УДК 004.41

ДЕКОМПОЗИЦІЯ ЛОГІЧНОГО ВИРАЗУ: ПОКРОКОВА ВІЗУАЛІЗАЦІЯ ОБЧИСЛЕНЬ

О. І. Ніколаєв
здобувач вищої освіти першого (бакалаврського) рівня, 2 курс,
спеціальності «Інженерія програмного забезпечення»,
навчально-науковий інститут кібернетики, інформаційних технологій та інженерії
Науковий керівник – к.т.н., доцент О. Р. Мічута

Національний університет водного господарства та природокористування,
м. Рівне, Україна

У роботі представлено програмний інструмент для аналізу логічних виразів із покроковою візуалізацією процесу обчислень. Застосунок дозволяє обробляти логічні вирази, будувати таблиці істинності, ідентифікувати їхні характеристики (тавтологія, суперечність, нейтральність) та сприяє глибшому розумінню основ алгебри висловлень студентами.
Ключові слова: алгебра висловлень, булева логіка, логічний вираз, таблиця істинності, інфіксна форма, постфіксна форма, алгоритм сортувальної станції.

The work presents a software tool for analysing logical expressions with step-by-step visualisation of the calculation process. The application allows processing logical expressions, building truth tables, identifying their characteristics (tautology, contradiction, neutrality) and promotes a deeper understanding of the basics of expression algebra by students.
Keywords: algebra of expressions, Boolean logic, logical expression, truth table, infixed form, postfixed form, shunting yard algorithm.

Освоєння теми логічних операцій і виразів є фундаментальним для студентів, що вивчають інформатику, математику та інші технічні науки. Логічні вирази та операції є основою для побудови алгоритмів, аналізу даних, умовних конструкцій у програмуванні та розробки експертних систем. Сучасна освітня програма вимагає від студентів не лише вміння розуміти ці концепції, а й ефективно їх застосовувати в задачах різного рівня складності. Для студентів, які ще не мали справи з програмуванням або які мають труднощі з математикою, розуміння принципів роботи логічних операцій та їх спрощень і перетворень у виразах стає серйозним викликом через абстрактність теми й складності побудови таблиць істинності для логічних виразів. Тому особливо актуальним є створення інструменту, що допоможе поступово опанувати цю тему завдяки покроковим поясненням процесу обчислення результатів.

Метою роботи було створення програмного інструменту, що допоможе користувачам (зокрема студентам) краще зрозуміти й опрацьовувати логічні вирази шляхом їх покрокового розбору та побудови таблиць істинності.

Для досягнення мети було виконано наступні **завдання**:

1. Розробити алгоритм для обробки введених логічних виразів згідно з пріоритетом виконання операцій.
2. Побудувати алгоритм створення таблиці істинності, який дасть змогу відобразити всі можливі комбінації значень для змінних у виразі та результати їхнього виконання, а також автоматично охарактеризувати його.

3. Розробити програмний комплекс зі зручним користувацьким інтерфейсом, який буде містити в собі вищенаведені завдання.

Математичні нотації для аналізу пріоритету. Математика впродовж своєї історії розвивала різні способи запису виразів і операцій. Одним з найпоширеніших є **інфіксний запис** – спосіб, в якому операції розміщуються між операндами. Проте цей спосіб не завжди був ідеальним для обчислень, особливо у контексті комп'ютерних обчислень. Тому були розроблені альтернативи, і одним із найцікавіших та корисних є **постфіксний запис** або **зворотний польський запис (RPN – Reverse Polish Notation)**, у якому операції йдуть після операндів. І хоча інфіксний запис використовується вже тисячі років, ідея постфіксного запису народилася лише у XX столітті, коли з'явилася потреба в оптимізації обчислень для машин.

Інфіксний запис є стандартною формою для запису математичних виразів в більшості сучасних культур і мов. Це та форма, до якої ми звикли з дитинства – операції виконуються між числами, і порядок їх виконання залежить від правил пріоритету операцій та дужок. Наприклад, вираз 3+4 простий і інтуїтивно зрозумілий. Однак у складніших виразах, таких як 2+3*4 операції множення виконуються перед додаванням, що вимагає знання пріоритету операцій.

