Зміст

Вступ 3

§ 1. Основні положення 4

§ 2. Перевірка правильності виразів 13

§ 3. Оцінка складності алгоритмів 16

§ 4. Оптимізація програм 19

§ 5. Результати і висновки 23

Література 24

Додаток 1. Програмна реалізація алгоритмів POSTFIX та ІТР 25

Додаток 2. Програмна реалізація простого компілятора виразів 29

**Вступ.**

 В наш час електронно-обчислювальні машини та інформаційні технології стали невід’ємною частиною промисловості та міцно ввійшли в побут простих громадян. І хоча сучасні комп’ютери здатні виконувати набагато складніші задачі ніж могли їхні пращури з 60-х років двадцятого століття, все рівно математична обробка інформації була, є і в майбутньому буде основою їх роботи. Кожен найсучасніший комп’ютер - чи він застосовується для обробки зображень, чи для проектування автомобілів, чи виступає в ролі ігрового автомату - все рівно, в першу чергу, він є обчислювальною машиною. Арифметичні та логічні вирази є невід’ємною частиною практично всіх, навіть вузькоспеціалізованих, комп’ютерних програм і тому потрібно мати у наявності алгоритми, які розпізнають і обчислюють їх якомога швидше і ефективніше. Тема обчислення виразів проходить через більшу частину програмування; з нею пов’язані синтаксис і семантика алгоритмічних мов, компіляція, формальні мови, структури даних, логіка, рекурсія і обчислювальна складність. Тому наукові розробки даного напрямку є важливими не тільки зараз - вони збережуть свою актуальність і в майбутньому.

 Яка ж мета ставилася при написанні даної роботи ? Серед основних цілей можна назвати слідуючі:

* систематизувати та узагальнити існуючі на даний час відомості з теорії аналізу та обчислення виразів;
* виявити корисні з практичної точки зору критерії оптимальності програм та розглянути ефективність різних алгоритмів обчислень відносно цих критеріїв;
* розглянувши сильні та слабкі сторони кожного алгоритму визначити клас задач, на яких цей алгоритм має переваги перед іншими;
* намітити основні способи оптимізації.

Оскільки критерієм істини є практика, то не менш важливим моментом даної дипломної роботи є практична реалізація алгоритмів обчислення виразів на одній з мов програмування і перевірка на практиці ефективності методів оптимізації.

**§1. Основні положення.**

 Існує принаймні три різні способи означення арифметичних виразів. Підручники з програмування для початківців, як правило, подають їх на прикладах. Цілком логічна основа цього підходу полягає в тому, що можна навчитися писати правильні вирази, переглянувши достатню кількість прикладів. Це у великій мірі схоже на навчання деякої не рідної для людини іноземної мови. Так як було відмічено, що більшість програмістів використовують в своїх програмах доволі прості вирази, то цей підхід у більшості випадків виявлявся цілком прийнятним.

 Більш формальний підхід полягає в тому, що синтаксис і семантику арифметичних і логічних виразів задають за допомогою контекстно-вільних правил перетворення, як це зроблено в описі Алголу. Проміжний підхід, зберігаючи деяку долю математичної точності визначень і в значній мірі розрахований на інтуїцію, полягає у визначенні цих виразів індуктивно або рекурсивно.

 **Арифметичний вираз** індуктивно визначається наступним чином:

1. Довільна змінна є арифметичним виразом.
2. Довільна константа є арифметичним виразом.
3. Довільне посилання на арифметичну функцію є арифметичним виразом.
4. Якщо Х – арифметичний вираз, то (Х) – теж арифметичний вираз.
5. Якщо Х і У обидва є арифметичними виразами, то арифметичними виразами також будуть (Х+У), (Х-У), (Х\*У), (Х/У), (Х^У).
6. Жоден об’єкт не є арифметичним виразом, якщо той факт, що він є арифметичним виразом не слідує з скінченого числа застосувань правил 1 – 5.

Це означення дає набір ефективних правил, придатних для побудови довільного арифметичного виразу в термінах змінних, констант, посилань на функції і операцій +, -, \*, /, ^.

Для спрощення подальшого викладу, ми до кінця даного параграфу накладемо наступні обмеження на арифметичні вирази:

* всі змінні та константи позначаються великими буквами латинського алфавіту;
* у виразах зустрічаються тільки бінарні алгебраїчні операції +, -, \*, /, ^;
* у виразах немає викликів алгебраїчних функцій.

Ці обмеження дають змогу побудувати прості і очевидні алгоритми маніпуляцій з виразами. В наступних параграфах ці обмеження буде знято, а всі алгоритми узагальнено.

**Твердження 1.**

У правильно побудованому арифметичному виразі, що містить лише бінарні алгебраїчні операції, кількість операндів на 1 більше кількості операцій.

Доведення. Правильно побудований вираз найменшої довжини має вигляд А або (А), де А - деякий операнд. В цьому виразі кількість операндів 1, кількість операцій 0, отже твердження виконується. Найменший вираз, що містить операцію має вигляд (А операція1 В). Тут кількість операндів 2, а операцій - 1, отже знову виконується твердження.

 Припустимо, що твердження виконується для всіх правильно побудованих виразів довжини не більше k. Згідно означення, отримати арифметичний вираз довжини k+1 можна тільки з двох виразів довжини не більше k, і тільки в такий спосіб - (ВИРАЗ1 ОПЕРАЦІЯ1 ВИРАЗ2). Якщо ВИРАЗ1 містить n операцій і n+1 операндів, ВИРАЗ2 - m операцій і m+1 операндів, то їх об’єднання (ВИРАЗ1 ОПЕРАЦІЯ1 ВИРАЗ2), яке є правильно побудованим арифметичним виразом, відповідно буде містити n+m+1 операцій (додалася ще одна операція - ОПЕРАЦІЯ1) і n+1+m+1 операндів. Отже твердження виконується. Згідно принципу математичної індукції воно буде виконуватися для виразів довільної довжини.

Розглянемо вираз

 (((А-В)\*С)+(D/(E^F)))

Якщо це правильний арифметичний вираз, то повинна існувати можливість побудови його за допомогою правил 1- 5. Неважко переконатися, що така можливість дійсно існує. Але можливе і інше представлення цього арифметичного виразу - за допомогою кореневого бінарного дерева, як показано на малюнку 1.

Відмітимо, що всі кінцеві вершини цього дерева відповідають змінним або операндам, а всі внутрішні вершини відповідають арифметичним операціям. Дерево є бінарним (тобто кожна вершина має або дві дочірні вершини, або не має їх взагалі), оскільки ми домовилися використовувати у виразах лише бінарні (двохмісні) алгебраїчні операції. Кожному правильно записаному арифметичному виразу однозначно ставиться у відповідність дерево виразу. Якщо задано таке дерево, то вираз однозначно обчислюється при відомих значеннях змінних. Потрібно також відмітити, що деякі з проміжних обчислень, необхідних для отримання значення всього виразу, можна провести паралельно. Наприклад віднімання А-В можна виконувати паралельно з піднесенням до степеня E^F.

 Мал. 1

 **Логічні вирази** можна означити так само, як і арифметичні. А саме:

1. Будь яка логічна константа є логічним виразом.
2. Будь яка логічна змінна є логічним виразом.
3. Довільний предикат при заданих значеннях аргументів є логічним виразом.
4. Якщо Х – логічний вираз, то (Х) – теж логічний вираз.
5. Якщо Х і У – обидва є логічними виразами, то логічними виразами також будуть (X AND У), (X OR У) і (NOT X).
6. Жоден об’єкт не є логічним виразом, якщо той факт, що він є логічним виразом не слідує з скінченого числа застосувань правил 1 – 5.

