**ТЕОРЕТИЧНІ ПИТАННЯ ДО ЕКЗАМЕНУ**

**з дисципліни**

**"МАТЕМАТИЧНІ ОСНОВИ ІНФОРМАТИКИ"**

кафедра комп’ютерних технологій, ДНУ, 2023/2024

1. **Висловлювання. Синтаксис. Обчислення висловлювань у даному стані.**

В математичних основах програмування висловлювання є основною будівельною одиницею логічних виразів, які обробляються комп'ютером. Вони використовуються для порівнянь, умовних операцій, управління потоком програми та багатьох інших завдань. Розглянемо основні аспекти висловлювань у математичних основах програмування:

Висловлювання:

Синтаксис: Висловлювання можуть включати логічні оператори (AND, OR, NOT), порівняння (>, <, =, ≠) та інші логічні елементи.

Змінні: Змінні можуть приймати логічні значення TRUE або FALSE.

Логічні оператори: AND (логічне І), OR (логічне АБО), NOT (логічне НЕ) - використовуються для побудови складних логічних виразів.

Обчислення висловлювань у даному стані:

Порівняння: Виконуються порівняння між значеннями, які можуть бути TRUE або FALSE.

Логічні операції: Висловлювання можуть об'єднуватися за допомогою логічних операцій для утворення складних умов.

Керування потоком: Результати висловлювань використовуються для управління потоком програми за допомогою конструкцій умовного виконання (if-else) та циклів.

Приклад:

a = 5

b = 3

c = 7

# Висловлювання

is\_greater = a > b # TRUE

is\_equal = a == c # FALSE

# Логічні операції

result = (is\_greater and not is\_equal) or (b < c) # TRUE

У математичних основах програмування висловлювання грають важливу роль у створенні умовних конструкцій, циклів, функцій та багатьох інших аспектів програмування.

1. **Правила старшинства операцій для висловлювань. Тавтології. Висловлювання як множини станів. Приклади.**

**"Математичні основи програмування" - це галузь науки, яка вивчає математичні концепції та методи, що застосовуються у програмуванні та розробці програмного забезпечення. Декілька ключових понять і правил, що стосуються цієї галузі, включають в себе:**

**Правила старшинства операцій: Це правила, які визначають порядок виконання операцій у виразах. Наприклад, в математичних виразах спочатку виконуються операції в дужках, потім множення або ділення, і нарешті додавання або віднімання. Це правило допомагає у визначенні точного порядку виконання операцій у складних виразах.**

**Тавтології: Тавтологія - це висловлення, яке є істинним за будь-яких обставин. У програмуванні тавтологія може виникати, наприклад, у виразах логічних умов, де вони завжди відповідають істині або хибі.**

**Висловлювання як множини станів: Це підхід до моделювання висловлювань, де кожне висловлювання представляється як множина можливих станів або ситуацій. Наприклад, у логіці висловлювання "Сьогодні сонячний день" може бути представлене як множина станів {сонячний день, несонячний день}.**

**Приклади:**

**Правила старшинства операцій:**

**Вираз: 2 + 3 \* 4**

**За правилом старшинства операцій, спочатку виконається множення, а потім додавання:**

**= 2 + (3 \* 4) = 2 + 12 = 14**

**Тавтології:**

**Висловлення: (x > 5) або (x <= 5)**

**Це тавтологія, оскільки будь-яке значення x або більше 5, або менше чи дорівнює 5.**

**Висловлювання як множини станів:**

**Висловлення: Користувач авторизований**

**Множина станів: {авторизований, неавторизований}**

**Це висловлення може бути істинним (користувач авторизований) або хибним (користувач неавторизований) в залежності від конкретного стану системи.**

1. **Еквівалентні перетворення. Закони еквівалентності. Правила підстановки і транзитивності.**

Еквівалентні перетворення є основою для спрощення логічних виразів і доведення правильності алгоритмів у математичних основах програмування. Вони ґрунтуються на законах еквівалентності, які дозволяють замінювати одні вирази на інші, не змінюючи їх значення. Розглянемо основні закони еквівалентності та правила підстановки і транзитивності.

