ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА КІБЕРНЕТИКИ
КИЇВСЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ ІМЕНІ ТАРАСА ШЕВЧЕНКА

МАТЕМАТИЧНА ЛОГІКА ТА ПРОГРАМУВАННЯ ДОСВІД ВИКЛАДАННЯ

МОНОГРАФІЯ

Видавничий дім
«Гельветика»
2022

УДК 378.1:004.4:510.6

М34

Рецензенти:

Терещенко Василь Миколайович	доктор фізико-математичних наук, професор Київського національного університету імені Тараса Шевченка
Теленик Сергій Федорович	доктор технічних наук, професор Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського»
Романенко Тетяна Василівна	доктор педагогічних наук, доцент Черкаського національного університету імені Богдана Хмельницького

Рекомендовано до друку вченою радою
факультету комп'ютерних наук та кібернетики
Київського національного університету імені Тараса Шевченка
(протокол № 6 від 25 січня 2022 року)

М34 Математична логіка та програмування. Досвід викладання : Колективна монографія. – Одеса : Видавничий дім «Гельветика», 2022. – 212 с.
ISBN

У монографії висвітлюються теоретичні та практичні аспекти підготовки фахівців у галузі інформаційних технологій у сучасних умовах. Розглянуто інтеграційні підходи до вивчення математичної логіки, програмування та споріднених до них дисциплін, які викладаються на кафедрі теорії та технології програмування Київського національного університету імені Тараса Шевченка.

Матеріали колективної монографії можуть бути використані в освітньому процесі при підготовці здобувачів першого та другого рівнів вищої освіти за спеціальністю 122 «Комп'ютерні науки». Монографія може зацікавити науковців, викладачів, аспірантів, студентів закладів вищої освіти.

ISBN

ЗМІСТ

ВСТУП

Швидкий плин сучасних інформаційних технологій безпосередньо має вплив на ринок праці майбутніх IT-фахівців.

В монографії «Математична логіка та програмування. Досвід викладання» розглянуто питання організації освітньої діяльності майбутніх фахівців галузі інформаційних технологій. В колективну монографію ввійшли статті, подані на семінарах кафедри теорії та технології програмування Київського національного університету імені Тараса Шевченка та семінарах Українського логічного товариства.

Зокрема, в першому розділі: описується трирівневий підхід, який розглядає основні поняття логіки на трьох рівнях: методологічному (філософському), професійному (науковому) та формальному (математичному), що дозволяє продемонструвати розвиток основних понять логіки, специфікувати їх для кожного рівня розгляду, та вказати зв'язки понять різних рівнів; розглядається концепція викладання навчальної дисципліни «Математична логіка» на основі спільного для логіки й програмування композиційно-номінативного підходу, де особливістю викладання є семантико-синтаксична спрямованість матеріалу; пропонується новий підхід до експлікації поняття алгоритму, який базується на уточненні поняття обчислювальної процедури та звуженні його до поняття алгоритму.

Другий розділ охоплює кілька методів перевірки програмного забезпечення, який вивчається в курсі «Верифікація та валідація програмних систем», зокрема, формальних методів доведення правильності програм; досліджує можливість уточнення та класифікації змінних в імперативних програмах, базуючись на понятті іменованих елементів та виокремленні двох різних рівнів абстракції; визначено основні архітектурні засади та надано основні фрагменти програмної реалізації web-фреймворку Miniature; проведено порівняльний аналіз з деякими промисловими потужними web-фреймворками; описано концепцію викладання обов'язкової навчальної дисципліни «Об'єктно-орієнтоване програмування» для студентів факультету комп'ютерних наук та кібернетики, що навчаються на першому (бакалаврському) рівні вищої освіти за освітньо-професійною програмою «Інформатика»; присвячено огляду основних факторів, які впливають на формування сучасного ринку мобільної розробки та зосереджено увагу на змісті навчальних курсів з мобільної розробки, на

прикладі, викладання курсу «Розробка ПЗ під мобільні платформи» в Київському національному університеті імені Тараса Шевченка.

В третьому розділі монографії представлено порівняльний аналіз обов'язкових дисциплін освітньо-професійної програми «Інформатика» першого (бакалаврського) рівня вищої освіти галузі знань 12 «Інформаційні технології», спеціальності 122 «Комп'ютерні науки», викладання яких забезпечується кафедрою теорії та технології програмування факультету комп'ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка з дисциплінами освітньо-професійних програм того ж рівня та спеціальності інших закладів вищої освіти; розглянуто методичні питання впровадження міждисциплінарних зв'язків у навчальний процес закладів вищої освіти, зокрема інтеграційному підходу при вивченні дисциплін «Методи специфікації програм», «Коректність програм та логіки програмування» та «Інструментальні середовища та технології програмування» галузі інформаційних технологій.

Колектив авторів сподівається, що матеріали монографії можуть бути використані в освітньому процесі при підготовці фахівців галузі знань 12 «Інформаційні технології», спеціальності 122 «Комп'ютерні науки» та буде корисними для науковців, викладачів, аспірантів й студентів закладів вищої освіти.

РОЗДІЛ 1. ТЕОРЕТИЧНІ АСПЕКТИ ВИКЛАДАННЯ МАТЕМАТИЧНОЇ ЛОГІКИ

Нікітченко Микола Степанович, д.ф.-м.н, професор

ТРИРІВНЕВА СХЕМА ЕКСПЛІКАЦІЇ БАЗОВИХ ПОНЯТЬ У ВИКЛАДАННІ КУРСУ «МАТЕМАТИЧНА ЛОГІКА»

У роботі пропонується викладати курс «Математична логіка» на основі трирівневого підходу, який розглядає основні поняття логіки на трьох рівнях: методологічному (філософському), професійному (науковому) та формальному (математичному). На першому рівні логіка специфікується як діалектика (теорія розвитку), на другому рівні – як логіка науки (загальної або конкретної), і на третьому рівні – як математична логіка. Це дозволяє продемонструвати розвиток основних понять логіки, специфікувати їх для кожного рівня розгляду, та вказати зв'язки понять різних рівнів.

Ключові слова: логіка, математична логіка, теоретико-функціональний підхід, викладання логіки

We proposes to teach the course "Mathematical Logic" on the basis of a three-level approach, which considers the basic concepts of logic at three levels: methodological (philosophical), professional (scientific) and formal (mathematical). At the first level, logic is specified as dialectics (theory of development), at the second level – as the logic of science (general or applied logic), and at the third level - as mathematical logic. This allows to demonstrate the development of basic concepts of logic, to specify them for each level of consideration, and to indicate the connections of concepts between different levels.

Keywords: logic, mathematical logic, function-theoretic approach, teaching logic

Викладання курсу «Математична логіка» для студентів галузі «Інформаційні технології» викликає занепокоєння. Це пов'язано із змінами, які відбулись за останні роки у тріаді «студент – зміст курсу – викладач».

Почнемо із студентів. Постійна та швидка зміна інформаційних технологій та нових методів отримання інформації, панування «кліпового» та утилітарного мислення, падіння інтересу до теоретичних знань, зниження рівня математичної підготовки абітурієнтів ускладнюють розуміння студентами основ математичної логіки. Що стосується викладачів, то умовно їх можна поділити на дві групи: у першу попадають досвідчені викладачі з ґрунтовною математичною підготовкою, але слабким рівнем знань сучасних інформаційних технологій та використанням математичної логіки у штучному інтелекті і розробці програмного забезпечення; у другу попадають більш молоді викладачі із знанням інформаційних технологій, але із недостатньою математичною підготовкою. Така ситуація не є дивною, тому що в Україні фактично немає кафедр з математичної логіки, які б готували високо професіональних спеціалістів. Нарешті, така складова тріади, як зміст курсу, також не відповідає сучасним вимогам. Справа в тому, що основи математичної логіки було закладено у середині 20-го століття, коли основна увага приділялась суто теоретичним питанням опису аксіоматичних систем (теорія доведень) та теоретичним питанням істинності (теорія моделей). Широке застосування логіки ставить більш гостро проблеми орієнтації логіки на прикладні задачі та на інтеграцію знань, орієнтованих на різні предметні області. Хоча наразі видано достатньо багато посібників, які орієнтовані на викладання математичної логіки для студентів галузі «Інформаційні технології», вони як і раніше, переважно обмежуються в теоретичній складовій викладанням пропозиційної логіки, логіки першого порядку та інколи темпоральної логіки, а в практичній складовій – розглядом певних застосувань логіки; типовим прикладом є підручник [1]. Такий підхід є фрагментарним та не дозволяє сформувати цілісне розуміння логіки.

Серед труднощів, з якими зустрічаються студенти при вивченні теоретичних дисциплін, та, зокрема, математичної логіки, виокремимо наступні:

1) недостатня методологічна підготовка;
2) розпливчате розуміння загальних категорій та професійних понять;

3) слабкі навички експлікації та формалізації знань.

Надамо деякі пояснення щодо вищеназваних труднощів.

Почнемо з методологічних питань. Тут найважливішим методом *(принципом) є сходження від абстрактного до конкретного* [2]. Застосування цього методу вимагає чіткого розуміння категорій *абстрактного* та *конкретного*. На жаль, студенти часто тлумачать ці категорії як *уявне (ідеальне)* та *реальне*. Таке тлумачення підтримується і словниками. Але в даному випадку ці терміни слід розуміти відповідно як *одностороннє* та *єдність багатостороннього*. В комп'ютерних науках цей метод (в обмеженому варіанті) є пануючим методом розроблення програмного забезпечення. Тому, зокрема, в освітніх програмах спеціальностей галузі «Інформаційні технології» перша загальна компетентність сформульована як «здатність до абстрактного мислення, аналізу та синтезу» [3]. Опанування цією компетентністю вимагає розкриття не лише категорій *абстрактного* та *конкретного*, але і категорій, пов'язаних з аналізом та синтезом, що наразі є слабким місцем в освітньому процесі.

Продемонструємо тепер труднощі у визначенні професійних понять. Розглянемо, наприклад, визначення формальної граматики, яке зазвичай має такий вигляд: *породжувальною граматикою* називається четвірка $G = (N, T, P, S)$, де N та T – *скінчені алфавіти відповідно нетермінальних та термінальних символів* $(N \cap T = \varnothing)$, $P \subset (N \bigcup T)^* N (N \bigcup T)^* \times (N \bigcup T)^*$ – *скінчена множина продукцій (правил),* S – *початковий символ* $(S \in N)$.

Неважко зрозуміти, що вказана четвірка $G = (N, T, P, S)$ насправді не є повним визначення породжувальної граматики – тут про породження не сказано жодного слова. Наведені чотири множини задають лише певні індивідуальні параметри конкретної граматики; в цьому сенсі їх можна назвати *екстенсіональними* характеристиками граматики. Що ж стосується загальних характеристик породжувальної граматики, а саме механізму породження та визначення мови, що породжується граматикою, то вони визначаються окремо від визначення самої граматики. Такі загальні характеристики будемо називати *інтенсіональними*. Таким чином, наведене визначення граматики є частковим, є екстенсіональним, і тому по своїй суті є невірним, бо ставить наголос на другорядних екстенсіональних аспектах,

ігноруючи більш важливіші інтенсіональні аспекти. Такі ж самі зауваження можна сформулювати відносно інших понять, наприклад, машин Тюрінга.

Нарешті, розглянемо труднощі математичної формалізації понять комп'ютерних наук. Зазвичай, під математичною формалізацію розуміють теоретико-множинну формалізацію. Така формалізація є пануючою на сучасному етапі розвитку комп'ютерних наук. Але наскільки така формалізація є адекватною? Для комп'ютерних наук найважливішим поняттям є поняття функції. Зазвичай, під функцією розуміють бінарне відношення між множинами вхідних та вихідних даних [4]. Таке зведення багатоаспектного поняття функції до бінарного відношення (до реляції, до графіка функції) відкидає інші аспекти функцій, які є важливими при створенні програмних систем, наприклад, такі як декомпозиція та композиція функцій. Але навіть таке реляційне зведення не є адекватним для складних функцій. Продемонструємо це на прикладі наступної *машини Тюрінга M* (використовуємо позначення з [5]):

$$M=(Q, A, \#, \delta, q_0, F),$$

де:

- $Q =\{ q_0, q_1, q_2\}$– множина станів;
- $A =\{1, \#\}$ – алфавіт;
- # – порожній символ;
- $\delta =\{ q_0 1 \rightarrow q_0 \# R,\ q_0 \# \rightarrow q_1 \#,\ q_1 \# \rightarrow q_1 1 R,\ \ q_1 \# \rightarrow q_2 \#\}$ – множина команд;
- q_0 – початковий стан;
- $F =\{ q_2\}$ – множина фінальних станів.

