Т. М. Сопронюк

Мовні процесори та формальні мови: від теорії до практики

Міністерство освіти і науки України
Чернівецький національний університет
імені Юрія Федьковича

Т. М. Сопронюк

Мовні процесори та формальні мови: від теорії до практики

Навчальний посібник

Чернівці
Чернівецький національний університет
імені Юрія Федьковича
2025

УДК 004.4 (075.8)
С 646

Друкується за ухвалою Вченої ради
Чернівецького національного університету
імені Юрія Федьковича
(протокол №2 від 24.02.2025 р.)

Рецензенти:

Петрик М. Р., завідувач кафедри програмної інженерії Тернопільського національного технічного університету імені Івана Пулюя, докт. фіз.-мат. наук, професор лауреат Державної премії України в галузі науки і техніки

Махней О. В., кандидат фізико-математичних наук, доцент, доцент кафедри диференціальних рівнянь і прикладної математики Прикарпатського національного університету імені Василя Стефаника

Сопронюк Т.М.

С 646 Мовні процесори та формальні мови: від теорії до практики : навч. посіб. Чернівці : Чернівец. нац. ун-т ім. Ю. Федьковича, 2025. 198 с.
ISBN 978-966-423-942-1

Навчальний посібник присвячений мовним процесорам, формальним мовам та алгоритмам їх аналізу. Розглянуто лексичний, синтаксичний та семантичний аналізи, а також генерація та оптимізація коду. Значну увагу приділено скінченним автоматам, регулярним виразам та граматикам, що є основою для аналізу мов програмування.

Для студентів спеціальностей 113 «Прикладна математика», 122 «Комп'ютерні науки», 124 «Системний аналіз».

УДК 004.4(075.8)

ISBN 978-966-423-942-1

ЗМІСТ

ВСТУП

В епоху цифрових технологій і програмування розробка ефективних мов програмування та їх обробки є надзвичайно важливим завданням. Мовні процесори відіграють ключову роль у створенні програмного забезпечення, адже вони забезпечують переклад вихідного коду, написаного мовою програмування, у форму, зрозумілу комп'ютеру. Без цих процесів неможливо було б реалізовувати сучасні технології, які використовують мільйони розробників по всьому світу.

Цей посібник присвячений фундаментальним принципам побудови мовних процесорів, які охоплюють лексичний, синтаксичний та семантичний аналізи, а також процеси генерації та оптимізації коду. Розглядаються також поняття формальних мов і граматик, регулярних множин та скінченних автоматів, які є основою для створення компіляторів та інтерпретаторів.

Розробка мовних процесорів охоплює різні рівні абстракції, починаючи від аналізу тексту та закінчуючи безпосереднім виконанням команд процесором або віртуальною машиною. Для цього використовуються спеціалізовані алгоритми, які дозволяють розробляти ефективні компілятори, інтерпретатори та засоби обробки програмного коду.

Окрім теоретичного матеріалу, посібник містить численні приклади, алгоритми та контрольні запитання, які допоможуть закріпити отримані знання та навички. Кожна тема супроводжується детальним поясненням ключових понять, що робить посібник доступним для студентів, викладачів та фахівців, які прагнуть поглибити свої знання в галузі мовних процесорів і формальних мов.

Цей посібник стане корисним помічником у вивченні мовних процесорів, формальних мов і скінченніх автоматів, допомагаючи краще зрозуміти їхні принципи та застосування.

Посібник сприятиме глибшому розумінню ключових концепцій формальних мов та автоматного програмування.

ТЕМА І. Розробка мовних процесорів

1 Поняття мовного процесора, типи мовних процесорів

Розробка мовних процесорів є важливим аспектом створення програмного забезпечення для роботи з формальними мовами. Мовний процесор — це програма, яка дозволяє зрозуміти директиви і речення вхідної мови, які використовує програміст, і перетворює їх у машинний код або інші формати.

У цьому розділі розглядаються поняття мовного процесора, його типи, основні фази та взаємодія етапів компіляції [2, 3, 4].

Мовний процесор — це програма, яка перетворює вихідний код, написаний мовою програмування високого рівня, у форму, яку може виконати комп'ютер. Це може включати компіляцію, інтерпретацію або використання гібридних підходів.

Основні типи мовних процесорів [7, 10]:

- **Компілятори:** Програми, які перетворюють вихідний код у машинний код для конкретної платформи. Наприклад, мови C та C++ зазвичай використовують компілятори. Компіляція забезпечує високу продуктивність виконання програм, оскільки кінцевий код вже оптимізовано для виконання апаратурою.

  Основні етапи роботи компілятора:

  – перетворення вихідного коду у проміжне представлення;
  – генерація оптимізованого машинного коду для конкретної архітектури.

  Переваги компіляторів: швидкість виконання, оптимізація під апаратну платформу.

- **Інтерпретатори:** Виконують вихідний код без попередньої компіляції, обробляючи його команду за командою. Приклади мов: Python, JavaScript. Цей підхід спрощує відлагодження та експериментування, але знижує швидкодію.

  Інтерпретатори забезпечують:

  – гнучкість у розробці;
  – можливість виконання програм одразу після внесення змін.

- **Гібридні системи:** Наприклад, Java та C#, які використовують байт-код, що виконується віртуальною машиною (JVM або CLR). Такий підхід поєднує переваги компіляції та інтерпретації, забезпечуючи платформну незалежність і добру продуктивність.

  Основні риси:

  – компіляція у байт-код;
  – виконання байт-коду у середовищі віртуальної машини;

- **Крос-компілятори** створюють код для виконання на іншій платформі. Це спеціальні компілятори, які створюють код для виконання на іншій платформі, відмінній від тієї, на якій виконується сам процес компіляції. Дозволяють створювати програми для платформ, які можуть не мати власних засобів компіляції. Вони часто використовуються у розробці програмного забезпечення для вбудованих систем, мобільних пристроїв та операційних систем.

- **Препроцесори** забезпечують обробку коду перед основною компіляцією. Препроцесори часто використовуються для макропідстановки, умовної компіляції, включення зовнішніх файлів та інших маніпуляцій з вихідним кодом.

Класифікація мовних процесорів

2 Основні фази мовного процесора

Мовний процесор зазвичай працює в кілька етапів [4].

Фази компіляції

1. **Лексичний аналіз (Lexical Analysis):**
   Вхідний код розбивається на окремі лексеми (наприклад, ключові слова, ідентифікатори, оператори). Ця фаза виявляє помилки, пов'язані з недопустимими символами, і створює потік токенів [6].

2. **Побудова таблиць (Symbol Table Construction):**
   Формується таблиця символів, яка містить інформацію про ідентифікатори, їх типи, області видимості, розташування в пам'яті тощо. Ця таблиця використовується у наступних фазах компіляції.

3. **Синтаксичний аналіз (Syntax Analysis):**
   Потік токенів перевіряється на відповідність правилам граматики мови програмування. У цій фазі створюється дерево синтаксичного розбору (синтаксичне дерево), яке представляє структуру програми [14, 15].

4. **Семантичний аналіз (Semantic Analysis):**
   Перевіряється логічна коректність коду. Наприклад, перевіряється сумісність типів даних, чи всі змінні оголошені перед використанням тощо. Помилки на цьому етапі вказують на порушення семантичних правил мови [7].

5. **Генерація проміжного коду (Intermediate Code Generation):**
   Дерево синтаксичного розбору або інші представлення коду переводяться у проміжну форму, яка є абстрактною і не залежить від конкретної архітектури процесора. Це може бути триадресний код або подібна форма [2, 3].

6. **Оптимізація (Optimization):**
   Проміжний код поліпшується для підвищення швидкості виконання чи зменшення використання пам'яті. Наприклад, видаляються непотрібні операції, змінюється порядок виконання команд [12].

7. **Генерація машинного коду (Code Generation):**
   Проміжний код переводиться у машинний код, який може виконуватися на цільовій платформі. Ця фаза включає вибір інструкцій процесора і розподіл регістрів [11].

2.1 Лексичний аналіз

На цьому етапі вхідний текст програми розбивається на лексеми — найменші значущі одиниці, такі як ключові слова, ідентифікатори, оператори та числові значення. Наприклад, у програмі `cost := (price + tax) * 0.98` будуть виділені такі лексеми:

- `cost`, `price`, `tax` — ідентифікатори,
- `:=` — оператор присвоєння,
- `+`, `*` — арифметичні оператори,
- `0.98` — числове значення.

Лексичний аналізатор видасть такі лексеми:

- `<identifier, cost>` - ідентифікатор "cost".
- `<assign, :=>` - оператор присвоєння.
- `<delimiter, (>` - відокремлювач "(".
- `<identifier, price>` - ідентифікатор "price".
- `<operator, +>` - оператор додавання.
- `<identifier, tax>` - ідентифікатор "tax".

- `<delimiter, )>` - відокремлювач ")".
- `<operator, *>` - оператор множення.
- `<number, 0.98>` - числове значення.

Для зручності реалізації в мовах програмування, таких як C++, використовують перелік типів лексем. Наприклад:

```
enum Token { // типи лексем
    identifier,   // Ідентифікатор
    assign,       // Присвоєння
    delimiter,    // Відокремлювач
    operator,     // Оператор
    number        // Число
};
```

2.2 Робота з таблицями

2.2.1 Організувація доступу до даних

Після ідентифікації лексем під час лексичного аналізу важливим етапом є збір інформації про ідентифікатори, функції та інші елементи програми. Це дозволяє організувати ефективний доступ до цих даних на подальших етапах компіляції, таких як синтаксичний аналіз і генерація коду. Інформація про ідентифікатори зберігається в спеціальних таблицях, де кожен запис включає інформацію про змінну або функцію.

Наприклад, у програмі на Pascal усі ідентифікатори повинні бути описані в розділі змінних. Після лексичного аналізу може бути створена таблиця ідентифікаторів, яка виглядатиме так:

Номер	Ідентифікатор	Значення
1	cost	$p1$
2	price	$p2$
3	tax	$p3$

У цій таблиці кожен рядок містить унікальний ідентифікатор змінної та її значення, яке може бути посиланням на пам'ять або конкретне значення, залежно від контексту.

Якщо пізніше в програмі з'являється якийсь новий ідентифікатор, необхідно перевірити, чи не було його раніше. Тому таблиця ідентифікаторів повинна забезпечувати:

- **Швидкий пошук інформації про ідентифікатор:** Для ефективного пошуку часто використовуються спеціальні структури даних, такі як хеш-таблиці.
- **Швидке додавання ідентифікаторів у таблицю:** Це дозволяє додавати нові змінні до таблиці під час компіляції.

Крім таблиць ідентифікаторів, в компіляторах можуть використовуватись інші таблиці:

- **Таблиці підпрограм:** Зберігають інформацію про функції, процедури, їх параметри, типи та інші атрибути.
- **Таблиці даних:** Використовуються для зберігання типів даних, таких як масиви, записи тощо.
- **Таблиці команд:** Містять інформацію про операції та інструкції в програмі.

Ці таблиці далі використовуються на етапі синтаксичного аналізу для перевірки коректності виразів і визначення порядку операцій.

Враховуючи отриману таблицю, можна записати результат лексичного аналізу для виразу (послідовності символів) $cost := (price + tax) * 0.98$ послідовність лексем:

$$< identifier, p1 >< assign >< delimiter, (>$$
$$< identifier, p2 >< operator, + >< identifier, p3 >$$
$$< delimiter,) >< operator, * >< number, q1 >$$

2.2.2 Хеш-таблиці

Для забезпечення швидкого доступу до інформації про ідентифікатори використовуються хеш-таблиці. Це спеціалізовані структури даних, які дозволяють здійснювати пошук, вставку та видалення елементів за час, близький до константи $O(1)$ в середньому випадку.

Хеш-таблиця працює за допомогою хеш-функції, яка перетворює ключ у відповідний індекс таблиці. У випадку колізій (коли два ключі мають однакове хеш-значення) використовуються різні методи їх вирішення:

- **Хешування з ланцюжками:** Колізії вирішуються шляхом зберігання кількох значень у списках, прив'язаних до одного індексу таблиці.
- **Відкрита адресація:** Якщо індекс таблиці вже зайнятий, використовується серія хеш-функцій для пошуку вільної комірки в таблиці.

При хешуванні з відкритою адресацією кожен ключ шукає наступну вільну комірку у таблиці, якщо поточна вже зайнята.

Один з простих прикладів хеш-функції, яка обчислює хеш за допомогою суми ASCII-кодів символів ключа, може виглядати так:

$$h(\text{key}) = \left(\sum \text{ASCII-коди символів ключа}\right) \mod n$$

де n — це розмір хеш-таблиці. Ця функція дозволяє швидко знайти позицію для зберігання значень у таблиці або для перевірки наявності ідентифікатора.

Розглянемо тепер методи вирішення конфліктів, тобто побудови додаткових (вторинних) функцій $h_1, h_2, \ldots, h_m$. Насамперед зазначимо, що значення $h_j(\alpha)$ має відрізнятися від $h_i(\alpha)$ для всіх $i \neq j$. Якщо $h_i(\alpha)$ дає конфлікт, безглуздо пробувати

$h_{i+k}(\alpha)$, якщо $h_{i+k}(\alpha) = h_i(\alpha)$. У більшості випадків бажано, щоб $m = n - 1$, оскільки це дозволяє знайти в таблиці порожню позицію, якщо вона є. У загальному випадку метод вирішення конфліктів впливає на ефективність усієї системи з розподіленою пам'яттю.

Існують різні підходи, засновані на первинній функції $h_0(\alpha)$ та вторинних функціях $h_1, h_2, \ldots, h_m$, наприклад:

- **Лінійне пробування:** поступовий пошук вільного місця за допомогою формули:

$$h_i(\alpha) = (h_0(\alpha) + i) \mod n, \quad i = 1, ..., m.$$

- **Квадратичне пробування:** зміщення збільшується квадратично:

$$h_i(\alpha) = (h_0(\alpha) + c_1 i + c_2 i^2) \mod n, \quad i = 1, ..., m.$$

  де c_1 і c_2 — константи.

- **Подвійне хешування:** використання другої хеш-функції $h2(\alpha)$ для визначення зміщення:

$$h_i(\alpha) = (h1_0(\alpha) + i \cdot h2_0(\alpha)) \mod n, \quad i = 1, ..., m.$$

Ці підходи забезпечують різну ефективність залежно від способу розподілу ключів у хеш-таблиці.

Переваги хеш-таблиць

- **Швидкий доступ:** Хеш-таблиці дозволяють здійснювати пошук за ключем за час $O(1)$ в середньому випадку.

- **Ефективність при великих обсягах даних:** Хеш-таблиці дуже ефективні при роботі з великими наборами даних, оскільки вони дозволяють мінімізувати час пошуку.

- **Універсальність:** Хеш-таблиці можуть бути використані не тільки для зберігання ідентифікаторів, а й для зберігання будь-якої іншої інформації, де потрібен швидкий доступ.

Приклад використання хеш-таблиці

Нехай для побудови хеш-таблиці для виразу cost := (tax + price) * 0.98 використовуємо хеш-функції:

$$h_0(\alpha) = \text{CODE}(\alpha) \mod 6$$

$$h_i(\alpha) = (\text{CODE}(\alpha) + i + 2) \mod 6, \quad i = 1, \ldots, 5$$

Коди літер починаються з одиниці, тобто: $a = 1, b = 2, \ldots$

Кроки побудови таблиці

1. **Обчислення хеш-значення для кожного ідентифікатора:**

- Для ідентифікатора `cost` обчислюємо:

$$\text{CODE}(\text{cost}) = 3 + 15 + 19 + 20 = 57$$

$$h_0(\text{cost}) = 57 \mod 6 = 3$$

Отже, `cost` розміщується за адресою 3.

- Для ідентифікатора `tax` обчислюємо:

$$\text{CODE}(\text{tax}) = 20 + 1 + 24 = 45$$

$$h_0(\text{tax}) = 45 \mod 6 = 3$$

Оскільки за адресою 3 вже є `cost`, застосовуємо хеш-функцію h_1:

$$h_1(\text{tax}) = (45 + 1 + 2) \mod 6 = 48 \mod 6 = 0$$

Отже, `tax` розміщується за адресою 0.

- Для ідентифікатора `price` обчислюємо:

$$\text{CODE}(\text{price}) = 16 + 18 + 9 + 3 + 5 = 51$$

$$h_0(\text{price}) = 51 \mod 6 = 3$$

Оскільки за адресою 3 вже є `cost`, застосовуємо хеш-функцію h_1:

$$h_1(\text{price}) = (51 + 1 + 2) \mod 6 = 54 \mod 6 = 0$$

Оскільки за адресою 0 вже є `tax`, застосовуємо хеш-функцію h_2:

$$h_2(\text{price}) = (51 + 2 + 2) \mod 6 = 55 \mod 6 = 1$$

Отже, `price` розміщується за адресою 1.

2. **Хеш-таблиця після додавання ідентифікаторів:**

Індекс	Ідентифікатор
0	`tax`
1	`price`
2	-
3	`cost`
4	-
5	-

Пояснення:

1. **Ідентифікатор** `cost`.

Спочатку для `cost` обчислюється хеш-функція h_0, і `cost` потрапляє в комірку з індексом 3.

2. **Ідентифікатор** `tax`.

Для `tax` спочатку розраховуємо хеш-функцію h_0, але оскільки комірка з індексом 3 вже зайнята, використовуємо h_1, що дає результат 0, отже, `tax` потрапляє в комірку з індексом 0.

3. **Ідентифікатор** `price`.

Для `price` спочатку обчислюємо хеш-функцію h_0, і вона потрапляє в комірку з індексом 3, але вона вже зайнята, тому

використовуємо h_1, потім h_2, і `price` потрапляє в комірку з індексом 1.

Отже, в таблиці для виразу cost := (tax + price) * 0.98 після додавання всіх ідентифікаторів буде заповнено три комірки: 0, 1 і 3, відповідно для `tax`, `price` і `cost`.

Для кожного з них застосовувалася відповідна хеш-функція, і, у разі необхідності, використано методи розв'язання колізій.

Таблиця використовує хеш-функцію для обчислення індексів для кожного ідентифікатора. Коли програма запитує значення для певного ідентифікатора, хеш-функція дозволяє швидко отримати необхідну інформацію, що значно прискорює компіляцію та виконання програм.

Висновки

Як бачімо, робота з таблицями ідентифікаторів та іншими таблицями в компіляторах є важливою частиною процесу аналізу програми. Хеш-таблиці - ефективний інструмент для забезпечення швидкого доступу до інформації про змінні, функції та інші елементи програми, що дозволяє значно підвищити ефективність компіляції та зменшити час на виконання програми.

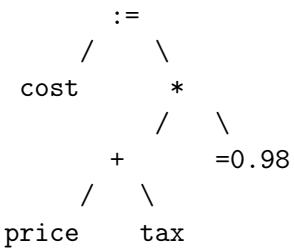
2.3 Синтаксичний аналіз

Синтаксичний аналіз являє собою процедуру, в рамках якої досліджується послідовність лексем з метою встановлення їх відповідності синтаксису мови програмування. Згідно зі структурою даної мови, окремі лексеми об'єднуються у форми операторів. Наступним кроком є поєднання цих операторів у більш складні блоки, які, у свою чергу, формують основу програми. Визначення таких правил поєднання становить основу синтаксису мови програмування.

2.3.1 Синтаксична структура оператора присвоєння

Розуміння синтаксичної структури оператора необхідне також для процесу генерації коду. Наприклад, для того, щоб правильно виконати оператор `cost := (price + tax) * 0.98`, спочатку виконується додавання в дужках, потім множення і присвоєння.

Синтаксичний аналіз будує дерево синтаксису (або інші структури), перевіряючи відповідність коду граматиці мови. Наприклад, для виразу `cost := (price + tax) * 0.98` формується дерево операцій, яке задає порядок виконання дій [2, 4]:



У цьому дереві кореневий вузол відповідає оператору присвоєння `:=`, а піддерева представляють операції множення та додавання.

Для реалізації пріоритетів операцій застосовуються правила пріоритету, які можна задати за допомогою таблиць або вбудованих алгоритмів, таких як метод рекурсивного спуску. Наприклад:

- Дужки `()` мають найвищий пріоритет.
- Множення `*` виконується перед додаванням `+`.
- Оператори виконуються зліва направо, якщо інше не задано.

2.3.2 Пріоритет операторів

У мові програмування символи мають визначений пріоритет для виконання операцій.

Ось основні правила для пріоритетів:

- 0 – “:=” (оператор присвоєння)
- 1 – “+” (оператор додавання)
- 2 – “*” (оператор множення)
- “(” – збільшує пріоритет операції на 3
- “)” – зменшує пріоритет операції на 3

Приклад 1:

Розглянемо вираз:

```
y := (x1 + x2) * 0.5
```

Пріоритети операцій у цьому виразі виглядають так:

$$0 \quad 4 \quad 2,$$

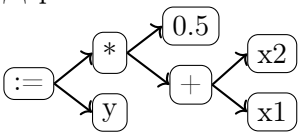
де

- 0 – оператор присвоєння “:=”
- 4 – додавання “+” (3+1)
- 2 – оператор множення “*”

Послідовність операцій:

- спочатку виконується множення “*”
- потім виконується додавання “+”
- останнім виконується присвоєння “:=”

Дерево синтаксичного розбору:



Приклад 2:

Розглянемо вираз:

```
a1 := (a2 + a3 * 5) * 3.28 + a4 * 7.2
```

Пріоритети операцій у цьому виразі:

0 4 5 2 1 2

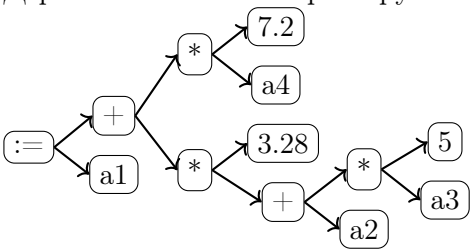
де

- 0 – оператор присвоєння “:=”
- 4 – оператор додавання “+”
- 5 – оператор множення “*” (a3 * 5)
- 2 – оператор множення “*”
- 1 – оператор додавання “+”
- 2 – оператор множення “*” (a4 * 7.2)

Послідовність виконання операцій:

- спочатку виконується множення “*” (a3 * 5)
- потім додавання “+” (a2 + a3 * 5)
- після цього множення “*” (* 3.28)
- потім множення “*” (a4 * 7.2)
- потім додавання “+”
- і нарешті присвоєння “:=”

Дерево синтаксичного розбору:



2.4 Семантичний аналіз

Семантичний аналіз є критичним етапом у процесі компіляції, що здійснюється після лексичного та синтаксичного аналізу. Його метою є перевірка коректності використання змінних, операторів та інших елементів мови. Це гарантує, що програма відповідає певним правилам мови програмування.

Семантичний аналіз перевіряє, чи дотримано такі умови:

- **Оголошення змінних:** Перед використанням змінних необхідно перевірити, чи були вони оголошені. Наприклад, спроба використати змінну, яка не була оголошена, призведе до помилки.
- **Типи даних:** Перевіряється правильність використання типів даних, наприклад, чи не намагається програма додавати змінні різних несумісних типів (наприклад, рядок та число).
- **Правила видимості змінних:** Перевіряється, чи знаходиться змінна в межах свого області видимості (scope). Наприклад, чи можна використовувати локальну змінну поза її області видимості.
- **Правильність операторів:** Перевірка, чи правильно використовуються оператори в контексті їх операндів. Наприклад, перевірка того, чи не спробує користувач поділити число на нуль.
- **Правила перевантаження операторів:** Якщо мова дозволяє перевантажувати оператори, перевіряється, чи правильно використовуються перевантажені операції для даних типів.

Приклад 1: Оголошення змінних

Розглянемо програму:

```
int a = 5;
// помилка: змінна b не була оголошена
b = 10;
```

Семантичний аналіз на цьому етапі повідомить про помилку, оскільки змінна `b` була використана без попереднього оголошення.

Приклад 2: Перевірка типів

Розглянемо такий фрагмент коду:

```
int a = 5;
float b = 3.14;
// помилка: змішування цілих чисел
// і чисел з плаваючою комою
a = a + b;
```

В даному прикладі спроба додати ціле число та число з плаваючою комою може бути дозволена у деяких мовах, однак в інших вона викличе помилку типу. Семантичний аналіз перевірить відповідність типів операндів.

Приклад 3: Перевірка видимості змінних

Вираз:

```
if (x > 10) {
    int y = x + 5;
}
// помилка: змінна y невидима за межами блока
y = y + 1;
```

Семантичний аналіз на цьому етапі виявить, що змінна `y` має локальну область видимості, яка обмежена блоком if, і не може бути використана поза цим блоком.

Семантичний аналіз виявить спробу виконання операції ділення на нуль, що недопустиме у багатьох мовах програмування.

Результати семантичного аналізу

Після виконання семантичного аналізу компілятор формує звіт про помилки. Це можуть бути як синтаксичні помилки (наприклад, помилки типів або порушення обсягу змінних), так і логічні помилки, які можуть виникнути при спробі виконати недопустиму операцію.

Успішне завершення семантичного аналізу дозволяє перейти до наступного етапу — генерації проміжного коду, де програма буде представлена у більш абстрактній формі, готовій до подальшої оптимізації та виконання.

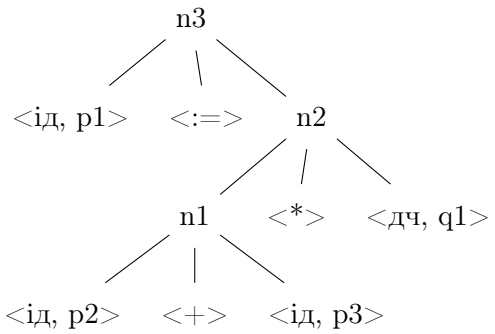
2.5 Генерація проміжного коду

Результат синтаксичного аналізу, що являє собою деревоподібну структуру, використовується для трансформації первинної програми у нову форму. Ця нова форма може бути безпосередньо машинним кодом, проте найчастіше вона являє собою проміжний код.

Генерація проміжного коду — це етап компіляції, що здійснюється після синтаксичного аналізу, коли програма перетворюється в проміжний вигляд, зручний для подальшої оптимізації та генерації машинного коду.

Розглянемо вираз `cost := (price + tax) * 0.98` [4].

Позначимо через n1, n2, n3 внутрішні вершини нашого дерева.



Нащадки кожної вершини задають аргументи, над якими треба проводити дії, або визначають саму дію.

2.5.1 Команди проміжного коду

Проміжний код складається з найпростіших команд, які включають коди операцій та невелику кількість операндів. Зазвичай, цей код представлений у вигляді, близькому до мови асемблерного типу, що робить його зручним для подальшої оптимізації та перетворення у машинний код. Такий підхід дозволяє спростити процес компіляції та забезпечує більшу гнучкість при роботі з різними архітектурами [2, 3, 4].

Для ілюстрації розглянемо, як написати проміжний код для обчислення значень виразу, який містить знаки + та ∗.

Уведемо позначення [4]:

R(m) - вміст комірки m.

= m − числове значення m.

Нехай у нас є машина, яка має один робочий регістр - суматор або регістр результату.

Маємо ще набір команд, які виглядають так:

Команда	Значення
Load m	R(m) → суматор
Add m	суматор +R(m) → суматор
Мру m	суматор ∗R(m) → суматор
Store m	суматор → R(m)
Load = m	m → суматор
Add = m	суматор +m → суматор
Мру = m	суматор ∗m → суматор

За допомогою дерева, що отримується в процесі синтаксичного аналізу й інформації, яка зберігається в таблиці ідентифікаторів, і чисел можна побудувати проміжний код. На практиці побудова дерева і генерація коду часто відбувається

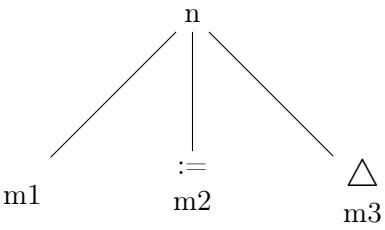
одночасно, але для розуміння будемо вважати, що спочатку будуємо дерево, а потім код.

2.5.2 Побудова проміжного коду

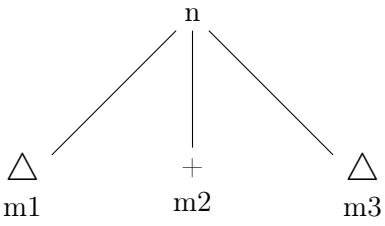
Розглянемо спосіб побудови коду, який має назву: «Синтаксично-керований метод перегляду». В цьому методі кожній вершині n дерева відповідають деякі команди проміжного коду. Код для вершини n отримується об'єднанням кодів прямих нащадків вершини n і деяких визначених команд.

У процесі генерації проміжного коду потрібно обробляти різні типи вершин.

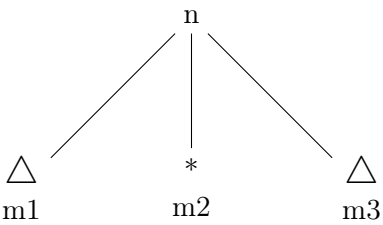
Зауважимо, що для випадків типу $cost := (price + tax) * 0.98$ існує тільки три типи вершини дерева: додавання, множення та присвоєння [2, 3, 4].



а) Вершина типу присвоєння



б) Вершина типу додавання



в) Вершина типу множення

Уведемо позначення [4]:

$C(n)$ - частина проміжного коду, що відповідає вершині n.

$l(n)$ - рівень вершини n.

$$l(n) = \begin{cases} 0, \text{ якщо немає нащадків} \\ \max_{1 \le i \le k} l(n_i) + 1, n_i - \text{ нащадки вершини } n \end{cases}$$

Визначимо код для кожного типу вершини:
1)

- Якщо n – лист, який відповідає ідентифікатору, то $C(n)$ – це ім'я змінної, яке відповідає ідентифікатору (наприклад, cost).
- Якщо n – лист, який відповідає дійсному числу, то $C(n)$ – дійсне число (наприклад, $= 0.98$).
- Якщо n – лист, який відповідає лексемам $< + >$, $< * >$, $<:=>$, то їм не відповідає ніякий код.

2) Якщо n – вершина типу а), m_1, m_2, m_3 – нащадки, тоді код цієї вершини:

$$C(n) \to \text{Load } C(m_3)$$

$$\text{Store } C(m_1)$$

3) Якщо n – вершина типу б), m_1, m_2, m_3 – нащадки, то вершині відповідає такий код:

$$C(n) \to C(m_3)$$

Store $\$l(n)$

Load $C(m_1)$

Add $\$l(n)$

4) Якщо n – вершина типу в), m_1, m_2, m_3 – нащадки, тоді код цієї вершини:

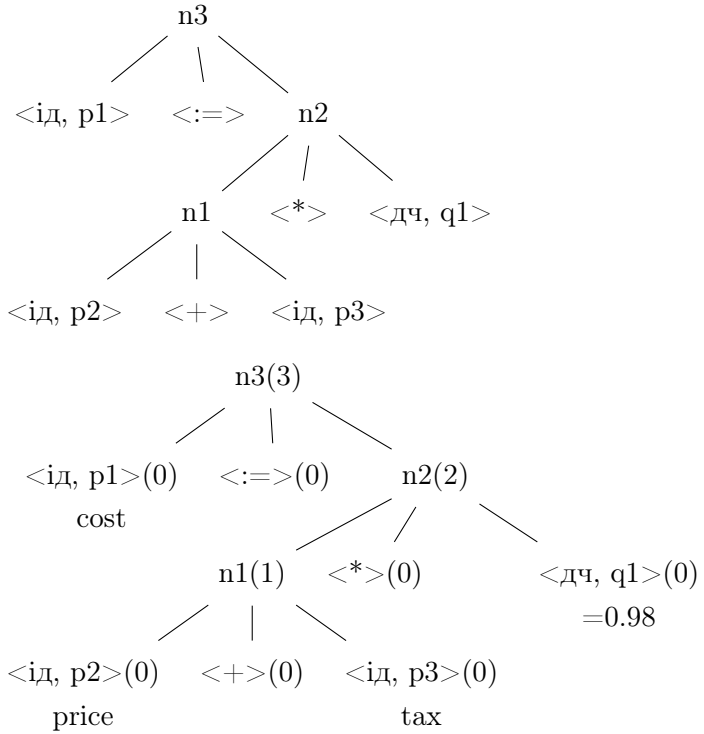
$$C(n) \to C(m_3)$$

Store $\$l(n)$

Load $C(m_1)$

Мру $\$l(n)$

Доповнимо дерево рівнями вершин (значення в дужках біля імені вершини) і кодами листів:



Для виразу cost := (price + tax) ∗ 0.98 обчислимо коди вершин для відповідного дерева:

Код $C(n_1)$	Код $C(n_2)$	Код $C(n_3)$
tax	= 0.98	Load $C(n_2)$
Store $1	Store $2	Store cost
Load price	Load $C(n_1)$	
Add $1	Мру $2	

Підставимо код $C(n1)$ в код $C(n2)$, після цього - код $C(n2)$ в код $C(n3)$. Отримаємо код, який відповідає вершині $C(n3)$:

Код $C(n_1)$	Код $C(n_2)$	Код $C(n_3)$
tax	Load = 0.98	Load = 0.98
Store $1	Store $2	Store $2
Load price	Load tax	Load tax
Add $1	Store $1	Store $1
	Load price	Load price
	Add $1	Add $1
	Мру $2	Мру $2
		Store cost

Отже код, який відповідає кореню дерева, має вигляд [4]:

Load =0.98

Store $2

Load tax

Store $1

Load price

Add $1

Мру $2

Store cost

Цей код відповідає оператору присвоєння, cost := (price + tax) ∗ 0.98.