Изображение выглядит как текст, снимок экрана, Шрифт, документ

Автоматически созданное описание

Изображение выглядит как текст, Шрифт, линия

Автоматически созданное описание

Еквівалентні перетворення є важливими для розуміння та маніпулювання логічними виразами в математичних основах програмування, оскільки вони дозволяють спростити та оптимізувати алгоритми, зберігаючи при цьому їх коректність.

1. **Предикати: розширені операції і стани. Обчислення предикатів.**

1. Розширені операції над предиктами:

Логічні операції:

Кон'юнкція (∧): істинність, якщо обидва предиката істинні.

Диз'юнкція (∨): істинність, якщо принаймні один предикат істинний.

Заперечення (¬): істинність, якщо початковий предикат хибний.

Імплікація (→): істинність, якщо з істиності початкового предиката випливає істинність кінцевого.

Еквівалентність (↔): істинність, якщо обидва предиката мають однакові значення істиності.

Порівняння:

Рівність (==): істинність, якщо два операнди мають однакові значення.

Нерівність (!=): істинність, якщо два операнди мають різні значення.

Більше (>): істинність, якщо перший операнд більший за другий.

Менше (<): істинність, якщо перший операнд менший за другий.

Більше або дорівнює (>=): істинність, якщо перший операнд більший або дорівнює другому.

Менше або дорівнює (<=): істинність, якщо перший операнд менший або дорівнює другому.

Кількісні оператори:

Існує (∃): істинність, якщо принаймні один елемент у множині задовольняє предикат.

Для всіх (∀): істинність, якщо всі елементи множини задовольняють предикат.

2. Стани предикатів:

Істинний: предикат описує дійсність.

Хибний: предикат не описує дійсність.

Невизначений: значення предиката не може бути визначене.

3. Обчислення предикатів:

Прості предикати: оцінюються безпосередньо, наприклад, "x > 5".

Складні предикати: обчислюються за допомогою комбінації простих предикатів і логічних операторів, наприклад, "(x > 5) ∧ (y < 10)".

4. Приклади:

"Число 5 є парним?" - предикат "x є парним", де x =

5. Оцінка: істинний.

"Чи всі числа в множині {1, 2, 3} менші за 5?" - предикат "∀x ∈ {1, 2, 3}, x < 5". Оцінка: хибний.

"Чи існує число в множині {1, 2, 3}, яке більше за 4?" - предикат "∃x ∈ {1, 2, 3}, x > 4". Оцінка: істинний.

1. **Квантори існування, загальності і кількості.**

Квантори існування, загальності і кількості є ключовими поняттями в математичній логіці та теорії множин. Вони використовуються для формалізації тверджень про існування, загальність та кількість об'єктів у математичних ствердженнях.

Квантори існування, загальності і кількості включають наступні концепції:

Квантор існування (∃): Використовується для вираження твердження про те, що певний об'єкт або об'єкти існують, якщо принаймні один об'єкт задовольняє певну властивість.

Приклад: "∃x (x > 3)" виражає твердження про існування принаймні одного числа, яке більше за 3.

Квантор загальності (∀): Використовується для вираження твердження, яке стосується всіх об'єктів в множині.

Приклад: "∀x (x > 0)" виражає загальне твердження, що каже, що для всіх x, x є більшим за 0.

Квантор кількості (k): Використовується для вираження твердження про точну кількість об'єктів, які задовольняють певну властивість.

Приклад: "kx (x < 5)" виражає твердження про те, що точно k об'єктів мають значення менше за 5.

Використання в математиці:

Ці квантори використовуються для формалізації і узагальнення математичних тверджень, що дозволяє точно визначати існування об'єктів, загальність властивостей та кількість об'єктів в математичних твердженнях. Вони є важливими для математичних доведень, теорії множин, математичної логіки та багатьох інших галузей математики.

