

## 1.2.2. Абстрактні моделі алгоритму. (за Черкаським М.В.)

Модель алгоритму належать до класу абстрактних, якщо правило безпосереднього перероблення не містить засобів здійснення обчислень:  $G \not\subseteq W$ , де  $G$  – конфігурація “апаратних” засобів.

Поширенім підкласом моделей *абстрактних алгоритмів* є функціональні залежності  $F$ :  $y = f(x)$ ;  $F \subset W$ . З параметричної моделі тут зафіковані лише правило безпосереднього перероблення, системи вхідних даних та кінцевих результатів. Інші параметри ігноруються:

$$M_{af} : \langle A, F \rangle,$$

де  $F$  – алгоритм у вигляді функціональної залежності.

Виконанню обчислювальних операцій без вказівки на засоби виконання відповідають також такі дві моделі: “**алгоритм – процес**” і “**алгоритм – припис**”

## 1.2.2. Абстрактні моделі алгоритму. (за Черкаським М.В.)

### “Алгоритм – процес”

Першу модель “алгоритм – процес” виконання чотирьох арифметичних операцій детально у словесній формі описав аль-Хорезмі (IX стор). Пізніше в працях Хр. Рудольфа (XVI стор.) і Г.В.Лейбніця (XVII стор.) модель аль-Хорезмі набула наступного тлумачення – “*Алгоритм означає будь-який регулярний обчислювальний процес, який за кінцеву кількість кроків розв’язує задачі визначеного класу*”. Це тлумачення близьке до наведеного у сучасній фундаментальній монографії: “*Кажучи неформально, алгоритм – це будь-яка коректно сформульована обчислювальна процедура, на вхід якої подається деяка величина або набір величин, і результатом виконання якої є вихідна величина або набір значень*”. Більш детальне тлумачення з описом властивостей алгоритму дано А Марковим:

“*a) Алгоритм – це процес послідовної побудови величин, який проходить в дискретному часі таким чином, що в початковий момент задається початкова скінчена система величин, а в кожний наступний момент система величин отримується за певним законом (програмою) із системи величин, які були в попередній момент часу (дискретність алгоритму).*

## 1.2.2. Абстрактні моделі алгоритму. (за Черкаським М.В.)

- б) Система величин, які утримуються в якийсь (не початковий) момент часу, однозначно визначається системою величин, отриманих в попередні моменти часу (детермінованість алгоритму).
- в) Закон отримання наступної системи величин із попередньої повинен бути простим і локальним (елементарність кроків алгоритму).
- г) Якщо спосіб отримання наступної величини із якої-небудь заданої величини не дає результату, то повинно бути вказано, що потрібно рахувати результатом алгоритму (спрямованість алгоритму).
- д) Початкова система величин може вибиратися із деякої потенційно нескінченної множини (масовість алгоритму)".

## 1.2.2. Абстрактні моделі алгоритму. (за Черкаським М.В.)

Це розширене тлумачення можна знайти в ряді сучасних джерел. Порівнюючи ці три тлумачення, не можна сказати, яке з них краще або вірніше, бо це лише неформальний опис моделі алгоритмічного процесу. Для такого опису не має математичних підстав порівняння. Розглянуті тлумачення відрізняються один від одного. Наприклад, процес вимірюється часом виконання, припис вимірюється кількістю інструкцій програми або блок-схем програми, але ці відміни незначні.

## 1.2.2. Абстрактні моделі алгоритму. (за Черкаським М.В.)

### “Алгоритм – припис”

Друга модель “алгоритм – припис” реалізується у вигляді блок-схем програм і програм на мові високого рівня. Прикладом тлумачення алгоритму на основі цієї моделі є: *“Алгоритм – це послідовність інструкцій для виконання деякого завдання”*.

Розробка та практичне використання моделей цього підкласу перетворилося в одну з провідних індустрій сучасності. Модель програми містить всі параметри, задекларовані в параметричній моделі. Є тільки одна особливість: правило безпосереднього перероблення задано лише набором взаємозв’язаних інструкцій – програмою. Правило безпосереднього перероблення не містить засобів виконання інструкцій. Програма відокремлена від засобів її реалізації:  $P \subset W$ . Модель визначається наступним кортежем:

$$M_{pr} : \langle A, Q, q_0, q_f, I, O, P \rangle$$

Основним об’єктом дослідження цієї моделі є часова складність. Програмний продукт – результат інтелектуальної діяльності. Але кількох характеристик складності програмної моделі алгоритму, які б оцінювали вклад людини в розробку програм, практично немає.

# Псевдо SH-модель. (за Черкаським М.В.)

Черкаський М.В. / Електротехнічні та комп'ютерні системи № 04(80), 2011

Ця абстрактна модель програми розглядає сукупність інструкцій  $\{B\}$  як множину об'єктів, зв'язаних між собою множиною з'єднань  $\{U\}$ . Правило безпосереднього перероблення псевдо SH-моделі задано цими двома множинами, що не перетинаються. Параметричний опис псевдо SH-моделі такий:

$$M_{ps} : \langle A, Q, q_0, q_f, I, O, G_s \rangle,$$

де  $G_s = (B, U)$  – граф, який відображає структуру блок-схеми програми,  $B$  – множина об'єктів,  $U$  – множина з'єднань між об'єктами.

У визначенні немає параметра  $P$  – програми, тому що модель  $M_{ps}$  сама є програмою. В перелік характеристик складності псевдо SH-моделі, крім часової, входять також об'єктна і структурна складність. Об'єкт це інструкція програми. Об'єктна складність дорівнює потужності множини об'єктів. Структурна складність оцінює ступінь нерегулярності зв'язків блок-схеми програми. Вона визначається логарифмічною мірою степені нерівномірності матриці інциденцій моделі. Структурна складність є інформаційною характеристикою. Одиницею структурної і об'єктної складності є біт.

## 1.2.3. Формальні алгоритмічні системи(ФАС).

### *ФАС з уявними засобами виконання інструкцій та SH-модель алгоритму*

Клас алгоритмів, в яких правило безпосереднього перероблення задано програмою і засобами виконання інструкцій, отримав назву “Формальні алгоритмічні системи” (ФАС). Моделі ФАС можна поділити на два підкласи.

До першого належать моделі ФАС з уявними засобами виконання інструкцій. Ці засоби тільки пояснюють, як може відбуватися обчислювальний процес. Їх наявність та складність не враховується у формальному кількісному описі моделі. Також припускається, що операції алгоритмічного процесу може взагалі здійснюватися людиною(без докладання інтелектуальних зусиль). Прикладам таких ФАС є машина Тюрінга, нормальні алгоритми Маркова, система Колмогорова та ін..

Тлумаченням алгоритму, що випливає з аналізу цих моделей є: “*Алгоритм – точний припис, який задає обчислювальний процес (що називається в цьому випадку алгоритмічним), що починається з довільного початкового даного (з деякої сукупності можливих для даного алгоритму початкових даних) і спрямований на отримання результату, який повністю визначається цим початковим даним*”. Тут точний припис і алгоритмічний процес об’єднані в одному тлумаченні.

## 1.2.3. Формальні алгоритмічні системи(ФАС).

*ФАС з уявними засобами виконання інструкцій та SH-модель алгоритму*

Другий клас формальних систем складається з моделей **комп'ютерних алгоритмів**. У цих моделях апаратні засоби задекларовані безпосередньо у її визначенні. Прикладом моделі другого класу є SH-модель алгоритму (апаратно-програмна модель алгоритму, SH – Software/Hardware). У визначенні цієї моделі у явній формі зафіксована наявність апаратних і програмних засобів виконання інструкцій. Правило безпосереднього перероблення задано апаратно програмними засобами:

$$W = (G, P),$$

де  $G$  – конфігурація апаратних засобів,

$$G = (X, U),$$

де  $X$  – множина елементарних перетворювачів,  $U$  – множина з'єднань елементарних перетворювачів;  $P$  – програма.

## 1.2.3. Формальні алгоритмічні системи(ФАС).

*ФАС з уявними засобами виконання інструкцій та SH-модель алгоритму*

Модель апаратно-програмної ФАС має назву “SH-модель алгоритму” (S – Software, H – Hardware). Параметричний опис SH моделі:

$$M_{SH} : \langle A, Q, a_0 q_f, q_0, I, O, G, P \rangle$$

Модель комп’ютерного алгоритму дозволила формально визначити властивість “елементарність”, сформулювати властивість “ієрархічність”, створити модель універсального обчислювача. Відображення змісту моделі комп’ютерного алгоритму знайшло в неформальному тлумаченні: “*Алгоритм - це фіксована для розв’язання деякого класу задач конфігурація апаратно-програмних засобів перетворення, передавання і зберігання даних, який задає обчислювальний процес (що називається в цьому випадку алгоритмічним), який починається з будь-яких початкових даних (з деякої потенційно нескінченної сукупності можливих для даного алгоритму початкових даних) і скерований на отримання результату, повністю визначеного цими початковими даними*”.



# Формальні граматики, мови та ієрархія Хомського

Говорять, що скінчений автомат  $M = (S, I, f, s_0, F)$  **допускає** (приймає) ланцюжок  $\alpha$ , якщо він переводить початковий стан  $s_0$  в заключний стан; це означає, що стан  $f(s_0, \alpha)$  – елемент множини  $F$ .

Мова  $L(M)$ , яку **розвізнає автомат**  $M$ , – це множина всіх ланцюжків, які допускає автомат  $M$ . Два автомати називають **еквівалентними**, якщо вони розвінюють одну й ту саму мову.

Якщо визначити **слово** як стрічку символів, що створена через конкатенацію (з'єднання), а **мову** як множину слів (може бути нескінченною множиною), то говорять, що мова  $L$  читається (приймається) автомatem  $M$ , якщо по закінченню обробки він переходить в один з кінцевих станів  $F$ .

Формальна граматика або просто граматика в теорії формальних мов — спосіб опису формальної мови, тобто виділення деякої підмножини з множини всіх слів деякого скінченного алфавіту. Розрізняють породжувальні і аналітичні граматики — перші ставлять правила, за допомогою яких можна побудувати будь-яке слово мови, а другі дозволяють по даному слову визначити, входить воно в мову чи ні.

**Алфавіт** — скінчена множина символів.  $\varepsilon$  — порожній ланцюжок. Нехай  $T$  — алфавіт, тоді:

$T^+$  — множина усіх можливих послідовностей, що складені з елементів цього алфавіту крім порожньої послідовності  $\varepsilon$ .

$T^*$  — множина усіх можливих послідовностей, що складені з елементів цього алфавіту, будь-якої довжини.

$T_k$  — множина усіх можливих послідовностей, що складені з елементів цього алфавіту, довжини не більше  $k$ .

**Мова** в алфавіті  $T$  це множина ланцюжків скінченної довжини в цьому алфавіті. Зрозуміло, що кожна мова в алфавіті  $T$  є підмножиною множини  $T^*$ .

**Формальна граматика** - це четвірка  $G = \{N, T, P, S\}$ . Де:

- **T** — алфавіт термінальних символів, терміналів (від англ. terminate - завершитись). Термінальні символи є алфавітом мови.
- **N** — алфавіт нетермінальних символів, нетерміналів.  $T \cap N = \emptyset$ ; Нетермінали не входять в алфавіт мови.
- **S** — аксіома, спеціально виділений нетермінальний символ з якого починається опис граматики.

$S \in N$

- **P** — скінчена підмножина множини  $(T \cup N)^+ \times (T \cup N)^*$ . Інколи визначають так:  $\alpha \in (T \cup N)^* \times N \times (T \cup N)^*$ ,  $\beta \in (T \cup N)^*$ .

Елемент  $(\alpha, \beta)$  з множини P називається правилом виводу і записується у вигляді  $\alpha \rightarrow \beta$ .

Таким чином, ліва частина правила не може бути порожньою. Правила з однаковою лівою частиною записують:

$\alpha \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$ ,  $\beta_i, i = 1, 2, \dots, n$  - називаються альтернативами правила виводу з ланцюжка  $\alpha$ .

**Формальнамова** породжена граматикою G - це множина термінальних рядків, що виводяться з аксіоми, тобто множина усіх правильних рядків.

$$L(G) = \{\alpha \in T^* \mid S \Rightarrow^* \alpha\}$$

З іншого боку, множина термінальних рядків, таких, що вони можуть бути виведені в граматиці G, називається мовою, що може бути розпізнана в даній граматиці, або допускається даною граматикою.

- Граматики  $G_1$  та  $G_2$  називаються **еквівалентними**, якщо  $L(G_1) = L(G_2)$ .
- Граматики  $G_1$  та  $G_2$  називаються **майже еквівалентними**, якщо  $L(G_1) \cup \{\varepsilon\} = L(G_2) \cup \{\varepsilon\}$ , тобто мови, ними породжувані, відрізняються не більш ніж на  $\varepsilon$ .

Граматика може бути **породжуючою** та **розвізнавальною**, це залежить від "напрямку" застосування правил. Для породжуючих граматик виведення починається з аксіоми і закінчується термінальним рядком (рядком, що складається тільки з терміналів). А розвізнавальні аналізують вхідний термінальний рядок на правильність, чи можна такий рядок вивести в цій граматиці.

Ієрархія Чомскі, або Ієрархія Чомскі-Шутценберг'ера (названа на честь мовознавця Ноама Чомскі та математика Марселя Шутценберг'ера) — ієрархія формальних граматик, які породжують формальні мови. Вперше описана Ноамом Чомскі в 1956 році. Чотири описані Чомскі типи граматик виходять від базової, необмеженої граматики (граматика типу 0), на яку послідовно накладають обмеження на правила продукції.

В залежності від типу найпростішої граматики, яка може згенерувати задану формальну мову, формальні мови ділять на відповідні категорії від типу 0 до типу 3.

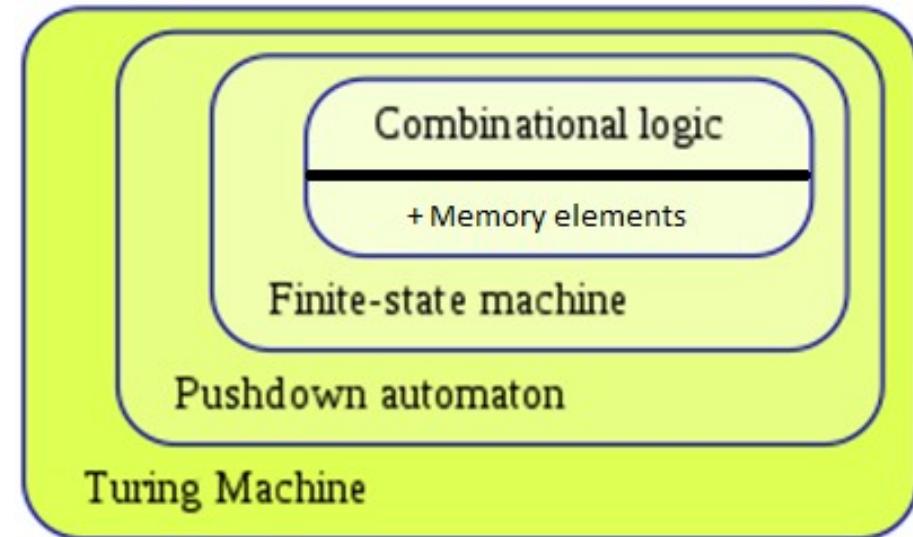
Граматика	Правила	Мови	Автомати	Скорочення
<b>Тип-0</b> <u>Довільна</u> <u>формальна</u> граматика	$\alpha \rightarrow \beta$ $\alpha \in V^* N V^*, \beta \in V^*$	<u>рекурсивно</u> <u>зліченна</u>	<u>Машина Тюринга</u>	KSV*
<b>Тип-1</b> <u>Контекстно-</u> <u>залежна</u> граматика	$\alpha A \beta \rightarrow \alpha'y'\beta$ $A \in N, \alpha, \beta \in V^*, y \in V^+$ S $\rightarrow \epsilon$ дозволене, коли серед правил P відсутнє $\alpha \rightarrow \beta S y$ .	<u>контекстно-</u> <u>залежна</u>	<u>Лінійний обмежений автомат</u>	КЗ
<b>Тип-2</b> <u>Контекстно-</u> <u>вільна</u> граматика	$A \rightarrow y$ $A \in N, y \in V^*$	<u>контекстно-</u> <u>вільна</u>	недетермінований автомат з магазинною пам'ятю	КВ
<b>Тип-3</b> <u>регулярна</u> граматика	$S \rightarrow \epsilon$ $A \rightarrow aB$ (праволінійна) або $A \rightarrow Ba$ (ліволінійна) $A \rightarrow a$ $A \rightarrow \epsilon$ $A, B \in N, a \in \Sigma$	<u>регулярна</u>	Скінчений автомат (як детермінований, так і недетермінований)	А

recursively enumerable

context-sensitive

context-free

regular



# Регулярні мови та вирази

- Окрім автоматів і граматик для опису регулярних мов зручно використовувати регулярні вирази.
- **Регулярний вираз є послідовністю, що описує множину рядків. Ці послідовності використовують для точного описання множини без перелічення всіх її елементів.** Наприклад, множина, що складається із слів «грати» та «г'рати» може бути описана регулярним виразом «[гг']рати». В більшості формалізмів, якщо існує регулярний вираз, що описує задану множину, тоді існує нескінчена кількість варіантів, які описують цю множину

*Регулярний вираз* над алфавітом  $\Sigma$  визначається рекурсивно так:

0 є регулярним виразом;

1 є регулярним виразом;

$a$  є регулярним виразом, якщо  $a \in \Sigma$ ;

$(e + f)$ ,

$(e \cdot f)$

$e^*$  є регулярними виразами, якщо  $e$  і  $f$  – регулярні вирази.