2.6 Оптимізація проміжного коду

Після генерації проміжного коду часто застосовуються методи оптимізації, які дозволяють покращити швидкодію та зменшити розмір програми [1].

Під оптимізацією коду мають на увазі спробу зробити програму або більш ефективною (швидкодіючою) або більш компактною. Можна спробувати перетворити програму в будь-який спосіб, але при перетвореннях вона повинна зберігати свою здатність виконувати потрібні дії.

Оптимізацію можна реалізувати на різних стадіях процесу:

- Працювати з первинною програмою, вносити в неї поліпшення ще на ранніх етапах.
- Оптимізувати структури, які утворюються після проведення синтаксичного аналізу, для поліпшення їх ефективності та продуктивності.
- Працювати з проміжним кодом, вдосконалюючи його перед перетворенням у машинний код або виконанням.

Основні стратегії оптимізації:

- **Усунення зайвих операцій**: Якщо операція не змінює результат (наприклад, додавання нуля або множення на одиницю), її можна вилучити.
- **Об'єднання операцій**: Можна об'єднувати кілька послідовних операцій (наприклад, додавання та множення), щоб зменшити кількість команд.
- **Перерахування проміжних результатів**: Якщо одне й те саме значення обчислюється кілька разів, його можна зберігати в тимчасових змінних.

Розглянемо правила оптимізації проміжного коду [4]:

1) Операція «+» є комутативною в тому випадку, коли на Add b не передавалось управління.

$$\left.\begin{array}{c}\text{Load } a\\ \text{Add b}\end{array}\right\} \Leftrightarrow \left\{\begin{array}{l}\text{Load } b\\ \text{Add a}\end{array}\right.$$

2) Операція «*» є комутативною

$$\left.\begin{array}{c}\text{Load } a\\ \text{Мру } b\end{array}\right\} \Leftrightarrow \left\{\begin{array}{l}\text{Load } b\\ \text{Мру } a\end{array}\right.$$

3) Послідовність команд вигляду

$$\begin{array}{c}\text{Store } a\\ \text{Load } a\end{array}$$

можна вилучити, при умові, що надалі комірка a не буде використовуватись, або не буде заповнена безпосередньо перед використанням.

4) Послідовність команд вигляду

$$\begin{array}{c}\text{Load } a\\ \text{Store } b\end{array}$$

можна вилучити, при умові, що за нею йде інший оператор Load a і немає переходу до Store b, а наступні входження b будуть замінені на a до того моменту, поки знову не з'явиться оператор Store b.

Оптимізуємо отриманий нами проміжний код, використовуючи правила оптимізації.

Load =0.98	Load =0.98		
Store $2	Store $2	**Load =0.98**	
Load tax	Load tax	**Store $2**	Load tax
Store $1	**Store $1**	Load tax	Add price
Load price	**Load $1**	Add price	Mpy =0.98
Add $1	Add price	**Mpy $2**	Store cost
Mpy $2	Mpy $2	Store cost	
Store cost	Store cost		

Взагалі кажучи, перетворень може бути багато і застосовувати їх можна в різних комбінаціях.

Для виразу y := (x1 + x2) * 0.5 процес оптимізації відбувається так:

Load =0.5		Load =0.5				
Store $2		Store $2		**Load =0.5**		
Load x2		Load x2		**Store $2**		Load x2
Store $1	$\overset{1}{\Rightarrow}$	**Store** $1	$\overset{3}{\Rightarrow}$	Load x2	$\overset{4}{\Rightarrow}$	Add x1
Load x1		**Load $1**		Add x1		Mpy =0.5
Add $1		Add x1		**Mpy $2**		Store y
Mpy $2		Mpy $2		Store y		
Store y		Store y				

Над стрілками підписано номера правил оптимізації. Жирним відмічені фрагменти коду, які беруть участь в оптимізації.

Зауваження

Програми складаються не тільки з операторів присвоєння. Всі інші конструкції мови в подібний спосіб перетворюються у проміжний код. Часто синтаксичний аналіз (побудова дерева) і генерація коду відбуваються паралельно.

2.7 Генерація машинного коду

Асемблювання є останнім кроком, на якому проміжний код перетворюється у машинний. Цей процес полягає в транс-

формації проміжного коду в команди, які можуть бути виконані процесором або віртуальною машиною.

При створенні машинного коду необхідно виконати такі завдання:

1) розташування даних у пам'яті;

2) визначення методу доступу до цих даних;

3) вибір регістрів для проведення обчислень.

2.8 Аналіз помилок

На більшості етапів роботи мовного процесора відбувається виявлення помилок. Помилки можуть бути різноманітними, наприклад:

- помилки, коли вхідні символи не формують жодної лексеми;
- помилки, коли вхідна програма може бути розділена на лексеми, але їхня послідовність не відповідає синтаксичним правилам;
- помилки, коли синтаксична структура програми правильна, але генерація коду неможлива. Це трапляється, наприклад, якщо змінна не була оголошена [9].

Для виявлення та обробки помилок використовуються механізми аналізу помилок, такі як таблиці символів, стекові механізми та евристичні методи виправлення [7].

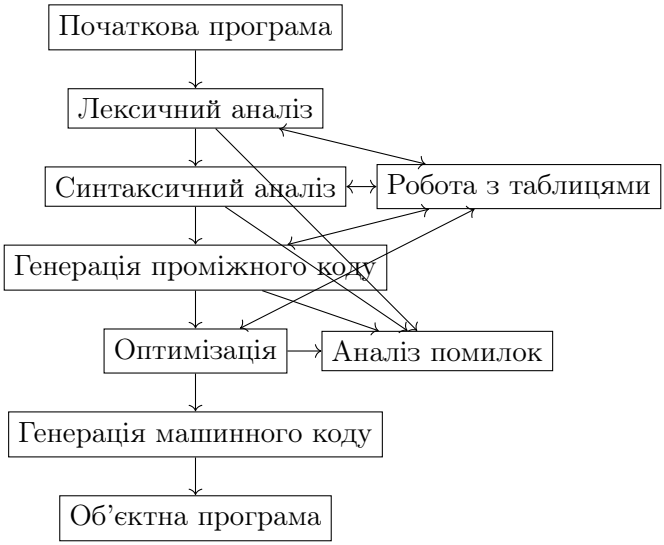
Після виявлення помилки компілятор може:

- надати повідомлення про помилку та припинити свою роботу;
- видати повідомлення, ігнорувати помилку, дати поради щодо її корекції та продовжити обробку для виявлення додаткових помилок.

На етапах лексичного, синтаксичного та семантичного аналізу початкова програма розбивається на складові частини. Етапи генерації проміжного коду, оптимізації та створення машинного коду відповідають за формування об'єктної програми. Усі фази компіляції використовують таблиці символів і працюють із помилками.

Залежно від складності мовного процесора, деякі з цих етапів можуть бути пропущені або об'єднані у єдиний модуль.

Можливу модель компілятора можна зобразити у вигляді такої схеми:



2.9 Взаємодія етапів компіляції

У різних компіляторах етапи компіляції можуть виконуватися паралельно або послідовно.

Наприклад:

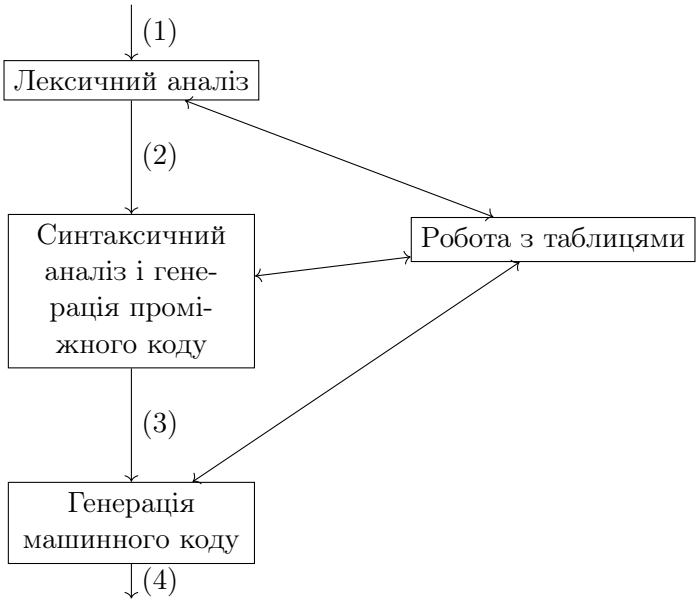
- **Однопрохідний компілятор**: усі етапи виконуються паралельно, з мінімальним використанням пам'яті.

- **Багатопрохідний компілятор**: програма проходить через кілька етапів, результати кожного з яких зберігаються у файлах.

  Цей підхід дозволяє зменшити використання оперативної пам'яті, але збільшує час виконання через багаторазове читання та запис файлів.

Приклад багатопрохідної компіляції

Розглянемо трипрохідний компілятор, в якому:
(1) – вихідний текст програми,
(2) – набір лексем,
(3) – код у проміжному поданні,
(4) – результат у вигляді машинного коду.



Проходи компілятора відбуваються наступним чином:

1. Початкова програма перетворюється на послідовність лексем.

2. Лексеми перетворюються на проміжний код.

3. Проміжний код перетворюється на машинний код.

Висновки

Отже, розробка мовних процесорів — це складний багатоступеневий процес, що включає аналіз, оптимізацію та генерацію коду. Кожен етап має важливе значення для забезпечення коректності та ефективності програмного забезпечення [8, 4].

Контрольні запитання та завдання

1. Поняття мовного процесора, типи мовних процесорів

1. Що таке мовний процесор? Які його основні завдання?

2. Які типи мовних процесорів ви знаєте? Поясніть їх відмінності.

3. У чому переваги компіляторів перед інтерпретаторами?

4. Як працюють гібридні системи мовних процесорів? Наведіть приклади.

5. Що таке крос-компілятори та препроцесори? Для чого вони використовуються?

2. Основні фази мовного процесора

1. Які фази компіляції виконує мовний процесор?

2. Що таке лексичний аналіз? Які його результати?

3. Як будується таблиця символів? Яка її роль у компіляції?

4. Що таке синтаксичний аналіз? Які структури він створює?

5. Які основні завдання виконує семантичний аналіз?

6. Що таке генерація проміжного коду? Яке його призначення?

7. Як оптимізується проміжний код? Які основні правила оптимізації?

8. Як виконується генерація машинного коду? Які задачі розв'язуються на цьому етапі?

3. Лексичний аналіз

1. Що таке лексеми? Наведіть приклад розбиття програми на лексеми.

2. Як лексичний аналізатор представляє лексеми для подальшої обробки?

3. Які типи помилок може виявити лексичний аналіз?

4. Робота з таблицями

1. Для чого використовуються таблиці символів у компіляторах?

2. Які основні види таблиць символів застосовуються у мовних процесорах?

3. Як використовуються хеш-таблиці для зберігання інформації про ідентифікатори?

4. Які методи використовуються для розв'язання колізій у хеш-таблицях?

5. Синтаксичний аналіз

1. Які основні завдання синтаксичного аналізу?
2. Що таке дерево синтаксичного розбору? Як його побудувати для заданого виразу?
3. Що таке рівень вершини?
4. Як використовуються тимчасові змінні в проміжному коді?
5. Як враховуються пріоритети операторів під час синтаксичного аналізу?

6. Генерація та оптимізація проміжного коду

1. Що таке проміжний код? Як його формують?
2. Які основні команди використовуються в проміжному коді?
3. Які чотири правила оптимізації проміжного коду ви знаєте? Поясніть їх на прикладах.

7. Генерація машинного коду

1. Що таке генерація машинного коду? Як вона пов'язана з проміжним кодом?
2. Які завдання вирішуються на етапі генерації машинного коду?

8. Аналіз помилок

1. Які типи помилок може виявити компілятор?
2. Як компілятор обробляє помилки, щоб не переривати процес компіляції?

9. Взаємодія етапів компіляції

1. У чому різниця між однопрохідним і багатопрохідним компіляторами?

2. Як взаємодіють різні етапи компіляції?

3. Що собою являє багатопрохідний компілятор? Поясніть його роботу.

Глосарій

Мовний процесор — програма, яка перетворює вихідний код, написаний мовою програмування високого рівня, у форму, яку може виконати комп'ютер. Це може включати компіляцію, інтерпретацію або використання гібридних підходів .

Компілятор — програма, яка перетворює вихідний код у машинний код для конкретної платформи. Наприклад, мови C та C++ зазвичай використовують компілятори.

Інтерпретатор — програма, яка виконує вихідний код без попередньої компіляції, обробляючи його команду за командою. Приклади мов: Python, JavaScript.

Гібридна система — мовний процесор, який використовує байт-код, що виконується віртуальною машиною (наприклад, JVM або CLR). Приклади мов: Java, C#.

Крос-компілятор — компілятор, який створює код для виконання на іншій платформі, ніж та, на якій він працює.

Препроцесор — програма, яка забезпечує обробку коду перед основною компіляцією, наприклад, розширення макросів або включення файлів.

Лексичний аналіз — етап компіляції, на якому вхідний текст програми розбивається на лексеми — найменші значущі одиниці, такі як ключові слова, ідентифікатори, оператори та числові значення.

Синтаксичний аналіз — етап компіляції, на якому перевіряється відповідність коду правилам граматики мови програмування, створюючи дерево синтаксичного розбору.

Семантичний аналіз — етап компіляції, на якому перевіряється коректність використання змінних, операторів та інших елементів мови, що гарантує, що програма відповідає певним правилам мови програмування.

Проміжний код — представлення програми у вигляді, який є проміжним між вихідним кодом і машинним кодом. Використовується для подальшої оптимізації та генерації машинного коду.

Оптимізація коду — процес поліпшення проміжного або машинного коду для збільшення швидкодії або зменшення розміру програми.

Машинний код — код, який може виконуватися процесором або віртуальною машиною без додаткової обробки.

Лексема — найменша значуща одиниця тексту програми, така як ключове слово, ідентифікатор, оператор або числове значення.

Дерево синтаксичного розбору — ієрархічна структура, яка представляє синтаксичну структуру програми, створена під час синтаксичного аналізу.

Таблиця символів — структура даних, яка зберігає інформацію про ідентифікатори, їх типи, області видимості та інші атрибути.

Хеш-таблиця — структура даних, яка використовується для швидкого пошуку інформації про ідентифікатори за допомогою хеш—функції.

Віртуальна машина — програмне середовище, яке виконує байт-код, згенерований компілятором для гібридних мов програмування, таких як Java або C# .

Тема II. Формальні мови і граматики

1 Означення формальних мов

1.1 Алфавіти та ланцюжки

Алфавіт (або словник) V — це скінченна множина символів [1].

Словом (або ланцюжком) в алфавіті V називається послідовність символів, які належать до V.

Множина всіх можливих скінченних слів, що утворені з елементів V, позначається V^* [1, 4].

Важливо зазначити, що ця множина нескінченна. До V^* також належить порожнє слово (слово нульової довжини), яке позначається символом ε.

Підмножину $V^* \setminus \{\varepsilon\}$ позначають V^+, а слово вигляду w^n утворюється шляхом повторення слова $w \in V^+$ n разів. Якщо $n = 0$, то $w^0 = \varepsilon$ [1].

Формальне означення ланцюжків

Формально, ланцюжок x в алфавіті V визначається так [15, 1]:

1. ε — це ланцюжок у V.

2. Якщо x — це ланцюжок у V, а $a \in V$, то xa також є ланцюжком у V.

3. Жоден інший об'єкт не є ланцюжком, окрім тих, що визначені в пунктах 1 і 2.

Операції над ланцюжками

Зчеплення (конкатенація) двох ланцюжків x і y позначається xy і утворюється шляхом об'єднання послідовностей символів із x та y [4].

Приклад: якщо $x = ab$, а $y = cd$, то $xy = abcd$.

Інші поняття:

Префікс: u — це префікс ланцюжка x, якщо існує v, таке що $x = uv$.

Суфікс: v — це суфікс ланцюжка x, якщо існує u, таке що $x = uv$.

Підланцюжок: z — це підланцюжок x, якщо існують u та v, такі що $x = uzv$.

Довжина ланцюжка $|x|$ визначається як кількість символів у x.

1.2 Формальна мова

Будь-яка підмножина множини V^* називається **формальною мовою** [15]. У подальшому тексті цей термін будемо скорочено називати *мовою*. Мова за словником V позначається L_V або просто L, якщо контекст використання алфавіту V очевидний.

Приклади мов [2]

1. $V_1 = \{a\}$, $L_1 = \{\varepsilon, a, aa, aaa, \dots\} = \{a^n \mid n \geq 0\}$. *Мова нескінченна.*

2. *Ідентифікатором* є послідовність букв і цифр, що починається з букви.

Наприклад, для алфавіту $V_2 = \{p, q, 1\}$, множина ідентифікаторів нескінченна: $\{p, q, p1, pp, pq, q1, qp, qq, \dots\}$.

3. $V_3 = \{p, q, r\}$, $L_3 = \{p^n q^n r^n \mid n \geq 0\}$.

Наприклад, $ppqqrr \in L_3$, але $rrqqpp \notin L_3$.

4. $V_4 = \{p\}$, $L_4 = \{\alpha \in V^* \mid |\alpha| = 2k,\ k = 0, 1, \dots\}$.

Наприклад, $pppp \in L_4$, але $ppp \notin L_4$.

5. $V_5 = \{p, q, r\}$, $L_5 = \{p^n q^m r^k \mid n, m, k \geq 0\}$.

Наприклад, $ppppqqrr \in L_5$, але $rrqqpp \notin L_5$.

6. $V_6 = \{p, q, r\}$, $L_6 = \{x \mid$ кількість входжень p, q і r однакова$\}$.

Наприклад, $rrpqpq \in L_6$, але $rrp \notin L_6$.

Проблема належності

Задача належності визначає, чи належить слово w мові L. Для розв'язання цієї задачі мова зазвичай задається кінцевим описом, що включає структуру її елементів. Часто використовується синтаксичний аналіз для перевірки структури слова [7, 8].

Приклад: структура правильних програм у мові Паскаль задається скінченним набором правил (наприклад, форма Бекуса-Наура), які використовуються для створення синтаксичних аналізаторів у компіляторах [16].

Регулярні операції над мовами

Розглянемо ключові регулярні операції, які можна виконувати над мовами: об'єднання, катенацію та ітерацію [6, 15]. Нехай L_1, L_2 та L позначають довільні мови, що визначені в алфавіті V.

- **Об'єднання** $(L_1 \cup L_2)$: Це операція, яка формує нову мову, яка містить есі слова, які належать або до L_1, або до L_2. Формально це множина dсіх слів w, де $w \in L_1$ або $w \in L_2$. Наприклад, об'єднання мов $\{a, ab\}$ та $\{a, b, ba\}$ дає результат $\{a, b, ab, ba\}$.

- **Катенація (конкатенація)** (L_1L_2): Це операція, яка формує нову мову шляхом поєднання кожного слова з L_1 з кожним словом з L_2. Результатом є мова, що складається з усіх можливих комбінацій vw, де $v \in L_1$ та $w \in L_2$. Наприклад, для $L_1 = \{a, bc\}$ і $L_2 = \{x, y\}$ катенація дає множину $\{ax, bcx, ay, bcy\}$. Якщо $L_1 = \{a, ab\}$ і $L_2 = \{\varepsilon, b\}$, то катенація дає множину $\{a, ab, abb\}$.

- **Ітерація** (L^*): Це операція, яка формує нову мову, яка містить усі можливі скінченні послідовності слів із L. Формально це визначається як $L^* = \{\varepsilon\} \cup L \cup L^2 \cup \ldots$. Тобто L^* містить нескінченну кількість слів, якщо в L є

непорожнє слово. Наприклад, вираз $\{ab\}^*$ задає множину $\{\varepsilon, ab, abab, ababab, \dots\}$, а вираз $\{a, b\}\{a, b, 1\}^*$ описує множину ідентифікаторів в алфавіті $\{a, b, 1\}$.

Зазначимо, що операція ітерації не є похідною від операцій катенації чи об'єднання, оскільки вона не може бути виражена за допомогою скінченних операцій [7].

Кожна мова починається з алфавіту та визначає правила для створення найпростіших виразів (лексем) і для побудови складніших виразів із простіших. Ці дві категорії правил зазвичай відомі як **лексична** та **синтаксична** системи мови [9].

2 Метамова БНФ

Кожному виразу мови, починаючи з найпростішого, відповідає певний зміст, який ми називаємо його семантикою. Наприклад, у мовах програмування семантика числової сталої — це певне число, яке зберігається у пам'яті комп'ютера, семантика імені змінної — це адреса в пам'яті, яку можна змінювати, а семантика оператора визначає дії комп'ютера під час виконання цього оператора [4, 7].

Правила, за якими вирази мови отримують свої значення, формують семантичну систему цієї мови.

У кожній мові існує набір понять. Наприклад, кожен конкретний оператор є реалізацією загального поняття «оператор», а ідентифікатор — це реалізація поняття «ідентифікатор», що являє собою послідовність буквено-цифрових символів, де перший символ є літерою.

Оператор присвоєння, наприклад, містить три складові: «ідентифікатор», знак «:=» і «вираз» [7].

Позначення понять мови здійснюється через нетермінальні символи, які ми будемо позначати кутовими дужками, наприклад, <ідентифікатор>. Ці символи не можна поділити. Термінали (лексеми) позначаються апострофами, наприклад,

’:=’ є терміналом. Комбінація терміналів та нетерміналів утворює метавираз [10, 15].

Ось приклад метавиразу:

$$< \text{ідентифікатор} >'{:=}'< \text{вираз} >';'$$

Кожне мовне поняття, яке має вигляд “<поняття>”, можна записати у формі:

$$< \text{поняття} >::=< \text{метавираз} > .$$

Синтаксичні правила, що записуються у вигляді “<поняття> ::= <метавираз>”, називаються формами Бекуса—Наура (БНФ) на честь авторів цих правил. Вони є стандартним способом опису граматик і часто скорочуються до абревіатури БНФ [7, 4]. Ліва частина запису є поняттям, а права частина — метавиразом. Знак “::=” не є частиною мови, це метасимвол.

Розглянемо приклад. Оператор присвоєння може бути записаний як:

$$< \text{оператор присвоєння} >::=< \text{змінна} >'{:=}'< \text{вираз} >$$

Цей запис описує загальну структуру оператора присвоєння, але не визначає конкретний вигляд таких елементів, як <змінна> та <вираз>. Отже, потрібно описати синтаксис цих понять [10]. Наприклад, <змінна> може бути визначена так:

$$< \text{змінна} >::=< \text{буква} >< \text{послідовність букв і цифр} > .$$

Для цього визначення можуть знадобитися додаткові БНФ для інших понять, таких як <буква> та <послідовність букв і цифр> [6].

Тепер розглянемо приклад з операторами присвоєння. Припустимо, ми маємо змінні, імена яких можуть бути лише ’p’, ’q’, ’r’, а вирази праворуч можуть містити сталу 3 або 5, імена ’p’, ’q’, ’r’ або їх суму чи різницю. Тоді синтаксис оператора присвоєння буде таким:

$$< \text{оператор присвоєння} > ::= < \text{змінна} > ' := ' < \text{вираз} >$$

де вираз може бути сталим, або ідентифікатором, або виразом з операціями додавання чи віднімання:

$$< \text{вираз} > ::= < \text{доданок} > |$$

$$< \text{доданок} > ' + ' < \text{доданок} > |$$

$$< \text{доданок} > ' - ' < \text{доданок} >$$

$$< \text{доданок} > ::= < \text{константа} > | < \text{змінна} >$$

де:

$$< \text{константа} > ::= ' 3' | ' 5'$$

$$< \text{змінна} > ::= ' p' | ' q' | ' r'$$

Цей набір правил задає синтаксис оператора присвоєння, виразів, сталих значень та імен, а також множини можливих значень для кожного з понять [15].

Рекурсивні БНФ: Ціле число та ідентифікатор

- `<integer>` визначає ціле число, яке може бути одиничною цифрою або рекурсивно поєднане з іншими цифрами:

$$< integer > ::= < digit > | < integer > < digit >$$

- `<identifier>` описує ідентифікатор, який може бути літерою або рекурсивно поєднаним з іншими літерами та цифрами:

$$< identifier > ::= < letter > |$$

$$< identifier > < letter > |$$

$$< identifier > < digit >$$

- `<digit>` — це цифра, що може бути будь-якою від '0' до '9':

$$< digit >::='0' \mid '1' \mid '2' \mid '3' \mid '4' \mid '5' \mid '6' \mid '7' \mid '8' \mid '9'$$

- `<letter>` — це будь-яка буква латинського алфавіту (великі та малі):

$$< letter >::='A' \mid 'B' \mid \cdots \mid 'z'.$$

Сукупність БНФ: Дійсне число

- `<real-number>` визначає дійсне число, яке складається з цілої частини, десяткової крапки та дробової частини [5, 14]:

$$< real - number >::=< integer - part >'.' < fraction >$$

- `<integer-part>` може бути порожнім або складатися з послідовності цифр:

$$< integer - part >::=< empty > \mid < digit - sequence >$$

- `<fraction>` — це послідовність цифр:

$$< fraction >::=< digit - sequence >$$

- `<digit-sequence>` — це будь-яка послідовність цифр:

$$< digit - sequence >::=< digit > \mid < digit >$$

$$< digit - sequence >$$

- `<empty>` позначає порожнє значення:

$$< empty >::= \varepsilon$$

Підсумуємо: БНФ — це система запису, що складається з терміналів, нетерміналів і метасимволів [7, 10]. Вона має чітко визначену структуру, де нетермінал, потім символ "::=" і правий вираз формують правильний запис.

Семантика БНФ полягає в описі структури та множин можливих представників понять, позначених нетерміналами. Отже, БНФ є метамовою, яка використовується для опису інших мов [4].

Існують різні метамови, деякі з яких застосовуються в строгих формальних системах, зокрема в логіці та математиці. Мова БНФ, описана в цьому контексті, є неформальним прикладом формальної метамови – частини мов формальних граматик. Вона була розроблена спеціально для опису синтаксису мов програмування [4, 7].

3 Розширена БНФ

Розглянемо розширену версію формальних граматик БНФ, що містить додаткові можливості для зручнішого запису та виразності. Для цього введемо концепцію еквівалентності БНФ: дві граматики вважаються еквівалентними, якщо вони описують одну й ту ж мову [7, 15].

Для зручності запису еквівалентних правил БНФ, їх можна подати більш компактно за допомогою додаткових символів, таких як "(", ")", "[", ']", "{", "}". БНФ, що використовує ці символи, називається розширеною БНФ (РБНФ). Розглянемо деякі приклади побудови таких граматик.

Нехай літери $X, Y, Z, \ldots, T$ позначають довільні метавирази, що можуть бути порожніми, а N — будь-який нетермінал. Ми можемо замінити кілька правил вигляду:

$$N ::= XZY$$

та

$$N ::= XTY$$

на одне скорочене правило:

$$N ::= X(Z \mid \cdots \mid T)Y.$$

У цьому випадку метасимволи “(“ та “)” просто групують альтернативи $Z, \ldots, T$ в одну частину виразу, не змінюючи їх значення. Наприклад, правила

$$<\text{вираз}> ::= <\text{доданок}>' +' <\text{доданок}> \mid$$

$$<\text{доданок}>' -' <\text{доданок}>$$

можна замінити на

$$<\text{вираз}> ::= <\text{доданок}>('+' \mid' -')<\text{доданок}>.$$

Заміна двох правил вигляду

$$N ::= XZY$$

$$N ::= XY$$

на таке правило:

$$N ::= X[Z]Y.$$

також приводить до еквівалентної граматики. У цьому випадку квадратні дужки означають, що частина виразу в середині може бути або наявною, або відсутньою [10, 14]. Наприклад, замість таких правил:

$$<\text{вираз}> ::= <\text{доданок}> \mid$$

$$<\text{доданок}>('+' \mid' -')<\text{доданок}>$$

можна використовувати правило:

$$<\text{вираз}> ::= <\text{доданок}>[('+' \mid' -')<\text{доданок}>].$$

Іноді для спрощення можна замінити складні поняття на більш прості нетермінали або навіть на окремі символи. Наприклад, замість нетермінала <доданок> з попереднього прикладу можна використати метавирази типу:

$$\text{<доданок>} ::= \text{<константа>} \mid \text{<змінна>} \mid$$
$$'1' \mid' 2' \mid' x' \mid' y' \mid' z'$$

Отже, початкову сукупність БНФ можна переробити у більш спрощену еквівалентну форму:

$$\text{<оператор присвоєння>} ::= \text{<змінна>}' :=' ('1' \mid' 2' \mid \text{<змінна>})$$
$$[('+' \mid' -')('1' \mid' 2' \mid \text{<змінна>})]$$
$$\text{<змінна>} ::=' x' \mid' y' \mid' z'.$$

Розглянемо приклад використання метасимволів "{}", що означають ітерації. У мовах програмування ідентифікатор зазвичай є послідовністю букв та цифр, що починається з літери. У цьому прикладі літерою буде лише одна з букв `A`, `B`, `C`, а цифри — це тільки `0` та `1`. Тому ідентифікатори можуть виглядати, наприклад, як `A`, `B1`, `BC`, `C1CAAB0` тощо. Для опису таких ідентифікаторів у вигляді БНФ використаємо такий запис:

$$\text{<Ід>} ::= \text{<Б>}\{\text{<Б>} \mid \text{<Ц>}\}$$

де

$$\text{<Б>} ::=' A' \mid' B' \mid' C'$$
$$\text{<Ц>} ::=' 0' \mid' 1'$$

Можна скоротити це правило, комбінуючи літери та цифри в одному виразі:

$$\text{<Ід>} ::= ('A' \mid' B' \mid' C')\{'A' \mid' B' \mid' C' \mid' 0' \mid' 1'\}.$$

У даному випадку фігурні дужки $\{X\}$ позначають усі можливі послідовності виразів, що можуть бути отримані з виразу X, включаючи порожні послідовності.

Перетворення рекурсивної БНФ у РБНФ без прямої рекурсії

Ми розглядаємо перетворення граматики оператора присвоєння з рекурсивної форми БНФ (BNF) на розширену форму РБНФ (EBNF). Це дозволяє уникнути прямої рекурсії, що може бути складною для обробки парсерами.

Рекурсивна БНФ

У рекурсивній формі граматика для оператора присвоєння та виразу виглядає так:

$$\begin{aligned}
&\text{<oper-assign>} ::= \text{<iden>}' :=' \text{<expr>} \\
&\text{<expr>} ::= \text{<iden>}' +' \text{<expr>} \\
&\quad | \ \text{<iden>}' *' \text{<expr>} \\
&\quad |' (' \text{<expr>}')' \\
&\quad | \ \text{<iden>}
\end{aligned}$$

- <oper-assign> — оператор присвоєння складається з ідентифікатора (<iden>), знака присвоєння ‘:=‘ та виразу (<expr>).
- <expr> — вираз може бути:
- додаванням або множенням ідентифікаторів,
- виразом у дужках,
- або самим ідентифікатором.

Ця граматика рекурсивна, оскільки правило для <expr> містить рекурсивне викликання самого себе.

Перетворення в РБНФ

У розширеній РБНФ (EBNF) ми можемо записати цю граматику без прямої рекурсії, використовуючи додаткові конструкції, такі як повторення та альтернативи. Ось як це виглядатиме:

$$\text{<oper-assign>} ::= \text{<iden>}' :=' \text{<expr>}$$

$$\text{<expr>} ::= \text{<iden>}\{('+' \mid' *')\text{<term>}\}$$

$$\text{<term>} ::=' ('\text{<expr>}')'$$

- <oper-assign> залишається незмінним: оператор присвоєння складається з ідентифікатора, знака присвоєння та виразу. - <expr> більше не має прямої рекурсії. Використовуємо конструкцію {...}, що означає, що вираз може повторюватися один або більше разів з операціями додавання або множення. - <term> визначається як вираз у дужках, що дозволяє коректно групувати частини виразу.