Приклад правильно записаного логічного виразу

 (((NOT A) AND B) OR (C AND (D OR E)))

Аналізуючи наведені вище приклади, можна відмітити, що означення арифметичних та логічних виразів містять, напевно, зайві дужки. Можна дати означення, що не потребують такої кількості дужок, але такі означення неминуче приводять до двозначності. Наприклад, вирази

 A + B / C або NOT A AND B

повинні обчислюватися з використанням правил пріоритету, які встановлюють, що ділення ( / ) має більший пріоритет, ніж додавання ( + ), а NOT має більший пріоритет, ніж AND. З врахуванням цих пріоритетів, вираз A+B/C еквівалентний виразу (A+(B/C)), а NOT A AND B еквівалентний ((NOT A) AND B). В наших означеннях ми ввели дужки для більшої математичної строгості.

 Необхідно відмітити, що три різні способи проходження дерева (мал.1) приводять до трьох різних способів запису відповідних виразів в вигляді лінійної послідовності символів. Розглянемо дерево простого арифметичного виразу (А+В), наведене на малюнку 2. Якщо ми почнемо з кореня (верхньої точки) (В) і запишемо +, потім перейдемо до лівої (Л) нижньої вершини і допишемо А, а далі повернемося знову в корінь і спустимося вправо (П) і напишемо В, то такий порядок проходу вершин дерева є **прямим**.

 В

 Мал. 2

Л П

Послідовність записаних символів, +АВ називається **префіксною** формою арифметичного виразу, оскільки знак операції (+) передує операндам А і В. Якщо якась з нижніх вершин А або В сама є кореневою вершиною, тобто містить не операнд а операцію, то для обчислення загального виразу ми повинні обчислити цей підвираз, і тому до цієї вершини рекурсивно застосовуємо те ж саме правило (вважаючи цю нижню вершину кореневою): записуємо операцію в вершині (В), лівий операнд (Л), правий операнд (П). В результаті ми отримаємо лінійний бездужковий запис виразу по якому можна однозначно відтворити вихідне бінарне дерево. Вкажемо правило яке дозволяє це здійснити.

Читаємо префіксний запис виразу посимвольно (він обов’язково починається знаком операції), записуємо знак операції у вершину, зчитуємо два наступні символи і якщо вони є операндами (змінними чи константами) то для даної вершини будуємо дві дочірні вершини, записуючи в ліву перший операнд, а в праву - другий; якщо якийсь з двох символів є не операндом а операцією, то для цієї дочірньої вершини рекурсивно застосовуємо це ж саме правило.

Префіксна форма запису є вдалою в тому розумінні, що нам не потрібно використовувати дужки і вводити пріоритет операцій, хоч ми все рівно можемо однозначно обчислити вираз.

 Більш звична для сприйняття форма арифметичного виразу А+В називається **інфіксною** формою, вона отримується при проходженні дерева в зворотному порядку, який можна позначити (Л)(В)(П). На жаль лінійна послідовність символів, яка отримується при цьому не дає змоги однозначно відтворити початкове дерево виразу, тому при обчисленні виразу в інфіксній формі можливі двозначності. По суті, розглянуте на початку параграфу означення арифметичних виразів вводить вирази саме в інфіксній формі, і для подолання проблеми неоднозначності тут застосовуються дужки.

**Постфіксна** форма арифметичного виразу – в якій знак операції слідує за операндами, наприклад АВ+, - отримується при кінцевому порядку (Л)(П)(В) проходження вершин дерева. Ця форма запису найчастіше використовується для внутрішнього представлення виразів в комп’ютерних програмах, оскільки вона, так як і інфіксна форма, дозволяє повністю однозначно обчислити значення виразу, не використовуючи дужок і пріоритет операцій, а також є зручною для реалізації.

 Для прикладу можна розглянути дерево на малюнку 1. Щоб пройти це дерево в прямому порядку потрібно почати з першої кореневої вершини (відміченої +), пройти в прямому порядку спочатку ліве піддерево з коренем ( \* ), а потім праве з коренем ( / ). Для проходження кожного піддерева знову застосовуємо те ж саме правило – корінь, ліве піддерево, праве піддерево. Таким чином, проходження всього дерева в прямому порядку породжує послідовність

 + \* - A B C / D ^ E F (префіксна)

Проходження цього дерева в оберненому і кінцевому порядках відповідно породжує наступні послідовності

A – B \* C + D / E ^ F (інфіксна)

 A B – C \* D E F ^ / + (постфіксна)

 Потрібно відмітити, що в усіх трьох виразах порядок входження змінних співпадає, міняється лише порядок знаків операцій.

 Все сказане вище в однаковій мірі справедливе як для арифметичних, так і для логічних виразів. Проте розгляд логічних виразів ми відкладемо до введення поняття лексичної згортки, яка дає можливість повністю абстрагуватися від типу виразу, що розглядається.

Опишемо алгоритм [1], який дозволяє обчислити значення арифметичного виразу в постфіксній формі.

 **Алгоритм POSTFIX.**

 Обчислити арифметичний вираз в постфіксній формі; вираз задано послідовністю S(1) S(2) . . . S(N), N ≥ 1, де S(J) – або буква (операнд, змінна, константа), або один зі знаків +, -, \*, /, ^ (тобто двохмісна арифметична операція)

Крок 0. j:=0

Крок 1. Для і від 1 до N виконувати [крок 2]. \\ переглядаємо вираз

Стоп. \\ значення виразу буде на вершині стеку

Крок 2. Якщо S(i) – операнд

 тоді j:=j+1; СТЕК(j):=S(i)

 інакше Т1:=СТЕК(j); Т2:=СТЕК(j-1);

Т3:=Т2 S(i) Т1; \\ виконуємо операцію

j:=j-1; СТЕК(j):=Т3; \\ результат заносимо в стек

 Доведення правильності алгоритму POSTFIX можна отримати за індукцією по довжині N постфіксного виразу S(1) S(2) . . . S(N). Таке доведення залежить від індуктивного означення арифметичного виразу і від процесу трансляції арифметичного виразу з інфіксної форми в постфіксну.

 Алгоритм POSTFIX правильно працює на постфіксних виразах довжини N=1 та N=3 (вважаємо, що не існує постфіксних виразів довжини N=2). Це безпосередньо перевіряється ручним обчисленням. Тому припустимо, що алгоритм POSTFIX правильно обчислює всі постфіксні вирази довжини не більше N.

 Розглянемо довільний постфіксний вираз S(1) S(2) . . . S(N+1) довжини N+1. Нехай k – найменше ціле число, таке, що S(k) – операція. Тоді очевидно, що ця операція повинна бути виконана над S(k-1) і S(k-2), які є операндами. Взагалі кажучи, з того, що S(i) – операція, не обов’язково слідує, що S(i-1) та S(i-2) - операнди. Але через те, що S(k) – перша операція в послідовності, S(k-1) та S(k-2) повинні бути операндами. На кроці 2 алгоритм POSTFIX забирає зі стеку S(k-1) та S(k-2), обчислює T = S(k-1) S(k) S(k-2) і поміщає Т на стек так, ніби він є ще одним операндом.

 В цей момент конфігурація стеку точно така, як була б у випадку, якщо б початковий постфіксний вираз був більш коротким: S(1) . . . S(k-3) T S(k+1) . . . S(N). Але оскільки значення такого зміненого виразу (можна показати, що це правильний постфіксний вираз) рівне значенню початкового виразу і оскільки за припущенням індукції алгоритм POSTFIX правильно працює на всіх виразах довжини не більше N, то можемо зробити висновок, що алгоритм POSTFIX правильно обчислює вихідний вираз.

 Постфіксний вираз можна обчислити за допомогою стеку за O(N) операцій. Це випливає з того, що при перегляді кожного з N символів S(1), S(2), . . . , S(N) виконується не більше ніж деяке постійне число операцій.