Для зменшення кількості дужок у виразах введено такий пріоритет раніше введених операцій. Визначено, що операція  $*$  має вищий пріоритет, ніж множення, а множення має вищий пріоритет, ніж додавання. Замість  $e \cdot f$  часто пишуть просто  $ef$ .

### Приклад

Нехай  $\Sigma = \{a, b\}$ . Тоді  $((a \cdot b)^* \cdot (1 + a))$  є регулярним виразом над алфавітом  $\Sigma$ .

"0" відповідає нейтральному елементу для операції об'єднання (тобто порожній множині), а "1" відповідає нейтральному елементу для операції конкатенації (тобто порожньому рядку).

Регулярні мови замкнені відносно операцій об'єднання, перетину, конкатенації, доповнення та зірочки Кліні.

Існують й інші операції, відносно яких замкнені регулярні мови. Так, наприклад, гомоморфне відображення регулярної мови також є регулярною мовою. Таким чином, регулярні мови замкнені відносно гомоморфізму.

Кожний регулярний вираз  $e$  над алфавітом  $\Sigma$  задає мову над алфавітом  $\Sigma$  (позначається  $L(e)$  де  $e$  – регулярний вираз), яка визначається рекурсивно так, якщо  $e, f$  – регулярні вирази:

$$L(0) = \emptyset;$$

$$L(1) = \varepsilon;$$

$$L(a) = \{a\}, \text{ якщо } a \in \Sigma;$$

$$L(e + f) = L(e) \cup L(f);$$

$$L(e \cdot f) = L(e) \cdot L(f);$$

$$L(e^*) = L(e)^*.$$

Замість  $L(e)$  часто пишуть просто  $e$ .

# Метасимволи регулярних виразів

- **.** (крапка) – один довільний символ (крім символу переходу на новий рядок)
- **\t** – табуляція
- **\n** – новий рядок
- **\\** – власне \ (backslash)
- **\s** – один довільний пробільний символ (пробіл, табуляція, новий рядок)
- **\S** – один довільний символ, який не входить у перелічені для \ s
- **\d** – одна довільна цифра (digit)
- **\D** – один довільний символ, який не може бути цифрою

# Метасимволи регулярних виразів

- **\w** – одна довільна літера або цифра або знак підкреслювання (word character), те саме що [A-Za-z0-9\_]
- **\W** – один довільний символ, який не входить у перелічені для \w
- **-** – позначає діапазон(у класі символів) або власне цей символ, якщо це перший символ у класі (**[-abc]**)
- **\$** – кінець рядка
- **^** – початок рядка або заперечення, якщо це перший символ у класі символів (**[^abc]**)
- **^\$** – пустий рядок
- **|** - логічне АБО (використовується у групі символів)

# Метасимволи регулярних виразів

- \< – початок слова
- \> – кінець слова
- \b – межа слова (початок або кінець) або символ backspace, якщо він знаходиться у класі символів
- \B – позиція, що не є межею слова
- \. – власне крапка
- \\$ – власне символ \$
- \[, \] – власне квадратні дужки
- \(, \) – власне круглі дужки
- \{, \} – власне фігурні дужки

# Метасимволи регулярних виразів

*Групування та повторення:*

- \* – нуль або більше разів повторений попередній символ (або група символів)
- + – один або більше разів повторений попередній символ (або група символів)
- ? – нуль або один раз повторений попередній символ (або група символів)\
- () – групують символи (всі, що присутні у дужках, можливе застосування логічного АБО (символ |))
- [] – визначають клас (або множину) символів – неупорядковану групу символів, з якої для відповідного регулярного виразу обирається один довільний
- {n} – рівно n разів повторений попередній символ (або група символів)
- {n, m} – попередній символ (або група символів) повторений від n до m разів
- {n, } – n або більше разів повторений попередній символ (або група символів)

# Регулярні вирази

## Приклади

- **[A-Za-z]** – один довільний символ латинської абетки, незалежно від регістру (тут не можна писати [A-z], бо між літерами у великому та малому регістрах знаходяться інші символи)
- **[0-9]** або **(0|1|2|3|4|5|6|7|8|9)** або **\d** – одна довільна цифра
- **\(\d\{3\}\) \d\{3\}-\d\{4\}** – номер телефону у форматі (044) 123-4567
- **#-[0-9a-fA-F]\{6\}** – шістнадцятковий код кольору (наприклад, #12CCAA)
- **([0-9][1-9][0-9]|1[1-9][0-9]|2[0-4][0-9]|25[0-5])** – довільне число з діапазону 0-255 (цей вираз можна скоротити)

№	Формат лексем
1	Up-Low2, первый символ Up
2	Low-Up2, первый символ Low
3	Up2
4	Low2
5	Up-Low4, первый символ Up
6	Low-Up4, первый символ Low
7	Up4
8	Low4
9	Up-Low6, первый символ Up
10	Low-Up6, первый символ Low
11	Up6
12	Low6
13	Up-Low8, первый символ Up
14	Low-Up8, первый символ Low
15	Up8
16	Low8
17	Up-Low2, первый символ _____
18	Low-Up2, первый символ _____
19	Up4, первый символ _____
20	Low4, первый символ _____
21	Up-Low4, первый символ _____
22	Low-Up4, первый символ _____
23	Up6, первый символ _____
24	Low6, первый символ _____
25	Up-Low6, первый символ _____
26	Low-Up6, первый символ _____
27	Up8, первый символ _____
28	Low8, первый символ _____
29	Up-Low8, первый символ _____
30	Low-Up8, первый символ _____

# Приклад коду

Лістинг

```
#include <fstream>
#include <iostream>
#include <algorithm>
#include <iterator>
#include <regex>

int main()
{
    std::string text =
        "Sir, in my heart there was a kind of fighting "
        "That would not let me sleep. Methought I lay "
        "Worse than the mutines in the bilboes. Rashly— "
        "And prais'd be rashness for it—let us know "
        "Our indiscretion sometimes serves us well ... "
        ; // – Hamlet, Act 5, Scene 2, 4–8

    std::regex token_re("\b[s][a-z]+\b", std::regex::icase);
    std::copy(std::sregex_token_iterator(text.begin(), text.end(), token_re, 0),
              std::sregex_token_iterator(),
              std::ostream_iterator<std::string>(std::cout, "\n"));

    return 0;
}
```

# Приклад коду

Лістинг

```
#include <fstream>
#include <iostream>
#include <algorithm>
#include <iterator>
#include <regex>

int main()
{
    std::string text =
        "Sir, in my heart there was a kind of fighting "
        "That would not let me sleep. Methought I lay "
        "Worse than the mutines in the bilboes. Rashly— "
        "And prais'd be rashness for it—let us know "
        "Our indiscretion sometimes serves us well ... "
        ; // – Hamlet, Act 5, Scene 2, 4–8

    std::regex token_re("\b[s][a-z]+\b", std::regex::icase);
    std::copy(std::sregex_token_iterator(text.begin(), text.end(), token_re, 0),
              std::sregex_token_iterator(),
              std::ostream_iterator<std::string>(std::cout, "\n"));

    return 0;
}
```

## Low3 – слово складається з трьох малих літер

```
#include <fstream>
#include <iostream>
#include <algorithm>
#include <iterator>
#include <regex>

int main()
{
    std::string text =
        "Sir, in my heart there was a kind of fighting "
        "That would not let me sleep. Methought I lay "
        "Worse than the mutines in the bilboes. Rashly– "
        "And prais'd be rashness for it–let us know "
        "Our indiscretion sometimes serves us well ... "
    ; // – Hamlet, Act 5, Scene 2, 4–8

//std::regex token_re("\b[a-z]{3}\b");
std::regex token_re("\b[a-z][a-z][a-z]\b");
std::copy(std::sregex_token_iterator(text.begin(), text.end(), token_re, 0),
    std::sregex_token_iterator(),
    std::ostream_iterator<std::string>(std::cout, "\n"));

return 0;
}
```

Low3 – слово складається з трьох малих літер

(a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t+u+v+w+x+y+z)(a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t+u+v+w+x+y+z)  
(a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t+u+v+w+x+y+z)

\b[a-z]{3}\b

\b[a-z][a-z][a-z]\b

```
#include <iostream>
#include <regex>

/* Example for regular expression:
(a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t+u+v+w+x+y+z) (a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t+u+v+w+x+y+z)
*/
std::regex token_re("[a-z]{3}");
//std::regex token_re("[a-z][a-z][a-z]");
//std::regex token_re("[a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t+u+v+w+x+y+z]");
//std::regex token_re("[a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z]");
int main(){
    std::string text = "qtv";
    std::cout << std::boolalpha << std::regex_match(text, token_re);

    return 0;
}
```



# КВ-МОВИ та нотації БНФ.

Всі правила для КВ-граматик вигляд:

$A \rightarrow \beta$ , де  $A \in N$ ,  $\beta \in V^*$ ,  $V = T \cup N$ .

$T$ : множина термінальних символів

$N$ : множина нетермінальних символів

$V = T \cup N$ : словник граматики (множина всіх термінальних та нетермінальних символів)

В усіх правилах зліва стоїть тільки один нетермінал, тобто на використання правила для даного нетерміналу не впливають символи, що оточують його. Ці символи називають **контекстом**.

Основна властивість КВ-мов: клас КВ-мов замкнутий щодо операції підстановки. Це означає, що якщо в кожний ланцюжок символів КВ-мови замість деякого символу підставити ланцюжок символів з іншої КВ-мови, то отриманий ланцюжок також належатиме КВ-мові.

Клас КС-мов замкнутий щодо операцій об'єднання, конкатенації, ітерації, зміни імен символів.

Зauważення:

Клас КС-мов не замкнений щодо операції перетину та операції доповнення.

# Нотація Бекуса-Наура

- Нотація Бекуса—Наура (англ. Backus-Naur form, BNF) — це спосіб запису правил контекстно-вільної граматики, тобто форма опису формальної мови.
- БНФ визначає скінченну кількість символів (нетерміналів). Крім того, вона визначає правила заміни символу на якусь послідовність букв (терміналів) і символів. Процес отримання ланцюжка букв можна визначити поетапно: спочатку є один символ (символи зазвичай знаходяться у кутових дужках, а їх назва не несе жодної інформації). Потім цей символ замінюється на деяку послідовність букв і символів, відповідно до одного з правил. Потім процес повторюється (на кожному кроці один із символів замінюється на послідовність, згідно з правилом). Зрештою , виходить ланцюжок , що складається з букв і не містить символів. Це означає , що отриманий ланцюжок може бути виведений з початкового символу.

# Нотація Бекуса-Наура

Нотація БНФ є набором «продукцій», кожна з яких відповідає зразку:

**<символ> ::= <вираз, що містить символи>**

**<вираз, що містить символи>** це послідовність символів або послідовності символів, розділених вертикальною рискою |, що повністю перелічують можливий вибір символ з лівої частини формули.

Наступні чотири символи є символами мета-мови, вони не визначені у мові, котру описують:

- < — лівий обмежувач виразу
- > — правий обмежувач виразу
- ::= — визначене як
- | — або

Інші символи належать до «абетки» описаної мови.

# Нотація Бекуса-Наура

*Приклад 1.* БНФ для поштової адреси:

```
<поштова-адреса> ::= <поштове-відділення> <вулична-адреса> <особа>
<поштове-відділення> ::= <індекс> ", " <місце> <EOL>
<місце> ::= <село> | <місто>
<вулична-адреса> ::= <вулиця> "," <будинок> <EOL>
<особа> ::= <прізвище> <ім'я> <EOL> | <прізвище> <ім'я> <по батькові> <EOL>
```

*Приклад 2.* Один зі способів означення натуральних чисел за допомогою БНФ:

```
<нуль> ::= 0
<ненульова цифра> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<цифра> ::= <нуль> | <ненульова цифра>
<послідовність цифр> ::= <нуль> | <ненульова цифра> | <цифра> <послідовність цифр>
<натуральне число> ::= <цифра> | <ненульова цифра> <послідовність цифр>
```

# Розширенна нотація Бекуса-Наура

- Розширенна форма Бéкуса — Нáура (англ. extended Backus–Naur form, EBNF) була розроблена Нíклавсом Вíртом, яка сьогодні існує в багатьох варіантах, перед усім — ISO-14977. РБНФ відрізняється від БНФ більш «ємкими» конструкціями, що дозволяють при тій же виразності спростити і скоротити в обсязі опис.

# Розширенна нотація Бекуса-Наура

При використанні розширеної форми Бекуса-Наура (EBNF):

- не термінальні символи записуються як окремі слова
- термінальні символи записуються в лапках
- вертикальна риска |, як і в БНФ, використовується для визначення альтернатив
- круглі дужки використовуються для групування
- квадратні дужки використовуються для визначення можливого входження символу або групи символів
- фігурні дужки використовуються для визначення можливого повторення символу або групи символів
- символ рівності використовується замість символу :: =
- конкатенація позначається комою
- символ крапки(крапки з комою) використовується для позначення кінця правила
- коментарі записуються між символами (\* ... \*)

# <http://www.cs.man.ac.uk/~pjj/bnf/ebnf.html>

Style	Name and link	Rule	Items		Concat (default space)	Choice	Optional	Repetition					Grouping	Other	Comment
			Terminal	Non Terminal				0 or more	1 or more	n	n to m	Other			
BNF	<a href="#">BNF (Algol 60)</a>	<code>&lt;name&gt; ::= ...</code>	<code>&lt;...&gt;</code>				<a href="#">note1</a>	<a href="#">note2</a>	<a href="#">note3</a>						<a href="#">note5</a>
Intermediate	<a href="#">ANSIC</a>	<code>name: ...</code>	<b>bold</b>	<i>italics</i>		indented line one of	<code>...opt</code>	<a href="#">note2</a>	<a href="#">note3</a>						<a href="#">note5</a>
	BNF-like description of <a href="#">URLs</a>	<code>name ...</code>		<code>...</code>			<code>[...]</code>	<a href="#">note2</a>	<a href="#">note3</a>						<a href="#">note5</a>
Wirth	<a href="#">Wirth</a>	<code>name=... . . .</code>	"..."				<code>[...]</code>	{...}	<a href="#">note4</a>					(...)	
	<a href="#">SAIF and Bungisoft</a>	<code>&lt;name&gt; ::= ... . . .</code>	"..."	<code>&lt;...&gt;</code>			<code>[...]</code>	{...}	<a href="#">note4</a>					(...)	<a href="#">note5</a>
	<a href="#">EBNF from Compiler Basics by J.A.Farrell</a>	<code>name==... . . .</code>	'.'			or [.....]	<code>[...]</code>	{...}*{...}	{...}						<a href="#">note5</a>
	<a href="#">Pascal EBNF Definition</a>	<code>name ...</code>	"..."	<b>bold</b>	<code>&lt;...&gt;</code>		<code>[...]</code>	{...}	<a href="#">note4</a>					(...)	<a href="#">note5</a>
	<a href="#">ISO Extended Pascal</a>	<code>name=... . . . name&gt;... . . .</code>	'.'				<code>[...]</code>	{...}	<a href="#">note4</a>					(...)	(*...*)
	<a href="#">ISO EBNF</a>	<code>name=... ; . . .</code>	"..."		:		<code>[...]</code>	{...}{...}-n*	{...}					(...)	?...?(*...*)
ABNF	<a href="#">EBNF from RFC822 (superceded by ABNF)</a>	<code>name=... integer . . .</code>	integer	"..."	<code>&lt;...&gt;</code>	/	<code>[...]</code> *1...	*...1*...n*n..n*m..n#m...	(...)					;	
	<a href="#">ABNF (RFC2234)</a>	<code>name=... integer . . .</code>	integer	"..."	<code>&lt;...&gt;</code>	/	<code>[...]</code> *1...	*...1*...n*n..n*m...	(...)					;	
XBNF	<a href="#">XBNF</a>	<code>name::=... . . .</code>	"..."				O(...)	#(...)	N(...)			L(...,...)	(...)	...&... ...~...	any text not in a rule
Regular expression	<a href="#">ISO Minimal BASIC</a>	<code>name=... ITALICS . . .</code>	ITALICS	<i>italics</i>		/	...?	...*	<a href="#">note4</a>						<a href="#">note5</a>
	<a href="#">EBNF for XML</a>	<code>name::=... . . .</code>	"..."				...?	...*	...+				(...)	?...[wfc:...] /*...*/ [vc:...]	

# Розширенна нотація Бекуса-Наура

*Приклад.* Один зі способів означення цілих чисел за допомогою РБНФ:

---

Integer = Sign UnsignedInteger.

UnsignedInteger = digit, {digit}.

Sign = [ "+" | "-" ].

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".

