|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ЗМІСТ [**ВСТУП** 2](#_Toc5144851)  [**1 ФОРМАЛЬНІ МОВИ ТА ГРАМАТИКИ** 3](#_Toc5144852)  [1.1 Мови та ланцюжки символів. Способи задання мов 3](#_Toc5144853)  [1.2 Граматики і розпізнавачі 8](#_Toc5144854)  [1.3 Класифікація мов та граматик 14](#_Toc5144855)  [**2 ЛЕКСИЧНІ АНАЛІЗАТОРИ** 18](#_Toc5144856)  [2.1 Визначення та призначення лексичного аналізатора 18](#_Toc5144857)  [2.2 Принцип побудови лексичного аналізатора 20](#_Toc5144858)  [**3 РЕГУЛЯРНІ ГРАМАТИКИ ТА СКІНЧЕННІ АВТОМАТИ** 24](#_Toc5144859)  [3.1 Регулярні та автоматні граматики 24](#_Toc5144860)  [3.2 Формальне визначення скінченного автомату. Відмінність між детермінованим та недетермінованим скінченним автоматом 25](#_Toc5144861)  [3.3 Призначення та види подання скінченного автомату 27](#_Toc5144862)  [3.4 Побудова скінченного автомату на основі ліволінійної граматики 29](#_Toc5144863)  [**4 РОЗРОБКА ПРОГРАМНОГО ПРОДУКТУ** 31](#_Toc5144864)  [4.1 Технічне завдання на розробку програми 31](#_Toc5144865)  [4.2 Вибір і обгрутнування критеріїв ефективності та якості ПЗ 31](#_Toc5144866)  [4.3 Проектування інтерфейсу користувача 35](#_Toc5144867)  [4.4 Алгоритм роботи програмного продукту 37](#_Toc5144868)  [4.5 Інструкція з експлуатації програмного продукту 39](#_Toc5144869)  [**ВИСНОВКИ** 44](#_Toc5144870)  [**ПЕРЕЛІК** **ПОСИЛАНЬ** 45](#_Toc5144871)  **ДОДАТОК А** Лістинг програми | | | | | | | | | | |
|  |  |  |  |  | **ККЗ.ДР.301.020.19.ПЗ** | | | | | |
|  |  |  |  |  |
| Зм | Арк | № докум. | Підпис | Дата |
| Розробив | | Павлов П.О. |  |  | **ПРОГРАМА НА МОВІС# «ПЕРЕТВОРЕННЯ РЕГУЛЯРНОЇ ГРАМАТИКИ В СКІНЧЕННИЙ АВТОМАТ»** | Літ | | | Аркуш | Аркушів |
| Керівник | | Довженко П.В. С.А. |  |  |  |  |  | 1 | 45 |
| Рецензент | |  |  |  | **РПЗ-54** | | | | |
| Н. контр. | | Жидка О. В. |  |  |
| Затвердив | | Румянцева А.М. |  |  |

# ВСТУП

Дослідження в галузі теорії автоматів почалися ще в середині 50-х років минулого століття. Не дивлячись на свою простоту, модель скінченного автомату виявилася надзвичайно зручною у застосуванні не тільки в інформатиці, але і в багатьох інших галузях інженерної діяльності. Великий інтерес до цієї теорії пояснюється саме широкими можливостями її застосування. Без перебільшення можна сказати, що теорія автоматів є одним з фундаментальних блоків сучасної теоретичної і практичної інформатики.

Як відомо, невід’ємною частиною створення скінченного автомату є побудова його регулярної граматики. Процес перетворення регулярної граматики до скінченного автомату вимагає багато часу та ресурсів для якісного та безпомилкового виконання поставленого завдання. Тому дуже гостро стоїть питання економії зусиль витрачених на процес перетворення. Це питання вирішується ефективно за допомогою використання програмних продуктів для перетворення регулярної граматики до скінченного автомату. Такі програмні продукти полегшують роботу з скінченними автоматами, і значно прискорюють сам процес перетворення за рахунок того, що одна й та ж послідовність дій виконується не вручну людиною, а автоматично за допомогою машини. Тому завдання створення такого програмного продукту є досить актуальним та необхідним.

Скінченні автомати широко використовується в автоматиці для написання алгоритмів функціонування широкого класу пристроїв і систем дискретної техніки, розробці програм для аналізу текстів, проведення тестування програмних продуктів.

Такий підхід до тестування та розробки є новим і перспективним напрямком.

Кінцевою метою даної роботи є розробка програмного забезпечення перетворення регулярної граматики до скінченного автомату. Відповідно до цього на шляху досягнення мети будуть вирішені такі задачі:

* аналіз існуючих програмних продуктів для перетворення регулярної граматики до скінченного автомату;
* дослідження алгоритмів перетворення;
* створення програмного продукту для перетворення регулярної граматики що задана користувачем, до відповідного їй скінченного автомату.

Для вирішення вище поставлених задач використовуються алгоритми для перетворення регулярної граматики спочатку до автоматного виду, після чого до скінченного автомату.

Створюване програмне забезпечення не маючи аналогів, може набути широкого розповсюдження серед фахівців які застосовують скінченні автомати, студентами під час виконання практичних та лабораторних робіт при проектуванні скінченних автоматів.

# 1 ФОРМАЛЬНІ МОВИ ТА ГРАМАТИКИ

## 1.1 Мови та ланцюжки символів. Способи задання мов

Ланцюжком символів або рядком називають довільну упорядковану скінченну послідовність символів, що записані один за одним. Поняття символ або буква є базовим поняттям в теорії формальних мов. Ланцюжок символів – це послідовність, в яку можуть входити будь-які доступні символи, та не обов’язково щоб вона несла певний сенс, так послідовність «арпр» також є прикладом ланцюжка символів. Для ланцюжка символів має значення три основних фактори:

* склад символів, що входять до ланцюжка;
* кількість символів, що входять до ланцюжка;
* порядок запису символів ланцюжка.

Ланцюжки символів рівні якщо вони мають один і той же склад символів, однакову кількість, а також однаковий порядок запису. Кількість символів називають довжиною ланцюжка та позначають знаком модуля |a|.

Основними операціями над ланцюжками є:

* конкатенація – це з'єднання двох ланцюжків один з одним, шляхом дописування другого ланцюжка в кінець першого;
* заміна або підстановка – це операція при якій одну частину першого ланцюжка можна замінити на будь-який ланцюжок символів;
* обернення – це операція при якій символи ланцюжка записуються в зворотному порядку;
* ітерація –це операція повторення ланцюжка символів певну кількість разів.

Порожній ланцюжок – ланцюжок символів, який не містить жодного символу, та довжина якого рівна нулю, позначається грецькою буквою «лямбда».

Для порожнього ланцюжка справедливі наступні правила:

* довжина порожнього ланцюжка дорівнює нулю;
* для всіх ланцюжків, конкатенація порожнього ланцюжка, з будь-яким ланцюжком дорівнює цьому ланцюжку;
* обернення порожнього ланцюжка дорівнює порожньому ланцюжку:
* для всіх ланцюжків ітерація порожнього ланцюжка, при умові, що кількість ітерацій більше нуля, дорівнює порожньому ланцюжку;
* для всіх ланцюжків, ітерація ланцюжка, при умові, що кількість ітерацій дорівнює нулю, дорівнює порожньому ланцюжку.

Мова – це заданий набір символів і правил, що встановлюють способи комбінації цих символів між собою для запису текстів. Основою будь-якої мови є алфавіт, що показує набір символів, що доступні мові.

Алфавіт – це скінченна множина символів, що доступні мові. Позначається алфавіт латинською буквою V. Існує два типи алфавітів, множина всіх ланцюжків над алфавітом без порожнього ланцюжка позначається зі степенем «+», множина всіх ланцюжків над алфавітом, що включає порожній ланцюжок позначається зі степенем «\*».

Мовою L над алфавітом V: L(V) називають деяку скінченну підмножину ланцюжків скінченної довжини з множини всіх ланцюжків над алфавітом. Множина ланцюжків мови не обов’язково має буди скінченною, її довжина може бути будь-якої довжини та формально нічим не обмежена.

Дві мови є однаковими або еквівалентними якщо множини доступних символів рівні. Дві мови є майже еквівалентними якщо множини доступних символів розрізняються лише на наявність порожнього ланцюжка.

Кожна мова – це множина ланцюжків символів над деяким алфавітом. Але окрім алфавіту мова також має правила побудови доступних ланцюжків, тому що зазвичай не всі ланцюжки задані алфавітом належать мові. Символи можуть об’єднуватися в слова та лексеми – елементарні конструкції мови, на їх основі будуються речення – більш складні конструкції. Всі вони в загальному вигляді є ланцюжками символів, але передбачають деякі правила побудови.

В загальному випадку мову можна описати трьома способами:

* перерахування всіх доступних ланцюжків мови;
* задання способу утворення ланцюжків символів мови (задання граматик мови);
* визначення методу розпізнавання ланцюжків мови.

Перший спосіб є формальним і на практиці не застосовується, так як більшість мов має нескінченну кількість допустимих ланцюжків і перерахувати їх неможливо.

Другий спосіб передбачає деякий опис правил, за допомого яких будуються ланцюжки. Тоді будь-який ланцюжок, побудований за допомогою цих правил з символів алфавіту мови, буде належати заданій мові.

Третій спосіб передбачає побудову деякого логічного пристрою (розпізнавача) – автомату, який на вході отримує ланцюжок символів, а на виході виводить відповідь: належить чи ні цей ланцюжок заданій мові.

Синтаксис мови – це набір правил, що визначають доступні конструкції мови. Синтаксис визначає форму мови – задає набір ланцюжків символів, які належать мові. Частіше за все синтаксис мови можна задати у вигляді суворого набору правил, але повністю це твердження справедливо лиже для чисто формальних мов. Навіть для більшості мов програмування набір заданих синтаксичних конструкцій потребує додаткових пояснень.

Семантика мови – це розділ мови, що визначає значення речень мови. Семантика визначає вміст мови – задає сенс для всіх доступних ланцюжків мови. Семантика для більшості мов визначає неформальні методи (відношення між знаками і тим, що вони позначають). Чисто формальні мови не мають будь-якого сенсу.

Лексика – це сукупність слів мови. Слово або лексична одиниця (лексема) мови – це конструкція, яка складається з елементів алфавіту і не вміщує в собі інші конструкції. Лексична одиниця може вміщувати лише елементарні символи і не може вміщувати інші лексичні одиниці.

Лексична одиниця мови це слова мови, а знаки і пробіли є розділювачами, що не створюють лексем. В мовах програмування лексичними одиницями є ключові слова, ідентифікатори, константи, мітки, знаки операцій.

Мови програмування займають деяке проміжне положення між формальними і природними мовами. З формальними мовами їх об’єднують жорсткі синтаксичні правила, на основі яких будуються речення мови. Від мов природного спілкування в мови програмування перейшли лексичні одиниці, що представляють основні ключові слова (частіше за все це слова англійської мови, але існують мови програмування, що мають ключові слова на російській мові).

Для задання мов мови програмування потрібно вирішити три питання:

* визначити множину доступних символів мови;
* визначити множину правильних програм мови;
* задати сенс для кожної правильної програми.

При визначені алфавіту мови, автоматично визначається множина допустимих символів. Для мов програмування алфавіт – це найчастіше набір символів, які можна ввести з клавіатури. Основну частину його складає половина таблиці міжнародного кодування символів, до якої додаються символи національних алфавітів.

Для всіх мов програмування існують правила, що визначають синтаксис мови, проте недостатньо для того щоб визначити всі допустимі речення мови програмування. Додаткові обмеження накладаються семантикою мови. Ці обмеження обговорюються в неформальному вигляді для кожної мови програмування. До таких обмежень можна віднести необхідність попереднього опису змінних і функцій, необхідність відповідності типів змінних і констант в виразах, формальних і фактичних параметрів в викликах функцій. Звідси слідує, що практично всі мови програмування, не є формальними мовами. Тому у всіх трансляторах крім синтаксичного розбору і аналізу речень мови додатково передбачено синтаксичний аналіз.

Виведенням називається процес породження речень мови на основі правил граматики, що визначає мову. Згідно визначенню, при виведені α→β виконується підстановка під ланцюжка замість іншого під ланцюжка. Ланцюжок β виводиться з ланцюжка α, у тому випадку якщо можна взяти декілька символів в ланцюжку α, замінити їх на інші символи згідно деякому правилу граматики і отримати ланцюжок β. Ланцюжок називається виведенням з іншого ланцюжка в тому випадку якщо виконується одне з двох умов: він безпосередньо виводиться з іншого ланцюжка, або існує інший ланцюжок, що виводиться з нього, а інший ланцюжок виводиться з нового. Це рекурсивне визначення виводимості ланцюжків. Сенс його у тому, що ланцюжок виводиться з іншого ланцюжка, якщо α→β, або ж можна побудувати послідовність безпосереднього виведення ланцюжків. Така послідовність називається виведенням, або ланцюжком виведенням. Кожен перехід від однієї безпосередньо виведеного ланцюжка до наступного в ланцюжку виведення називається кроком виведення. Кроків виведення в ланцюжку виведення на один більше, ніж проміжних ланцюжків.