Перетворення з BNF в EBNF дозволяє уникнути проблеми нескінченної рекурсії і спрощує побудову парсерів.

Розширена форма БНФ (EBNF) для слів, які відповідають масці

Маска: `ab:f::`

Опис

Метасимвол – **:**
Зміст метасимволу – непорожній ланцюжок бінарних цифр.

Слова, що відповідають масці

Слова: `ab1100f01`, `ab000f00`, `ab1f011`.

EBNF для слів

$$\langle \text{word} \rangle ::= \texttt{'ab'}\langle \text{meta} \rangle \texttt{'f'}\langle \text{meta} \rangle \langle \text{meta} \rangle$$
$$\langle \text{meta} \rangle ::= (0 \mid 1)\{0 \mid 1\}.$$

Пояснення

- Нетермінал `<word>` визначає загальну структуру слова.
- Нетермінал `<meta>` описує непорожній ланцюжок, що складається із символів `0` та `1`.

4 Граматики Хомського. Основні поняття

Граматика Хомського є формальною системою, що описується четвіркою $G = (N, T, P, S)$. Тут [4, 9]:

- T — алфавіт описуваної мови, який складається з множини термінальних символів (терміналів);
- N — множина нетермінальних символів, які використовуються для позначення понять мови;
- P — набір правил виведення (продукцій) виду $v \to w$, де

  $$v \in (T \cup N)^* N (T \cup N)^*, \quad w \in (T \cup N)^*,$$

  тобто ліва частина правила повинна містити принаймні один нетермінал, а права є довільною послідовністю символів з $T \cup N$;
- S — початковий нетермінал із множини N, що представляє головне поняття мови.

Множина продукцій P є скінченною підмножиною множини всіх можливих правил вигляду $v \to w$. Якщо $(n, w) \in P$, то в граматиці G існує правило $n \to w$. Якщо $u \to v \in P$ і

ланцюжок α містить u як підланцюжок, то замінивши u на v у α, ми отримаємо новий ланцюжок, що також породжується граматикою G.

Граматика може бути записана як у формалізмі Хомського, так і у форматі БНФ (Backus-Naur Form). Порівняємо обидва підходи на прикладі граматики, що генерує 4 речення.

Граматика у формалізмі Хомського (на прикладі граматики, яка генерує 4 речення)

Формалізація:

$$G = (N, T, P, S),$$

де:

- $T = \{$заняття, викладач, проводить$\}$ — термінальний алфавіт.
- $N = \{S, N, V\}$ — множина нетерміналів:
  - S — речення,
  - N — іменник,
  - V — дієслово.
- P — множина правил:

$$\begin{aligned} S &\to N\,V\,N, \\ N &\to \text{заняття}, \\ N &\to \text{викладач}, \\ V &\to \text{проводить}. \end{aligned}$$

Формалізація у форматі БНФ:

```
<S> ::= <N> <V> <N>
<N> ::= "заняття" | "викладач"
<V> ::= "проводить"
```

Виведення ланцюжків за допомогою граматики

У даній граматиці ми виконаємо всі чотири можливі виведення ланцюжків із використанням подвійної стрілки $\Rightarrow$ для позначення кроків виведення [9].

1. **Виведення 1:**

$$\begin{aligned} S &\Rightarrow N\,V\,N, \\ &\Rightarrow \text{заняття}\,V\,N, \\ &\Rightarrow \text{заняття проводить}\,N, \\ &\Rightarrow \text{заняття проводить викладач}. \end{aligned}$$

2. **Виведення 2:**

$$\begin{aligned} S &\Rightarrow N\,V\,N, \\ &\Rightarrow \text{викладач}\,V\,N, \\ &\Rightarrow \text{викладач проводить}\,N, \\ &\Rightarrow \text{викладач проводить заняття}. \end{aligned}$$

3. **Виведення 3:**

$$\begin{aligned} S &\Rightarrow N\,V\,N, \\ &\Rightarrow \text{заняття}\,V\,N, \\ &\Rightarrow \text{заняття проводить}\,N, \\ &\Rightarrow \text{заняття проводить заняття}. \end{aligned}$$

4. **Виведення 4:**

$$\begin{aligned} S &\Rightarrow N\,V\,N, \\ &\Rightarrow \text{викладач}\,V\,N, \\ &\Rightarrow \text{викладач проводить}\,N, \\ &\Rightarrow \text{викладач проводить викладач}. \end{aligned}$$

Отже, граматика G породжує такі речення:

- заняття проводить викладач,
- викладач проводить заняття,
- заняття проводить заняття,
- викладач проводить викладач.

Приклад граматики, яка продукує нескінченну кількість слів

$$N = \{S, A, B, C\}, \quad T = \{a, b, c\}$$

$$P : \begin{array}{l} S \to ABC \\ A \to aA \mid a \\ B \to bB \mid b \\ C \to cC \mid c \end{array}$$

Приклад виведення

$$\begin{aligned} S &\Rightarrow ABC \\ &\Rightarrow aABC \\ &\Rightarrow aaABC \\ &\Rightarrow aaaBC \\ &\Rightarrow aaabC \\ &\Rightarrow aaabcC \\ &\Rightarrow aaabcc \end{aligned}$$

Опис граматики

Граматика описує слова, які:

- починаються з однієї або більшої кількості символів a;

- за якими йде один або більше символів b;
- і далі один або більше символів c.

Мова, описана цією граматикою, має нескінченну кількість виведень [9].

Позначення продукцій

Для зручності запису правила можуть бути об'єднані:

- $v \to w_1$ і $v \to w_2$ записують як $v \to w_1 \mid w_2$,
- $v \to w_1 u w_2$ записують як $v \to w_1[u]w_2$,
- $v \to v_1 u_1 v_2$ і $v \to v_1 u_2 v_2$ записують як $v \to v_1(u_1 \mid u_2)v_2$.

Приклад граматики

Розглянемо граматику $G_1 = (\{A, B, C, D\}, \{a, 1, 2\}, P, A)$, де

$$P = \{A \to BC,\ A \to BD,\ A \to B,\ B \to a,\ C \to 1,\ D \to 2\}.$$

Цю граматику можна переписати в скороченій формі:

$$P = \{A \to B[C \mid D],\ B \to a,\ C \to 1,\ D \to 2\}.$$

Задання мови граматикою

Мова, що задається граматикою G, визначається через такі поняття:

1. Вводиться відношення *безпосередньої виводимості*, позначене як $\Rightarrow_G$:

   $$v \Rightarrow_G w, \quad \text{якщо } v = x_1 u x_2,\ w = x_1 y x_2,\ u \to y \in P.$$

   Наприклад, у граматиці G_1: $BC \Rightarrow aC$ за допомогою $B \to a$, а $aC \Rightarrow a1$ за допомогою $C \to 1$.

2. Визначається відношення *виводимості*, позначене як $\Rightarrow_G^*$:

$$v \Rightarrow_G^* w, \quad \text{якщо } v = w \text{ або існує послідовність}$$

$$v \Rightarrow w_1 \Rightarrow \cdots \Rightarrow w.$$

Наприклад, у G_1: $BC \Rightarrow_G^* a1$, оскільки $BC \Rightarrow aC \Rightarrow a1$.

3. Слово u називається *вивідним*, якщо $S \Rightarrow_G^* u$. Всі такі слова утворюють мову $L(G)$:

$$L(G) = \{u \mid u \in T^* \text{ та } S \Rightarrow_G^* u\}.$$

Еквівалентність граматик

Дві граматики називаються еквівалентними, якщо вони задають одну й ту саму мову. Наприклад:

$$\bar{G}_1 = (\{A\}, \{a, 1, 2\}, \{A \to a[1 \mid 2]\}, A)$$

еквівалентна G_1, а граматика

$$G_2 = (\{I, L, D\}, \{a, \ldots, z, 0, \ldots, 9\},$$

$$\{I \to L|IL|ID, L \to a|\ldots|z, D \to 0|\ldots|9\}, I)$$

еквівалентна $\bar{G}_2$:

$$\bar{G}_2 = (\{I, C\}, \{a, \ldots, z, 0, \ldots, 9\},$$

$$\{I \to (a \mid \cdots \mid z)C, C \to \varepsilon \mid C(a \mid \cdots \mid z \mid 0 \mid \cdots \mid 9)\}, I)$$

Зазначимо. що граматики G_2 і $\bar{G}_2$ описують множину ідентифікаторів в алфавіті $\{a, \ldots, z, 0, \ldots, 9\}$.

Додаткові позначення

- Символи $a, b, \ldots, 0, 1, \ldots, 9$ позначають термінали.
- Літери A, B, C, D, E, S позначають нетермінали.
- Символи E, S вказують на початкові нетермінали.
- $U, V, \ldots, Z$ можуть бути терміналами або нетерміналами.
- α, β — ланцюжки, що містять термінали та нетермінали.
- $u, v, \ldots, z$ — ланцюжки, які містять лише термінали.

Класифікація граматик Хомського. Приклади

Граматики класифікують залежно від структури їх правил. Нижче наведено класифікацію, що отримала назву **ієрархія Хомського** [4, 6].

Означення 1

Граматика G називається *праволінійною* (або *ліволінійною*), якщо всі правила з P мають вигляд:

$$A \to wB \quad (\text{або } A \to Bw),$$

де $w \in T^*$, $A, B \in N$.

Приклад

Розглянемо граматику G з такими правилами:

$$S \to aA, \quad A \to bB, \quad B \to c.$$

Ця граматика є праволінійною, оскільки всі правила відповідають формату $A \to wB$.

Приклади граматик: праволінійна та ліволінійна

1. Непорожній ланцюжок бінарних цифр (двійкові числа)

Ліволінійна граматика:

$$G_{\text{left}} = (N, T, P, S)$$

- $N = \{S\}$ (нетермінальні символи),
- $T = \{0, 1\}$ (термінальні символи),
- $P = \{S \to 0,\ S \to 1,\ S \to S0,\ S \to S1\}$ (правила продукції),
- Початковий символ: S.

2. Трійкові числа (зокрема, порожній ланцюжок)

Праволінійна граматика:

$$G_{\text{right}} = (N, T, P, S)$$

- $N = \{S\}$ (нетермінальні символи),
- $T = \{0, 1, 2\}$ (термінальні символи),
- $P = \{S \to 0S,\ S \to 1S,\ S \to 2S,\ S \to \varepsilon\}$ (правила продукції),
- Початковий символ: S.

Означення 2

Граматика G називається *контекстно-вільною*, якщо всі правила з P мають вигляд

$$A \to \alpha, \quad A \in N,\ \alpha \in (T \cup N)^*.$$

Приклад

Розглянемо граматику:

$$S \to aSb \mid \varepsilon.$$

Ця граматика контекстно-вільна, оскільки лівою частиною кожного правила є лише один нетермінал.

Приклад

Розглянемо контекстно-вільну граматику G, яка генерує слова, що відповідають масці `ab:f::` з метасимволом `:`, де:

- `a`, `b`, `f` — фіксовані символи;
- `:` позначає місце для непорожнього ланцюжка бінарних цифр (0 або 1).

Слова, які відповідають масці:

- `ab1100f01`
- `ab000f00`
- `ab1f011`

Граматика

Граматика G задається у вигляді

$$G = (N, T, P, S),$$

де

- $N = \{W, M\}$ — множина нетерміналів;
- $T = \{a, b, f, 0, 1\}$ — множина терміналів;

- P — множина правил виводу:

$$P = \begin{cases} W \to abMfMM, \\ M \to 0, \\ M \to 1, \\ M \to M1, \\ M \to M0; \end{cases}$$

- $S = W$ — початковий символ.

Пояснення

- Початковий нетермінал W задає основну структуру слова: починається на `ab`, далі йде M, потім символ `f`, за яким йдуть ще два M.

- Нетермінал M відповідає за непорожній ланцюжок бінарних цифр. За допомогою правил:

$$M \to 0, \quad M \to 1, \quad M \to M0, \quad M \to M1,$$

  M може генерувати будь-який ланцюжок бінарних символів, що є непорожнім.

Отже, граматика G генерує всі слова, які відповідають метасимволу `ab:f::`, наприклад, `ab1100f01`, `ab000f00`, `ab1f011`, і багато інших.

Означення 3

Граматика G називається *контекстно-залежною* (або *нескоротною*), якщо всі правила з P мають вигляд

$$\alpha \to \beta, \quad \alpha \in (T \cup N)^+, \quad \beta \in (T \cup N)^*, \quad |\alpha| \leq |\beta|,$$

де T — множина термінальних символів, а N — множина нетермінальних символів.

Ключовою особливістю контекстно-залежних граматик є те, що вони **не допускають** правил вигляду $A \to \varepsilon$, тобто виведення порожнього символу. Це означає, що жоден нетермінальний символ не може бути замінений на порожню послідовність символів.

Отже, кожне правило контекстно-залежної граматики повинно мати ліву частину, яка не коротша за праву частину. Це забезпечує неможливість скорочувати слова під час виведення, що дозволяє моделювати мови з більш складною структурою, ніж це можливо для контекстно-вільних граматик.

Приклад

Розглянемо граматику

$$S \to aSb \mid ab.$$

Ця граматика задає мову парної кількості символів a та b, де перша половина ланцюжка збігається з другою.

Приклад

Нехай задано граматику:

$$G = (\{B, C, S\}, \{a, b, c\}, P, S),$$

де P — множина правил:

$$\begin{aligned} &S \to aSBC \mid abC, \\ &CB \to BC, \\ &bB \to bb, \\ &bC \to bc, \\ &cC \to cc. \end{aligned}$$

Мова, яку генерує ця граматика:

$$L(G) = \{a^n b^n c^n \mid n \geq 1\}.$$

Покажемо приклад виведення слова $aabbcc$:

$$S \Rightarrow aSBC \Rightarrow aaBCBC \Rightarrow aabCBC \Rightarrow$$

$$aabBC \Rightarrow aabbCC \Rightarrow aabbcC \Rightarrow aabbcc.$$

Отже, слово $aabbcc \in L(G)$.

Означення 4

Граматика, яка не відповідає жодним обмеженням з Означень 1–3, називається *необмеженою* або *граматикою загального вигляду.*

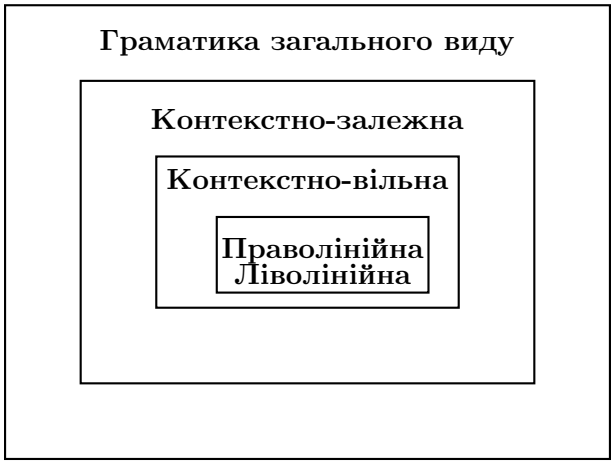
Приклад

Розглянемо граматику

$$S \to aS \mid b.$$

Ця граматика не обмежена у вигляді своїх правил.

Ієрархію граматик з означень 1-4 можна зобразити наступним чином:



Означення 5

Праволінійна граматика $G = (T, N, P, S)$ називається *регулярною* (або автоматною), якщо:

- кожне правило з P, окрім $S \to \varepsilon$, має вигляд $A \to Ba$ або $A \to a$, де $A, B \in N$, $a \in T$;
- якщо $S \to \varepsilon \in P$, то S не з'являється у правих частинах інших правил.

Приклади

- $G = (\{A, S\}, \{a, b\}, P, S)$, де

$$P: \ S \to abA \mid \varepsilon, \ A \to Saa \mid b.$$

  Ця граматика нерегулярна й неправолінійна.

- $G = (\{A, C, S\}, \{a, b\}, P, S)$, де

$$P: \ S \to aC \mid \varepsilon, \ A \to a, \ C \to bA \mid bC \mid \varepsilon.$$

  Ця граматика регулярна.

- $G = (\{S\}, \{0, 1\}, P, S)$, де

$$P: \ S \to 0S \mid 1S \mid \varepsilon.$$

  Ця граматика праволінійна, але нерегулярна.

Означення 6

Граматика називається *розширеною*, якщо вона задається списком пар:

$$(A_i, r_i), \quad A_i \in N, \ r_i \text{ — регулярний вираз.}$$

Термінальний алфавіт T утворюється зі символів, що входять у r_i, виключаючи елементи $N \cup \{(,), *, |\}$.

Приклади розширених граматик

- $\tilde{G} = \{S \to AB,\ B \to x \mid y,\ A \to z \mid w\}$.
- $\bar{G} = \{S \to (z \mid w)(x \mid y)\}$.
- $\tilde{G} \sim \bar{G}$, тобто ці граматики задають одну й ту саму мову.

Контрольні запитання та завдання

1. Формальні мови: основні поняття

1. Що таке алфавіт формальної мови? Наведіть приклад.
2. Як визначається порожнє слово і як воно позначається?
3. Що таке множина V^* і які її властивості?
4. У чому полягає різниця між множинами V^* і V^+?
5. Що таке формальна мова? Як вона позначається?
6. Наведіть приклади нескінченних і скінченних формальних мов.
7. Що таке задача належності? Як вона застосовується до формальних мов?

2. Операції над ланцюжками та мовами

1. Що таке конкатенація ланцюжків? Наведіть приклад.
2. Поясніть, що таке префікс, суфікс та підланцюжок. Дайте приклади.
3. Які основні регулярні операції застосовуються до формальних мов?
4. Як визначається ітерація мови? Наведіть приклад використання.

5. Як обчислюється об'єднання та конкатенація формальних мов?

3. Метамови та граматики

1. Що таке метамова Бекуса—Наура (БНФ)? Яка її мета?
2. Як виглядає запис оператора присвоєння у формі БНФ? Наведіть приклад.
3. У чому полягає відмінність між БНФ та розширеною БНФ (EBNF)?
4. Які символи використовуються в розширеній БНФ для спрощення запису?
5. Як перетворити рекурсивну граматику у БНФ на граматику у EBNF?

4. Граматики Хомського

1. Що таке граматика Хомського? Які її основні компоненти?
2. Як визначається граматика у формалізмі Хомського?
3. Наведіть приклад граматики, яка генерує речення з чотирьох слів.
4. Як записати граматику у форматі БНФ? Наведіть приклад.
5. Що таке продукція? Як вона позначається у граматиці?

5. Класифікація граматик Хомського

1. Які типи граматик виділяються за ієрархією Хомського?
2. Що таке праволінійна граматика? Наведіть приклад.

3. Що таке контекстно-вільна граматика? Наведіть приклад.

4. Як відрізнити контекстно-залежну граматику від інших типів?

5. Що таке регулярна граматика? Як вона визначається?

Глосарій

Алфавіт — скінченна множина символів, яка використовується для побудови слів у формальній мові. Позначається як V. Наприклад, алфавіт $V = \{a, b, c\}$.

Ланцюжок (слово) — послідовність символів, що належать до алфавіту V. Наприклад, для алфавіту $V = \{a, b\}$, ланцюжком може бути $abab$.

Множина V^* — множина всіх можливих скінченних слів, утворених з елементів алфавіту V, включаючи порожнє слово ε.

Множина V^+ — множина всіх можливих скінченних слів, утворених з елементів алфавіту V, без порожнього слова ε.

Порожнє слово (ε) — слово нульової довжини, яке не містить жодного символу. Воно є елементом множини V^*.

Формальна мова — будь-яка підмножина множини V^*. Наприклад, мова $L = \{a^n \mid n \geq 0\}$ над алфавітом $V = \{a\}$.

Конкатенація — операція об'єднання двох ланцюжків. Якщо $x = ab$ і $y = cd$, то конкатенація $xy = abcd$.

Префікс — ланцюжок u є префіксом ланцюжка x, якщо існує ланцюжок v, такий що $x = uv$.

Суфікс — ланцюжок v є суфіксом ланцюжка x, якщо існує ланцюжок u, такий що $x = uv$.

Підланцюжок — ланцюжок z є підланцюжком ланцюжка x, якщо існують ланцюжки u та v, такі що $x = uzv$.

Довжина ланцюжка ($|x|$) — кількість символів у ланцюжку x. Наприклад, $|abc| = 3$.

Задача належності — задача визначення, чи належить слово w до мови L. Наприклад, чи належить слово $abab$ до мови $L = \{ab, ba\}$.

Метамова Бекуса—Наура (БНФ) — формальна система для опису синтаксису мов програмування. Використовує правила вигляду < поняття >::=< метавираз >.

Термінал — символ, який є частиною алфавіту мови. Наприклад, у мові програмування терміналами можуть бути ключові слова, оператори або роздільники.

Нетермінал — символ, який використовується для позначення понять у граматиці. Наприклад, < вираз > або < оператор >.

Граматика Хомського — формальна система, що описується четвіркою $G = (N, T, P, S)$, де N — множина нетерміналів, T — множина терміналів, P — множина правил виведення, S — початковий нетермінал.

Праволінійна граматика — граматика, у якій всі правила мають вигляд $A \to wB$ або $A \to w$, де A, B — нетермінали, w — ланцюжок терміналів.

Контекстно—вільна граматика — граматика, у якій всі правила мають вигляд $A \to \alpha$, де A — нетермінал, α — ланцюжок терміналів і нетерміналів.

Контекстно—залежна граматика — граматика, у якій всі правила мають вигляд $\alpha \to \beta$, де $|\alpha| \leq |\beta|$, і α містить принаймні один нетермінал.

Регулярна граматика — праволінійна граматика, у якій всі правила мають вигляд $A \to aB$ або $A \to a$, де A, B — нетермінали, a — термінал.

Розширена БНФ (EBNF) — розширена версія БНФ, яка дозволяє використовувати додаткові символи, такі як {}, [], (), для більш компактного запису граматик.

Ітерація (L^*) — операція, яка формує нову мову, що містить всі можливі скінченні послідовності слів з мови L, включаючи порожнє слово ε.

Катенація (L_1L_2) — операція, яка формує нову мову шляхом поєднання кожного слова з L_1 з кожним словом з L_2.

Об'єднання ($L_1 \cup L_2$) — операція, яка формує нову мову, що містить усі слова, які належать або до L_1, або до L_2.

Тема III. Регулярні множини і регулярні вирази

1 Основні поняття регулярних множин

Регулярні множини є фундаментальним поняттям у теорії формальних мов [1]. Вони визначаються за допомогою регулярних виразів, які описують множини рядків, що задовольняють певні умови. У цій темі розглянемо основні визначення, властивості та приклади регулярних множин [2].

Нехай V — скінченний алфавіт.

Регулярна множина в алфавіті V визначається рекурсивно так [1, 4]:

1. $\varnothing$ є регулярною множиною над V;
2. $\{\varepsilon\}$ є регулярною множиною над V;
3. $\{a\}$ є регулярною множиною для кожного $a \in V$;
4. Якщо A і B — регулярні множини, то
   - $A \cup B$ — регулярна множина;
   - AB (конкатенація) — регулярна множина;
   - A^* (ітерація) — регулярна множина.
5. Жодна інша множина не регулярна.

Отже, множина регулярна, якщо вона може бути отримана за допомогою вказаних операцій: об'єднання, конкатенації або ітерації з базових випадків. Регулярні множини також можуть бути описані за допомогою регулярних виразів, праволінійних граматик, а також автоматів (детермінованих чи недетермінованих) [1, 4].

Регулярний вираз у алфавіті V визначається так [1, 5]:

1. $\varnothing$ є регулярним виразом, який позначає порожню множину.

2. ε є регулярним виразом, який позначає множину, що містить лише порожнє слово $\{\varepsilon\}$.

3. Для кожного символу $a \in V$, a є регулярним виразом, який позначає множину, що містить лише цей символ $\{a\}$.

4. Якщо p і q є регулярними виразами, то:

   - $(p + q)$ позначає об'єднання множин $P \cup Q$.
   - (pq) позначає конкатенацію множин PQ.
   - $(p)^*$ позначає ітерацію множини P^*.

5. Жодний інший вираз не регулярний.

Це рекурсивне означення дозволяє побудувати складніші регулярні вирази на основі простіших. Регулярні вирази широко використовуються для опису формальних мов та для побудови лексичних аналізаторів [1, 5].

Щоб спростити запис регулярних виразів, визначимо пріоритети операцій: ітерація $(*)$, конкатенація, об'єднання $(+)$. Можна використовувати й круглі дужки для явного позначення порядку виконання.

Приклади регулярних виразів

1. Вираз 10 позначає множину $\{10\}$;

2. Вираз 1^* позначає множину $\{1\}^*$;

3. Вираз $(0 + 1)^*$ позначає множину $\{0, 1\}^*$;

4. $(1 + 0)^*101$ — множина всіх рядків, що складаються з символів алфавіта $V = \{0, 1\}$ і закінчуються на 101;

5. $(a + b + c)(a + b + c + 1 + 2)^*$ — множина всіх рядків що складаються з символів алфавіта $V = \{a, b, c, 1, 2\}$, які починаються на a, b або c;

6. $(10 + 00)^*(10 + 01)(00 + 11)^*$ — множина рядків, у яких кількість нулів і одиниць парна.

Для кожного регулярного виразу існує відповідна регулярна мова, і навпаки, для кожної регулярної мови можна знайти регулярний вираз. Два регулярні вирази вважаються рівними, якщо вони позначають одну й ту саму регулярну множину [1, 5].

2 Тотожності для регулярних виразів

Нехай p, q, r — регулярні вирази. Тоді справедливі такі тотожності [1, 4]:

1. $p + q = q + p$ (комутативність об'єднання)
2. $(p + q) + r = p + (q + r)$ (асоціативність об'єднання)
3. $p(qr) = (pq)r$ (асоціативність конкатенації)
4. $p + p = p$ (ідемпотентність об'єднання)
5. $p\varnothing = \varnothing p = \varnothing$ (конкатенація з порожньою множиною)
6. $p\varepsilon = \varepsilon p = p$ (конкатенація з порожнім словом)
7. $\varnothing^* = \varepsilon$ (ітерація порожньої множини)
8. $p^* = p^*p^*$ (ітероване повторення)
9. $p^* = \varepsilon + pp^*$ (розклад ітерації)
10. $\varnothing + p = p$ (поглинання порожньої множини об'єднанням)
11. $\varepsilon p = p\varepsilon = p$ (нейтральний елемент для конкатенації)
12. $p(p^*) = (p^*)p = p^*$ (ітерація конкатенації)

Доведення першої тотожності.

Нехай r, q позначають множини R, Q. Тоді $r + q$ позначає $R \cup Q$, а $q + r$ позначає $Q \cup R$. Оскільки $R \cup Q = Q \cup R$, то $r + q = q + r$.

Надалі не будемо розрізняти регулярний вираз і регулярну мову (вираз a представляє множину $\{a\}$).

Теорема

Мова регулярна тоді і тільки тоді, коли її можна визначити за допомогою праволінійної граматики (без доведення) [1, 4].

3 Побудова регулярного виразу за праволінійною граматикою

Для побудови регулярного виразу за праволінійною граматикою використовуються рівняння з регулярними коефіцієнтами [1, 5].

Розглянемо рівняння, у яких коефіцієнти та невідомі є регулярними множинами або виразами. Такі рівняння називаються **рівняннями з регулярними коефіцієнтами** [4, 5].

Ці рівняння мають специфічний вигляд і можуть бути представлені у формі:

$$X = \alpha X + \beta, \tag{1}$$

де α та β — регулярні вирази.

Покажемо, що $X = \alpha^* \beta$ є розв'язком рівняння (1):

$$\alpha^* \beta = \alpha(\alpha^* \beta) + \beta.$$

Для цього розглянемо ліву та праву частини рівняння:
Ліва частина:

$$\alpha^* \beta$$

Права частина:

$$\alpha(\alpha^* \beta) + \beta = \alpha\alpha^* \beta + \beta.$$

Оскільки α^* позначає множину всіх можливих скінченних послідовностей слів, які утворені символами з α, включаючи порожню послідовність, то:

$$\alpha^*\beta = \{\varepsilon, \alpha, \alpha\alpha, \alpha\alpha\alpha, \dots\}\beta.$$

Кожне слово у $\alpha^*\beta$ може бути представлене як послідовність символів з α, можливо порожньої довжини, яка закінчується ланцюжком β. Тоді отримуємо:

$$\alpha^*\beta = \alpha\alpha^*\beta + \beta.$$

Отже, $X = \alpha^*\beta$ є розв'язком рівняння (1).

Рівняння з регулярними коефіцієнтами широко використовуються у теорії формальних мов та автоматів для опису складних структур та властивостей мов. Вони дозволяють аналізувати та моделювати поведінку систем, які працюють з регулярними множинами [4, 5].

Для рівнянь з регулярними коефіцієнтами може бути нескінченно багато розв'язків. Якщо в (1) α представляє ε, то $X = \alpha^*(\beta + \gamma)$ - також розв'язок (1), де γ - довільний регулярний вираз ($\beta + \gamma = \beta + \gamma + \beta$).

Оскільки розв'язків безліч, то будемо брати найменший, який буде називатися найменшою нерухомою точкою. Для рівняння (1)

$$X = \alpha^*\beta$$

- найменша нерухома точка.

Розглянемо тепер систему рівнянь з регулярними коефіцієнтами:

$$\begin{cases} X_1 = \alpha_{11}X_1 + \alpha_{12}X_2 + \dots + \alpha_{1n}X_n + \beta_1 \\ X_2 = \alpha_{21}X_1 + \alpha_{22}X_2 + \dots + \alpha_{2n}X_n + \beta_2 \\ \dots\dots\dots\dots\dots\dots\dots\dots \\ X_n = \alpha_{n1}X_1 + \alpha_{n2}X_2 + \dots + \alpha_{nn}X_n + \beta_n \end{cases}$$

Тут α_{ij}, β_i — відомі регулярні вирази, X_i — невідомі

регулярні вирази $(i, j = 1, 2, \ldots, n), \quad \alpha_{11} \neq \varnothing$.

Перепишемо перше рівняння

$$X_1 = \alpha_{11} X_1 + \beta, \text{де } \beta = \alpha_{12} X_2 + \ldots + \alpha_{1n} X_n + \beta_1.$$

Найменша нерухома точка цього рівняння

$$X_1 = (\alpha_{11})^* \beta.$$

Використовуючи подібний підхід для інших рівнянь, можна розв'язати систему рівнянь з регулярними коефіцієнтами [4, 5].

Приклад 1: Розв'язати систему рівнянь із регулярними коефіцієнтами:

$$\begin{cases} A_1 = (1 + 01^*) A_1 + A_2 \\ A_2 = 1A_1 + 00A_3 + 11 \\ A_3 = \varepsilon + A_1 \end{cases}$$

Розв'язання:

Розв'язок рівняння з регулярними коефіцієнтами $X = \alpha X + \beta$ визначається формулою $X = \alpha^* \beta$. Використаємо її:

$$A_1 = (01^* + 1)^* A_2$$
$$A_3 = \varepsilon + A_1$$
$$A_2 = 11 + 1A_1 + 00(\varepsilon + A_1)$$
$$A_2 = 11 + 00 + (1 + 00)A_1$$
$$A_2 = 11 + 00 + (1 + 00)(01^* + 1)^* A_2$$

$$A_2 = ((1 + 00)(01^* + 1)^*)(11 + 00)$$
$$A_1 = (01^* + 1)^*(11 + 00)(01^* + 1)^*(11 + 00)$$
$$A_3 = \varepsilon + (01^* + 1)((1 + 00)(01^* + 1)^*)(11 + 00)$$

Приклад 2: Побудова регулярного виразу за праволінійною граматикою

Розглянемо граматику $G = (\{S, X, Y\}, \{0, 1\}, P, S)$, де правила P виглядають так:

$$S \to 0X \mid 1S \mid \varepsilon$$
$$X \to 0Y \mid 1X$$
$$Y \to 0S \mid 1Y$$

Ця граматика описує мову, в якій кількість символів 0 у рядках кратна трьом [4, 5].