---

## Приклад коду

Наведена програма (*лістинг 2*) перевіряє коректність виразу, синтаксис якого заданий РБНФ (*лістинг 1*):

Лістинг 1

```
expression = term_a , { ("+" | "-"), <term_a> }*.
term_a     = term_m , { ("*" | "/"), <term_m> }*.
term_m     = value | "+" term_m | "-" term_m | "(", expression , ")".
```

Лістинг 2

```
#include <iostream>
#include <string>
#include <boost/spirit/include/qi.hpp>

namespace qi = boost::spirit::qi;

template <typename Iterator>
struct calculator_simple : qi::grammar<Iterator>{
    calculator_simple() : calculator_simple::base_type(expression){
        expression = term   >> *( '+' >> term   | '-' >> term );
        term      = factor >> *( '*' >> factor | '/' >> factor );

        factor    =
            qi::uint_
            | '(' >> expression >> ')'
            | '+'
            | '-';

    }

    qi::rule<Iterator> expression, term, factor;
};
```

```
int main(){
    std::cout << "Welcome to the expression parser!\n\n";
    std::cout << "Type an expression or [q or Q] to quit\n\n";

    typedef std::string      str_t;
    typedef str_t::iterator str_t_it;

    str_t expression;

    calculator_simple<str_t_it> calc;

    while(true){
        std::getline(std::cin, expression);
        if(expression == "q" || expression == "Q") break;
        str_t_it begin = expression.begin(), end = expression.end();

        bool success = qi::parse(begin, end, calc);

        std::cout << "-----\n";
        if(success && begin == end)
            std::cout << "Parsing succeeded\n";
        else
            std::cout << "Parsing failed\nstopped at: \""
            << str_t(begin, end) << "\"\n";
        std::cout << "-----\n";
    }
}
```

Low3 – слово складається з трьох малих літер

**low3 = low\_case\_letter , low\_case\_letter , low\_case\_letter.**

**low\_case\_letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm'**  
**| 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'.**

```
#include <iostream>
#include <string>
#include <boost/spirit/include/qi.hpp>

#define USE_SIMPLE_CODE

namespace qi = boost::spirit::qi;

/* Example for EBNF:
low3 = low_case_letter , low_case_letter , low_case_letter.
low_case_letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'.
*/
template <typename Iterator>
struct grammar_simple : qi::grammar<Iterator>{
    grammar_simple() : grammar_simple::base_type(low3){
        low3 = low_case_letter >> low_case_letter >> low_case_letter;
        low_case_letter = a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z;
        a = 'a';
        b = 'b';
        c = 'c';
        d = 'd';
        e = 'e';
        f = 'f';
        g = 'g';
        h = 'h';
        i = 'i';
        j = 'j';
        k = 'k';
        l = 'l';
        m = 'm';
        n = 'n';
        o = 'o';
        p = 'p';
        q = 'q';
        r = 'r';
        s = 's';
        t = 't';
        u = 'u';
        v = 'v';
        w = 'w';
        x = 'x';
        y = 'y';
        z = 'z';
    }

    qi::rule<Iterator> low3, low_case_letter, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z;
};

};
```



# Абстрактний автомат

**Абстрактний автомат** — теоретична модель апаратної або програмної обчислювальної системи, побудована на основі теорії автоматів.

Абстрактний автомат складається з:

- вхідна стрічка;
- пристрій керування;
- допоміжна/робоча пам'ять.

**Вхідна стрічка.** Її можна розглядати як лінійну послідовність кліток, або комірок, причому кожна комірка точно містить один вхідний символ з деякого скінченного вхідного алфавіту. За допомогою вхідної стрічки зображується інформація, що поступає на вхід автомата.

**Вхідна голівка.** У кожний момент вона читає одну вхідну комірку. За один крок роботи автомата вхідна голівка може зміститися на одну комірку вліво, вправо або залишитися нерухомою. Автомат, що не переміщує свою голівку називається одностороннім.

**Пам'ять автомата.** Пам'ять автомата - структура, в якій записуються, зберігаються і читаються додаткові дані, що використовуються автомatem при роботі. Для кожного виду автоматів строго визначено тип пам'яті. Автомат може не мати назву, - тип пам'яті часто визначає назву автомата.

# Абстрактний автомат

Робота автомата складається з послідовності тактів. Кожен такт складається з таких дій:

1. Читається поточний вхідний символ.
2. За допомогою функції доступу досліджується пам'ять, і до неї заноситься деяка інформація.
3. Змінюється стан пристрою керування.
4. Записується вихідна інформація у комірки вхідної стрічки.
5. Вхідна голівка зміщується на одну комірку вліво, вправо або зберігається у початковому стані.

Поточний вхідний символ та інформація, що витягнута з пам'яті, разом з поточним станом пристрою керування визначають, яким повинен бути цей такт. Таким чином, на кожному такті визначається поточна конфігурація автомата, до якої входять:

- поточний стан пристрою керування;
- поточний зміст вхідної стрічки;
- поточний зміст робочої пам'яті.

Конфігурація автомата змінюється на кожному такті під впливом пристрою керування.

# Абстрактний автомат

**Пристрій керування.** Складається зі скінченної множини станів  $S = \{s_0\dots s_n\}$  і відображень, які залежно від попередньої конфігурації дозволяють визначити нову конфігурацію автомата, напрямок переміщення входної голівки (якщо вона не одностороння), інформацію на друку (якщо в автоматі передбачено функцію друку), інформацією, що заносять у пам'ять і зчитують звідти (якщо автомат має робочу пам'ять).

Можуть бути два типи пристрою керування:

- Недетермінований — дляожної конфігурації існує скінчена множина можливих конфігурацій(більше однієї), так що в будь-яку з них автомат може перейти на наступному кроці.
- Детермінований — дляожної конфігурації існує не більше однієї можливої наступної конфігурації.

# Скінчений автомат

**Скінчений автомат**(англ. finite-state machine, машина зі скінченою кількістю станів) — особливий різновид автомата — абстракції, що використовується для опису шляху зміни стану об'єкта в залежності від поточного стану та інформації отриманої ззовні. Його особливістю є скінченість множини станів автомата. Поняття скінченого автомата було запропоновано як математичну модель технічних пристрій дискретної дії, оскільки будь-який такий пристрій (в силу скінченості своїх розмірів) може мати тільки скінченну кількість станів.

Скінчені автомати можуть розв'язувати велику кількість задач, серед яких автоматизація проектування електронних пристрій, проектування комунікаційних протоколів, синтаксичний аналіз та інші інженерні застосування. В біології і дослідженнях штучного інтелекту, автомати або їх ієрархії іноді використовуються для описання неврологічних систем і в лінгвістиці для описання граматики природних мов.

**Початковий стан.** Початковий стан зазвичай показується зі стрілкою з невизначеного напряму.

**Допустимі (або кінцеві) стани.** Допустимі стани (також відомі як кінцеві стани) це такі, що якщо автомат знаходитьться в них це означає, що входний рядок, наскільки він опрацьований, належить мові що розпізнається. Зазвичай позначається двома колами.

## А Текст

## Зображення

 Документи

## Веб-сайти

Визначити мову **англійська** **українська** ▾

українська англійська

**finite**

'finit

**Прикметник** Переглянути в словнику.



кінцевий

кінцевий

termina

скінчений

finite



[Більше перекладів](#)

## Розгорнути все

обмежений

Прикме..

limited

restricted

confined

**finite**

small

rest ▼

[Надіслати відгук](#)



# Глухов В. С. «Комп'ютерна логіка схем з пам'яттю»

## 2 ЧАСОВІ ФУНКЦІЇ АЛГЕБРИ ЛОГІКИ

### 2.1 Елемент затримки

...

### 2.2 Дискретний час

...

### 2.3 Часові ФАЛ 1-, 2- та 3-го роду

У часових ФАЛ 1-го роду функція алгебри логіки  $f$ , яка залежить від змінних  $a$ ,  $b$ ,  $c$ , ..., також залежить і від часу – змінної  $t$  (рис. 2.8, а).

У часових ФАЛ 2-го роду функція алгебри логіки  $f$ , яка залежить від значень змінних  $a_t$ ,  $b_t$ ,  $c_t$ , ... у теперішній момент часу  $t$ , також залежить і від їхніх значень  $a_{t-1}$ ,  $b_{t-1}$ ,  $c_{t-1}$ , ... у попередній момент часу  $t-1$  (рис. 2.8, б).

У часових ФАЛ 3-го роду значення  $Y_t$ , функції алгебри логіки  $f$  у теперішній момент часу  $t$  залежить від значень змінних  $a_t$ ,  $b_t$ ,  $c_t$ , ... у теперішній момент часу  $t$ , а також і від свого власного значення  $Y_{t-1} \dots$  у попередній момент часу  $t-1$  (рис. 2.8, в).

Будь-який зворотний зв'язок (на рис. 2.8 його позначено товстішою лінією) і є функцією, яка змінює час – на одному кінці цієї лінії, на виході елемента  $F$ , час теперішній, а на другому її кінці, на вході елемента  $F$ , час вже минулій. Зворотний зв'язок забезпечує плин часу, постійно перетворює теперішній час у минулій.

# Глухов В. С. «Комп'ютерна логіка схем з пам'яттю»

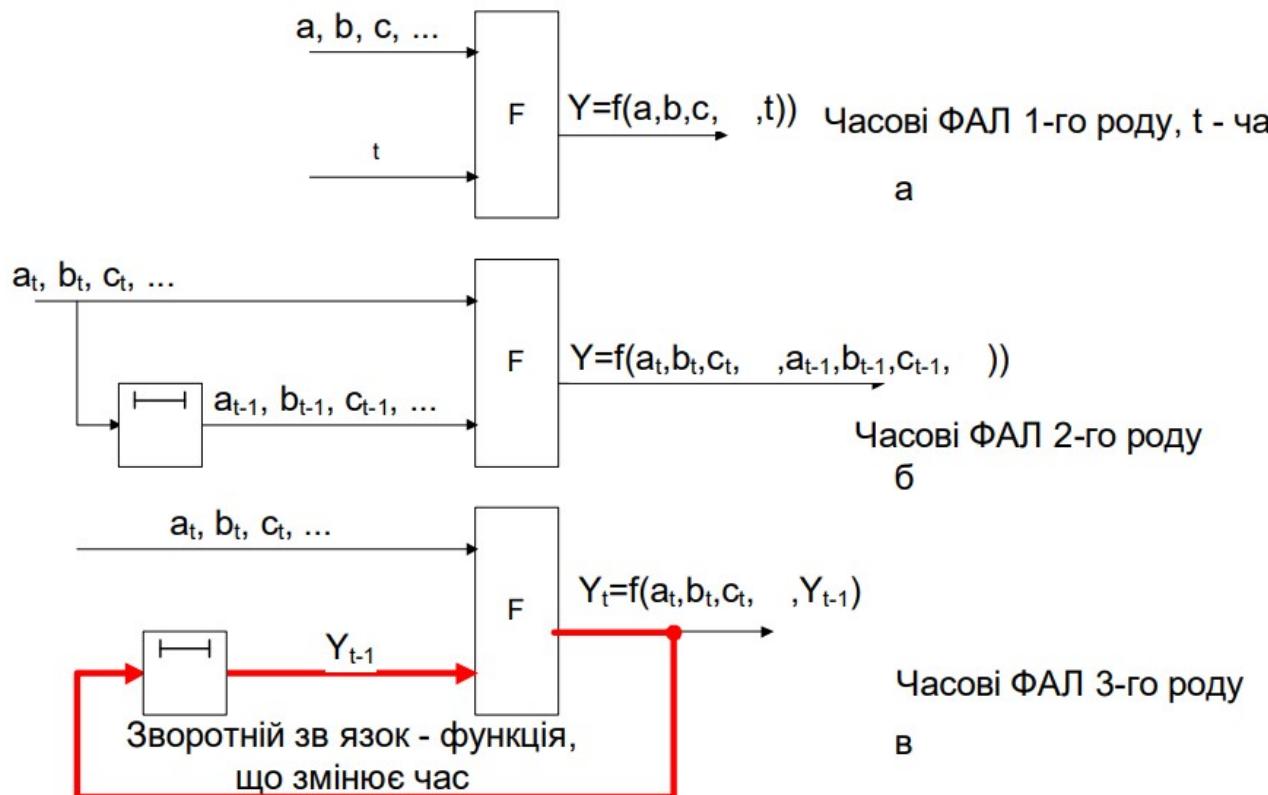
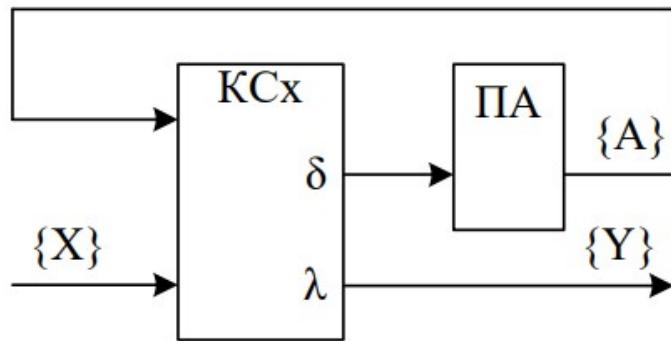


Рис. 2.8. Часові ФАЛ 1-, 2-та 3-го роду

# Глухов В. С. «Комп'ютерна логіка схем з пам'яттю»

## Загальна структурна схема цифрового автомата

Загальну структурну схему цифрового автомата представлено на рис. 1.11.



КСх - комбінаційна схема

ПА - пам'ять автомата

$\delta$  – функція переходів

$\lambda$  – функція виходів

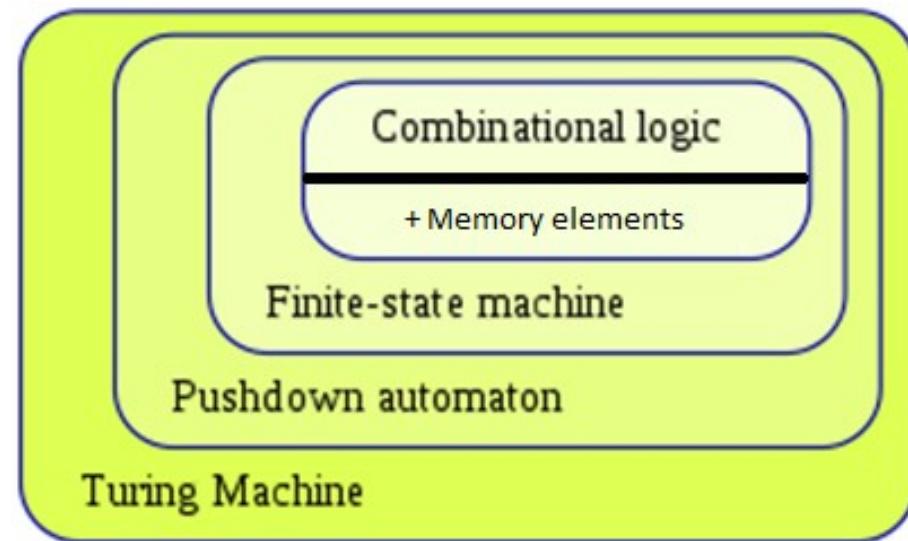
$\{X\}$  – множина вхідних сигналів

$\{Y\}$  – множина вихідних сигналів

$\{A\}$  – множина внутрішніх станів

*Rис. 1.11. Загальна структурна схема цифрового автомата*



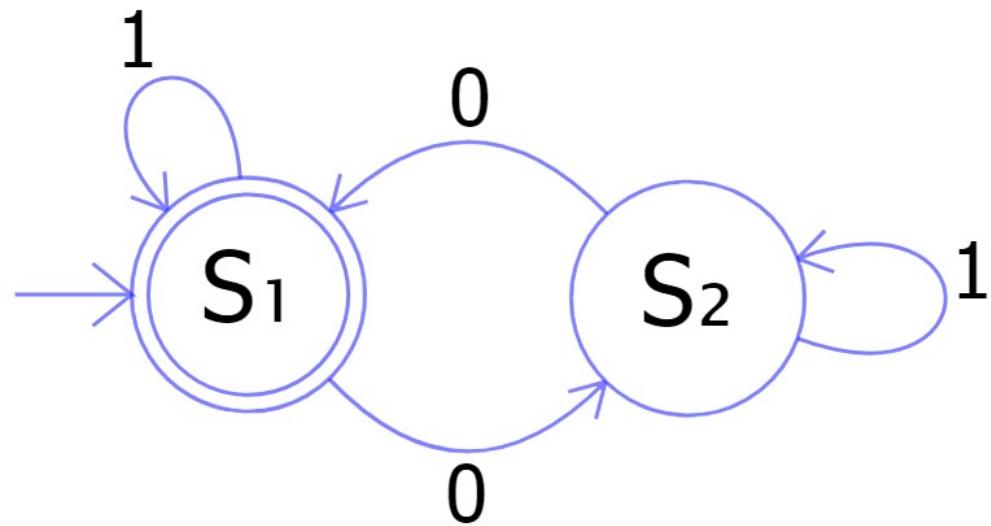


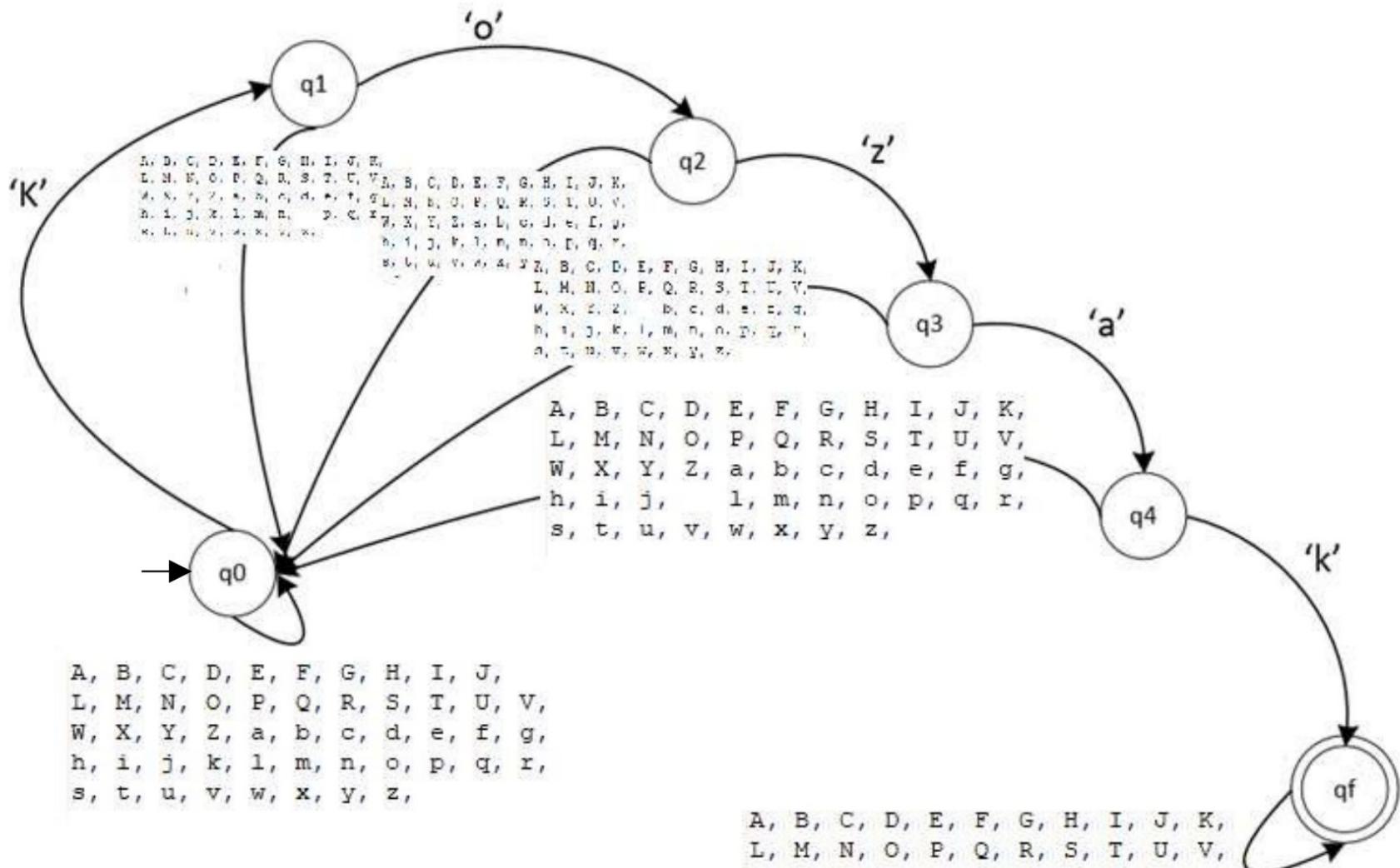
## 1.2.4. Скінчений автомат.