Виведення називається завершеним, якщо на основі ланцюжка β, отриманий в результаті вивід, неможливо більше зробити жодного кроку. Іншими словами, вивід називається завершеним, якщо ланцюжок, отриманий в результаті виведення, порожній або вміщує лише термінальні символи граматики. Такий ланцюжок називається завершеним ланцюжком виведення.

Ланцюжок символів називається сентенціальною формою якщо він виводиться з цільового символу граматики. Якщо ланцюжок отриманий в результаті завершеного виведення, то він називається завершеною сентенціальною формою.

Виведення називається лівостороннім, якщо в ньому на кожному кроці виведення правило граматики застосовується завжди до крайнього лівого символу в ланцюжку. Іншими словами, виведення називається лівостороннім, якщо на кожному кроці виведення відбувається підстановка ланцюжка символів на основі правила граматики замість крайнього лівого нетермінального символу в початковому ланцюжку. Аналогічно виведення називається правостороннім, якщо в ньому на кожному кроці виведення правило граматики застосовується завжди до крайнього правого нетермінального символу в ланцюжку.

Деревом виведення граматики називається дерево (граф), яке відповідає деякому ланцюжку виведення і задовольняє наступні вимоги:

* кожна вершина дерева позначається символом граматики;
* коренем дерева є вершина, що позначена цільовим символом граматики;
* листками дерева (кінцевими вершинами) є вершини, позначені термінальними символами граматики або символом порожнього ланцюжка;
* якщо деякий вузол дерева позначений нетермінальним символом А, а залежні до нього вузли – символами b, то існує правило А→b.

За структурою правил дерево виведення у вказаному вигляді завжди можна побудувати тільки для граматик типів 2 і 3. Для граматик інших типів дерево виведення в такому вигляді можна побудувати не завжди. Приклад дерева виведення зображено на рис. 1.1.

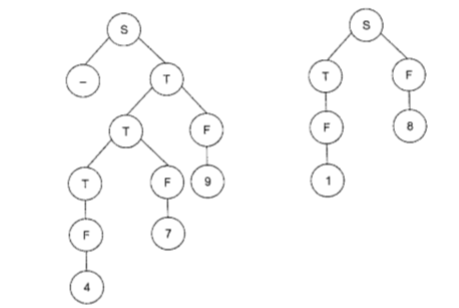


Рисунок 1.1 – Приклад дерева виведення

Для того, щоб побудувати дерево виведення, достатньо мати тільки ланцюжок виведення. Дерево виведення можна побудувати двома способами: зверху до низу та знизу до верху. Для суворо формалізованої побудови дерева виведення завжди зручніше користуватися суворо визначеним виведенням: або лівостороннім, або правостороннім. Для побудови дерева виведення зверху до низу побудова починається з цільового символу, який розміщується в корінь дерева. Потім в граматиці вибирається необхідне правило, і на першому кроці виведення кореневий символ відкривається на декілька символів першого рівня. На другому кроці серед усіх кінцевих вершин обирається крайня (крайня ліва – для лівостороннього виведення, крайня права – для правостороннього) вершина, позначена нетермінальним символом, для цієї вершини обирається потрібне правило граматики, і вона відчиняється на декілька вершин наступного рівня. Побудова дерева завершується, коли всі кінцеві вершини позначені термінальним символом, в іншому випадку потрібно повернутися до другого кроку і продовжити побудову.

Побудова дерева виведення знизу до верху починається з листків дерева. В якості листків обираються термінальні символи скінченного ланцюжка виведення, які на першому кроці побудови утворюють останній рівень дерева. Побудова дерева йде шарами. На другому кроці побудови в граматиці обирається правило, права частина якого відповідає крайнім символам в шарі дерева (крайнім правим символам при правосторонньому виведені і крайнім лівим – при лівосторонньому). Обрані вершини шару з’єднуються з новою вершиною, яка обирається з лівої частини правил. Нова вершина потрапляє в шар дерева замість обраних вершин. Побудова дерева завершена, якщо досягнута кінцева точка (позначена цільовим символом), а інакше потрібно повернутися до другого кроку і повторити його відносно отриманого шару дерева.

Оскільки всі відомі мови програмування мають нотацію запису зліва направо, компілятор також завжди читає вхідну програму зліва направо. Тому для побудови дерева виведення методом зверху до низу, як правило, використовується лівостороннє виведення, а для побудування знизу до верху – правостороннє виведення. На цю здатність компіляторів потрібно звернути увагу. Нотація читання програм зліва направо впливає не тільки на порядок розбору програми компілятором (для користувача це, як правило, не має значення) але й на порядок виконання операцій – при відсутності дужок більшість рівних операцій виконуються в порядку зліва направо, а це вже має значення [1].

## 1.2 Граматики і розпізнавачі

Граматика – це опис способу побудови речень деякої мови. Визначивши граматику показуються правила породження ланцюжків символів, що належать до даної мови.

Правило – це упорядкована пара ланцюжків символів. В правилах важливий порядок ланцюжка, тому їх частіше записують у вигляді a→b.

Граматика мови програмування вміщує правила двох типів: перші (визначають синтаксичні конструкції мови) легко піддаються формальному опису; другі (визначають семантичні обмеження мови) зазвичай показуються в неформальній формі. Тому будь-який опис мови програмування зазвичай складається з двох частин: спочатку формально викладаються правила побудови синтаксичних конструкцій, а потім на її звичній мові надається опис семантичних правил.

Дві граматики називаються еквівалентними, якщо вони визначають одну і ту ж мову. Дві граматики називаються майже еквівалентними, якщо задані ними мови розрізняються лише наявністю в них порожнього ланцюжка.

Формально граматика G визначається як четвірка G(VT, VN, P, S), де:

* VT – множина термінальних символів або алфавіт термінальних символів;
* VN – множина нетермінальних символів або алфавіт нетермінальних символів;
* P – множина правил граматики виду a→b, де a належить множині термінальних та нетермінальних символів;
* S – цільовий або початковий символ граматики.

Алфавіти термінальних та нетермінальних символів граматики не перетинаються. Це означає, що кожний символ в граматиці може бути або термінальним, або нетермінальним, проте не може бути термінальним і нетермінальним одночасно. Цільовий символ граматики завжди є нетермінальним символом. Множину термінальних символів з нетермінальними називають повним алфавітом граматики.

У множині правил граматики може бути декілька правил, що мають однакові ліві частини. Тоді ці правила об’єднуються разом і записуються через символ «|». Таку форму запису правил називають формою Бекуса-Наура. Вона передбачає, як правило, що нетермінальні символи беруться в кутові дужки «< >». Іноді знак «стрілка» в правилах граматики заміняють знаком «::=», але це всього незначна модифікація форми запису, яка не змінює її суть.

Форма Бекуса-Наура – зручний з формальної точки зору, але не завжди доступний для розуміння спосіб запису формальних граматик. Рекурсивні визначення правильні для формального аналізу ланцюжків мови, але не зручні з точки зору людини. Але при створені мови програмування важливо, щоб його граматику розуміли не тільки ті, кому потрібно буде створювати компілятори для цієї мови, але й користувачі мови, майбутні розробники програм. Тому існують інші способи описання правил формальних граматик, які орієнтовані на більшу зрозумілість для людини. Існують два найбільш розповсюджених способи: запис граматик з використанням метасимволів та запис правил граматик в графічному вигляді.

Запис граматик з використанням метасимволів передбачає, що в рядку правила граматики можуть зустрічатися спеціальні символи – метасимволи, – які мають особливий сенс і трактуються спеціальним чином. В якості таких метасимволів найчастіше використовуються наступні символи: круглі дужки, фігурні дужки, квадратні дужки, лапки та кома. Ці метасимволи мають наступний сенс:

* круглі дужки означають, що з усіх перерахованих всередині них ланцюжків символів в даному місці правила граматики може знаходитись один ланцюжок;
* квадратні дужки означають, що вказаний в них ланцюжок може зустрічатися, а може й не зустрічатися в даному місці правил граматики (тобто може бути в ньому один або жодного разу);
* фігурні дужки означають, що вказаний в них ланцюжок символів може не зустрічатися в даному місці граматики жодного разу, зустрічатися один або скільки завгодно разів;
* кома слугує для того, щоб розділяти ланцюжки символів всередині круглих дужок;
* лапки використовуються у тому випадку, коли один з метасимволів потрібно включити в ланцюжок звичайним чином – тобто коли одна з дужок або кома повинні знаходитись в ланцюжку символів (якщо лапки потрібно включити до ланцюжка символів, то її потрібно повторити двічі).

Форма запису правил з використанням метасимволів – це зручний і зрозумілий спосіб представлення правил граматики. Вона в багатьох випадках дозволяє повністю позбутися від рекурсії, замінивши її символом ітерації «фігурні дужки». Крім зазначених метасимволів в цілях зручності запису в описі граматик іноді використовують і інші метасимволи, при цьому попередньо дається пояснення їх сенсу. Принцип запису від цього не змінюється.

При запису правил в графічному вигляді вся граматика представляється в формі набору спеціальним чином побудованих діаграм. Ця форма була запропонована при описанні граматики мови Pascal, а потім вона отримала широкі розповсюдження в літературі. Вона доступна не для всіх типів граматик, а тільки для тих типів, де в лівій частині присутні не більше одного символа, проте цього достатньо, щоб її можна було використовувати для описання граматик відомих мов програмування.

В такій формі запису кожному нетермінальному символу граматики співпадає діаграма, побудована в вигляді направленого графу. Граф має наступні типи вершин:

* точка входу (на діаграмі ніяк не позначено, з неї просто розпочинається вхідна дуга графу);
* нетермінальний символ (на діаграмі позначається прямокутником, в який вписано позначення символу);
* ланцюжок термінальних символів (на діаграмі позначається овалом, кругом або прямокутником з закругленими краями, всередині якого записаний ланцюжок);
* вузлова точка (на діаграмі позначається жирною точкою або зафарбованим кругом);
* точка виходу (ніяк не позначена, в неї просто входить вихідна дуга графу).

Кожна діаграма має лише одну точку входу і одну точку виходу, по скільки завгодно вершин інших трьох типів. Вершини з’єднуються між собою направленими дугами графу (лініями зі стрілками). З вхідної точки дуги можуть тільки виходити, а у вхідну точку тільки входити. В інші вершини дуги можуть як входити, так і виходити.

Щоб побудувати ланцюжок символів, на основі нетермінального символу граматики, потрібно продивитися діаграму цього символу. Тоді, почавши рух від точки, потрібно рухатися по дугам графу діаграму через будь-які вирини впритул до точки виходу. При цьому, проходячи через вершину, позначеному нетермінальним символом, цей символ потрібно помістити в результуючий ланцюжок. При проходженні через вершину, позначену ланцюжком термінальних символів, ці символи також потрібно помістити в результуючий ланцюжок. При проходженні через вузлові точки діаграми над результуючим ланцюжком ніяких дій виконувати не потрібно. Через будь-яку вершину графу діаграми, в залежності від можливого шляху руху, можна пройти один раз, ні разу або скільки завгодно разів. Як тільки ми потрапимо в точку виходу діаграм, побудування результуючого ланцюжка буде завершено.

Результуючий ланцюжок, в свою чергу, може містити нетермінальні символи. Щоб замінити їх на ланцюжки термінальних символів, потрібно, знову ж, продивлюватись їх діаграми. І так до тих пір, поки в ланцюжку не залишаться лише термінальні символи. Очевидно, що для того, щоб побудувати ланцюжок символів заданої мови, потрібно почати перегляд діаграм цільового символу граматики.

Особливістю формальних граматик є те, що вони дозволяють нескінченну множину ланцюжків мови за допомогою набору правил (звичайно, множина ланцюжків мови також може бути скінченною але навіть для простих реальних мов ця умова зазвичай не виконується). Для зменшення кількості правил можна вводити таке поняття як рекурсія, вона виражається в тому, що один з нетермінальних символів визначається сам через себе. Рекурсія буває явною – коли символ визначається сам через себе в одному правилі, або неявною – коли теж саме але через ланцюжок правил. Щоб рекурсія не була нескінченною, для символу, що викликає рекурсію повинні існувати також і інші правила, які визначать його, не включаючи його самого, і не дозволять виникненню нескінченної рекурсії.

Принцип рекурсії – важливе поняття в уявлені про формальні граматики. Рекурсія, явна або неявна, завжди присутня в граматиках будь-яких реальних мов програмування. Саме вона дозволяє будувати нескінченні множини ланцюжків мови, і говорити про їх породження неможливо без поняття принципу рекурсії. Як правило, в граматиці реальної мови програмування знаходиться не одна, а ціла множина правил, побудованих за допомогою рекурсії.

Розпізнавач – це спеціальний автомат, який дозволяє визначити приналежність ланцюжка символів деякій мові. Завдання розпізнавача полягає у тому, щоб на основі вхідного ланцюжка дати відповідь на питання, належить чи ні він заданій мові.

Розпізнавачі складаються з наступних основних компонентів:

* стрічки, вміщають вхідний ланцюжок символів, і зчитувальної голівки, що знаходить наступний символ в цьому ланцюжку;
* пристрій керування, який координує роботу розпізнавача, має деякий набір станів і скінченну пам’ять (для зберігання свого стану і деякою проміжною інформацією);
* зовнішня пам’ятть, яка може зберігати деяку інформацію в процесі роботи розпізнавача і, на відміну від пам’яті керуючого пристрою, має необмежений об’єм.