Система рівнянь із регулярними коефіцієнтами

Для побудови регулярного виразу за цією граматикою поставимо у відповідність нетерміналам S, X та Y невідомі регулярні вирази:

$$S \to x_1, \quad X \to x_2, \quad Y \to x_3.$$

Тоді система рівнянь з регулярними коефіцієнтами матиме вигляд:

$$\begin{cases} x_1 = 0x_2 + 1x_1 + \varepsilon \\ x_2 = 0x_3 + 1x_2 \\ x_3 = 0x_1 + 1x_3 \end{cases}$$

Розв'язання системи рівнянь

Розв'яжемо систему крок за кроком:

1. Для першого рівняння:

$$x_1 = 1x_1 + 0x_2 + \varepsilon,$$

застосовуючи формулу для розв'язку рівнянь виду $X = \alpha X + \beta$, отримуємо:

$$x_1 = 1^*(0x_2 + \varepsilon).$$

2. Для другого рівняння:

$$x_2 = 1x_2 + 0x_3$$

Аналогічно, розв'язок:

$$x_2 = 1^*0x_3.$$

3. Для третього рівняння:

$$x_3 = 1x_3 + 0x_1,$$

підставляючи x_2 у вираз для x_3, отримуємо:

$$\begin{aligned}
x_3 &= 01^*(0x_2 + \varepsilon) + 1x_3 \\
x_3 &= 01^*0x_2 + 01^* + 1x_3 \\
x_3 &= 01^*01^*0x_3 + 01^* + 1x_3 \\
x_3 &= \underbrace{(01^*01^*0 + 1^*)}_{\alpha} x_3 + \underbrace{01^*}_{\beta} \\
x_3 &= (01^*01^*0 + 1)^*01^* \\
x_2 &= 1^*0(01^*01^*0 + 1)^*01^* \\
x_1 &= 1^*(01^*0(01^*01^*0 + 1)^*01^* + \varepsilon)
\end{aligned}$$

Останній вираз для x_1 відповідає початковому нетерміналу, а отже, описує ту ж саму множину, що й граматика G.

4 Алгоритм побудови праволінійної граматики за регулярним виразом

Для побудови праволінійної граматики за регулярним виразом використовуються такі кроки [4, 5]:

1. **Для регулярного виразу** $\varnothing$ граматика описується так: $G = (\{S\}, T, \varnothing, S)$.

2. **Для регулярного виразу** ε граматика описується так: $G = (\{S\}, T, \{S \to \varepsilon\}, S)$.

3. **Для регулярного виразу** a**, де** $a \in T$ граматика описується так: $G = (\{S\}, T, \{S \to a\}, S)$.

Нехай маємо два регулярні вирази l_1 та l_2, яким відповідають граматики $G_1 = (N_1, T, P_1, S_1)$ та $G_2 = (N_2, T, P_2, S_2)$, де $N_1 \cap N_2 = \varnothing$.

4. **Для виразу** l_1+l_2: Граматика $G_3 = (N_1 \cup N_2 \cup \{S_3\}, T, P_1 \cup P_2 \cup \{S_3 \to S_1 \mid S_2\}, S_3)$.

5. **Для виразу** $l_1 l_2$: Граматика $G_4 = (N_1 \cup N_2, T, P_4, S_1)$, де:

$$P_4 : \begin{cases} \text{а) Якщо } A \to xB \in P_1, \text{ то } A \to xB \in P_4. \\ \text{б) Якщо } A \to x \in P_1, \text{ то } A \to xS_2 \in P_4. \\ \text{в) Всі правила з } P_2 \text{ належать } P_4. \end{cases}$$

6. **Для виразу** $(l_1)^*$:
Граматика $G_5 = (N_1 \cup \{S_5\}, T, P_5, S_5)$, де:

$$P_5 : \begin{cases} \text{а) } S_5 \to S_1 \mid \varepsilon \text{ належить } P_5. \\ \text{б) Якщо } A \to x \in P_1, \text{ то } \begin{cases} A \to x \in P_5, \\ A \to xS_5 \in P_5. \end{cases} \\ \text{в) Якщо } A \to xB \in P_1, \text{ то } A \to xB \in P_5. \end{cases}$$

Приклад 3. Поставити у відповідність регулярному виразу $l = (101)^*(010)^*$ праволінійну граматику.

1.
$$l_1 = 101$$
$$P_1 : S_1 \to 101$$

2.
$$l_2 = 010$$
$$P_2 : S_2 \to 010$$

3.

$$l_3 = (101)^*$$

$$\begin{aligned} P_3 : S_3 &\to S_1 \mid \varepsilon \\ S_1 &\to 101 \\ S_1 &\to 101S_3 \end{aligned}$$

4.

$$l_4 = (010)^*$$

$$\begin{aligned} P_4 : S_4 &\to S_2 \mid \varepsilon \\ S_2 &\to 010 \\ S_2 &\to 010S_4 \end{aligned}$$

5. Для виразу

$$l = (101)^*(010)^*$$

$$\begin{aligned} P_5 : S_3 &\to S_1 \mid S_4 \\ S_1 &\to 101S_3 \\ S_4 &\to S_2 \mid \varepsilon \\ S_2 &\to 010 \\ S_2 &\to 010S_4 \end{aligned}$$

Перепозначимо: $S_3 = S, S_1 = A, S_2 = B, S_4 = C$. Отже, регулярному виразу l ставиться у відповідність праволінійна граматика $G = (\{S, A, B, C\}, \{0, 1\}, P, S)$ з правилами

$$\begin{aligned} &P : \\ &S \to A \mid C \\ &A \to 101C \mid 101S \\ &C \to B \mid \varepsilon \\ &B \to 010 \mid 010C \end{aligned}$$

Контрольні запитання та завдання

1. Регулярні множини і регулярні вирази

1. Що таке регулярна множина? Як вона визначається рекурсивно?
2. Назвіть базові випадки регулярних множин.
3. Які операції можна застосовувати до регулярних множин?
4. Як визначаються регулярні вирази? Наведіть приклади.
5. Поясніть порядок пріоритетів операцій у регулярних виразах.
6. Запишіть регулярний вираз, що позначає множину всіх рядків, які складаються з 0 і 1 та закінчуються на 01.
7. Доведіть, що $p + p = p$ для регулярних виразів.
8. Що таке тотожності для регулярних виразів? Наведіть приклади.
9. Чи мова регулярна, якщо вона не може бути описана за допомогою праволінійної граматики?
10. Який алгоритм перевірки рівності двох регулярних виразів?

2. Побудова регулярного виразу за праволінійною граматикою

1. Що таке рівняння з регулярними коефіцієнтами? Як виглядає його загальна форма?
2. Як розв'язуються рівняння виду $X = \alpha X + \beta$?
3. Як визначити найменшу нерухому точку для рівняння з регулярними коефіцієнтами?

4. Як розв'язати систему рівнянь з регулярними коефіцієнтами?

5. Побудуйте регулярний вираз для наступної праволінійної граматики: $S \to aS \mid b \mid \varepsilon$.

6. Що означає "найменша нерухома точка"для системи рівнянь?

7. Поясніть кроки побудови системи рівнянь з регулярними коефіцієнтами за праволінійною граматикою.

8. Наведіть приклад граматики, що описує множину рядків, де кількість 1 парна.

9. Як розв'язати систему рівнянь для праволінійної граматики, що визначає мову
$L = \{w \mid w$ містить парну кількість символів $a\}$?

10. Чому для кожної регулярної множини можна побудувати праволінійну граматику?

3. Алгоритм побудови праволінійної граматики за регулярним виразом

1. Який вигляд має праволінійна граматика для регулярного виразу $\varnothing$?

2. Як побудувати праволінійну граматику для регулярного виразу $a, a \in T$?

3. Опишіть алгоритм побудови граматики для регулярного виразу виду $l_1 + l_2$.

4. Як побудувати граматику для регулярного виразу виду $l_1 l_2$?

5. Який вигляд має праволінійна граматика для регулярного виразу $(l_1)^*$?

6. Побудуйте граматику для регулярного виразу $(ab)^*$.

7. Чи існують різні граматики для одного і того ж регулярного виразу? Поясніть.

8. Як визначається множина нетерміналів граматики при побудові граматики для регулярного виразу?

9. Побудуйте праволінійну граматику для регулярного виразу $(01 + 10)^*$.

10. Які властивості праволінійної граматики дозволяють коректно описати регулярну мову?

Глосарій

Регулярна множина — множина слів, яка може бути описана за допомогою регулярних виразів. Визначається рекурсивно через базові випадки та операції об'єднання, конкатенації та ітерації.

Регулярний вираз — формальний запис, який описує регулярну множину. Використовується для задання шаблонів слів у формальних мовах.

Алфавіт (V) — скінченна множина символів, які використовуються для побудови слів у формальній мові.

Порожнє слово (ε) — слово нульової довжини, яке не містить жодного символу. Є елементом множини V^*.

Конкатенація — операція об'єднання двох слів. Якщо $x = ab$ і $y = cd$, то конкатенація $xy = abcd$.

Ітерація (L^*) — операція, яка формує нову множину слів, що містить всі можливі скінченні послідовності слів з множини L, включаючи порожнє слово ε.

Об'єднання ($L_1 \cup L_2$) — операція, яка формує нову множину слів, що містить усі слова з L_1 та L_2.

Праволінійна граматика — граматика, у якій всі правила мають вигляд $A \to wB$ або $A \to w$, де A, B — нетермінали, w — слово з терміналів.

Рівняння з регулярними коефіцієнтами —рівняння, у яких коефіцієнти та невідомі є регулярними виразами. Використовуються для опису регулярних множин.

Найменша нерухома точка — найменший розв'язок рівняння з регулярними коефіцієнтами. Наприклад, для рівняння $X = \alpha X + \beta$, найменша нерухома точка — $X = \alpha^*\beta$.

Система рівнянь з регулярними коефіцієнтами — набір рівнянь, у яких кожне рівняння має вигляд $X_i = \alpha_{i1}X_1 + \alpha_{i2}X_2 + \cdots + \alpha_{in}X_n + \beta_i$, де α_{ij} та β_i — регулярні вирази.

Ітерація ($(l_1)^*$) — операція, яка дозволяє повторювати слово або множину слів нуль або більше разів. Наприклад, $(ab)^*$ описує множину слів $\{\varepsilon, ab, abab, ababab, \dots\}$.

Праволінійна граматика за регулярним виразом — алгоритм побудови граматики, яка описує мову, задану регулярним виразом. Використовує правила вигляду $A \to wB$ або $A \to w$.

Регулярна мова — мова, яка може бути описана за допомогою регулярного виразу, праволінійної граматики або скінченного автомата.

Тотожності для регулярних виразів — набір правил, які дозволяють спрощувати регулярні вирази. Наприклад, $p + p = p$, $p\varnothing = \varnothing$, $p^* = p^*p^*$.

Порожня множина ($\varnothing$) —множина, яка не містить жодного слова. Є регулярною множиною.

Мова (L) — множина слів, які належать до певної формальної мови. Наприклад, мова $L = \{a^n \mid n \geq 0\}$ над алфавітом $V = \{a\}$.

Нетермінал — символ, який використовується у граматиці для позначення понять. Наприклад, S, A, B.

Термінал — символ, який є частиною алфавіту мови. Наприклад, a, b, 0, 1.

Правило виведення (P) — правило, яке описує, як нетермінали можуть бути замінені на інші нетермінали або термінали. Наприклад, $S \to aB$.

Початковий нетермінал (S) —нетермінал, з якого починається виведення слів у граматиці.

Система рівнянь для граматики — набір рівнянь, які описують залежності між нетерміналами граматики. Використовується для побудови регулярних виразів за граматикою.

Регулярний вираз за граматикою — вираз, який описує мову, задану граматикою. Отримується шляхом розв'язання системи рівнянь з регулярними коефіцієнтами.

Алгоритм побудови граматики — послідовність кроків для створення праволінійної граматики за заданим регулярним виразом.

Тема IV. Скінченні автомати

1 Основні поняття та визначення

Визначення 1. Недетермінований скінченний автомат M описується п'ятіркою:

$$M = (Q, \Sigma, \delta, q_0, F),$$

де:

- Q — скінченна множина станів, $Q = \{q_0, q_1, \ldots, q_n\}$;
- Σ — скінченна множина вхідних символів, $\Sigma = \{a_1, a_2, \ldots, a_m\}$;
- δ — функція переходів, $\delta : Q \times \Sigma \to P(Q)$;
- $q_0 \in Q$ — початковий стан;
- $F \subseteq Q$ — множина заключних станів.

На початку роботи автомата вхідна стрічка містить вхідне слово, яке потрібно обробити. Автомат виконує послідовність кроків, які залежать від поточного стану та вхідного символу. Кожен крок обчислювального процесу визначається наступним чином:

1. Автомат знаходиться у певному **стані** $q \in Q$, який описує його поточний режим роботи.
2. Автомат отримує **вхідний символ** $a \in \Sigma$ з вхідного словника або алфавіту.
3. Залежно від поточного стану та вхідного символу, автомат переходить до нового **стану**. Цей стан визначається функцією переходів δ(q, a).

Цей процес повторюється для кожного вхідного символу, поки не буде вичерпано всі вхідні дані. Якщо в кінці роботи автомат буде знаходитись в заключному стані, це означає, що автомат прийняв вхідне слово [6, 7].

Означення 2. Конфігурація автомата — це пара (q, ω), де $q \in Q$ — поточний стан, а ω — вхідний ланцюжок.

Конфігурація (q_0, ω) називається **початковою**, де q_0 — початковий стан, а ω — вхідний ланцюжок. Конфігурація (q, ε), де $q \in F$, називається **заключною**, де F — множина заключних станів [6].

Автомат переходить з конфігурації $(q, a\omega)$ у конфігурацію (q', ω), якщо q' належить множині станів, до яких можна перейти зі стану q за символом a, тобто $q' \in \delta(q, a)$. Цей перехід позначається так:

$$(q, a\omega) \to (q', \omega).$$

Робота скінченого автомата описується як послідовність конфігурацій. Нехай $K_0, K_1, \ldots, K_p$ — деякі конфігурації автомата. Якщо існує послідовність переходів:

$$K_0 \mapsto K_1 \mapsto \cdots \mapsto K_p,$$

то це позначається так:

$$K_0 \overset{*}{\mapsto} K_p \quad \text{або} \quad K_0 \overset{p}{\mapsto} K_p,$$

де $\overset{*}{\mapsto}$ означає перехід за будь-яку кількість кроків, а $\overset{p}{\mapsto}$ — перехід за p кроків [6].

Означення 3. Автомат M допускає ланцюжок ω, якщо існує послідовність конфігурацій, що перетворює початкову конфігурацію (q_0, ω) у заключну (q, ε), де $q \in F$ [6, 7].

Означення 4. Мова, яку визначає автомат M, — це множина всіх ланцюжків, які допускаються автоматом:

$$L(M) = \{\omega \in \Sigma^* \mid (q_0, \omega) \mapsto^* (q, \varepsilon), q \in F\}.$$

Означення 5. Автомат M називається **недетермінованим (НСА)**, якщо для деяких $q \in Q$ та $a \in \Sigma$, функція переходів $\delta(q, a)$ містить більше одного стану [6].

Означення 6. Автомат M називається **детермінованим (ДСА)**, якщо для кожних $q \in Q$ та $a \in \Sigma$, функція переходів $\delta(q, a)$ містить не більше одного стану [6].

Означення 7. Автомат M називається **повністю визначеним**, якщо для кожних $q \in Q$ та $a \in \Sigma$, функція переходів $\delta(q, a)$ містить один стан [6].

1.1 Приклад детермінованого автомата

Задача 1. Побудувати автомат, який розпізнає ланцюжки з двома символами x підряд у алфавіті $\Sigma = \{x, y\}$.

Автомат M визначається як

$$M = (\{q_0, q_1, q_2\}, \{x, y\}, \delta, q_0, \{q_2\}),$$

де

- q_0 — початковий стан (два символи x підряд ще не зустрічалися);
- q_1 — стан, де останній символ — x;
- q_2 — заключний стан (два символи x підряд зустрілися).

Функція переходів δ:

$$\delta(q_0, x) = q_1, \quad \delta(q_0, y) = q_0, \quad \delta(q_1, x) = q_2, \quad \delta(q_1, y) = q_0,$$
$$\delta(q_2, x) = q_2, \quad \delta(q_2, y) = q_2.$$

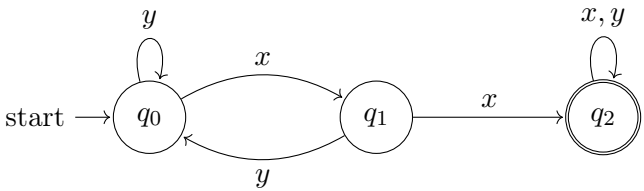
Таблиця переходів:

State	x	y	Final
q_0	q_1	q_0	false
q_1	q_2	q_0	false
q_2	q_2	q_2	true

Автомат **детермінований** і **повністю визначений**, оскільки для кожного стану $q \in Q$ та кожного символу $a \in \Sigma$ функція переходів $\delta(q, a)$ містить рівно один стан. Це означає, що

для будь-якого стану та вхідного символу існує лише один можливий перехід [6].

Діаграма переходів автомата



Приклад роботи автомата

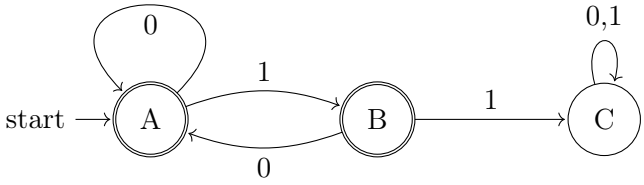
Розглянемо приклад роботи автомата на ланцюжку $xyxxx$. Нехай автомат знаходиться у початковому стані q_0. Послідовність конфігурацій буде такою:

$$(q_0, xyxxx) \mapsto (q_1, yxxx) \mapsto (q_0, xxx)$$

$$\mapsto (q_1, xx) \mapsto (q_2, x) \mapsto (q_2, e).$$

Ця послідовність показує, як автомат переходить між станами під час обробки ланцюжка. У результаті автомат досягає заключного стану q_2, що означає, що ланцюжок $xyxxx$ належить мові, яку розпізнає автомат [6].

Задача 2. Побудувати автомат, який розпізнає ланцюжки без послідовності 11 у алфавіті $\Sigma = \{0, 1\}$.



Пояснення:

- У стані A рядок до цього часу не містить 11 і не закінчується на 1.
- У стані B рядок до цього часу не містить 11, але закінчується на одну 1.
- У стані C було побачено послідовність з двох одиниць (11). Це означає, що рядок мові автомата не належить.

1.2 Приклад недетермінованого автомата

Задача 3. Побудувати недетермінований автомат, який розпізнає ланцюжки, де останній символ зустрічався раніше в алфавіті $\{a, b, c\}$.

Автомат M визначається як

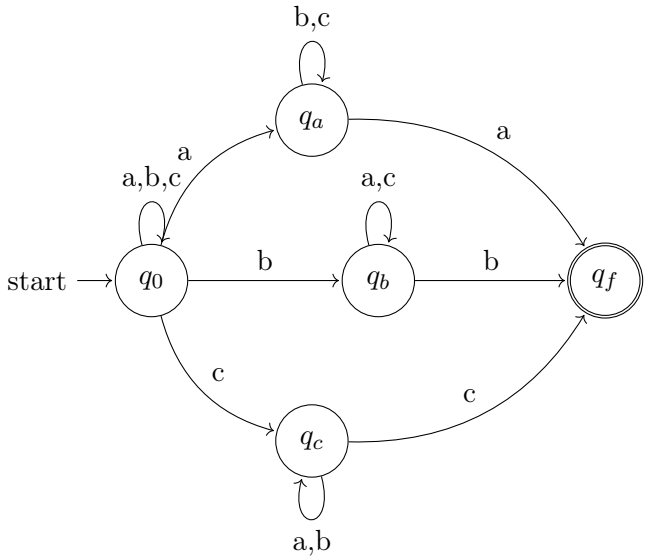
$$M = (\{q_0, q_a, q_b, q_c, q_f\}, \{a, b, c\}, \delta, q_0, \{q_f\}),$$

де

- q_0 — початковий стан;
- q_a, q_b, q_c — стани, де відповідний символ (a, b, c) вже зустрічався;
- q_f — заключний стан.

Функція переходів δ:

State	a	b	c	Final
q_0	$\{q_0, q_a\}$	$\{q_0, q_b\}$	$\{q_0, q_c\}$	false
q_a	$\{q_a, q_f\}$	q_a	q_a	false
q_b	q_b	$\{q_b, q_f\}$	q_b	false
q_c	q_c	q_c	$\{q_c, q_f\}$	false
q_f	$\varnothing$	$\varnothing$	$\varnothing$	true



Аналіз ланцюжка

Розглянемо ланцюжок *abcba*. Проаналізуємо його обробку автоматом:

$$
\begin{aligned}
(q_0, abcba) &\mapsto (q_0, bcba) \mapsto (q_0, cba) \mapsto (q_0, ba) \mapsto (q_0, a) \mapsto (q_0, \varepsilon) \\
&\mapsto (q_a, bcba) \mapsto (q_a, cba) \mapsto (q_a, ba) \mapsto (q_a, a) \mapsto (q_a, \varepsilon) \\
&\mapsto (q_f, \varepsilon) \\
(q_0, abcba) &\mapsto (q_0, bcba) \mapsto (q_b, cba) \mapsto (q_b, ba) \mapsto (q_b, a) \mapsto (q_b, \varepsilon) \\
&\mapsto (q_f, a) \\
(q_0, abcba) &\mapsto (q_0, bcba) \mapsto (q_0, cba) \mapsto (q_c, ba) \mapsto (q_c, a) \mapsto (q_c, \varepsilon) \\
(q_0, abcba) &\mapsto (q_0, bcba) \mapsto (q_0, cba) \mapsto (q_b, ba) \mapsto (q_b, a) \mapsto (q_b, \varepsilon) \\
(q_0, abcba) &\mapsto (q_0, bcba) \mapsto (q_0, cba) \mapsto (q_0, ba) \mapsto (q_0, a) \mapsto (q_a, \varepsilon)
\end{aligned}
$$

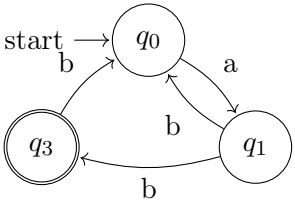
Під час обробки ланцюжка *abcba* знайшлася лише одна альтернатива переходу в заключну конфігурацію, але ланцюжок все одно допустимий, оскільки автомат досягає заключного стану q_f, обробивши весь ланцюжок [6].

2 Перетворення недетермінованого автомата на детермінований

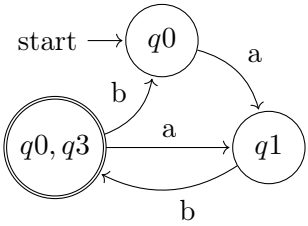
Розглянемо недетермінований та детермінований скінченні автомати, які розпізнають ланцюжки з символів a, b, які містять тільки фрагменти "ab", "abb" будь-яку кількість разів, в будь-якому порядку.

Для такої задачі побудувати такий недетермінований автомат досить просто, а детермінований – набагато складніше.

Як бачимо, зображений ніжче автомат є недетермінованим, оскільки з стану q_1 виходить два ребра, навантажені символом b.



Перетворимо його в детермінований скінченний автомат, об'єднавши стани q_0 і q_3 в один стан і, таким чином, зробивши так, що при переході зі стану q_0 по символу b вже немає неоднозначності:



У цьому автоматі з кожного стану по кожному символу є не більше одного переходу. Отже, автомат детермінований.

Зазначимо, що програмувати ДСА і працювати з ДСА набагато простіше, ніж з НСА [6].

Теорема. Будь-який недетермінований скінченний автомат можна перетворити на еквівалентний детермінований скінченний автомат [6].

Алгоритм перетворення:

1. Побудова множини станів Q' для детермінованого автомата M'. Кожен стан Q' — це підмножина станів Q недетермінованого автомата.

2. Початковий стан $q'_0 = \{q_0\}$.

3. Множина заключних станів F' містить усі стани Q', які містять хоча б один заключний стан з F.

4. Функція переходів δ' визначається як:

$$\delta'(S, a) = \bigcup_{q \in S} \delta(q, a),$$

де $S \subseteq Q$.

3 Приклад детермінізації методом побудови тільки досяжних станів

Розглянемо недетермінований автомат M з задачі 2, який розпізнає ланцюжки, де останній символ зустрічався раніше в алфавіті $\{a, b, c\}$. Автомат M має стани $Q = \{q_0, q_a, q_b, q_c, q_f\}$, де q_f — заключний стан.

Побудуємо детермінований автомат

$$M' = (Q', \{a, b, c\}, \delta', q'_0, F').$$

1. **Множина станів** Q': Оскільки Q має 5 станів, Q' може містити до $2^5 = 32$ станів. Однак ми розглядаємо лише досяжні стани:

$$Q' = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, P\},$$

де

- $A = \{q_0\}$
- $B = \{q_0, q_a\}$
- $C = \{q_0, q_b\}$
- $D = \{q_0, q_c\}$
- $E = \{q_0, q_a, q_f\}$
- $F = \{q_0, q_a, q_b\}$
- $G = \{q_0, q_a, q_c\}$
- $H = \{q_0, q_b, q_f\}$
- $I = \{q_0, q_b, q_c\}$
- $J = \{q_0, q_c, q_f\}$
- $K = \{q_0, q_a, q_b, q_f\}$
- $L = \{q_0, q_a, q_b, q_c\}$
- $M = \{q_0, q_a, q_c, q_f\}$
- $N = \{q_0, q_b, q_c, q_f\}$
- $P = \{q_0, q_a, q_b, q_c, q_f\}$

2. **Початковий стан** q_0': $q_0' = A = \{q_0\}$.

3. **Множина заключних станів** F': F' має всі стани Q', які включають q_f:

$$F' = \{E, H, J, K, M, N, P\}.$$

4. **Функція переходів** δ': Наприклад:

$$\delta'(A, a) = B, \quad \delta'(A, b) = C, \quad \delta'(A, c) = D.$$

Аналогічно визначаються переходи для інших станів.

Таблиця переходів для M'

State	a	b	c	Final
$A = \{q_0\}$	B	C	D	false
$B = \{q_0, q_a\}$	E	F	G	false
$C = \{q_0, q_b\}$	F	H	I	false
$D = \{q_0, q_c\}$	G	I	J	false
$E = \{q_0, q_a, q_f\}$	E	F	G	true
$F = \{q_0, q_a, q_b\}$	K	K	L	false
$G = \{q_0, q_a, q_c\}$	M	L	M	false
$H = \{q_0, q_b, q_f\}$	F	H	I	true
$I = \{q_0, q_b, q_c\}$	L	N	N	true
$J = \{q_0, q_c, q_f\}$	G	I	J	true
$K = \{q_0, q_a, q_b, q_f\}$	K	K	L	true
$L = \{q_0, q_a, q_b, q_c\}$	P	P	P	false
$M = \{q_0, q_a, q_c, q_f\}$	M	L	M	true
$N = \{q_0, q_b, q_c, q_f\}$	L	N	N	true
$P = \{q_0, q_a, q_b, q_c, q_f\}$	P	P	P	true

Після перетворення недетермінованого автомата на детермінований, кількість станів значно збільшилася (з 5 до 15). Однак, детермінований автомат має чітко визначені переходи, що полегшує його аналіз та використання [6].

4 Приклад детермінізації з можливою появою недосяжних станів

Процес перетворення недетермінованого скінченного автомата в детермінований скінченний автомат включає такі кроки:

Визначення множини станів

Для кожного стану НСА ми створюємо множину станів у ДСА. Наприклад, якщо НСА має стани A, B, C, тоді у ДСА можуть бути множини станів, такі як $\{A\}$, $\{B\}$, $\{A, B\}$, $\{A, C\}$ тощо. Кожен стан детермінованого автомату можна позначити одною літерою. Отже, стани автомату, які позначають множину більше ніж з одного символа, будуть новими станами детермінованого скінченного автомату.

Для нашого прикладу множина станів детермінованого скінченного автомату буде такою

$$Q' = \{A, B, D, E\},$$

де

$$D = \{A, B\}, \quad E = \{A, C\}.$$

Визначення початкового та заключного станів

Початковий стан ДСА — це множина, що містить початковий стан НСА. Заключний стан ДСА — це будь-яка множина, що містить хоча б один заключний стан НСА.

Нехай задано такий НСА:

	0	1
S	$\{C\}$	$\{A, E\}$
(A)	$\{B\}$	$\varnothing$
B	$\varnothing$	A
C	$\varnothing$	D
D	C	E
E	G	$\varnothing$
(G)	$\varnothing$	$\varnothing$

Побудуємо детермінований автомат

$$M' = (Q', \{0, 1\}, \delta', q_0', F').$$

Створюємо таблицю переходів для ДСА δ', базуючись на переходах НСА.

	0	1
S	C	$M = \{A, E\}$
(M)	$N = \{B, G\}$	$\varnothing$
(N)	$\varnothing$	A
(A)	B	$\varnothing$
B	$\varnothing$	A
C	$\varnothing$	D
D	C	E
E	G	$\varnothing$
(G)	$\varnothing$	$\varnothing$

Як бачимо, у стовпчику станів заключні стани позначені круглими дужками.

Тут

$$Q' = \{S, M, N, A, B, C, D, E, G\},$$

$$F = \{M, N, A, G\} \quad q'_0 = S.$$

Як бачимо, з'явилися нові два стани – M і N.

У загальному випадку при такому підході можлива поява недосяжних станів. У подальшому розглянемо алгоритм вилучення недосяжних станів.

5 Алгоритм роботи повністю визначеного автомату

Цей алгоритм описує роботу детермінованого скінченного автомату (ДСА), який перевіряє, чи належить вхідний рядок мові, що розпізнається автоматом [6].

Algorithm 1 Алгоритм роботи ДСА

```
1: q ← q_0                                   ▷ Початковий стан автомата
2: a ← GetChar() ▷ Зчитування першого символу з вхідного рядка
3: while a ≠ eof do                          ▷ Поки не кінець рядка
4:     q ← Delta(q, a)    ▷ Перехід у новий стан за функцією переходів Delta
5:     a ← GetChar()          ▷ Зчитування наступного символу
6: end while
7: if q ∈ F then              ▷ Якщо автомат у заключному стані
8:     return "так"             ▷ Рядок належить мові автомата
9: else
10:    return "ні"              ▷ Рядок не належить мові автомата
11: end if
```

Пояснення:

- **Початковий стан (q_0)**: Автомат починає роботу зі стартового стану.
- **Функція переходів (Delta)**: Визначає, у який стан перейти, залежно від поточного стану та вхідного символу.
- **Заключні стани (F)**: Якщо після обробки всього рядка автомат опиняється в одному з заключних станів, рядок належить мові автомата.

Контрольні запитання та завдання

Основні поняття та визначення

1. Що таке скінченний автомат? Які його основні компоненти?
2. Що таке конфігурація скінченного автомата? Як позначається початкова та заключна конфігурація?

3. Як визначається мова, що допускається скінченним автоматом?

4. У чому відмінність детермінованого скінченного автомата (ДСА) від недетермінованого (НСА)?

5. Що означає, що автомат є «повністю визначеним»?

Перетворення НСА на ДСА

1. У чому полягає основна ідея перетворення НСА на ДСА?

2. Як визначаються стани Q' детермінованого автомата?

3. Яке правило використовується для визначення початкового та заключних станів у ДСА?

4. Чи завжди можна перетворити НСА на еквівалентний ДСА? Чому?

5. Які переваги має ДСА порівняно з НСА?

Глосарій

Скінченний автомат — математична модель, яка описується п'ятіркою $M = (Q, \Sigma, \delta, q_0, F)$, де:

- Q — скінченна множина станів;
- Σ — скінченна множина вхідних символів;
- δ — функція переходів;
- q_0 — початковий стан;
- F — множина заключних станів.

Конфігурація автомата — пара (q, ω), де $q \in Q$ — поточний стан, а ω — вхідний ланцюжок.

Початкова конфігурація — конфігурація (q_0, ω), де q_0 — початковий стан, а ω — вхідний ланцюжок.

Заключна конфігурація — конфігурація (q, ε), де $q \in F$ — заключний стан, а ε — порожній ланцюжок.

Мова автомата — множина всіх ланцюжків, які допускаються автоматом:

$$L(M) = \{\omega \in \Sigma^* \mid (q_0, \omega) \mapsto^* (q, \varepsilon), q \in F\}.$$

Детермінований скінченний автомат (ДСА) — автомат, у якого для кожного стану $q \in Q$ та символу $a \in \Sigma$ функція переходів $\delta(q, a)$ містить не більше одного стану.

Недетермінований скінченний автомат (НСА) — автомат, у якого для деяких станів $q \in Q$ та символів $a \in \Sigma$ функція переходів $\delta(q, a)$ містить більше одного стану.

Повністю визначений автомат — автомат, у якого для кожного стану $q \in Q$ та символу $a \in \Sigma$ функція переходів $\delta(q, a)$ містить рівно один стан.

Функція переходів (δ) — функція, яка визначає, у який стан перейде автомат залежно від поточного стану та вхідного символу.