1. **Кратні підстановки. Правило транзитивності для імплікації**.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

1. **Функціональне означення масивів.**

У математичних основах програмування масиви можна розглядати з функціональної точки зору як відображення індексів на значення. Формальне функціональне означення масиву може бути записане наступним чином:

Изображение выглядит как текст, снимок экрана, Шрифт, документ

Автоматически созданное описание

Функціональне означення масивів є корисним для формального опису та аналізу операцій з масивами, таких як індексування, присвоєння значень, обхід масиву та інші. Воно також дозволяє розглядати масиви як особливі випадки загальних функцій і застосовувати до них відповідні математичні властивості та операції.

1. **Функціональне означення багатовимірних масивів.**

**Функціональне означення багатовимірного масиву описує його як функцію від декількох індексів, де кожен індекс відповідає одному виміру.**

**Приклад:**

def двовимірний\_масив(i, j):

"""

Функція, яка описує двовимірний масив.

Args:

i: Ціле число, що відповідає рядку.

j: Ціле число, що відповідає стовпчику.

Returns:

Елемент масиву на перетині рядка i і стовпчика j.

"""

...

У цьому прикладі двовимірний\_масив - це функція, яка приймає два цілих числа i і j як аргументи і повертає значення елемента масиву на перетині рядка i і стовпчика j.

3. Рекурсивне означення

Багатовимірні масиви також можна означити рекурсивно.

Приклад:

def багатовимірний\_масив(\*індекси):

"""

Функція, яка описує багатовимірний масив рекурсивно.

Args:

\*індекси: Кортеж цілих чисел, що відповідають індексам у кожному вимірі.

Returns:

Елемент масиву за заданими індексами.

"""

if len(індекси) == 0:

return ...

else:

return багатовимірний\_масив(\*індекси[1:])[індекси[0]]

Функціональне означення багатовимірних масивів

1. Вступ

Багатовимірні масиви - це узагальнення одновимірних масивів, які дозволяють організувати дані в декількох вимірах. Наприклад, двовимірний масив може використовуватися для представлення таблиці, а тривимірний - для представлення куба.

2. Функціональне означення

Функціональне означення багатовимірного масиву описує його як функцію від декількох індексів, де кожен індекс відповідає одному виміру.

Приклад:

Python

def двовимірний\_масив(i, j):

"""

Функція, яка описує двовимірний масив.

Args:

i: Ціле число, що відповідає рядку.

j: Ціле число, що відповідає стовпчику.

Returns:

Елемент масиву на перетині рядка i і стовпчика j.

"""

...

Use code with caution.

У цьому прикладі двовимірний\_масив - це функція, яка приймає два цілих числа i і j як аргументи і повертає значення елемента масиву на перетині рядка i і стовпчика j.

3. Рекурсивне означення

Багатовимірні масиви також можна означити рекурсивно.

Приклад:

Python

def багатовимірний\_масив(\*індекси):

"""

Функція, яка описує багатовимірний масив рекурсивно.

Args:

\*індекси: Кортеж цілих чисел, що відповідають індексам у кожному вимірі.

Returns:

Елемент масиву за заданими індексами.

"""

if len(індекси) == 0:

return ...

else:

return багатовимірний\_масив(\*індекси[1:])[індекси[0]]

Use code with caution.

У цьому прикладі багатовимірний\_масив - це рекурсивна функція, яка приймає кортеж індексів як аргумент. Якщо індекси порожні, то функція повертає значення за замовчуванням. В іншому випадку, вона рекурсивно викликає себе з усіма індексами, за винятком першого, і використовує перший індекс як ключ для доступу до значення в масиві.

4. Переваги функціонального означення

Функціональне означення багатовимірних масивів має ряд переваг:

Універсальність: Функціональне означення може бути використано для масивів будь-якої розмірності.

Простота: Функціональне означення є лаконічним і чітким.

Модульність: Функціональне означення можна легко використовувати в модульних програмах.

5. Недоліки функціонального означення

Функціональне означення багатовимірних масивів також має ряд недоліків:

Ефективність: Функціональне означення може бути менш ефективним, ніж ітеративне означення, especially for large arrays.

Складність: Функціональне означення може бути складним для розуміння, especially for beginners.