Розпізнавач працює з символами свого алфавіту – алфавіту розпізнавача. Алфавіт розпізнавача скінченний. Він включає в себе всі доступні символи вхідних ланцюжків, а також деякий додатковий алфавіт символів, які можуть оброблятися пристроєм керування і зберігатися в робочій пам’яті розпізнавача. В процесі своєї роботи розпізнавач може виконувати деякі елементарні операції:

* читання наступного символу з вхідного ланцюжка;
* переведення вхідного ланцюжка на задану кількість символів (направо або на ліво);
* доступ до робочої пам’яті для читання або запису інформації:
* перетворення інформації в пам’яті керуючого пристрою, зміна стану пристрою керування.

Розпізнавач працює по крокам, або тактами. На початку такту, як правило, зчитується наступний символ з вхідного ланцюжка, і в залежності від цього символу пристрій керування визначає, які дії потрібно виконати. Вся робота розпізнавача складається з послідовності тактів. На початку кожного такту стан розпізнавача визначається його конфігурацією. В процесі роботи конфігурації розпізнавача змінюються. Конфігурація розпізнавача визначається наступними параметрами:

* складом вхідного ланцюжка символів і положенням зчитувальної голівки;
* станом пристрою керування;
* складом зовнішньої пам’яті.

Для розпізнавача завжди задається конфігурація, яка вважається початковою конфігурацією. В початковій конфігурації зчитувальна голівка знаходить перший символ вхідного ланцюжка. В кінцевій конфігурації зчитувальна голівка, як правило, знаходиться вкінці вихідного ланцюжка. Мова, що визначається розпізнавачем – це множина всіх ланцюжків, які допускає розпізнавач.

За типом зчитувального пристрою розпізнавачі можуть бути двосторонніми або односторонніми. Односторонні розпізнавачі допускають переміщення зчитувальної голівки по стрічці вхідних символів тільки в одному напрямку. Це означає, що на кожному кроці роботи розпізнавача зчитувальна голівка може або переміститися по стрічці на деяку кількість позицій і заданому напрямку, або залишитися на своєму місці. Оскільки всі мови програмування передбачають нотацію читання вихідної програми зліва направо, то так і працюють всі інші розпізнавачі. Тому коли говорять про односторонні розпізнавачі, то спочатку мають на увазі лівосторонні, які читають вхідний ланцюжок зліва направо і не повертаються назад до вже прочитаної частини ланцюжка. Двосторонні розпізнавачі допускають, що зчитувальна голівка може переміщатися відносно стрічки вхідних символів в обох напрямках, як вперед, від початку стрічки до кінця, так і назад, повертаючись до вже прочитаних символів.

За типом пристрою керування розпізнавача бувають детерміновані і недетерміновані. Розпізнавач називається детермінованим в тому випадку, якщо для кожної допустимої конфігурації розпізнавача, яка виникла на деякому кроці його роботи, існує єдина можлива конфігурація, в яку розпізнавач перейде на наступний крок роботи. В іншому випадку розпізнавач називається недетермінованим. Недетермінований розпізнавач може мати таку доступну конфігурацію, для якої існує деяка скінченна множина конфігурацій, можливих на наступному кроці роботи. Достатньо мати хоча б одну таку конфігурацію, щоб розпізнавач був недетермінованим.

За типом зовнішньої пам’яті розпізновачі бувають наступних видів:

* розпізнавачі без зовнішньої пам’яті;
* розпізнавачі з обмеженою зовнішньою пам’яттю;
* розпізнавачі з необмеженою зовнішньою пам’яттю;

У розпізнавачів з обмеженою зовнішньою пам’яттю розмір пам’яті обмежений в залежності від довжини вхідного ланцюжка. Крім цього, для таких розпізнавачів може бути вказаний спосіб організації зовнішньої пам’яті – стек, черга, список та інші. Розпізнавачі з необмеженою зовнішньою пам’яттю передбачають, що для їх роботи потрібно буде зовнішня пам’ять необмеженого обсягу. У розпізнавачів без зовнішньої пам’яті повністю відсутня зовнішня пам’ять. В процесі їх роботи використовується тільки пристрій керування.

Тип розпізнавачів в класифікації визначає складність створення такого розпізнавача, відповідно, складність розробки відповідного програмного забезпечення для компілятора. Чим вище в класифікації стоїть розпізнавач, тим складніше складати алгоритм, що забезпечує його роботу. Розробляти двосторонні розпізнавачі складніше, чим односторонні. Недетерміновані розпізнавачі по складності вище детермінованих.

Класифікація розпізнавачів визначає складність алгоритму роботи розпізнавача. Але складність розпізнавача також напряму зв’язана з типом мови, вхідні ланцюжки якого може приймати розпізнавач. Доведено, що для кожного з чотирьох типів мов існує свій тип розпізнавача з певним складом компонентів і з заданою складністю алгоритму роботи.

Для мов з фразовою структурою потрібен розпізнавач, рівний потужністю з машиною Тюрінга – недетермінований двосторонній автомат, що має необмежену внутрішню пам’ять. Тому для мов даного типу неможна гарантувати, що за обмежений час на обмежених обчислювальних ресурсах розпізнавач завершить роботу і отримає рішення про те, належить чи ні вхідний ланцюжок заданій мові. Звідси можна заключити, що практичного застосування мов з фразовою структурою немає.

Для контекстно-залежних мов розпізнавачами є двосторонні недетерміновані автомати з лінійно обмеженою зовнішньою пам’яттю. Алгоритм роботи такого автомату в загальному випадку має експоненціальну складність – кількість кроків, необхідних автомату для розпізнання вхідного ланцюжка, екпоненціально залежить від довжини цього ланцюжка. Час необхідний на розбір вхідного ланцюжка по заданому алгоритму, експоненціально залежить від довжини вхідного ланцюжка символів.

Для контекстно-вільних мов розпізнавачами є односторонні недетерміновані автомати з магазинною зовнішньою пам’яттю. При найпростішій реалізації алгоритму роботи такого автомату він має експоненціальну складність, проте шляхом деяких вдосконалень алгоритму можна досягнути поліноміальної залежності часу, необхідного для розбору вхідного ланцюжка, від довжини цього ланцюжка.

Для регулярних мов розпізнавачами є односторонні недетерміновані автомати без зовнішньої пам’яті – скінченні автомати. Це досить простий тип розпізнавача, який передбачає лінійну залежність часу розбору вхідного ланцюжка від її довжини. Крім того, скінченні автомати мають важливу особливість: будь-який недетермінований скінченний автомат завжди може бути перетворений в детермінований. Ця обставина спрощує розробку програмного забезпечення, що забезпечують функціональні розпізнавачі.

Акцептори і розпізнавачі продукують двійковий вихід, кажучи або так або ні на питання прийняті автоматом вхідні дані чи ні. Всі стани СА можуть бути або допустимі або ні. Коли всі вхідні дані оброблені, якщо поточний стан є допустимим, значить вхід прийнятий; інакше відхилений. Як правило на вхід подаються символи (літери); дії не використовуються.

Автомат також може бути описаний як такий, що визначає мову, яка містить всі слова розпізнавані цим автоматом, але не ті які ним відхиляються; тоді ми кажемо, що ця мова розпізнається автоматом. За визначенням, мови розпізнавані СА це регулярні мови тобто мова є регулярною якщо існує деякий СА, який розпізнає її.

Для кожної мови програмування потрібно не тільки вміти побудувати текст програми на цій мові, але й визначити приналежність тексту до даної мови. Тому це завдання вирішують компілятори паралельно з іншими задачами (компілятор повинен не тільки розпізнавати вхідну програму, але й побудувати еквівалентну їй результуючу програму). У відношенні початкової програми компілятор виступає в ролі генератора ланцюжків цієї мови. Граматики і розпізнавачі – два незалежних метода, які реально можуть бути використані для визначення будь-якої мови. Проте при створені компілятора для деякої мови програмування виникає завдання, яке потребує зв’язку між собою цих двох методів задання мови.

Розробники компілятора завжди мають справу з вже відомою мовою програмування. Граматика для синтаксичних конструкцій цієї мови відома. Вона, як правило закладається у тому, щоб побудувати розпізнавач для заданої мови, який потім буде основою синтаксичного аналізатора в компіляторі. Таким чином, завдання розбору в загальному вигляді закладається в наступному: на основі заданої граматики деякої мови побудувати розпізнавач для цієї мови. Задана граматика і розпізнавач повинні бути еквівалентними, тобто визначають один одного. Завдання розбору в загальному вигляді може бути вирішене не для всіх мов. Розробників компіляторів цікавлять більше за все синтаксичні конструкції мов програмування. Для цих конструкцій доведено, що завдання розбору для них вирішується.

Оскільки мови програмування не є чисто формальними мовами і несуть в собі деякий сенс (семантику), то завдання розбору для створення реальних компіляторів застосовується дещо ширше, чим для чисто формальних мов. Компілятор повинен не тільки встановлювати належність вхідного ланцюжка символів заданій мові, але й визначити її сенсове навантаження. Для цього потрібно знайти ті правила граматики, на основі яких ланцюжок буде побудований. Якщо ж вхідний ланцюжок не належить заданій мові – початкова програма налічує помилку, – розробнику програми не цікаво просто пізнати сам факт належності помилки. В цьому випадку задача розбору не тільки встановити факт присутності помилки у вхідному ланцюжку, але й визначити тип помилки й те місце у вхідному ланцюжку символів, де вона зустрічається [8].

## 1.3 Класифікація мов та граматик

Згідно класифікації, запропонованій американським вченим лінгвістом Ноамом Хомським, професором Масачусиньського технологічного університету, формальні граматики класифікуються по структурі їх правил. Якщо всі без виключення правила граматики задовольняють деяку задану структуру, то таку граматик відносять до певного типу. Достатньо мати в граматиці одне правило, що не задовольняє вимоги структури правил, і вона вже не потрапляє в заданий тип. По класифікації Хомського розрізняють чотири типи граматик.

Тип 0: граматики з фразовою структурою. На структуру їх правил не накладаються жодні обмеження: для граматики вигляду G(VT, VN, P, S), V = VN Ս VT правила мають вигляд: α→β. Це найзагальніший спільний тип граматик. До нього входять всі правила без винятку, але частина з них, меже бути також віднесена до інших класифікацій типів. Справа в тому, що граматики, які відносяться лише до типу 0, і не можуть відноситись до інших типів, є найскладнішими по структурі. Практичного застосування граматики, що відноситься до типу 0 немає.

Тип 1: контекстно-залежні (КЗ) і нескорочувальні граматики. До цього типу входять два основні типи класу граматик:

* контекстно-залежні граматики G(VT, VN, P, S), V = VN Ս VT правила мають вигляд: ;
* нескорочувальні граматики G(VT, VN, P, S), V = VN Ս VT правила мають вигляд:

Структура правил КЗ-граматик така, що при побудові речень заданої ними мови один і той же нетермінальний символ може бути замінений на інший ланцюжок символів в залежності від контексту, в якому він зустрічається. Нескорочувальні граматики мають таку структуру правил, що й при побудові речень мови, що була задана граматикою, будь-який ланцюжок символів може бути замінена на ланцюжок символів не меншої довжини. При побудові компіляторів такі граматики не застосовуються, оскільки синтаксичні конструкції мов програмування, продивляються компіляторами, мають більш просту структуру і можуть бути побудовані за допомогою граматик інших типів.

Тип 2: контекстно-вільні граматики. Контекстно-вільні граматики мають правила виду: А→β. Такі граматики також іноді називаються нескорочувальними контекстно-вільними граматиками (в правій частині правила у них повинен завжди буде стояти як мінімум один символ). Існує також майже еквівалентний клас граматик – скорочувальні контекстно-вільні граматики, правила яких можуть мати вигляд: А→β. Різниця між цими двома класами граматик закладається, лише у тому, щоб в скорочувальних контекстно-вільних граматиках в правій частині може бути присутній порожній ланцюжок, а в нескорочувальних – ні. Звідси зрозуміло, що мова, задана нескорочувальною контекстно-вільною граматикою, не може містити порожнього ланцюжка. Контекстно-вільні граматики широко використовуються при описі синтаксичних конструкцій мов програмування. Синтаксис більшості відомих мов програмування в основі має саме контекстно-вільну граматику. Всередині типу контекстно-вільних граматик окрім класів нескорочувальних та скорочувальних граматик виділяють ще цілу множину різних класів граматик, і всі вони відносяться до типу 2.

Тип 3: регулярні граматики. До типу регулярних відносяться два еквівалентних класів граматик: ліволінійні і праволінійні. Ліволінійні граматики можуть мати правила двох видів А→Вα або А→α. В свою чергу, праволінійні граматики можуть мати правила також двох видів: А→αВ або А→α. Ці два типи граматик еквівалентні і відносяться до типу регулярних граматик. Регулярні граматики використовуються при описі простіших конструкцій мов програмування: ідентифікаторів, констант, рядків, коментарів. Ці граматики виключно прості і зручні в використанні, тому в компіляторах на їх основі будуються функції лексичного аналізу вхідної мови.