### 1.2.4.1. Детермінований скінчений автомат(DFA).

деметрінований скінчений автомат або детермінований скінчений автомат акцептор є  $(\Sigma, S, s_0, \delta, F)$ , де:

- $\Sigma$  вхідна абетка (скінчений, не порожній набір символів).
- $S$  – скінчений, не порожній набір станів.
- $s_0$  – початковий стан, елемент з  $S$ .
- $\delta$  – функція переходу:  $\delta : S \times \Sigma \rightarrow S$
- $F$  набір кінцевих станів, (можливо порожня) підмножина  $S$ .





```
#include "stdio.h"

#define DECLSTATE(NAME, ...) typedef enum {__VA_ARGS__, size##NAME} NAME;
#define GET_ENUM_SIZE(NAME) size##NAME

DECLSTATE(A,
          vA/*A*/,
          vB/*B*/,
          vC/*C*/,
          vD/*D*/,
          vE/*E*/,
          vF/*F*/,
          vG/*G*/,
          vH/*H*/,
          vI/*I*/,
          vJ/*J*/,
          vK/*K*/,
          vL/*L*/,
          vM/*M*/,
          vN/*N*/,
          vO/*O*/,
          vP/*P*/,
          vQ/*Q*/,
          vR/*R*/,
          vS/*S*/,
          vT/*T*/,
          vU/*U*/,
          vV/*V*/,
          vW/*W*/,
          vX/*X*/,
          vY/*Y*/,
          vZ/*Z*/,
          sZ/*[*/,
          sY/*\*/,
          sC/*]*/,
          sH/*^*/,
          sB/*_*/,
          sT/*`*/,
          va/*a*/,
          vb/*b*/,
          vc/*c*/,
          vd/*d*/,
          ve/*e*/,
```

```
vf/*f*/,
vg/*g*/,
vh/*h*/,
vi/*i*/,
vj/*j*/,
vk/*k*/,
vl/*l*/,
vm/*m*/,
vn/*n*/,
vo/*o*/,
vp/*p*/,
vq/*q*/,
vr/*r*/,
vs/*s*/,
vt/*t*/,
vu/*u*/,
vv/*v*/,
vw/*w*/,
vx/*x*/,
vy/*y*/,
vz/*z*/,
sF/*end mark*/
)

DECLSTATE(Q,
q0,
q1,
q2,
q3,
q4,
qf
)
```

```
typedef unsigned char INSTRUCTION;
typedef INSTRUCTION PROGRAM[GET_ENUM_SIZE(A)][GET_ENUM_SIZE(Q)];

PROGRAM program = {
    //   q0  q1  q2  q3  q4  qf
    /*A*/{q0, q0, q0, q0, q0, qf},
    /*B*/{q0, q0, q0, q0, q0, qf},
    /*C*/{q0, q0, q0, q0, q0, qf},
    /*D*/{q0, q0, q0, q0, q0, qf},
    /*E*/{q0, q0, q0, q0, q0, qf},
    /*F*/{q0, q0, q0, q0, q0, qf},
    /*G*/{q0, q0, q0, q0, q0, qf},
    /*H*/{q0, q0, q0, q0, q0, qf},
    /*I*/{q0, q0, q0, q0, q0, qf},
    /*J*/{q0, q0, q0, q0, q0, qf},
    /*K*/{q1, q0, q0, q0, q0, qf},
    /*L*/{q0, q0, q0, q0, q0, qf},
    /*M*/{q0, q0, q0, q0, q0, qf},
    /*N*/{q0, q0, q0, q0, q0, qf},
    /*O*/{q0, q0, q0, q0, q0, qf},
    /*P*/{q0, q0, q0, q0, q0, qf},
    /*Q*/{q0, q0, q0, q0, q0, qf},
    /*R*/{q0, q0, q0, q0, q0, qf},
    /*S*/{q0, q0, q0, q0, q0, qf},
    /*T*/{q0, q0, q0, q0, q0, qf},
    /*U*/{q0, q0, q0, q0, q0, qf},
    /*V*/{q0, q0, q0, q0, q0, qf},
    /*W*/{q0, q0, q0, q0, q0, qf},
    /*X*/{q0, q0, q0, q0, q0, qf},
    /*Y*/{q0, q0, q0, q0, q0, qf},
    /*Z*/{q0, q0, q0, q0, q0, qf},
    /*[*/{q0, q0, q0, q0, q0, qf},
```

```
/*\*/{q0, q0, q0, q0, q0, qf},  
/*]/*/{q0, q0, q0, q0, q0, qf},  
/*^*/{q0, q0, q0, q0, q0, qf},  
/*_*/{q0, q0, q0, q0, q0, qf},  
/*`*/{q0, q0, q0, q0, q0, qf},  
/*a*/{q0, q0, q0, q4, q0, qf},  
/*b*/{q0, q0, q0, q0, q0, qf},  
/*c*/{q0, q0, q0, q0, q0, qf},  
/*d*/{q0, q0, q0, q0, q0, qf},  
/*e*/{q0, q0, q0, q0, q0, qf},  
/*f*/{q0, q0, q0, q0, q0, qf},  
/*g*/{q0, q0, q0, q0, q0, qf},  
/*h*/{q0, q0, q0, q0, q0, qf},  
/*i*/{q0, q0, q0, q0, q0, qf},  
/*j*/{q0, q0, q0, q0, q0, qf},  
/*k*/{q0, q0, q0, q0, qf, qf},  
/*l*/{q0, q0, q0, q0, q0, qf},  
/*m*/{q0, q0, q0, q0, q0, qf},  
/*n*/{q0, q0, q0, q0, q0, qf},  
/*o*/{q0, q2, q0, q0, q0, qf},  
/*p*/{q0, q0, q0, q0, q0, qf},  
/*q*/{q0, q0, q0, q0, q0, qf},  
/*r*/{q0, q0, q0, q0, q0, qf},  
/*s*/{q0, q0, q0, q0, q0, qf},  
/*t*/{q0, q0, q0, q0, q0, qf},  
/*u*/{q0, q0, q0, q0, q0, qf},  
/*v*/{q0, q0, q0, q0, q0, qf},  
/*w*/{q0, q0, q0, q0, q0, qf},  
/*x*/{q0, q0, q0, q0, q0, qf},  
/*y*/{q0, q0, q0, q0, q0, qf},  
/*z*/{q0, q0, q3, q0, q0, qf}  
};
```

```
typedef struct structDFA{
    unsigned char * data;
    PROGRAM * program;
    void(*run)(struct structDFA * dfa);
    Q state;
} DFA;

void runner(DFA * dfa){
    for (; *dfa->data != sF; ++dfa->data){
        dfa->state = (*dfa->program)[*dfa->data][dfa->state];
    }
}

#define MAX_TEXT_SIZE 256
int main(){
    unsigned char data[MAX_TEXT_SIZE] = "Kozak Nazar Bohdanovych", *data_ = data;
    DFA dfa = { data, (PROGRAM*)program, runner, q0 };

    for (; *data_; *data_ -= 'A', *data_ %= GET_ENUM_SIZE(A), ++data_);
    *data_ = sF;
    dfa.run(&dfa);

    if (dfa.state == qf){
        printf("DFA: finit state\r\n");
    }
    else{
        printf("DFA: no finit state\r\n");
    }

    getchar();
    return 0;
}
```

```
#include "stdio.h"

#define DECLSTATE(NAME, ...) typedef enum {__VA_ARGS__, size##NAME} NAME;
#define GET_ENUM_SIZE(NAME) size##NAME

DECLSTATE(A,
          vA/*A*/,
          vB/*B*/,
          vC/*C*/,
          vD/*D*/,
          vE/*E*/,
          vF/*F*/,
          vG/*G*/,
          vH/*H*/,
          vI/*I*/,
          vJ/*J*/,
          vK/*K*/,
          vL/*L*/,
          vM/*M*/,
          vN/*N*/,
          vO/*O*/,
          vP/*P*/,
          vQ/*Q*/,
          vR/*R*/,
          vS/*S*/,
          vT/*T*/,
          vU/*U*/,
          vV/*V*/,
          vW/*W*/,
          vX/*X*/,
          vY/*Y*/,
          vZ/*Z*/,
          sZ/*[*/,
          sY/*\*/,
          sC/*]*/,
          sH/*^*/,
          sB/*_*/,
          sT/*`*/,
          va/*a*/,
          vb/*b*/,
          vc/*c*/,
          vd/*d*/,
          ve/*e*/,
          vf/*f*/.
```

```
vg/*g*/,
vh/*h*/,
vi/*i*/,
vj/*j*/,
vk/*k*/,
vl/*l*/,
vm/*m*/,
vn/*n*/,
vo/*o*/,
vp/*p*/,
vq/*q*/,
vr/*r*/,
vs/*s*/,
vt/*t*/,
vu/*u*/,
vv/*v*/,
vw/*w*/,
```

```

vx/*x*/,
vy/*y*/,
vz/*z*/,
sF/*end mark*/
)

DECLSTATE(Q,
q0,
q1,
q2,
q3,
q4,
qf
)

typedef unsigned char INSTRUCTION;
#define MAX_RULE_COUNT 16
typedef struct {
    unsigned char input;
    INSTRUCTION toState;
} SimpleRule;
typedef SimpleRule PROGRAM[GET_ENUM_SIZE(Q)][MAX_RULE_COUNT];

#define DEFAULT GET_ENUM_SIZE(A)

PROGRAM program = {
    /           q0           q1           q2
q3           q4           qf
    { { vK, q1 }, { DEFAULT, q0 } }, { { vo, q2 }, { DEFAULT, q0 } }, { { vz, q3 }, { DEFAULT, q0 } },
    { { va, q4 }, { DEFAULT, q0 } }, { { vk, qf }, { DEFAULT, q0 } }, { { DEFAULT, qf } },
};

typedef struct structDFA{
    unsigned char * data;
    PROGRAM * program;
    void(*run)(struct structDFA * dfa);
    Q state;
} DFA;

```

```
void runner(DFA * dfa){
    for (; *dfa->data != sF; ++dfa->data){
        SimpleRule *prevRule, *rule;
        for (prevRule = rule = (*dfa->program)[dfa->state]; prevRule->input != DEFAULT; prevRule = rule, ++rule){
            if (DEFAULT == rule->input){
                dfa->state = rule->toState;
                break;
            }
            else if (*dfa->data == rule->input){
                dfa->state = rule->toState;
                break;
            }
        }
    }
}

#define MAX_TEXT_SIZE 256
int main(){
    unsigned char data[MAX_TEXT_SIZE] = "Kozak Nazar Bohdanovych", *data_ = data;
    DFA dfa = { data, (PROGRAM*)program, runner, q0 };

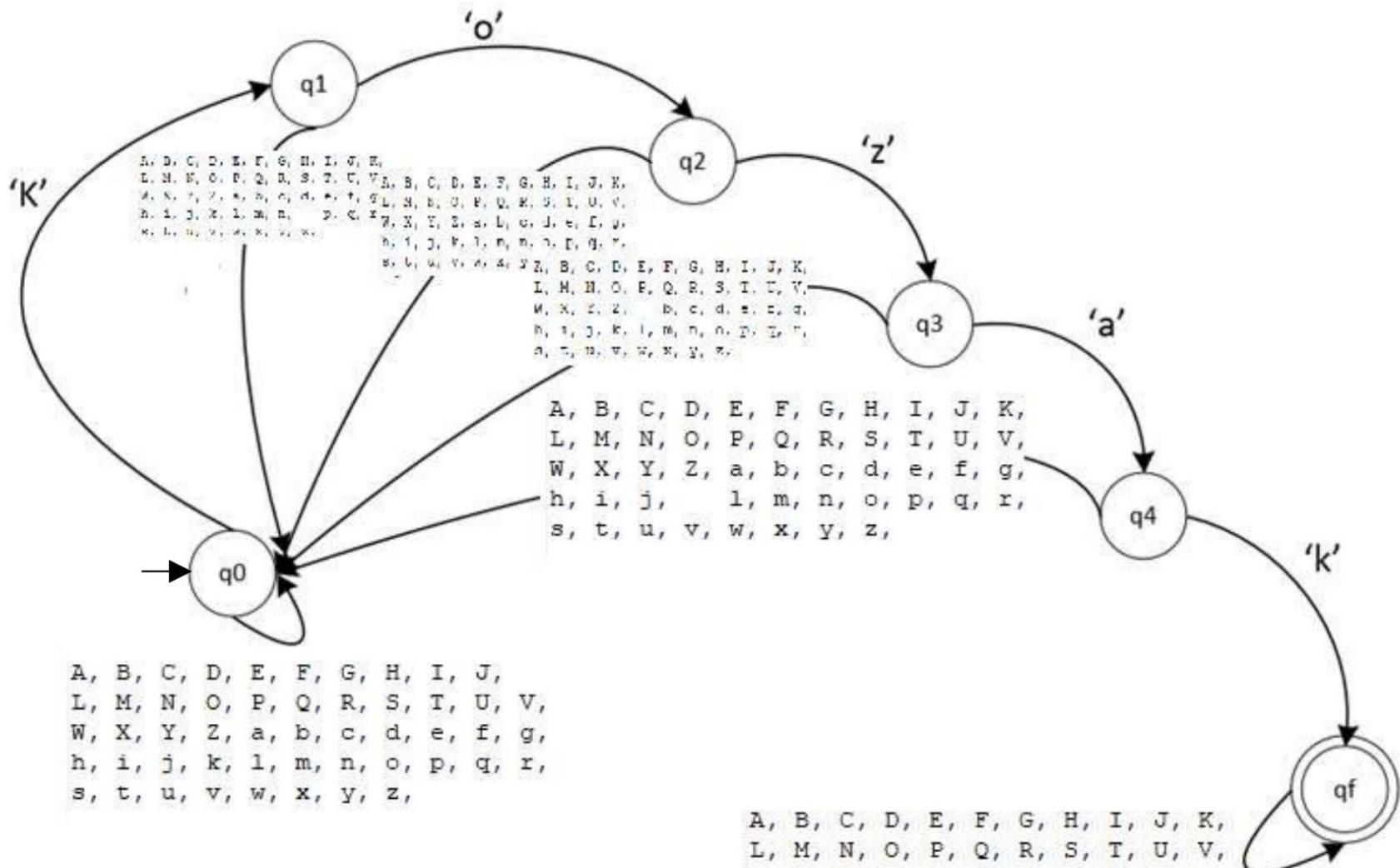
    for (; *data_; *data_ -= 'A', *data_ %= GET_ENUM_SIZE(A), ++data_);
        *data_ = sF;
    dfa.run(&dfa);

    if (dfa.state == qf){
        printf("DFA: finit state\r\n");
    }
    else{
        printf("DFA: no finit state\r\n");
    }

    getchar();
    return 0;
}
```

# Контрольне завдання №3

Нарисувати граф, що відображає детермінований скінчений автомат для виявлення у вхідній стрічці вашого прізвища.



## 1.2.4. Скінчений автомат.

### 1.2.4.1. Недетермінований скінчений автомат(NFA).

**Недетермінований автомат** це автомат, який при даному вхідному символі і внутрішньому стані може переходити в декілька різних внутрішніх станів.

НСА формально представляє п'ятірка,  $(Q, \Sigma, \Delta, q_0, F)$ , в якій:

- скінчена множина станів  $Q$
- скінчена множина вхідних символів  $\Sigma$
- функція переходу  $\Delta : Q \times \{\Sigma \cup \epsilon\} \rightarrow P(Q)$ .
- початковий стан  $q_0 \in Q$
- набір станів  $F$  позначених як *допустимі* (або *кінцеві*) стани  $F \subseteq Q$ .