1. **Специфікації програм. Приклади специфікацій.**

**Специфікація програми - це формальний або напівформальний опис вимог, які повинна задовольняти програма. Вона визначає необхідну поведінку програми та дозволяє перевірити, чи програма працює так, як очікувалося. Специфікація програми зазвичай складається з трьох основних частин:**

**Передумови (preconditions) - умови, які повинні виконуватись перед виконанням програми. Вони визначають припустимі вхідні дані та стани, за яких програма може бути виконана.**

**Функціональні вимоги - опис того, що саме повинна робити програма. Це визначає необхідну поведінку програми та очікувані результати для різних вхідних даних.**

**Постумови (postconditions) - умови, які мають виконуватись після успішного завершення роботи програми. Вони визначають властивості результатів або кінцевих станів програми.**

**Приклади специфікацій:**

**Приклад 1: Функція знаходження максимуму**

**Передумова: Масив arr непорожній і всі його елементи є числовими значеннями.**

**Функція: findMax(arr)**

**Функціональна вимога: Знайти максимальне значення серед елементів масиву arr.**

**Постумова: Результат є максимальним значенням серед елементів arr.**

**Приклад 2: Процедура сортування масиву**

**Передумова: Масив arr непорожній, і його елементи можуть бути порівняні за допомогою оператора <.**

**Процедура: sortArray(arr)**

**Функціональна вимога: Відсортувати елементи масиву arr в порядку зростання.**

**Постумова: Елементи масиву arr відсортовані в порядку зростання.**

**Приклад 3: Програма розрахунку площі трикутника**

**Передумова: Сторони a, b і c є додатними числами, що задовольняють нерівність трикутника (a + b > c, b + c > a, a + c > b).**

**Програма: calculateTriangleArea(a, b, c)**

**Функціональна вимога: Обчислити площу трикутника за формулою Герона.**

**Постумова: Результат є дійсним числом, що дорівнює площі трикутника із сторонами a, b і c.**

**Специфікації програм допомагають чітко визначити очікувану поведінку та вимоги до програми, що полегшує процес її розробки, тестування та верифікації. Вони також забезпечують кращу комунікацію між розробниками та користувачами програми.**

1. Найслабші передумови. Визначення найпростіших команд за допомогою найслабших передумов.

Найслабша передумова (weakest precondition) - це найбільш загальна умова на вхідні дані програми (або її фрагменту), за якої гарантується виконання певної постумови після успішного завершення виконання програми. Іншими словами, якщо найслабша передумова виконується, то після виконання програми буде виконуватись відповідна постумова.

Найслабші передумови використовуються для формального визначення семантики команд програми та для перевірки коректності програм. Вони дозволяють виразити зв'язок між станом програми до і після виконання команди.

Визначення найслабших передумов для найпростіших команд:

1. \*\*Присвоєння (x := e):\*\*

Найслабша передумова для команди присвоєння x := e та постумови Q - це Q[x/e], тобто результат заміни всіх вільних входжень x у Q на вираз e.

Наприклад, якщо Q = (x > 0), то для команди x := y + 1 найслабша передумова wp(x := y + 1, x > 0) = (y + 1 > 0).

2. \*\*Складена команда (S; T):\*\*

Для складеної команди, що складається з послідовно виконуваних підкоманд S і T, найслабша передумова для постумови Q визначається рекурсивно:

wp(S; T, Q) = wp(S, wp(T, Q))

Тобто, найслабша передумова для (S; T) з постумовою Q - це найслабша передумова для S з постумовою wp(T, Q).

3. \*\*Умовний оператор (if B then S else T):\*\*

Найслабша передумова для умовного оператора з постумовою Q:

wp(if B then S else T, Q) = (B ⇒ wp(S, Q)) ∧ (¬B ⇒ wp(T, Q))

Тобто, передумова вимагає, щоб для істинного B виконувалась найслабша передумова гілки S з постумовою Q, а для хибного B - найслабша передумова гілки T з постумовою Q.