Інфіксний запис використовується людством вже тисячі років. У стародавньому Єгипті, Месопотамії та Греції математики використовували подібні методи для запису арифметичних операцій, хоча конкретні знаки та правила не були такими, як зараз. Стародавні греки, зокрема Діофант Александрійський, використовували абстрактні символи для представлення математичних ідей, що можна розглядати як праформа інфіксного запису. Однак поняття пріоритету операцій і дужок не було ще сформульоване чітко, а математичні вирази були представлені досить простими способами, більше орієнтуючись на геометричні побудови [1].

З розвитком алгебри в середньовіччі та пізніше в епоху Ренесансу, інфіксний запис став набагато структурованішим. В середньовічних арабських рукописах вже можна було побачити використання букв для позначення змінних і складних арифметичних операцій. Але навіть у цей час математичні вирази записувалися на основі інфіксної форми, і для вирішення більш складних виразів використовувалися правила маніпулювання рівняннями [2].

Попри свою поширеність і зрозумілість, інфіксний запис має певні недоліки, особливо в контексті обчислень, виконуваних машинами. Для того, щоб комп'ютер чи калькулятор правильно обчислював вирази в інфіксній формі, йому потрібно вирішувати, в якому порядку виконувати операції, зважаючи на пріоритети. Проблема ускладнюється, коли вирази містять дужки, оскільки комп'ютер мусить аналізувати ці дужки, щоб коректно визначити порядок операцій.

На допомогу прийшов постфіксний запис, ідея якого була запропонована польським математиком Яном Лукасевичем в 1920-х роках. Лукасевич працював над теорією логіки і прагнув створити таку систему запису, де не буде потрібно використовувати дужки та правила пріоритету операцій. У своєму дослідженні він запропонував польську нотацію (тепер відому як зворотна польська нотація), в якій оператори слідують за операндами. Так, для виразу 3+4 постфіксна форма виглядатиме як 34+, а для більш складного виразу 3+4*5 – як 345*+ [3].

Чому це зручно? У постфіксному записі порядок операцій стає однозначним без необхідності в дужках або визначенні пріоритетів. Оскільки оператор завжди йде після операндів, програма, що обчислює вираз, може працювати за алгоритмом з використанням ***стека***, що значно спрощує обчислення.

Зворотна польська нотація знайшла своє застосування в комп'ютерних науках і обчислювальних системах. У 1970-х роках вона була впроваджена в мову програмування Forth, яка використовує постфіксний запис для зручності програмування та обчислень. Ідея

була також використана при розробці калькуляторів Hewlett-Packard (компанія, відома як HP, сьогодні – виробник ноутбуків, робочих станцій, систем друку), які використовують ***RPN*** як основний метод введення виразів [4].

Стек – це абстрактна структура даних, яка організована за принципом **LIFO** (Last In, First Out), тобто останній елемент, що був доданий, витягується першим. В контексті обчислення логічних виразів стек використовується для двох основних завдань: перетворення виразу інфіксної форми в постфіксну форму і для виконання власне обчислень вже в постфіксній формі.

Для перетворення математичних виразів інфіксної форми в постфіксну форму Едсгером Дейкстрою у 1960 р. був розроблений алгоритм сортувальної станції (англ. Shunting Yard Algorithm). Його метою було зробити ефективний метод для парсингу математичних виразів, який не вимагає великого обсягу пам'яті та забезпечує швидке виконання [5].

Алгоритм складається з наступних кроків:

1. Створюється стек «operators» для зберігання операторів і змінна рядкового типу «*postfix*».
2. Прохід по виразу зліва направо: для кожного символу «*token*» у виразі «*infix*».
3. Якщо символ – пробіл, він ігнорується.
4. Якщо символ є операндом, він одразу додається до рядка «*postfix*».
5. Обробка відкритих дужок: якщо зустрічається «(», вона додається до стека «*operators*», щоб позначити початок нового підвиразу.
6. Обробка закритих дужок: при зустрічі «)» витягуються всі оператори зі стека «operators» і додаються до рядка «*postfix*», доки не буде знайдена відповідна «(». Відкриваюча дужка «(» видаляється зі стека, але не додається в «postfix».
7. Обробка операторів: якщо символ є оператором (наприклад, $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$), функція перевіряє пріоритет поточного оператора порівняно з оператором в стеці. Усі оператори в стеці з більшим або рівним пріоритетом вилучаються і додаються в «*postfix*», щоб зберегти правильний порядок обчислень. Потім поточний оператор поміщається у стек «*operators*».
8. Після завершення проходу по виразу всі оператори, що залишилися в стеці, переносяться в рядок «*postfix*».