На перший погляд здається, що ця недетермінована машина породжує функцію, яка кожному натуральному числу ставить у відповідність довільне натуральне число. Іншими словами, графіком цієї функції є повне бінарне відношення на множині натуральних чисел *Nat*. Але це не враховує випадку, коли машина Тюрінга не завершує свою роботу, що означає невизначеність значення функції. Звичайно, можна спробувати побудувати іншу теоретико-множинну модель функції, яка ще буде задавати область невизначеності. Але, як відомо з теорії алгоритмів, область

невизначеності частково рекурсивної функції може бути не рекурсивно перелічною. Тому теоретико-множинні тлумачення функцій не є адекватними.

Можна наводити ще багато прикладів, пов’язаних з опануванням основ логіки та математичної логіки зокрема. Виокремимо тут ще одну складність: студенти знають, що є різні логіки, зокрема, діалектична логіка, традиційна логіка та математична логіка. Але про їх зв'язок їм важко щось сказати, і це ускладнює розуміння єдності основних понять логіки.

Яким же чином можна спробувати покращити ситуацію? Ми пропонуємо трирівневий підхід до викладання математичної логіки. Цей підхід полягає в тому, що виокремлюються три рівні розгляду логіки: методологічний (філософський), професійний (науковий) та формальний (математичний). На філософському рівні логіка специфікується як діалектика (теорія розвитку), на другому рівні – як логіка конкретних наук (прикладні логіки), і на третьому рівні – як математична логіка.

Спеціалісти галузі інформаційних технологій переважно опікуються професійним та формальним рівнем і недостатньо уваги приділяють методологічному рівню. Хоча треба зазначити, що важливість методологічного (філософського) рівня визнається навіть на державному рівні. Так, у постанові Кабінету Міністрів України № 261 від 23 березня 2016 року відзначається, що здобувачі ступеня доктора філософії повинні оволодіти «загальнонауковими (філософськими) компетентностями, спрямованими на формування системного наукового світогляду, професійної етики та загального культурного кругозору» [6].

Слід лише додати, що оволодіння загальнонауковими (філософськими) компетентностями необхідно не лише докторам філософії, але й бакалаврам і магістрам (хоча б на початковому рівні).

Перейдемо тепер до короткого опису трьох вказаних рівнів, орієнтованого на розкриття основних понять логіки та математичної логіки зокрема.

Методологічний (філософський) рівень визначення логіки

Головними категоріями методологічного рівня є категорії абстрактного та конкретного. Але, як вже відзначалось раніше, словникові

визначення не дають потрібного нам визначення цих термінів. Дійсно, у словниках читаємо: абстрактний – який виник у результаті абстрагування, відірваний від дійсності, не матеріальний, уявний, існуючий як ідея, протилежне конкретний; конкретний – який існує реально, цілком точний, предметно визначений, протилежне абстрактний. Таким чином, основним словниковим тлумаченням абстрактного є уявне, ідеальне, а конкретного – реальне. Таке тлумачення будемо називати побутовим. Але нам потрібне не побутове, а наукове тлумачення, коли абстрактне та конкретне розглядаються як філософські категорії. У цьому випадку абстрактне є ОДНОСТОРОННЄ (отримане в результаті абстрагування), а конкретне тепер тлумачиться не як реальне, а як ЄДНІСТЬ БАГАТОСТОРОННЬОГО. Наведемо простий приклад.

Розглянемо таку формулу пропозиційної логіки: $\varphi \vee \psi \wedge \varphi$. Перше, що ми відзначаємо, це її зовнішній, синтаксичний вигляд. Отже, перша абстракція – синтаксична. Але цієї абстракції не вистачає для розуміння формули – потрібно ще сказати, яке її значення. Традиційно, у якості значення пропозиційної формули береться певна булева функція. Це є друга абстракція. Але ці дві ізольовані абстракції ще не дають адекватного розуміння обраної формули – необхідна єдність цих двох абстракцій. Така єдність задається відображенням інтерпретації формул у значення.

Таким чином, конкретне, як єдність багатостороннього, визначається такою тріадою:

одностороннє (*абстрактне*) –

– *багатостороннє* (*багато абстракцій*) –

– *єдність багатостороннього* (*конкретне*).

Зауважимо, що ця тріада побудована згідно закону заперечення заперечення [2]. Дійсно, один – це теза; багато – це антитеза, яка є запереченням одного (але яке зберігає одиниці у собі); єдність багатьох – це синтез, який є запереченням багатьох із збереженням їх в одному.

Переходимо тепер до опису основних принципів методологічного рівня.

Приймаємо схему, запропоновану Г.В.Ф. Гегелем [2] та його послідовниками [7, 8]. Термін поняття тлумачимо у загальному сенсі, яке включає як філософські поняття (категорії), так і наукові та математичні поняття.

Ми виокремлюємо такі принципи [9].

Принцип розвитку від абстрактного до конкретного: розвиток – це безумовно орієнтована зміна поняття від абстрактного до конкретного (від простого до складного, від нижчого рівня до вищого, від старого до нового).

Тріадичний принцип розвитку: однією з основних схем розвитку є тріада: *теза – антитеза – синтез*. Інші схеми також застосовуються, але достатньо рідко.

Принцип єдності теорії та практики: теорію та практику слід розглядати як взаємообумовлені поняття. Цей принцип обґрунтовує розвиток понять у праксеологічній перспективі, тобто цей розвиток має базуватися на аналізі дій людини над об'єктами. Праксеологічний аспект є одним з головних філософських аспектів, що пов'язують категорії суб'єкта та об'єкта. Можна сказати, що цей аспект є базовим, на якому розвиваються онтологічні, гносеологічні та аксіологічні аспекти.

Принцип єдності трьох моментів у визначенні поняття: поняття мають бути визначені у єдності трьох їх моментів: *всезагального* (*універсального*), *особливого* та *одиничного (індивідуального)*. Тут універсальний момент є провідним, йому підпорядковуються особливі та індивідуальні моменти. Як наслідок, потрібно, щоб основна увага приділялася розкриттю універсальних моментів понять та формулюванню чітких відмінностей між цими трьома моментами.

Виходячи з цих принципів, ми тепер можемо виділити поняття (категорії), пов'язані з логікою на філософському рівні. Будемо спиратись на категорії, розроблені в діалектиці (в теорії розвитку) [2, 7, 8].

Почнемо з антропологічного підходу. Це означає, що ми зосереджуємось на людині як на початковому пункті наших досліджень. Виникає закономірне запитання: яка найголовніша характеристика людини? Аналіз різних сторін людини дозволяє стверджувати, що найважливішим для її оцінки є послідовність її дій (її діяльність). Цей висновок підтверджується в різних ситуаціях і формулюється багатьма авторами. Скажімо, для Кожева, як і для Гегеля, «справжнє буття людини – це її діяльність» [10]. Ця теза передбачає такі три моменти: 1) людина має бути активною, діяльною; 2) її діяльність має бути продуманою (раціональною,

розумною, осмисленою); 3) діяльність людини має бути плідною (логічною, продуктивною, ефективною), в тому сенсі, що вона дає бажані результати.

Таким чином, ми виділили три категорії: ДІЯЛЬНІСТЬ, МИСЛЕННЯ та ЛОГІКА. Як вони пов'язані? Ми можемо сказати, що МИСЛЕННЯ – це знята ДІЯЛЬНІСТЬ, а ЛОГІКА – зняте МИСЛЕННЯ. (Тут зняття є філософською категорією, яка означає заперечення, але із збереженням головного.) Більше того, ці три категорії складають *тріаду розвитку логіки*, в якій ДІЯЛЬНІСТЬ – це теза, МИСЛЕННЯ – антитеза, а ЛОГІКА – синтез. Тому ЛОГІКА визначає єдність ДІЯЛЬНОСТІ та МИСЛЕННЯ. Міра їх єдності називається відношенням ІСТИНОСТІ.

В традиційних формулюваннях можна сказати, що *об'єктом* логіки є *мислення*, а *предметом – відношення істинності*.

Тріада розвитку логіки задає основну схему визначення логіки. Але в цій тріаді категорія ДІЯЛЬНІСТЬ не специфікована. Робимо це за допомогою тріади Фіхте: СУБ'ЄКТ – ОБ'ЄКТ – ДІЯЛЬНІСТЬ. Це означає, що ДІЯЛЬНІСТЬ інтегрує категорії СУБ'ЄКТ і ОБ'ЄКТ.

Підводячи підсумок, отримуємо *пентаду розвитку логіки*:

СУБ'ЄКТ – ОБ'ЄКТ – ДІЯЛЬНІСТЬ – МИСЛЕННЯ – ЛОГІКА.

У цій пентаді категорія ДІЯЛЬНОСТІ відіграє центральну роль. Тому запропонований підхід до визначення логіки можна назвати *діяльнісним*. Подальше пояснення вищезазначених категорій призводить до визначення логіки на науковому та математичному рівнях.

Професійний (науковий) рівень визначення логіки

Перехід від філософського до наукового рівня можна розглядати як побудову проекцій нескінченних філософських категорій на скінчені (у філософському сенсі) наукові поняття. Тому ми називаємо такі проєкції фінітизаціями. Вони мають бути орієнтовані на прикладну область, яка досліджується на обраному науковому рівні. Такі поняття утворюють основу прикладних (наукових) логік.

Почнемо з категорій УНІВЕРСАЛЬНИЙ, ОСОБЛИВИЙ та ОДИНИЧНИЙ (ІНДИВІДУАЛЬНИЙ). Їхні фінітізації називаються відповідно *інтенсіональністю*, *особливою інтенсіональністю* та

екстенсіональністю. Ця термінологія індукована принципом екстенсіональності в теорії множин. Переформулюючи принцип єдності трьох моментів філософського рівня, ми отримуємо *принцип єдності у визначенні понять на науковому рівні*: наукові поняття мають бути визначені в єдності їх трьох моментів: інтенсіональності, особливої інтенсіональності та екстенсіональності, які відображають специфіку предметної області.

Тепер постає питання: яку фінітизацію категорії ДІЯЛЬНІСТЬ на науковому рівні можна запропонувати? Беручи до уваги, що така фінітизація має бути логічно-орієнтованою, ми припускаємо, що це є діяльність із *вибору/класифікації об'єктів*. Забігаючи наперед, варто зазначити, що діяльність з вибору призведе до двовалентної (двозначної) логіки, тоді як діяльність з класифікації призведе до багатозначної логіки. Тим не менш, необхідне подальше уточнення сформульованої діяльності.

Ми робимо таке уточнення, орієнтуючись на рівні розгляду об'єктів. За схемою, запропонованою Гегелем, ми можемо розглядати об'єкти (буття) на різних рівнях, зокрема, щодо логіки, нас цікавлять рівні *наявного*, *реального* та *дійсного* буття [2, 8].

На рівні наявного буття ми зосереджуємось на категорії ЯКІСТЬ; на рівні реального буття ми концентруємось на категоріях РІЧЬ та ВЛАСТИВІСТЬ; а на рівні дійсного буття ми визначаємо категорії ДІЯ і РЕАКЦІЯ, ПРИЧИНА і НАСЛІДОК.

Фінітизація вищезазначених категорій призводить до трьох типів моделей об'єктів, подальше уточнення яаих дає три типи математичної логіки: пропозиційна, предикатна та програмна логіки. Важливо підкреслити, що в цьому випадку ми вказали пряму відповідність цих логік рівням буття та відповідних категорій.