 Потрібно відмітити деякі властивості алгоритму POSTFIX:

1. Алгоритм вважає, що вхідна послідовність є правильним постфіксним виразом; відсутні тести, що гарантують правильність вхідної послідовності.
2. Алгоритм розроблено для обробки виразів, в яких операнди не повинні складатися з кількох букв або цифр.
3. Недопустимою є поява у виразі числових констант чи звернень до функцій.
4. У виразі не повинно бути одномісних операцій, наприклад -А або +А-В.
5. Алгоритм не ініціалізує стек нулями і не заповнює його нулями після того, як значення були використані.

Має місце наступне твердження:

**Твердження 2.** Арифметичний вираз, записаний у префіксній формі, може бути обчислений алгоритмом POSTFIX, якщо цей вираз переписати ззаду-наперед, а у алгоритмі виконувати операції, міняючи місцями операнди відносно знаку операції (для некомутативних операцій).

Доведення даного твердження можна отримати за індукцією по кількості вершин дерева виразу. Якщо дерево містить одну вершину (операнд), то нічого обчислювати взагалі не потрібно і твердження виконується. Якщо дерево містить три вершини

то префіксна форма запису виразу буде " –AB ". Якщо її переписати ззаду-наперед " ВА– " і застосувати модифікований алгоритм POSTFIX (міняючи місцями операнди відносно операції), то отримаємо правильний результат (значення А-В), отже твердження виконується.

 Припустимо, що твердження виконується для всіх виразів, дерево яких містить не більше n вершин. Тоді вираз з n+1 - вершинним деревом, згідно означення арифметичного виразу, складається з двох менших підвиразів (піддерев), і має вигляд "Оперція1 Вираз1 Вираз2" у префіксній формі. Переписавши його ззаду-наперед "Вираз2 Вираз1 Операція1" і припускаючи, що Вираз1 та Вираз2 обчислюються модифікованим алгоритмом POSTFIX правильно, ми отримаємо на виході правильний результат, тобто "Значення виразу1" Операція1 "Значення виразу2". Твердження доведено.

Отже ми бачимо, що арифметичний вираз дуже легко обчислюється, як тільки його переведено в постфіксну або префіксну форму. Але в більшості мов програмування вимагається запис арифметичних виразів в інфіксній формі. Наступний алгоритм розроблено для переведення виразів з інфіксної форми в постфіксну [1]. Він базується на наступному правилі пріоритетів:

|  |  |
| --- | --- |
| Символ | Пріоритет |
| ( , ) | 0 |
| + , - | 1 |
| \* , / | 2 |
| ^ | 3 |

Таблиця 1.

 Хоч насправді пріоритет дужок є найвищим, ми для даного алгоритму штучно знижуємо його, для простішої обробки послідовності у стекові. Але хоч і неявно, а все ж верховенство пріоритету дужок проявляє себе в тому, що для їх обробки використовуються окремі процедури.

 Алгоритм працює, читаючи символи інфіксного виразу зліва направо. Всі операнди (змінні) попадають на вхід по мірі прочитування виразу; інші символи поміщаються на стек і або видаляються, або поступають на вихід пізніше в відповідності з наведеним вище правилом пріоритетів.

 **Алгоритм ITP** (інфіксна в постфіксну).

 Перевести інфіксний вираз S(1) S(2) . . . S(N), N≥1, в постфіксну форму. S(i) - або буква (тобто операнд, або змінна), ліва або права дужка, або знак операції (тобто один із символів +, -, \*, /, ^ ). Алгоритм використовує стек і пріоритетну функцію Р, згідно якої "ε", "(", ")" мають пріоритет 0, Р(+)=Р(-)=1, Р(\*)=Р(/)=2, Р(^)=3.

Крок 0: СТЕК(1):=ε; j:=1;

Крок 1: Для і від 1 до N виконувати: { Кроки 2 - 5 }

Крок 2: Якщо S(i) - буква то { Вивести S(i) }

Крок 3: Якщо S(i)="(" то { j:=j+1; CTEK(j):=S(i) }

Крок 4: Якщо S(i)=")" то

 {

 Поки СТЕК(j)≠"(" виконувати:{ Вивести СТЕК(j); j:=j-1; }

 j:=j-1;

 }

Крок 5: Якщо S(i) - знак операції то

{

Поки Р(S(i)) ≤ P(СТЕК(j)) виконувати: { Вивести СТЕК(j); j:=j-1; }

 j:=j+1; СТЕК(j):=S(i);

 }

Крок 6: Поки СТЕК(j)≠ε виконувати: { Вивести СТЕК(j); j:=j-1; }

 Стоп.

 Складність алгоритму ITP виявляється О(N). Це випливає з таких властивостей алгоритму:

1. Один прохід виконується над N символами послідовності.
2. На стек може попасти найбільше N/2 символів (тобто операцій).
3. Всі символи, що залишаються на стекові, можуть бути виведені після закінчення перегляду послідовності не більше ніж за O(N) операцій.
4. Для обробки довільного символу в послідовності потрібно не більше постійного числа операцій.

Неважко переконатися в тому факті, що алгоритм ІТР можна використати і для переводу виразів з інфіксної форми в префіксну. Для цього потрібно:

* вираз в інфіксній формі переписати посимвольно ззаду-наперед;
* застосувати до цієї послідовності алгоритм ІТР;
* результат знову переписати ззаду-наперед.

Отримана послідовність якраз і буде префіксною формою запису виразу.

Розглянувши базові алгоритми роботи з виразами, звернемо увагу на іншу, більш прагматичну сторону обчислень. Зокрема потрібно з’ясувати в складі якого програмного пакету відбувається обчислення виразу, щоб мати змогу обрати більш ефективний метод. Розрізняють два принципово різні підходи до обчислення виразів - інтерпретативний і компілятивний.

**Компілятивний** підхід полягає в тому, що для даного виразу генерується окрема програма в машинних кодах, яка обчислює вираз при заданих значеннях змінних. Особливістю цього підходу є те, що спочатку затрачається багато машинних ресурсів для перевірки виразу, переведення його в зручну форму, генерації програми на мові низького рівня, але зате потім згенерована програма, вже будучи цілком самостійною, обчислює значення виразу з великою швидкістю. Такий підхід є просто незамінним у тому випадку, коли потрібно обчислити серію значень деякої формули на різних наборах вхідних даних.

**Інтерпретативний** підхід має на меті одноразове обчислення виразу. Процес обчислення не такий швидкий як у відкомпільованої програми, але зате немає потреби витрачати багато часу і ресурсів на компіляцію.

Не можна однозначно сказати який з цих підходів є кращим. Кожен з них має свою сферу застосування. У таких відомих математичних пакетах як MathCad, Mathematica, Derive та інших застосовують інтерпретативний підхід, оскільки користувачів найчастіше цікавить отримання одиничного результату. Зате при обчисленнях в пакетах математичного моделювання кожна зайва операція в циклі обробки моделі значно збільшує час, необхідний для розрахунків, тому тут найчастіше звертаються до компіляції виразів.

**§2. Перевірка правильності виразів.**

Перед тим, як почати обчислювати значення виразу, логічним є спочатку перевірити його на правильність. Взагалі кажучи, задача зчитування довільної послідовності символів і прийняття рішення про те, що вона являє собою правильний арифметичний чи логічний вираз, є нетривіальною. Її вирішення торкається теорії формальних мов і складає окрему частину теорії компіляції.