Типи граматик відносяться між собою особливим чином. З визначення типів 2 і 3 видно, що будь-яка регулярна граматика є контекстно-вільною граматикою, але не навпаки. Одна й та ж граматика може бути віднесена до типу 0, оскільки він не накладає жодних обмежень на правила. В той же час існують скорочувальні контекстно-вільні граматики , які не є ні контекстно-залежними, ні нескорочувальними, оскільки можуть містити правила з порожнім ланцюжком.

Мови класифікуються в співвідношені з типами граматик, за допомогою яких вони задані. Причому оскільки одна і та ж мова в загальному випадку може бути задана скільки завгодно великою кількістю граматик, які можуть відноситися до різних класифікаційних типів, то для класифікації мови серед всіх його граматик вибирається граматика з максимально можливим класифікаційним типом. Від класифікаційного типу мови залежить не тільки те, за допомогою якої граматики можна побудувати речення цієї мови, але також і те, наскільки важко розпізнати ці речення. Розпізнати додаток – означає побудувати розпізнавач для мови.

Складність мови убуває з використанням зі збільшенням класифікаційного типу мови. Найскладнішими є мови типу 0, найпростішими – мови типу 3. Згідно класифікації граматик, також існує чотири типи мов.

Тип 0: мови з фразовою структурую. Це найскладніші мови, які можуть бути задані тільки граматикою, що відноситься до типу 0. Для розпізнання ланцюжків таких мов потрібні обчислювачі, рівно потужні машині Тюрінга. Тому можна сказати, що якщо мова відноситься до типу 0, то для нього неможливо побудувати компілятор, який гарантовано виконав би розбір речень мови за обмежений час на основі обмежених обчислювальних ресурсів. Нажаль всі мови спілкування між людьми, відносяться до цього типу мов. Одне й те ж слово в звичайній мові може не тільки мати різний сенс в залежності від контексту, але й мати різні ролі в реченні. Саме тому велика складність в автоматизації перекладу текстів, що написані на звичайній мові, а також відсутні компілятори, які б сприймали програми на основі таких мов.

Тип 1: контекстно-залежні мови. Тип 1 – другий по складності тип мов. В загальному випадку час на розпізнання речення мови, що відноситься до типу 1, залежить від довжини початкового ланцюжка символів. Мови та граматики, що відносяться до типу 1, застосовуються в аналізі і перекладі текстів на звичайних мовах. Розпізнавачі, побудовані на їх основі, дозволяють аналізувати тексти з урахуванням контекстної залежності в реченнях вхідної мови (але вони не враховують вміст тексту, тому в загальному випадку для точного перекладу зі звичайної мови потрібно втручання людини). На основі таких граматик може виконуватись автоматизований переклад з однієї звичайної мови на іншу, ним можуть користуватися сервісні функції перевірки орфографії і правопису в мовних процесорах. В компіляторах контекстно-залежні мови не використовуються, оскільки мови програмування мають більш просту структуру.

Тип 2: контекстно-вільні мови. Контекстно-вільні мови знаходяться в основі синтаксичних конструкцій більшості сучасних мов програмування, на їх основі функціонують деякі досить складні командні процесори, що допускають керуючі команди циклу і умови. В загальному випадку час на розпізнання речень мови, що відноситься до типу 1, полі номінально залежить від довжини вхідного ланцюжка символів (в залежності від класу мови це або кубічна, або квадратична залежність). Проте серед контекстно-вільних мов існує багато класів мов, для яких ця залежність лінійна. Практично всі мови програмування можна віднести до одного з таких класів.

Тип 3: регулярні мови. Регулярні мови – найпростіший тип мов. Тому вони є найвикористовуванішим типом мов в області обчислювальних систем. Час на розпізнавання речень регулярної мови лінійно залежить від довжини вхідного ланцюжка. Регулярні мови лежать в основі найпростіших конструкцій мов програмування (ідентифікаторів, констант та інших), крім того, на їх основі будуються деякі мнемокоди машинних команд (мови асемблера), а також командні процесори, символьні керуючі команди та інші схожі структури. Регулярні мови – дуже зручний засіб. Для роботи з ними можна використовувати регулярні множини і вирази, скінченні автомати [3].

# 2 ЛЕКСИЧНІ АНАЛІЗАТОРИ

2.1 Визначення та призначення лексичного аналізатора

Лексема (лексична одиниця мови) – структурна одиниця мови, яка складається з елементарних символів мови і не містить в своєму складі інших структурних одиниць мови.

Лексичний аналізатор – це частина компілятора, яка зчитує вхідну програму і виділяє в її тексті лексеми вихідної мови. На вхід лексичного аналізатора поступає текст вихідної програми, а вихідна інформація передається для подальшої обробки компілятором на етапі синтаксичного аналізу.

Основні причини включення в компілятор аналізатора:

* застосування лексичного аналізатора спрощує роботу з текстом вихідної програми на етапі синтаксичного розбору і скорочує об’єм інформації, що обробляється, так як лексичний аналізатор структурує вхідний текст програми, що надходить аналізатору, і відкидає всю незначну інформацію;
* для виділення в тексті і розбору лексем є можливість застосувати просту, ефективну і теоретично добре оброблювану техніку аналізу, в той час як на етапі синтаксичного аналізу конструкцій вихідної мови використовується достатньо складні алгоритми розбору;
* сканер відокремлює складний по конструкції синтаксичний аналізатор від роботи безпосередньо з текстом вихідної програми, структура якого може варіюватися в залежності від версії вихідної мови – при такій конструкції компілятора для переходу від однієї версії мови до іншої достатньо лише перебудувати відносно простий лексичний аналізатор.

В основному лексичні аналізатори виконують виключення з тексту вхідної програми коментарів і незначущих пробілів а також виділення лексем наступних типів: ідентифікаторів, рядкових символьних і числових констант, ключових слів вхідної мови. В найпростішому випадку фази лексичного і синтаксичного аналізу можуть виконуватися компілятором послідовно, але для багатьох мов програмування інформації на етапі лексичного аналізу може бути не достатньо для однозначного визначення типу і границь чергової лексеми.

В більшості компіляторів лексичні і синтаксичні аналізатори взаємозалежні частини. Можливо два принципово різних методи організації взаємозв’язку лексичного аналізу і синтаксичного розбору, а саме послідовний і паралельний. При послідовному варіанті лексичний аналізатор переглядає весь текст вхідної програми від початку до кінця і перетворює його в структурований набір даних. В таблиці лексем ключові слова мови ідентифікатори і константи як правило заміняються на спеціальні коди які визначаються при реалізації компілятора.

Для ідентифікаторів і констант встановлюється зв’язок між таблицею лексем і таблицею ідентифікаторів які заповнюються паралельно. В цьому варіанті лексичний аналізатор переглядає весь текст вхідної програми з початку і до кінця, таблиця лексем будується вся одразу, і до неї компілятор більше не повертається, і всю подальшу обробку виконують інші фази компіляції, такий варіант називають послідовним.

При паралельному варіанті лексичний розбір вхідного тексту виконується поетапно, так, що синтаксичний аналізатор виконавши розбір синтаксичної частини тексту звертається до сканера за наступною.

При цьому він може повідомити інформацію про те, яку лексеми варто очікувати. В процесі розбору може навіть відбуватися крок назад щоб виконати аналіз тексту на іншій основі, і тільки після того, як синтаксичний аналізатор успішно виконає розбір чергової конструкції мови, лексичний аналізатор поміщає знайдені лексеми в таблицю лексем і таблицю ідентифікаторів, і продовжує розбір далі.

Результатом роботи лексичного аналізатора є перелік всіх знайдених в тексті вихідної програми лексем з урахуванням характеристик кожної лексеми. Цей перелік лексем можна представити у вигляді таблиці, яка має назву таблиця лексем. Кожній лексемі в таблиці лексем відповідає деякий унікальний умовний код, який залежить від типу лексеми, і додаткова службова інформація. Крім того, інформація про деякі типи лексем, що були знайдені в вихідній програмі, повинні поміщатися в таблицю ідентифікаторів.

Таблиця лексем фактично вміщує увесь текст вихідної програми, що був оброблений лексичним аналізатором. До неї входять всі можливі типи лексем, крім того, будь-яка лексема може зустрічатися в ній декілька разів.

Таблиця ідентифікаторів вміщує тільки окремі типи лексем – ідентифікатори і константи. До неї потрапляють такі лексеми, як ключові слова вхідної мови, знаки операцій і роздільники. Крім того, кожна лексема, може зустрічатися в таблиці ідентифікаторів лише один раз. Також можна відмітити, що лексеми в таблиці лексем обов’язково розташовуються в тому ж порядку, як і в вихідній програмі, а в таблиці ідентифікаторів лексеми розташовуються в будь-якому порядку, щоб забезпечити зручність пошуку. У таблиці 2.1 зображено приклад таблиці лексем для базових конструкцій мови програмування Pascal [5].

Таблиця 2.1 – Приклад таблиці лексем мови програмування Pascal

|  |  |  |  |
| --- | --- | --- | --- |
| **Лексема** | **Код лексеми** | **Лексема** | **Код лексеми** |
| program | 1 | / | 19 |
| var | 2 | ( | 20 |
| begin | 3 | ) | 21 |
| end | 4 | < | 22 |
| float | 5 | > | 23 |
| dowhile | 6 | <= | 24 |
| enddo | 7 | >= | 25 |
| if | 8 | == | 26 |
| else | 9 | <> | 27 |
| endif | 10 | [ | 28 |
| input | 11 | ] | 29 |
| output | 12 | or | 30 |
| : | 13 | and | 31 |
| , | 14 | not | 32 |
| = | 15 | idn | 33 |
| **Лексема** | **Код лексеми** | **Лексема** | **Код лексеми** |
| + | 16 | con | 34 |
| - | 17 | ˩ | 35 |
| \* | \* |  |  |

## 2.2 Принцип побудови лексичного аналізатора

Лексичний аналізатор має справу з такими об'єктами, як різного роду константи і ідентифікатори (до останніх відносяться і ключові слова). Мова констант і ідентифікаторів в більшості випадків є регулярною, тобто може бути описана за допомогою регулярних граматик. Розпізнавачем для регулярних мов є скінченні автомати (СА). Існують правила, за допомогою яких для будь-якої регулярної граматики може бути побудований недетермінований кінцевий автомат, що розпізнає ланцюжка мови, заданого цієї граматикою. Кінцевий автомат для кожного вхідного ланцюжка мови дає відповідь на питання про те, чи належить чи ні ланцюжок мові, задану автоматом.

Однак в загальному випадку задача сканера дещо ширше, ніж просто перевірка ланцюжка символів лексеми на відповідність її вхідній мові. Крім цього, сканер повинен виконати наступні дії:

* чітко визначити межі лексеми, які в початковому тексті явно не задані;
* виконати дії для збереження інформації про виявлену лексему (або видати повідомлення про помилку, якщо лексема невірна).

Визначення меж лексем. Виділення кордонів лексем представляє певну проблему. Адже у вхідному тексті програми лексеми не обмежені ніякими спеціальними символами. Якщо говорити в термінах програми-сканера, то визначення меж лексем – це виділення тих рядків в загальному потоці вхідних символів, для яких треба виконувати розпізнавання. У загальному випадку ця задача може бути складною, і тоді потрібна паралельна робота сканера (лексичного аналізатора), синтаксичного розбору і, можливо, семантичного аналізу. Для більшості вхідних мов кордони лексем розпізнаються по заданим термінальним символам. Ці символи – прогалини, знаки операцій, символи коментарів, а також роздільники (коми, крапки з комою та інше). Набір таких термінальних символів може варіюватися в залежності від синтаксису вхідного мови.

Важливо відзначити, що знаки операцій також є лексемами і необхідно не пропустити їх при розпізнаванні тексту.

Як правило, сканери діють за таким принципом: черговий символ з вхідного потоку даних додається в лексему завжди, коли він може бути туди доданий. Як тільки символ не може бути доданий в лексему, то вважається, що він є кордоном лексеми і початком наступної лексеми (якщо символ не є порожнім роздільником – пропуском, знаком табуляції або переведення рядка, знаком коментаря). Такий принцип не завжди дозволяє правильно визначити межі лексем в тому випадку, коли вони не розділені порожніми символами.

Тому в більшості компіляторів лексичний і синтаксичний аналізатори – це взаємопов'язані частини. Можливі два принципово різних методи взаємозв'язку лексичного і синтаксичного аналізу:

* послідовний;
* паралельний.

При послідовній взаємодії лексичний аналізатор переглядає весь текст вихідної програми від початку до кінця і перетворює його в таблицю лексем. Таблиця лексем заповнюється відразу повністю, компілятор використовує її для наступних фаз компіляції. Подальшу обробку таблиці лексем виконують такі фази компіляції. Якщо лексичний аналізатор не зміг правильно визначити тип лексеми, то вважається, що вихідна програма містить помилку. Робота синтаксичного і лексичного аналізаторів в варіанті їх послідовної взаємодії зображена у вигляді схеми на рис. 2.1 [3, ст.101.].