4. \*\*Цикл (while B do S):\*\*

Найслабша передумова для циклу з постумовою Q визначається як найслабша розв'язуюча формула для рівняння:

wp(while B do S, Q) = I

I ∧ B ⇒ wp(S, I)

I ∧ ¬B ⇒ Q

Це означає, що найслабша передумова I циклу з постумовою Q повинна задовольняти дві умови: якщо I істинна і B істинне, то I повинна імплікувати найслабшу передумову тіла циклу S з постумовою I (інваріант циклу); і якщо I істинна, а B хибне, то I повинна імплікувати Q (умова виходу з циклу).

Визначення найслабших передумов дозволяє формально описати семантику програм і використовувати їх для перевірки коректності та доведення властивостей програм.

1. **Властивості найслабших передумов. Доведення властивості монотонності.**

Найслабші передумови мають декілька важливих властивостей, які використовуються для формального доведення коректності та аналізу програм:

1. \*\*Строга монотонність\*\*: Якщо постумова Q1 імплікує Q2, тобто Q1 ⇒ Q2, то найслабша передумова wp(S, Q1) також імплікує wp(S, Q2) для будь-якої команди S.

Доведення:

Нехай Q1 ⇒ Q2 - істинне для будь-яких станів.

Потрібно показати, що wp(S, Q1) ⇒ wp(S, Q2) також істинне для будь-яких станів.

Розглянемо довільний стан σ, для якого wp(S, Q1) істинне. Це означає, що після виконання команди S у стані σ, постумова Q1 виконується.

Оскільки Q1 ⇒ Q2, то з виконання Q1 випливає виконання Q2.

Отже, після виконання команди S у стані σ, постумова Q2 також виконується.

За визначенням найслабшої передумови, це означає, що wp(S, Q2) істинна для стану σ.

Таким чином, для довільного стану σ з wp(S, Q1) випливає wp(S, Q2), що і потрібно було довести.

2. \*\*Субституційна властивість\*\*: Для будь-якої команди S, постумови Q і вираження e,

wp(S, Q[x/e]) = wp(S[x/e], Q),

де Q[x/e] означає заміну всіх вільних входжень змінної x у Q на вираз e, а S[x/e] - заміну x на e в команді S.

3. \*\*Диз'юнктивність\*\*: wp(S, Q1 ∨ Q2) = wp(S, Q1) ∨ wp(S, Q2)

4. \*\*Кон'юнктивність\*\*: wp(S, Q1 ∧ Q2) = wp(S, Q1) ∧ wp(S, Q2)

5. \*\*Тавтологічність\*\*: wp(S, true) = true

Ці властивості використовуються для спрощення обчислень і перетворень найслабших передумов під час аналізу та верифікації програм.

1. **Властивості найслабших передумов. Доведення властивості дистрибутивності диз'юнкції.**

**Однією з важливих властивостей найслабших передумов є дистрибутивність диз'юнкції: для будь-якої команди S та постумов Q1 і Q2 виконується рівність:**

**wp(S, Q1 ∨ Q2) = wp(S, Q1) ∨ wp(S, Q2)**

**Доведемо цю властивість.**

**Доведення:**

**Нехай σ - довільний стан перед виконанням команди S.**

**1) Покажемо, що wp(S, Q1 ∨ Q2) ⇒ (wp(S, Q1) ∨ wp(S, Q2)):**

**Нехай wp(S, Q1 ∨ Q2) істинна для стану σ. За визначенням найслабшої передумови це означає, що після виконання команди S у стані σ, постумова (Q1 ∨ Q2) буде істинною.**

**Отже, або Q1 буде істинною після виконання S, або Q2 буде істинною після виконання S.**

**У першому випадку за визначенням wp(S, Q1) буде істинною для σ.**

**У другому випадку за визначенням wp(S, Q2) буде істинною для σ.**

**Таким чином, або wp(S, Q1), або wp(S, Q2) буде істинною для σ, що і потрібно було довести.**

**2) Покажемо, що (wp(S, Q1) ∨ wp(S, Q2)) ⇒ wp(S, Q1 ∨ Q2):**

**Нехай (wp(S, Q1) ∨ wp(S, Q2)) істинна для стану σ.**

**Це означає, що або wp(S, Q1) істинна для σ, або wp(S, Q2) істинна для σ.**

**Розглянемо ці випадки:**

**а) Якщо wp(S, Q1) істинна для σ, то за визначенням після виконання S у стані σ постумова Q1 буде істинною, а отже, і диз'юнкція (Q1 ∨ Q2) буде істинною.**