Початковий стан (q_0) — стан, з якого починається робота автомата.

Заключний стан (F) — множина станів, у яких автомат завершує роботу, якщо вхідний ланцюжок оброблено повністю.

Перетворення НСА на ДСА — процес побудови детермінованого автомата, який розпізнає ту саму мову, що й недетермінований автомат.

Множина станів Q' — у детермінованому автоматі кожен стан є підмножиною станів недетермінованого автомата.

Алгоритм роботи ДСА — послідовність кроків, які виконує детермінований автомат для обробки вхідного ланцюжка та визначення, чи належить він мові автомата.

Недосяжні стани — стани детермінованого автомата, які не можуть бути досягнуті з початкового стану за будь-якої послідовності переходів.

Таблиця переходів — таблиця, яка описує функцію переходів автомата. Кожен рядок відповідає стану, а кожен стовпець — вхідному символу.

Діаграма переходів — графічне зображення автомата, де стани зображуються вузлами, а переходи — стрілками, позначеними вхідними символами.

Регулярна мова — мова, яка може бути описана за допомогою скінченного автомата, регулярного виразу або праволінійної граматики.

Алгоритм детермінізації — алгоритм, який перетворює недетермінований автомат на детермінований шляхом побудови множини станів та функції переходів.

Підмножина станів — у контексті детермінізації, кожен стан детермінованого автомата є підмножиною станів недетермінованого автомата.

Еквівалентність автоматів — два автомати вважаються еквівалентними, якщо вони розпізнають одну й ту саму мову.

Тема V. Мінімізація скінченного автомата

1 Основні означення

Мінімізація скінченних автоматів передбачає побудову найменшого еквівалентного автомата шляхом видалення недосяжних станів і об'єднання еквівалентних [2,4].

Означення 1. Послідовність символів $x \in \Sigma^*$ відрізняє стани q_1 та q_2, якщо з q_1 по x можна перейти до стану q_3, а з q_2 по x — до стану q_4, причому q_3 і q_4 належать до різних множин F та $Q \setminus F$.

Означення 2. Два стани q_1 і q_2 вважаються нерозрізнюваними, якщо не існує жодної послідовності символів $x \in \Sigma^*$, яка б змогла їх розрізнити.

Означення 3. Якщо для стану q немає вхідного ланцюжка $x \in \Sigma^*$, який переводить початковий стан q_0 в q, то цей стан вважається недосяжним.

Означення 4. Автомат називається мінімальним, якщо:

1) у ньому відсутні недосяжні стани;

2) будь-які два різні стани розрізнювані.

2 Алгоритм видалення недосяжних станів

Щоб визначити всі досяжні стани скінченного автомата, розглянемо його як граф, де вершини відповідають станам, а ребра — переходам:

1. Ініціалізуємо список L, додаючи до нього початковий стан q_0, і позначаємо його як оброблений.

2. Поки список L не порожній:

   - вилучаємо перший елемент зі списку L;

- додаємо до L усі суміжні стани, які ще не були оброблені, та позначаємо їх.

3. Після завершення всі позначені стани є досяжними, а решта — недосяжними і видаляються.

3 Мінімізація за допомогою класів еквівалентності

3.1 Побудова відношення еквівалентності

Означення

Нехай L — деяка мова над алфавітом V. Відношення еквівалентності $\equiv^k$ для цієї мови визначається таким чином: два слова u і v з V^* називаються еквівалентними за відношенням $\equiv^k$, якщо для будь-якого слова $w \in V^*$ довжиною не більше k маємо:

$$uw \in L \iff vw \in L$$

Це означає, що слова u і v не можна розрізнити за допомогою контексту довжиною не більше k.

Класи еквівалентності

Кожне відношення еквівалентності $\equiv^k$ породжує розбиття множини всіх слів V^* на класи еквівалентності. Клас еквівалентності слова u позначається $[u]_{\equiv^k}$ і складається з усіх слів v, які еквівалентні u за відношенням $\equiv^k$:

$$[u]_{\equiv^k} = \{v \in V^* \mid v \equiv^k u\}$$

Алгоритм побудови

Для побудови відношення еквівалентності $\equiv^k$ можна використовувати такий алгоритм:

1. **Ініціалізація**: Починаємо з відношення еквівалентності $\equiv^0$, яке розрізняє слова лише за тим, чи належать вони до мови L чи ні. Тобто, два слова u і v є еквівалентними за відношенням $\equiv^0$, якщо $u \in L \iff v \in L$.

2. **Ітераційний процес**: Для кожного i від 0 до $k-1$ будуємо відношення еквівалентності $\equiv^{i+1}$ на основі $\equiv^i$:

   - Розглядаємо всі пари слів (u, v) такі, що $u \equiv^i v$.
   - Перевіряємо, чи для всіх слів w довжиною 1 маємо $uw \equiv^i vw$. Якщо так, то $u \equiv^{i+1} v$; інакше u і v розрізняються за відношенням $\equiv^{i+1}$.

3. **Завершення**: Після завершення k-ї ітерації отримуємо відношення еквівалентності $\equiv^k$.

3.2 Алгоритм знаходження класів еквівалентності ДСА

Цей метод полягає у покроковому уточненні класів еквівалентності:

1. Початково розбиваємо множину станів Q на два класи: заключні F і незаключні $Q \setminus F$.

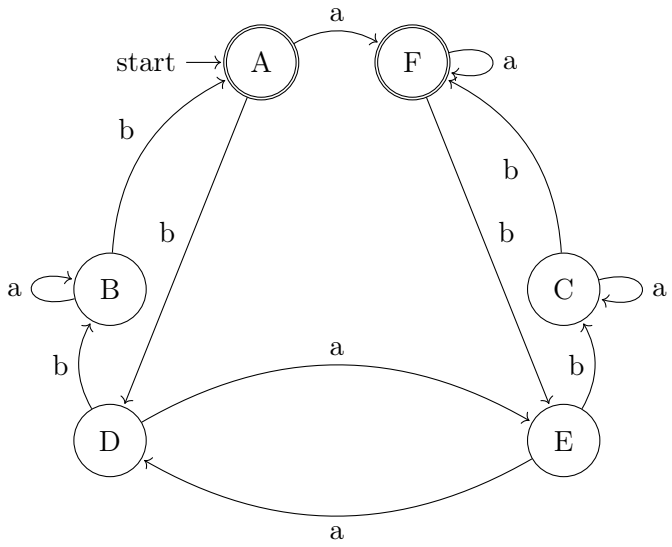
2. У кожному кроці перевіряємо, чи ланцюжки певної довжини k розрізняють стани. Якщо стани розрізняються, вони належать до різних класів.

3. Процес триває, доки класи еквівалентності не перестануть змінюватися.

Теорема. Мінімізований автомат, побудований цим методом, має найменшу кількість станів серед усіх еквівалентних автоматів.

3.3 Приклад 1 мінімізації скінченного автомата за допомогою класів еквівалентності

Розглянемо повністю визначений автомат, який розпізнає ланцюжки з кількістю символів b, кратних 3 [2, 4].



State	a	b	Final
A	F	D	true
B	B	A	false
C	C	F	false
D	E	B	true
E	D	C	false
F	F	E	false

Крок 1: Визначення недосяжних станів

У даному автоматі відсутні недосяжні стани.

Крок 2: Побудова відношення еквівалентності $\equiv^0$

Розділяємо стани на два класи еквівалентності:

$$K_1 = \{A, F\}, \quad K_2 = \{B, C, D, E\}.$$

Перший клас еквівалентності містить заключні стани автомату. А другий - незаключні.

Крок 3: Побудова відношення еквівалентності $\equiv^1$

а) Аналіз класу K_1

Для класу $K_1 = \{A, F\}$ виконуємо перевірку:

$$\begin{aligned} &\delta(A, a) = F, \quad \delta(F, a) = F \\ &\delta(A, b) = D, \quad \delta(F, b) = E \\ &A \equiv^1 F \end{aligned}$$

Символ a не розрізняє стани класу K_1, аналогічно символ b також не розрізняє ці стани.

б) Аналіз класу K_2

Для класу $K_2 = \{B, C, D, E\}$ виконуємо перевірку:

$$\begin{aligned} &\delta(B, a) = B, \quad \delta(C, a) = C, \quad \delta(D, a) = E, \quad \delta(E, a) = D \\ &\delta(B, b) = A, \quad \delta(C, b) = F, \quad \delta(D, b) = B, \quad \delta(E, b) = C \end{aligned}$$

Символ a не розрізняє стани класу K_2, але символ b розрізняє їх. В результаті клас K_2 розділяється на два нові класи:

$$K_3 = \{B, C\}, \quad K_4 = \{D, E\}.$$

Отже, $B \equiv^1 C$ та $D \equiv^1 E$.

Крок 4: Побудова відношення еквівалентності $\equiv^2$

а) Аналіз класу K_1

Для класу $K_1 = \{A, F\}$ виконуємо перевірку:

$$\delta(A, a) = F, \quad \delta(F, a) = F, \quad \delta(A, b) = D, \quad \delta(F, b) = E$$

б) Аналіз класів K_3 та K_4

Для класу $K_3 = \{B, C\}$:

$$\delta(B, a) = B, \quad \delta(C, a) = C, \quad \delta(B, b) = A, \quad \delta(C, b) = F$$

Для класу $K_4 = \{D, E\}$:

$$\delta(D, a) = E, \quad \delta(E, a) = D, \quad \delta(D, b) = B, \quad \delta(E, b) = C$$

Класи K_3 та K_4 не розділилися. Отже, отримуємо наступні класи еквівалентності:

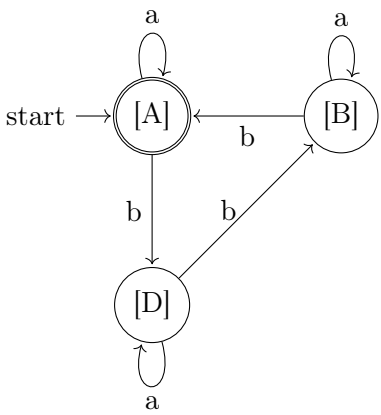
$$\underbrace{K_1 = \{A, F\}}_{[A]}, \quad \underbrace{K_3 = \{B, C\}}_{[B]}, \quad \underbrace{K_4 = \{D, E\}}_{[D]}.$$

Класи залишилися незмінними, що свідчить про завершення мінімізації. Подальше розділення класів K_1, K_3, K_4 неможливе.

Результат мінімізації

Після завершення мінімізації отримуємо мінімізований автомат з трьома класами еквівалентності:

$$[A] = \{A, F\}, \quad [B] = \{B, C\}, \quad [D] = \{D, E\}.$$

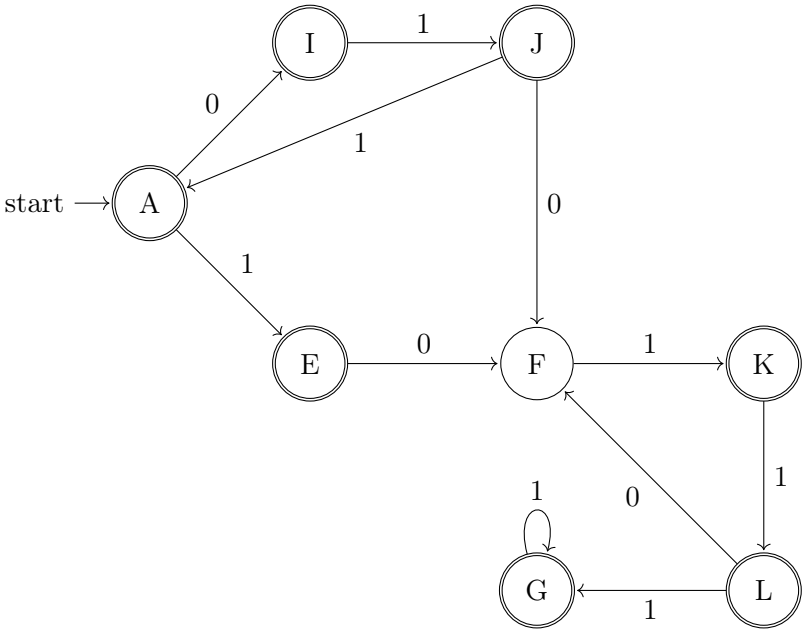


State	**a**	**b**	Final
[A]	[A]	[D]	true
[B]	[B]	[A]	false
[D]	[D]	[B]	false

3.4 Приклад 2 мінімізації скінченного автомата за допомогою класів еквівалентності

Тепер покажемо простіший спосіб реалізації цього методу мінімізації. На кожному кроці будемо заповнювати табліцю переходів автомату класами еквівалентності.

Розглянемо детермінований скінченний автомат:



Крок 1: Визначення недосяжних станів

У даному автоматі відсутні недосяжні стани.

Крок 2: Побудова відношення еквівалентності $\equiv^0$ (початковий розподіл)

На початку всі стани діляться на дві групи: незаключні та заключні стани.

$$\equiv^0:$$

$$K_1^0 = \{A, I, J, E, K, L, G\}$$

$$K_2^0 = \{F\}$$

Таблиця переходів:

	0	1
A	K_1^0	K_1^0
I	$-$	K_1^0
J	K_2^0	K_1^0
E	K_2^0	$-$
F	$-$	K_1^0
K	$-$	K_1^0
L	K_2^0	K_1^0
G	$-$	K_1^0

У цій таблиці перехід у відповідний стан замінений переходом у клас еквівалентності, в якому знаходиться цей стан.

Крок 3: Побудова відношення еквівалентності $\equiv^1$ (перша ітерація)

Після першого проходу по таблиці переходів, розподіл виглядає так:

$$\equiv^1:$$

$$K_1^1 = \{A\}$$

$$K_2^1 = \{I, K, G\}$$

$$K_3^1 = \{J, L\}$$

$$K_4^1 = \{F\}$$

$$K_5^1 = \{E\}$$

Тут ми групуємо стани, в яких є переходи в однакові класи еквівалентності (тобто, рядки таблиці співпадають). При цьому в один клас попадають лише ті стани, які на попередньому кроці не належали різним класам еквівалентності. Класи еквівалентності на кожному кроці можуть лише розпадатися на дрібніші і не можуть об'єднуватись.

Таблиця переходів:

	0	1
A	K_2^1	K_5^1
I	–	K_3^1
J	K_4^1	K_1^1
E	K_4^1	–
F	–	K_2^1
K	–	K_3^1
L	K_4^1	K_2^1
G	–	K_2^1

Крок 4: Побудова відношення еквівалентності $\equiv^2$ (друга ітерація)

Після другого проходу, розподіл змінюється:

$$\equiv^2:$$

$$K_1^2 = \{A\}$$

$$K_2^2 = \{I, K\}$$

$$K_3^2 = \{J\}$$

$$K_4^2 = \{F\}$$

$$K_5^2 = \{E\}$$

$$K_6^2 = \{L\}$$

$$K_7^2 = \{G\}$$

Таблиця переходів:

	0	1
A	K_2^2	K_5^2
I	$-$	K_3^2
J	K_4^2	K_1^2
E	K_4^2	$-$
F	$-$	K_2^2
K	$-$	K_6^2
L	K_4^2	K_7^2
G	$-$	K_7^2

Крок 5: Побудова відношення еквівалентності $\equiv^3$ (третя ітерація)

На останньому етапі отримуємо остаточний розподіл:

$$\equiv^3:$$

$$K_1^3 = \{A\}$$

$$K_2^3 = \{K\}$$

$$K_3^3 = \{J\}$$

$$K_4^3 = \{F\}$$

$$K_5^3 = \{E\}$$

$$K_6^3 = \{L\}$$

$$K_7^3 = \{G\}$$

$$K_8^3 = \{I\}$$

Таблиця переходів:

	0	1
A	K_2^3	K_5^3
I	–	K_3^3
J	K_4^3	K_1^3
E	K_4^3	–
F	–	K_2^3
K	–	K_6^3
L	K_4^3	K_7^3
G	–	K_7^3

Результат мінімізації

Після трьох ітерацій видно, що кожен клас еквівалентності містить один єдиний стан автомату. Це означає, що кількість станів неможливо зменшити, тобто вихідний автомат вже був мінімізований.

4 Функція переходів і узагальнена функція переходів

Розширена (узагальнена) функція переходів $\hat{\delta}\,(q_0, \omega)$ визначає множину станів, до яких автомат може перейти після обробки ланцюжка ω, починаючи зі стану q_0 [2, 4].

Рекурсивний алгоритм побудови розширеної функції переходів

Розширена функція переходів $\hat{\delta}$ визначається так:
1) Для порожнього ланцюжка ε:

$$\hat{\delta}(q, \varepsilon) = \{q\}$$

2) Для ланцюжка xc, де $x \in \Sigma^*$ та $c \in \Sigma$:

$$\hat{\delta}(q, xc) = \delta(\hat{\delta}(q, x), c)$$

Це означає, що для знаходження $\hat{\delta}(q, \omega)$ спочатку обчислюється $\hat{\delta}(q, x)$, а потім виконуються всі переходи з отриманих станів за символом c.

Більш детально, якщо $\omega = xc$, де $x \in \Sigma^*$ та $c \in \Sigma$, і відомо, що:

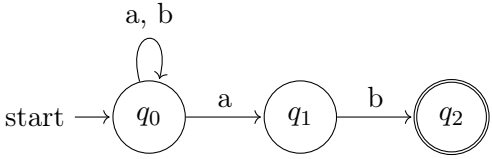
$$\hat{\delta}(q, x) = \{p_1, p_2, \ldots, p_n\},$$

то:

$$\hat{\delta}(q, \omega) = \bigcup_{i=1}^{n} \delta(p_i, c).$$

4.1 Розширена функція переходів для НСА

Розглянемо скінченний автомат, який розпізнає ланцюжки, що закінчуються на "ab". Автомат має три стани: q_0, q_1 та q_2, де q_0 є початковим станом, а q_2 — заключним.



Нехай автомат має обробити ланцюжок "aabab". Під час обробки можливі такі альтернативи:

$$\hat{\delta}\,(q_0, aab) = \{q_0, q_2\}$$
$$\hat{\delta}\,(q_0, aa) = \{q_0, q_1\}$$

Застосування алгоритму до ланцюжка $\omega = aabab$

Побудуємо розширену функцію переходів для ланцюжка "aabab":

1) Для порожнього ланцюжка:

$$\hat{\delta}(q_0, \varepsilon) = \{q_0\}$$

2) Для ланцюжка "a":

$$\hat{\delta}(q_0, a) = \delta(q_0, a) = \{q_0, q_1\}$$

3) Для ланцюжка "aa":

$$\hat{\delta}(q_0, aa) = \delta(q_0, a) \cup \delta(q_1, a) = \{q_0, q_1\} \cup \varnothing = \{q_0, q_1\}$$

4) Для ланцюжка "aab":

$$\hat{\delta}(q_0, aab) = \delta(q_0, b) \cup \delta(q_1, b) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$$

5) Для ланцюжка "aaba":

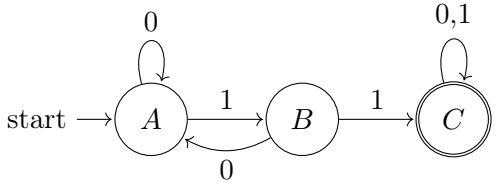
$$\hat{\delta}(q_0, aaba) = \delta(q_0, a) \cup \delta(q_2, a) = \{q_0, q_1\} \cup \varnothing = \{q_0, q_1\}$$

6) Для ланцюжка "aabab":

$$\hat{\delta}(q_0, aabab) = \delta(q_0, b) \cup \delta(q_1, b) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$$

4.2 Розширена функція переходів для ДСА

Розглянемо детермінований скінченний автомат (ДСА) з трьома станами: A, B та заключним станом C.



State	0	1	*Final*
A	A	B	$false$
B	A	C	$false$
C	C	C	$true$

Розширена функція переходів для ДСА, на відміну від НСА, повертає один стан або порожню множину.

Розширена функція переходів $\hat{\delta}$ визначається рекурсивно:

Приклад обчислення

Розглянемо обчислення $\hat{\delta}(B, 011)$:

$$\begin{aligned}\hat{\delta}(B, 011) &= \delta(\hat{\delta}(B, 01), 1) \\ &= \delta(\delta(\hat{\delta}(B, 0), 1), 1) \\ &= \delta(\delta(A, 1), 1) \\ &= \delta(B, 1) \\ &= C\end{aligned}$$

5 Мінімізація за допомогою таблиці нееквівалентних станів

Цей метод використовує таблицю, де кожна комірка відповідає парі станів [2, 4]:

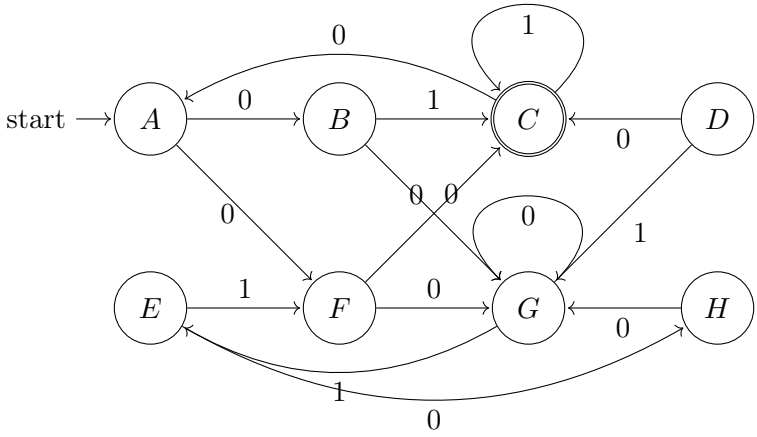
1. Всі пари станів, де один стан є заключним, а інший — ні, позначаються як нееквівалентні.
2. Для кожної пари станів p, q, перевіряється, чи існує символ $a \in \Sigma$, який переводить ці стани в пари, вже позначені як нееквівалентні.
3. Якщо така пара існує, p і q також позначаються як нееквівалентні.
4. Повторюємо кроки, доки не буде знайдено всі нееквівалентні стани.

5.1 Приклад мінімізації скінченного автомата за допомогою таблиці нееквівалентних станів

Мінімізація скінченного автомата за допомогою таблиці нееквівалентних станів полягає у визначенні пар станів, які не є еквівалентними, та їх подальшому об'єднанні.

Діаграма автомата

Автомат має такі стани: A, B, C, D, E, F, G, H, де A є початковим станом, а C — заключним, D — недосяжний стан. Нижче наведено діаграму переходів автомата:



Аналіз станів

Для визначення еквівалентних станів використовується розширена функція переходів $\hat{\delta}$. Продемонструємо приклади обчислень:

$$\hat{\delta}(C, \varepsilon) \in \mathbf{F}$$
$$\hat{\delta}(G, \varepsilon) \notin \mathbf{F}$$
$$\hat{\delta}(A, 1) = F \notin \mathbf{F}$$
$$\hat{\delta}(H, 1) = C \in \mathbf{F}$$
$$\hat{\delta}(A, 01) = C \in \mathbf{F}$$
$$\hat{\delta}(G, 01) = E \notin \mathbf{F}$$

Нижче наведено таблицю, яка показує пари нееквівалентних станів:

	A	B	C	E	F	G
B	$\times$					
C	$\times_\varepsilon$	$\times_\varepsilon$				
E		$\times$	$\times_\varepsilon$			
F	$\times_{01}$	$\times$	$\times_\varepsilon$	$\times$		
G	$\times$	$\times$	$\times_\varepsilon$	$\times$	$\times$	
H	$\times_1$		$\times_\varepsilon$	$\times$	$\times$	$\times$

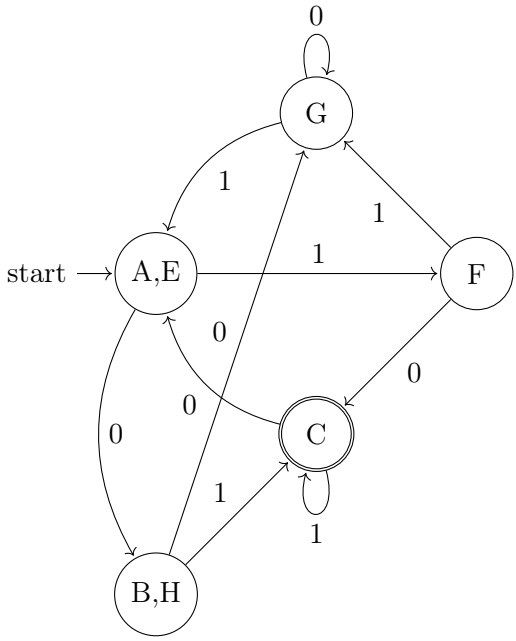
У таблиці:

- Символ $\times$ позначає, що стани нееквівалентні.
- Символ $\times_\varepsilon$ вказує на нееквівалентність станів за певною умовою (стани розрізнив порожній ланцюжок).
- Символи $\times_{01}$ та $\times_1$ позначають додаткові умови нееквівалентності (стани розрізнились ланцюжками 01 і 1 відповідно).

5.2 Результати мінімізації

- **Недосяжний стан**: D — недосяжний стан, тому його вилучено.
- **Еквівалентні стани**: Стани $\{A, E\}$ та $\{B, H\}$ еквівалентні.

  Отже, отримуємо мінімізований автомат з меншою кількістю станів, який зберігає свою функціональність.



Контрольні запитання та завдання

1. Основні означення

1. Що таке мінімізація скінченного автомата?
2. Що означає, що стани q_1 та q_2 є розрізнюваними?
3. Як визначити, чи є стан недосяжним?
4. Які умови необхідно виконати, щоб автомат був мінімальним?

2. Алгоритм видалення недосяжних станів

1. Як визначити початковий список досяжних станів?
2. Що відбувається з недосяжними станами після виконання алгоритму?

3. Чому недосяжні стани не впливають на функціональність автомата?

3. Мінімізація за допомогою класів еквівалентності

1. Що таке класи еквівалентності в контексті мінімізації автомата?

2. Яка початкова розбивка станів під час мінімізації?

3. Як уточнюються класи еквівалентності на кожному кроці?

4. Коли процес мінімізації можна вважати завершеним?

5. Яка теорема гарантує мінімальну кількість станів у мінімізованому автоматі?

4. Приклад мінімізації

1. Як аналіз класів еквівалентності допомагає зменшити кількість станів?

2. Як визначити, що два стани належать одному класу еквівалентності?

3. Які стани залишаються після мінімізації в прикладі з автоматом, що розпізнає ланцюжки з кількістю символів b, кратних 3?

Загальні питання

1. Чим мінімізований автомат відрізняється від початкового?

2. Чи впливає порядок обробки станів на результат мінімізації?

3. Які практичні переваги мінімізації скінченних автоматів?

Глосарій

Мінімізація скінченного автомата — процес побудови найменшого еквівалентного автомата шляхом видалення недосяжних станів та об'єднання еквівалентних станів.

Еквівалентні стани — два стани q_1 та q_2 вважаються еквівалентними, якщо для будь-якого вхідного ланцюжка $x \in \Sigma^*$ результати переходів з цих станів призводять до однакових заключних станів.

Недосяжний стан — стан q вважається недосяжним, якщо не існує жодного вхідного ланцюжка $x \in \Sigma^*$, який переводить початковий стан q_0 в q.

Розрізнювані стани — два стани q_1 та q_2 вважаються розрізнюваними, якщо існує вхідний ланцюжок $x \in \Sigma^*$, який переводить q_1 і q_2 в стани, що належать до різних множин F та $Q \setminus F$.

Класи еквівалентності — множини станів, які є еквівалентними за певним відношенням еквівалентності. Використовуються для мінімізації автомата.

Відношення еквівалентності $\equiv^k$ — відношення, яке визначає, чи є два слова u та v еквівалентними за умови, що для будь-якого ланцюжка w довжиною не більше k виконується $uw \in L \iff vw \in L$.

Розширена функція переходів $\hat{\delta}$ — функція, яка визначає множину станів, до яких автомат може перейти після обробки вхідного ланцюжка ω, починаючи зі стану q_0.

Мінімальний автомат — автомат, який має найменшу кількість станів серед усіх еквівалентних автоматів. Він не містить недосяжних станів, і всі його стани є розрізнюваними.

Алгоритм видалення недосяжних станів — алгоритм, який визначає всі досяжні стани автомата шляхом обходу графу станів і видаляє з автомата стани, які не можуть бути досягнуті з початкового стану.

Алгоритм знаходження класів еквівалентності — алгоритм, який використовує покрокове уточнення класів еквівалентності станів для мінімізації автомата.

Таблиця нееквівалентних станів — таблиця, яка використовується для визначення пар станів, які не є еквівалентними. Використовується в алгоритмі мінімізації автомата.

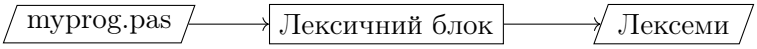
Тема VI. Лексичні аналізатори

1 Поняття лексичного аналізатора

Розробка лексичного аналізатора передбачає створення автоматів, які ідентифікують різні види лексем. Такі автомати можуть бути організовані у послідовній або паралельній структурі.

Лексичний аналізатор – це спеціалізована програма, яка здійснює лексичний аналіз і включає два основні етапи:

- перетворення вхідних символів у набір лексем, виділення лексем певного типу з потоку символів або ідентифікація окремого типу лексем;
- визначення значень для певних категорій лексем.



Деякі типи лексем деякої мови програмування високого рівня:

- ключові слова (while, for, return, downto, ...);
- ідентифікатори (sa, i2, x1, abc, ...);
- знаки операцій ($!=$, $\leq$, $<$, $==$, ...);
- числові константи (4251, -12.03, ...).

Припустимо, що нам потрібно розпізнавати лише один тип лексем. У такому випадку процес побудови спрощеного лексичного аналізатора включає такі етапи:

1. Побудова регулярної граматики або регулярного виразу.
2. Побудова недетермінованого скінченного автомата.
3. Побудова детермінованого скінченного автомата.

4. Мінімізація скінченного автомата.

5. Програмна реалізація скінченного автомата.

Цей процес є коректним, оскільки існує таке твердження.

Твердження

Одну й ту саму регулярну мову L можна представити у вигляді:

1. Регулярної множини.

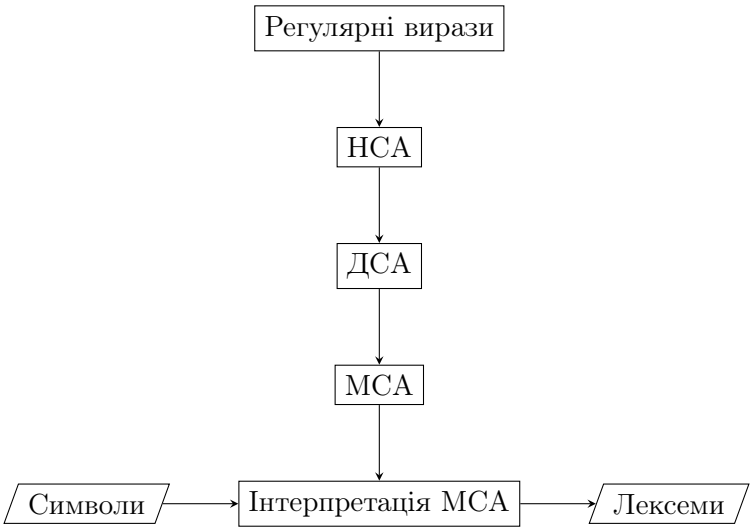
2. Регулярного виразу.

3. Праволінійної граматики.

4. Недетермінованого скінченного автомата.

5. Детермінованого скінченного автомата.

Можливий варіант реалізації лексичного аналізатора:

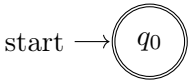


2 Алгоритм побудови НСА за регулярним виразом

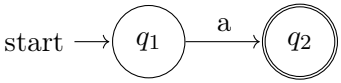
Нехай $L = L(R_0)$ — мова, задана регулярним виразом R_0. Побудуємо недетермінований скінченний автомат M_0, такий, що $L = L(M_0)$ [1,4].

2.1 Базові випадки

1. $R_0 = \varepsilon$.
Автомат $M_0 = (\{q_0\}, \Sigma, \varnothing, q_0, \{q_0\})$.



2. $R_0 = a$, де $a \in \Sigma$.
Автомат $M_0 = (\{q_1, q_2\}, \Sigma, \delta_0, q_1, \{q_2\})$, де $\delta_0(q_1, a) = \{q_2\}$.



Нехай для регулярних виразів R_1 та R_2 вже побудовані автомати M_1 та M_2.

$$M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1),$$
$$M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2),$$

де

$$Q_1 \cap Q_2 = \varnothing.$$

Розглянемо, як побудувати автомати для різних операцій над цими виразами.