В рамках цього дослідження розглянемо лише зв'язок пропозиційної логіки з категорією ЯКІСТЬ.

Як відомо [2], наявний рівень буття (об'єкта, предмета) характеризується категоріями ЯКІСТЬ, КІЛЬКІСТЬ та МІРА. КІЛЬКІСТЬ є знятою ЯКІСТЮ, МІРА є єдністю ЯКОСТІ та КІЛЬКОСТІ. Якості предмета є те, без чого предмет перестає бути сам собою. На цьому рівні якості незалежні одна від одної. Це дозволяє характеризувати предмети за

допомогою набору їх якостей. Тому головною (логічною) діяльністю рівня наявного буття є *вибір предметів за їх якостями*. Подальша побудова формальної моделі цієї діяльності призводить до визначення пропозиційної логіки (дивись наступний розділ).

Формальний (математичний) рівень визначення логіки

Традиційно, основні поняття логіки та комп'ютерних наук формалізуються на основі теоретико-множинного підходу. Такий підхід дає багато переваг у формалізації та вивченні таких понять. Його головною особливістю є орієнтація на екстенсіональні аспекти понять, що визначається аксіомою екстенсіональні теорії множин. Але подальший розвиток логіки, штучного інтелекту та комп'ютерних наук вимагає залучення інтенсіональних аспектів. І першим з них є діяльнісний аспект, який переважно уточнюється за допомогою поняття функції. Треба відмітити, що інтуїтивне поняття функції має багато аспектів, які не завжди можна уточнити в рамках теоретико-множинного відходу. Тому ми виступаємо за *принцип теоретико-функціональної формалізації* основних понять логіки, штучного інтелекту та комп'ютерних наук. Звичайно, в цьому випадку ми розглядаємо поняття функції в єдності її інтенсіонального та екстенсіонального аспектів. Щодо поняття множини, його також слід розглядати у такій єдності; тому поняття функції та множини не зводяться одне до одного, а швидше доповнюють одне одне.

Ми формалізуємо поняття функції, використовуючи принципи *номінативності* та *композиційності*.

Принцип номінативності підкреслює важливість відношень іменування в побудові об'єктних моделей. Принцип композиційності стверджує, що властивості складних систем (програмних моделей) визначаються властивостями їх компонентів і структурою системи. Номінативність виводиться як фінітизація категорій РІЧ, ФОРМА та ЗМІСТ, а композиційність – як фінітизація категорій ЧАСТИНА та ЦІЛЕ. У цьому випадку категорія РІЧ моделюється за допомогою номінативних даних, а категорія ДІЯ – за допомогою номінативних функцій [11]. Таким чином, дії можна формалізувати двома частковими алгебрами: алгеброю даних і алгеброю функцій [12, 13].

Запропоновані формальні моделі програм (як фінітизація категорії ДІЯ) тепер можна поширити на формальну логіку програм. Розглядаючи дії (програми) як перетворення даних, ми представляємо вхідні та вихідні дані як номінативні дані. Подаючи класи даних за допомогою предикатів, ми отримуємо трійки Хоара: (передумова, програма, післяумова). Отримані логіки називаються *композиційно-номінативними програмними логіками*. Їх формальні визначення подані в [12, 13]. Логіки предикатів розглядаємо як часткові випадки логіки програм (дій). Зараз продемонструємо сказане лише на прикладі пропозиційної логіки.

Як було відзначено раніше, пропозиційна логіка фактично є логікою рівня наявного буття, яка визначається категорією ЯКОСТІ.

Як же побудувати формальну (математичну) модель такої логіки? Повертаємось до діяльності на цьому рівні: *вибір об'єктів за їх якостями*.

Якщо клас об'єктів позначити O, а потрібні характеристики об'єктів – q, то тоді задача діяльності полягає в виборі (виділенні) тих об'єктів з O, які мають характеристики q. Зауважимо, що у випадку теоретико-множинного підходу O розглядається як множина, а q – як властивість, тоді задача вибору може бути записана аксіомою виділення: існує множина $\{o \mid o \in O,\ q(o)\}$. Але у нашому випадку ми не можемо застосовувати теоретико-множинний підхід, тому що ми до об'єктів висуваємо більш слабкі вимоги, ніж теорія множин до елементів. Наприклад, ми не маємо відношення належності та рівності. Тому продовжимо формалізацію, спираючись на інтуїтивне розуміння якостей об'єктів.

Спочатку потрібно надати чіткий опис якостей, за котрими здійснюється відбір. Позначимо їх клас як Q. Наявність якості у об'єкта позначаємо 1, а її відсутність – 0.

Далі, треба окреслити клас об'єктів O, які розглядаються, бо цілком можливо, що не всі якості з Q можуть бути у об'єктах цього класу. Обмеження класу називаються *онтологічними (буттєвими) припущеннями*.

Розглянемо простий приклад. Треба вибрати електролампочку, яка замінить перегорілу лампочку з цоколем E14. Якості лампочок такі: цоколь E14, цоколь E27, цоколь G13, лампочка розжарювання, світлодіодна лампочка, розмір 100 мм, розмір 600 мм, потужність 12W, потужність 24W.

Щоб скоротити запис введемо відповідно такі позначення якостей: E14, E27, G13, ILB, LED, 100mm, 600mm, 12W, 24W. Для вказаного класу об'єктів (електричних лампочок) несумісними є, наприклад, якості E14, E27, G13; у кожного об'єкта може бути не більше однієї з цих якостей. Це обмеження є одним з онтологічних припущень.

Нам також потрібно вказати на *гносеологічні (пізнавальні) припущення*. Такі припущення у нашому випадка стосуються розпізнавання якостей. Основні можливі випадки (у порядку посилення розпізнавання) наступні:

- можна розпізнати наявність у предмета певних якостей, але нічого не можна сказати про наявність чи відсутність інших якостей;
- можна розпізнати не тільки наявність у предмета певних якостей, але і відсутність якихось інших якостей, а про якості, які залишились, сказати нічого не можна;
- можна розпізнавати наявність або відсутність будь якої якості.

Стосовно нашого прикладу у першому випадку ми беремо лампочки і пробуємо розпізнати наявність якості E14. Тоді математичною моделлю такої лампочки буде $[\mathrm{E14} \mapsto 1]$. Це означає, що ми вміємо відбирати лампочки з якістю E14.

У другому випадку можлива така модель: $[\mathrm{E14} \mapsto 1, \mathrm{24W} \mapsto 1, \mathrm{ILB} \mapsto 0]$. Це означає, що ми вміємо відбираємо лампочки з якостями E14 та 24W, але без якості ILB.

У третьому випадку прописуємо наявність чи відсутність усіх якостей.

Таким чином, математичною моделлю довільного об'єкту є певне дане, яке є частковою функцією з класу $Q \to \{1, 0\}$. Такі дані називаються *номінативними*, бо засновані на класі імен *Q*. Тоді найпростішою моделлю класу об'єктів є деяка сукупність номінативних даних. Втім, таке подання досить складне з точки зору діяльності з таким описом, бо вимагає окремого опису кожного індивідуального об'єкта. Існують різні рецепти подолання такого роду труднощів, але головним тут є метод *композиції/декомпозиції*: складніший опис будується з більш простих за допомогою операцій, які називаються *композиціями*.

Структура номінативного даного «підказує» вибір композицій. Дійсно, повернемось до прикладу $[\mathrm{E14} \mapsto 1, \mathrm{24W} \mapsto 1, \mathrm{ILB} \mapsto 0]$. Описуючи це дане ми вживаємо сполучник «та» та прийменник «без». Це дозволяє ввести композиції кон'юнкції $\wedge$ та заперечення $\neg$, які засновані відповідно на бінарній булевій функції кон'юнкції та унарній функції заперечення. Тоді наше номінативне дане індукує таку характеристичну функцію вибору: $\mathrm{E14} \wedge \mathrm{24W} \wedge \neg \mathrm{ILB}$, де імена якостей E14, 24W, ILB тепер задають пропозиційні функції (пропозиційні предикати) типу $(Q \to \{1,0\}) \to \{1,0\}$, тобто відображують номінативні дані у множину $\{1,0\}$. Композиції є операціями на класі $(Q \to \{1,0\}) \to \{1,0\}$. Це означає, що ми отримали алгебру пропозиційних функцій. Зауважимо, що з методологічної точки зору ми здійснили перехід від одиничних об'єктів до загального (їх класів) з виокремленням особливого, яке характеризує клас.

Таким чином, ми побудували математичну модель об'єктів та їх класів пропозиційного рівня. Далі розбудова пропозиційної логіки іде звичайним чином [12].

Висновки. Сучасний стан викладання курсу математичної логіки вимагає нових підходів та методів. Для покращення ситуації ми пропонуємо викладати цей курс на основі трирівневого підходу, який розглядає основні поняття логіки на трьох рівнях: методологічному (філософському), професійному (науковому) та формальному (математичному). На філософському рівні логіка специфікується як діалектика (теорія розвитку), на другому рівні – як логіка конкретної науки (прикладна логіка), і на третьому рівні – як математична логіка. Це дозволяє продемонструвати розвиток основних понять логіки, специфікувати їх для кожного рівня розгляду, та вказати зв'язки понять на різних рівнях. Така схема викладання логіки надає студентам знання про основні філософські категорії та методи дослідження, які можуть використовуватись у будь-яких прикладних областях, про роль логіки у різних науках і її застосування та про її формалізацію математичними методами.

Підхід потребує деталізації та розроблення відповідної методики викладання. Наступні кроки плануємо описати у подальших роботах.

Список використаних джерел

1. Mordechai Ben-Ari. Mathematical Logic for Computer Science, Springer-Verlag London, 2012.
2. Г.В.Ф. Гегель. Наука логики. – Спб.– 1997.
3. Освітньо-професійна програма «Інформатика» за спеціальністю 122 “Комп’ютерні науки” (ф-т КНК, КНУ імені Тараса Шевченка).– 2019.– Режим доступу: http://csc.knu.ua/media/filer_public/06/cd/06cdfecd-f915-4240-8238-39ae555f1d64/bac_122_inf_2019.pdf
4. Бурбаки Н. Теория множеств, Либроком, 2010.
5. Нікітченко М. С. Теорія програмування. Ч. 1. / М. С. Нікітченко. – Ніжин, 2010.
6. Постанова Кабінету Міністрів України № 261 від 23 березня 2016 року. – 2016.– Режим доступу: https://zakon.rada.gov.ua/laws/show/261-2016-%D0%BF#Text
7. Босенко В. А. Всеобщая теория развития: наук. вид. / В. А. Босенко. – Киев: 2001. – 468 с.
8. Гаврилюк В.А. Методологические основоположения системы категорий развития.– Киев: Издательство Европейского университета, 2009.– 176 с.
9. Nikitchenko M. Gnoseology-based Approach to Foundations of Informatics. In Proceedings of the ICTERI-2011, CEUR Workshop Proceedings, vol. 716, (2011), pp. 27–40.
10. Filoni M. “Man is action, not being” Hegel contra Heidegger in an unpublished essay by Kojève. Philosophical Inquiries, vol. 8, No. 2 (2020), pp. 203-208.
11. Nikitchenko M. Towards Defining Program Logics via Three-level Scheme, in Proc. MFOI-2020, Kyiv: Interservice, 2021, pp. 324-329.
12. Нікітченко М. С. Математична логіка та теорія алгоритмів / М. С. Нікітченко, С. С. Шкільняк. – К. : ВПЦ "Київський університет", 2008.
13. Нікітченко М. С. Прикладна логіка / М. С. Нікітченко, С. С. Шкільняк. – К. : ВПЦ "Київський університет", 2013.