Рисунок 2.1 – Послідовна взаємодія лексичного і синтаксичного аналізатора

При паралельній взаємодії лексичний аналіз тексту вихідної програми виконується поетапно по кроках. Лексичний аналізатор виділяє чергову лексему в вихідному коді і передає її синтаксичному аналізатору. Синтаксичний аналізатор може підтвердити правильність знайденої лексеми і звернутися до лексичному аналізатору за наступною лексемою або ж відкинути знайдену лексему. У другому випадку він може поінформувати лексичний аналізатор про те, що треба повернутися назад до вже переглянутого раніше фрагменту вихідного коду і повідомити йому додаткову інформацію про те, лексему якого типу слід очікувати. Взаємодіючи між собою таким чином, лексичний і синтаксичний аналізатори можуть перебрати кілька можливих варіантів лексем, і, якщо жоден з них не підходить, буде вважатися, що вихідна програма містить помилку. Тільки після того, як синтаксичний аналізатор успішно виконає розбір черговий конструкції вихідного мови, лексичний аналізатор поміщає знайдені лексеми в таблицю лексем і в таблицю ідентифікаторів і продовжує розбір далі в тому ж порядку. Робота синтаксичного і лексичного аналізаторів в варіанті їх паралельного взаємодії зображена у вигляді схеми на рис. 2.2 [3, ст.101.].

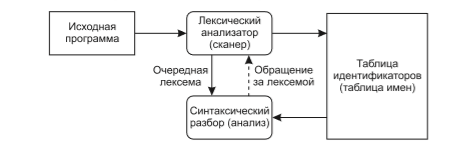


Рисунок 2.2 – Паралельна взаємодія лексичного і синтаксичного аналізатора

Послідовна робота лексичного і синтаксичного аналізаторів, представлена на рис. 2.1, є більш простим варіантом їх взаємодії. Вона простіше в реалізації і забезпечує більш високу швидкість роботи компілятора, ніж їх паралельне взаємодія, показане на рис. 2.2. Тому розробники компіляторів прагнуть організувати взаємодію лексичного і синтаксичного аналізаторів саме таким чином. Для більшості мов програмування кордону лексем розпізнаються по заданих термінальним символам. Ці символи – прогалини, знаки операцій, символи коментарів, а також роздільники (коми, крапки з комою та інше). Набір таких термінальних символів залежить від синтаксису вхідного мови. Важливо відзначити, що знаки операцій і роздільники самі також є лексемами. Але для багатьох мов програмування на етапі лексичного аналізу може бути недостатньо інформації для однозначного визначення типу і кордонів чергової лексеми. Однак навіть і тоді розробники компіляторів прагнуть уникнути паралельної роботи лексичного і синтаксичного аналізаторів. У ряді випадків допомагає принцип вибору з усіх можливих лексем лексеми найбільшої довжини: черговий символ з вхідного потоку додається в лексему завжди, коли він може бути туди доданий. Коли символ не може бути доданий в лексему, то вважається, що він є кордоном лексеми і початком наступної лексеми.

Однак розробники компіляторів свідомо йдуть на те, що відсікають деякі правильні, але не цілком зчитувальні варіанти вихідних програм. Спроби ускладнити лексичний розпізнавач неминуче приведуть до необхідності його взаємозв'язку з синтаксичним розбором. Це заважає організації їх паралельної роботи і неминуче знизить ефективність роботи всього компілятора. Накладні витрати ніяк не виправдовуються досягається ефектом – щоб розпізнавати рядки з сумнівними лексемами, досить лише зобов'язати користувача явно вказати за допомогою пробілів (або інших незначних символів) кордону лексем, що значно простіше.

Лексичний аналізатор працює прямо, якщо для даного вхідного тексту (ланцюжки символів вхідного мови) і поточного положення покажчика в ньому він визначає лексему, розташовану безпосередньо праворуч від покажчика, і зрушує сам покажчик вправо від частини тексту, що утворює цю лексему. При прямій роботі лексичного аналізатора можливо його послідовне взаємодія з синтаксичним розпізнавачем.

Лексичний аналізатор працює непрямо, якщо для даного вхідного тексту (ланцюжки символів вхідного мови), заданого типу лексеми і поточного положення покажчика в ньому він визначає лексему, розташовану безпосередньо праворуч від покажчика, і якщо вона відповідає необхідному типу, то зрушує покажчик вправо від частини тексту, що утворює цю лексему. На відміну від прямого варіанта даного лексичному аналізатору на вході потрібно задавати також і тип очікуваної лексеми. Тому при непрямий роботі лексичного аналізатора потрібно його паралельне взаємодія з синтаксичним розпізнавачем.

Виконання дій, пов'язаних з лексемами. Виконання дій в процесі розпізнавання лексем представляє для сканера набагато меншу проблему. Фактично СА, який лежить в основі розпізнавача лексем, повинен мати не тільки вхідну мову, а й вихідну. Він повинен не тільки вміти розпізнати правильну лексему на вході, а й породити пов'язану з нею послідовність символів на виході. У такій конфігурації СА перетворюється в кінцевий перетворювач.

Для сканера дії по виявленню лексеми можуть трактуватися дещо ширше, ніж тільки породження ланцюжка символів вихідного мови. Сканер повинен вміти виконувати такі дії, як запис знайденої лексеми в таблицю лексем, пошук її в таблиці символів і запис нової лексеми в таблицю символів. Набір дій визначається реалізацією компілятора. Зазвичай ці дії виконуються відразу ж після виявлення кінця розпізнається лексеми. В скінченному автоматі сканера ці дії можна відобразити порівняно просто – достатньо мати можливість з кожним переходом на графі автомата (або у функції переходів автомата) зв'язати виконання деякої довільної функції f (q, a), де q – поточний стан автомата, а – поточний вхідний символ. Функція може виконувати довільні дії, доступні сканера, в тому числі працювати з сховищами даних, наявними в компіляторі (функція може бути і порожньою – не виконувати ніяких дій). Таку функцію, якщо вона є, зазвичай записують на графі переходів КА під дугами, що з'єднують стани скінченного автомату.

Можливі й інші дії, передбачені реалізацією компілятора. Таку функцію f (q, a), якщо вона є, зазвичай записують на графі переходів кінцевого автомата під дугами, що з'єднують стану автомата. Функція f (q, a) може бути порожньою (не виконувати ніяких дій), тоді відповідний запис відсутній [7].

.

# 3 **РЕГУЛЯРНІ ГРАМАТИКИ ТА СКІНЧЕННІ АВТОМАТИ**

## 3.1 Регулярні та автоматні граматики

Регулярна граматика – формальна граматика типу 3 по ієрархії Хомського. Регулярні граматики визначають в точності всі регулярні мови, і тому еквівалентні кінцевим автоматам і регулярними виразами. Регулярні граматики є підмножиною контекстно-вільних.

До регулярних граматик відносяться два типи: ліволінійна та праволінійна граматики. Ліволінійні граматики G(VT,VN,P,S), можуть мати правила також двох видів: A→ aB, A→ a, де А та В нетермінальні символи, a термінальний символ. . Праволінійні граматики G(VT,VN,P,S), можуть мати правила двох видів: A→ Bа, A→ a, де А та В нетермінальні символи, a термінальний символ.

Доведено, що ці два класи граматик еквівалентні. Для будь-якої регулярної мови, задану праволінійною граматикою, може бути побудована ліволінійна граматика, що визначає еквівалентну мову; і навпаки – для будь-якої регулярної мови, заданого ліволінійною граматикою, може бути побудована праволінійна граматика, що задає еквівалентну мову. Різницю між ліволінійними та праволінійними граматиками закладається в тому, в якому порядку будуються речення мови: зліва направо для ліволінійних, або справа наліво для праволінійних.

Будь-яка контекстно-вільна граматика може бути легко перетворена в вид, в якому правила складаються тільки з ліво-регулярних або право-регулярних (для контекстно-вільних граматик допустимо наявність тих і інших одночасно). Отже, такі граматики можуть висловити все контекстно-вільні мови. Регулярні граматики можуть містити або ліво-регулярні правила, або право-регулярні, але не обидва види одночасно. Тому вони можуть описати лише підмножина мов, які називаються регулярними мовами.

Серед усіх регулярних граматик можна виділяти окремий клас – автоматні граматики. Вони також можуть мати правила двох видів ліволінійна і праволінійна. Різниця між автоматними граматиками і звичайними регулярними граматиками закладається в наступному: там, де в правилах звичайних регулярних граматик може бути присутній ланцюжок термінальних символів, в автоматних граматиках може бути присутній тільки один термінальний символ. Будь-яка автоматна граматика є регулярною граматикою, але не навпаки – не будь-яка регулярна граматика є автоматною.

Доведено, що класи звичайних регулярних граматик і автоматних граматик майже еквівалентні. Це означає, що для будь-якої мови, яка задана регулярною граматикою, можна побудувати автоматну граматику, що визначає майже еквівалентні мови. Щоб класи автоматних і регулярних граматик були повністю еквівалентні, в автоматних граматиках дозволяється правило, коли цільовий символ слідує до порожнього символу. При цьому цільовий символ не повинен зустрічатися в правій частині інших правил граматики. Тоді мова, задана автоматною граматикою, може включати порожній ланцюжок. Автоматні граматика, також так як звичайні ліволінійні і праволінійні граматики задають регулярні граматики

Існують алгоритми, які дозволяють перетворити звичайну регулярну граматику до автоматного вигляду – тобто побудувати еквівалентну їй автоматну граматику.

## 3.2 Формальне визначення скінченного автомату. Відмінність між детермінованим та недетермінованим скінченним автоматом

Кінцевий автомат (або просто FSM – Finite-state machine) це модель обчислень, заснована на гіпотетичної машині станів. В один момент часу тільки один стан може бути активним. Отже, для виконання будь-яких дій машина повинна змінювати свій стан.

Кінцевим автоматом (СА) називають п’ятірку наступного виду: ) де:

* Q – скінченна множина станів автомату;
* V – скінченна множина доступних вхідних символів (алфавіту автомату);
* Δ – функція переходів;
* – початковий стан автомату;
* – непорожня множина скінченних станів автомату.

Скінченний автомат називають повністю визначеним, якщо в кожному його стані існує функція переходу для всіх можливих вхідних символів.

Робота скінченного автомату представляє послідовність кроків. На кожному кроці роботи автомат знаходиться в одному зі своїх станів, на наступному кроці він може перейти на наступний крок роботи або залишитися на поточному стані. Те , в який стан перейде автомат на наступному кроці роботи, визначає функція переходів. Вона залежить не тільки від поточного стану, але й від символу з алфавіту, що був відправлений на вхід автомату. Коли функція переходу дозволяє декілька наступних станів автомату, то СА може перейти в будь-який з цих станів. На початку роботи автомат завжди знаходиться в початковому стані. Робота скінченного автомату продовжується доти доки на його вхід надходять символи з вхідного ланцюжка.

Конфігурацію скінченного автомату на кожному кроці роботи можна визначити у вигляді: поточного стану автомату, ланцюжку вхідних символів, положення вказівника в ланцюжку символів. Скінченний автомат приймає ланцюжок символів, якщо, отримав на вхід цей ланцюжок, він з початкового стану може перейти в один з кінцевих станів. В зворотному випадку скінченний автомат не приймає ланцюжок символів. Мов задана скінченним автоматом – це множина всіх ланцюжків символів які сприймаються даним автоматом. Два скінченних автомати еквівалентні якщо вони задають одну й ту ж мову.

Скінченні автомати зазвичай використовуються для організації та подання потоку виконання чого-небудь. Це особливо корисно при реалізації штучного інтелекту в іграх.

Діаграма станів (або іноді граф переходів) – графічне представлення безлічі станів і функції переходів. Являє собою розмічений орієнтований граф, вершини якого – стану КА, дуги – переходи з одного стану в інший, а мітки дуг – символи, за якими здійснюється перехід з одного стану в інший. Якщо перехід зі стану q1 в q2 може бути здійснений по одному з декількох символів, то всі вони повинні бути надписані над дугою діаграми.

Таблиця переходів – табличне представлення функції δ. Зазвичай в такій таблиці кожному рядку відповідає один стан, а одну – один допустимий вхідний символ. В осередку на перетині рядка і стовпця записується стан, в яке повинен перейти автомат, якщо в даному стані він вважав даний вхідний символ.

Скінченні автомати зазвичай представляють у вигляді графу переходів. Граф переходів скінченного автомату – це направлений граф, вершини якого позначені символами станів скінченного автомату, і в якому є дуга позначена символом . Початковий та кінцевий стан позначають спеціальним чином (початковий стан –додатковою пунктирною лінією, а кінцевий стан – додатковою суцільною лінією). Для моделювання роботи скінченного автомату його зручно перевести до повністю визначеного вигляду, щоб виключити ситуації, з яких немає переходів по вхідним символах. Для цього в скінченний автомат додають ще один стан, який можна умовно назвати «помилка». На цей стан переходять всі невизначені переходи, а всі переходи з нього замикають на нього ж.

Скінченні автомати підрозділяються на детерміновані і недетерміновані.

Детермінованим кінцевим автоматом (ДКА) називається такий автомат, в якому немає дуг з міткою ε (пропозиція, що не містить жодного символу), і з будь-якого стану по будь-якому символу можливий перехід не більше, ніж в один стан.

Недетермінований кінцевий автомат (НКА) є узагальненням детермінованого. Не детермінованість автоматів може досягатися двома способами: або можуть існувати переходи, помічені порожній ланцюжком ε, або з одного стану можуть виходити кілька переходів, позначених одним і тим же символом.