Цілком очевидно, що ті обмеження, які ми накладали раніше на арифметичні вирази є дуже жорсткими і для практичних обчислень їх потрібно зняти. Тому надалі вважаємо наступне:

* змінні та константи в арифметичному виразі можуть позначатися не тільки великими буквами латинського алфавіту, але і довільним ідентифікатором (**Ідентифікатор** - послідовність букв і цифр, яка починається з букви і не містить пробілів всередині);
* допустимим є використання числових констант, записаних у вигляді десяткового дробу;
* допустимими є посилання на N-місні арифметичні функції, якщо це посилання має вигляд ІДЕНТИФІКАТОР(ПАРАМЕТР\_1, . . . , ПАРАМЕТР\_N), де N - натуральне число, ІДЕНТИФІКАТОР - ідентифікатор, що однозначно визначає арифметичну функцію, ПАРАМЕТР\_1 . . . ПАРАМЕТР\_N - правильні арифметичні вирази;
* допускається застосування правил пріоритету операцій (табл. 1) замість використання дужок.

Очевидно також те, що для всіх ідентифікаторів, які зустрічаються у виразі, нам потрібно мати таблицю значень. Ця таблиця ставить у відповідність всім змінним і константам їх числове значення, а для арифметичних функцій визначає адресу в пам’яті, де знаходиться процедура обчислення значення даної функції.

Серед методів аналізу виразів можна виділити три основні: лексичний, синтаксичний і семантичний.

**Лексичний аналіз** (від грецького lexikos - той, що відноситься до слова) ставить на меті визначити правильність запису та використання символів, знаків, слів, ідентифікаторів - так званих **лексем** мови. У випадку арифметичних виразів лексичний аналіз має визначити:

* правильність запису ідентифікаторів (чи всі функції та змінні названо правильно);
* правильність запису числових констант ( "123,125.45" - неправильний запис);
* відсутність сторонніх символів (чи, наприклад, не вжито десь фігурні дужки замість круглих).

На практиці разом з лексичним аналізом виконують **лексичну згортку** виразу. Ця процедура спрямована на спрощення подальших маніпуляцій з виразом і суть її полягає в тому, що всі символьні ідентифікатори та числові константи у виразі замінюються спецсимволами-посиланнями на таблицю значень лексем. Тобто замість символьного запису назви змінної чи послідовності символів-цифр, які позначають число, ми одразу будемо мати адресу комірки пам’яті, де зберігається значення (змінної чи константи), а замість назви функції - адресу початку процедури, яка виконує обчислення даної функції.

Приклад лексичної згортки:

вираз Value(12) + 136 / Max

згортається до A(B)+C/D

де А -> адреса функції Value

 В -> 12

 C -> 136

 D -> значення змінної/константи Мах.

 Після виконання згортки вирази набувають значно зручнішого для подальшої роботи вигляду. Саме тому алгоритми, розглянуті в першому параграфі, не потребують модифікації для роботи з багатосимвольними ідентифікаторами - ця задача повинна вирішуватися ще на етапі лексичного аналізу.

 Опишемо правило, згідно якого виконується лексичний аналіз і згортка. Переглядаємо вираз посимвольно і розрізняємо такі випадки:

* якщо черговий символ є одиничним - дужка, знак операції, кома (допускається для розділення параметрів функції) - то його записуємо у вихідну послідовність;
* якщо символ є буквою, то ми натрапили на початок ідентифікатора, тому переглядаємо вираз далі до тих пір, доки не зустрінемо термінальний символ (під **термінальним символом** будемо розуміти такий, який не може належати даному типу лексем; наприклад ідентифікатору не може належати пробіл, кома, знак операції, числовій константі не може належати буква і т.д.), після чого прочитану послідовність символів шукаємо в таблиці значень ідентифікаторів; якщо не знайшли, то повідомляємо про помилку, а якщо знайшли - то записуємо у вихідну послідовність спецсимвол згортки, а у нову таблицю лексем добавляємо запис про значення цього спецсимволу;
* якщо символ є цифрою, то ми знаходимося на початку числової константи, і тому читаємо послідовність до появи термінального символу (яким може бути навіть повторна поява крапки), після цього записуємо у вихідну послідовність спецсимвол, а у таблицю лексем - значення числової константи для даного спецсимвола.

Продовжувати перегляд початкового виразу потрібно з того символу, який був термінальним для попередньої лексеми, або слідує за розглянутим одиничним символом.

Для прискорення перегляду таблиці допустимих лексем (а у математичних пакетах кількість реалізованих арифметичних функцій вимірюється сотнями) потрібно цю таблицю відсортувати в лексикографічному порядку і виконувати пошук методом дихотомії.

**Синтаксичний аналіз** (від грецького syntaxis - стрій, порядок) акцентує увагу на типах зв’язків між лексемами, на способах об’єднання лексем між собою, конструювання складних виразів із простіших. У нашому випадку синтаксичний аналіз відповідає за наступне:

* правильність розкладення дужок (приклад неправильного розкладення ")А+В(" );
* правильність взаємного розміщення знаків операцій і операндів (в інфіксній формі представлення виразу запис "/АВ+С" є неправильним );
* правильність виклику арифметичних функцій (чи є потрібна кількість аргументів, чи записані вони через кому, чи немає зайвих).

**Семантичний аналіз** (від грецького semantikos - означаючий) покликаний дослідити зміст та сутність лексем і елементів виразу. Зокрема він відповідає за виявлення явних спроб ділення на нуль, взяття квадратного кореня із від’ємного числа і тому подібних випадків.

 Після того, як до виразу було застосовано всі описані типи аналізів і виконано лексичну згортку, він може бути оброблений раніше розглянутими алгоритмами, які потрібно незначно модифікувати, з врахуванням того факту, що операція або арифметична функція може потребувати не двох операндів, а іншу їх кількість; тобто в алгоритмі POSTFIX просто доведеться забирати з стеку необхідну кількість операндів, а в алгоритмі ІТР застосовувати рекурсивний виклик самого себе для обробки кожного аргументу функції (який, взагалі кажучи, сам може бути складним арифметичним виразом).

Що ж стосується перевірки правильності виразів, то на практиці широко застосовується метод "здорового мінімалізму" - якщо алгоритм може обчислити значення виразу, то вважаємо, що вираз записано правильно. Тому відсіювати доводиться тільки ті випадки, коли обчислення виразу принципово не може відбутися, оскільки веде до помилки. В деяких випадках такий підхід суттєво зменшує кількість обчислень.

**§3. Оцінка складності алгоритмів.**

Під складністю алгоритму ми будемо розуміти сумарну складність операцій, що виконуються під час роботи алгоритму. Складність найбільш поширених операцій вважається такою:

* ввід та вивід елемента даних, присвоювання змінної - 1,
* запис на стек та зчитування зі стеку - 1,
* виклик процедури або функції - 1,
* умовний перехід (порівняння) - 1,
* виконання арифметичних чи логічних операцій - по кількості операцій,
* складність циклу рівна складності тіла циклу, помноженій на кількість повторів.

Введемо наступні означення. **Ємністю арифметичної функції** назвемо кількість дійсних (таких, що належать полю дійсних чисел) аргументів цієї функції. Всі натуральні і цілі числа надалі узагальнюємо до дійсних чисел. Наприклад: sign(x), (-x) - функції ємності 1, а  - функція ємності 2. Якщо абстрагуватися від формату запису арифметичної функції, то можна сказати, що всі двохмісні алгебраїчні операції є функціями ємності 2. Приймаючи це до уваги, можемо сформулювати наступне твердження:

**Твердження 3.**

Кількість операндів у правильно записаному арифметичному виразі є числом, яке можна отримати, віднявши від сумарної ємності всіх функцій (операцій), що входять у вираз, кількість цих самих функцій і додавши одиницю:

К-сть операндів = сумарна ємність – к-сть функцій + 1.