У цій роботі цей алгоритм є модифікованим для обробки підвиразів.

Основні етапи роботи застосунку. Загальний алгоритм роботи програми продемонстровано на рис. 1.

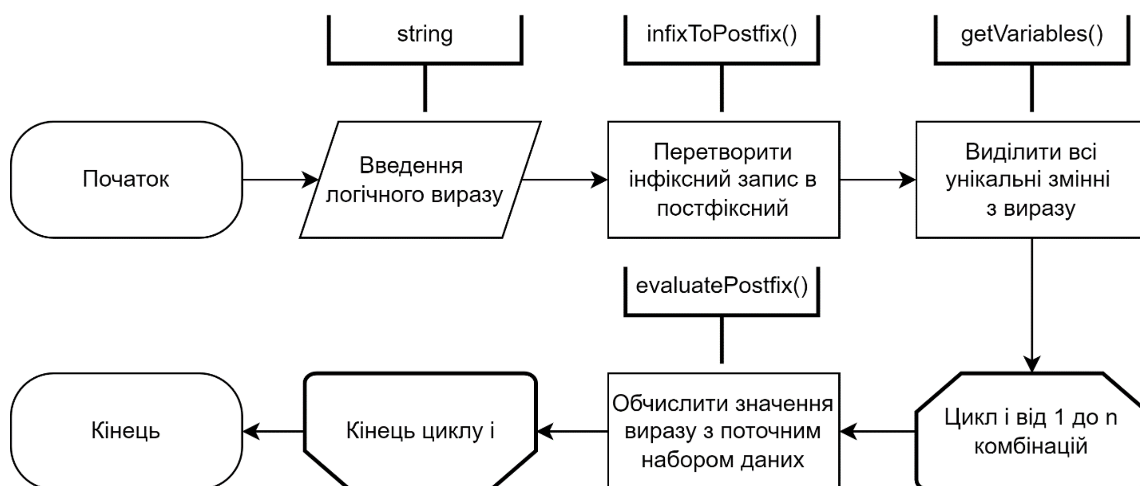

Рис. 1. Загальний алгоритм роботи програми

Він складається з наступних пунктів:

1. Програма зчитує введений користувачем логічний вираз у інфіксній формі.
2. Функція «***infixToPostfix()***» перетворює вираз у постфіксну форму запису, також розбиваючи вираз на підвирази.
3. Функція «***getVariables()***» виділяє всі унікальні змінні з логічного виразу.
4. Виводиться таблиця істинності для всіх можливих комбінацій значень змінних: функція «***evaluatePostfix()***» обчислює значення виразу з поточним набором даних.
5. Після проходження всіх рядків таблиці істинності визначається, чи є вираз тавтологією (всі значення істинні), суперечністю (всі значення хибні) або нейтральним (містить як істинні, так і хибні значення).

Алгоритм функції infixToPostfix(). Функція приймає такі параметри: посилання на рядок «*infix*», посилання на вектор рядків «*subexpressions*».