Багато скінченних автоматів можна мінімізувати. Мінімізація скінченного автомату закладається в побудові еквівалентного СА з меншим числом станів. В процесі мінімізації потрібно побудувати автомат з мінімально можливим числом станів, еквівалентний даному СА. Для мінімізації автомату використовується алгоритм побудови еквівалентних станів СА. Доведено, що алгоритм побудови множини класів еквівалентності завершиться максимум для n = m-2, де m – загальна кількість станів автомату.

Алгоритм мінімізації скінченного автомату закладається в наступному:

* з автомату виключаються всі недосяжні стани;
* будуються класи еквівалентності автомату;
* класи еквівалентності станів вихідного СА стають станами результуючого мінімізованого скінченного автомату;
* функція переходів результуючого СА звичайним чином будується на основі функції переходів вихідного скінченного автомату.

Для цього алгоритму доведено: по-перше, що він будує мінімізований скінченний автомат, еквівалентний заданому; по-друге, що він будує скінченний автомат з мінімально можливим числом станів.

## 3.3 Призначення та види подання скінченного автомату

Все різноманіття видів цифрових машин можна віднести до одного класу кінцевих автоматів. Це означає, що вони володіють загальними властивостями:

* по-перше, будь-яка цифрова обчислювальна машина складається з кінцевої безлічі елементів, кожен з яких в будь-який даний момент часу може знаходитися лише в одному з кінцевого числа стійких станів. Тому і вся машина має лише кінцеву безліч стійких станів;
* по-друге, кожна цифрова машина працює послідовно, її операції синхронізовані сигналами ретельно налаштованого електронного годинника (зазвичай від 106 до 109 сигналів в секунду). Відповідно до цього стану машини змінюються в чіткій послідовності;
* по-третє, цифрова обчислювальна машина є детермінованим пристроєм: при наявності повної інформації про внутрішніх станах всіх елементів машини і всіх її входів наступний стан машини визначено однозначно.
* без перебільшення можна сказати, що теорія автоматів є одним з фундаментальних блоків сучасної теоретичної і практичної інформатики. Разом з класичними додатками теорії автоматів, такими як проектування вбудованих систем логічного управління, обробка текстів і побудова компіляторів штучних мов, останнім часом з'явилися нові, нетрадиційні області застосування цієї теорії – специфікація і верифікація систем взаємодіючих процесів (зокрема, протоколів комунікації), мови опису документів і об'єктно-орієнтованих програмних систем, оптимізація, логічних програм.

Скінченні автомати значно розширюють можливості управління виконанням фонових завдань. Їх використання робить можливим надання фоновими потоками інформації про стан виконання, а також звернення інших потоків із запитами до фонового потоку на виконання певних дій, наприклад, із запитом на припинення виконання фонової роботи.

Додатки найкраще проектувати на основі одно поточному моделі, залучаючи фонову обробку лише тоді, коли вона необхідна.

Скінченні автомати вельми зручно застосовувати в іграх, в яких персонажі, що переміщаються в області гри, можуть діяти в різних режимах (наприклад, персонаж біжить вперед або назад, підіймається сходами, падає та інше).

Використання автоматів з чітко визначеними правилами для кожного персонажа, які задають варіанти його поведінки в різних станах і допустимі переходи з одного стану в інший, дозволяє створювати складні варіанти поведінки як персонажів, керованих комп'ютером, так і персонажів, керованих користувачем.

Ідея застосування кінцевих автоматів є надзвичайно корисною концепцією, плідність якої пройшла перевірку часом. Використання кінцевих автоматів дозволяє розробникам створювати добре організовані додатки з гнучкими можливостями. Їх застосування дозволяє створювати ясний, зрозумілий і надійно функціонує код.

Намагайтеся уникати неявного управління станами програми, але не шкодуйте часу на розробку явно сформованої моделі станів.

Використання кінцевих автоматів стало вже звичайною при проектуванні додатків для настільних комп'ютерів, серверів і мобільних пристроїв.

Переваги, які забезпечує застосуванням кінцевих автоматів, найпомітніше проявляються в разі додатків для мобільних пристроїв, що вимагають економного розподілу екранного простору, пам'яті, обчислювальної потужності і інших ресурсів.

Можна визначити основні переваги скінченного автомату:

* формалізація. При описі завдання через кінцевий автомат необхідно замислюватися про всі станах, в яких може опинитися об'єкт. Так отримуємо і документацію, і можемо виявити не розглянуті в постановці завдання моменти. Відповідно, спрощується тестування коду;
* контроль потоків даних. Явне управління послідовністю викликів (потоком управління);
* контроль помилок. Якщо кінцевий автомат потрапляє в помилкове стан, то це просто означає, що при проектуванні забули визначити ще один стан. Єдина точка входу для логування та збору статистики. Наприклад, SwiftyStateMachine дозволяє явно вказати конкретний блок, в якому можна залоговані, що відбувається з нашими даними. Це істотно спрощує налагодження додатків;
* історія операцій. Використовуючи кінцевий автомат, можна реалізувати скасування операцій. Або, навпаки, відновити всю картину переходів між станами. Стек операцій зазвичай зберігається в самому стані.

Призначені для користувача інтерфейси – одна з найбільш очевидних областей, де скінченні автомати можуть знаходити широке застосування. Поведінка елементів управління, спільно використовують одну і ту ж форму. Деякі елементи керування відображаються і ховаються в один і той же час, деякі елементи переміщаються при відображенні інших елементів та інше.

Кінцевий автомат підходить для управління всіма потребами призначеного для користувача інтерфейсу мобільного додатка. При цьому необхідно стежити за ефективним використанням екранного простору в процесі того, як користувач переводить додаток з одного стану в інший. При наявності кінцевого автомата, який показує, приховує і переміщує елементи управління по екрану відповідно до необхідності, це завдання можна вирішити досить ефективно.

Абстрагування всіх режимів роботи призначеного для користувача інтерфейсу в одному автоматі забезпечує максимальну гнучкість процесу внесення змін до моделей екранного дисплея, позбавляючи від необхідності переглядати і змінювати безліч коду, розподіленого між різними функціями і обробниками подій користувальницького інтерфейсу.

Одна з популярних задач аналізу тексту, пов'язана із застосуванням кінцевого автомата, – спам-фільтри. Нехай є набір стоп-слів і вхідна послідовність. Потрібно або відфільтрувати цю послідовність, або взагалі її не виводити. Формально, це завдання пошуку під рядка в певному рядку. Для її вирішення використовується алгоритм Кнута-Морріса-Пратта, програмна реалізація якого є кінцевий автомат. Як стану виступає зміщення вхідної послідовності і кількість знайдених символів в патерні – стоп-слові.

Ще одним варіантом застосування кінцевого автомата є app coordinators – це набір інструкцій, набір послідовності жорстко заданих дій, повністю описує призначену для користувача історію: авторизацію, виконання замовлення і так далі.

Скінченні автомати значно розширюють можливості управління виконанням фонових завдань. Їх використання робить можливим надання фоновими потоками інформації про стан виконання, а також звернення інших потоків із запитами до фонового потоку на виконання певних дій, наприклад, із запитом на припинення виконання фонової роботи.

Ідея застосування кінцевих автоматів є надзвичайно корисною концепцією, плідність якої пройшла перевірку часом. Використання кінцевих автоматів дозволяє розробникам створювати добре організовані додатки з гнучкими можливостями. Їх застосування дозволяє створювати ясний, зрозумілий і надійно функціонує код.

Отже, скінченні автомати варто застосовувати, коли дійсно потрібна формалізація, висока тестованість коду і потрібно розподілити завдання між розробниками [1].

## 3.4 Побудова скінченного автомату на основі ліволінійної граматики

На основі заданої регулярної граматики можна побудувати еквівалентний їй скінченний автомат і, навпаки, для заданого скінченного автомату можна побудувати еквівалентну йому регулярну граматику.

Регулярні граматики використовуються для визначення лексичних конструкцій мов програмування. Створивши автомат на основі відомої граматики, ми отримаємо розпізнавач для лексичної конструкції даної мови. Таким чином вдається вирішити задачу розбору для лексичних конструкцій мови, заданих довільною регулярної граматики.

Задача формується наступним чином: є ліволінійна граматика, що задає мову, необхідно побудувати еквівалентний їй скінченний автомат, що задає ту ж мову. Задача вирішується в два етапи:

* вхідну ліволінійну граматику потрібно перевести до автоматного вигляду;
* на основі отриманої автоматної ліволінійної граматики будується шуканий автомат.

Алгоритм перетворення до автоматного виду:

* побудувати множину нетермінальних символів;
* почати продивлятись множину правил Р граматики G, для правил що мають більше ніж один термінальний символ створюються нетермінальні символи та правила з цими символами, які містять по одному термінальному символу, правила що містять тільки термінальний символ, або по одному термінальному та нетермінальному символу переносяться до множини правил Р автоматної граматики;
* продивлятися множину правил Р граматики G. Якщо знаходиться правило виду А→В, якщо в ньому присутні правила виду В→С, В→ Са, В→ а то в нього додаються правила виду А→С, А→ Са, А→ а (при цьому потрібно враховувати, що в граматиці не повинно мати однакових правил, і якщо якісь правила вже присутні в граматиці, то повторно його додавати не потрібно);
* якщо на кроці 3 не було знайдено хоча б одне правило вигляду А→В або де є порожня множина то потрібно повторити крок 3, або ж перейти до кроку 5;
* цільовий символ граматики G стає символ S.

Алгоритм побудови скінченного автомату на основі ліволінійної граматики:

* будуємо стани автомату Q. Стани автомату будуються таким чином, щоб кожному нетермінальному символу з множини нетермінальних символів граматики співпадав один стан з множини Q автомату. Крім того, в множину станів автомату додається ще один додатковий стан, що буде позначатися H. Зберігаючи позначення нетермінальних символів граматики G, для множини станів автомату;
* вхідним алфавітом автомату М є множина термінальних символів граматики G;
* продивляємось всі множини правил вхідної граматики. Якщо зустрічається правило виду А→ а, то в функцію переходів автомату додаємо стан (Н, а) = А. Якщо зустрічається правило виду А→ Ва, то в функцію переходів автомату М додаємо стан (В, а) = А;
* початковим станом автомату М є стан Н;
* множина скінченних станів автомату М складається з одного стану. Цей стан є станом, співпадаючи цільовому символу граматику.

# 4 РОЗРОБКА ПРОГРАМНОГО ПРОДУКТУ

## 4.1 Технічне завдання на розробку програми

До дипломної роботи було поставлено наступне технічне завдання: створити програмне забезпечення яке дає можливість перетворити регулярну граматику, яку користувач задає власноруч, до скінченного автомату.

Після аналізу заданої теми, було поставлено наступні вимоги до розробки програмного продукту:

* оптимізувати програмний продукт, для збільшення швидкодії, швидкості на виконання певної дії, для користувачів з застарілим програмним або апаратним забезпеченням;
* створити програмний інтерфейс який буде зрозумілий для використання користувачам, які не розуміють принципи системного програмування, а також застосування до інтерфейсу правил ергономічності, гарної взаємодії людини та техніки;
* створення кросплатформенного програмного продукту на мові програмування С#, для запуску програми на операційних системах Windows, Linux;
* використання патерну проектування MVC для розділення коду програми на логічні структури;
* використання зручної середи програмування Visual Studio для збільшення швидкості розробки програмного забезпечення на мові програмування C#;
* використання Entity фреймворку, для створення ергономічного та новітнього інтерфейсу;
* створення функцій програмного продукту для перегляду функцій переходу скінченного автомату, графу перетворення скінченного автомату, структуру скінченного автомату.

Основними вимогами для програмного та апаратного забезпечення користувача, для забезпечення ефективності програмного продукту, є:

* ліцензійна операційна система Windows або Linux;
* встановлений фреймворк .Net версії 4.5.1та вище, для запуску додатків на мові програмування C#;
* графічна карта з відео пам’яттю 512 мегабайт та більше, для створення графу скінченного автомату.

## 4.2 Вибір і обґрунтування критеріїв ефективності та якості ПЗ

Провівши аналіз програмного забезпечення було перевірено його надійність та здатність бути стійким при помилках, на основі аналізу було побудовано основні UML діаграми: діаграма прецедентів, діаграма стійкості, діаграма послідовності, діаграма станів та діаграма пакетів.

Діаграма прецедентів – в UML, діаграма, на якій зображено відношення між акторами та прецедентами в системі. Також, перекладається як діаграма варіантів використання. Вона містить дані про взаємодію користувача та основних форм та дій над ними, зображена на рис. 4.1.

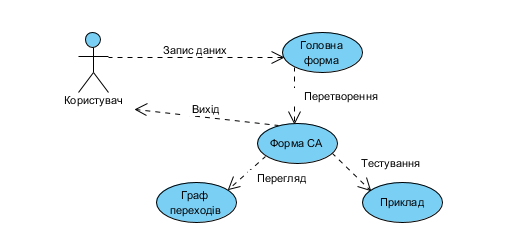


Рисунок 4.1 – Діаграма прецедентів

Діаграма стійкості розробляється на базі шаблону (або патерну – pattern) проектування MVC (Model-View-Controller). Тобто в системі, що розробляється, мають бути наявні три типи програмних об’єктів:

* Model – це об’єкти, які моделюють дані предметної області;
* View – це об’єкти, які реалізують відображення даних із моделі;
* Controller – це об’єкти, які обробляють дані моделі для подальшого їх відображення.