2.2 Об'єднання регулярних виразів

Для побудови автомата M_0, який розпізнає об'єднання $R_1 + R_2$, виконаємо такі кроки:

- Додамо новий початковий стан q_0.
- Визначимо переходи з q_0 так, щоб вони відповідали переходам з початкових станів автоматів M_1 та M_2.
- Старі початкові стани q_1 та q_2 стають недосяжними.
- Новий початковий стан q_0 стає заключним, якщо хоча б один зі старих початкових станів був заключним.

Автомат $M_0 = (Q_1 \cup Q_2 \cup \{q_0\}, \Sigma, \delta_0, q_0, F_0)$, де:

- q_0 — новий стан.
- $\delta_0(q, a) = \delta_1(q, a)$ для $q \in Q_1$.
- $\delta_0(q, a) = \delta_2(q, a)$ для $q \in Q_2$.
- $\delta_0(q_0, a) = \delta_1(q_1, a) \cup \delta_2(q_2, a)$.
- $F_0 = F_1 \cup F_2$, якщо $q_1 \notin F_1$ та $q_2 \notin F_2$.
- $F_0 = F_1 \cup F_2 \cup q_0$, якщо $q_1 \in F_1$ або $q_2 \in F_2$.

2.3 Конкатенація регулярних виразів

Для побудови автомата M_0, який розпізнає конкатенацію $R_1 R_2$, виконаємо такі кроки:

- Автомат M_0 починає роботу, моделюючи автомат M_1.
- Коли M_0 досягає заключного стану автомата M_1, він переходить у стани автомата M_2, які виходять зі стану q_2, і продовжує роботу, моделюючи M_2.
- Стан q_2 стає недосяжним.

Автомат $M_0 = (Q_1 \cup Q_2, \Sigma, \delta_0, q_1, F_2)$, де:

- $\delta_0(q, a) = \delta_1(q, a)$ для $q \in Q_1 \setminus F_1$.
- $\delta_0(q, a) = \delta_2(q, a)$ для $q \in Q_2$.
- $\delta_0(q, a) = \delta_1(q, a) \cup \delta_2(q_2, a)$ для $q \in F_1$.
- $F_0 = F_2$, якщо $q_2 \notin F_2$.
- $F_0 = F_1 \cup F_2$, якщо $q_2 \in F_2$.

2.4 Ітерація "*"регулярного виразу

Для побудови автомата M_0, який розпізнає ітерацію R_1^*, виконаємо наступні кроки:

- Додамо новий початковий стан q_0.
- Визначимо переходи з q_0 так, щоб вони відповідали переходам з початкового стану автомата M_1.
- Стан q_0 стає одночасно початковим і заключним.
- Коли M_0 досягає заключного стану автомата M_1, він переходить у стани автомата M_1, які виходять зі стану q_1, для повторення.
- Стан q_1 стає недосяжним.

Автомат $M_0 = (Q_1 \cup \{q_0\}, \Sigma, \delta_0, q_0, F_1 \cup \{q_0\})$, де:

- q_0 — новий стан.
- $\delta_0(q_0, a) = \delta_1(q_1, a)$.
- $\delta_0(q, a) = \delta_1(q, a)$ для $q \in Q_1 \setminus F_1$.
- $\delta_0(q, a) = \delta_1(q, a) \cup \delta_1(q_1, a)$ для $q \in F_1$.

2.5 Ітерація "+"регулярного виразу

Для побудови автомата M_0, який розпізнає ітерацію R_1^+, виконаємо такі кроки:

- Автомат M_0 починає роботу, моделюючи автомат M_1.
- Коли M_0 досягає заключного стану автомата M_1, він переходить у стани автомата M_1, які виходять зі стану q_1, для повторення.

Автомат $M_0 = (Q_1, \Sigma, \delta_0, q_1, F_1)$, де:

- $\delta_0(q, a) = \delta_1(q, a)$ для $q \notin F_1$.
- $\delta_0(q, a) = \delta_1(q, a) \cup \delta_1(q_1, a)$ для $q \in F_1$.

3 Приклади побудови НСА за регулярним виразом

3.1 Приклад покрокової побудови НСА

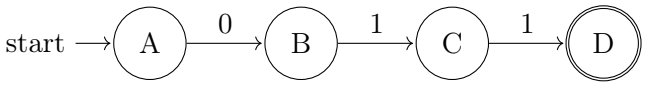
Розглянемо регулярний вираз:

$$R = ((011)^* \mid 110)(01)^*$$

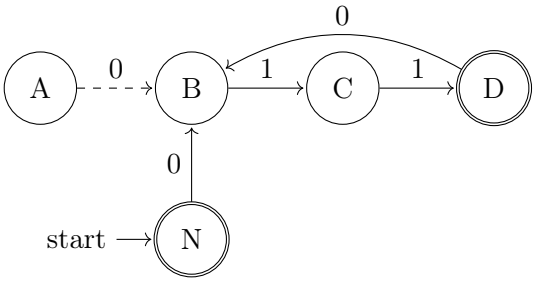
Побудуємо спочатку прості автомати для елементарних регулярних виразів, а потім будемо з'єднувати їх для отримання результату, щоб виконувалась рівність:

$$L(R) = L(M)$$

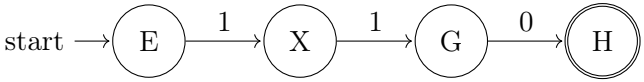
1. **011:**



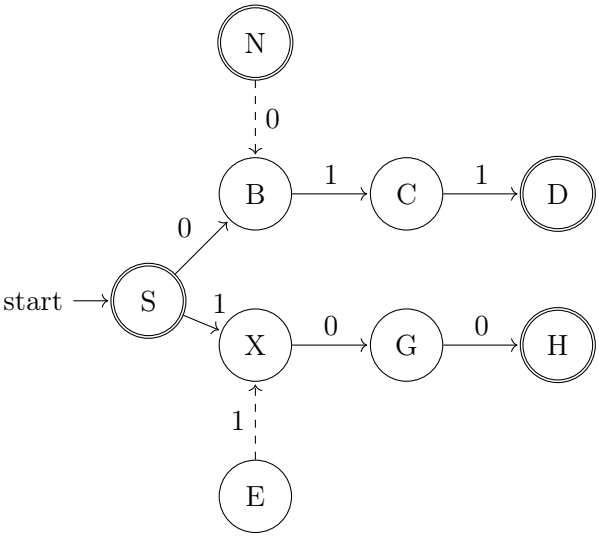
2. **(011)*:**



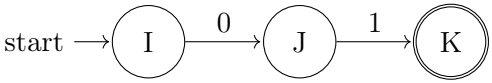
3. **110:**



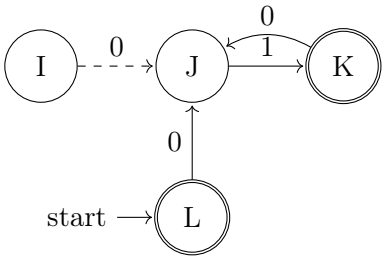
4. **(011)* | 110:**



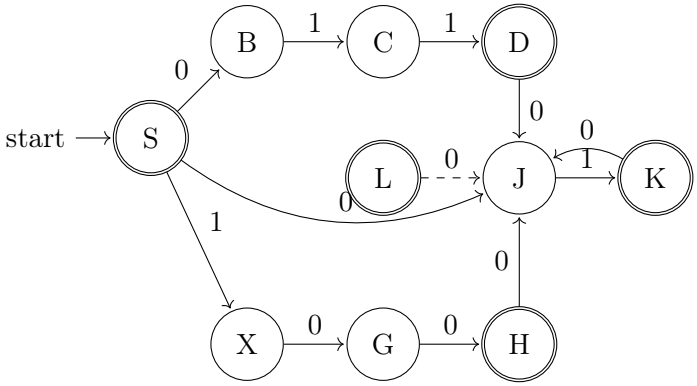
5. **01:**



6. **(01)*:**



7. **R = ((011)* | 110)(01)*:**



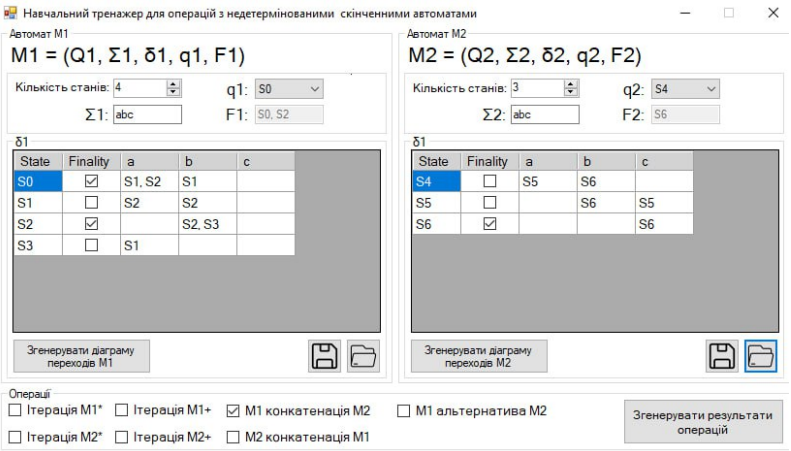
HCA:

$$M = (\{S, B, C, D, X, G, H, J, K\}, \{0, 1\}, \delta, S, F)$$

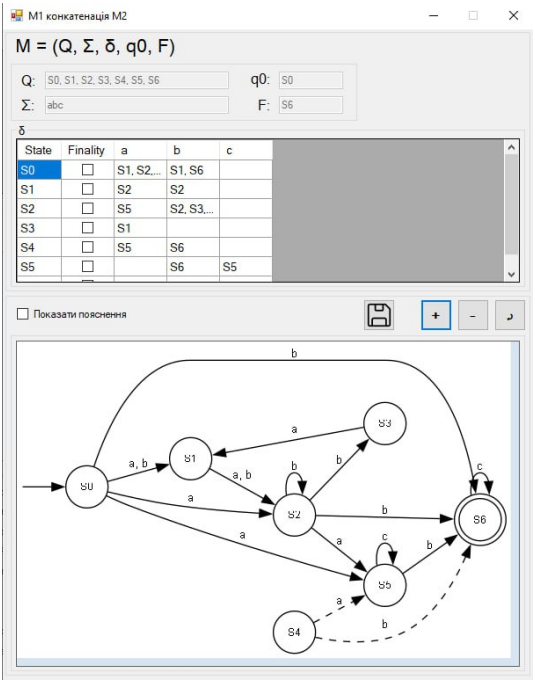
$$F = \{S, D, H, K\}, \quad L(R) = L(M).$$

3.2 Навчальний тренажер для операцій з недетермінованими скінченними автоматами

Для засвоєння етапів перетворення регулярного виразу в НСА розроблено програму, яка дозволяє вводити 2 автомати [21]:



і виконувати операції конкатенації, об'єднання та ітерації з цими автоматами. Нижче наведено результат виконання конкатенації над введеними операндами [21].



4 Побудова лексичного аналізатора, що розпізнає одну лексему

Побудуємо спрощений лексичний аналізатор, який розпізнає дійсні числа. Спочатку визначимо структуру дійсного числа:

1. Можливий знак (+ або −).
2. Послідовність цифр (можливо, порожня).
3. Десяткова крапка.
4. Послідовність цифр (хоча б одна).
5. Можлива експоненціальна частина, яка складається з:
   - Букви E.
   - Можливого знака (+ або −).
   - Послідовності цифр (хоча б одна).

4.1 Регулярний вираз для дійсного числа

$$(+|-|\varepsilon)\, d^*\, .\, d^+\, (E\,(+|-|\varepsilon)\, d^+|\varepsilon)$$

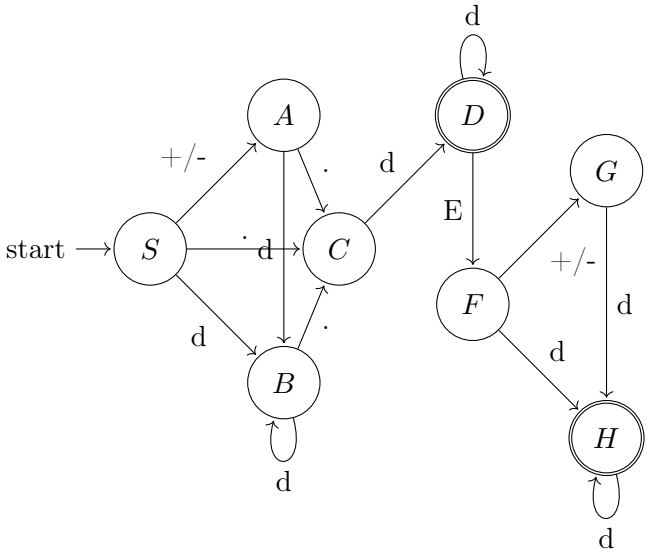
Тут d позначає будь-яку цифру в діапазоні від 0 до 9.

4.2 Детермінований скінченний автомат (ДСА)

За регулярними разом побудуємо скінченний автомат.
Таблиця переходів ДСА

State	+	−	d	.	E	$Final$
S	A	A	B	C	∅	false
A	∅	∅	B	C	∅	false
B	∅	∅	B	C	∅	false
C	∅	∅	D	∅	∅	false
D	∅	∅	D	∅	F	true
F	G	G	H	∅	∅	false
G	∅	∅	H	∅	∅	false
H	∅	∅	H	∅	∅	true

Діаграма переходів ДСА



4.3 Мінімізований скінченний автомат (МСА)

Проведемо неформальну мінімізацію автомата.

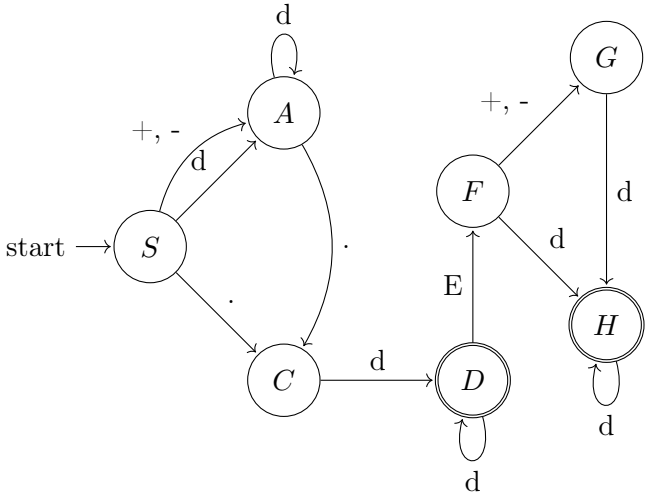
A, B - еквівалентні (однакові рядки таблиці);

G, H - нееквівалентні, бо один заключний, а другий – не заключний, хоча рядки однакові.

Після мінімізації отримуємо таку таблицю переходів:

State	+	−	*d*	.	*E*	*Final*
S	A	A	A	C	∅	false
A	∅	∅	A	C	∅	false
C	∅	∅	D	∅	∅	false
D	∅	∅	D	∅	F	true
F	G	G	H	∅	∅	false
G	∅	∅	H	∅	∅	false
H	∅	∅	H	∅	∅	true

Діаграма переходів МСА



4.4 Праволінійна граматика

Для мінімізованого автомата випишемо праволінійну граматику без порожніх продукцій:

$$S \to +A| - A|dA|.C$$
$$A \to dA|.C$$
$$C \to dD|d$$
$$D \to dD|EF|d$$
$$F \to +G| - G|dH|d$$
$$G \to dH|d$$
$$H \to dH|d$$

Як бачимо, при перетворенні СА у праволінійну граматику, крім очевидних продукцій, праві частини яких закінчуються нетерміналом, з'являються продукції з термінальною правою частиною.

5 Програмна реалізація детермінованого скінченного автомата

5.1 Перший спосіб: автоматне програмування

```
#include <iostream>
#include <cctype> // для isdigit

// Функція для перевірки, чи є рядок дійсним чис
    лом
bool isRealNumber(const std::string& s) {
    char state = 'S'; // Початковий стан
    for (char ch : s) {
        switch (state) {
            case 'S': // Початковий стан
                if (ch == '+' || ch == '-' ||
                    isdigit(ch)) {
                    state = 'A';
                } else if (ch == '.') {
                    state = 'C';
                } else {
                    return false;
```

```
16                  }
17                  break;
18              case 'A': // Ціла частина
19                  if (isdigit(ch)) {
20                      state = 'A';
21                  } else if (ch == '.') {
22                      state = 'C';
23                  } else {
24                      return false;
25                  }
26                  break;
27              case 'C': // Крапка
28                  if (isdigit(ch)) {
29                      state = 'D';
30                  } else {
31                      return false;
32                  }
33                  break;
34              case 'D': // Дробова частина
35                  if (isdigit(ch)) {
36                      state = 'D';
37                  } else if (ch == 'E') {
38                      state = 'F';
39                  } else {
40                      return false;
41                  }
42                  break;
43              case 'F': // Експонента
44                  if (ch == '+' || ch == '-') {
45                      state = 'G';
46                  } else if (isdigit(ch)) {
47                      state = 'H';
48                  } else {
49                      return false;
50                  }
51                  break;
52              case 'G': // Знак експоненти
53                  if (isdigit(ch)) {
54                      state = 'H';
55                  } else {
```

```
                    return false;
                }
                break;
            case 'H': // Число в експоненті
                if (!isdigit(ch)) {
                    return false;
                }
                break;
            default:
                return false;
        }
    }
    // Перевірка кінцевого стану
    return (state == 'D' || state == 'H');
}

int main() {
    std::string input;
    std::cout << "Введіть дійсне число: ";
    std::cin >> input;

    if (isRealNumber(input)) {
        std::cout << "Так, це дійсне число.\n";
    } else {
        std::cout << "Ні, це не дійсне число.\n"
            ;
    }

    return 0;
}
```

5.2 Другий спосіб: універсальний підхід

```
#include <iostream>
#include <cctype> // для isdigit
#include <unordered_map> // для зручної роботи з
    таблицею переходів

```

```
// Функція для перевірки, чи є рядок дійсним чис
    лом
bool isRealNumber(const std::string& s) {
    // Таблиця переходів
    const int table[8][6] = {
        {0, 0, 0, 0, 0, 0}, // стан 0 (невірний)
        {0, 2, 2, 2, 3, 0}, // стан 1
        {0, 0, 0, 2, 3, 0}, // стан 2
        {0, 0, 0, 4, 0, 0}, // стан 3
        {0, 0, 0, 4, 0, 5}, // стан 4
        {0, 6, 6, 7, 0, 0}, // стан 5
        {0, 0, 0, 7, 0, 0}, // стан 6
        {0, 0, 0, 7, 0, 0}  // стан 7
    };

    // Визначення символів
    std::unordered_map<char, int> symbol = {
        {'+', 1}, {'-', 2}, {'.', 4}, {'E', 5}
    };
    for (char c = '0'; c <= '9'; ++c) {
        symbol[c] = 3;
    }

    int state = 1; // Початковий стан
    for (char ch : s) {
        int col = symbol.count(ch) ? symbol[ch]
            : 0; // Визначення стовпця
        state = table[state][col]; // Перехід до
            нового стану
        if (state == 0) break; // Невірний стан
    }

    // Перевірка кінцевого стану
    return (state == 4 || state == 7);
}

int main() {
    std::string input;
    std::cout << "Введіть дійсне число: ";
    std::cin >> input;
```

```
42
43      if (isRealNumber(input)) {
44          std::cout << "Так, це дійсне число.\n";
45      } else {
46          std::cout << "Ні, це не дійсне число.\n"
                ;
47      }
48
49      return 0;
50  }
```

Контрольні запитання та завдання

1. Поняття лексичного аналізатора

1. Що таке лексичний аналізатор і яку функцію він виконує?
2. Які основні етапи роботи лексичного аналізатора?
3. Перерахуйте типи лексем, які розпізнаються мовами програмування.
4. У чому полягає процес побудови спрощеного лексичного аналізатора?
5. Які способи представлення регулярної мови можна використати для розробки лексичного аналізатора?

2. Алгоритм побудови НСА за регулярним виразом

1. Як побудувати недетермінований скінченний автомат (НСА) для регулярного виразу $R_0 = \varepsilon$?
2. Як виглядає НСА для регулярного виразу $R_0 = a$, де $a \in \Sigma$?

3. Які етапи необхідні для побудови НСА для об'єднання регулярних виразів $(R_1 + R_2)$?

4. Як будується НСА для конкатенації регулярних виразів $(R_1 R_2)$?

5. Які особливості побудови НСА для ітерації регулярного виразу (R_1^* та R_1^+)?

3. Приклад побудови НСА за регулярним виразом

1. Які етапи побудови НСА для складного регулярного виразу, такого як $R = ((011)^* \mid 110)(01)^*$?

2. Як об'єднуються прості автомати для створення НСА складного регулярного виразу?

Глосарій

Лексичний аналізатор – спеціалізована програма, яка здійснює лексичний аналіз, перетворюючи вхідні символи у набір лексем та визначаючи значення для певних категорій лексем.

Лексема – елементарна одиниця тексту, яка має певне значення в мові програмування (наприклад, ключове слово, ідентифікатор, знак операції, числова константа).

Регулярний вираз – формальний опис множини рядків, який використовується для задання шаблонів пошуку в тексті або для опису лексем.

Скінченний автомат (автомат) – математична модель, яка складається з набору станів, переходів між ними та правил, що визначають поведінку автомата на основі вхідних символів.

Недетермінований скінченний автомат (НСА) – скінченний автомат, у якому для одного стану та вхідного символу може існувати кілька можливих переходів.

Детермінований скінченний автомат (ДСА) – скінченний автомат, у якому для кожного стану та вхідного символу існує лише один можливий перехід.

Мінімізований скінченний автомат (МСА) – детермінований скінченний автомат з мінімальною кількістю станів, який розпізнає ту саму мову, що й початковий автомат.

Праволінійна граматика – формальна граматика, у якій всі правила мають вигляд $A \to aB$ або $A \to a$, де A та B – нетермінали, а a – термінал.

Лексичний аналіз – процес розбиття вхідного тексту на лексеми, які потім використовуються для подальшого аналізу (наприклад, синтаксичного аналізу).

Регулярна мова – мова, яка може бути описана регулярним виразом, регулярною граматикою або скінченним автоматом.

Автоматне програмування – підхід до програмування, який використовує скінченні автомати для реалізації логіки програми.

Універсальний підхід – метод реалізації лексичного аналізатора, який використовує таблицю переходів для опису поведінки автомата.

Тема VII. Синтаксичний аналіз

1 Синтаксичний аналіз та контекстно-вільні граматики. Дерева розбору

1.1 Контекстно-вільні граматики та їх особливості

Контекстно-вільна граматика — це граматика, у якій усі ліві частини правил складаються лише з одного нетермінального символу. Завдяки цьому правила не залежать від контексту, в якому вони застосовуються.

Формально, граматика $G = (N, T, P, S)$ контекстно-вільна, якщо всі правила P мають вигляд $A \to w$, де $A \in N$, $w \in (N \cup T)^*$. Наприклад, якщо є правило $A \to w$, то його можна застосувати до будь-якого ланцюжка uAv, незалежно від символів u та v.

1.2 Синтаксичний аналіз ланцюжків

Розглянемо граматику:

$$G = (\{X, Y\}, \{x, +, [,]\}, P, X),$$

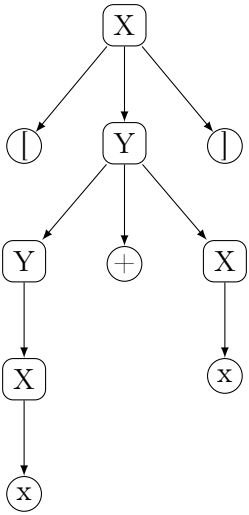
де множина правил P визначена так:

$$\begin{aligned} &X \to x, \\ &X \to [Y], \\ &Y \to X, \\ &Y \to Y + X. \end{aligned}$$

Ланцюжок $[x + x]$ належить мові $L(G)$, тому що його можна побудувати так:

$$\underline{X} \Rightarrow \underline{Y} \Rightarrow [\underline{X} + X] \Rightarrow [\underline{X} + X] \Rightarrow [x + \underline{X}] \Rightarrow [x + x].$$

Дерево розбору для ланцюжка $[x + x]$



Властивості дерев розбору

Дерево розбору для граматики $G = (N, T, P, S)$ має такі властивості:

1. Коренем дерева є початковий нетермінал S.
2. Листові вузли позначають термінальні символи або порожній ланцюжок.
3. Якщо вузол позначено нетерміналом A, а його дочірні вузли — це $b_1 b_2 \dots b_n$, то існує правило $A \to b_1 b_2 \dots b_n$.

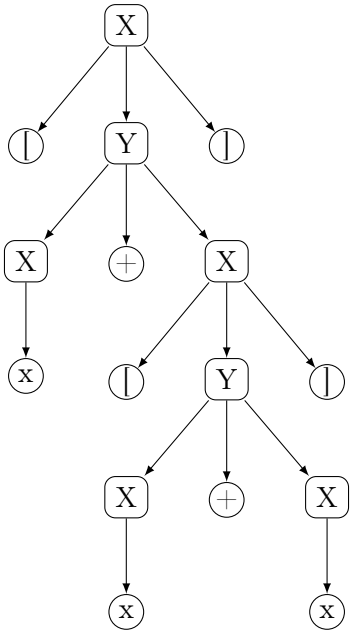
Процес заміни нетермінала на праву частину правила називається розгортанням. Зворотний процес — згортанням.

Ланцюжок $[x + [x + x]]$ належить мові $L(G)$, оскільки його можна побудувати так:

$$\underline{X} \Rightarrow [\underline{Y}] \Rightarrow [\underline{Y} + X] \Rightarrow [\underline{X} + X] \Rightarrow [x + \underline{X}] \Rightarrow [x + [\underline{Y}]]$$

$$\Rightarrow [x + [\underline{X} + X]] \Rightarrow [x + [x + \underline{X}]] \Rightarrow [x + [x + x]].$$

Дерево розбору для ланцюжка $[x + [x + x]]$



1.3 Методи синтаксичного аналізу

Синтаксичний аналіз здійснюється двома основними способами:

- **Низхідний аналіз** — починається з початкового символу S і поступово розгортає правила до термінального ланцюжка.
- **Висхідний аналіз** — починається з термінального ланцюжка і поступово згортається до початкового символу.

Розглянемо дві стратегії синтаксичного аналізу, які використовують ці підходи, на прикладі граматики.

Нехай задано граматику G з правилами [2, 4]:

$$E \to E + T \mid T,$$
$$T \to T * F \mid F,$$
$$F \to (E) \mid a,$$

Аналіз ланцюжка $a + a * a$ для граматики G

Нетермінали E, T, F позначають "вираз"(*Expression*), "доданок"(*Term*) і "множник"(*Factor*) відповідно. Вони використовуються для опису математичних виразів із операціями $+$ і $*$, а також їх компонентів.

Розглянемо процес побудови слова $a + a * a$ за стратегією, де кожного разу розгортається перший нетермінал зліва:

$$\underline{E} \Rightarrow \underline{E} + T \Rightarrow \underline{T} + T \Rightarrow \underline{F} + T \Rightarrow a + \underline{F}$$

$$\Rightarrow a + \underline{T} * F \Rightarrow a + a * \underline{F} \Rightarrow a + a * a.$$

На кожному кроці підкреслено нетермінал, який піддається розгортанню. Цей підхід називається *низхідним синтаксичним аналізом*, оскільки побудова починається з початкового символу граматики і прямує до термінального рядка.

Тепер розглянемо інший підхід: розгортання здійснюється для останнього праворуч нетермінала. Послідовність виглядає так:

$$\underline{E} \Rightarrow \underline{E} + T \Rightarrow \underline{E} + T * F \Rightarrow \underline{E} + T * a \Rightarrow \underline{E} + F * a$$

$$\Rightarrow \underline{E} + a * a \Rightarrow \underline{T} + a * a \Rightarrow \underline{F} + a * a \Rightarrow a + a * a.$$

У цій стратегії, якщо рухатися в зворотному порядку, отримуємо процес згортання, що починається з термінального рядка. Цей підхід називається *висхідним аналізом*, адже він працює від терміналів до початкового символу граматики.

Оскільки компілятори обробляють текст програм зліва направо і зверху вниз, для створення дерева розбору методом «від верху до низу» зазвичай використовують лівостороннє

виведення. У свою чергу, для методу «від низу до верху» переважно застосовується правостороннє виведення. У результаті побудовані дерева розбору для обох підходів будуть однаковими для одного й того самого слова $a + a * a$.

Граматика вважається *однозначною*, якщо для будь-якого рядка, який вона породжує, існує єдине дерево розбору. У протилежному випадку вона є *неоднозначною*.

Основна складність при побудові алгоритмів синтаксичного аналізу полягає у виборі правила продукції для розгортання або згортання. У прикладах цей вибір був очевидним, оскільки структура термінального рядка була відома. Але в загальному випадку структура рядка заздалегідь невідома, тому необхідно перебирати правила для знаходження потрібного.

Основною проблемою при побудові алгоритмів синтаксичного аналізу є необхідність вибору правил, які потрібно застосувати для розгортання чи згортання. У наведених прикладах вибір правил був можливим завдяки тому, що структура термінального слова була відома заздалегідь. Однак у загальному випадку структура слова до початку його аналізу невідома, що призводить до необхідності перебирати всі можливі правила для знаходження потрібного.

Хоча теоретично можна розробити алгоритм аналізу, який базується на перебиранні всіх можливих правил, такий підхід практично неприйнятний через його високу обчислювальну складність. Одним із способів досягнення ефективних алгоритмів аналізу є обмеження структури правил граматики та звуження множини контекстно-вільних граматик, що дозволяє уникнути необхідності перебирання.

2 LA(1)-граматики

LA(1)-граматики дозволяють вибирати потрібне правило для розгортання під час низхідного аналізу, орієнтуючись на перший символ ще не обробленої частини вхідного рядка. На-

зва "$LA(1)$" походить від англійського *Look Ahead 1 symbol*, що означає "подивись на один символ вперед"[2].

Розглянемо граматику $G = (N, T, P, S)$, де N — нетермінали, T — термінали, P — правила, а S — початковий символ.

Нехай потрібно визначити, чи належить слово w мові $L(G)$.

Якщо $S \to w_1 \mid \cdots \mid w_p$ — це всі правила, в яких нетермінал S стоїть зліва, то відповідне правило $S \to w_i$ можна обрати на основі першого символу слова v, якщо множини перших символів, які можуть бути виведені з $w_1, w_2, \ldots, w_p$, не перетинаються.

У загальному випадку, якщо нерозпізнана частина слова $a_i \ldots a_j$ повинна бути виведена з нетермінала A, а $A \to v_1 \mid \cdots \mid v_k$ — це всі правила з A у лівій частині, то правило $A \to v_i$ обирається відповідно до першого символу a_i, якщо множини перших символів, які можуть бути виведені з $v_1, v_2, \ldots, v_k$, не перетинаються.

Для кожного нетермінала A та кожної правої частини v_i правил $A \to v_1 \mid v_2 \mid \cdots \mid v_k$ визначимо множини

$$\text{first}(v_i) = \{a \mid a \in T \text{ і } v_i \Rightarrow^* az \text{ для деякого слова } z\},$$

$$\text{first}(A) = \bigcup \text{first}(v_i).$$

Якщо граматика містить ε-правила вигляду $A \to \varepsilon$, то, якщо $Av \Rightarrow^* b_1 \ldots b_n$, символ b_1 може починати слово, виведене як з A, так і з v. Визначити, з чого саме виводиться слово, можна за умови, що $\text{first}(A)$ не містить перших символів слів, виведених з v.

Множину цих перших символів позначимо $\text{follow}(A)$:

Граматика називається LA(1)-граматикою, якщо:

1. для кожного нетермінала A з правилами $A \to w_1 | \ldots | w_k$, де $w_i \neq \varepsilon$ для всіх $i = 1, \ldots, k$, виконується умова

$$\text{first}(w_i) \cap \text{first}(w_j) = \varnothing \quad \text{при } i, j = 1, \ldots, k, i \neq j;$$

2. для кожного нетермінала A, який має правило $A \to \varepsilon$, виконується

$$\text{first}(A) \cap \text{follow}(A) = \varnothing.$$

Ці умови називаються LA(1)-умовами.

Не всі контекстно-вільні мови можуть бути описані LA(1)-граматиками, але синтаксис більшості сучасних мов програмування може бути заданий такими граматиками. Часто мова природно задається не LA(1)-граматикою, але її можна перетворити на еквівалентну LA(1)-граматику.