1. Початок – ініціалізація стека «operators», рядка «postfix», стека «*subexprStack*».
2. Зчитування вхідного символу (проходження по кожному символу рядка «infix»). Для кожного символу здійснюється перевірка:

- Якщо символ пробіл, пропустити та перейти до наступного символу.
- Якщо символ є змінною, додати символ до рядка «*postifx*».
- Якщо символ це відкриваюча дужка, додати символ до стека «*operators*».
- Якщо символ це закриваюча дужка, вилучається оператор зі стека «*operators*» і формується підвираз в стек підвиразів «*subexprStack*»:
  - Якщо оператор це заперечення, забирається один операнд з «*subexprStack*» і формується новий підвираз виду «a».
  - Якщо оператор є бінарним, забираються два операнди, формуючи підвираз виду «лівийОперанд оператор правийОперанд».
- Якщо символ є оператором, його необхідно обробити за допомогою кількох кроків для забезпечення правильного порядку виконання операцій:
  - Перевірити, чи стек «*operators*» не є порожнім і чи пріоритет оператора на вершині стека є більшим або рівним пріоритету поточного оператора.
  - Якщо виконується попередня умова, вийняти оператор зі стека «operators», додати його до рядка «postfix».
  - Далі оператор обробляється залежно від його типу (унарний, бінарний). Якщо це заперечення:
    1) Вийняти верхній елемент зі стека «*subexprStack*», що є операндом для операції заперечення.
    2) Створити підвираз, додавши «!» до операнда.
  - Якщо оператор бінарний:
    1) Вийняти два верхніх елементи зі стека «*subexprStack*» – перший елемент є правим операндом, другий – лівим.
    2) Зформувати підвираз, об’єднавши лівий операнд та правий операнд.
  - Додати підвираз до вектора рядків «*subexpressions*». Цей підвираз також треба зберегти назад у «*subexprStack*» для можливого подальшого використання.

  Поточний оператор додати до стека «*operators*».

3. У деяких випадках (наприклад, коли у вхідному виразі оператор стоїть між підвиразами, взятими у дужки) у стеці «*operators*» залишається один елемент. У такому випадку, треба додати його до рядка «*postfix*». Якщо оператор, що залишився, це заперечення, необхідно дістати операнд зі стека «*subexprStack*», створити підвираз додавши «!» до операнда. У випадку, якщо оператор є бінарним, вийняти два верхніх операнди зі стека «*subexprStack*», зформувати підвираз об’єднавши їх.
4. Повернути постфіксну форму виразу.

Алгоритм функції evaluatePostfix(). Після того, як логічний вираз переведено у постфіксну форму, наступним етапом є його обчислення для різних наборів значень змінних.

Цю задачу виконує функція «***evaluatePostfix()***», яка приймає наступні параметри: посилання на рядок з логічним виразом «*postfix*»; словник «values», який зіставляє кожну змінну «char» з її логічним значенням «bool»; вектор рядків «*subexpressions*»; словник «*intermediateResults*», у якому зберігаються результати підвиразів для кожного підвиразу з «*subexpressions*».

1. Оголошуються та ініціалізуються стек *operands*, змінні *subexprIndex*, *finalResult*.
2. Здійснюється перевірка кожного символу постфіксного виразу «*postfix*»:

● Якщо символ – змінна, її значення береться зі словника «*values*» і додається до стека «*operands*».

● Якщо символ – оператор:

- Унарний оператор (заперечення): витягується верхній елемент зі стека «*operands*», застосовується оператор заперечення, результат повертається в стек «*operands*».
- Бінарний оператор: витягуються два верхніх значення з «*operands*», до них застосовується оператор, і результат повертається в стек.

● Після кожної операції результат зберігається у словник «*intermediateResults*», використовуючи відповідний підвираз з «*subexpressions*», для чого функція відстежує індекс «*subexprIndex*», який збільшується після кожного збереження.

3. Коли всі символи оброблені, значення на вершині «*operands*» є остаточним результатом обчисленого виразу. Функція повертає його як *finalResult*.

Блок-схеми функцій «***infixToPostfix()***» та «***evaluatePostfix()***» зображені на рис. 3 та рис. 2 відповідно.