Основна ідея шаблону MVC полягає у відокремленні даних від їх відображення. Таким чином, якщо у процесі розробки виникне потреба у зміненні моделі даних, це ніяк не впливає на відображення (не треба буде нічого змінювати у компонентах View). Діаграма стійкості зображена на рис. 4.2.

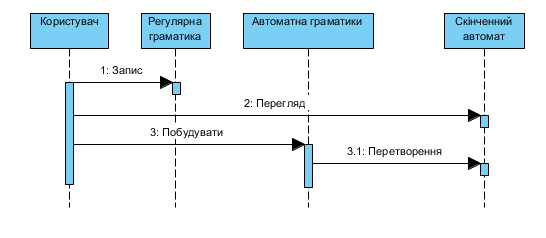


Рисунок 4.2 – Діаграма стійкості

Діаграма послідовності ([англ.](https://uk.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D1%96%D0%B9%D1%81%D1%8C%D0%BA%D0%B0_%D0%BC%D0%BE%D0%B2%D0%B0" \o "Англійська мова) sequence diagram) — різновид діаграми в [UML](https://uk.wikipedia.org/wiki/UML). Діаграма послідовності відображає взаємодії об'єктів впорядкованих за часом. На діаграмі зображуються лише ті об’єкти, які безпосередньо беруть участь у взаємодії. Крайнім зліва зображається об’єкт який являється ініціатором взаємодії, з права від нього об’єкт з яким він взаємодіє. Зокрема, такі діаграми відображають задіяні [об'єкти](https://uk.wikipedia.org/wiki/%D0%9E%D0%B1%27%D1%94%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)) та послідовність відправлених повідомлень. На ній відображена послідовність дій користувача та програми поступово, зображена діаграма на рис. 4.3.

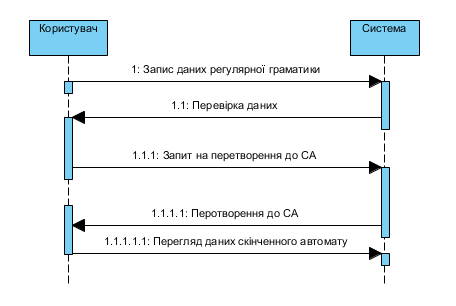


Рисунок 4.3 – Діаграма послідовності

Діаграми пакетів уніфікованої мови моделювання ([UML](https://uk.wikipedia.org/wiki/Unified_Modeling_Language)) відображають залежності між пакетами, з яких і складається модель.

Пакет (package) — елемент моделі, який використовують для групування інших елементів моделі. Елементи моделі, які входять до складу певного пакету, називаються членами пакету. Пакет володіє усіма своїми членами. Про членів пакету кажуть, що вони є у власності пакету, тобто належать йому. Якщо певний пакет видаляється з моделі, то з неї також видаляються усі члени, що знаходяться у власності цього пакету. Вважається, що між двома елементами існує залежність якщо зміни в визначенні одного елементу можуть викликати зміни в другому елементі. Зображена діаграма на рис. 4.4.

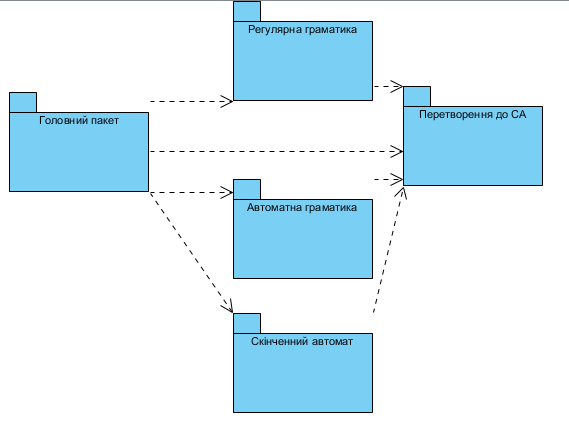


Рисунок 4.4 – Діаграма пакетів

Діаграма станів — діаграма, що визначає зміну станів об'єкта у часі, одна з діаграм моделювання поведінки в [UML](https://uk.wikipedia.org/wiki/UML). Представляє об'єкт як [автомат з теорії автоматів](https://uk.wikipedia.org/wiki/%D0%A1%D0%BA%D1%96%D0%BD%D1%87%D0%B5%D0%BD%D0%BD%D0%B8%D0%B9_%D0%B0%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82) зі стандартизованими умовними позначеннями. Вершинами графу є можливі стани автомата, зображувані відповідними графічними символами, а дуги позначають його переходи зі стану в стан.

Елементами діаграми є:

* коло, що позначає початковий стан;
* коло з маленьким колом усередині, що позначає кінцевий стан (якщо є);
* округлений прямокутник, що позначає окремий стан. Верхівка прямокутника містить назву стану, в середині може бути горизонтальна лінія, під якою записуються активності, що відбуваються в даному стані;
* стрілка, що позначає перехід. Назва події (якщо є), що викликає перехід, відзначається над/під стрілкою. [Вартовий вираз](https://uk.wikipedia.org/wiki/%D0%92%D0%B0%D1%80%D1%82%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)) може бути доданий перед «/» і укладений у квадратні дужки (назва\_події), він означає, що перехід відбувається лише за умови істинності виразу. Якщо при переході відбувається якась активність, то воно додається після «/» (назва події);
* товста горизонтальна лінія, яка є точкою об'єднання або розгалуження переходів.

Зображена діаграма станів на рис. 4.5.

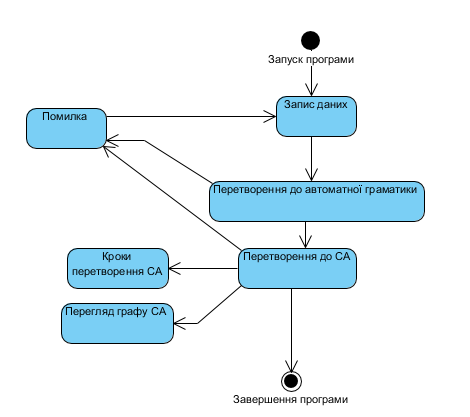


Рисунок 4.5 – Діаграма станів

## 4.3 Проектування інтерфейсу користувача

Як основну мову програмування для створення програмного продукту було обрано мову C#.

Мова C# є дуже близьким родичем мови програмування Java. Мова Java була створена компанією Sun Microsystems, коли глобальний розвиток інтернету поставив задачу розсереджених обчислень. Взявши за основу популярну мову C++, Java виключила з неї потенційно небезпечні речі (типу вказівників без контролю виходу за межі). Для розсереджених обчислень була створена концепція віртуальної машини та машинно-незалежного байт-коду, свого роду посередника між вихідним текстом програм і апаратними інструкціями комп'ютера чи іншого інтелектуального пристрою.

Мова Java набула чималої популярності, і була ліцензована також і компанією Microsoft. Але з плином часу Sun почала винуватити Microsoft, що та при створенні свого клону Java робить її сумісною виключно з платформою Windows, чим суперечить самій концепції машинно-незалежного середовища виконання і порушує ліцензійну угоду. Microsoft відмовилася піти назустріч вимогам Sun, і тому з'ясування стосунків набуло статусу судового процесу. Суд визнав позицію Sun справедливою, і зобов'язав Microsoft відмовитися від позаліцензійного використання Java.

У цій ситуації в Microsoft вирішили, користуючись своєю вагою на ринку, створити свій власний аналог Java, мови, в якій корпорація стане повновладним господарем. Ця новостворена мова отримала назву C#. Вона успадкувала від Java концепції віртуальної машини (середовище .NET), байт-коду (MSIL) і більшої безпеки вихідного коду програм, плюс врахувала досвід використання програм на Java.

Нововведення до C# надало можливість зручнішої взаємодії порівняно з мовами-попередниками, з кодом програм, написаних на інших мовах, що є суттєвим при створенні великих проектів. Використання платформи .NET надає можливість сумісного виконання програм, розроблених на різних мовах.

Станом на сьогодні, мову C# визначено флагманською мовою корпорації Microsoft, бо вона найповніше використовує нові можливості .NET. Решта мов програмування, хоч і підтримуються, але визнані такими, що мають спадкові прогалини щодо використання .NET.

C# розроблялась як мова програмування прикладного рівня для CLR(Common Language Runtime, Загальномовне Середовище Виконання) і тому вона залежить, перш за все, від можливостей самої CLR. Це стосується, перш за все, системи типів, що використовуються в C#. Присутність або відсутність тих або інших виразних особливостей мови диктується тим, чи може конкретна мовна особливість бути трансльована у відповідні конструкції CLR. Так, з розвитком CLR від версії 1.1 до 2.0 значно збагатився і сам C#; подібної взаємодії слід чекати і надалі. (Проте ця закономірність буде порушена з виходом C# 3.0, що є розширеннями мови, що не спираються на розширення платформи .NET.) CLR надає C#, як і всім іншим .NET-орієнтованим мовам, багато можливостей, яких позбавлені «класичні» мови програмування. Наприклад, збірка сміття не реалізована в самому C#, а проводиться CLR для програм, написаних на C# точно так, як і це робиться для програм на VB.NET, J# тощо.

Специфікація C# визначає мінімальний набір бібліотек типів і класів, на який має розраховувати компілятор. На практиці, C# найчастіше використовується з якоюсь реалізацією CLI (Common Language Infrastructure, Загальна Мовна Інфраструктура), яка стандартизована як ECMA-335 CLI.

C# стандартизована в ECMA та ISO.

У серпні 2000 Microsoft Corporation, Hewlett-Packard та Intel Corporation виступили спонсорами стандартизації специфікації мови C#, а також CLI в організації зі стандартизації ECMA International. У грудні 2001 ECMA випустила ECMA-334 Специфікація мови C#. C# стала стандартом ISO у 2003 (ISO/IEC 23270:2006 – Information technology – Programming languages – C#). До того ECMA ще встигла адаптувати еквівалентну специфікацію як другу редакцію C# у грудні 2002.

У червні 2005 ECMA схвалила редакцію 3 специфікації C#, і відредагувала ECMA-334. Доповнення включали часткові класи, анонімні методи, тип null, і генерики (аналоги шаблонів C++).

Для створення користувацького інтерфейсу на базі мови програмування C# було обрано Entity Framework.

Entity Framework являє собою спеціальну об'єктно-орієнтовану технологію на базі фреймворка .NET для роботи з даними. Якщо традиційні засоби ADO.NET дозволяють створювати підключення, команди та інші об'єкти для взаємодії з базами даних, то Entity Framework являє собою більш високий рівень абстракції, який дозволяє абстрагуватися від самої бази даних і працювати з даними незалежно від типу сховища. Якщо на фізичному рівні ми оперуємо таблицями, індексами, первинними і зовнішніми ключами, але на концептуальному рівні, який нам пропонує Entity Framework, ми вже працює з об'єктами.

Перша версія Entity Framework – 1.0 вийшла ще в 2008 році і представляла дуже обмежену функціональність, базову підтримку ORM (object-relational mapping – відображення даних на реальні об'єкти) і один єдиний підхід до взаємодії з бд – Database First. З виходом версії 4.0 у 2010 році багато чого змінилося – з цього часу Entity Framework став рекомендованої технологією для доступу до даних, а в сам фреймворк були введені нові можливості взаємодії з бд – підходи Model First і Code First.

Додаткові поліпшення функціоналу пішли з виходом версії 5.0 в 2012 році. І нарешті, в 2013 році був випущений Entity Framework 6.0, що володіє можливістю асинхронного доступу до даних.

Центральною концепцією Entity Framework є поняття сутності або entity. Сутність представляє набір даних, асоційованих з певним об'єктом. Тому дана технологія передбачає роботу не з таблицями, а з об'єктами і їх наборами [5].

## 4.4 Алгоритм роботи програмного продукту

На основі заданої регулярної граматики можна побудувати еквівалентний їй скінченний автомат і, навпаки, для заданого скінченного автомату можна побудувати еквівалентну йому регулярну граматику.

Регулярні граматики використовуються для визначення лексичних конструкцій мов програмування. Створивши автомат на основі відомої граматики, ми отримаємо розпізнавач для лексичної конструкції даної мови. Таким чином вдається вирішити задачу розбору для лексичних конструкцій мови, заданих довільною регулярної граматики.

Задача формується наступним чином: є ліволінійна граматика, що задає мову, необхідно побудувати еквівалентну їй автоматну граматику, після чого потрібно перетворити її в скінченний автомат, що задає ту ж мову. Задача вирішується в два етапи:

* вхідну ліволінійну граматику потрібно перевести до автоматного вигляду;
* на основі отриманої автоматної ліволінійної граматики будується шуканий автомат.

Алгоритм перетворення регулярної граматики заданої користувачем до автоматної граматики:

* побудувати множину нетермінальних символів вона буде такою ж як і множина символів регулярної граматики;
* почати продивлятись множину правил Р граматики G, для правил що мають більше ніж один термінальний символ створюються нетермінальні символи та правила з цими символами, які містять по одному термінальному символу, правила що містять тільки термінальний символ, або по одному термінальному та нетермінальному символу переносяться до множини правил Р автоматної граматики;
* продивлятися множину правил Р граматики G. Якщо знаходиться правило виду А→В, якщо в ньому присутні правила виду В→С, В→ Са, В→ а то в нього додаються правила виду А→С, А→ Са, А→ а (при цьому потрібно враховувати, що в граматиці не повинно мати однакових правил, і якщо якісь правила вже присутні в граматиці, то повторно його додавати не потрібно);
* якщо на кроці 3 не було знайдено хоча б одне правило вигляду А→В або якщо правило слідує до порожньої множини то потрібно повторити крок 3, або ж перейти до кроку 5;
* цільовий символ граматики G стає символ S.