Очевидно, що твердження 2 є узагальненням твердження 1, бо якщо вираз містить тільки двохмісні арифметичні операції, то сумарна ємність вдвічі більша за кількість функцій. Доводиться дане твердження аналогічно попередньому.

Поняття сумарної ємності всіх функцій арифметичного виразу є важливим в тому плані, що воно є певним інваріантом, відносно якого можна робити оцінки по кількості операцій алгоритмів.

Найбільш важливими з точки зору практичного використання є наступні критерії ефективності алгоритмів:

* **критерій 1** - кількість машинних операцій, необхідних для обчислення виразу;
* **критерій 2** - об’єм пам’яті, що використовується алгоритмом;
* **критерій 3** - загальний час обробки виразу.

Критерії 1 та 3 дуже подібні між собою, проте не еквівалентні. Прогрес у мікроелектроніці привів до того, що сучасні комп’ютери, навіть ті, які мають лише один процесор, можуть виконувати декілька команд одночасно. Тому, виконуючи бодай часткове розпаралелювання обчислень, можна отримати суттєвий приріст швидкості виконання.

Ввівши необхідні нам поняття, ми можемо перейти до оцінки різних алгоритмів згідно вказаних критеріїв.

Перш за все, ми повинні звернути увагу на те, що алгоритми переводу виразу з інфіксної форми представлення до префіксної та обчислення виразу в префіксній формі можуть бути одразу виключені з розгляду згідно наступних причин:

1. алгоритм переводу виразу в префіксну форму є еквівалентним за складністю алгоритму ITP (інфіксна в постфіксну - §1);
2. алгоритм обчислення виразу в префіксній формі є еквівалентний за складністю алгоритму POSTFIX (див. твердження 2, §1);
3. алгоритм POSTFIX більш зручний для реалізації.

Тоді нам залишається розглянути тільки алгоритми обчислення виразу безпосередньо в інфіксній формі та обчислення за допомогою переведення в постфіксну форму.

Раніше ми вже відмічали, що алгоритми POSTFIX та ІТР мають складність O(N). Тоді їх послідовне застосування дає нам алгоритм обчислення виразів в інфіксній формі, складність якого буде також О(N). Спробуємо уточнити дану оцінку. Для цього просто підрахуємо операції, що виконуються в процесі роботи. Для алгоритму ІТР кількість операцій буде

К = N + КД + КО \* 3 + КБ + ППО ,

де N - кількість операцій порівняння символів (визначення наступного символу послідовності);

КД - кількість дужок (на стек попадають лише відкриваючі дужки, але ж їх потім звідти ще й забирають, тому операцій вдвічі більше);

КО - кількість операцій (кожна операція попадає на стек, знімається звідти і виводиться у вихідну послідовність);

КБ - кількість букв-ідентифікаторів (кожен ідентифікатор одразу виводиться);

ППО - порівняння пріоритетів операцій (перед тим, як занести операцію на стек, звідти видаляються операції з меншим пріоритетом).

Можна також дати наступні оцінки для розглядуваних величин:

 КД ≤ N/2 (не більше половини символів у виразі є дужками);

 KO ≤ N/2 (не більше половини символів є знаками операцій);

 КБ ≤ N/2 +1;

 ППО ≤ КО ≤ N/2.

Звідси можемо зробити висновок, що загальна кількість операцій алгоритму ІТР має порядок 4N.

 Для алгоритму POSTFIX має місце така формула:

 К = N + 2\*СЄ + КО,

де N - кількість операцій порівняння символів (визначення наступного символу послідовності);

 СЄ - сумарна ємність арифметичних функцій та операцій виразу (для кожної функції операнди спочатку попадають на стек, а потім знімаються звідти для обчислень);

КО - кількість операцій (має місце просте виконання дій або виклик арифметичних функцій).

Величину СЄ можна оцінити так:

 СЄ < N (випливає з твердження 3).

Тому загальна оцінка кількості операцій алгоритму POSTFIX має порядок 3.5 \* N.

 Тепер розглянемо деякий абстрактний алгоритм обчислення виразу в інфіксній формі (назвемо його INFIX), який знаходить шукане значення без конвертації виразу в постфіксну форму. Оцінимо той мінімально можливий набір операцій, що повинен виконати INFIX.

 К = N + КД + 3\*КО + 2\*КБ + ППО,

де N - операції аналізу символу, КД - занесення і видалення дужок зі стеку, або рекурсивний виклик алгоритму для підвиразу в дужках; 3\*КО - операції спочатку повинні заноситися на стек, до вияснення їх пріоритету відносно сусідніх операцій, потім зніматися звідти і виконуватися; 2\*КБ - букви (операнди, ідентифікатори, константи) спочатку повинні попадати на стек, потім зніматися для обчислень; ППО - порівняння пріоритетів операцій.

Тому загальну кількість операцій можна оцінити порядком 4.5\*N.

 Ми приходимо до висновку, що в загальному випадку жоден алгоритм, що обчислює арифметичний чи логічний вираз безпосередньо в інфіксній формі є не кращим за алгоритм POSTFIX, котрий обчислює той самий вираз в постфіксній формі запису. Тут потрібно також прийняти до уваги той факт, що запис виразу в постфіксній формі є коротшим, оскільки не містить дужок та ком між аргументами функцій, і найчастіше вже є лексично згорнутим, отже наведені вище оцінки алгоритмів будуть ще більше відрізнятися між собою (за суттєвої переваги POSTFIX). Отже складність алгоритму INFIX є більшою за складність POSTFIX, і в той же час меншою за складність комбінації ITP+POSTFIX (якщо ні, то ми просто замінимоINFIX на ITP+POSTFIX).

 Цей факт дуже корисно використовувати при розробці компіляторів. При використанні алгоритму POSTFIX левову частку обчислень (аналіз, перевід виразу в постфіксну форму) можна виконати на етапі компіляції, а вже безпосередньо згенерована програма буде виконувати значно менше обчислень ніж у випадку безпосереднього обчислення інфіксного виразу. Крім того, алгоритм POSTFIX потребує для роботи не більше ніж N/2 вільних елементів стекового простору, в той час як для INFIX потрібно резервувати N елементів пам’яті, оскільки на стек попадають як операнди, так і операції разом з дужками (або з рекурсивними самовикликами для обчислення підвиразів у дужках).

**§4. Оптимізація програм.**

Взагалі кажучи, безпосереднє обчислення виразів у інфіксній формі слабо піддається оптимізації. І, зважаючи на те, що складність алгоритмів обчислення є лінійною, для інтерпретаторів проблема оптимізації гостро не стоїть. Зовсім інша ситуація спостерігається в області компіляторів, де іде боротьба з кожним "зайвим" тактом роботи процесора. Існують різні методи машинно-залежної та машинно-незалежної оптимізації коду [3]. Вони можуть використовуватися на всіх рівнях синтаксичного представлення і використовувати відомості про весь текст програми для оптимізації обчислення одного виразу. Одним з найпростіших методів є так зване "**розмноження констант**". При його використанні довільне посилання на константу замінюється самим значенням константи. В наступному прикладі підвищити ефективність коду можна завдяки видаленню трьох адресних посилань і заміні їх константами:

 у = 3;

 а = (у + 1) \* ( у - 5) + с / у;

переводиться в

 у = 3;

 а = (3 + 1) \* ( 3 - 5) + с / 3;

Розмноження констант може навіть зменшити кількість необхідних присвоювань. Також воно тісно пов’язане з методом "**згортки констант**" (константною арифметикою), що зводить вирази з константами до якомога простішої форми. Сталі величини використовуються в програмі або безпосередньо (як у випадку чисел та цифр), або опосередковано (як у випадку декларування констант).

