](https://en.wikipedia.org/wiki/Set_union). In other words, if applying liveness analysis to a function with a particular number of logical branches within it, the analysis is performed starting from the end of the function working towards the beginning (hence "backwards"), and a variable is considered live if any of the branches moving forward within the function might potentially (hence "may") need the variable's current value. This is in contrast to a "backwards must" analysis which would instead enforce this condition on all branches moving forward.

The dataflow equations used for a given basic block *s* and exiting block *f* in live variable analysis are the following:

GEN[s]: The set of variables that are used in s before any assignment in the same basic block.

KILL[s]: The set of variables that are assigned a value in s (in many books that discuss compiler design, KILL (s) is also defined as the set of variables assigned a value in s *before any use*, but this does not change the solution of the dataflow equation):

LIVEin[s]=GEN[s]∪(LIVEout[s]−KILL[s])

LIVEout[final]=∅

LIVEout[s]=⋃p∈succ[s]LIVEin[p]

GEN[d:y←f(x1,⋯,xn)]={x1,...,xn}

KILL[d:y←f(x1,⋯,xn)]={y}

The in-state of a block is the set of variables that are live at the start of it. It initially contains all variables live (contained) in the block, before the transfer function is applied and the actual contained values are computed. The transfer function of a statement is applied by killing the variables that are written within this block (remove them from the set of live variables). The out-state of a block is the set of variables that are live at the end of the block and is computed by the union of the block's successors' in-states.

Initial code:

|  |
| --- |
| b1: a = 3;  b = 5;  d = 4;  x = 100;  if a > b then  b2: c = a + b;  d = 2;  b3: endif  c = 4;  return b \* d + c; |

Live-in and live-out sets:

// in: {}; predecessor blocks: none

b1: a = 3;

b = 5;

d = 4;

x = 100; //x is never being used later thus not in the out set {a,b,d}

if a > b then

// out: {a,b,d} //union of all (in) successors of b1 => b2: {a,b}, and // b3:{b,d}

// in: {a,b}; predecessor blocks: b1

b2: c = a + b;

d = 2;

// out: {b,d}

// in: {b,d}; predecessor blocks: b1 and b2

b3: endif

c = 4;

return b \* d + c;

// out:{}

The in-state of b3 only contains *b* and *d*, since *c* has been written. The out-state of b1 is the union of the in-states of b2 and b3. The definition of *c* in b2 can be removed, since *c* is not live immediately after the statement.

Solving the data flow equations starts with initializing all in-states and out-states to the empty set. The work list is initialized by inserting the exit point (b3) in the work list (typical for backward flow). Its computed in-state differs from the previous one, so its predecessors b1 and b2 are inserted and the process continues. The progress is summarized in the table below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **processing** | **out-state** | **old in-state** | **new in-state** | **work list** |
| b3 | {} | {} | {b,d} | (b1,b2) |
| b1 | {b,d} | {} | {} | (b2) |
| b2 | {b,d} | {} | {a,b} | (b1) |
| b1 | {a,b,d} | {} | {} | () |

Note that b1 was entered in the list before b2, which forced processing b1 twice (b1 was re-entered as predecessor of b2). Inserting b2 before b1 would have allowed earlier completion.

## Regular expressions

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_expressions/Cheatsheet>

1.6.1. ^[ \t]+|[ \t]+$ matches excess whitespace at the beginning or end of a line.

| **Characters** | **Meaning** |
| --- | --- |
| ^ | [**Input boundary beginning assertion:**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Regular_expressions/Input_boundary_assertion) Matches the beginning of input. If the [multiline](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp/multiline) (m) flag is enabled, also matches immediately after a line break character. For example, /^A/ does not match the "A" in "an A", but does match the first "A" in "An A".  **Note:** This character has a different meaning when it appears at the start of a [character class](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_expressions/Character_classes). |
| $ | [**Input boundary end assertion:**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Regular_expressions/Input_boundary_assertion) Matches the end of input. If the [multiline](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp/multiline) (m) flag is enabled, also matches immediately before a line break character. For example, /t$/ does not match the "t" in "eater", but does match it in "eat". |

| **Characters** | **Meaning** |
| --- | --- |
| [xyz] [a-c] | [**Character class:**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Regular_expressions/Character_class) Matches any one of the enclosed characters. You can specify a range of characters by using a hyphen, but if the hyphen appears as the first or last character enclosed in the square brackets, it is taken as a literal hyphen to be included in the character class as a normal character.  For example, [abcd] is the same as [a-d]. They match the "b" in "brisket", and the "c" in "chop".  For example, [abcd-] and [-abcd] match the "b" in "brisket", the "c" in "chop", and the "-" (hyphen) in "non-profit".  For example, [\w-] is the same as [A-Za-z0-9\_-]. They both match the "b" in "brisket", the "c" in "chop", and the "n" in "non-profit". |
| [^xyz] [^a-c] | [**Negated character class:**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Regular_expressions/Character_class) Matches anything that is not enclosed in the square brackets. You can specify a range of characters by using a hyphen, but if the hyphen appears as the first character after the ^ or the last character enclosed in the square brackets, it is taken as a literal hyphen to be included in the character class as a normal character. For example, [^abc] is the same as [^a-c]. They initially match "o" in "bacon" and "h" in "chop".  **Note:** The ^ character may also indicate the [beginning of input](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_expressions/Assertions). |
| \t | Matches a horizontal tab |

x+ Matches the preceding item "x" 1 or more times. Equivalent to {1,}. For example, /a+/ matches the "a" in "candy" and all the "a"'s in "caaaaaaandy".

|  |  |
| --- | --- |
| x|y | [**Disjunction:**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Regular_expressions/Disjunction) Matches either "x" or "y". Each component, separated by a pipe (|), is called an alternative. For example, /green|red/ matches "green" in "green apple" and "red" in "red apple".  **Note:** A disjunction is another way to specify "a set of choices", but it's not a character class. Disjunctions are not atoms — you need to use a [group](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_expressions/Groups_and_backreferences) to make it part of a bigger pattern. [abc] is functionally equivalent to (?:a|b|c). |

An advanced regular expression that matches any numeral is

* + 1. [+-]?(\d+(\.\d\*)?|\.\d+)([eE][+-]?\d+)?