Шкільняк Степан Степанович, д.ф.-м.н., професор

ПРОГРАМНО-ОБУМОВЛЕНА КОНЦЕПЦІЯ ВИКЛАДАННЯ ДИСЦИПЛІНИ «МАТЕМАТИЧНА ЛОГІКА»

Роботу присвячено концепції викладання навчальної дисципліни «Математична логіка» на основі спільного для логіки й програмування композиційно-номінативного підходу. Особливістю викладання є семантико-синтаксична спрямованість матеріалу. Спочатку описуються семантичні моделі логік, після цього будуються відповідні формально-аксіоматичні числення. Другою особливістю є розгляд логічних систем на основі розповсюджених в програмуванні часткових квазіарних відображень.

Ключові слова: навчальна дисципліна, композиційно-номінативний підхід, логіка, частковий предикат.

The work is dedicated to the conception of teaching the discipline "Mathematical Logic" based on the common for logic and programming composition nominative approach. The key feature is a semantic-syntactic style of the teaching material. First, we specify semantic models of logics, and then construct corresponding formal axiomatic calculi. Another notable feature is a consideration of logical systems based on partial quasiary mappings specific for programming.

Keywords: academic discipline, composition nominative approach, logic, partial predicate.

Математична логіка є невід'ємною компонентою програми підготовки фахівців в галузях комп'ютерних наук та інформатики. Найперше це зумовлено тим, що апарат математичної логіки є основою сучасних інформаційних та програмних систем, він належить до основних засобів моделювання різноманітних предметних областей. Стосовно загальносвітоглядного аспекту, поняття й методи математичної логіки дають обґрунтування правильності тих чи інших способів отримання істинного знання.

Математична логіка має чітко окреслене прикладне спрямування. Саме апарат математичної логіки (алгебра логіки) лежить в основі схемотехні-

ки комп'ютерів. В межах математичної логіки запропоновано перші формалізації поняття алгоритму, мови програмування базуються на тих чи інших уточненнях цього поняття. Засоби і методи математичної логіки дають змогу описувати різноманітні предметні області, моделювати інформаційні процеси і системи, проводити пошук доведень за допомогою формально-логічних числень.

Розвиток інформаційних технологій та пов'язана з цим поява нових задач і проблем ведуть до залучення для їх розв'язку все нових понять і засобів математичної логіки. Це теорія доведень, модальні логіки (в першу чергу, темпоральні та епістемічні), алгоритмічні логіки Флойда-Хоара, багатозначні, нечіткі, ймовірнісні, можливісні, динамічні, програмні логіки. Результати, отримані в області математичної логіки, та задачі, які розв'язуються її засобами і методами, знаходять різноманітні застосування в різних сферах діяльності людини. Ідеї та методи математичної логіки з кожним роком усе глибше пронизують математику, інформатику, психологію, філософію, лінгвістику.

Таким чином, глибоке оволодіння апаратом математичної логіки вкрай необхідне для формування кваліфікованих спеціалістів з інформаційних технологій. Тому особливе значення має належний рівень викладання навчальної дисципліни «Математична логіка» із врахуванням новітніх наукових досягнень з метою набуття студентами компетенцій, знань, умінь та навиків, що дасть змогу надалі успішно застосовувати їх в навчанні, в науковій та в практичній діяльності.

Проблема побудови програмно-орієнтованих логік

Поняття й методи математичної логіки засвідчують високу ефективність при розв'язанні широкого кола задач інформатики та програмування. На даний момент створено (див., напр., [1–2]) низку різноманітних логічних систем, які успішно для цього використовуються Такі системи завичай базуються на класичній логіці предикатів (див., напр., [3–7]).

Особливе місце класичної логіки зумовлене наступним.

1. Закони логіки предикатів мають універсальний характер, виражають загальні закони мислення людини, вони істинні в усіх предметних областях.

2. Класична логіка всебічно досліджена, вона має великий досвід застосування, для неї побудовано багато систем автоматизованого доведення.

3. Класична логіка є основою низки спеціальних логік (модальних, темпоральних, епістемічних, алгоритмічних тощо).

Проте класична логіка, незважаючи на численні позитивні вартості, має низку принципових обмежень, які ускладнюють її використання в інформатиці й програмуванні. Вона недостатньо враховує структурованість, неповноту, частковість інформації про предметну область, її динаміку.

Проведений в [8] аналіз основних понять класичної логіки дозволяє зробити наступні висновки.

1. При побудові класичної логіки характерне превалювання синтаксичних аспектів, хоча для програмування набагато важливішими є семантичні. Синтактико-семантичний підхід до побудови класичної логіки проявляється в тому, що спочатку визначають мову логіки і, досить часто, відношення виведення (синтаксичні аспекти), а вже потім – інтерпретації та відношення виконуваності (семантичні аспекти).

2. В класичній логіці функції та предикати трактуються як однозначні скінченно-арні відображення, причому предикати – тотальні (всюди визначені), а в програмуванні й моделюванні використовуються набагато потужніші класи часткових функцій і предикатів над іменними (номінативними) даними.

3. В класичній логіці пропозиційні зв’язки трактуються як тотальні n-арні булеві функції, квантори не мають самостійного семантичного визначення, їх смисл розкривається в процесі інтерпретації формул.

Зазначені обмеження зумовлені в першу чергу тим фактом, що на час формування класичної логіки програмування ще не існувало, тому логіка будувалась на базі наявного математичного апарату, який орієнтувався на використання однозначних скінченно-арних відображень. Синтактико-се-

мантична схема побудови класичної логіки зумовлена також тим, що синтаксис завжди простіший, тому побудову починали саме з нього.

Обмеження класичної логіки предикатів мотивують необхідність побудови нових логік, які більше орієнтовані на потреби інформатики й програмування. При побудові програмно-орієнтованих логік доцільно керуватися [8] наступними положеннями:

– необхідно використовувати семантико-синтаксичний підхід до побудови логіки;

– необхідно переходити від *n*-арних до номінативних (квазіарних) предикатів при поданні денотатів формул;

– пропозиційні зв’язки і квантори необхідно трактувати як композиції квазіарних предикатів.

Таким чином, на перший план висувається проблема побудови нових, програмно-орієнтованих логік. Таку побудову ведемо в семантико-синтаксичному стилі. Центральний момент цієї побудови – вибір класу номінативних (квазіарних) предикатів як семантичної основи нових логік. Цей вибір має кілька важливих наслідків:

– з'являється можливість побудови логік різного рівня абстракції (інтесіональні аспекти логіки);

– ми будуємо логіки часткових (необов’язково тотальних однозначних) предикатів;

– з'являється можливість відокремити семантику від синтаксису і провести її окреме дослідження в рамках алгебр предикатів;

– логіку та програмування можна будувати на спільній концептуальній основі.

На основі розгляду основних конструкцій мов програмування та їх формалізації, проведеного [9] на прикладі простої мови програмування мови *SIPL* [10], можна зробити наступні висновки:

1) формалізація здійснюється на базі різних видів алгебр даних та програмних алгебр (алгебр функцій);

2) побудова складних даних із простіших відбувається на основі відношень іменування (номінативних відношень);

3) побудова складних функцій, що задають семантику програм (програмних функцій), відбувається за допомогою композицій (алгебраїчних операцій над функціями);

4) основними властивостями програмних функцій є їх еквітонність та монотонність.

Зазначені твердження є основоположними для побудови логік, які орієнтовані на дослідження властивостей програм. Таким чином, провідну роль при побудові програмно-орієнтованих логік посідають алгебраїчні та композиційно-номінативні аспекти. Тому в основі програмно-обумовленої концепції побудови нових логік лежить вибір запропонованого в [11] *композиційно-номінативного підходу* (КНП) як бази такої побудови. Цей підхід до побудови моделей програм і орієнтованих на них логік задає принципи визначення та дослідження формальних мов програм та логічних систем.

Стисло розглянемо основні принципи композиційно-номінативного підходу.

Базовим методологічним принципом, що використовується при експлікації понять логіки та програмування, є *принцип розвитку* від абстрактного до конкретного: поняття досліджуваного об’єкта визначається в процесі розвитку. Розвиток починається з найабстрактніших визначень, що відбивають загальні властивості об’єкта, і поступово переходить до конкретніших визначень, які відбивають специфічні властивості об’єкта.

Принцип розвитку є дуже важливим для логіки та програмування, він веде до ієрархії визначень об’єкта на різних рівнях абстрактності та загальності. Визначення об’єкта на певному рівні абстракції (*інтенсіональний* аспект) має бути доповненим визначенням класу об’єктів, які належать цьому рівню (*екстенсіональний* аспект), при цьому обидва аспекти мають вивчатися в їх єдності. Це дає змогу сформулювати загальнонауковий *принцип єдності* (інтегрованості) аспектів: поняття мають бути подані в єдності їх інтенсіональних та екстенсіональних аспектів, причому інтенсіональний аспект в єдності має провідну роль. Ми традиційно тлумачимо інтенсіонал як зміст поняття, екстенсіонал – як його обсяг.

Втіленням пріоритету інтенсіонального аспекту над екстенсіональним є *принцип пріоритетності* семантики над синтаксисом: семантичний та синтаксичний аспекти спочатку вивчаються окремо, а потім сумісно із пріоритетом семантичного аспекту.

Основними конкретно-науковими принципами є принципи композиційності та номінативності. Вони, власне, і задають особливості КНП.

Принцип *композиційності* трактує засоби побудови програм (функцій, предикатів) як алгебраїчні операції. Для логіки це означає зведення логічних зв'язок і кванторів до композицій предикатів.

Принцип *номінативності* вимагає використання відношень іменування для побудови семантичних моделей та опису програм (функцій, предикатів).

Наведені принципи визначають напрямки та особливості експлікації основних понять логіки та програмування, вони є основою, ядром композиційно-номінативного підходу до побудови програмних і логічних систем.

Програмні поняття формалізуються за допомогою програмних систем, які розкривають поняття програми на різних рівнях розгляду. В програмних поняттях можна виділити семантичний, синтаксичний та денотаційний аспекти, тому програмна система визначається як *композиційно-номінативна система* (*Cs*, *Ds*, *Dns*), де *Cs* – композиційна, *Ds* – дескриптивна, *Dns* – денотаційна системи.

Композиційні системи визначають засоби побудови функцій над деякою множиною даних. В загальному випадку вони мають вигляд (*D*, *Fn*, *Cm*), де *D* – множина даних, *Fn* – множина функцій над *D*, *Cm* – множина композицій (операцій) над *Fn*. Така система задає дві алгебри: алгебру даних (*D*, *Fn*) та алгебру функцій (програмну алгебру) (*Fn*, *Cm*).

Дескриптивні системи задають дескрипції (терми, формули), що є описами функцій. Дескрипції визначаються індуктивно, використовуючи імена базових функцій та імена композицій. Зазвичай дескрипціями є терми програмної алгебри.

Денотаційні системи задають денотати (значення) дескрипцій.

Центральним поняттям логіки є поняття предиката. З математичного погляду предикати – це функції, значеннями яких є істиннісні (булеві)

значення. Тому класи предикатів теж можна задавати композиційно-номінативними системами. Предикатні композиційно-номінативні системи [12] є семантичною основою різноманітних логік. Це дозволяє подавати логічні та програмні системи в єдиному стилі та вивчати їх на основі спільного підходу.

Таким чином, КНП задає принципи визначення та дослідження формальних мов програм та логік. Його суть полягає в наступному:

– семантичний аспект мови є провідним, синтаксичний – похідним; тому спочатку визначаються та досліджуються семантичні аспекти, а потім – синтаксичні;

– семантичні аспекти уточнюються за допомогою композиційно-номінативних систем, які визначають структури даних, функцій (зокрема, предикатів) та композицій; такі системи можна подавати у вигляді двох семантичних алгебр – алгебри даних та алгебри функцій (предикатів);

– дані уточнюються як номінативні, тобто побудовані на базі відношення ім'я$\mapsto$значення; основні структури даних мов логіки та програмування можна подати як конкретизації номінативних даних;

– властивості семантичних алгебр індукують синтаксичні аспекти мов;

– побудова мов іде в напрямку від абстрактного до конкретного, що приводить до мов різного рівня абстракції та загальності.