**б) Якщо wp(S, Q2) істинна для σ, то аналогічно після виконання S у стані σ постумова Q2 буде істинною, а отже, і диз'юнкція (Q1 ∨ Q2) буде істинною.**

**У обох випадках після виконання S у стані σ постумова (Q1 ∨ Q2) буде істинною, що за визначенням означає істинність wp(S, Q1 ∨ Q2) для стану σ.**

**Підсумовуючи доведення в обох напрямках, ми встановили рівність:**

**wp(S, Q1 ∨ Q2) = wp(S, Q1) ∨ wp(S, Q2)**

**Таким чином, ми довели властивість дистрибутивності диз'юнкції для найслабших передумов.**

1. **Означення присвоювань і кратних присвоювань. Приклади.**

**\*\*Присвоювання\*\***

**Присвоювання - це одна з найпростіших і найважливіших команд у програмуванні. Воно дозволяє зберігати значення виразу у змінній. У математичних основах програмування присвоювання має таке формальне означення:**

**Нехай x є змінною, а e - виразом, значення якого може бути обчислене. Тоді команда присвоювання записується як:**

**x := e**

**Семантика цієї команди полягає в тому, що значення виразу e обчислюється, а потім присвоюється змінній x. Після виконання цієї команди значення змінної x стає рівним обчисленому значенню виразу e.**

**Наприклад:**

**- `x := 5` - присвоїти змінній x значення 5**

**- `y := x + 3` - присвоїти змінній y значення, отримане додаванням поточного значення x і 3**

**- `a := b \* c` - присвоїти змінній a добуток значень змінних b і c**

**\*\*Кратні присвоювання\*\***

**Кратне присвоювання - це команда, яка дозволяє присвоїти значення одночасно декільком змінним. Її формальне означення:**

**(x1, x2, ..., xn) := (e1, e2, ..., en)**

**Семантика кратного присвоювання полягає в тому, що спочатку обчислюються значення виразів e1, e2, ..., en, а потім ці значення присвоюються відповідним змінним x1, x2, ..., xn.**

**Наприклад:**

**- `(x, y) := (3, 4)` - присвоїти x значення 3, а y значення 4**

**- `(a, b, c) := (x + y, y - x, x \* y)` - присвоїти a суму x + y, b різницю y - x, а c добуток x \* y**

**Кратні присвоювання часто використовуються для обміну значеннями між змінними або для розпакування кортежів або структур даних.**

**Приклади:**

**- `(x, y) := (y, x)` - обмінює значення x і y**

**- `(real, imag) := complexNumber()` - присвоює дійсну та уявну частини комплексного числа окремим змінним**

**Присвоювання та кратні присвоювання є основою для багатьох складніших конструкцій у програмуванні, таких як циклі, умовні оператори тощо.**

1. Означення присвоювання елементу масиву. Приклади.

Присвоювання елементу масиву є важливою операцією, яка дозволяє змінювати значення елементів масиву. У математичних основах програмування воно має таке формальне означення:

Нехай A - масив, i - індекс елемента масиву, а e - вираз, значення якого може бути обчислене. Тоді команда присвоювання елементу масиву записується як:

A[i] := e

Семантика цієї команди полягає в тому, що спочатку обчислюється значення виразу e, а потім це значення присвоюється елементу масиву A з індексом i. Після виконання цієї команди елемент A[i] буде містити нове значення, рівне обчисленому значенню виразу e.