Алгоритм побудови скінченного автомату на основі перетвореної до автоматного виду регулярної граматики:

* будуємо стани автомату Q. Стани автомату будуються таким чином, щоб кожному нетермінальному символу з множини нетермінальних символів граматики співпадав один стан з множини Q автомату. Крім того, в множину станів автомату додається ще один додатковий стан, що буде позначатися H. Зберігаючи позначення нетермінальних символів граматики G, для множини станів автомату;
* вхідним алфавітом автомату М є множина термінальних символів граматики G;
* продивляємось всі множини правил вхідної граматики. Якщо зустрічається правило виду А→ а, то в функцію переходів автомату додаємо стан (Н, а) = А. Якщо зустрічається правило виду А→ Ва, то в функцію переходів автомату М додаємо стан (В, а) = А;
* початковим станом автомату М є стан Н;
* множина скінченних станів автомату М складається з одного стану. Цей стан є станом, співпадаючи цільовому символу граматику.

Алгоритм роботи програми зображено на рис. 4.6:

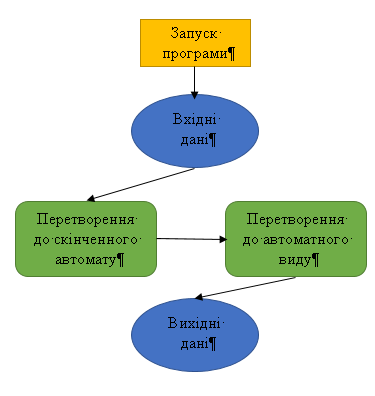


Рисунок 4.6 – Алгоритм роботи програми

## 4.5 Інструкція з експлуатації програмного продукту

Для запуску даного програмного продукту потрібно щоб на комп’ютері був встановлений .Net Framework, потім запускаємо exe файл, перед користувачем з’явиться головне меню програми, зображене на рис. 4.7.

Програма дозволяє ввести власні нетермінальні символи, термінальні символи, цільовий символ, а також заповнити граматику власними правилами з термінальних та нетермінальних символами, що задав користувач. При натисканні на кнопку «Додати правило» до граматики буде додано правило, якщо воно співпадає з правилами запису регулярних граматик. Натиснувши на кнопку «Приклад», граматика автоматично заповниться нетермінальними символам, термінальними символами, цільовим символом а також правилами регулярної граматики механічного годинника, користувач може додавати свої символи та правила. Натиснувши на кнопку «Перетворити до СА», користувач потрапить до меню скінченного автомату, програма автоматично перетворить регулярну граматику до автоматного виду, після чого перетворить її до скінченного автомату.



Рисунок 4.7 – Зображення головного меню програми

В меню скінченного автомату, що зображене на рис. 4.8, користувач має можливість продивитися автоматну граматику, яка була отримана шляхом перетворення регулярної граматики до автоматного виду, а також запис скінченного автомату, також для користувача буде наведено список правил автоматної граматики, а також функції переходів для скінченного автомату. Також користувачу буде доступно три кнопки. Натиснувши на кнопку «Показати граф СА» користувач зможе продивитися граф скінченного автомату. Натиснувши на кнопку «Кроки переведення до СА» користувач може продивитися кроки переведення до скінченного автомату. Натиснувши на кнопку «Показати приклад» користувач має змогу продивитися фізичне представлення скінченного автомату у вигляді механічного годинника.

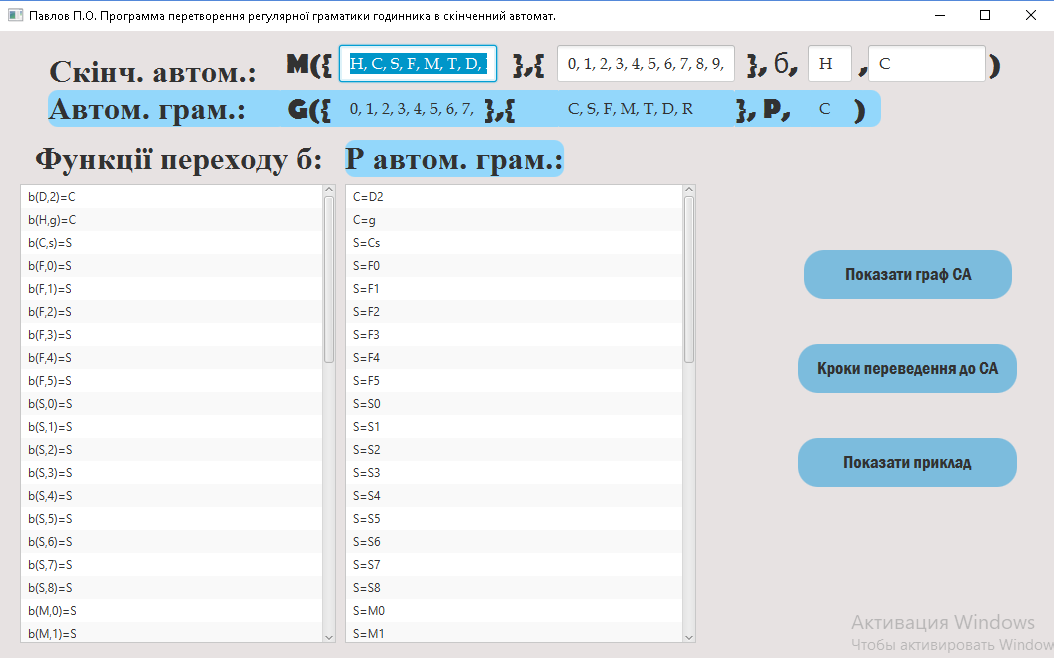


Рисунок 4.8 – Зображення меню скінченного автомату

На основі функцій переходів програма може побудувати граф скінченного автомату. В даній програмі граф малюється за допомогою HTML сторінки а також мови програмування JavaScript, після чого відображається як веб-сторінка в окремому вікні програми. Граф — це сукупність об'єктів із зв'язками між ними. Об'єкти розглядаються як вершини, або вузли графу, а зв'язки — як дуги, або ребра. Для різних областей використання види графів можуть відрізнятися орієнтованістю, обмеженнями на кількість зв'язків і додатковими даними про вершини або ребра. Граф скінченного автомату зображено на рис. 4.9.

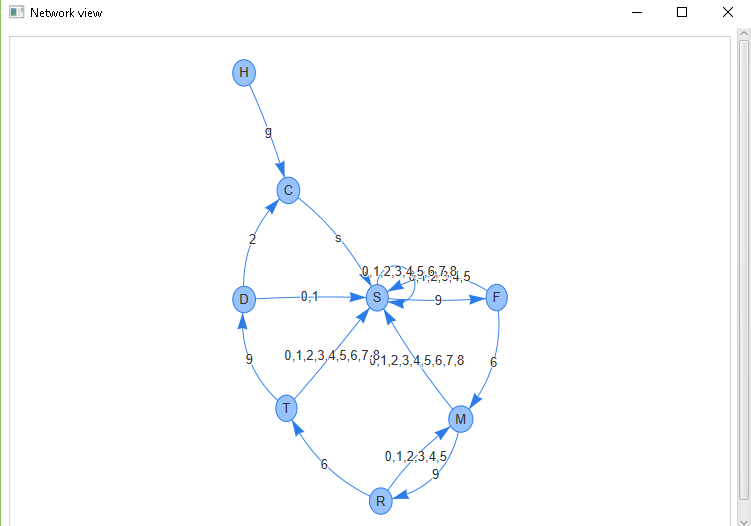


Рисунок 4.9 – Зображення графу скінченного автомату

У вікні кроків переведення до скінченного автомату, записані дії які виконувала програма спочатку для переведення до автоматного виду, програма записувала всі зміни, а також кроки створення автоматної граматики, потім дії переведення з автоматної граматики до скінченного автомату, додавання даних, а також створення функцій переходів. Вікно кроків зображене на рис. 4.10.

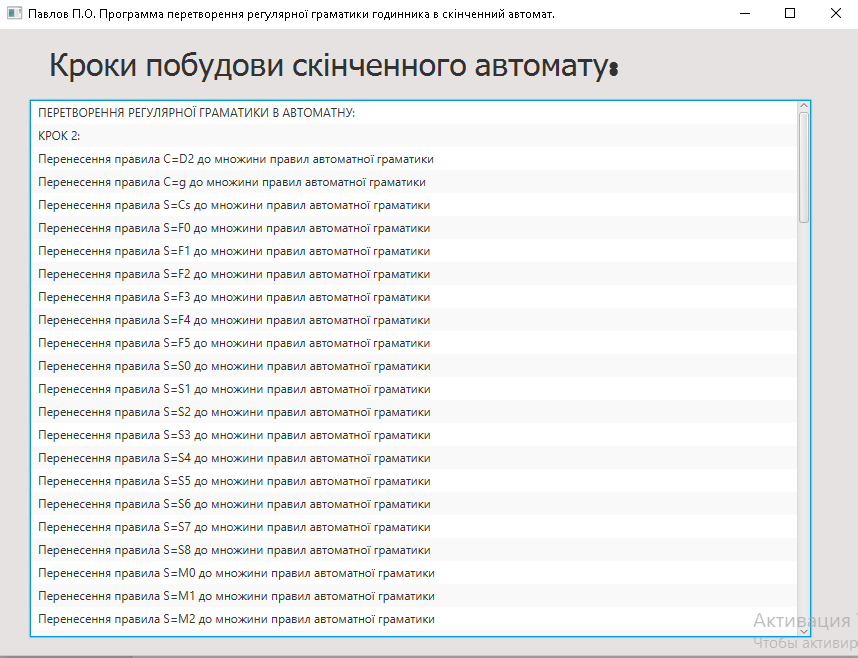


Рисунок 4.10 – Зображення форми з кроками побудови скінченного автомату

Для показу роботи скінченного автомату в програмі було створено приклад його використання. Найбільш застосовуваним в побуті пристроєм є механічний годинник, тому саме його було створено з використанням скінченного автомату. Годинник був створений за допомогою мови розмітки HTML та стилізований за допомогою каскадних таблиць CSS, для створення функціональності годинника з використанням скінченного автомату було створено анімації за допомогою скриптової мови програмування JavaScript. Зображений приклад на рис. 4.11.



Рисунок 4.11 – Фізичне представлення скінченного автомату у вигляді механічного годинника

# ВИСНОВКИ

В результаті виконання даної роботи була здійснена розробка програмного продутку для перетворення заданої користувачем регулярної граматики до скінченного автомату. Дане програмне забезпечення може бути активно використане у побудові скінченних автоматів для автоматизації тестування програмних продуктів, створенні логіки мобільних додатків, а також студентами при виконанні практичних та лабораторних робіт при проектуванні скінченних автоматів.

Тема перетворення регулярної граматики до скінченного автомату є досить цікавою і представляє собою широкі можливості для подальших досліджень в галузі скінченних автоматів.

Також було розглянуто деталі загального алгоритму перетворення регулярної граматики до автоматного вигляду, та автоматної граматики до скінченного автомату, та наведені практичні рекомендації по покращенню ефективності роботи алгоритму.

Результатом роботи є готовий до використання програмний продукт, який задовольняє вимогам поставленим до системи користувачем, та повністю відповідає до цілей наведених у роботі.

В цілому ідея перетворення регулярної граматики до скінченного автомату є досить перспективною для використання її у широкому виробництві. Тому представлені напрямки подальшої роботи в цій галузі у вигляді розробки фреймворку для перетворення регулярної граматики до скінченного автомату, а також інших алгоритмів на основі скінченного автомату. Наведені думки свідчать про досить велику площадку для подальших досліджень в області скінченних автоматів.

Використання скінченних автоматів набирає все більших обертів в сучасному виробництві програмного забезпечення, і має всі можливості для того щоб стати основною технологією для розробки та тестування програмних продуктів.

# **ПЕРЕЛІК ПОСИЛАНЬ**

1. Дехтярь М.И. Введение в схемы, автоматы и алгоритмы. – М.: Наука, 2002. С. 642.
2. Коган Д.И., Бабкина Т.С. Концепции конечного автомата и регулярного языка. Операции над регулярными языками. М.: Наука, 2000.
3. Молчанов А.Ю. Системное програмное обеспечение. – М.: Питер, 2003. – 392с.
4. Professor Web [Електронний ресурс] – Режим доступу до ресурсу: https://professorweb.ru/.
5. Metanit.com [Електронний ресурс] – Режим доступу до ресурсу: https://metanit.com/cpp/tutorial/
6. Cyberforum [Електронний ресурс] – Режим доступу до ресурсу: http://www.cyberforum.ru/
7. Devcolibri [Електронний ресурс] – Режим доступу до ресурсу: https://devcolibri.com
8. StackOverflow [Електронний ресурс] – Режим доступу до ресурсу: https://stackoverflow.com/