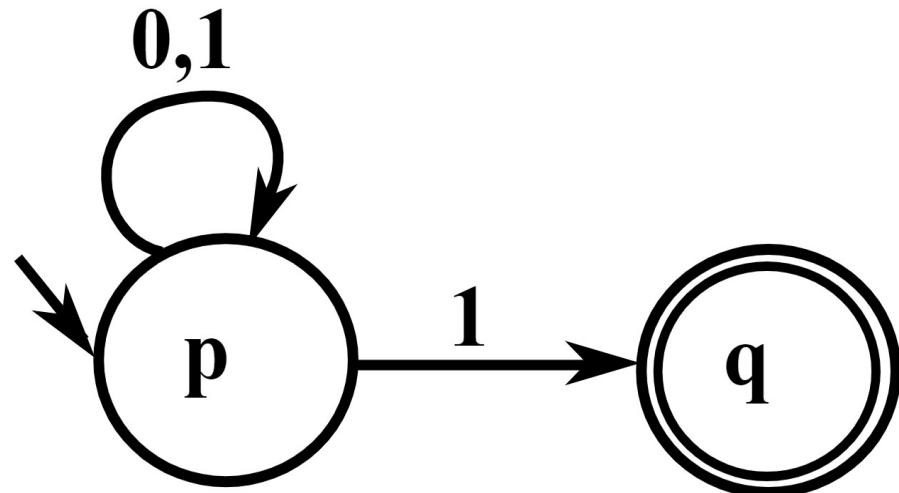
Тут,  $P(Q)$  позначає множину всіх підмножин  $Q$ . Нехай  $w = a_1 a_2 \dots a_n$  буде словом в абетці  $\Sigma$ .

Автомат  $M$  приймає слово  $w$  якщо послідовність станів,  $r_0, r_1, \dots, r_n$ , існує в  $Q$  з такими умовами:

$$r_0 = q_0$$

$$r_{i+1} \in \Delta(r_i, a_{i+1}), \text{ for } i = 0, \dots, n-1$$

$$r_n \in F.$$

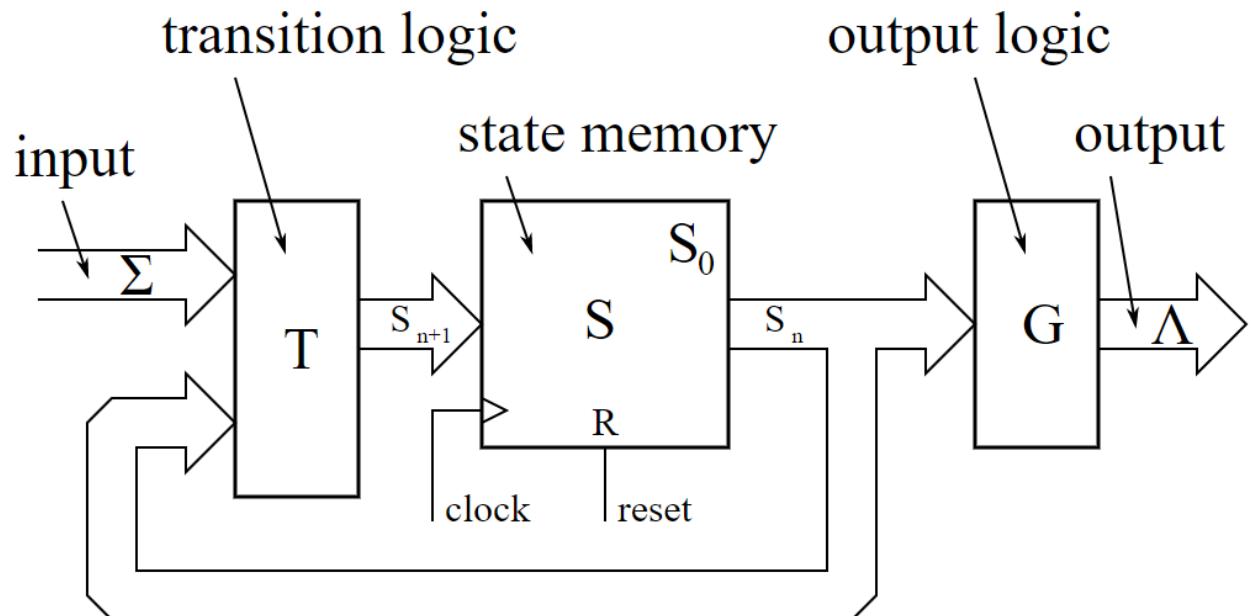


## 1.2.5. Перетворювачі(трансдуктори) на основі детермінованого скінченного автомата.

### 1.2.5.1. Автомат Мура(Moore machine).

Автомат Мура це 6 елементів ( $S$ ,  $S_0$ ,  $\Sigma$ ,  $\Lambda$ ,  $T$ ,  $G$ ):

- множина внутрішніх станів  $S$  (внутрішній алфавіт);
- початковий стан  $S_0$ , який є елементом ( $S$ );
- скінчена множина вхідних сигналів  $\Sigma$  (вхідний алфавіт);
- скінчена множина вихідних сигналів  $\Lambda$  (вихідний алфавіт);
- функція переходу ( $T: S \times \Sigma \rightarrow S$ ), яка відображає стан і вхідний алфавіт у наступний стан;
- виходна функція ( $G: S \rightarrow \Lambda$ ), яка відображає стан у вихідний алфавіт.

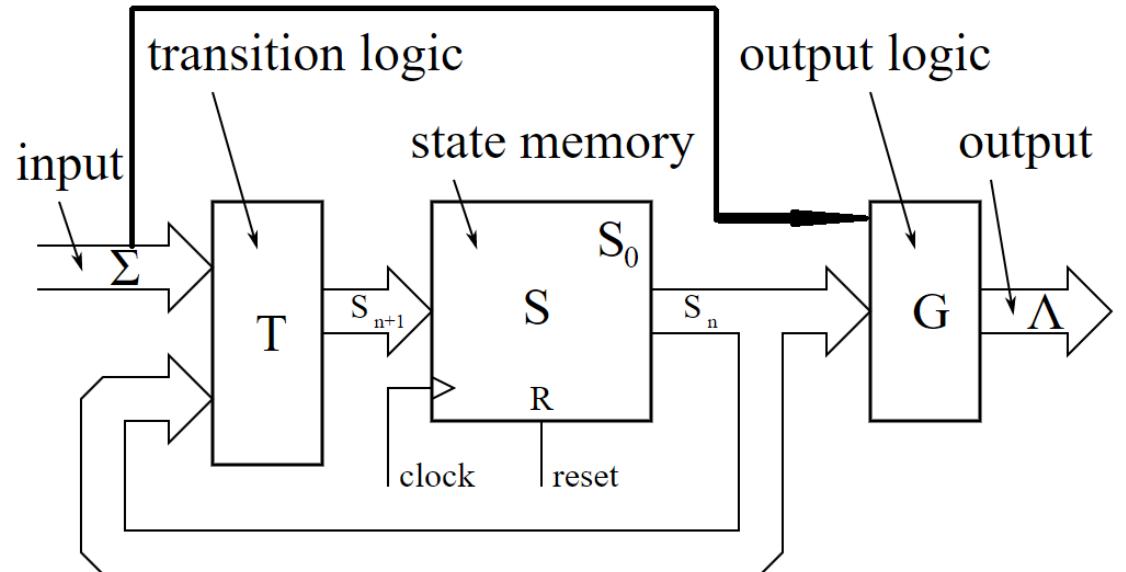


## 1.2.5. Перетворювачі(трансдуктори) на основі детермінованого скінченного автомата.

### 1.2.5.2. АВТОМАТ МІЛІ(Mealy machine).

Аutomat Mіlі це 6 елементів ( $S$ ,  $S_0$ ,  $\Sigma$ ,  $\Lambda$ ,  $T$ ,  $G$ ):

- множина внутрішніх станів  $S$  (внутрішній алфавіт);
- початковий стан  $S_0$ , який є елементом ( $S$ );
- скінчена множина вхідних сигналів  $\Sigma$  (вхідний алфавіт);
- скінчена множина вихідних сигналів  $\Lambda$  (вихідний алфавіт);
- функція переходу ( $T: S \times \Sigma \rightarrow S$ ), яка відображає стан і вхідний алфавіт у наступний стан;
- вихідна функція ( $G: S \rightarrow \Lambda$ ), яка відображає стан і вхідний алфавіт у вихідний алфавіт.



## 1.2.6. Магазинний автомат(PDA).

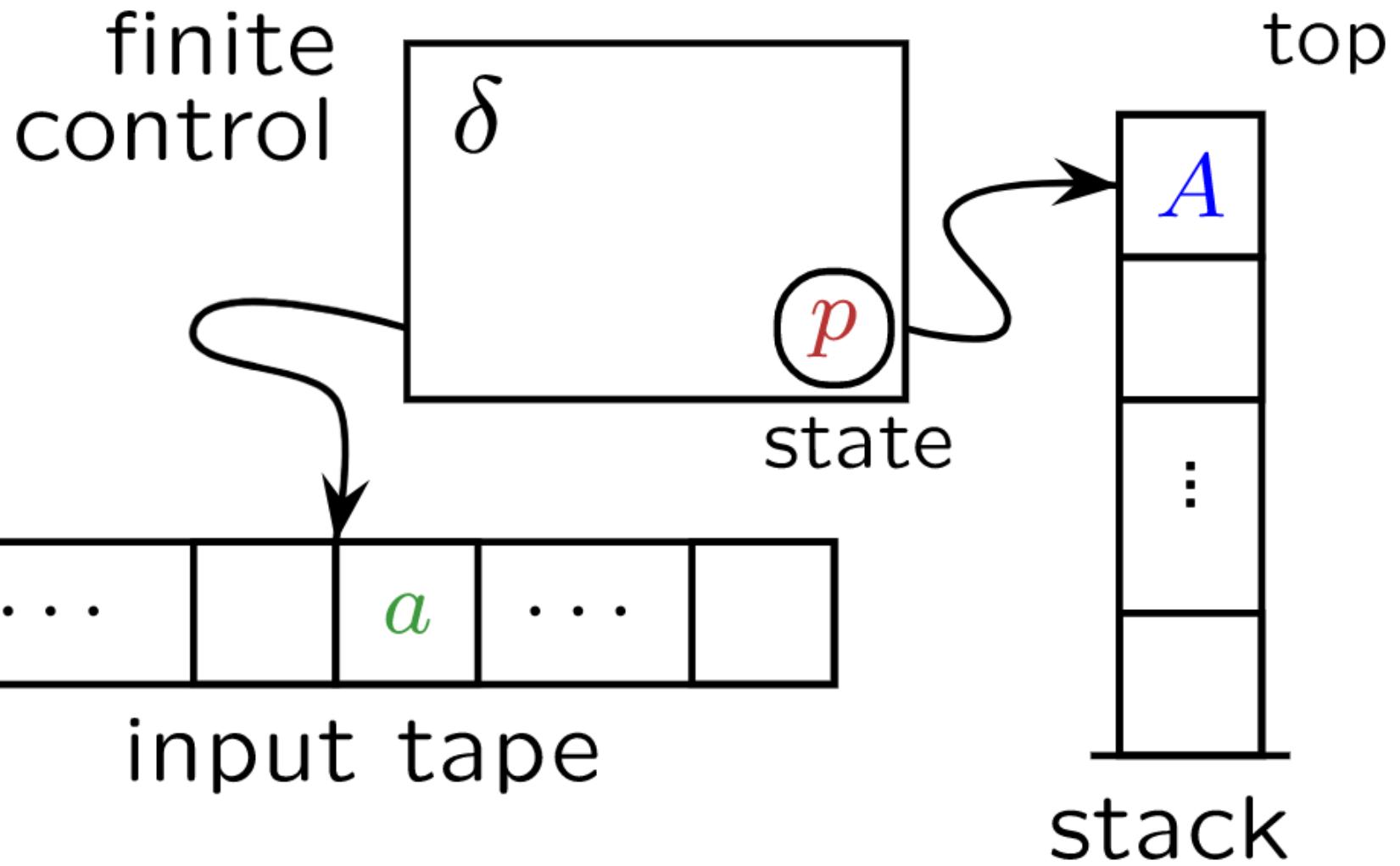
Автомат з магазинною пам'яттю (МП автомат) використовує стек для зберігання станів.

$M = (K, \Sigma, \pi, s, F, S, m)$ , де

- $K$  — скінчена множина станів автомата
- $s \in K$  — єдиний допустимий початковий стан автомата
- $F \subset K$  — множина кінцевих станів, причому допускається  $F=\emptyset$ , і  $F=K$
- $\Sigma$  — скінчена множина символів вхідного алфавіту, з якого формуються строки, що читаються автоматом
- $S$  — алфавіт пам'яті (магазину)
- $m \in S$  — нульовий символ пам'яті.
- $\pi$  — функція переходів:  $\pi : K \times \Sigma \times S \rightarrow K \times S$

Пам'ять працює як стек, тобто для читання доступний останній записаний в неї елемент. Таким чином, функція переходу є відображенням  $\pi : K \times \Sigma \times S \rightarrow K \times S$ . Тобто, по комбінації поточного стану, вхідного символу і символу на вершині магазину автомат вибирає наступний стан і, можливо, символ для запису в магазин. У випадку, коли у правій частині автоматного правила присутній  $e$ , в магазин нічого не додається, а елемент з вершини стирається. Якщо магазин порожній, то спрацьовують правила з  $e$  в лівій частині.

У чистому вигляді автомати з магазинною пам'яттю використовуються вкрай рідко. Зазвичай ця модель використовується для наочного подання відмінності звичайних скінчених автоматів від синтаксичних граматик. Реалізація автоматів з магазинною пам'яттю відрізняється від кінцевих автоматів тим, що поточний стан автомата сильно залежить від будь-якого попереднього.

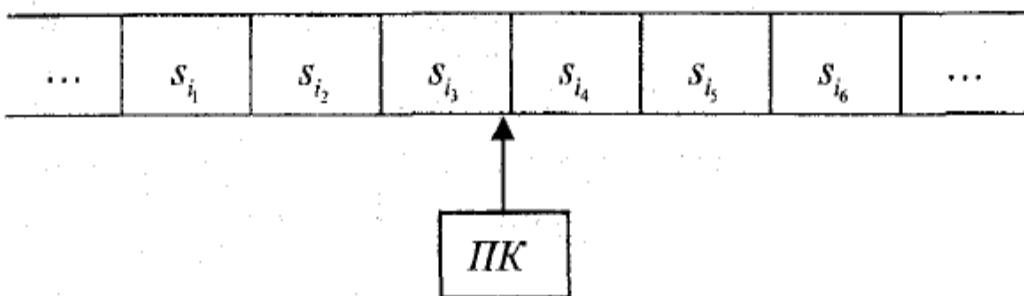


## 1.2.7. Машина Тюрінга та теза Черча.

### 1.2.7.1. Машина Тюрінга.

#### Загальна концепція

- На змістовному рівні машина Тюрінга (МТ) є деякою гіпотетичною(умовою) машиною, яка складається з трьох головних компонентів: *інформаційної стрічки, головки для зчитування і запису та пристрою керування*.
- В клітинах живих організмів є кілька механізмів, які є аналогами машини Тюрінга



## 1.2.7. Машина Тюрінга та теза Черча.

### 1.2.7.1. Машина Тюрінга.

#### Формальне визначення

Модель однострічкової детермінованої МТ:

$$M = \langle A, I, Q, q_0, q_f, a_0, p \rangle,$$

де  $A$  – кінцева множина символів зовнішнього алфавіту,

$I$  – кінцева множина символів зовнішнього алфавіту на стрічці,

$Q$  – кінцева множина символів внутрішнього алфавіту,

$q_0$  – початковий стан,

$q_f$  – кінцевий стан,

$$q_0, q_f \in Q$$

$a_0$  – позначення порожньої комірки стрічки,

$p$  – така програма, яка не може мати двох команд, у яких би збігалися два перші символи:

$$\{A\}x\{Q\}^\diamond \{A\}\{L,R,S\}\{Q\},$$

де  $L$  – зсувати головку вліво,

$R$  – зсувати головку вправо,

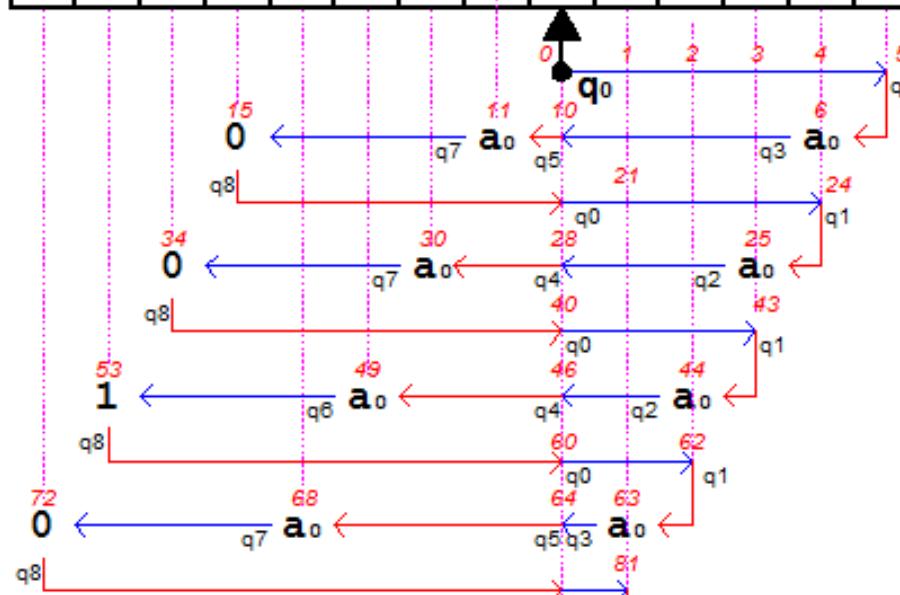
$S$  – головка залишається на місці.