Наприклад:

- `arr[0] := 10` - присвоїти першому елементу масиву arr значення 10

- `matrix[i][j] := x + y` - присвоїти елементу двовимірного масиву matrix з індексами i та j суму значень x та y

- `vector[n] := vector[n] \* 2` - подвоїти значення n-го елемента масиву vector

Присвоювання елементам масиву часто використовується в циклах для ітерації та оновлення значень елементів масиву. Наприклад:

```

for i := 0 to n-1 do

arr[i] := i \* i // Заповнити arr квадратами чисел від 0 до n-1

end for

```

Також можливі кратні присвоювання елементів масиву, наприклад:

```

(arr[0], arr[1]) := (x, y) // Присвоїти перші два елементи масиву arr значенням x та y

```

Присвоювання елементів масиву є фундаментальною операцією, яка забезпечує можливість маніпуляції даними, збереженими у масивах, і використовується в багатьох алгоритмах обробки даних.

1. Означення команди вибору. Правило обчислення. Приклади.

## Означення команди вибору

\*\*Команда вибору\*\* (також відома як \*\*умовний оператор\*\* або \*\*тернарний оператор) - це оператор програмування, який використовується для вибору одного з двох значень на основі значення булевого виразу.\*\*

\*\*Синтаксис:\*\*

```python

значення = вираз if умова else інше\_значення

```

\*\*Де:\*\*

\* \*\*значення:\*\* Змінна, яка буде містити результат команди вибору.

\* \*\*вираз:\*\* Булевий вираз, який визначає, яке значення буде обрано.

\* \*\*умова:\*\* Значення, яке буде обрано, якщо \*\*вираз\*\* істинний.

\* \*\*інше\_значення:\*\* Значення, яке буде обрано, якщо \*\*вираз\*\* хибний.

## Правило обчислення

1. Обчислити значення \*\*виразу\*\*.

2. Якщо \*\*вираз\*\* істинний, то результатом команди вибору буде \*\*умова\*\*.

3. Якщо \*\*вираз\*\* хибний, то результатом команди вибору буде \*\*інше\_значення\*\*.

## Приклади

\*\*1. Присвоєння значення змінній на основі булевого значення:\*\*

```python

число = 10

результат = "позитивне" if число > 0 else "негативне"

print(результат) # Виведе "позитивне"

```

\*\*2. Вибір одного з двох рядків:\*\*

```python

вік = 18

повідомлення = "Ви можете проголосувати" if вік >= 18 else "Ви не можете проголосувати"

print(повідомлення) # Виведе "Ви можете проголосувати"

```

\*\*3. Виконання одного з двох блоків коду:\*\*

```python

число = 5

if число % 2 == 0:

print("Число парне")

else:

print("Число непарне")

```

\*\*4. Вкладені команди вибору:\*\*

```python

оцінка = 90

результат = "відмінно" if оцінка >= 90 else "добре" if оцінка >= 80 else "задовільно"

print(результат) # Виведе "відмінно"

```

1. Теорема про команду вибору. Доведення. Приклад застосування.

Изображение выглядит как текст, снимок экрана, Шрифт, документ

Автоматически созданное описание

Розглянемо ситуацію, де є n лотерейних квитків, кожен з яких має певну кількість виграшів. Ми хочемо вибрати по одному виграшу з кожного квитка так, щоб максимізувати загальну кількість виграшів. Застосування теореми про команду вибору допоможе гарантувати, що ми можемо вибрати виграш з кожного квитка таким чином, щоб уникнути пропускання жодного з них.

1. Означення команди циклу. Правило обчислення. Приклади.

Команда циклу є однією з основних керуючих конструкцій у програмуванні. Вона дозволяє повторювати виконання певної послідовності команд (тіла циклу) доки виконується певна умова. У математичних основах програмування команда циклу має таке формальне означення:

Нехай B - логічний вираз (умова циклу), а S - послідовність команд (тіло циклу). Тоді команда циклу записується як:

while B do

S

end while

Семантика (правило обчислення) цієї команди полягає в наступному:

1. Спочатку обчислюється значення логічного виразу B.

2. Якщо B істинне, то виконується послідовність команд S (тіло циклу), після чого управління передається на крок 1 (умова циклу перевіряється знову).

3. Якщо B хибне, то виконання циклу завершується, і управління передається на наступну команду після циклу.

Цикл повторюється доки умова B істинна. Як тільки B стає хибною, цикл завершується.