Логіки, збудовані на основі композиційно-номінативного підходу, названо композиційно-номінативними. Передумовою їх виникнення стала необхідність посилення можливостей класичної логіки для розв'язку нових задач інформатики й програмування. У цих логіках основний акцент робиться на дослідженні семантичних аспектів, а синтаксичні аспекти вважаються похідними від семантичних. Композиційно-номінативні логіки (КНЛ) базуються на загальних класах часткових відображень, заданих на номінативних (іменних) даних. Такі відображення названо квазіарними.

Композиційно-номінативні логіки будуємо за семантико-синтаксичною схемою. Це означає наступне.

1. Спочатку задаємо інтенсіональні (змістовні) моделі логік. Такі моделі найперше визначаються рівнями розгляду даних, тому для їх задання фіксуємо рівень абстракції розгляду. Інтенсіональні моделі індукують клас формул (мову логіки) відповідного рівня (синтаксичний аспект).

2. Будуємо відповідні розглянутому рівню екстенсіональні моделі, які задають семантичні аспекти логік. Екстенсіональна семантична модель задається як предикатна композиційна система (D, Pr, Cm), вона визначає алгебру (алгебраїчну систему) даних (D, Pr) та алгебру предикатів (Pr, Cm). Терми алгебри предикатів трактуються як формули мови логіки. Композиції визначають універсальні методи побудови предикатів, вони є основою, ядром логіки певного типу.

3. Будуємо формально-аксіоматичні логічні числення. Такі числення задають синтаксичні аспекти логік. Основними їх типами є формальні системи гільбертівського типу та системи ґенценівського типу (секвенційні числення, системи натурального виведення).

Таким чином, КНЛ будуємо за семантико-синтаксичною схемою. Визначення КНЛ реалізують єдність інтенсіонального та екстенсіонального аспектів. Дослідження семантичних аспектів КНЛ зводиться до вивчення властивостей алгебр предикатів, які є основним поняттям КНЛ.

Спектр композиційно-номінативних логік

Застосування КНП дає змогу побудувати (див., напр., [7–9, 13–20]) низку логічних моделей різноманітних предметних областей, що знаходяться на різних рівнях абстрактності та загальності.

Побудову КНЛ починаємо з гранично-абстрактних рівнів, поступово їх конкретизуючи.

Ці рівні відрізняються трактуванням рівня розгляду даних.

Пропозиційний рівень. На цьому рівні дані трактуються гранично абстрактно, як "чорні скриньки". Жодна властивість даних не є доступною. Предикати, задані на даних цього рівня, мають вигляд $A \to \{T, F\}$, де $\{T, F\}$ – множина істиннісних значень, A – сукупність абстрактних даних. Ці дані не можна відрізнити одне від одного, тому на пропозиційному рівні кожний предикат веде себе однаково на кожному даному. На цьому рівні композиції

фактично працюють лише з виробленими предикатами істиннісними значеннями, що й пояснює трактування логічних зв'язок в класичній логіці як булевих функцій.

Базовими композиціями логік пропозиційного рівня будемо вважати диз'юнкцію $\vee$ та заперечення $\neg$.

Сингулярний рівень може трактуватися як конкретизація пропозиційного. На такому рівні дані трактуються гранично конкретно, як "білі" скриньки. На цьому рівні фіксується єдиний клас даних, тому такий рівень називають сингулярним. Предикати, задані на даних цього рівня, мають вигляд $D \to \{T, F\}$, де D – множина конкретних даних. Такі предикати трактуються гранично абстрактно. Композиціями сингулярного рівня є [9] гранично конкретні аплікативні композиції. Практично кожний логічний засіб можна трактувати як аплікативну композицію.

Номінативний рівень є синтезом двох перших. На цьому рівні дані розглядаються як "рябі" скриньки, побудовані з "білих" і "чорних"; такі дані називаються номінатами. Відповідні логіки відносимо до номінативного рівня. На відміну від двох перших рівнів, що визначають порівняно прості класи логік, номінативний рівень є дуже багатим і розпадається на низку підрівнів. Найважливішим є підрівень однозначних номінатів – *іменних множин*. Іменна множина (ІМ) – це множина пар, 1-а компонента пари – це ім'я, а 2-а – значення цього імені. Імена трактуємо гранично конкретно, предметні значення – гранично абстрактно. Ми розглядаємо *однозначні* ІМ, це означає, що в конкретній ІМ одне ім'я не може іменувати два різних значення.

Поняття ІМ дуже широке, до ІМ можна віднести пари, *n*-ки, послідовності; індексовані множини теж можна трактувати як однозначні ІМ.

ІМ доцільно трактувати як функції, звідки таке визначення.

V-A-іменна множина (*V-A*-ІМ) – це довільна однозначна функція $\delta : V \to A$. Тут V та A трактуємо як множини предметних імен та предметних значень.

Множину всіх *V-A*-ІМ позначимо ${}^{V}A$.

Вводимо функцію $asn : {}^{V}A \to 2^{V}$ таку:

$$asn(\delta) = \{v \in V \mid v \mapsto a \in \delta \text{ для деякого } a \in A\}$$

Множину всіх $\delta \in {}^{V}A$ таких, що $asn(\delta) = X$, де $X \subseteq V$, позначаємо A^X.

Функції, задані на ІМ, називають *квазіарними*.

V-A-квазіарна функція – це функція вигляду $f : {}^{V}A \to R$.

Функцію вигляду $f : A^X \to R$ назвемо *X-арною.*

Традиційні *n*-арні функції вигляду $f : A^n \to R$ можна трактувати як $\{1,..., n\}$-арні.

Для логік природно розглядати квазіарні функції двох типів.

Функцію вигляду $P : {}^{V}A \to \{T, F\}$ назвемо *V-A*-квазіарним предикатом.

Функцію вигляду $f : {}^{V}A \to A$ назвемо квазіарною функцією на *A.*

Класи *V*-квазіарних функцій і *V*-квазіарних предикатів на *A* позначаємо Fn^A і Pr^A.

Кожний квазіарний предикат $P : {}^{V}A \to \{T, F\}$ однозначно задаємо двома множинами: областю істинності $T(P) = \{d \mid P(d)\downarrow = T\}$ та областю хибності $F(P) = \{d \mid P(d)\downarrow = F\}$.

Квазіарний предикат $P : {}^{V}A \to \{T, F\}$ назвемо:

– *однозначним*, якщо $T(P) \cap F(P) = \varnothing$;

– *тотальним*, якщо $T(P) \cup F(P) = {}^{V}A$;

– *неспростовним*, або *частково істинним*, якщо $F(P) = \varnothing$;

– *виконуваним*, якщо $T(P) \neq \varnothing$;

– *всюди невизначеним* (позн. $\perp$), якщо $T(P) = F(P) = \varnothing$;

– *тотожно істинним* (позн. T), якщо $T(P) = {}^{V}A$ і $F(P) = \varnothing$;

– *тотожно хибним* (позн. F), якщо $T(P) = \varnothing$ і $F(P) = {}^{V}A$.

Логіки квазіарних предикатів – це КНЛ рівня іменних множин.

На рівні ІМ далі можна виділити *номінативно-безквантворні* та *першопорядкові* рівні.

Номінативно-безквантворні рівні:

– реномінативний (рівень РНЛ);

– реномінативний з слабкою рівністю (рівень $РНЛ_{=}$);

– реномінативний з строгою рівністю (рівень $РНЛ_{\equiv}$);

– безквантворно-функціональний (рівень БКЛ);

– безквантворно-функціональний з слабкою рівністю (рівень $БКЛ_{=}$);

– безквантворно-функціональний з строгою рівністю (рівень БКЛ$_{\equiv}$).

Першопорядкові рівні:

– чистий першопорядковий, або кванторний рівень (рівень ЧКНЛ);
– чистий першопорядковий рівень з слабкою рівністю (рівень ЧКНЛ$_{=}$);
– чистий першопорядковий рівень з строгою рівністю (рівень ЧКНЛ$_{\equiv}$);
– функціональний рівень (рівень ФКНЛ);
– функціональний рівень з слабкою рівністю (рівень ФКНЛ$_{=}$);
– функціональний рівень з строгою рівністю. (рівень ФКНЛ$_{\equiv}$).

Реномінативний рівень є найабстрактнішим серед номінативних. Починаючи з реномінативного рівня, можна перейменовувати компоненти даних. Це дає змогу ввести композицію реномінації (перейменування) $\mathrm{R}_{\bar{x}}^{\bar{v}}$. Базовими композиціями реномінативних логік (РНЛ) є $\neg$, $\vee$, $\mathrm{R}_{\bar{x}}^{\bar{v}}$.

Реномінативні рівні з рівністю. Тут можна ототожнювати й розрізняти значення предметних імен за допомогою спеціальних 0-арних композицій – параметризованих за іменами предикатів рівності. Можна розглядати два їх різновиди – предикати строгої (точної) рівності $\equiv_{xy}$ та слабкої рівності $=_{xy}$. Відповідно говоримо про РНЛ зі слабкою рівністю (РНЛ$_{=}$) та РНЛ зі строгою рівністю (РНЛ$_{\equiv}$).

Базовими композиціями РНЛ$_{=}$ є $\neg$, $\vee$, $\mathrm{R}_{\bar{x}}^{\bar{v}}$, $=_{xy}$.

Базовими композиціями РНЛ$_{\equiv}$ є $\neg$, $\vee$, $\mathrm{R}_{\bar{x}}^{\bar{v}}$, $\equiv_{xy}$.

Безквантворно-функціональний рівень. Тут маємо можливості формування нових значень для аргументів функцій і предикатів. Це дає змогу ввести композицію суперпозиції $\mathrm{S}^{\bar{x}}$. Виділення *V*-*A*-квазіарних функцій та *V*-*A*-квазіарних предикатів індукує виділення суперпозицій двох типів: $(Fn^A)^{n+1} \to Fn^A$ та $Pr^A \times (Fn^A)^n \to Pr^A$.

Для виділення окремих компонент даних визначаємо спеціальні 0-арні композиції – функції деномінації (розіменування) $'x$, де $x \in V$. При їх введенні реномінації можна промоделювати за допомогою суперпозицій.

КНЛ безквантворно-функціонального рівня назвемо БКНЛ.

Базові композиції БКНЛ: $\neg, \vee, S^{\bar{x}}, {}'x$.

Безквантворно-функціональні рівні з рівністю. Тут додатково можна ототожнювати й розрізняти предметні значення за допомогою спеціальної композиції рівності вигляду $Fn^A \times Fn^A \rightarrow Pr^A$. Маємо два її різновиди: слабкої рівності = та строгої (точної) рівності ≡. Відповідно говоримо про БКНЛ зі слабкою рівністю (БКНЛ$_=$) та БКНЛ зі строгою рівністю (БКНЛ$_\equiv$).

Базовими композиціями БКНЛ$_=$ є $\neg, \vee, S^{\bar{x}}, {}'x, =$.

Базовими композиціями БКНЛ$_\equiv$ є $\neg, \vee, S^{\bar{x}}, {}'x, \equiv$.

Для *першопорядкових* рівнів характерним є використання композицій квантифікації. На цих рівнях можна застосовувати квазіарні предикати до всіх предметних значень, що дозволяє ввести композиції квантифікації (квантори) $\exists x$ та $\forall x$.

Чистий першопорядковий (***кванторний***) ***рівень.*** Тут маємо чисті першопорядкові КНЛ, або ЧКНЛ). Базові композиції ЧКНЛ: $\neg, \vee, R^{\bar{v}}_{\bar{x}}, \exists x$.

Чисті першопорядкові рівні з рівністю. Тут маємо ЧКНЛ зі слабкою рівністю (ЧКНЛ$_=$) та ЧКНЛ зі строгою рівністю (ЧКНЛ$_\equiv$).

Базовими композиціями ЧКНЛ$_=$ є $\neg, \vee, R^{\bar{v}}_{\bar{x}}, \exists x, =_{xy}$.