A	Q	$q_0$	$q_1$	$q_2$	$q_3$	$q_4$	$q_5$	$q_6$	$q_7$	$q_8$
$a_0$	$Lq_1$					$Lq_4$	$Lq_5$	$1Rq_8$	$0Rq_8$	$Rq_8$
0	$Rq_0$	$a_0Lq_3$	$Lq_2$	$Lq_3$	$a_0Lq_7$	$a_0Lq_7$	$Lq_6$	$Lq_7$	$Rq_8$	
1	$Rq_0$	$a_0Lq_2$	$Lq_2$	$Lq_3$	$a_0Lq_6$	$a_0Lq_7$	$Lq_6$	$Lq_7$	$Rq_8$	
$\wedge$	$Rq_0$	$q_f$	$Lq_4$	$Lq_5$						$Rq_0$

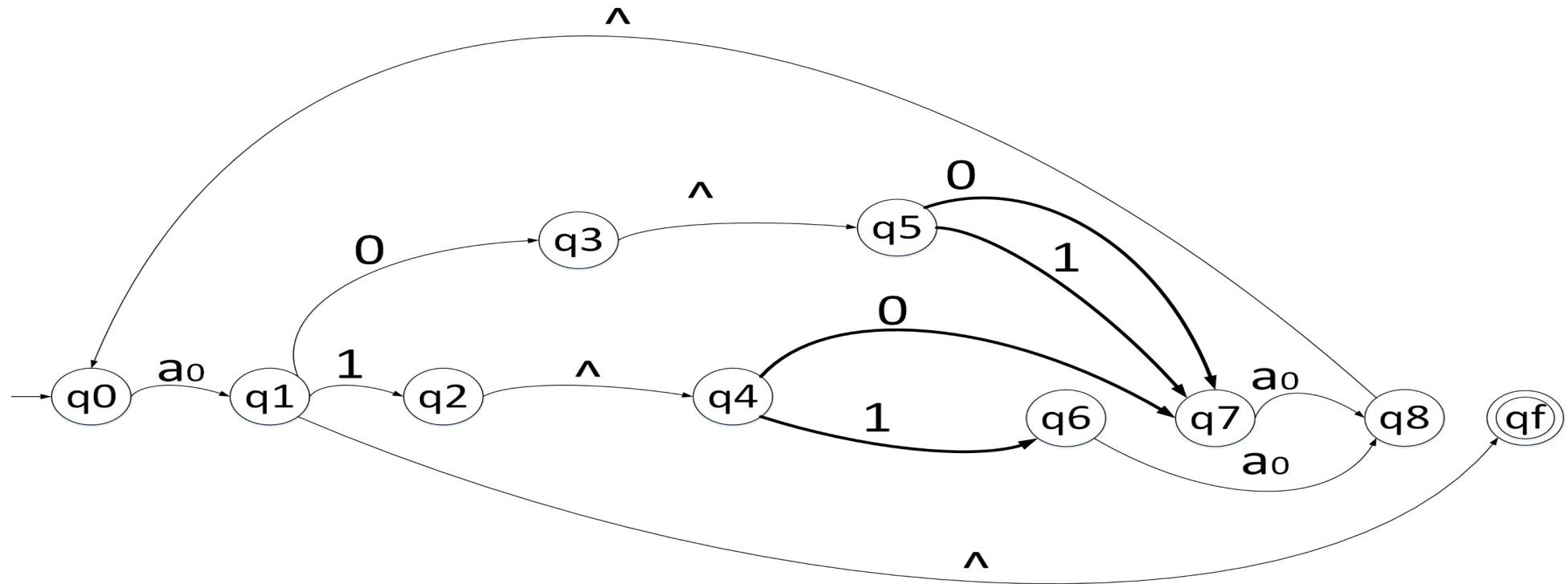
P = 29

$a_0 \ a_0 \ a_0 \ a_0 \ 0 \ 1 \ 0 \ 1 \ \wedge \ 0 \ 1 \ 1 \ 0 \ a_0$

M=14



(X  $\wedge$  Y), де X= 0101, Y = 0110.  $q_f$  L=82



A	Q	$q_0$	$q_1$	$q_2$	$q_3$	$q_4$	$q_5$	$q_6$	$q_7$	$q_8$
$a_0$		$Lq_1$				$Lq_4$	$Lq_5$	$1Rq_8$	$0Rq_8$	$Rq_8$
0		$Rq_0$	$a_0Lq_3$	$Lq_2$	$Lq_3$	$a_0Lq_7$	$a_0Lq_7$	$Lq_6$	$Lq_7$	$Rq_8$
1		$Rq_0$	$a_0Lq_2$	$Lq_2$	$Lq_3$	$a_0Lq_6$	$a_0Lq_7$	$Lq_6$	$Lq_7$	$Rq_8$
$\wedge$		$Rq_0$	$q_f$	$Lq_4$	$Lq_5$					$Rq_0$

```
#include <stdio.h>

#define DECLSTATE(NAME, ...) typedef enum {__VA_ARGS__, size##NAME} NAME;

#define GET_ENUM_SIZE(NAME) size##NAME

    DECLSTATE(A,
    a0,
    v0/* 0 */,
    v1/* 1 */,
    sT/* ^ */
    )

    DECLSTATE(Q,
    q0,
    q1,
    q2,
    q3,
    q4,
    q5,
    q6,
    q7,
    q8,
    qf
```

```
)  
  
#define NO_ACTION 0x7f  
  
#define NO_RULE {NO_ACTION, NO_ACTION, NO_ACTION}  
  
#define S 0  
#define L 1  
#define R 2  
  
typedef unsigned char INSTRUCTION[3];  
typedef INSTRUCTION PROGRAM[GET_ENUM_SIZE(A)][GET_ENUM_SIZE(Q)];  
  
PROGRAM initProgram = { /* default pass */  
    //          q0          q1          q2  
q3          q4          q5          q6  
q7          q8  
    /* a0 */{ { NO_ACTION, R, q1 }, { NO_ACTION, R, q1 }, { NO_ACTION, R, q1 }, {  
        NO_ACTION, R, q1 }, { NO_ACTION, R, q1 }, { NO_ACTION, R, q1 }, { NO_ACTION, R, q1 }, {  
        NO_ACTION, R, q1 }, { NO_ACTION, R, q1 } },  
    /* 0 */{ { NO_ACTION, R, q1 }, { NO_ACTION, R, q1 }, { NO_ACTION, R, q1 }, {  
        NO_ACTION, R, q1 }, { NO_ACTION, R, q1 }, { NO_ACTION, R, q1 }, { NO_ACTION, R, q1 }, {  
        NO_ACTION, R, q1 }, { NO_ACTION, R, q1 } },  
    /* 1 */{ { NO_ACTION, R, q1 }, { NO_ACTION, R, q1 }, { NO_ACTION, R, q1 }, {  
        NO_ACTION, R, q1 }, { NO_ACTION, R, q1 }, { NO_ACTION, R, q1 }, { NO_ACTION, R, q1 }, {  
        NO_ACTION, R, q1 }, { NO_ACTION, R, q1 } },  
    /* ^ */{ { NO_ACTION, S, q0 }, { NO_ACTION, S, q0 }, { NO_ACTION, S, q0 }, {  
        NO_ACTION, S, q0 }, { NO_ACTION, S, q0 }, { NO_ACTION, S, q0 }, { NO_ACTION, S, q0 }, {  
        NO_ACTION, S, q0 }, { NO_ACTION, S, q0 } }  
};
```

```

PROGRAM program = { /* default pass */
    //          q0          q1          q2
    q3          q4          q5          q6
    q7          q8
    /* a0 */{ { NO_ACTION, L, q1 }, NO_RULE,
    NO_RULE,           { NO_ACTION, L, q4 }, { NO_ACTION, L, q5 }, { v1, R, q8 },      {
    v0, R, q8 },           { NO_ACTION, R, q8 } },
    /* 0 */{ { NO_ACTION, R, q0 }, { a0, L, q3 },           { NO_ACTION, L, q2 }, {
    NO_ACTION, L, q3 }, { a0, L, q7 },           { a0, L, q7 },           { NO_ACTION, L, q6 }, {
    NO_ACTION, L, q7 }, { NO_ACTION, R, q8 } },
    /* 1 */{ { NO_ACTION, R, q0 }, { a0, L, q2 },           { NO_ACTION, L, q2 }, {
    NO_ACTION, L, q3 }, { a0, L, q6 },           { a0, L, q7 },           { NO_ACTION, L, q6 }, {
    NO_ACTION, L, q7 }, { NO_ACTION, R, q8 } },
    /* ^ */{ { NO_ACTION, R, q0 }, { NO_ACTION, S, qf }, { NO_ACTION, L, q4 }, {
    NO_ACTION, L, q5 }, NO_RULE,           NO_RULE,           NO_RULE,
    NO_RULE,           { NO_ACTION, R, q0 } }
};

typedef unsigned char tapeElementType;
#define TAPE_SIZE 256

//#define MT_BEGIN_POSITION_STATE 127

typedef struct structMT{
    tapeElementType tape[TAPE_SIZE];
    PROGRAM * initProgram;
    PROGRAM * program;
    void(*stepRun)(struct structMT * mt, PROGRAM program);
    void(*run)(struct structMT * mt);
    void(*statePrint)(struct structMT * mt);
    unsigned char * debugType;
    Q state;
    unsigned int positionState;
}

```

```
 } MT;

void stepRunner(MT * mt, PROGRAM program){
    INSTRUCTION * instruction;

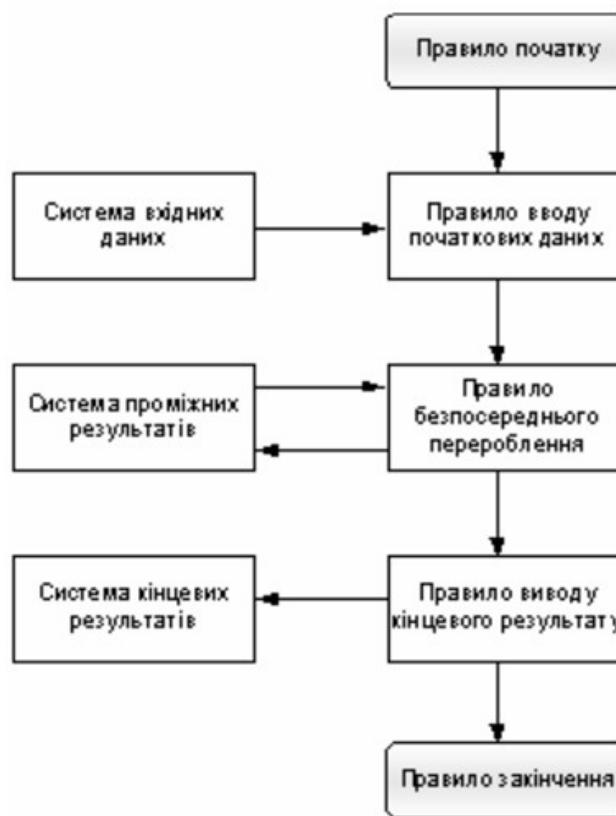
    if (!mt){
        return;
    }

    instruction = program[mt->tape[mt->positionState]][mt->state];

    if ((*instruction)[0] != NO_ACTION){
        mt->tape[mt->positionState] = (*instruction)[0];
    }

    if ((*instruction)[1] == L){
        --mt->positionState;
    }
    else if ((*instruction)[1] == R){
        ++mt->positionState;
    }

    mt->state = (*instruction)[2];
}
```



Для формального опису параметричної моделі використаємо кортеж параметрів:

$$M_{par} : \langle A, Q, q_0, q_f, I, O, W \rangle,$$

де  $A$  – кінцева множина символів зовнішнього алфавіту, які позначають елементи систем вхідних даних, проміжних та кінцевих результатів;  $Q$  – кінцева множина символів внутрішнього алфавіту;  $q_0, q_f$  – початковий і кінцевий стани роботи моделі,  $q_0, q_f \in Q$ ;  $I, O$  – правила вводу і виводу;  $W$  – правило безпосереднього перероблення.

Розглянута параметрична модель, назвемо її узагальненою, може бути використана як основа для конструювання та дослідження нових і вже існуючих моделей. Ці моделі складають два класи: абстрактні алгоритми і формальні алгоритмічні системи. Вони розрізняються напрямком використання, а також повнотою та деталізацією параметрів алгоритму.

```
void runner(MT * mt){
    if (!mt){
        return;
    }

    // init
    mt->state = q0;
    mt->positionState = 0;
    do{
        mt->stepRun(mt, mt->initProgram);
    } while (mt->state != q0);

    printf("Begin tape state:\r\n");
    mt->statePrint(mt);
    printf("\r\n\r\n");

    printf("Tape state:\r\n");
    // run program
    while (mt->state != qf){
        mt->stepRun(mt, mt->program);
        if (mt->statePrint){
            mt->statePrint(mt);
            if (*mt->debugType == 2){
                _sleep(250);
            }
            else if (*mt->debugType == 3){
                getchar();
            }
            else{
                _sleep(25);
            }
        }
    }
}
```

```
#define MAX_PRINT_TAPE_ELEMENT_COUT 14
void statePrinter(MT * mt){
    unsigned int index = 0;
    if (!mt){
        return;
    }

    for (; index < MAX_PRINT_TAPE_ELEMENT_COUT; ++index){
        switch (mt->tape[index]){
            case a0:
                if (index == mt->positionState)
                    printf("[a0] ");
                else{
                    printf(" a0 ");
                }
                break;
            case v0:
                if (index == mt->positionState)
                    printf("[0] ");
                else{
                    printf(" 0 ");
                }
                break;
            case v1:
                if (index == mt->positionState)
                    printf("[1] ");
                else{
                    printf(" 1 ");
                }
                break;
            case sT:
                if (index == mt->positionState)
                    printf("[^] ");
                else{
                    printf(" ^ ");
                }
                break;
            default:
                if (index == mt->positionState)
                    printf("[?] ");
                else{
                    printf(" ? ");
                }
                break;
        }
    }

    if (mt->state == qf){
        printf(" qf");
    }
    else{
        printf(" q%d", mt->state);
    }

    printf("\r");
}
```

```
tapeElementType tape[TAPE_SIZE];

#define INIT_TAPE_DATE { a0, a0, a0, a0, v1, v0, v1, v1, sT, v1, v0, v1, v1 }

int main(){
    int ch;
    MT mt = { INIT_TAPE_DATE, initProgram, program, stepRunner, runner, statePrinter,
(char*)&ch };
    printf("mode: \r\ndefault - run all\r\n      '2' - live\r\n      '3' - live by press
\"Enter\"\r\n");
    ch = getchar();
    ch -= '0';

    mt.run(&mt);
    getchar();

    return 0;
}
```

# 1.2.7. Машина Тюрінга та теза Черча.

## 1.2.7.1. Машина Тюрінга.

### Варіанти машини Тюрінга

- Варіанти машини Тюрінга:
  - Змінна машина Тюрінга
  - Нейронна машина Тюрінга
  - Недетермінована машина Тюрінга
  - Квантова машина Тюрінга
  - Машина Поста-Тюрінга
  - Імовірісна машина Тюрінга
  - Незмінна машина Тюрінга
  - Незмінна машина Тюрінга з рухом праворуч
  - Багатострічкова машина Тюрінга
  - Симетрична машина Тюрінга
  - Повна машина Тюрінга
  - Однозначна машина Тюрінга
  - Універсальна машина Тюрінга
  - Машина Зенона

## 1.2.7. Машина Тюрінга та теза Черча.

### 1.2.7.2. Теза Черча.

Клас алгоритмічно-обчислюваних функцій збігається з класом частково-рекурсивних функцій, функцій обчислюваних за Тюрінгом та іншими формальними уточненнями інтуїтивного поняття алгоритму.

З цього випливає, що якщо функція належить до класу певної формалізації алгоритмічно-обчислюваної функції, то вона є алгоритмічно-обчислювана.

Теза не доводиться. А еквівалентність класів формалізмів – доведено.

## 1.2.8. Машина Поста.

- Машина Поста має нескінченну інформаційну стрічку, на яку записують інформацію в двійковому алфавіті {0,1}. Алгоритм задають як скінченну впорядковану послідовність пронумерованих правил — команд Поста.
- Є шість типів команд Поста:
  - 1) відмітити активну комірку стрічки, тобто записати в неї 1 і перейти до виконання i-ї команди;
  - 2) стерти відмітку активної комірки, тобто записати в неї 0 і перейти до виконання i-ї команди;
  - 3) змістити активну комірку на одну позицію вправо і перейти до виконання i-ї команди;
  - 4) змістити активну комірку на одну позицію вліво і перейти до виконання i-ї команди;
  - 5) якщо активна комірка відмічена, то перейти до виконання i-ї команди, інакше - до виконання j-ї команди;
  - 6) зупинка, закінчення роботи алгоритму.

# 1.2.9. Нормальні алгоритми Маркова.

Для формалізації поняття алгоритму А. Марков 1954 р. розробив систему нормальних алгоритмів. Алгоритми Маркова — це формальна математична система. Вони були основою для першої мови обробки рядів СОМІТ. Крім того, є подібність між моделлю Маркова і мовою СНОБОЛ, яка з'явилась після СОМІТ.