Згортка констант зводить наступний фрагмент програми на мові C

 #define TWO 2

a = 1 + TWO;

до його еквівалентної форми

 а = 3;

під час компіляції, завдяки чому видаляються зайві арифметичні операції із стадії виконання програми. Згортання констант можна застосовувати як до цілочислових констант, так і до логічних та констант з плаваючою комою.

 "**Алгебраїчні спрощення**" - це різновид згортки констант, який видаляє арифметичні тотожності. Код, що генерується для таких операторів як

 х = у + 0;

 х = у \* 0;

 х = у / 1.0;

повинен бути простим константним присвоюванням і не повинен містити команд для виконання арифметичних операцій. Подібні вирази можуть бути і частиною більш складного виразу, тому завчасне їх виявлення може значно спростити обчислювальний процес.

 "**Логічні спрощення**" - аналог попереднього методу оптимізації, що стосується логічних виразів. Оператори

 x = y AND False;

 x = y AND True;

 x = y OR True;

 x = y OR False;

та їх неявні модифікації можуть бути спрощені для уникнення зайвих обчислень. Можливі випадки навіть повного припинення обчислення логічного виразу, якщо в процесі роботи було виявлено незалежність результату від значень підвиразів.

 "**Видалення спільних підвиразів**" - це процес зменшення кількості зайвих обчислень. Замість того, щоб генерувати код для обчислення значення кожен раз, коли воно використовується, оптимізуючий компілятор може спробувати виділити вираз таким чином, щоб його значення обчислювалося тільки один раз. Там, де це можливо, наступні появи цього ж виразу використовують раніше обчислене значення. Наприклад у\*3 є спільним підвиразом у наступному тексті:

 if( a[y\*3] < 0 || b[y\*3] > 10)

 a[y\*3] = 0;

 Виділення його приводить до логічно еквівалентного тексту, що містить менше обчислень:

Т1 = у\*3;

if( a[Т1] < 0 || b[Т1] > 10)

 a[Т1] = 0;

 Видалення спільних підвиразів зазвичай відбувається всередині оператора або блоку операторів. "Глибоке видалення спільних підвиразів" є більш складним методом, оскільки тут аналізу підлягають базові блоки конкретної мови програмування.

 "**Зменшення потужності**" має на увазі заміщення операцій, які потребують більшого часу виконання, більш швидкими. Це є класичний приклад машинно-залежної оптимізації. Компілятор може використовувати зменшення потужності різними способами. Наприклад, застосовуючи зменшення потужності до вже згенерованого коду, компілятор може замінювати операції, які множать або ділять цілі числа на степені двійки операціями зсуву.

 "**Видалення зайвих присвоювань**" включає знаходження проміжку існування змінної і знищення присвоювань значень цієї змінної, якщо ці присвоювання не можуть змінити логіку програми. Цей метод звільняє обмежені ресурси, такі як стековий простір або машинні регістри (чому важливо мати багато вільних регістрів буде описано далі). В наступній послідовності команд:

 а = х + у\*3 - с;

 b = 0;

 a = b;

 перший оператор є зайвим присвоюванням і може бути безпечно видалений. Зайві присвоювання можуть виникати ненавмисно, коли проміжок існування змінної великий і між входженнями змінної є доволі багато коду. Зайві присвоювання можуть бути також результатом попередніх проходів оптимізації.

 Ціль "**розподілу змінних по регістрах**" полягає в спробі забезпечити оптимальне використання регістрів шляхом збереження часто використовуваних змінних в регістрах так довго, як це можливо, щоб виключити більш повільний доступ до пам’яті. Кількість регістрів, доступних для використання, залежить від архітектури процесора. Найбільш поширене сімейство процесорів Intel 80x86 резервує багато регістрів для спеціального використання і має декілька універсальних регістрів.

 При призначенні змінних регістрам компілятор приймає до уваги не тільки змінні, які потрібно виділити, але також і регістри, яким вони призначаються. Вибір змінних залежить від частоти їх використання, проміжків існування поточних регістрових змінних (які визначаються при аналізі потоків даних) і кількості доступних регістрів. В залежності від ступеня виконуваної компілятором оптимізації проміжок життя змінної може визначатися всередині єдиного оператора, всередині базового блоку або перекривати декілька базових блоків. Змінна зберігається в регістрі тільки якщо вона буде знову використовуватися. Якщо надалі немає посилань на змінну, то вона зберігається в оперативній пам’яті, звільняючи регістр для іншої змінної.

 Оскільки оптимізуючому компілятору відомий проміжок життя змінної, то він не буде навмисно генерувати зайві операції збереження і завантаження регістрів. Зайві операції збереження знищуються шляхом видалення зайвих присвоювань; зайві операції завантаження обходяться за допомогою вдосконаленого розподілу змінних по регістрам.

 Компілятор, що генерує код для математичного сопроцесора, прискорює виконання програми, яка виконує багато операцій з плаваючою комою. Для того, щоб підтримувати сопроцесор і максимізувати ефективність плаваючої арифметики, оптимізуючий компілятор може генерувати безпосередньо послідовність команд сопроцесора, на противагу використанню програмної емуляції функцій плаваючої арифметики.

 Розглянувши основні типи оптимізації, потрібно також зауважити, що її застосування не завжди дає потрібний ефект. Оптимізація не є панацеєю і її використання не безкоштовне. В залежності від ступеня оптимізації час, потрібний для компіляції, може значно зрости. Для невеликих програм потрібний час можна не приймати до уваги, але для великих він може бути важливим.

 Оптимізація також може утруднити відлагодження програми внаслідок генерації коду, який важко безпосередньо пов’язати з вихідними операторами в програмі. Оптимізація може не очікувано внести помилки в код, згенерований з цілком правильного тексту програми. Ситуація, коли до змінної звертаються як безпосередньо за іменем, так і засобами одного чи декількох вказівників, може утруднити роботу компілятора по визначенню того, чи "живе" ще змінна і, відповідно, повинна залишатися в регістрі, чи вона "померла" і тоді має бути збереженою в пам’яті.

**§5. Результати і висновки.**

В даній роботі було розглянуто основні способи представлення арифметичних та логічних виразів - дерево виразу, інфіксна, префіксна та постфіксна форми запису; вказано на однозначність обчислення виразу в префіксній та постфіксній формах, і на необхідність використання дужок або пріоритету операцій в інфіксній формі. Розглянуто алгоритм POSTFIX обчислення виразу в постфіксній формі, доведено правильність його роботи, описано алгоритм переведення виразу в постфіксну форму (ІТР). В теоретичній частині сформульовано та доведено твердження 1 - 3, розглянуто основні типи аналізу виразів, звернуто увагу на корисність та важливість виконання лексичної згортки виразу перед його обчисленням, показано недоцільність використання префіксної форми запису виразів та дано оцінку складності алгоритмів POSTFIX, ІТР і абстрактного алгоритму обчислення виразу в інфіксній формі INFIX. Зокрема виявлено, що жоден алгоритм INFIX в загальному випадку не є кращим за алгоритм POSTFIX, на основі чого зроблено висновок про корисність використання постфіксної форми запису в компіляторах. Також наведено багато методів оптимізації вихідного коду компілятора.

В практичній частині було реалізовано алгоритми POSTFIX та ІТР в одній програмі, з метою їх сумісного використання. Також розроблено простий компілятор виразів, який генерує послідовність команд на асемблері процесорів сімейства Intel 80x86 для обчислення значення вхідного виразу. В додатках подано тексти програм і приклади їх роботи.

**Використана література.**

1. Дьяконов В.Ю. и др. Системное программирование, – М.: 1990. - с. 254–264.
2. Креншоу Дж. Давайте создадим компилятор. Мережа інтернет, http://www.kulichki.com/kit/crenshaw/crenshaw.html .
3. Aaby А. Compiler construction using Flex and Bison. Мережа інтернет, http://www.kulichki.com/kit .