Приклади:

1. Обчислення суми елементів масиву:

```

sum := 0

i := 0

while i < n do

sum := sum + arr[i]

i := i + 1

end while

```

2. Пошук максимального елемента в масиві:

```

max := arr[0]

i := 1

while i < n do

if arr[i] > max then

max := arr[i]

end if

i := i + 1

end while

```

3. Обчислення факторіалу числа:

```

n := 5

factorial := 1

i := 2

while i <= n do

factorial := factorial \* i

i := i + 1

end while

```

Команда циклу є потужною конструкцією, яка дозволяє реалізувати багато різноманітних алгоритмів. Однак, слід бути обережними з нескінченними циклами, коли умова циклу ніколи не стає хибною, що призводить до зациклювання програми.

1. Теорема про команду циклу. Леми 1, 2. Доведення теореми.

Теорема про команду циклу є важливим результатом в теорії комбінаторики. Вона стверджує, що якщо кожна точка (або вершина) у орієнтованому графі має ступінь (кількість ребер, які виходять з цієї точки) щонайменше *d*, тоді граф має орієнтований цикл довжини принаймні *d*+1.

**Лема 1:** Якщо в орієнтованому графі є вершина ступеня менше *d*, тоді існує орієнтована вершина ступеня 0.

**Лема 2:** Якщо у орієнтованому графі кожна вершина має ступінь щонайменше *d*, тоді існує орієнтований цикл довжини принаймні *d*+1.

**Доведення теореми:**

1. **Доведення леми 1:**

Нехай *G* - орієнтований граф, існує вершина *v* зі ступенем менше *d*. Розглянемо послідовність вершин, які можна отримати з *v* шляхом переходу по ребрах. Так як ступінь *v* менше *d*, то довжина цієї послідовності не може перевищувати *d*. Оскільки у графі *G* існує скінченна кількість вершин, ця послідовність не може продовжуватися вічно. Отже, ми досягнемо вершини *u*, яка не з'являється у цій послідовності. Таким чином, ступінь *u* дорівнює 0

1. **Доведення леми 2:**

Нехай у графі *G* кожна вершина має ступінь щонайменше *d*. Розглянемо послідовність вершин, які можна отримати з будь-якої вершини *v* шляхом переходу по ребрах. Оскільки кожна вершина має ступінь щонайменше *d*, довжина цієї послідовності не може перевищувати *d*. Але згідно леми 1, якщо ми оберемо ступінь *d* для вершини *v*, тоді ми знайдемо іншу вершину *u*, до якої можна просунутися з *v*, і так далі. Оскільки у графі *G* є обмежена кількість вершин, ця послідовність не може тривати нескінченно. Отже, ми досягнемо вершини *u*′, яка з'являється вперше після вершини *v*. Таким чином, ми знаходимо цикл довжини принаймні *d*+1.

Отже, теорема про команду циклу доведена.

**Приклад застосування:**

Розглянемо орієнтований граф, де кожна вершина має ступінь щонайменше 3. Застосовуючи теорему про команду циклу, ми можемо визначити, що у цьому графі є цикл довжиною принаймні 4. Це може бути корисно при аналізі мережевих структур, де ми шукаємо елементи, які можуть утворювати цикли певної довжини.

Начало формы

1. Теорема про команду циклу. Леми 3, 4. Доведення теореми.

Теорема про команду циклу є важливою теоремою в теорії множин, яка розширює теорему про команду вибору. Вона стверджує, що якщо у нас є сукупність скінченних множин, кожна з яких має щонайменше два елементи, то ми можемо вибрати по одному елементу з кожної множини так, щоб вони не перетиналися, тобто не містили спільних елементів.

Перш ніж навести доведення теореми, розглянемо леми 3 і 4, які необхідні для цього.

Изображение выглядит как текст, снимок экрана, документ, Шрифт

Автоматически созданное описание

Отже, за принципом математичної індукції, твердження теореми про команду циклу доведено.

1. Наслідок з теореми про цикл. Приклад.

Изображение выглядит как текст, снимок экрана, Шрифт, документ

Автоматически созданное описание