Базовими композиціями ЧКНЛ$_\equiv$ є $\neg, \vee, R^{\bar{v}}_{\bar{x}}, \exists x, \equiv_{xy}$.

Першопорядковий функціональний рівень. Тут маємо першопорядкові КНЛ функціонального рівня (ФКНЛ).

Базові композиції ФКНЛ: $\neg, \vee, S^{\bar{x}}, {}'x, \exists x$.

Першопорядкові функціональні рівні з рівністю. Тут маємо ФКНЛ зі слабкою рівністю (ФКНЛ$_=$) та ФКНЛ зі строгою рівністю (ФКНЛ$_\equiv$).

Базовими композиціями ФКНЛ$_=$ є $\neg, \vee, S^{\bar{x}}, {}'x, \exists x, =_{xy}$.

Базовими композиціями ФКНЛ$_\equiv$ є $\neg, \vee, S^{\bar{x}}, {}'x, \exists x, \equiv_{xy}$.

На наступному рівні абстракції дані розглядаються як ***ієрархічні.*** Вони будуються індуктивно з множин предметних імен та предметних значень. Відповідні логіки названо логіками ієрархічних номінативних даних.

Ієрархічно-номінативний рівень дуже багатий, зроблено лише перші кроки його дослідження. В курсі "Математична логіка" ієрархічно-номінативні логіки не розглядаємо.

Особливості логік квазіарних предикатів. В класичній логіці використовують однозначні X-арні функції та тотальні однозначні X-арні предикати, де $X \subseteq V$ – скінченні множини предметних імен (змінних), причому базові функції та предикати – тотальні однозначні n-арні. Наприклад, $\{x, y, z\}$-арний предикат $x < y \vee y < z$; $\{u, v\}$-арна функція $u + v$.

X-арна функція чи X-арний предикат класичної логіки може залежати лише від предметних імен множини X. Неформально кажучи, для X-арних функцій та предикатів маємо фіксований формат вхідних даних. Водночас квазіарні функції та предикати можуть залежати від ***наперед необмеженої*** множини предметних імен. Це є ***визначальною властивістю*** логіки квазіарних предикатів.

Важливими властивостями програмних відображень є їх еквітонність та монотонність. Монотонність означає, що при розширенні вхідного даного вихідне дане теж розширюється. Еквітонність означає, що при розширенні вхідного даного вихідне дане не змінюється. Звідси маємо такі визначення.

Квазіарна функція f *монотонна*, якщо: $d \subseteq d' \Rightarrow f(d) \subseteq f(d')$.

Квазіарна функція (предикат) q *еквітонна*, якщо: $d \subseteq d' \Rightarrow q(d) = q(d')$.

Для часткових предикатів невірні деякі важливі закони класичної логіки. Зокрема, немає транзитивності імплікації та еквіваленції квазіарних предикатів, невірне правило modus ponens. Справді, задамо предикат P як $\perp$, Q – як F, тоді $P \to Q$ теж $\perp$. Отже, P та $P \to Q$ неспростовні, водночас Q – тотожно хибний, що й спростовує modus ponens.

Для спростування транзитивності імплікації та еквіваленції задамо P як T, Q – як $\perp$, S – як F; тоді $P \to Q$, $P \leftrightarrow Q$, $Q \to S$, $Q \leftrightarrow S$ неспростовні, водночас $P \to S$, $P \leftrightarrow S$ тотожно хибні.

Нетранзитивними є також предикати слабкої рівності $=_{xy}$. При цьому предикати $=_{xy}$ та константні предикати F, $\perp$ є еквітонними. Це означає, що відмінність логіки класичної та логіки квазіарних предикатів виявляється вже на рівні *еквітонних* предикатів. Водночас для логік еквітонних предикатів зберігаються основні закони класичної логіки, тому логіки еквітонних предикатів названо [8] *некласичними*.

Предикати строгої рівності $\equiv_{xy}$ тотальні транзитивні, проте немонотонні й нееквітонні, тому не діють в логіках еквітонних предикатів.

Для загального випадку квазіарних предикатів вже невірні деякі пов'язані з кванторами логічні закони; зокрема, невірними є традиційні закони $T(P) \subseteq T(\exists xP)$ та $T(\forall xP) \subseteq T(P)$.

Справді, візьмемо відомі [15] предикати-індикатори Ex наявності у вхідних даних компоненти з іменем x, вони задаються так:

$$Ex(d) = T \text{ при } x \in asn(d);\ \ Ex(d) = F \text{ при } x \notin asn(d).$$

Предикати-індикатори тотальні однозначні, проте нееквітонні й немонотонні. Для таких d, що $x \notin asn(d)$, маємо: $\neg Ex(d) = T$ та $\exists x \neg Ex(d) = F$; $\forall x Ex(d) = T$ та $Ex(d) = F$.

Водночас для еквітонних предикатів зазначені закони $T(P) \subseteq T(\exists xP)$ та $T(\forall xP) \subseteq T(P)$ є вірними.

Викладені вище міркування покладені в основу робочої програми навчальної дисципліни «Математична логіка» для студентів 2 курсу освітнього ступеня ″бакалавр″ галузі знань ″Інформаційні технології″ спеціальності ″Комп'ютерні науки″ освітньо-професійної програми ″Інформатика″.

Подальший розвиток зазначені міркування отримали в новій робочій програмі навчальної дисципліни «Прикладні та композиційні логіки» для студентів 2 курсу освітнього ступеня ″магістр″ галузі знань ″Інформаційні технології″ спеціальності ″Комп'ютерні науки″ освітньо-професійної програми ″Інформатика″. Акцент зроблено на вивченні немонотонних логік квазіарних предикатів та побудові для них систем пошуку виведень секвенційного типу, багатозначних логік, логік над ієрархічними номінативними

даними, композиційно-номінативних модальних та темпоральних логік. Детальніше про це – в наступних працях.

Особливості програми дисципліни «Математична логіка»

Навчальна дисципліна «Математична логіка» є складовою циклу професійної підготовки фахівців освітнього ступеня ″бакалавр″ галузі знань ″Інформаційні технології″ спеціальності ″Комп'ютерні науки″ освітньо-професійної програми ″Інформатика″.

Метою вивчення зазначеної дисципліни є засвоєння базових знань з основ математичної логіки, включаючи вивчення формальних логіко-математичних мов та їх можливостей для опису і моделювання предметних областей, систем пошуку доведень та формально-логічних числень.

Завданням вивчення дисципліни «Математична логіка» є набуття компетенцій, знань, умінь та навиків на рівні новітніх досягнень у математичній логіці згідно кваліфікації фахівець з інформаційних технологій.

В результаті вивчення навчальної дисципліни «Математична логіка» студент повинен:

знати основні поняття, засоби і методи математичної логіки, їх застосування в інформатиці й програмуванні; знати мови пропозиційної логіки та логіки 1-го порядку, їх можливості для опису предметних областей; мати сучасні уявлення про програмно-орієнтовані логіки часткових квазіарних предикатів; мати сучасні уявлення про основні методи пошуку доведень та засоби логічного виведення; мати сучасні уявлення про нетрадиційні логіки (багатозначні, інтуїціоністські, модальні; темпоральні, епістемічні) та їх застосування.

вміти описувати на формальних мовах твердження стосовно тих чи інших предметних областей; встановлювати істинність та виконуваність, наявність логічного наслідку; встановлювати (використовуючи метод автоморфізмів) виразність та невиразність предикатів у семантичних моделях мови; проводити виведення в численнях Гільбертівського типу, резолютивні виведення; проводити виведення в численнях Ґенценівського (секвенційного) типу класичних логік та логік еквітонних квазіарних предикатів.

Дисципліна "Математична логіка" є базовою для засвоєння матеріалу нормативних дисциплін "Теорія алгоритмів", "Бази даних та інформаційні системи", а також "Теорія програмування", "Системне програмування", "Інформаційні технології", "Алгебраїчні структури, криптографія та захист інформації", "Теорія прийняття рішень", "Теорія прийняття рішень", низки спецкурсів відповідного напряму.

Опишемо структуру навчальної дисципліни «Математична логіка».

В 1-й частині курсу розглядаються основні поняття логіки, пропозиційна логіка та логіки номінативно-безквантворних рівнів.

Вступна тема присвячена становленню та розвитку логіки. Розглянуто базові поняття логіки – поняття висловлення, предиката, числення, логічної системи. Наведено основні закони традиційної логіки.

Вивчення логічних систем починається з найбільш абстрактного, пропозиційного рівня. Визначаються композиції пропозиційного рівня – логічні зв'язки, описується мова пропозиційної логіки, після цього розглядаються пропозиційні числення Гільбертівського типу. Дається перше знайомство із системами пошуку виведень на прикладі методу резолюцій пропозиційної логіки та пропозиційних секвенційних числень. Для секвенційних числень доводяться теореми коректності та повноти.

Далі визначаються фундаментальні поняття іменної множини, квазіарної функції, квазіарного предиката. Описуються композиції номінативних рівнів для квазіарних функцій та предикатів: *реномінації, суперпозиції, рівності, квантори***.** Наводиться спектр логік квазіарних предикатів. Детально розглядаються реномінативні логіки, описуються їх мови, визначаються нормальні форми та субтавтології. Дається поняття про реномінативні логіки з рівністю та логіки безквантворно-функціональних рівнів.

2-а частина курсу присвячена розгляду семантичних аспектів логік 1-го порядку. Спочатку розглядаються класичні логіки 1-го порядку, їх мови та семантичні моделі (класичні алгебраїчні системи). Особливий акцент зроблено на мові арифметики. Вивчаються виразність предикатів, множин та функцій в алгебраїчних системах, істинність та виконуваність формул, відношення тавтологічного, логічного, слабкого логічного наслідку, логічної еквівалентності. Розглядаються еквівалентні перетворення формул, теореми

еквівалентності та рівності, нормальні форми (пренексна, сколемівська). Вивчаються гомоморфізми, ізоморфізми, автоморфізми алгебраїчних систем, їх елементарна еквівалентність; наводиться метод автоморфізмів для доведення невиразності предикатів в алгебраїчних системах. Далі розглядаються першопорядкові неокласичні логіки еквітонних предикатів, їх семантичні властивості, нормальні форми формул. Описується відношення неспростовнісного логічного наслідку для множин формул, його властивості.

В 3-й частині курсу розглядаються формально-аксіоматичні числення класичних та неокласичних логік 1-го порядку. Спочатку вивчаються числення Гільбертівського типу класичних логік 1-го порядку – теорії 1-го порядку. Наводяться приклади таких числень, зокрема, числення предикатів 1-го порядку та формальна арифметика. Розглядаються поняття несуперечливості та максимальності (повноти) теорій 1-го порядку, теорема Ґьоделя про повноту та її наслідки (теореми компактності, Льовенгейма-Сколема), категоричність теорій 1-го порядку. Наводяться теореми Ґьоделя про неповноту.

Далі розглядаються системи пошуку виведень в логіках 1-го порядку. Наводяться теорема Ербрана та метод спростування Ербрана, метод резолюцій для логік 1-го порядку. Вивчаються секвенційні числення класичних логік 1-го порядку та логік еквітонних предикатів 1-го порядку, для цих числень доводяться теореми про контрмоделі, теореми коректності та повноти. На цій основі розглядаються інтерполяційна теорема, семантична і синтаксична визначність, теореми про визначність.

4-а частина курсу присвячена розгляду нетрадиційних логік. Дається поняття про логіки вищих порядків. Далі роглядаються логіки квазіарних предикатів без обмежень монотонності. Описано різновиди квазіарних предикатів, класи інтерпретацій (семантики). Вводиться низка відношень логічного наслідку в логіках квазіарних предикатів, досліджено їх властивості.

Після цього роглядаються багатозначні логіки, зокрема, 3-значні логіки Лукасєвича та Кліні, 4-значна логіка Белнапа. Далі описуються інтуїціоністські логіки, їх реляційна семантика (семантика можливих світів) та аксіоматичні системи.