**Додаткова література.**

1. Баррон Д. Рекурсивные методы в программировании, - М.: 1974
2. Дмитриева М.В., Кубенский А.А. Элементы современного программирования, - СПб.: 1991.
3. Барашенков В.В. Анализ и преобразование операторных схем алгоритмов: учебное пособие. - Л.: ЛЭТИ, 1979.
4. Кинг Д. Создание еффективного программного обеспечения. - М.: Мир, 1991.
5. Барашенков В.В. Интерпретация операторных схем алгоритмов. - Л.: ЛЭТИ, 1978.
6. Бентли Дж. Жемчужины творчества программистов. - М.: Мир, 1990.
7. Шауман А.М. Основы машинной арифметики. - 1979.
8. Ахо, Альфред В. и др. Построение и анализ вычислительных алгоритмов. - М.: Мир, 1979
9. Криницкий Н.А. Программирование и алгоритмические языки. - 1975.
10. Прикладные вопросы системного анализа. Межвузовский тематический сборник. Вып. 2. Куйбышев, 1976.
11. Автоматическое построение ассоциативного списка со сжатием информации. - К., 1976.
12. Квиттнер П. Задачи, программы, вычисления, результаты. - М.: Мир, 1980.
13. Шауман А.М. Основы машинной арифметики. - 1979.
14. Малоземов В.Н. Певный А.Б. Рекуррентные вычисления. - Л., изд. ун-та, 1976.

**Додаток 1.**

**Програмна реалізація алгоритмів POSTFIX та ITP.**

Для демонстрації роботи алгоритмів POSTFIX та ITP було складено програму, котра, отримавши на вході список змінних разом із значеннями та арифметичний вираз у інфіксній формі, спочатку переводить вираз у постфіксну форму (функція ITP), а потім обчислює його значення (функція CalculatePostfix).

Обмеження, що накладаються програмою на вхідні дані:

* всі змінні позначаються одним символом;
* у виразі немає числових констант;
* вираз не містить посилання на арифметичні функції;
* вираз є правильно записаним (відсутні перевірки на правильність).

Для обчислення виразу в постфіксній формі функція CalculatePostfix використовує стек, реалізований в наступному модулі:

Unit Stack;

Interface

Var S:Array[1..100] Of Real;

 Top:Integer;

Procedure Push(Number:Real);

Procedure Pop(Var Number:Real);

Function IsEmpty:Boolean;

Procedure ClearStack;

Implementation

Procedure ClearStack;

Begin

 Top:=0;

End;

Procedure Push(Number:Real);

Begin

 Inc(Top);

 S[Top]:=Number;

End;

Procedure Pop(Var Number:Real);

Begin

 Number:=S[Top];

 Dec(Top);

End;

Function IsEmpty:Boolean;

Begin

 If Top=0 Then IsEmpty:=True Else IsEmpty:=False;

End;

Begin

 Top:=0;

End.

 Текст програми:

Uses Crt, Stack;

Type Id=Record

 c:Char;

 v:Real;

 End;

Var Ex,S:String;

 Stek:Array[1..255] Of Char;

 N:Integer;

 Tab:Array[1..26] Of Id;

Function InputExpression:String;

Var s:String;

 i:Integer;

Begin

 WriteLn('Скільки змінних буде у виразі ?');

 ReadLn(N);

 For i:=1 To N Do

 Begin

 WriteLn('Введіть один символ для позначення змінної номер ',i);

 ReadLn(Tab[i].c);

 WriteLn('Введіть значення ',Tab[i].c);

 ReadLn(Tab[i].v);

 End;

 WriteLn('Введіь арифметичний вираз у інфіксній формі ');

 ReadLn(s);

 InputExpression:=s;

End;

Function IsOperation(c:Char):Boolean;

Begin

 If c in ['/','\*','+','-','^'] Then IsOperation:=True

 Else IsOperation:=False;

End;

Function IsVariable(c:Char):Integer;

Var i:Integer;

Begin

 For i:=1 To N Do

 If Tab[i].c=c Then

 Begin

 IsVariable:=i;

 Exit;

 End;

 IsVariable:=0;

End;

Function MakeOperation(p1,p2:Real; Op:Char):Real;

Begin

 Case Op Of

 '+' : MakeOperation:=p1+p2;

 '-' : MakeOperation:=p1-p2;

 '\*' : MakeOperation:=p1\*p2;

 '/' : MakeOperation:=p1/p2;

 '^' : MakeOperation:=Exp(Ln(p1)\*p2);

 End;

End;

Function CalculatePostfix(Ex:String):Real;

Var i,j:Integer;

 p1,p2:Real;

 b:Boolean;

Begin

 For i:=1 To Length(Ex) Do

 Begin

 b:=IsOperation(Ex[i]);

 j:=IsVariable(Ex[i]);

 If b Then

 Begin

 Pop(p1);

 Pop(p2);

 Push(MakeOperation(p2,p1,Ex[i]));

 End

 Else

 Begin

 Push(Tab[IsVariable(Ex[i])].v);

 End;

 End;

 Pop(p1);

 CalculatePostfix:=p1;

End;

Function Priority(c:Char):Byte;

Begin

 Case c Of

 'e' : Priority:=0;

 '(' : Priority:=0;

 ')' : Priority:=0;

 '-' : Priority:=1;

 '+' : Priority:=1;

 '\*' : Priority:=2;

 '/' : Priority:=2;

 '^' : Priority:=3;

 End;

End;

Function ITP(Ex:String):String;

Var i,j:Integer;

 Rez:String;

Begin

 Rez:='';

 Stek[1]:='e'; j:=1;

 For i:=1 To Length(Ex) Do

 Begin

 If Ex[i] in ['A'..'Z','a'..'z'] Then Rez:=Rez+Ex[i];

 If Ex[i]='(' Then Begin j:=j+1; Stek[j]:='('; End;

 If Ex[i]=')' Then

 Begin

 While Stek[j]<>'(' Do Begin Rez:=Rez+Stek[j]; Dec(j); End;

 Dec(j);

 End;

 If Ex[i] in ['+','-','\*','/','^'] Then

 Begin

 While Priority(Ex[i])<=Priority(Stek[j]) Do

 Begin Rez:=Rez+Stek[j]; Dec(j); End;

 Inc(j); Stek[j]:=Ex[i];

 End;

 End;

 While Stek[j]<>'e' Do

 Begin Rez:=Rez+Stek[j]; Dec(j); End;

 ITP:=Rez;

End;

Begin

 ClrScr;

 Ex:=InputExpression;

 S:=ITP(Ex);

 WriteLn('Вираз у постфіксній формі: ',S);

 WriteLn('Результат ',CalculatePostfix(S):12:6);

 WriteLn('Натисніть довільну клавішу');

 ReadLn;

End.

 **Приклади роботи програми.**

**Приклад 1.**

Скiльки змiнних буде у виразi ?

4

Введiть один символ для позначення змiнної номер 1

a

Введiть значення a

1

Введiть один символ для позначення змiнної номер 2

b

Введiть значення b

2

Введiть один символ для позначення змiнної номер 3

c

Введiть значення c

3

Введiть один символ для позначення змiнної номер 4

d

Введiть значення d

4

Введiть арифметичний вираз у iнфiкснiй формi

b^(c\*(d+a))

Вираз у постфiкснiй формi: bcda+\*^

Результат 32768.000000

Натиснiть довiльну клавiшу

**Приклад 2.**

Скiльки змiнних буде у виразi ?