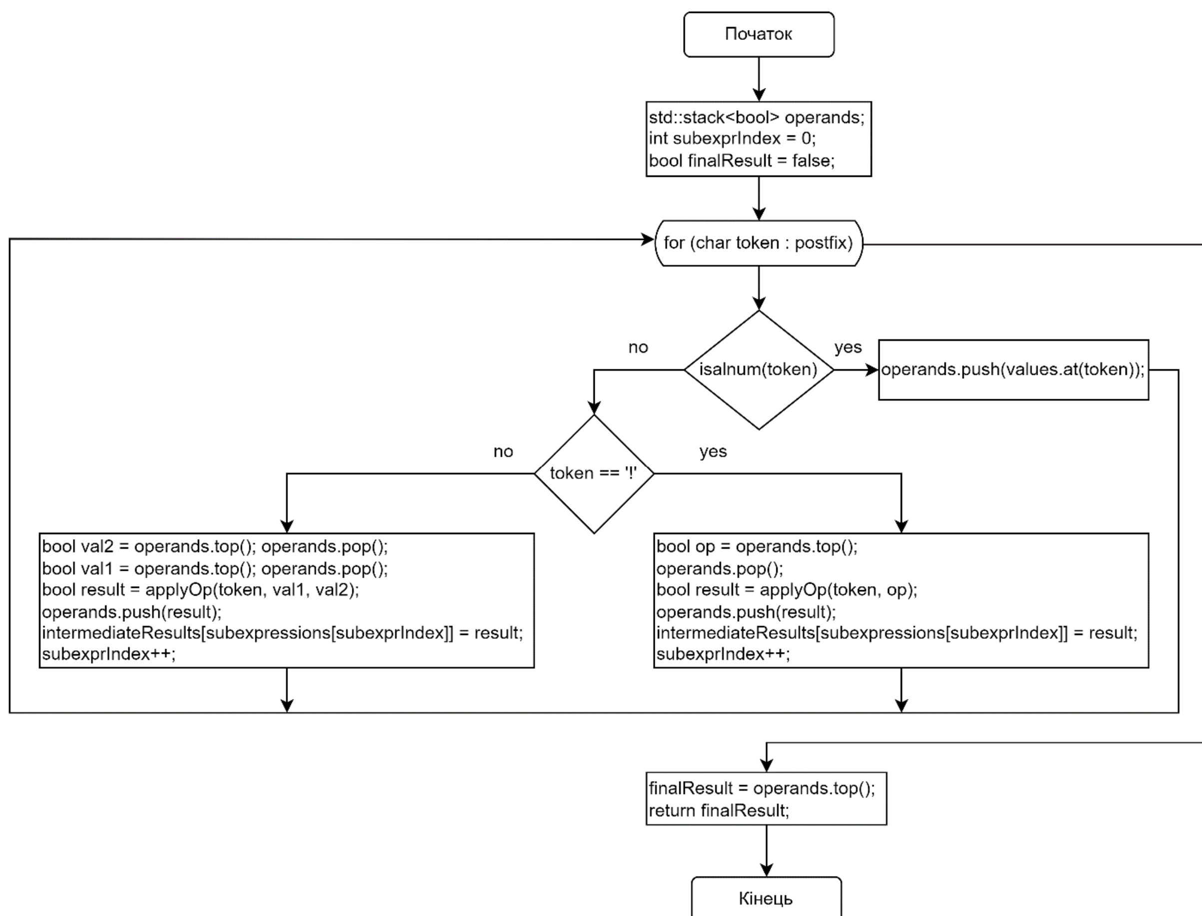

Рис. 2. Алгоритм роботи функції «***evaluatePostfix()***»