Завершується курс розглядом модальних логік. На відміну від традиційних логік, які орієнтовані на опис одного конкретного стану світу, мо-

дальні логіки враховують можливість його зміни і розвитку, що дає змогу застосовувати їх для аналізу та моделювання різноманітних аспектів діяльності людини. Найперше це стосується розробки інформаційних систем, зокрема, систем знань і експертних систем, верифікації та специфікації програм, задач опису та моделювання складних динамічних систем. Описуються алетичні модальні логіки, їх реляційна семантика, наводяться аксіоматичні системи. Велика увага приділена розгляду темпоральних логік та епістемічних логік знання. Можливості класичних модальних логік і композиційно-номінативних логік часткових предикатів поєднують композиційно-номінативні модальні логіки. Найважливішим їх класом є транзиційні модальні логіки, які враховують аспект зміни і розвитку предметних областей. Виділено різновиди цих логік: загальні, темпоральні, мультимодальні, епістемічні. Розглядається застосування темпоральних та епістемічних логік в інформатиці й програмуванні, відзначається роль темпоральних логік для специфікації та верифікації програм.

Висновки. Характерною особливістю програмно-обумовленої концепції викладання навчальної дисципліни «Математична логіка» є використання спільного для логіки й програмування композиційно-номінативного підходу. Цей підхід спирається на загальнометодологічний принцип розвитку від абстрактного до конкретного, основними його науковими принципами є принципи композиційності та номінативності. Принцип композиційності трактує засоби побудови програм, функцій, предикатів як алгебраїчні операції. Принцип номінативності означає використання важливих для математики й програмування відношень іменування для побудови семантичних моделей та опису даних, програм, предикатів. Використання композиційно-номінативного підходу індукує семантико-синтаксичну спрямованість викладення матеріалу. Акцент робиться на дослідженні семантичних аспектів, а синтаксичні аспекти є похідними від семантичних. Спочатку розглядаються семантичні моделі логік та вивчаються їх властивості, після цього будуються формально-аксіоматичні логічні числення. Визначальним є розгляд логічних систем на основі часткових квазіарних відображень, які широко розповсюджені в

інформатиці й програмуванні та узагальнюють традиційні скінченно-арні та *n*-арні відображення. Логіки часткових квазіарних предикатів є істотним узагальненням класичних логік, вони більше орієнтовані на потреби інформатики й програмування.

Таким чином, програмно-обумовлена концепція викладання навчальної дисципліни «Математична логіка» дає змогу використовувати новітні наукові досягнення в галузі математичної логіки, розвинути природний зв'язок логіки й прорамування, посилити прикладне спрямування дисципліни. Це дає змогу студентам отримати належні компетенції, знання та уміння в галузі математичної логіки, які враховують останні наукові результати, що дозволяє плідно застосовувати їх для подальшого вивчення нормативних і спеціальних дисциплін, в наукових дослідженнях та в практичній діяльності.

Список використаних джерел

1. Handbook of Logic in Computer Science. Edited by S. Abramsky, Dov M. Gabbay and T. S. E. Maibaum. – Oxford Univ. Press. – Vol. 1–5, 1993–2000.
2. Kröger F., Merz S. Temporal logic and state systems. – Berlin-Heidelberg: Springer-Verlag, 2008.
3. Gabbay D. Elementary logic (A procedural perspective). – Prentice Hall Europe, 1998.
4. Клини С. Математическая логика. – М.: Наука, 1973.
5. Мендельсон Э. Введение в математическую логику. – М.: Наука, 1976.
6. Шенфилд Дж. Математическая логика. – М.: Наука, 1975.
7. Нікітченко М.С., Шкільняк С.С. Математична логіка та теорія алгоритмів. – К.: ВПЦ Київський університет, 2008.
8. Никитченко Н.С., Шкильняк С.С. Неоклассические логики предикатов // Пробл. программирования. – 2010. – № 2–3. – С. 3–17.
9. Нікітченко М.С., Шкільняк С.С. Прикладна логіка. – К.: ВПЦ Київський університет, 2013.
10. Нікітченко М.С. Теорія програмування. Частина 1. – Ніжин, НДУ, 2010.

11. Никитченко М.С. Композиционно-номинативный подход к уточнению понятия программы // Пробл. программирования. – 1999. – № 1. – С. 16–31.
12. Никитченко М.С. Предикатные композиционно-номинативные системы // Пробл. программирования. – 1999. – № 2. – С. 3–19.
13. Шкільняк С.С. Спектр секвенційних чисель першопорядкових композиційно-номінативних логік // Проблеми програмування. – 2013. – № 3. – С. 22–37.
14. Шкільняк О.С. Модальні логіки немонотонних часткових предикатів // Вісник Київського ун-ту. Серія: фіз.-мат. науки. – 2015. – Вип. 3. – С. 141–147.
15. Нікітченко М.С., Шкільняк О.С., Шкільняк С.С. Чисті першопорядкові логіки квазіарних предикатів // Проблеми програмування. – 2016. – № 2–3. – С. 73–86.
16. Шкільняк С.С., Волковицький Д.Б. Композиційно-номінативні логіки безквантворних рівнів // Проблеми програмування. – 2016. – № 2–3. – С. 48–62.
17. Mykola S. Nikitchenko and Stepan S. Shkilniak. Algebras and logics of partial quasiary predicates // Algebra and Discrete Mathematics, Vol. 23 (2017), No 2, pp. 263–278.
18. Нікітченко М.С., Шкільняк О.С., Шкільняк С.С. Логіки загальних недетермінованих предикатів: семантичні аспекти // Проблеми програмування. – 2018. – № 2–3. – С. 31–45.
19. Шкільняк С.С. Першопорядкові композиційно-номінативні логіки з предикатами слабкої та строгої рівності // Проблеми програмування. – 2019. – № 3. – С. 28–44.
20. Нікітченко М.С., Шкільняк О.С., Шкільняк С.С. Секвенційні числення першопорядкових логік часткових предикатів з розширеними реномінаціями та композицією предикатного доповнення // Проблеми програмування. – 2020. – № 2–3. – С. 182–197.

Басараб Іван Андрійович, к.ф.-м.н., с.н.с.
Губський Богдан Володимирович, д.е.н., к.ф.-м.н., професор

СКІНЧЕННО-ЗБІЖНІ ТА ЦИКЛІЧНІ ФУНКЦІЇ, ПОСЛІДОВНОСТІ КОЛЛАТЦА І ГУДСТЕЙНА

Розглядається деякий клас числових функцій, названих скінченно-збіжними та циклічними. Характеристика скінченно-збіжних функцій проявляється в тому, що після багатократного (в смислі суперпозиції) обчислення значення функції на довільному аргументі результатом буде якесь (фіксоване для функції) число – точка збіжності. Окремим випадком таких функцій є циклічні функції. Також наводяться приклади циклічних та скінчено-збіжних функцій, асоційованих з послідовностями Коллатца та Гудстейна. Дослідження такого роду функцій може бути корисним в інших галузях математики, фізики, економіки та в науці про фрактальні структури (спеціальні геометричні структури, де присутня рекурсивна процедура відтворення подібностей).

Ключові слова: скінченно-збіжна функція, циклічна функція, послідовність Коллатца, послідовність Гудстейна

A class of numerical functions called finitely convergent and cyclic is considered. The characteristic of finite-converging functions is manifested in the fact that after multiple (in the sense of superposition) calculation of the value of the function on an arbitrary argument, the result will be some (fixed for the function) number – the point of convergence. A special case of such functions are cyclic functions. Examples of cyclic and finite-convergent functions associated with the Kollatz and Goodstein sequences are also given. The study of such functions can be useful in other fields of mathematics, physics, economics and the science of fractal structures (special geometric structures, where there is a recursive procedure for reproducing similarities).

Keywords: finite-convergent function, cyclic function, Kollatz sequence, Goodstein sequence

1. Визначення скінченно-збіжних та циклічних функцій

Нехай $\mathrm{N} = \{1, 2, 3, \ldots\}$, $\mathrm{N}_1 = \{2n+1 \mid n \geq 0\}$, $\mathrm{N}_2 = \{2n \mid n > 0\}$ і D – множина дійсних чисел, а f: D $\to$ D функція з областю визначення D_f і областю значень E_f. Через f^k – k-кратну суперпозицію функції f для $k \in$ N,

($f^k(d) = f^{k-1}(f(d))$ для кожного $d \in D_f$). Очевидно, що з кожним елементом $d \in D_f$ асоціюється одна єдина послідовність

$$[d] = (d = d_1,\ d_2 = f(d_1), \ldots, d_{k+1} = f(d_k) = f^k(d), \ldots .) \qquad (1)$$

В загальному випадку деякі члени такої послідовності можуть повторюватися.

Зрозуміло, що в залежності від функції f такі послідовності можуть бути як скінченними так і нескінченними.

Визначення 1. Функція f: D $\rightarrow$ D називається *скінченно-збіжною на множині* чисел A $\subset D_f$, якщо $\exists\ c \in E_f\ \forall\ a \in$ A $\exists$ k$\in N$ $f^k(a) = c$. Найменше (якщо в D_f існує найменше число) або найбільше (якщо найменшого числа в D_f не існує, а існує найбільше) з таких чисел c будемо називати *точкою збіжності* функції f на множині A (або *локальною точкою збіжності*), а найменше число k – довжиною збіжності (кожному числу a відповідатиме своя довжина збіжності). Якщо A = D_f, то фразу «на множині чисел A» будемо опускати.

Початкову підпослідовність послідовності [d], в якій *вперше* появляється деяке число d_k, позначатимемо [d: d_k]. Очевидно, що для скінченно-збіжної функції кожна послідовність (1) містить початкову підпослідовність [d: c], яка є скінченною та закінчується числом c – точкою збіжності. Якщо [d] містить деяку підпослідовність [d_i], то цей факт будемо записувати [d_i]◀[d].

***Визначення* 2**. Функція f: D $\rightarrow$ D називається *циклічною на множині* чисел A $\subset D_f$, якщо f є скінченно-збіжною на A і точка збіжності $c \in D_f$. Число c будемо називати *точкою циклу* функції f. Для циклічних функцій кожну таку початкову підпослідовність будемо називати *буфером циклу*, а число числа k – *довжиною буфера циклу*.

З цього визначення випливає, що якщо точку циклу c вилучити з D_f, то так змінена функція залишиться просто скінченно-збіжною. Крім цього, із функціональності та циклічності утворення таких послідовностей випливає, що всі вони відрізняються початковими підпослідовністями, але мають спільну кінцеву підпослідовність, яка містить щонайменше один член c.

***Приклад* 1**. Кожна стала функція (функція-константа) з $E_f = \{c\}$ є циклічною з точкою циклу c. В більш загальному випадку для функції f область визначення може бути об'єднанням множин A_1, A_2,A_3, …, які попарно не перетинаються (кількість таких множин може бути як скінченною так і нескінченною) і на кожній з них функція є сталою зі значенням c_i, яке є точкою циклу на множині A_i (i = 1, 2, 3,…). Прикладом такої функції може бути добре відома функція sign(d), яка приймає значення 1, 0 або -1, якщо число d відповідно додатне, дорівнює нулю або від'ємне. Числа 1, 0, -1 для sign є точками циклів відповідно на множинах A_1 – всі додатні дійсні числа, $A_2 = \{0\}$ і A_3 – всі від'ємні дійсні числа. До цього класу прикладів можна також віднести функцію [d], яка числу d ставить у відповідність його цілу частину. Очевидно, що для цієї функції кількість множин A_i є нескінченною. Для всіх наведених циклічних функцій довжини циклів та їх буферів рівні 1.