\d [**Digit character class escape:**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Regular_expressions/Character_class_escape) Matches any digit (Arabic numeral). Equivalent to [0-9]. For example, /\d/ or /[0-9]/ matches "2" in "B2 is the suite number".

## Backus-Naur form

As an example, consider this possible BNF for a U.S. [postal address](https://en.wikipedia.org/wiki/Address_(geography)):

<**postal-address**> ::= <**name-part**> <**street-address**> <**zip-part**>

<**name-part**> ::= <**personal-part**> <**last-name**> <**opt-suffix-part**> <**EOL**> | <**personal-part**> <**name-part**>

<**personal-part**> ::= <**first-name**> | <**initial**> "."

<**street-address**> ::= <**house-num**> <**street-name**> <**opt-apt-num**> <**EOL**>

<**zip-part**> ::= <**town-name**> "," <**state-code**> <**ZIP-code**> <**EOL**>

<**opt-suffix-part**> ::= "Sr." | "Jr." | <**roman-numeral**> | ""

<**opt-apt-num**> ::= "Apt" <**apt-num**> | ""

<https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form>

This translates into English as:

* A postal address consists of a name-part, followed by a [street-address](https://en.wikipedia.org/wiki/Street_name) part, followed by a [zip-code](https://en.wikipedia.org/wiki/ZIP_Code) part.
* A name-part consists of either: a personal-part followed by a [last name](https://en.wikipedia.org/wiki/Last_name) followed by an optional [suffix](https://en.wikipedia.org/wiki/Suffix_(name)) (Jr. Sr., or dynastic number) and [end-of-line](https://en.wikipedia.org/wiki/End-of-line), or a personal part followed by a name part (this rule illustrates the use of [recursion](https://en.wikipedia.org/wiki/Recursion_(computer_science)) in BNFs, covering the case of people who use multiple first and middle names and initials).[[3]](https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form#cite_note-3)
* A personal-part consists of either a [first name](https://en.wikipedia.org/wiki/First_name) or an [initial](https://en.wikipedia.org/wiki/Initial) followed by a dot.
* A street address consists of a house number, followed by a street name, followed by an optional [apartment](https://en.wikipedia.org/wiki/Apartment) specifier, followed by an end-of-line.
* A zip-part consists of a [town](https://en.wikipedia.org/wiki/Town)-name, followed by a comma, followed by a [state code](https://en.wikipedia.org/wiki/U.S._postal_abbreviations), followed by a ZIP-code followed by an end-of-line.
* An opt-suffix-part consists of a suffix, such as "Sr.", "Jr." or a [roman-numeral](https://en.wikipedia.org/wiki/Roman_numerals), or an empty string (i.e. nothing).
* An opt-apt-num consists of a prefix "Apt" followed by an apartment number, or an empty string (i.e. nothing).

Note that many things (such as the format of a first-name, apartment number, ZIP-code, and Roman numeral) are left unspecified here. If necessary, they may be described using additional BNF rules.

## Theoretical questions

* + 1. What are the seven factors necessitating code reverse engineering?
    2. What are the six key steps in reverse engineering?
    3. What are the five phases of the phase reengineering model?
    4. What are the six layers of the source code reengineering reference model concerning reverse engineering?
    5. What are the three layers of the source code reengineering reference model concerning forward engineering?
    6. What are the five reengineering approaches?
    7. What are the three reengineering strategies?
    8. What are the four types of changes used in reengineering?

# Taxonomy

* 1. What are the three software maintenance classification types?
  2. What four clusters are considered in the evidence-based software maintenance classification?
  3. What three questions are used to decide on the type of the software maintenance activity?
  4. What are the four categories of the software maintenance concepts?
  5. What are the two approaches to the study of the software evolution?
  6. What three program classes are considered in the SPE-taxonomy of software evolution?
  7. What eight laws of evolution are defined for E-type programs?
  8. Why management of assumptions is important for E-type software projects?
  9. Why anti-regressive activity is important for E-type programs evolution?
  10. How release safety is evaluated?
  11. Why wrapper, glue, and tailoring modules are used for COTS-based systems evolution? What are they?
  12. What are the five sources of difficulties of COTS-based systems maintenance?
  13. What are the five activities for CBS systems maintenance?
  14. What are the five configuration management activities for CBS systems maintenance?

# Introduction (Concepts)

* 1. Waterfall model effort distribution
  2. Eight software evolution Lehman laws
  3. Five COTS-based system maintenance differences with respect to in-house developed systems
  4. Software maintenance life-cycle
  5. Six ISO/IEC 14764 standard maintenance activities
  6. Four software configuration management elements
  7. Relation between software evolution and reengineering
  8. Six options to manage legacy systems
  9. Traceability and dependability analysis concepts
  10. Program refactoring concept
  11. Program comprehension concept
  12. Software reuse concept