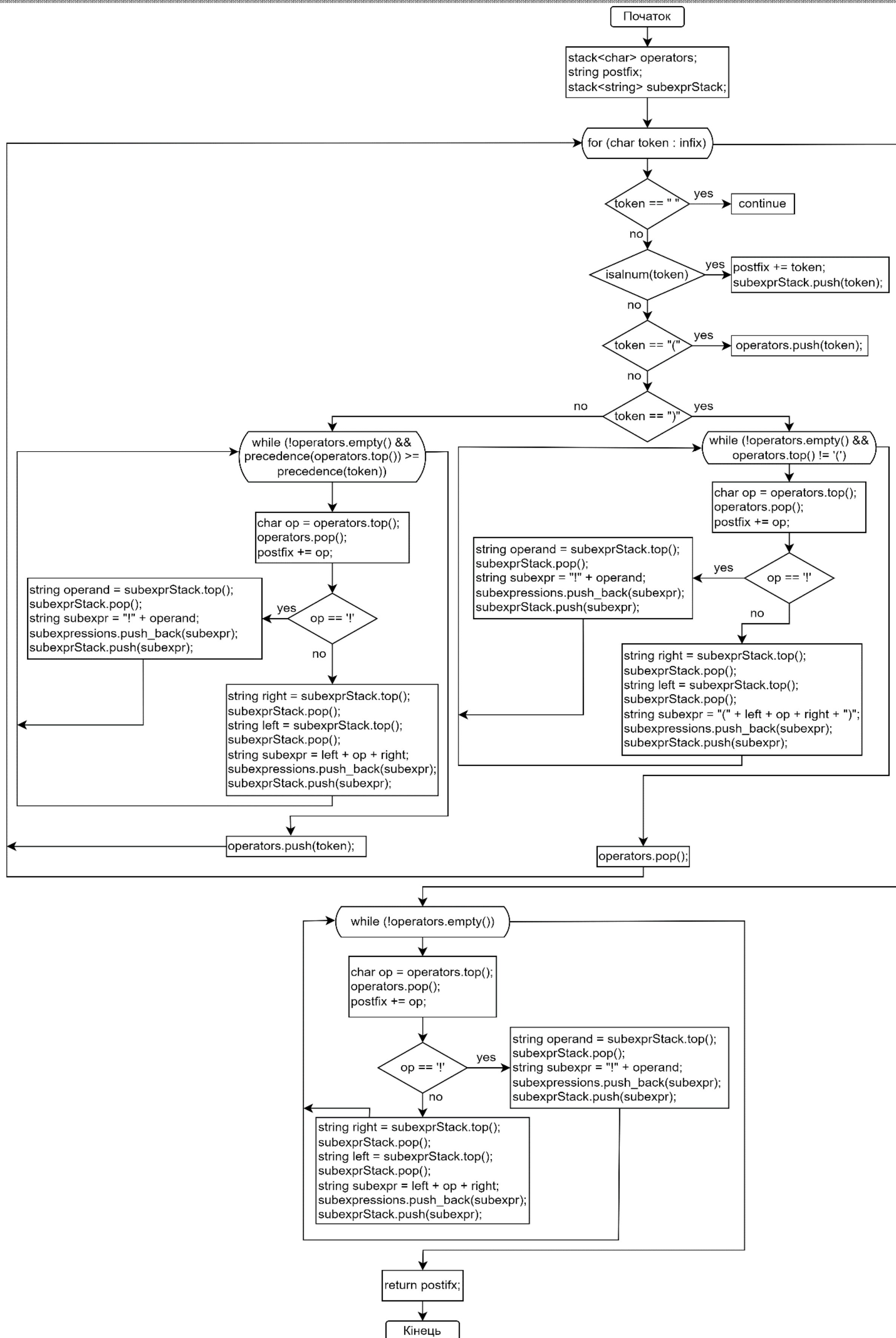

Рис. 3. Блок-схема алгоритму функції «***infixToPostfix()***»

Оцінка обчислювальної складності алгоритму. Асимптотична складність алгоритму – це оцінка того, як змінюється час виконання або обсяг пам'яті, необхідний для алгоритму, залежно від розміру вхідних даних. Зазвичай її записують у нотації «*O*», наприклад, $O(n)$, $O(n^2)$, $O(\log n)$ тощо. Основна ідея – оцінити максимальне зростання обчислень при збільшенні обсягу вхідних даних, і це допомагає передбачити ефективність алгоритму з великими наборами даних [6].

1. Функція ***getVariables()*** проходить через весь вхідний вираз і збирає унікальні змінні. Її складність – $O(n)$, де *n* – довжина виразу.
2. Функція ***infixToPostix()*** має складність $O(n)$, оскільки проходить по кожному символу інфіксного виразу, виконуючи операції з операторами та операндами.
3. Основна частина складності алгоритму пов'язана з побудовою таблиці істинності (функція ***buildTruthTable()***), оскільки тут відбувається ітерація по всіх можливих значеннях змінних і обчислення логічного виразу для кожного набору значень. Якщо кількість унікальних змінних дорівнює *k,* то загальна кількість рядків таблиці істинності – $O(2^k)$.
4. Функція ***evaluatePostfix()*** виконує обчислення постфіксного виразу для кожного рядка таблиці істинності. Її складність – $O(n)$, де *n* – довжина постфіксного виразу.