3

Введiть один символ для позначення змiнної номер 1

W

Введiть значення W

1.5

Введiть один символ для позначення змiнної номер 2

P

Введiть значення P

10

Введiть один символ для позначення змiнної номер 3

R

Введiть значення R

1.05

Введiть арифметичний вираз у iнфiкснiй формi

W\*R^P

Вираз у iнфiкснiй формi: WRP^\*

Результат 2.443342

Натиснiть довiльну клавiшу

**Додаток 2.**

**Програмна реалізація простого компілятора виразів.**

Нижче подано текст програми, написаної для компілятивної обробки [2] арифметичних виразів, записаних в інфіксній формі. Ця програма може бути використана як частина компілятора, котра відповідає за генерацію асемблерного коду для операторів присвоєння.

Особливості даної програми:

* однопрохідна генерація коду;
* відсутній лексичний сканер, тому всі ідентифікатори включаються в асемблерний код без перевірки;
* всі обчислення є цілочисловими;
* арифметичні функції не приймають параметрів, а результат повертають в регістрі AX;
* розпізнається більшість синтаксичних помилок;
* коректно розпізнаються пробіли та символи табуляції у виразі.

Текст програми:

const TAB = ^I;

 CR = ^M;

Var Look: Char;

Procedure GetChar;

Begin

 Read(Look);

End;

Procedure Error(s: String);

Begin

 WriteLn;

 WriteLn(^G, 'Помилка : ', s, '.');

End;

Procedure Abort(s: String);

Begin

 Error(s);

 Halt;

End;

Procedure Expected(s: String);

Begin

 Abort(s + ' Очікується');

End;

Function IsAlpha(c: Char): Boolean;

Begin

 IsAlpha := UpCase(c) In ['A'..'Z'];

End;

Function IsDigit(c: Char): Boolean;

Begin

 IsDigit := c In ['0'..'9'];

End;

Function IsAlNum(c: Char): Boolean;

Begin

 IsAlNum := IsAlpha(c) Or IsDigit(c);

End;

Function IsAdDop(c: Char): Boolean;

Begin

 IsAdDop := c In ['+', '-'];

End;

Function IsWhite(c: Char): Boolean;

Begin

 IsWhite := c In [' ', TAB];

End;

Procedure SkipWhite;

Begin

 While IsWhite(Look) Do

 GetChar;

End;

Procedure Match(x: Char);

Begin

 If Look <> x Then Expected('''' + x + '''')

 Else Begin

 GetChar;

 SkipWhite;

 End;

End;

Function GetName: String;

Var Token: String;

Begin

 Token := '';

 If NOT IsAlpha(Look) Then Expected('Ім’я');

 While IsAlNum(Look) Do Begin

 Token := Token + UpCase(Look);

 GetChar;

 End;

 GetName := Token;

 SkipWhite;

End;

Function GetNum: String;

Var Value: String;

Begin

 Value := '';

 If NOT IsDigit(Look) Then Expected('Ціле число');

 While IsDigit(Look) Do Begin

 Value := Value + Look;

 GetChar;

 End;

 GetNum := Value;

 SkipWhite;

End;

Procedure Emit(s: String);

Begin

 Write(TAB, s);

End;

Procedure EmitLn(s: String);

Begin

 Emit(s);

 WriteLn;

End;

Procedure Ident;

Var Name: String[8];

Begin

 Name:= GetName;

 If Look = '(' Then Begin

 Match('(');

 Match(')');

 EmitLn('CALL ' + Name);

 End

 Else

 EmitLn('MOV ' + Name + ',AX');

End;

Procedure Expression; Forward;

Procedure Factor;

Begin

 If Look = '(' Then Begin

 Match('(');

 Expression;

 Match(')');

 End

 Else If IsAlpha(Look) Then

 Ident

 Else

 EmitLn('MOV ' + GetNum + ',AX');

End;

Procedure Multiply;

Begin

 Match('\*');

 Factor;

 EmitLn('POP DX');

 EmitLn('MUL AX,DX');

End;

Procedure Divide;

Begin

 Match('/');

 Factor;

 EmitLn('POP DX');

 EmitLn('DIV DX,AX');

 EmitLn('MOV DX,AX');

End;

Procedure Term;

Begin

 Factor;

 While Look In ['\*', '/'] Do Begin

 EmitLn('PUSH AX');

 Case Look Of

 '\*': Multiply;

 '/': Divide;

 End;

 End;

End;

Procedure Add;

Begin

 Match('+');

 Term;

 EmitLn('POP DX');

 EmitLn('ADD AX,DX');

End;

Procedure Subtract;

Begin

 Match('-');

 Term;

 EmitLn('POP DX');

 EmitLn('SUB DX,AX');

 EmitLn('MOV DX,AX');

End;

Procedure Expression;

Begin

 If IsAdDop(Look) Then

 EmitLn('XOR AX,AX')

 Else

 Term;

 While IsAdDop(Look) Do Begin

 EmitLn('PUSH AX');

 Case Look Of

 '+': Add;

 '-': Subtract;

 End;

 End;

End;

Procedure Assignment;

Var Name: String[8];

Begin

 Name := GetName;

 Match('=');

 Expression;

 EmitLn('MOV AX,' + Name);

End;

Procedure Init;

Begin

 GetChar;

 SkipWhite;

End;

Begin

 WriteLn('Введіть оператор присвоєння');

 Init;

 WriteLn('Результуючий код на асемблерi');

 Assignment;

 If Look <> CR Then Expected('Новий рядок');

End.

 **Приклади роботи програми.**

**Приклад 1.**

Введiть оператор присвоєння

a=123\*(b+e)-func33()/3

Результуючий код на асемблерi

 MOV 123,AX

 PUSH AX

 MOV B,AX

 PUSH AX

 MOV E,AX

 POP DX

 ADD AX,DX

 POP DX

 MUL AX,DX

 PUSH AX

 CALL FUNC33

 PUSH AX

 MOV 3,AX

 POP DX

 DIV DX,AX

 MOV DX,AX

 POP DX

 SUB DX,AX

 MOV DX,AX

 MOV AX,A

**Приклад 2.**

Введiть оператор присвоєння

Variable=Val1\*Val2+Stat() - 11 / epsilon

Результуючий код на асемблерi

 MOV VAL1,AX

 PUSH AX

 MOV VAL2,AX

 POP DX

 MUL AX,DX

 PUSH AX

 CALL STAT

 POP DX

 ADD AX,DX

 PUSH AX

 MOV 11,AX

 PUSH AX

 MOV EPSILON,AX

 POP DX

 DIV DX,AX

 MOV DX,AX

 POP DX

 SUB DX,AX

 MOV DX,AX

 MOV AX,VARIABLE

**Приклад 3.**

Введiть оператор присвоєння

RESULT=(MAX-MIN)\*(-3/LENGTH)

Результуючий код на асемблерi

 MOV MAX,AX

 PUSH AX

 MOV MIN,AX

 POP DX

 SUB DX,AX

 MOV DX,AX

 PUSH AX

 XOR AX,AX

 PUSH AX

 MOV 3,AX

 PUSH AX

 MOV LENGTH,AX

 POP DX

 DIV DX,AX

 MOV DX,AX

 POP DX

 SUB DX,AX

 MOV DX,AX

 POP DX

 MUL AX,DX

 MOV AX,RESULT

 **Приклад 4.**

Введiть оператор присвоєння

OUT=IN+10\*(value()

Результуючий код на асемблерi

 MOV IN,AX

 PUSH AX

 MOV 10,AX

 PUSH AX

 CALL VALUE

Помилка: ')' Очiкується.

 **Приклад 5.**

Введiть оператор присвоєння

Size=Long\*Wide+X-

Результуючий код на асемблерi

 MOV LONG,AX

 PUSH AX

 MOV WIDE,AX

 POP DX

 MUL AX,DX

 PUSH AX

 MOV X,AX

 POP DX

 ADD AX,DX

 PUSH AX

Помилка: Цiле число Очiкується.