Приклад 1: Множини first та перевірка LA(1)-умов

Розглянемо граматику $G_1 = (\{A, B, C\}, \{x, y, z\}, P, A)$:

$$P: \begin{array}{l} A \to B \\ A \to yC \\ B \to z \\ B \to xA \\ C \to x \\ C \to zB \end{array}$$

Множини first

Для кожного нетермінала та правила обчислимо множини first:

$$\begin{aligned} \text{first}(B) &= \{z, x\}, \\ \text{first}(yC) &= \{y\}, \\ \text{first}(z) &= \{z\}, \\ \text{first}(xA) &= \{x\}, \\ \text{first}(x) &= \{x\}, \\ \text{first}(zB) &= \{z\}. \end{aligned}$$

Перевірка LA(1)-умов

1. Для нетермінала A:

$$\text{first}(B) = \{z, x\}, \quad \text{first}(yC) = \{y\}.$$

Оскільки $\{z, x\} \cap \{y\} = \varnothing$, умова (1) виконується.

2. Для нетермінала B:

$$\text{first}(z) = \{z\}, \quad \text{first}(xA) = \{x\}.$$

Оскільки $\{z\} \cap \{x\} = \varnothing$, умова (1) виконується.

3. Для нетермінала C:

$$\text{first}(x) = \{x\}, \quad \text{first}(zB) = \{z\}.$$

Оскільки $\{x\} \cap \{z\} = \varnothing$, умова (1) виконується.

Оскільки в граматиці G_1 немає ε-правил, умова (2) автоматично виконується. Отже, граматика G_1 є LA(1)-граматикою.

Приклад 2: Множини first та перевірка LA(1)-умов

Розглянемо граматику G_0:

$$\begin{aligned} & E \to E + T \mid T \\ P: \; & T \to T * F \mid F \\ & F \to (E) \mid a \end{aligned}$$

Множини first

Для кожного нетермінала та правила обчислимо множини first:

$$\begin{aligned}
\text{first}(E + T) = \text{first}(E) &= \{(, a\},\\
\text{first}(T) &= \{(, a\},\\
\text{first}(T * F) = \text{first}(T) &= \{(, a\},\\
\text{first}(F) &= \{(, a\},\\
\text{first}((E)) &= \{(\ \},\\
\text{first}(a) &= \{a\}.
\end{aligned}$$

Перевірка LA(1)-умов

1. Для нетермінала E:

$$\text{first}(E + T) = \{(, a\}, \quad \text{first}(T) = \{(, a\}.$$

Оскільки $\{(, a\} \cap \{(, a\} \neq \varnothing$, умова (1) не виконується.

2. Для нетермінала T:

$$\text{first}(T * F) = \{(, a\}, \quad \text{first}(F) = \{(, a\}.$$

Оскільки $\{(, a\} \cap \{(, a\} \neq \varnothing$, умова (1) не виконується.

3. Для нетермінала F:

$$\text{first}((E)) = \{(\ \}, \quad \text{first}(a) = \{a\}.$$

Оскільки $\{(\ \} \cap \{a\} = \varnothing$, умова (1) виконується.

Оскільки в граматиці G_0 є правила, які порушують LA(1)-умови, вона не є LA(1)-граматикою. Проте, як показано раніше, її можна перетворити на еквівалентну LA(1)-граматику. Для цього введемо новий нетермінал B та правила:

$$B \to \varepsilon \mid *FB, \quad T \to FB.$$

Тепер:

$$\begin{aligned}
\text{first}(T) = \text{first}(F) &= \{a, (\ \},\\
\text{first}(*FB) &= \{*\}, \quad \text{умова (1) виконується},\\
\text{first}(B) &= \{*\},\\
\text{follow}(B) = \text{follow}(T) = \text{follow}(E) &= \{+,)\},\\
\text{first}(B) \cap \text{follow}(B) &= \varnothing, \quad \text{умова (2) виконується}.
\end{aligned}$$

Аналогічно, додавши нетермінал A та правила:

$$E \to TA, \quad A \to \varepsilon \mid +EA,$$

ми отримаємо LA(1)-граматику.

Після перетворення граматика G_0 має наступні правила P_0, які задовольняють LA(1)-умовам:

$$P_0 : \begin{array}{l} E \to T\,A \\ A \to +\,T\,A \mid \varepsilon \\ T \to F\,B \\ B \to *\,F\,B \mid \varepsilon \\ F \to (\,E\,) \mid a \end{array}$$

3 Ліворекурсивні та розширені правила. Синтаксичні діаграми

Правило вигляду $A \to Av$ називається ліворекурсивним. Якщо граматика містить правила $A \to Av \mid u$, де $u \neq \varepsilon$, то $\text{first}(Av) = \text{first}(u) \neq \varnothing$, і граматика не є LA(1)-граматикою. Однак такі правила дозволяють виводити слова з множини $\{u, uv, uvv, \dots\}$, яка може бути описана регулярним виразом uv^* або $u\{v\}$.

Замість правил $A \to Av \mid u$ можна записати розширене правило $A \to u\{v\}$. Такі правила називаються розширеними, і вони не розширюють множину мов, які можуть бути описані контекстно-вільними граматиками.

Приклад 3

Розглянемо граматику $G = (\{A, B\}, \{x, +, [,]\}, P, A)$:

$$P : \begin{array}{l} A \to x \\ A \to [B] \\ B \to A \\ B \to B + A \end{array}$$

Цю граматику можна переписати за допомогою розширених правил:

$$P_1 : \begin{matrix} A \to x \mid [B] \\ B \to A \mid B + A \end{matrix}$$

$$P_2 : \begin{matrix} A \to x \mid [B] \\ B \to A(+A)^* \end{matrix}$$

$$P_3 : A \to x \mid [A(+A)^*]$$

Розширені правила дозволяють більш компактно описувати граматики, зберігаючи їхню еквівалентність.

Приклад 4

Розглянемо граматику G_0, яка описує арифметичні вирази:

$$P : \begin{matrix} E \to E + T \mid T \\ T \to T * F \mid F \\ F \to (E) \mid a \end{matrix}$$

Цю граматику можна переписати за допомогою розширених правил:

$$P_1 : \begin{matrix} E \to T\{+T\} \\ T \to F\{*F\} \\ F \to (E) \mid a \end{matrix}$$

Кожне розширене правило може бути представлене у вигляді синтаксичної діаграми, що полегшує візуалізацію граматики.

3.1 Синтаксичні діаграми

Означення та побудова

Синтаксична діаграма — це орієнтований граф, який має дві фіксовані вершини: початкову вершину, з якої виходить одна дуга, і кінцеву вершину, в яку входить одна дуга. Дуги графа можуть бути позначені термінальними або нетермінальними символами.

Цей тип графа широко використовується для візуального зображення граматичних правил формальних мов. Він дозволяє легко розуміти структуру правил та спосіб виведення слів згідно з цими правилами.

Властивості синтаксичної діаграми

Орієнтованість: Кожна дуга у графі має напрямок, що вказує на послідовність переходів від початкової до кінцевої вершини.

Фіксовані вершини: Існує початкова вершина, з якої починається будь-який шлях, і кінцева вершина, в яку завершується будь-який шлях.

Позначення дуг: Дуги можуть бути позначені термінальними символами (елементами алфавіту) або нетермінальними символами (перемінними граматики).

Алгоритм побудови синтаксичної діаграми

Базовий випадок: Якщо α є одиночним символом (термінальним або нетермінальним), то синтаксична діаграма складається з трьох вершин: початкової, середньої (позначеної символом α) та кінцевої. Єдина дуга виходить з початкової вершини до середньої, а друга дуга виходить з середньої вершини до кінцевої.

Рекурсивний випадок: Якщо α є складним регулярним виразом, то синтаксичну діаграму можна побудувати так:

- розкладемо α на простіші компоненти за допомогою операцій об'єднання, катенації та ітерації;

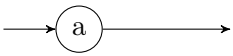
- для кожного компонента побудуємо окрему синтаксичну діаграму за допомогою базового випадку або рекурсивного випадку;

- об'єднаємо отримані діаграми в одну загальну діаграму, враховуючи порядок операцій у регулярному виразі α.

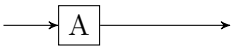
Синтаксична діаграма дозволяє візуально представити структуру граматичних правил та спосіб виведення слів з мови, заданої граматикою.

Отже, синтаксичну діаграму для правила $B \to \alpha$, де $B \in N$, а α — регулярний вираз у алфавіті $N \cup T$, можна побудувати рекурсивно:

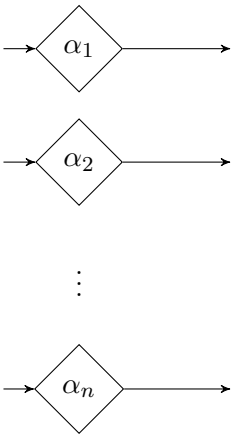
1. Якщо $\alpha = a$, де $a \in T$, то діаграма має вигляд:



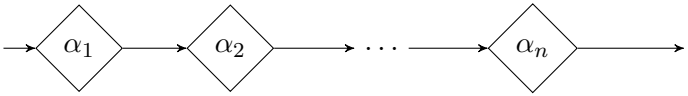
2. Якщо $\alpha = A$, де $A \in N$, то діаграма має вигляд:



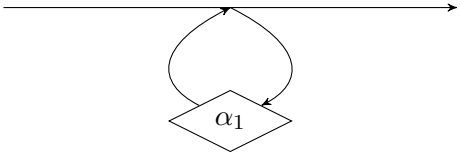
3. Якщо $\alpha = \alpha_1|\alpha_2|\ldots|\alpha_n$, де α_i — регулярні вирази, то діаграма має вигляд:



4. Якщо $\alpha = \alpha_1\alpha_2 \dots \alpha_n$, де α_i — регулярні вирази, то діаграма має вигляд:



5. Якщо $\alpha = \alpha_1^*$, де α_1 — регулярний вираз, то діаграма має вигляд:

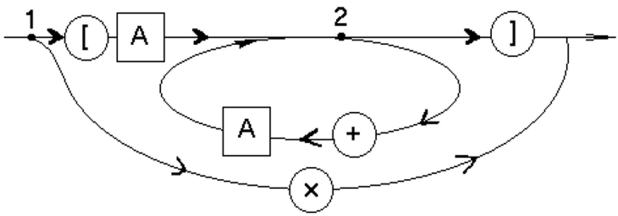


3.2 Синтаксичні аналізатори для розпізнавання виразів

Використовуючи метод рекурсивного спуску, побудуємо синтаксичні аналізатори для граматик з прикладів 3 і 4.

Розглянемо синтаксичну діаграму для граматики G з правилами P_3 з прикладу 3:

$G = \{(A, B); \{x, +, [1], P3, A\}, P3 : A \to x[[A(+A)]]\}$



Грунтуючись на цій діаграмі, напишемо код синтаксичного аналізатора мовою C:

```
#include <iostream>
#include <cstdio>  // для getchar()
#include <cstdlib> // для exit()

char c; // Поточний символ
```

```
6
7  // Функція для нетермінала A
8  void A() {
9      if (c == '[') {
10         // Зчитуємо наступний символ
11         c = getchar();
12         // Рекурсивний виклик для нетермінала A
13         A();
14         while (c == '+') {
15             // Зчитуємо наступний символ
16             c = getchar();
17             // Рекурсивний виклик для нетермінал
                   а A
18             A();
19         }
20         // Перевірка на закриваючу дужку
21         if (c != ']') {
22             std::cerr << "Слово не належить мові
                   .\n";
23             exit(0);
24         }
25         // Зчитуємо наступний символ
26         c = getchar();
27     }
28     else if (c == 'x') {
29     // Зчитуємо наступний символ
30         c = getchar();
31     }
32     else {
33         std::cerr << "Слово не належить мові.\n"
               ;
34         exit(0);
35     }
36 }
37
38 int main() {
39     std::cout << "Введіть рядок для аналізу: ";
40     c = getchar(); // Зчитуємо перший символ
41
42     A(); // Виклик функції для нетермінала A
```

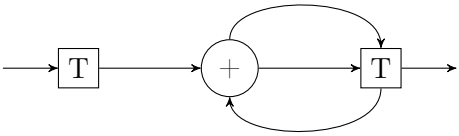
```
43
44      // Перевірка на кінець рядка
45      if (c == '\n') {
46          std::cout << "Слово належить мові.\n";
47      }
48      else {
49          std::cout << "Слово не належить мові.\n"
                ;
50      }
51
52      return 0;
53  }
```

Якщо на вхід програми подати рядок, вона проаналізує його і виведе результат про належність слова мові.

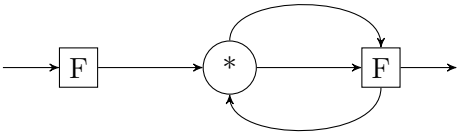
Побудуємо синтаксичні діаграми, які відповідають правилам P_1 граматики Gз прикладу 4:

$E \to T\{+T\}, \quad T \to F\{*F\}, , \quad F \to (E)|a.$

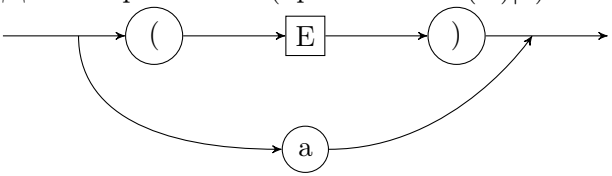
Для нетермінала E (правило $E \to T\{+T\}$):



Для нетермінала T (правило $T \to F\{*F\}$):



Для нетермінала F (правило $F \to (E)|a$):



На основі цих діаграм напишемо функції для рекурсивного спуску:

```
#include <iostream>
#include <cstdio>  // для getchar()
#include <cstdlib> // для exit()

char c; // Поточний символ

// Функції для нетерміналів
void T(); // Нетермінал T
void F(); // Нетермінал F

// Функція для обробки помилок
void error() {
    std::cerr << "Слово не належить мові.\n";
    exit(0);
}

// Нетермінал E
void E() {
    T(); // Виклик нетермінала T
    while (c == '+') {
        // Зчитуємо наступний символ
        c = getchar();
        // Виклик нетермінала T
        T();
    }
}

// Нетермінал T
void T() {
    F(); // Виклик нетермінала F
    while (c == '*') {
        // Зчитуємо наступний символ
        c = getchar();
        // Виклик нетермінала F
        F();
    }
}

// Нетермінал F
void F() {
```

```
41      if (c == '(') {
42          // Зчитуємо наступний символ
43          c = getchar();
44          // Виклик нетермінала E
45          E();
46          // Перевірка на закриваючу дужку
47          if (c != ')')
48              error();
49          // Зчитуємо наступний символ
50          c = getchar();
51      }
52      else if (c == 'a') {
53          // Зчитуємо наступний символ
54          c = getchar();
55      }
56      else {
57      // Помилка, якщо символ не відповідає правил
            ам
58          error();
59      }
60  }
61
62  int main() {
63      std::cout << "Введіть рядок для аналізу: ";
64      c = getchar(); // Зчитуємо перший символ
65
66      E(); // Виклик нетермінала E
67
68      // Перевірка на кінець рядка
69      if (c == '\n') {
70          std::cout << "Слово належить мові.\n";
71      }
72      else {
73          std::cout << "Слово не належить мові.\n"
                ;
74      }
75
76      return 0;
77  }
```

У цій програмі реалізовано метод рекурсивного спуску на основі синтаксичних діаграм. Спочатку викликається функція $E()$, яка відповідає головному нетерміналу. Вона, у свою чергу, викликає функцію $T()$ для аналізу доданків, а та — функцію $F()$ для аналізу множників. Функція $F()$ може викликати $E()$ через непряму рекурсію.

У подальшому синтаксичні діаграми можна використовувати не лише для розпізнавання рядків, але й для виконання обчислень.

Контрольні запитання та завдання

1. Синтаксичний аналіз та контекстно-вільні граматики. Дерева розбору

1. Що таке контекстно-вільна граматика?
2. Наведіть приклад правила контекстно-вільної граматики.
3. Що таке дерево розбору?
4. Які основні методи синтаксичного аналізу?
5. У чому різниця між низхідним і висхідним аналізом?
6. Наведіть приклад низхідного аналізу для заданої граматики.

LA(1)-граматики

1. Що таке LA(1)-граматика?
2. Наведіть приклад граматики, яка не є LA(1)-граматикою.
3. Наведіть приклад LA(1)-граматики.

4. Як перевірити, чи є граматика LA(1)-граматикою?
5. Як перетворити граматику на LA(1)-граматику?
6. Що таке множина ‘First‘ для правила граматики?
7. Як обчислити множину ‘First‘ для нетермінала?
8. Що таке множина ‘Follow‘ і як її визначити?

Ліворекурсивні та розширені правила. Синтаксичні діаграми

1. Що таке ліворекурсивне правило?
2. Чому ліворекурсивні правила ускладнюють синтаксичний аналіз?
3. Як усунути ліворекурсивні правила?
4. Що таке розширені правила?
5. Як побудувати синтаксичну діаграму для заданого правила?
6. Що таке метод рекурсивного спуску?
7. Як побудувати синтаксичний аналізатор за допомогою методу рекурсивного спуску?
8. Як побудувати синтаксичну діаграму для арифметичного виразу?
9. Як використовувати синтаксичні діаграми для побудови аналізатора?
10. Наведіть приклад синтаксичної діаграми для граматики арифметичних виразів.

Глосарій

Контекстно-вільна граматика – формальна граматика, у якій всі правила мають вигляд $A \to w$, де A – нетермінал, а w – рядок із терміналів та нетерміналів. Вона використовується для опису синтаксису мов програмування.

Дерево розбору – структура даних, яка представляє синтаксичну структуру ланцюжка, побудованого за правилами граматики. Корінь дерева – початковий символ граматики, листя – термінали, а внутрішні вузли – нетермінали.

Низхідний аналіз – метод синтаксичного аналізу, який починається з початкового символу граматики і розгортає правила до термінального ланцюжка.

Висхідний аналіз – метод синтаксичного аналізу, який починається з термінального ланцюжка і згортає його до початкового символу граматики.

LA(1)-граматика – граматика, яка дозволяє вибирати правило для розгортання на основі одного символу вперед (Look Ahead 1). Вона задовольняє умови, що множини ‘First‘ для різних правил не перетинаються, а також що множини ‘First‘ і ‘Follow‘ для нетерміналів не перетинаються.

Множина First – множина терміналів, які можуть починати рядки, виведені з даного нетермінала або правої частини правила.

Множина Follow – множина терміналів, які можуть іти за даним нетерміналом у виведенні.

Ліворекурсивне правило – правило вигляду $A \to Av$, яке може призводити до нескінченного циклу під час синтаксичного аналізу.

Розширені правила – правила, які дозволяють компактно описувати регулярні вирази, такі як $A \to u\{v\}$, що еквівалентно $A \to u \mid uv$.

Синтаксична діаграма – графічне зображення правил граматики, яке використовується для візуалізації процесу виведення ланцюжків.

Метод рекурсивного спуску – метод синтаксичного аналізу, який реалізує низхідний аналіз за допомогою рекурсивних функцій, що відповідають нетерміналам граматики.

Нетермінал – символ у граматиці, який може бути розгорнутий у послідовність інших символів (терміналів або нетерміналів).

Термінал – символ у граматиці, який не може бути розгорнутий далі. Він представляє конкретний елемент мови, наприклад літеру або число.

Синтаксичний аналізатор – програма або алгоритм, який перевіряє, чи належить вхідний рядок мові, заданій граматикою, і будує дерево розбору.

Еквівалентна граматика – граматика, яка описує ту саму мову, що й інша граматика, але може мати іншу структуру правил.

Правило продукції – правило у граматиці, яке описує, як нетермінал може бути замінений на послідовність інших символів.

Розгортання – процес заміни нетермінала на праву частину правила під час синтаксичного аналізу.

Згортання – процес заміни послідовності символів на нетермінал під час висхідного синтаксичного аналізу.

Однозначна граматика – граматика, для якої кожен ланцюжок має єдине дерево розбору.

Неоднозначна граматика – граматика, для якої існує хоча б один ланцюжок, який має більше одного дерева розбору.

ТЕМА VIII. Автоматні мови та та регулярні вирази

1 Регулярні та нерегулярні мови. Лема про накачку

1.1 Регулярні мови

Регулярні мови — це класи мов, які можна описати за допомогою регулярних виразів або розпізнати за допомогою скінченних автоматів [2].

Формально, мова $L \subseteq \Sigma^*$ називається регулярною, якщо існує скінченний автомат $M = (Q, \Sigma, \delta, q_0, F)$, для якого виконується $L = L(M)$.

Регулярні мови мають низку властивостей:

- замкнутість щодо операцій об'єднання, перетину, доповнення, конкатенації та ітерації;

- простий спосіб побудови регулярних виразів для їхнього опису.

Приклади регулярних мов включають множини ланцюжків з обмеженнями на їхню довжину, набір символів або певну структуру.

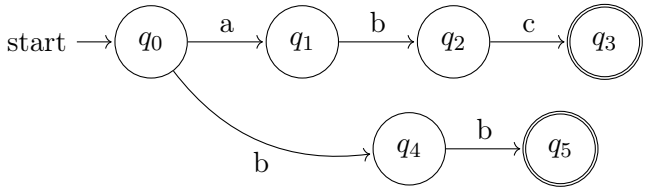
1.2 Нерегулярні мови

Нерегулярні мови — це мови, які не можна описати регулярними виразами або розпізнати за допомогою скінченних автоматів. Такі мови часто мають складніші структури, які вимагають більш потужних обчислювальних моделей, таких як контекстно-вільні граматики або машини Тюрінга. Один із найвідоміших прикладів нерегулярної мови — це мова $L = \{a^n b^n \mid n \geq 0\}$, яка не може бути розпізнана скінченним автоматом через обмеження пам'яті.

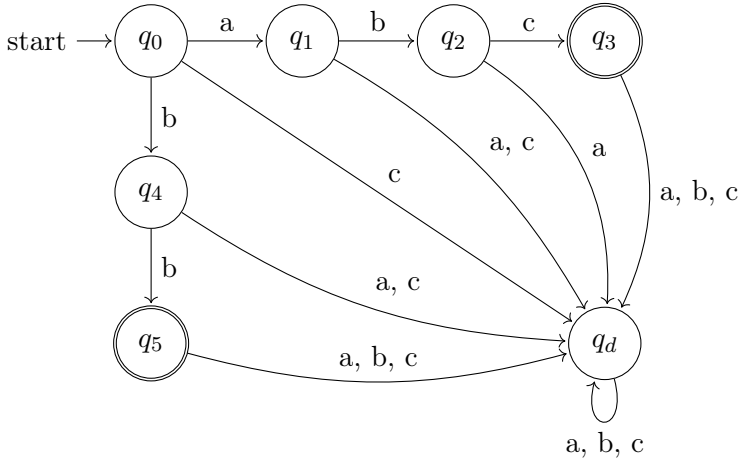
Приклади автоматів для деяких формальних мов

1. $\Sigma = \{a, b, c\}$; $L = \{abc, bb\}$ - регулярна мова, оскільки існує скінченний автомат, який описує цю мову.

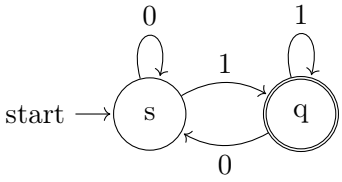
Детермінований скінченний автомат:



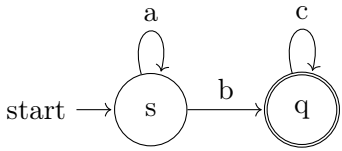
Повністю визначений скінченний автомат (доповнений мертвим станом q_d):



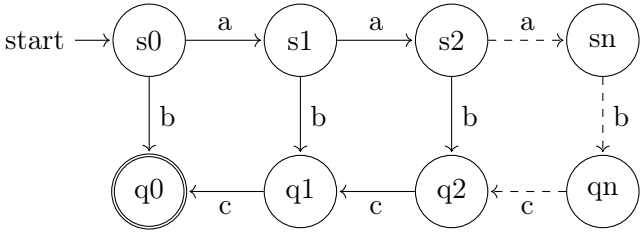
2. $\Sigma = \{0, 1\}$; L - множина парних двійкових чисел (регулярна мова)



3. $\Sigma = \{a, b, c\}$; $L = \{a^n bc^m \mid n, m \geq 0\}$ - регулярна мова



4. $\Sigma = \{a, b, c\}$; $L = \{a^n bc^n \mid n \geq 0\}$ - нерегулярна мова



Як бачимо, це невдала спроба побудувати автомат, оскільки ми можемо продовжувати праворуч будувати нові вершини і ребра скільки завгодно, нескінченно.

1.3 Лема про накачку

Лема про накачку є інструментом для доведення того, що мова нерегулярна [2, 6]. Формулювання леми таке:

Лема про накачку для регулярних мов: Нехай L — регулярна мова. Тоді існує така константа $p > 0$ (називається довжиною накачки), що для будь-якого слова $w \in L$ з $|w| \geq p$, можна представити w у вигляді $w = xyz$, де:

1. $|xy| \leq p$,
2. $|y| > 0$,
3. $xy^k z \in L$ для всіх $k \geq 0$.

Ця лема допомагає знаходити протиріччя, якщо припустити, що нерегулярна мова регулярна. Наприклад, для мови $L = \{a^n b^n \mid n \geq 0\}$ будь-який розподіл $w = xyz$ не дозволяє задовольнити умови леми при збільшенні k, що доводить нерегулярність L.

Доведення

Нехай L — регулярна мова. Це означає, що для L існує скінченний автомат $M = (Q, \Sigma, \delta, q_0, F)$, де:

- Q — скінченна множина станів;
- Σ — скінченний алфавіт;
- $\delta\colon Q \times \Sigma \to Q$ — функція переходів;
- $q_0 \in Q$ — початковий стан;
- $F \subseteq Q$ — множина заключних станів.

Позначимо p як кількість станів M, тобто $|Q| = p$. Розглянемо будь-яке слово $w \in L$, для якого $|w| \geq p$. Згідно з принципом Діріхле, під час зчитування слова w автоматом M щонайменше один стан повториться.

Розіб'ємо w на три частини $w = xyz$, де:

- x — префікс, що приводить автомат у перший повторюваний стан;
- y — частина слова, яка відповідає циклу (шляху між повторюваними станами);
- z — залишок, який доводить автомат до заключного стану.

Отже, слово w можна представити як xyz, де виконуються такі умови:

1. $xy^kz \in L$ для будь-якого $k \geq 0$;
2. $|xy| \leq p$;
3. $|y| \geq 1$.

Це і є твердження леми про накачку.

1.4 Доведення нерегулярності мов за допомогою леми про накачку

Приклад 1: Нехай L_{eq} — це мова рядків, у яких кількість нулів дорівнює кількості одиниць [6].

Припустимо, що L_{eq} є регулярною мовою. Розглянемо рядок $w = 0^n 1^n$, де $w \in L_{eq}$.

Відповідно до леми про накачку, рядок w можна подати у вигляді $w = xyz$, де виконується:

$$|xy| \leq n,\ y \neq \varepsilon, \text{ і для всіх } k \geq 0,\ xy^k z \in L_{eq}.$$

Розділимо $w = 0^n 1^n$ так, що x складається лише з нулів, y — теж тільки з нулів, а z включає решту нулів та всі одиниці:

$$w = \underbrace{000\ldots}_{x}\underbrace{0}_{y}\underbrace{0111\ldots11}_{z}.$$

Тепер розглянемо випадок $k = 2$. Тоді xy^2z матиме більше нулів, ніж одиниць:

$$xy^2z = \underbrace{000\ldots}_{x}\underbrace{00}_{y^2}\underbrace{0111\ldots11}_{z}.$$

Тому $xy^2z \notin L_{eq}$, оскільки кількість нулів перевищує кількість одиниць. Це суперечить припущенню про регулярність мови L_{eq}.

Отже, мова L_{eq} нерегулярна.

Приклад 2:

Розглянемо мову $L = \{a^n b^n \mid n \geq 0\}$. Доведемо, що L не є регулярною за допомогою леми про накачку.

Припустимо, що L регулярна. Тоді існує число p, таке що будь-яке слово $w \in L$ з довжиною $|w| \geq p$ можна розбити на $w = xyz$, де виконуються умови леми.

Візьмемо слово $w = a^p b^p$. Згідно з умовою, $|xy| \leq p$, тому x і y складаються лише із символів a. Нехай $y = a^k$, де $k \geq 1$.

Тепер розглянемо слово $w' = xy^2z = a^{p+k}b^p$. Оскільки $|a| \neq |b|$, то $w' \notin L$. Це суперечить припущенню про регулярність L. Отже, L нерегулярна.

2 Перевірка еквівалентності регулярних мов

Еквівалентність регулярних мов можна перевірити за допомогою алгоритму заповнення таблиці нееквівалентних станів. Цей метод дозволяє порівнювати мови, навіть якщо вони представлені різними способами, наприклад, одним — регулярним виразом, а іншим — недетермінованим скінченним автоматом.

2.1 Перетворення представлень

Спочатку необхідно перетворити кожне представлення в детермінований скінчений автомат (ДСА). Це стандартний крок у теорії формальних мов та автоматів.

Після цього можна уявити собі ДСА, множина станів якого є об'єднанням множин станів для мов L_1 і L_2. Технічно цей ДСА має два початкових стану, але при перевірці еквівалентності конкретний початковий стан не має значення. Отже, можна вибрати будь-який з них як єдиний початковий стан.

2.2 Алгоритм перевірки еквівалентності

Далі перевіряємо еквівалентність початкових станів двох заданих ДСА, використовуючи алгоритм заповнення таблиці нееквівалентних станів. Якщо вони еквівалентні, то $L_1 = L_2$, а якщо ні, то $L_1 \neq L_2$.

Далі перевіряємо, чи є початкові стани двох заданих ДСА еквівалентними, використовуючи алгоритм заповнення таблиці еквівалентних станів. Якщо вони еквівалентні, то можна зробити висновок, що $L_1 = L_2$. В іншому випадку, якщо початкові стани не є еквівалентними, то $L_1 \neq L_2$.

2.3 Алгоритм заповнення таблиці нееквівалентних станів

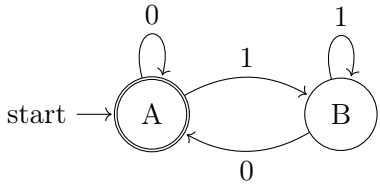
1. *Ініціалізація*: Спочатку створюємо таблицю, яка містить всі пари станів з обох ДСА. Позначаємо всі пари, які очевидно нееквівалентні (наприклад, коли один стан є заключним, а інший — ні).

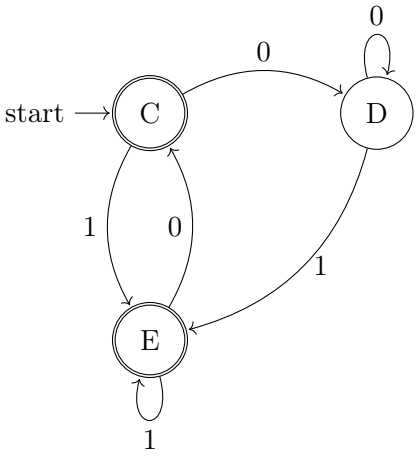
2. *Заповнення таблиці*: Рекурсивно заповнюємо таблицю, додаючи нові пари нееквівалентних станів на основі вже відомих нееквівалентних пар. Для цього розглядаємо кожну пару станів (p, q) і перевіряємо, чи для всіх символів алфавіту переходи з p та q ведуть до нееквівалентних станів.

3. *Завершення*: Коли більше немає нових пар нееквівалентних станів, алгоритм завершується. Якщо початкові стани обох ДСА не потрапили до списку нееквівалентних пар, то мови L_1 і L_2 еквівалентні.

Приклад

Розглянемо два детермінованих скінченних автомати (ДСА). Обидва ДСА допускають порожній ланцюжок та всі ланцюжки, які закінчуються символом 0. Це відповідає мові, описаній регулярним виразом $(0+1)^*0+\varepsilon$. Припустимо, що на рисунку зображено один з таких ДСА, який має п'ять станів, позначених від A до E [2,4].





Для того, щоб виконати заповнення таблиці, спочатку потрібно позначити символом × усі ті комірки, які відповідають парам станів, де лише один стан є заключним, а другий - ні.

	A	B	C	D
B	×			
C		×		
D			×	
E	×		×	×

У результаті цього аналізу можна побачити, що інші чотири пари $\{A, C\}, \{A, D\}, \{B, D\}$ і $\{B, E\}$ є парами еквівалентних станів.

Під час одного циклу розглядаються всі можливі пари станів з метою перевірки, чи є серед них пара станів-спадкоємців, яка може бути розрізнена.

Оскільки перевірка показала, що стани A і C еквівалентні та початкові в обох автоматах, можна зробити висновок, що ці детерміновані скінченні автомати обробляють одну й ту ж мову.