З огляду на вищесказане, можна сказати, що складність обчислення всіх рядків таблиці дорівнює $O(2^k \cdot n)$. Оскільки $O(n)$ у ***getVariables()*** та $O(n)$ у ***infixToPostix()*** є незначними у порівнянні з $O(2^k)$ у ***buildTruthTable()*** при великих значеннях *k*, можна ними знехтувати у фінальній оцінці. Загальна складність алгоритму буде:

$$O(n) + O(n) + O(2^k \cdot n) \approx O(2^k \cdot n).$$

Демонстрація роботи програми та її тестування. У ході тестування було обрано два вирази: простий вираз з малою кількістю змінних та складний вираз з більшою кількістю змінних. Простим виразом будемо вважати вираз з трьома змінними – $a \vee b \wedge c$, складним – $(a \wedge b) \vee c \Leftrightarrow d$. Оскільки очікувана складність алгоритму дорівнює $O(2^k \cdot n)$, де *k* – це кількість унікальних змінних, а *n* – довжина постфіксного виразу, очікується, що складний вираз буде виконуватися довше за простий у $\frac{2^4 \cdot 7}{2^3 \cdot 5} = 2{,}8$ разів.

Щоб перевірити, наскільки точно теоретичні прогнози відповідають дійсності, проведемо експеримент з побудови таблиць істинності для обраних виразів. Очікується, що цей показник буде близьким до розрахованого теоретичного значення 2,8 (див. рис. 4, рис. 5).

```
Введіть логічний вираз: a+b*c

a    b    c    b*c   a+b*c
-------------------------
0    0    0    0     0
0    0    1    0     0
0    1    0    0     0
0    1    1    1     1
1    0    0    0     1
1    0    1    0     1
1    1    0    0     1
1    1    1    1     1
Логічний вираз є нейтральним.
Логічний вираз виконуваний.
Час виконання алгоритму: 6 мс
```

Рис. 4. Результат виконання програми для виразу $a \vee b \wedge c$

```
Введіть логічний вираз: (a*b)+c=d

a    b    c    d    (a*b)   (a*b)+c   (a*b)+c=d
-----------------------------------------------
0    0    0    0    0       0         1
0    0    0    1    0       0         0
0    0    1    0    0       1         0
0    0    1    1    0       1         1
0    1    0    0    0       0         1
0    1    0    1    0       0         0
0    1    1    0    0       1         0
0    1    1    1    0       1         1
1    0    0    0    0       0         1
1    0    0    1    0       0         0
1    0    1    0    0       1         0
1    0    1    1    0       1         1
1    1    0    0    1       1         0
1    1    0    1    1       1         1
1    1    1    0    1       1         0
1    1    1    1    1       1         1
Логічний вираз є нейтральним.
Логічний вираз виконуваний.
Час виконання алгоритму: 17 мс
```

Рис. 5. Результат виконання програми для виразу $(a \wedge b) \vee c \Leftrightarrow d$

Отримане експериментальне значення $\frac{17}{6}=2,83$ майже ідентичне теоретичному прогнозу 2,8, що підтверджує правильність оцінки складності алгоритму та його поведінку при збільшенні унікальних змінних.

Для наочності було проведено тестування на виразах, що містять від 1 до 12 унікальних змінних. Результати підтвердили, що збільшення кількості змінних призводить до експоненціального зростання обчислювальних витрат. Це чітко відображено на діаграмі нижче, яка ілюструє зміну часу виконання залежно від кількості унікальних змінних (рис. 6).