**Простою продукцією** (формулою підстановки) називають запис вигляду:

$$u \rightarrow w$$

, де  $u$ ,  $w$  - рядки в алфавіті  $V$ . У цьому разі  $V$  не містить символів ' $\rightarrow$ ' та ' $:$ '. Величину  $u$  називають ангицедентом , а  $w$  - консеквентом.

Вважають, що формула  $u \rightarrow w$  може бути застосована до рядка  $Z \in F$  , якщо є хоча б одне входження  $u$  в  $Z$ . В іншому випадку ця формула не застосовна до рядка  $Z$ . Якщо формула може бути застосована, то канонічне (перше ліворуч) входження  $u$  в  $Z$  замінюється на  $w$ . Наприклад, якщо формулу ' $ba$ '  $\rightarrow$  ' $c$ ' застосувати до входного рядка ' $ababab$ ' , то в підсумку отримаємо рядок ' $acbab$ '. Проте формула ' $baa$ '  $\rightarrow$  ' $c$ ' до рядка ' $ababab$ ' не може бути застосована.

**Заключною продукцією** (заключною підстановкою) називають запис вигляду:

$$u \rightarrow -w$$

**Упорядковану множину продукцій  $P_1, P_7, \dots, P_n$  називають нормальним алгоритмом, чи алгоритмом Маркова.**

# 1.2.9. Нормальні алгоритми Маркова.

## Приклад.

Нехай над словами з алфавіту  $V = \{a, b, c\}$  задано алгоритм з такими формулами підстановки

$$P_1 : 'ab' \rightarrow 'b',$$

$$P_2 : 'ac' \rightarrow 'c',$$

$$P_3 : 'aa' \rightarrow 'a'.$$

Цей алгоритм видає всі входження символу 'a' з рядка, за винятком випадку, коли 'a' є в кінці рядка.

Простежимо роботу алгоритму, якщо входний рядок має вигляд 'bacaaaba'. Нехай символ  $\Rightarrow$  означає результат перетворення, а підрядок, який підлягає заміні, будемо підкреслювати:

$$'bacaaaba' \Rightarrow 'bacaaba' \Rightarrow 'bacbaa' \Rightarrow 'bcbaa' \Rightarrow 'bcba'.$$

$P_1$

$P_1$

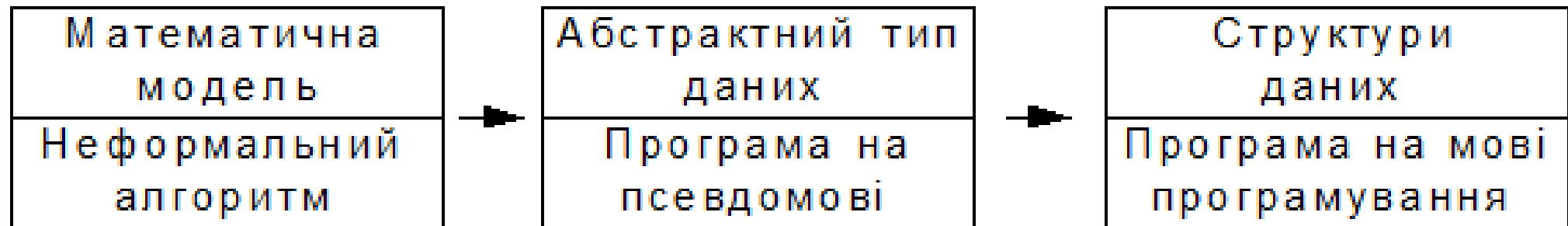
$P_2$

$P_3$

Оскільки далі жодна з формул не може бути застосована, то на цьому робота алгоритму завершується.



## 2.2.1. Покрокове проектування алгоритмів.



## 2.2.1. Покрокове проектування алгоритмів.

В основу процесу проектування програми з розбиттям її на достатню кількість дрібних кроків можна покласти наступну послідовність дій:

1. Вихідним станом процесу проектування є більш-менш точне формулювання мети алгоритму, або конкретніше – результату, який повинен бути отриманий при його виконанні. Формулювання проводиться на природній мові.
2. Проводиться збір фактів, які стосуються будь-яких характеристик алгоритму, і робиться спроба їх представлення засобами мови.
3. Створюється образна модель процесу, який відбувається, використовуються графічні та інші способи представлення, які дозволяють краще зрозуміти виконання алгоритму в динаміці.
4. В образній моделі виділяється найбільш суттєва частина, для якої підбирається найбільш точне словесне формулювання.
5. Проводиться визначення змінних, які необхідні для формального представлення даного кроку алгоритму.
6. Вибирається одна з конструкцій – проста послідовність дій, умовна конструкція або циклу. Складні частини вибраної формальної конструкції (умова, заголовок циклу) залишаються в словесному формулюванні.
7. Для інших неформалізованих частин алгоритму, які залишились у словесному формулюванні – перерахована послідовність дій повторюється.



## **2.1.1. Вербальне подання алгоритму. (Словесна форма)**

**Словесна форма** це словесно сформульована послідовність правил перетворення інформації. При цьому формальні правила перетворення інформації формулюються, нумеруються та вказується послідовність їх виконання. Така форма є прийнятною для досить простих або, навпаки, складних задач, розв'язання яких можна скласти з готових блоків (процедур обробки інформації), а за допомогою словесного опису вказати порядок їх виклику.

Приклад словесного опису алгоритму:

1. Прирівнюємо  $i$  до 1.
2. Прирівнюємо  $c_i$  до  $a_i + b_i$ .
3. Перевіримо, чи  $i$  дорівнює  $n$ . Якщо так, то обчислення припиняємо. Якщо ні, то збільшуємо  $i$  на 1 та переходимо до п. 2.

## **2.1.1. Вербальне подання алгоритму. (Словесна форма)**

**Словесно-формульна форма** це поєднання формул перетворення інформації та словесного визначення послідовності їх виконання. Використовує загальноприйняті математичні позначення, коментарі до них, що пояснюють дії, та їх послідовність, яка визначається за допомогою міток.

Для визначення порядку вводять мітки. Якщо не вказано, до якої мітки переходити, то вважається, що це перехід до наступної дії.

Приклад словесного-формульного опису алгоритму:

$i = 1$

Q1 :  $c_i = a_i + b_i$  .

Якщо  $i = n$ , то перейти до Q3 , інакше — до Q2.

Q2 :  $i = i + 1$ . Перейти до Q1.

Q3 : Закінчiti обчислювання.

## Подання алгоритму псевдокодом

**Псевдокод** — це неформальний запис алгоритму, який використовує структуру поширених мов програмування, але нехтує деталями коду, неістотними для розуміння алгоритму (опис типів, виклик підпрограм тощо).

\* Нет існує формальних правил написання псевдокоду.

# Подання алгоритму псевдокодом

Type of operation	Symbol	
Assignment	$\leftarrow$ or $:=$	$c \leftarrow 2\pi r$ , $c := 2\pi r$
Comparison	$=, \neq, <, >, \leq, \geq$	
Arithmetic	$+, -, \times, /, \text{mod}$	
Floor/ceiling	$\lfloor, \lceil, \lceil, \rfloor$	$a \leftarrow \lfloor b \rfloor + \lceil c \rceil$
Logical	<b>and, or</b>	
Sums, products	$\Sigma \Pi$	$h \leftarrow \sum_{a \in A} 1/a$

# Граф потоку керування

**Граф потоку керування** (англ. control flow graph, CFG)

— в теорії компіляції — множина всіх можливих шляхів виконання програми, представленіх у вигляді графа.

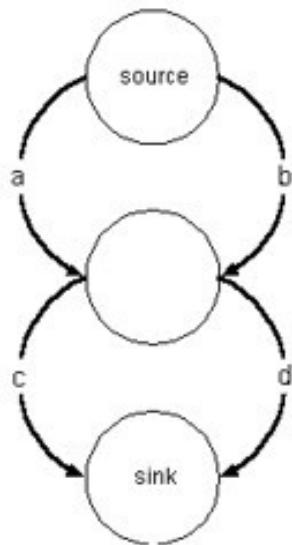


Fig 1. Simplified Control Flowgraph

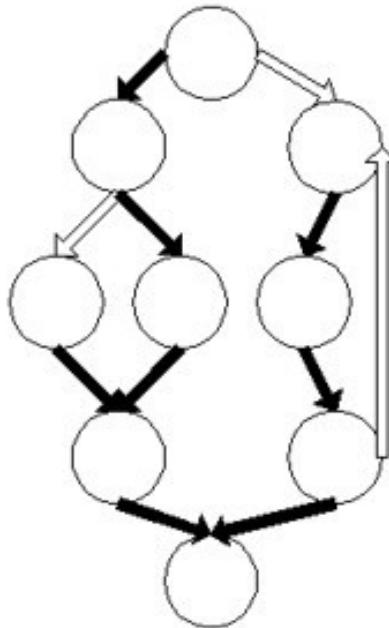


Fig 2. Flowgraph with marked edges

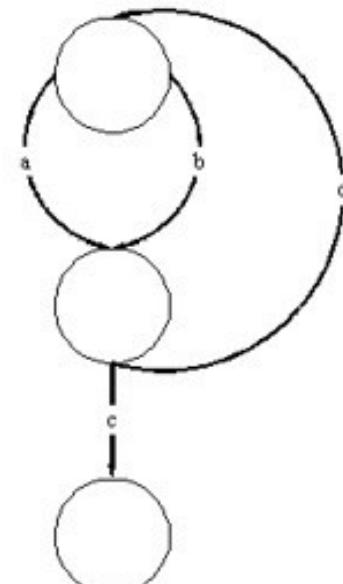


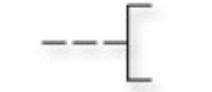
Fig 3. Simplified Flowgraph with Looping

# БЛОК-СХЕМА АЛГОРИТМУ

ГОСТ 19.002-80 та ГОСТ 19.003-80

## Блок-схема

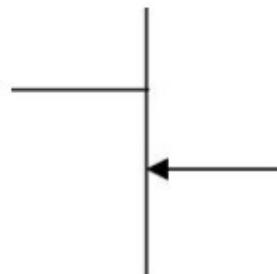
алгоритму – це графічне представлення логічної структури алгоритму, де кожний етап обробки інформації зображається у вигляді геометричних символів (блоків).

Термінатор		Цикл з параметром	
Процес		Межа циклу	
Умова		З'єднувач	
Функція (процедура)		Коментар	
ввід/вивід			

Символи на схемі повинні бути рівномірно розташовані, мати розміри, вибрані в однакових пропорційних співвідношеннях щодо ширини та висоти, достатні для внесення тексту. Якщо обсяг тексту перевищує розмір символа, слід використовувати символ “коментар”.

Потоки даних або потоки управління на схемах зображаються лініями. Напрям потоку зліва направо та зверху вниз вважається стандартним і стрілкою не позначається. В іншому випадку на лінії слід обов'язково вказувати стрілку.

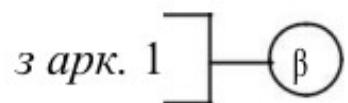
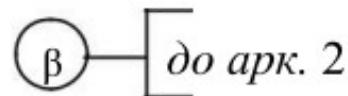
Якщо дві чи більше ліній зливаються в одну, то місце з'єднання повинно бути зміщене.



Лінії мають підходити до символа зліва, або зверху, а виходити справа, або знизу. Лінії повинні бути направлені до центра символа.

При необхідності розриву лінії (для уникнення зайвих перетинів чи надто довгих ліній) використовують символ “з’єднувач”, який у випадку продовження схеми на кількох аркушах подають разом із символом “коментар”.

Зовнішній з’єднувач (на аркуші 1) Внутрішній з’єднувач (на аркуші 2)

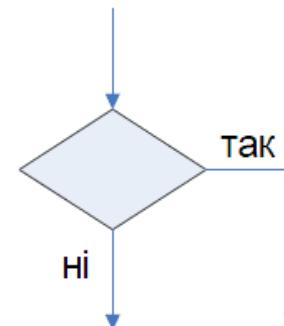


У схемах можна використовувати детальне представлення, яке позначається символом процесу або даних з горизонтальною смужкою у верхній частині.

# Блок-схема алгоритму з використанням лише базових алгоритмічних конструкцій

- Застосовується обмежений набір символів.

- Позначення розміщаються в один стовпчик.
- Однотипна форма умовної конструкції:



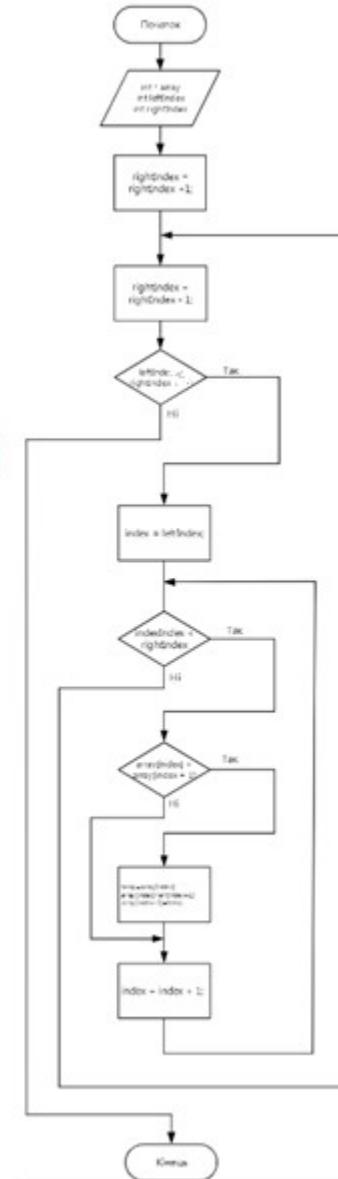
Термінатор	
Процес/ Функція (процедура)	
Умова	
ввід/вивід	
З'єднувач	

```

void bubbleSort(int * array, int leftIndex, int rightIndex){
    int index, temp;

    ++rightIndex;
    while (leftIndex < --rightIndex) {
        for (index = leftIndex; index < rightIndex; ++index) {
            if (array[index] > array[index + 1]) {
                temp = array[index];
                array[index] = array[index + 1];
                array[index + 1] = temp;
            }
        }
    }
}

```

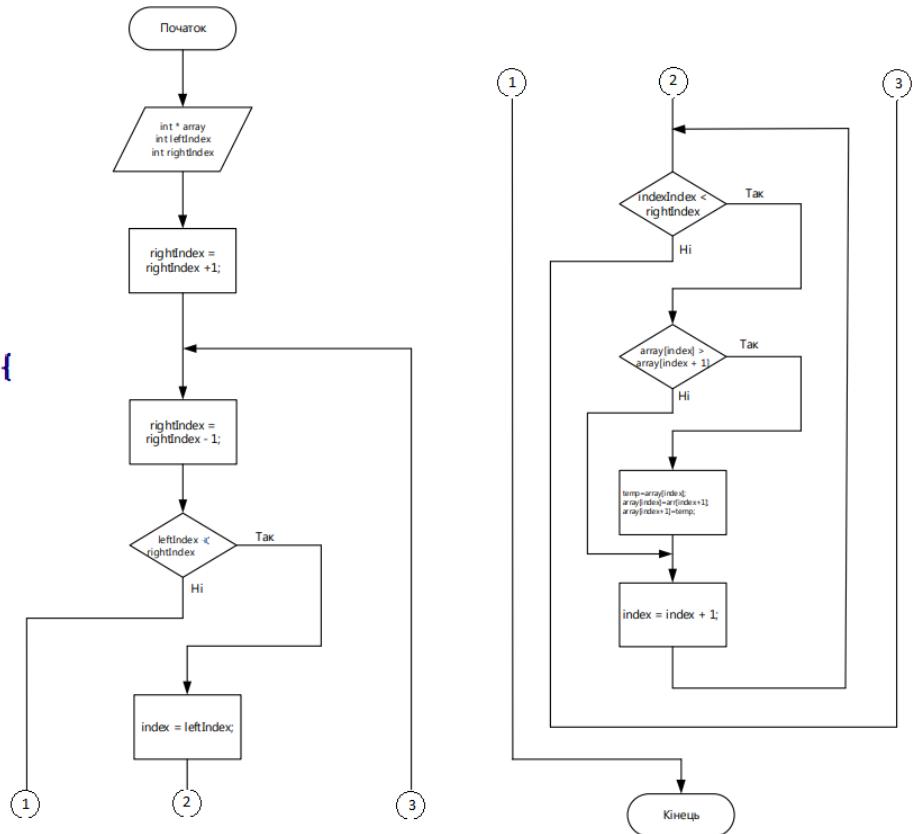


```

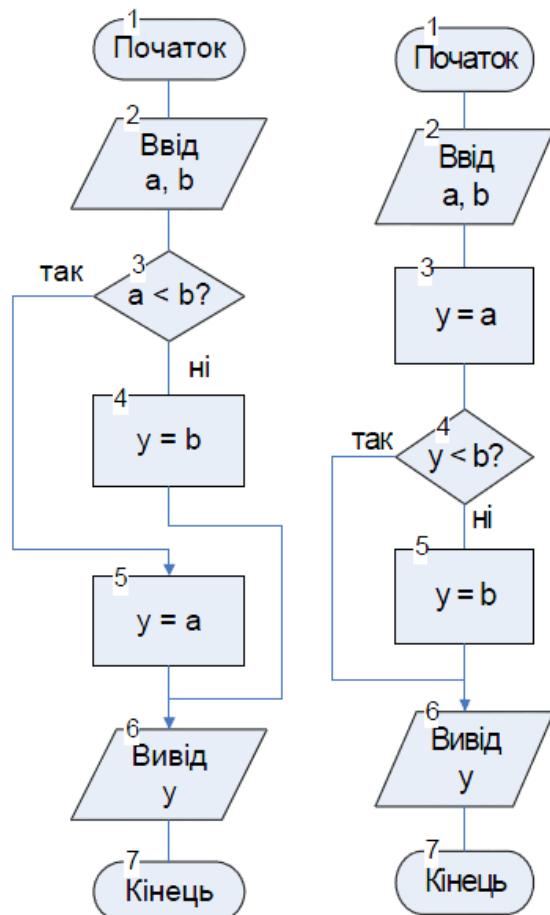
void bubbleSort(int * array, int leftIndex, int rightIndex){
    int index, temp;

    ++rightIndex;
    while (leftIndex < --rightIndex) {
        for (index = leftIndex; index < rightIndex; ++index) {
            if (array[index] > array[index + 1]) {
                temp = array[index];
                array[index] = array[index + 1];
                array[index + 1] = temp;
            }
        }
    }
}

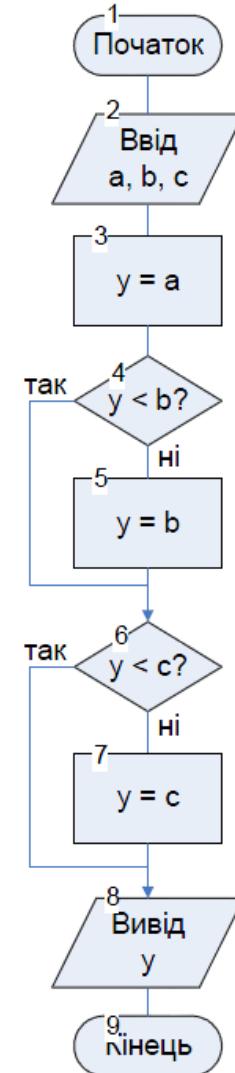
```