Отже, алгоритм заповнення таблиці нееквівалентних станів дозволяє ефективно перевіряти еквівалентність регуляр-

них мов, незалежно від того, як вони представлені (регулярним виразом, недетермінованим або детермінованим скінченним автоматом).

3 Перетворення ДСК у регулярний вираз шляхом вилучення станів

Узагальнений скінченний автомат – це модифікація звичайного скінченного автомата, де переходи між станами позначаються не окремими символами, а регулярними виразами. Мітка шляху такого автомата визначається як конкатенація регулярних виразів, розташованих на переходах. Слово w допускається узагальненим автоматом, якщо воно належить мові, що задається міткою одного з успішних шляхів (шлях, який завершується у заключному стані).

Якщо в автоматі існує декілька переходів із загальним початком і кінцем (паралельні переходи), їх можна об'єднати в один перехід за допомогою операції $+$.

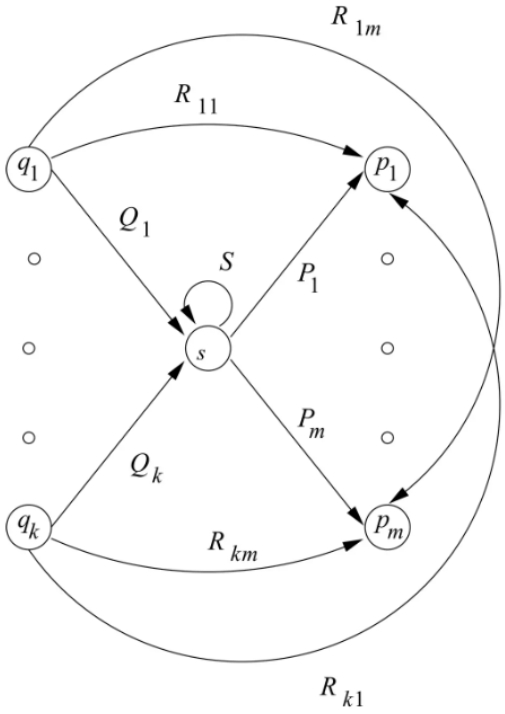
Нехай дано скінченний автомат $A = (Q, \Sigma, \delta, q_0, F)$, який має n заключних станів. Для його перетворення спочатку будуємо еквівалентний узагальнений автомат A_0, видаляючи всі незаключні стани, крім початкового. Далі для кожного заключного стану $q \in F$ створюємо скорочений узагальнений автомат.

Кожному скороченому автомату $A_1, \ldots, A_n$ відповідає регулярний вираз $R_1, \ldots, R_n$. Підсумковий регулярний вираз R для автомата A має вигляд:

$$R = R_1 + R_2 + \cdots + R_n.$$

Розглянемо процес вилучення станів:

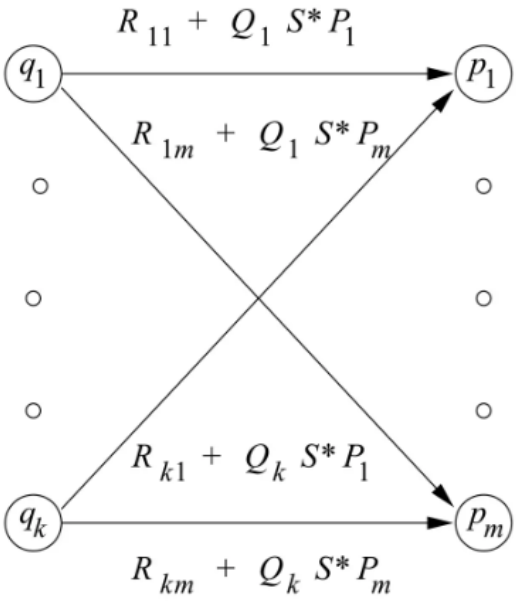
Для виключення стану s необхідно оновити дуги між усіма парами станів q і p, які з'єднуються через s, використовуючи всі можливі шляхи. Оскільки кількість таких шляхів може бути нескінченною, вони описуються за допомогою регулярних виразів.



Позначимо [6]:

- $q_1, \ldots, q_k$ – стани, що передують s,
- $p_1, \ldots, p_m$ – стани, що йдуть після s,
- Q_i – регулярний вираз, що позначає дугу між q_i та s,
- P_j – регулярний вираз, що позначає дугу між s та p_j,
- S – петля на стані s (якщо така існує),
- R_{ij} – регулярний вираз, що описує дугу між q_i і p_j (або 0, якщо дуга відсутня).

Після вилучення стану s отримуємо такий автомат:



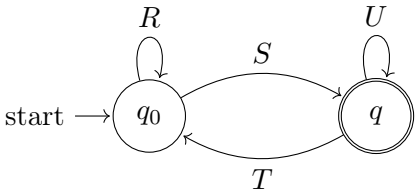
Регулярний вираз для всіх шляхів через s [6]:

$$R_{ij} = R_{ij} + Q_i S^* P_j,$$

де S^* враховує повторні проходи через петлю на стані s.

3.1 Стратегія перетворення автомата у регулярний вираз

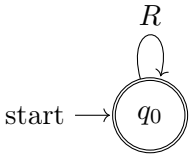
1. Для кожного заключного стану $q \neq q_0$ створюємо скорочений автомат A_q, у якому залишається лише початковий стан q_0 та сам q. Методом вилучення станів створюємо автомат з дугами, позначеними регулярними виразами [6].



Відповідний регулярний вираз:

$$E_q = (R + SU^*T)^* \, SU^*.$$

2. Якщо початковий стан q_0 заключний, усі інші стани видаляються, отримуємо автомат з одним станом:



Регулярний вираз:

$$E_q = R^*.$$

3. Остаточний вираз отримується об'єднанням:

$$R = \bigoplus_{q \in F} E_q.$$

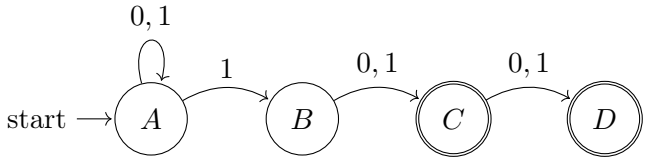
3.2 Приклад

Розглянемо автомат, що допускає ланцюжки з 0 та 1, у яких на другій або третій позиції з кінця знаходиться одиниця [6].

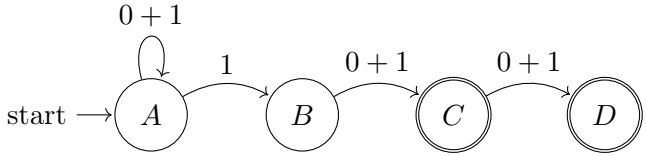
Автомат:

$$L(\mathcal{A}) = \{w \mid w = x1b \text{ або } w = x1bc,$$

$$x \in \{0,1\}^*, \, \{b,c\} \subseteq \{0,1\}\}.$$



Перепишемо автомат, позначивши дуги регулярними виразами.



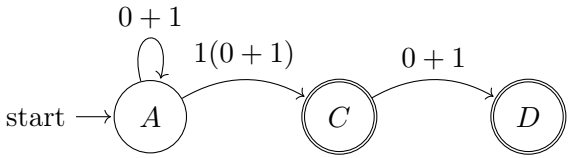
Зазначимо, що стан B не буде присутнім ні в одному з скорочених автоматів (для стану C і D). Він не заключний і не початковий. Виключимо його до того, як будемо будувати скорочені автомати для заключних станів C і D.

Є один стан A, який передує B і один стан C, який іде за B. Тому $Q_1 = 1, P_1 = 0 + 1, R_{11} = \varnothing$ (немає шляху з A в C), $S = \varnothing$ (немає петлі в стані B). Підставимо отримані значення у вираз

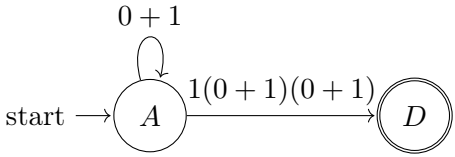
$$R_{11} + Q_1 S^*(P_1)$$

і одержимо новий вираз над дугою зі стану A в стан C: $\varnothing + 1\varnothing^*(0 + 1) = 1(0 + 1)$.

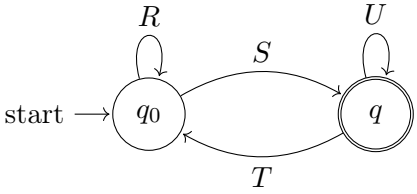
Скорочений автомат A_{CD} без незаключного стану B має вигляд:



Виключимо стан C і отримаємо скорочений автомат A_D:



Порівняємо отриманий автомат з автоматом, який було описано в алгоритмі:



Тут $R = 0 + 1, S = 1(0 + 1)(0 + 1), T = \varnothing, U = \varnothing \Rightarrow U^* = \varepsilon, SU^*T = \varnothing$, оскільки $T = \varnothing$. Тобто, $(R+SU^*T)^*SU^* = R^*S = (0 + 1)^*1(0 + 1)(0 + 1)$.

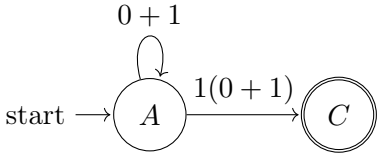
Вираз

$$E_D = (0 + 1)^*1(0 + 1)(0 + 1)$$

описує ланцюжки, в яких на третій позиції з кінця стоїть 1.

Для заключного стану C треба скоротити стан D в автоматі A_{CD}. Оскільки після D немає станів p_i, то треба просто вилучити дугу, яка веде з C в D разом зі станом D.

З автомату A_{CD} отримаємо скорочений автомат A_C



і відповідний регулярний вираз

$$E_C = (0 + 1)^*1(0 + 1).$$

Цей вираз описує ланцюжки, в яких другий з кінця символ дорівнює 1.

Остаточний результуючий регулярний вираз:

$$E_D + E_C = (0+1)^*1(0+1)(0+1) + (0+1)^*1(0+1)$$

об'єднання скорочених автоматів для заключних станів C і D.

Порядок вилучення станів:

1. Спочатку вилучаються всі незаключні стани, які не є початковими, щоб не повторювати цю процедуру для кожного заключного стану (в прикладі - це стан B).

2. Для заключних станів іноді деякі дії по виключенню треба повторювати, але частину з них можна сумістити. Наприклад, нехай автомат має 3 заключні стани p, r, q.

   (а) Спочатку можна виключити p, потім, виключивши q, отримаємо скорочений автомат для r, а виключивши r, - скорочений автомат для q. Тобто для двох скорочених автоматів стан p виключався 1 раз.

   (б) Щоб побудувати автомат для p треба повернутися до початкового автомату зі станами p, r, q, з якого виключити стани r і q.

4 Перетворення НСА в регулярний вираз за допомогою рекурентних формул

Нехай $M = (Q, \Sigma, \delta, q_0, F)$ — недетермінований скінченний автомат, описаний через діаграму переходів. Завдання полягає в побудові регулярного виразу R, що описує множину рядків, якими позначаються шляхи на діаграмі автомата M, що ведуть від початкового стану до заключних (тобто $L(R) = L(M)$). Для цього необхідно вибрати ці шляхи серед

усіх можливих, що проходять через певну підмножину станів автомата, яка поступово розширюється.

На початковому етапі формуються регулярні вирази для найпростішого випадку — шляхів, які не мають проміжних станів, тобто являють собою лише вершини або дуги графу. Далі, індуктивно, будуємо регулярні вирази для шляхів, що мають один проміжний стан, і продовжуємо до складніших випадків. У кінцевому підсумку отримуємо регулярні вирази для всіх можливих шляхів автомата.

4.1 Алгоритм побудови

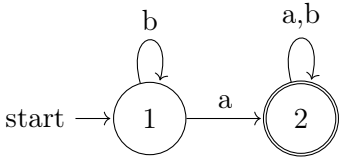
- Нумеруємо стани автомата M натуральними числами. $Q = \{1, \ldots, n\}$, де стан 1 — початковий.
- Позначимо через $R_{ij}(k)$ регулярний вираз, який описує множину шляхів від стану i до стану j, що не мають проміжних станів із номерами більшими за k (початковий та кінцевий стани не проміжні).
- Поетапно обчислюємо для кожної пари i, j станів автомата вирази $R_{ij}(0), \ldots, R_{ij}(n)$.
- В кінці, для отримання остаточного регулярного виразу, беремо альтернативу всіх виразів $R_{ij}(n)$, що представляють шляхи від початкового стану до заключних.

Рекурентне обчислення $R_{ij}(k)$ для всіх пар станів автомата [6]:

- Для $k = 0$: $R_{ij}(0)$ — регулярний вираз, що описує шлях від стану i до стану j, без проміжних станів.
- Для $k \neq 0$: $R_{ij}(k) = R_{ij}(k-1) + R_{ik}(k-1)(R_{kk}(k-1))^* R_{kj}(k-1)$ — регулярний вираз, що включає всі можливі шляхи, що проходять через проміжні стани.

4.2 Приклад

Розглянемо приклад детермінованого скінченного автомата в алфавіті $\Sigma = \{a, b\}$, який приймає рядки, що містять хоча б один символ a. Автомат задано діаграмою переходів:



Згідно з алгоритмом, для кожної пари станів автомата обчислюємо вирази $R_{ij}(0)$, $R_{ij}(1)$, $R_{ij}(2)$:

$$k = 0: \quad R_{11}(0) = \varepsilon + b, \quad R_{12}(0) = a,$$
$$R_{21}(0) = \varnothing, \quad R_{22}(0) = \varepsilon + a + b.$$

$$k = 1: \quad R_{ij}(1) = R_i(0) + R_{i1}(0)\,(R_{11}(0))^*\,R_{1j}(0)$$

$$R_{11}(1) = \varepsilon + b + (\varepsilon + b)(\varepsilon + b)^*(\varepsilon + b) = b^*$$
$$R_{12}(1) = a + (\varepsilon + b)(\varepsilon + b)^* a = b^* a$$
$$R_{21}(1) = \varnothing + \varnothing(\varepsilon + b)^*(\varepsilon + b) = \varnothing$$
$$R_{22}(1) = \varepsilon + a + b + \varnothing(\varepsilon + b)^* a = \varepsilon + a + b$$

$$k = 2: \quad R_{ij}(2) = R_i(1) + R_{i2}(1)\,(R_{22}(1))^*\,R_{2j}(1)$$

$$R_{11}(2) = b^* + b^* a(\varepsilon + a + b)^* \varnothing = b^*$$
$$R_{12}(2) = b^* a + b^* a(\varepsilon + a + b)^*(\varepsilon + a + b) = b^* a(a + b)^*$$
$$R_{21}(2) = \varnothing + (\varepsilon + a + b)(\varepsilon + a + b)^* \varnothing = \varnothing$$
$$R_{22}(2) = \varepsilon + a + b + (\varepsilon + a + b)(\varepsilon + a + b)^*(\varepsilon + a + b) = (a + b)^*$$

Під час виконання обчислень використовувалися основні властивості та тотожності для регулярних виразів.

Підсумковий регулярний вираз записується так:

$$R = \bigoplus_{j \in F} R_{1j}(n),$$

що являє собою об'єднання регулярних виразів, які описують всі можливі шляхи від початкового стану до заключних станів автомата.

Одержаний регулярний вираз:

$$R_{12}(2) = b^* a (a + b)^*.$$

5 Перетворення праволінійної граматики у НСА

5.1 Перетворення праволінійної граматики у граматику спеціального вигляду

Перетворення праволінійної граматики у граматику спеціального вигляду застосовується для подальшого використання граматики в алгоритмах аналізу, наприклад, для побудови скінченного автомата чи перевірки належності рядків мові.

Праволінійна граматика має правила вигляду:

$$A \to xB, \quad A \to x, \quad A, B \in N,\, x \in \Sigma^*,$$

де N — множина нетермінальних символів, а Σ — множина термінальних символів.

Це забезпечує зручність подальшого аналізу та обробки граматики.

Граматика спеціального вигляду [2, 4] має правила вигляду

$$A \to aB \quad \text{або} \quad A \to a, \quad A, B \in N, a \in T.$$

Граматика спеціального вигляду детальніша та включає додаткові правила, які:

1. Замінюють термінальні символи, що йдуть послідовно, на проміжні нетермінали.

2. Розбивають складні правила на послідовність простіших правил.

Покажемо на прикладі, як перетворюється праволінійна граматика у граматику спеціального вигляду.

Розглянемо праволінійну граматику, що задана правилами:

1. $\langle S\rangle \to x\langle A\rangle$,
2. $\langle S\rangle \to yz$,
3. $\langle S\rangle \to \langle A\rangle$,
4. $\langle A\rangle \to xyy\langle S\rangle$,
5. $\langle A\rangle \to z\langle A\rangle$,
6. $\langle A\rangle \to \varepsilon$.

Ця граматика визначає мову, що містить рядки над алфавітом $\{x, y, z\}$, з різними комбінаціями термінальних і нетермінальних символів. Наша мета — перетворити її у граматику спеціального вигляду.

Перетворення виконується в кілька кроків:

1. Розбиваємо складні правила з термінальними символами, які йдуть послідовно (наприклад, yz або xyy), на прості правила з використанням проміжних нетерміналів.

2. Додаємо нові нетермінальні символи для кожного розбитого термінального фрагмента.

3. Зберігаємо правила, які вже відповідають спеціальному вигляду.

1. Друге правило $\langle S\rangle \to yz$ розбите на:

$$\langle S\rangle \to y\langle zEpsilon\rangle,$$

$$\langle zEpsilon\rangle \to z\langle Epsilon\rangle,$$

$$\langle Epsilon \rangle \to \varepsilon$$

Це дозволяє обробляти yz послідовно.

2. Правило четверте $\langle A \rangle \to xyy\langle S \rangle$ розбите на:

$$\langle A \rangle \to x\langle yyS \rangle, \quad \langle yyS \rangle \to y\langle yS \rangle, \quad \langle yS \rangle \to y\langle S \rangle.$$

3. Третє правило $\langle S \rangle \to \langle A \rangle$ замінимо на 3 правила, підставляючи праві частини нетермінала $\langle A \rangle$, замість самого нетермінала $\langle A \rangle$:

$$\langle S \rangle \to x\langle yyS \rangle, \quad \langle S \rangle \to z\langle A \rangle, \quad \langle S \rangle \to \varepsilon.$$

4. Інші правила, такі як $\langle S \rangle \to x\langle A \rangle$, $\langle A \rangle \to z\langle A \rangle$ і $\langle A \rangle \to \varepsilon$, залишаються без змін, оскільки вони вже відповідають спеціальному вигляду.

Після перетворення отримаємо граматику спеціального вигляду:

$$\langle S \rangle \to x\langle A \rangle$$
$$\langle S \rangle \to y\langle zEpsilon \rangle$$
$$\langle zEpsilon \rangle \to z\langle Epsilon \rangle$$
$$\langle Epsilon \rangle \to \varepsilon$$
$$\langle S \rangle \to x\langle yyS \rangle$$
$$\langle yyS \rangle \to y\langle yS \rangle$$
$$\langle yS \rangle \to y\langle S \rangle$$
$$\langle S \rangle \to z\langle A \rangle$$
$$\langle S \rangle \to \varepsilon$$
$$\langle A \rangle \to x\langle yyS \rangle$$
$$\langle A \rangle \to z\langle A \rangle$$
$$\langle A \rangle \to \varepsilon$$

5.2 Перетворення граматики спеціального вигляду у НСА

Отримана граматика спеціального вигляду дозволяє зручно аналізувати та обробляти рядки, що належать мові. Вона складається лише з простих правил, де кожне правило має максимум один термінальний символ перед нетермінальним. Це полегшує перетворення граматики у скінченний автомат чи інші формальні моделі.

Цей автомат приймає всі рядки, що належать мові описаній граматикою:

$$M = (Q, \Sigma, \delta, q_0, F)$$

де:

- Q — множина станів:

$$Q = \{\langle S\rangle, \langle A\rangle, \langle zEpsilon\rangle, \langle Epsilon\rangle, \langle yyS\rangle, \langle yS\rangle\}$$

- Σ — алфавіт введення:

$$\Sigma = \{x, y, z\}$$

- δ — функція переходів:

$$\begin{aligned}
&\delta(\langle S\rangle, x) = \{\langle A\rangle, \langle yyS\rangle\},\\
&\delta(\langle S\rangle, y) = \{\langle zEpsilon\rangle\},\\
&\delta(\langle S\rangle, z) = \{\langle A\rangle\},\\
&\delta(\langle A\rangle, x) = \{\langle yyS\rangle\},\\
&\delta(\langle zEpsilon\rangle, z) = \{\langle Epsilon\rangle\},\\
&\delta(\langle yyS\rangle, y) = \{\langle yS\rangle\},\\
&\delta(\langle yS\rangle, y) = \{\langle S\rangle\}.
\end{aligned}$$

- q_0 — початковий стан:

$$q_0 = \langle S\rangle$$

- F — множина заключних станів:

$$F = \{\langle S\rangle, \langle A\rangle, \langle Epsilon\rangle\}$$

State	x	y	z	Final
$\langle S\rangle$	$\langle A\rangle$, $\langle yyS\rangle$	$\langle zEpsilon\rangle$	$\langle A\rangle$	Yes
$\langle A\rangle$	$\langle yyS\rangle$	$\varnothing$	$\langle A\rangle$	Yes
$\langle zEpsilon\rangle$	$\varnothing$	$\varnothing$	$\langle Epsilon\rangle$	No
$\langle Epsilon\rangle$	$\varnothing$	$\varnothing$	$\varnothing$	Yes
$\langle yyS\rangle$	$\varnothing$	$\langle yS\rangle$	$\varnothing$	No
$\langle yS\rangle$	$\varnothing$	$\langle S\rangle$	$\varnothing$	No

Контрольні запитання та завдання

1. Регулярні множини і регулярні вирази

1. Що таке регулярна мова? Наведіть формальне визначення.

2. Наведіть приклад регулярної мови та побудуйте для неї скінченний автомат.

3. Що таке нерегулярна мова? Наведіть приклад.

4. Чому мова $L = \{a^n b^n \mid n \geq 0\}$ нерегулярна?

5. Які методи використовуються для доведення нерегулярності мови?

6. Наведіть приклад нерегулярної мови та поясніть, чому вона не може бути описана скінченним автоматом.

7. Сформулюйте лему про накачку для регулярних мов.

8. Як лема про накачку допомагає довести нерегулярність мови?

9. Наведіть приклад мови та доведіть її нерегулярність за допомогою леми про накачку.

10. Чому лема про накачку не може бути використана для доведення регулярності мови?

2. Перевірка еквівалентності регулярних мов

1. Які способи представлення регулярних мов ви знаєте?

2. Як перетворити регулярний вираз у детермінований скінченний автомат?

3. Наведіть приклад перетворення регулярного виразу у скінченний автомат.

4. Як працює алгоритм заповнення таблиці нееквівалентних станів?

5. Наведіть приклад двох автоматів та перевірте їх еквівалентність за допомогою цього алгоритму.

6. Чому важливо перевіряти еквівалентність регулярних мов?

3. Перетворення ДСК у регулярний вираз шляхом вилучення станів

1. Як вилучення станів допомагає отримати регулярний вираз?

2. Які кроки необхідно виконати для вилучення стану з автомата?

3. Наведіть приклад автомата та перетворіть його у регулярний вираз шляхом вилучення станів.

4. Як обробляються петлі на станах під час вилучення?

5. Чому важливо правильно вибирати порядок вилучення станів?

4. Перетворення НСА в регулярний вираз за допомогою рекурентних формул

1. Як працює алгоритм побудови регулярного виразу за допомогою рекурентних формул?

2. Які переваги має цей метод порівняно з іншими способами перетворення?

3. Наведіть приклад автомата та побудуйте регулярний вираз за допомогою рекурентних формул.

4. Як обробляються цикли в автоматі під час побудови регулярного виразу?

5. Перетворення праволінійної граматики у НСА

1. Що таке праволінійна граматика? Наведіть приклад.

2. Як перетворити праволінійну граматику у граматику спеціального вигляду?

3. Як перетворити граматику спеціального вигляду у недетермінований скінченний автомат?

Глосарій

Регулярна мова — мова, яка може бути описана за допомогою регулярних виразів або розпізнана скінченним автоматом. Приклади включають мови з обмеженнями на довжину, символи або структуру.

Нерегулярна мова — мова, яка не може бути описана регулярними виразами або розпізнана скінченним автоматом. Прикладом є мова $L = \{a^n b^n \mid n \geq 0\}$.

Лема про накачку — інструмент для доведення нерегулярності мови. Вона стверджує, що для будь-якої регулярної мови існує константа p, така що будь-яке слово довжиною $\geq p$ можна розбити на xyz, де $|xy| \leq p$, $|y| > 0$, і $xy^k z$ належить мові для всіх $k \geq 0$.

Скінченний автомат — математична модель, яка розпізнає регулярні мови. Включає множину станів, алфавіт, функцію переходів, початковий стан та множину заключних станів.

Недетермінований скінченний автомат (НСА) — скінченний автомат, у якому для кожного стану та символу може існувати кілька можливих переходів.

Еквівалентність регулярних мов — дві регулярні мови вважаються еквівалентними, якщо вони описують одну й ту саму множину слів.

Праволінійна граматика — граматика, у якій всі правила мають вигляд $A \to xB$ або $A \to x$, де A, B — нетермінали, а x — термінал або порожній ланцюжок.

Граматика спеціального вигляду — граматика, у якій кожне правило має вигляд $A \to aB$ або $A \to a$, де A, B — нетермінали, а a — термінал.

ВИСНОВКИ

Теорія формальних мов і скінченних автоматів є основою для побудови мовних процесорів, компіляторів та аналізаторів синтаксису.

Розробка мовних процесорів є складним багатоступеневим процесом, що включає аналіз, трансформацію, оптимізацію та генерацію коду. Успішне створення компілятора чи інтерпретатора потребує глибокого розуміння лексичного, синтаксичного та семантичного аналізу, а також алгоритмів роботи зі структурами даних.

Формальні мови, регулярні множини, граматики Хомського та скінченні автомати є основою для створення мовних процесорів. Лексичний аналіз дозволяє розділити вихідний код на складові частини, синтаксичний аналіз перевіряє його відповідність правилам мови, а семантичний аналіз забезпечує логічну правильність програми. Генерація проміжного та машинного коду забезпечує кінцевий результат у вигляді виконуваного коду. Крім того, оптимізація дозволяє поліпшити продуктивність програм, зменшити їхній розмір та підвищити ефективність виконання.

Знання, викладені у цьому посібнику, важливі для розробки компіляторів та інтерпретаторів. Вони можуть бути застосовані для створення власних мов програмування, автоматизованої обробки тексту та аналізу програмного коду. Розуміння принципів побудови мовних процесорів також допоможе у створенні ефективного коду, що необхідне для сучасного програмного забезпечення.

Матеріали цього посібника можуть слугувати основою для подальшого вивчення компіляторів, мов програмування та алгоритмів обробки тексту, а також допоможуть студентам та розробникам у розумінні принципів роботи мовних процесорів.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

[1] *Сопронюк, Т. М.* Системне програмування. Частина I. Елементи теорії формальних мов : навчальний посібник у двох частинах. Чернівці : Чернівец. нац. ун-т ім. Ю. Федьковича, 2008. 84 с.

[2] *Сопронюк, Т. М.* Системне програмування. Частина II. Елементи теорії компіляції : навчальний посібник у двох частинах. Чернівці : Чернівец. нац. ун-т ім. Ю. Федьковича, 2008. 84 с.

[3] *Сопронюк, Т. М., Сопронюк, А. Ю., Дробот, А. В.* Фази побудови мовного процесора для платформи .NET // Буковинський математичний журнал. 2023. Т.11. № 2. С. 71–84. URL : https://doi.org/10.31861/bmj2023.02.07

[4] *Aho, A. V., Ullman, J. D.* The Theory of Parsing, Translation, and Compiling. Volume I: Parsing. Englewood Cliffs, NJ : Prentice-Hall, 1972. 542 p. URL : https://file.fouladi.ir/courses/compiler/books/theory_of_parsing_vol1.pdf

[5] *Aho, A. V., Ullman, J. D.* The Theory of Parsing, Translation, and Compiling. Volume II: Compiling. Englewood Cliffs, NJ : Prentice-Hall, 1973. 740 p. URL : https://www-2.dc.uba.ar/staff/becher/Aho-Ullman-Parsing-V2.pdf

[6] *Hopcroft, J. E., Motwani, R., Ullman, J. D.* Introduction to Automata Theory, Languages, and Computation. 3rd ed. Boston : Addison-Wesley, 2006. 521 p. URL : https://users.encs.concordia.ca/~grahne/hmu_slides/

[7] *Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D.* Compilers: Principles, Techniques, and Tools. 2nd ed. Boston : Addison-Wesley, 2006. 1009 p.

[8] *Aho, A. V., Sethi, R., Ullman, J. D.* Compilers: Principles, Techniques, and Tools. 1st ed. Reading : Addison-Wesley, 1986. 796 p.

[9] *Sipser, M.* Introduction to the Theory of Computation. 3rd ed. Boston : Cengage Learning, 2012. 504 p.

[10] *Grune, D., van Reeuwijk, K., Bal, H. E., Jacobs, C. J. H., Langendoen, K.* Modern Compiler Design. 2nd ed. New York : Springer, 2012. 822 p.

[11] *Appel, A. W.* Modern Compiler Implementation in C. Cambridge : Cambridge University Press, 2004. 560 p.

[12] *Cooper, K. D., Torczon, L.* Engineering a Compiler. 2nd ed. Burlington : Morgan Kaufmann, 2011. 824 p.

[13] *Lawson, M. V.* Finite Automata. Edinburgh : Heriot-Watt University, 2009. 250 p.

[14] *Sippu, S., Soisalon-Soininen, E.* Parsing Techniques: A Practical Guide. New York : Springer, 1990. 539 p.

[15] *Гавриленко, С. Ю.* Формальні мови, граматики та автомати : навчальний посібник. Харків : НТУ «ХПІ», 2021. 133 с. URL : `https://repository.kpi.kharkov.ua/server/api/core/bitstreams/5847efdd-6ff5-4f7f-9de8-6f6555ad4cc0/content`

[16] *Спекторський, І. Я., Статкевич, В. М.* Формальні мови та автомати. Київ : КПІ ім. Ігоря Сікорського, 2019. 167 с.

[17] *Гавриленко, С. Ю., Клименко, А. М., Любченко, Н. Ю. та ін.* Теорія цифрових автоматів та формальних мов. Харків : НТУ «ХПІ», 2010. 176 с.

[18] *Гавриленко, С. Ю., Клименко, А. М., Носков, В. І.* Логіка дискретних автоматів : навч. посіб. Харків : НТУ «ХПІ», 2014. 129 с.

[19] *Захарія, Л. М., Заяць, М. М.* Формальні мови та граматики : навч. посіб. Львів : Львівська політехніка, 2016. 196 с.

[20] *Гаврилків, В. М.* Формальні мови та алгоритмічні моделі : навч. посіб. Івано-Франківськ : Сімик, 2012. 172 с. URL : http://gavrylkiv.pu.if.ua/papers/FormalLanguages.pdf

[21] *Паранчич М. Ю., Сопронюк Т.М.* Навчальний тренажер для операцій з недетермінованими скінченними автоматами.: матеріали Міжнародної наукової інтернетконференції "Інформаційне суспільство: технологічні, економічні та технічні аспекти становлення", випуск 96 (м. Тернопіль, Україна, м. Ополе, Польща, 11-12 лютого 2025 р.), 2025. С. 34-36. URL : http://www.konferenciaonline.org.ua/ua/article/id-2090/

[22] *Вікіпедія. Формальні граматики.* URL : http://uk.wikipedia.org/wiki/Формальні_граматики

[23] *Вікіпедія. Скінченний автомат.* URL : https://uk.wikipedia.org/wiki/Скінченний_автомат

ДЛЯ НОТАТОК

ДЛЯ НОТАТОК

Навчальне видання

Сопронюк *Тетяна Миколаївна*

МОВНІ ПРОЦЕСОРИ ТА ФОРМАЛЬНІ МОВИ: ВІД ТЕОРІЇ ДО ПРАКТИКИ

Навчальний посібник

Літературне редагування ***Колодій О.В.***

Дизайн обкладинки ***Кудрінська О.М.***

Підписано до друку 04.03.2025. Формат 60 х 84/16.
Папір офсетний. Друк різографічний. Ум.-друк. арк. 10,8.
Обл. вид. арк. 11,6. Зам. Н-017.
Видавництво та друкарня Чернівецького національного університету
58002, Чернівці, вул. Коцюбинського, 2

Свідоцтво суб'єкта видавничої справи ДК №891 від 08.04.2002 р.