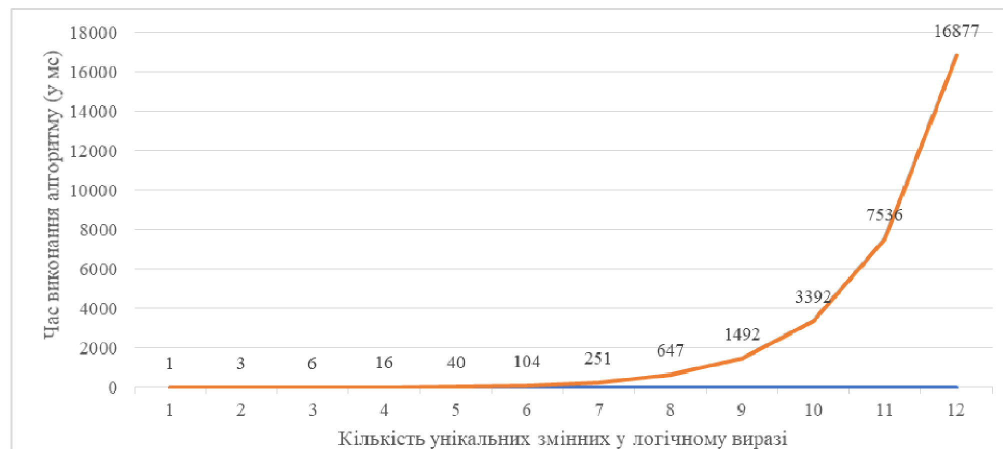

Рис. 6. Залежність часу виконання від кількості унікальних змінних у виразі

Висновки. Під час виконання наукової роботи було розроблено програмний застосунок для аналізу логічних виразів. Він базується на алгоритмі сортувальної станції, який у межах цього дослідження було значно модифіковано. На відміну від класичного алгоритму, що призначений для перетворення математичних виразів у інфіксному записі в вирази постфіксного запису, у цій роботі алгоритм було розширено для додаткового розбиття логічного виразу на підвирази. Це дозволяє виділяти проміжні результати на кожному етапі обчислень, що є критично важливим для візуалізації складного логічного виразу та надання користувачеві прозорих пояснень і принципів побудови таблиці істинності.

Завдяки цьому застосунок не лише обчислює підсумкове значення виразу, а й забезпечує користувача можливістю покрокового розуміння логічного процесу, що відбувається всередині. Такий підхід робить його особливо корисним для тих, хто починає вивчати алгебру висловлень, адже допомагає краще засвоїти концепти булевої логіки, розуміючи складні комбінації операцій та їхні пріоритети. Відображення фінальних результатів для всіх можливих комбінацій значень виразу дозволяє визначити його тип: тавтологія, суперечність, нейтральний. Також можна значно розширити функціональні можливості цього застосунку. Зокрема, додавання обчислень досконалої диз'юнктивної та кон'юнктивної нормальних форм (ДДНФ і ДКНФ) могло б надати користувачам глибший інструмент для аналізу логічних виразів. Також можливість інтеграції додаткових логічних операцій (виключне *або*, штрих Шефера, стрілка Пірса, виключне *або* з інверсією) і функцій автоматичного спрощення виразів зробить програму ще більш універсальною.

1. Muscat J. Diophantus of Alexandria Arithmetica. *L-UniversitÃ ta' Malta*. URL: https://staff.um.edu.mt/jmus1/Diophantus.pdf (дата звернення: 07.11.2024). **2.** J. Katz V. A History of Mathematics: An Introduction. Addison-Wesley, 2010. 879 с. **3.** Reverse Polish Notation. *Wolfram MathWorld* : The Web's Most Extensive Mathematics Resource. URL: https://mathworld.wolfram.com /ReversePolishNotation.html (дата звернення: 07.11.2024). **4.** HP 9100A/B. The Museum of HP Calculators. URL: https://www.hpmuseum.org/hp9100.htm (дата звернення: 07.11.2024). **5.** The Shunting Yard Algorithm. URL: https://mathcenter.oxford.emory.edu/site/cs171/shuntingYardAlgorithm/ (дата звернення: 07.11.2024). **6.** What is Big O Notation Explained : Space and Time Complexity. *freeCodeCamp.org.* URL: https://www.freecodecamp.org/news/big-o-notation-why-it-matters-and-why-it-doesnt-1674cfa8a23c/ (дата звернення: 12.11.2024).