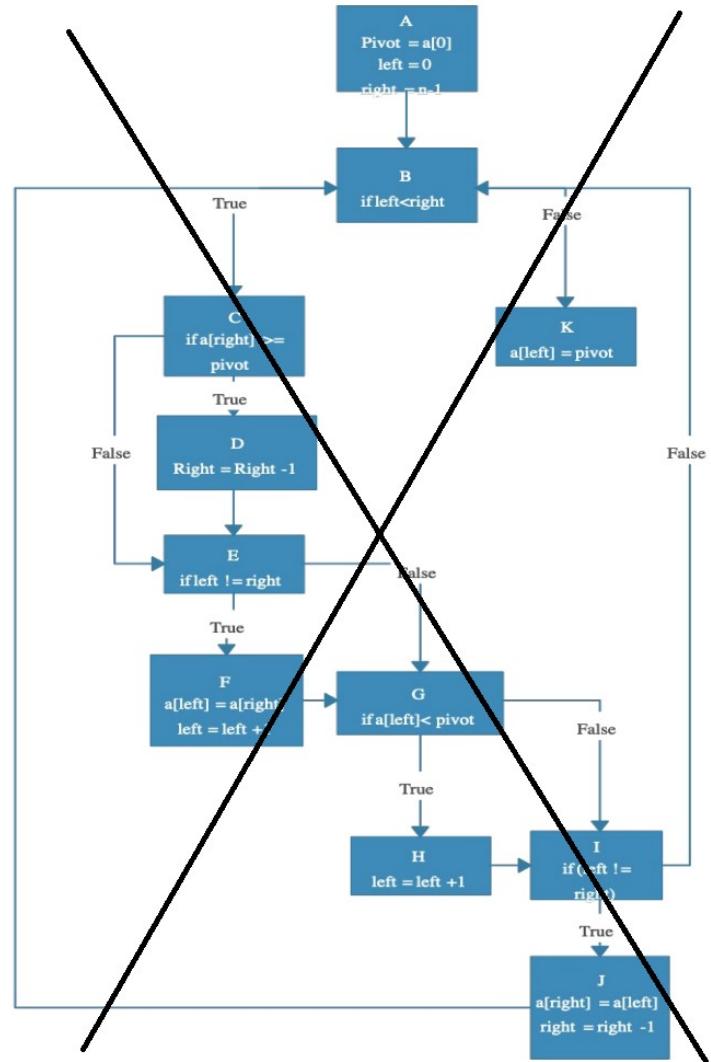
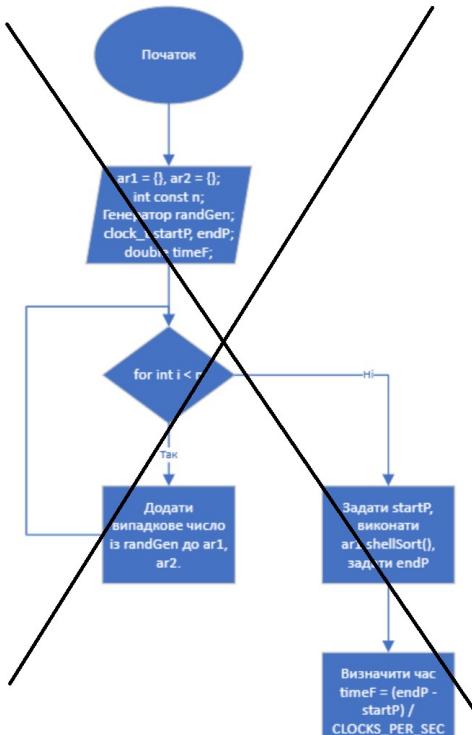
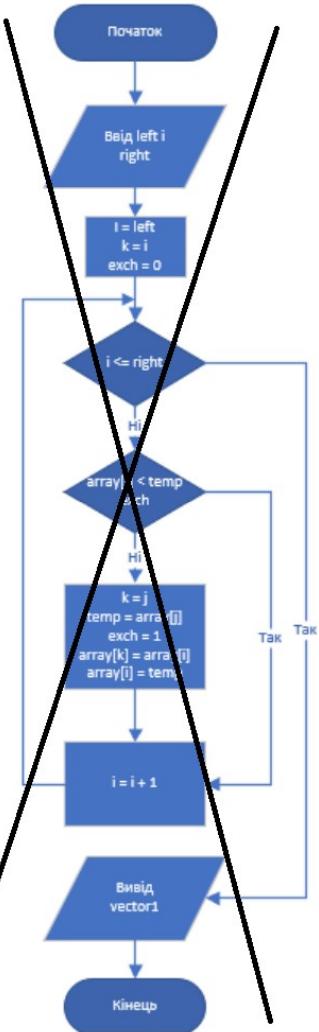


$y = \min(a, b)$



$y = \min(a, b, c)$

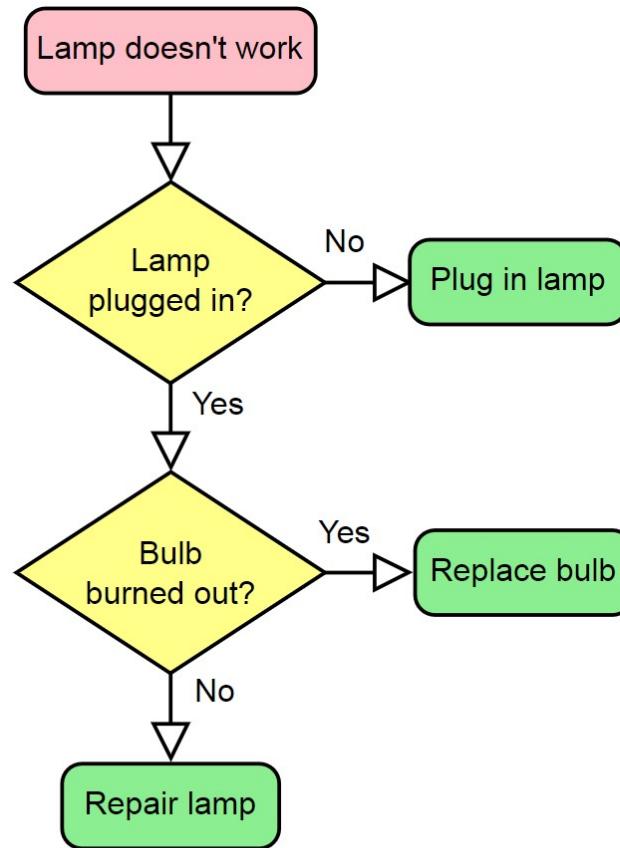




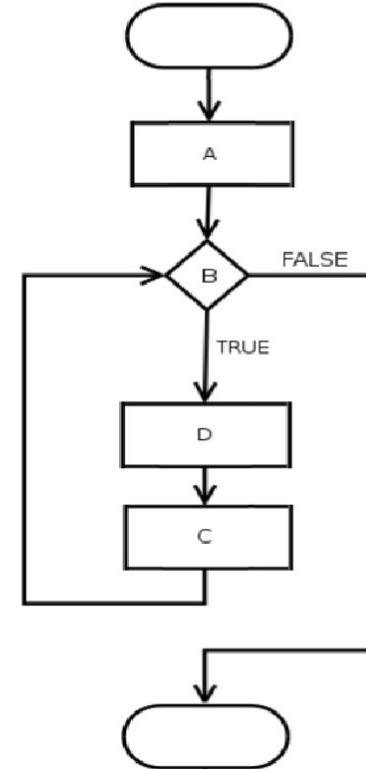
# Контрольне завдання №5

**Скласти блок-схему алгоритму знаходження  
серед двох значень максимального.**

# Flowchart



for(A;B;C)  
D;



# Структурограма. Діаграма Нассі-Шнейдермана.

Спосіб зображення алгоритму за допомогою структурограми (схеми Нассі-Шнейдермана) реалізує в собі вимоги структурного програмування. Він дає змогу зобразити схему передачі управління не за допомогою ліній потоку, а вкладеними структурами.

Деякі із зображуваних графічних символів відповідають зображенню символів на схемах, виконаних згідно зі стандартами Єдиної системи програмної документації (ЕСПД).

Допустимим є використання таких блоків.

1. Блок обробки (обчислень):

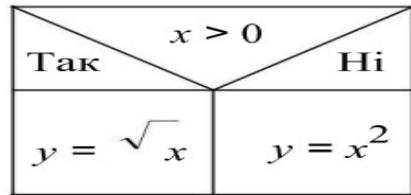
$$y = a + b$$

2. Блок послідовності:

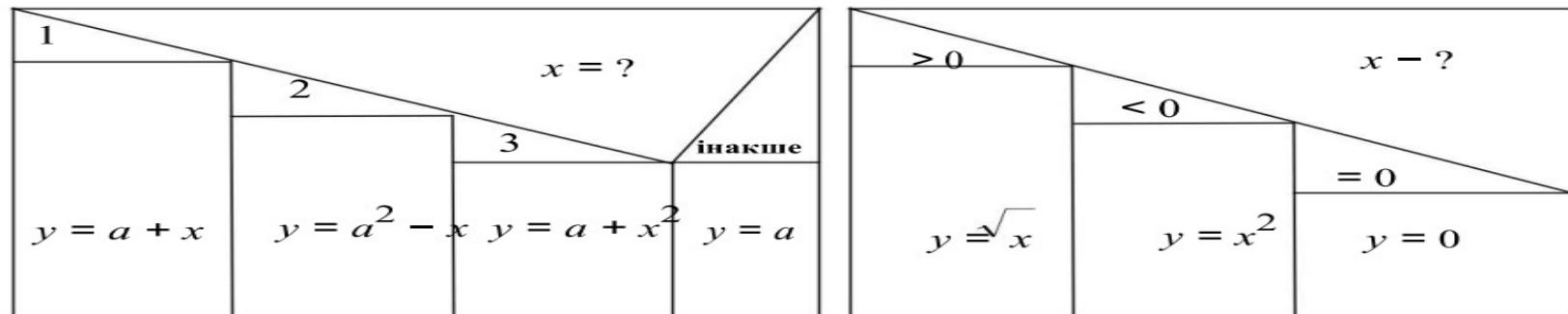
$y = a + b$
$z = y + c$

# Структурограма. Діаграма Нассі-Шнайдермана.

3. Блок для розгалужень:



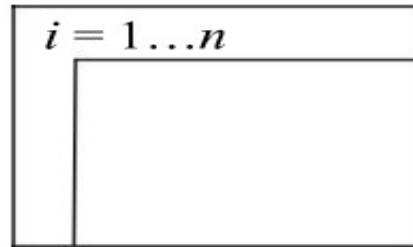
4. Блок варіанту:



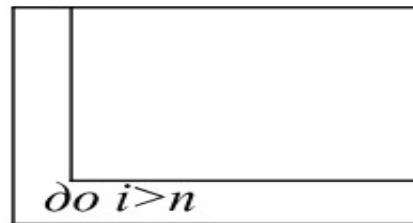
Ті варіанти, які можна точно сформулювати, розташовують зліва. Решту – об’єднують в один, що розташований справа і є виходом за недотриманням умови.

# Структурограма. Діаграма Нассі-Шнайдермана.

5. Блок циклу з параметром або циклу з передумовою:



6. Блок циклу з постумовою:



Кожен блок структурограми має форму прямокутника і може бути вписаний в будь-який інший. Блоки заповнюються формульно-словесно.

# UML

UML (англ. Unified Modeling Language) — уніфікована мова моделювання, використовується переважно з парадигмою об'єктно-орієнтованого програмування.

## Structure Diagrams:

- Class diagram
- Component diagram
- Composite structure diagram
  - Collaboration (UML2.0)
- Deployment diagram
- Object diagram
- Package diagram

## Behavior Diagrams:

- Activity diagram
- State Machine diagram
- Use case diagram

## Interaction Diagrams:

- Collaboration (UML1.x) / Communication diagram (UML2.0)
- Interaction overview diagram (UML2.0)
- Sequence diagram
- UML Timing Diagram (UML2.0)

## Структурні діаграми:

- Класів
- Компонент
- Композитної/складеної структури
  - Кооперації (UML2.0)
- Розгортування
- Об'єктів
- Пакетів

## Діаграми поведінки:

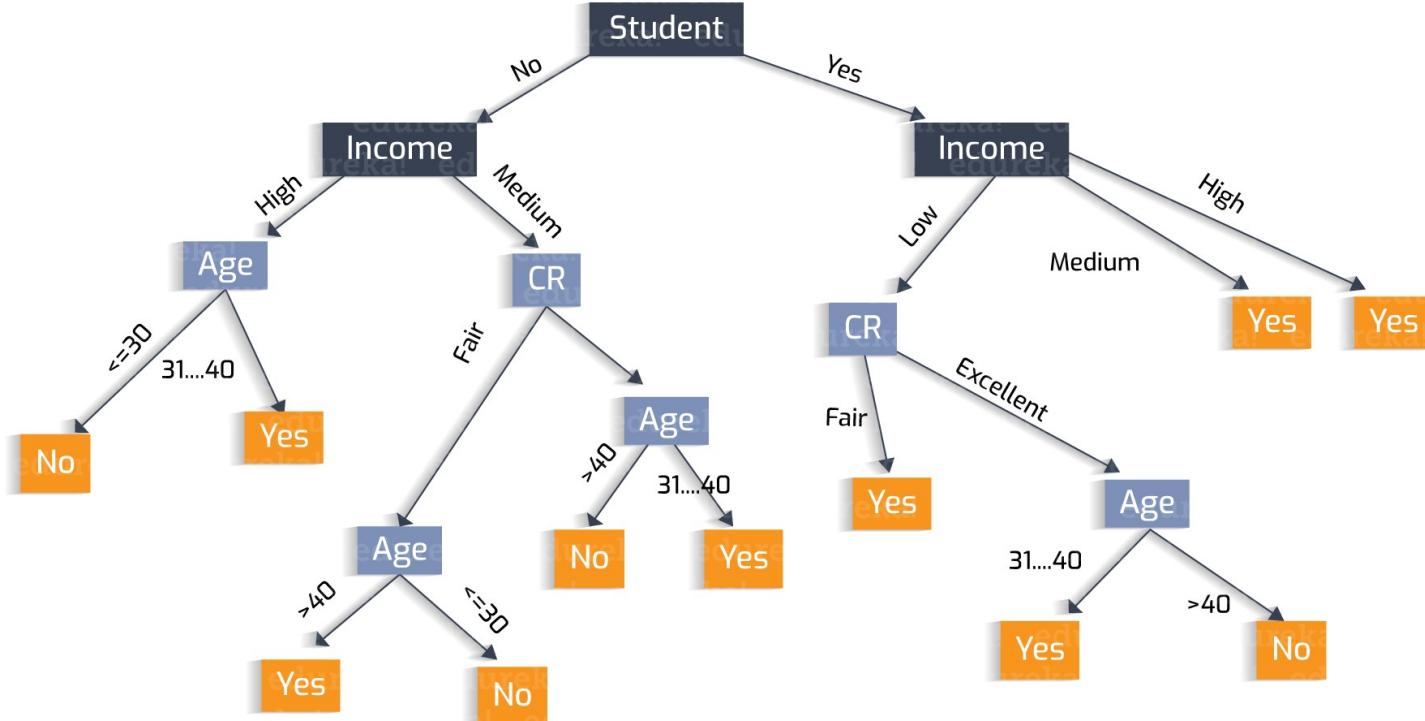
- Діяльності
- Станів
- Прецедентів

## Діаграми взаємодії:

- Кооперації (UML1.x) / Комунікації (UML2.0)
- Огляду взаємодії (UML2.0)
- Послідовності
- Синхронізації (UML2.0)



## 2.2.2.6.1. Дерево розв'язків.



# 2.2.2.6.1. Дерево розв'язків.