***Приклад* 2**. Нехай для фіксованого числа p>1 функція f: N $\rightarrow$ N задається наступним чином: $f(n) = n/p$, якщо n кратне p (ділиться націло на p), і $f(n) = n+1$ – в протилежному випадку. Стверджується, що така функція є циклічною з точкою циклу 1 та довжиною циклу рівним p, а *довжиною* буфера циклу не більшою за p. Дійсно, для довільного $n \neq p$ серед чисел $n+1$, $n+2$, …, $n+d$ знайдеться таке мінімальне число $n+d$, яке буде кратним числу p. Числа $n+1$, $n+2$, …, $n+d$ утворені в результаті застосування функції $f^{\,d}$. Якщо $n+d = pm$ для деякого натурального m, то $f(n+d) = m$. Залишається розглянути два випадки: $n < p$ і $n > p$. В першому випадку отримаємо $d = p–n$. В другому випадку маємо $n = pt+q$ для деяких натуральних t і $q< p$. Тоді отримаємо $d = p – q$. Третій можливий випадок, коли n кратне p, не розглядаємо, оскільки число n/p може бути знову кратним p або відповідати одному з розглянутих вище випадків.

Наведені приклади циклічних функцій характеризуються тим, що відповідні довжини буферів циклів є або фіксованими, або обмеженими величинами.

Приклад 3. Функція в множині цілих чисел.

Розглянемо наступну функцію: $F(n) = 2n$ для непарних чисел n, $F(n) = n/2$-1 для парних додатних чисел і, нарешті, $F(n) = n/2$+1 для парних

від'ємних чисел *n*. Стверджується, що так визначена функція є *скінченно-збіжною* в точці 0 – нейтральному цілому числу. Наведемо кілька прикладів послідовностей, породжених функцією *F*:

1, 2, 0
2, 0
3, 6, 2, 0
4, 1, 2, 0
-1, -2, 0
-2, 0
-3, -6, -2, 0
-4, -1, -2, 0

З визначення функції $F(n)$, а також з наведених прикладів видно, що послідовність $[n]$, породжена цією функцією, закінчується нулем тоді і тільки тоді, якщо вона має кінцеву підпослідовність [2:0] або [-2: 0].

Оскільки число 0 займає в множині цілих чисел нейтральну позицію, то простіше прийняти, що $F(0) = 0$. Це також аргументується тим, що яку би з наведених вище ознак числу 0 ми не приписали, все одно послідовність, утворена функцією $F(n)$, буде закінчуватися числом 0. З визначення самої функції $F(n)$ випливає, що вона є парною, тобто $F(-n) = - F(n)$. Тому для доведення циклічності $F(n)$ на всій її області визначення буде достатньо показати, що ця функція є циклічною на множині невідємних цілих чисел. Для цього скористаємося методом математичної індукції за числом *n*. Нехай для всіх $k \leq n$, послідовність $[k]$ закінчується числом 0. Далі, якщо $n+1 = 2m$ для деякого цілого *m*, то маємо $F(2m) = m-1 < (n-1)/2 < n$, а за припущенням послідовність $[m-1]$ закінчується нулем. При $n+1 = 2m+1$ маємо послідовність $(2m+1, 4m+2, 2m, m-1)$, в якій кінцева підпослідовність $[m-1]$ також закінчується нулем. Отже, для числа $n+1$ послідовність $[n+1]$ закінчується нулем. Крок індукції доведено.

2. Властивості циклічних функцій

Властивість 1. Для функції f альтернативно виконується одна з умов :

а) якщо $E_f \subset D_f$, то $\forall\ d \in D_f \setminus E_f\ (f(d) = c)$;

б) якщо $D_f \subset E_f$, то $E_f \setminus D_f = \{c\}$;

в) $E_f = D_f$.

Властивість 1.а) очевидна з визначення циклічної функції. Дійсно, якщо $\exists d \in D_f \setminus E_f$ ($f(d) \neq c$), тоді значення $f^2(d)$ буде невизначене і досягти точки циклу починаючи з числа d буде неможливо.

Нехай у випадку 1.б) множина $E_f \setminus D_f$ містить ще якесь число $d \neq c$. Тоді значення $f(d)$ також буде невизначене (аналогічно випадку 1.а). Більше того, властивість 1.б) гарантує, що точка циклу c не належить області визначення функції f і вона (точка) є в певному смислі *ізольованою*.

Якщо не виконуються ні 1.а), ні 1.б), то залишається варіант 1.в).

Властивість 2. Якщо E_f містить n >1 елементів (у випадку n =1 маємо сталу функцію і $E_f = \{c\}$), то довжина буфера циклу m для кожного $d \in D_f$ обмежена числом n+1. У випадку $m = n$+1 послідовність [d] містить всі елементи множини E_f.

Дослідження властивостей скінченно-збіжних та циклічних функцій з певних причин (і нам здається це очевидним) слід зосередитися в обмеженому класі числових функцій, а саме функцій в множині раціональних, зокрема, цілих та натуральних чисел. Це пов'язане, перш за все, з тим, що в визначені скінченно-збіжних функцій присутня вимога скінченності довжини збіжності (або довжини буфера циклу у випадку циклічних функцій).

3. Гіпотеза Коллатца "3n+1"

Дискусії навколо гіпотези Коллатца (була сформульована німецьким математиком Лотаром Коллатцом в Гамбурзі біля 1930р.) ось уже більше 90-та років не вщухають. Суть цієї гіпотези полягає в наступному: нехай задана функція K (далі її будемо інколи називати функцією Коллатца) в множині натуральних чисел **N** таким чином, що $K(n) = h(n) = n/2$ для $n \in \mathrm{N}_2$ і $K(n) = g(n) = 3n + 1$ для $n \in \mathrm{N}_1$, а згідно гіпотези Коллатца функція K є циклічною з точкою циклу 1. Насправді вона була сформульована в термінах послідовностей, які отримали назву «сіракузьких» послідовностей (цей термін ввів в 1950р. Г.Хассе, працюючи в Сіракузькому університеті). За

весь цей період спроб довести або спростувати гіпотезу Коллатца було немало. В цій роботі ми не будемо робити розширений аналіз різноманітних підходів до розв'язання цієї проблеми. Такий аналіз час від часу наводиться в літературі по цій проблематиці [1]. Як нам здається, на сьогодні акцент в проблематиці гіпотези Коллатца переноситься не тільки і не стільки на ще один варіант доведення справедливості цієї гіпотези, як на узагальнення формулювання самої гіпотези та на використання отриманих результатів в інших галузях точних наук.

Розглянемо ряд властивостей послідовностей, асоційованих з функцією Коллатца.

Нехай для довільного $n \in N_1$ має місце наступна послідовність:

$$[n] = (n = n_0,\ 2^{p_1} \cdot n_1\ ,\ n_1,\ 2^{p_2} \cdot n_2\ ,\ \ldots\ ,\ 2^{p_k} \cdot n_k\ ,\ldots) \qquad (2)$$

де $n_i \in N_1$, $g(n_i) = 2^{p_{i+1}} \cdot n_{i+1}$, $h^{p_i}(2^{p_i} \cdot n_i) = n_i$ і $p_i > 0$ для $i \geq 0$. Відмітимо, що послідовність (2) є скороченим аналогом повної послідовності (1), оскільки по кожному члену $2^{p_i} \cdot n_i$ однозначно відтворюється підпослідовність

$$[2^{p_i} \cdot n_i : n_i\ ] = (2^{p_i} \cdot n_i,\ 2^{p_i-1} \cdot n_i,\ \ldots,\ 2 \cdot n_i\ ,\ n_i\ ).$$

Таким чином, якщо функція K є циклічною з точкою циклу 1, то для n_0 існує таке число k, що $n_k = 1$ і член послідовності $2^{p_k} \cdot n_k$ через p_k кроків перетвориться в 1. Неважко замітити, що послідовності (2) однозначно відповідає послідовність чисел

$$(n_0\ ,\ p_1,\ p_2,\ \ldots\ ,\ p_k,\ \ldots)\ ,\ \text{де}\ \ \forall\ k \geq 1 (\ p_k > 0).$$

Далі через $L(n_0) = k + p_1 + p_2 + \ \ldots\ + p_k$ позначатимемо довжину буфера циклу для числа $n_0 > 0$: $L(1) = 3$, $L(2^p) = p$, $L(2^p n) = p + L(n)$ і $L(n_0) > 2k$. Наприклад,

$$L(2) = 0+1 = 1,\ L(3) = 2 + 1 + 4 = 7,\ L(4) = 0+2 = 2,\ L(5) = 1 + 4 = 5.$$

Наведемо декілька цікавих властивостей послідовностей, породжених функцією Коллатца.

(1). Якщо функція K є циклічною, то для довільного числа $n > 1$ має місце $(16,8,4,2,1) \blacktriangleleft [n]$.

Ця властивість легко виводиться з наступної властивості функції g (як складової функції Коллатца): значеннями g на непарних числах $2m+1$є члени арифметичної прогресії $4 + 6\,m$, $m \geq 0$ і тільки вони.

(2). Якщо в послідовності (2) має місце $n_k = 1$, то p_k – парне число.

Доведення цієї властивості базується на тому, що число $2^{p_k} - 1$ ділиться на 3, якщо p_k – парне число (доводиться методом математичної індукції за p_k).

(3). В послідовності (2) для довільного $k > 1$ виконується нерівність $3^k \cdot n_0 < 2^q \cdot n_k$, де $q = p_1+p_2+\ldots+p_k$.

Дійсно, з визначення функції g маємо $g(n_i) = 3n_i + 1 = 2^{p_{i+1}} \cdot n_{i+1}$, а звідси $-3n_i < 2^{p_{i+1}} \cdot n_{i+1}$. Виписавши всі такі нерівності для $i = 0, 1, \ldots, k$, перемноживши відповідно їх ліві та праві частини, а потім скоротивши обидві частини отриманої нерівності на їх спільний співмножник $n_1 n_2 \ldots n_{k-1}$, отримаємо нашу нерівність $3^k \cdot n_0 < 2^q \cdot n_k$, де $q = p_1+p_2+\ldots+p_k$.

Як наслідок цієї властивості маємо: при $n_k = 1$ (припущення, що в точці n_0 функція Коллатца досягає точки циклу 1) виконується нерівність $n_0 < 2^q / 3^k$, з якої випливають обмеження для k і q

$$k < q \cdot \log_3 2 - \log_3 n_0 \text{ і } q > k \cdot \log_2 3 + \log_2 n_0.$$

Можна припустити, що $n_0 > 3$, оскільки $g(1) = 4$. Тому $q > k \cdot \log_2 3 + \log_2 n_0 > k \cdot \log_2 3 + \log_2 3 = (k + 1) \log_2 3 > 3(k + 1)/2$. Звідси випливає обмеження довжини буфера циклу для точки n_0: $L(n_0) = k + q > (5k + 3)/2$. З іншого боку при $n_0 > 3$ маємо $k < q \cdot \log_3 2 - \log_3 n_0 < q \cdot \log_3 2 - 1$ і $L(n_0) = k + q < q \cdot \log_3 2 - 1 + q < 2q - 1$.

(4). В послідовності (2) $n_k = 1$ тоді і тільки тоді, коли $n_{k-1} = (4^m - 1)/3 = 4^{m-1} + 4^{m-2} + \ldots + 4 + 1$. Дійсно, рівність $n_{k-1} = (4^m - 1)/3$ випливає з властивостей **1** і **2** , оскільки $3n_{k-1} + 1 = 2^{p_k} \cdot n_k = 2^{2m} \cdot 1$, де $p_k = 2m$.

Шляхом використання математичної індукції за m та очевидної тотожності $(4^{m+1} - 1)/3 = 4 \cdot (4^m - 1)/3 + 1$ легко доводиться, що $(4^{m+1} - 1)/3 = 5 \cdot (4 \cdot \alpha_k) + 1$ при $m \in N_1$ і, відповідно, $(4^{m+1} - 1)/3 = 5 \cdot (4 \cdot \beta_k + 1)$ при $\in N_2$ (тут

$\alpha_k > 0$ і $\beta_k \geq 0$ – деякі натуральні числа, які залежать від n_{k-1}). Так, для m = 1, 2, 3 і 4 маємо відповідно n_{k-1}=5·0+1=1, 5·1=5, 5·4+1=21, 5·17=85.

(5). Числа виду $3n$ можуть бути лише першими членами послідовності (2), оскільки рівняння $3m + 1 = 2^p \cdot 3n$ або $1 = 3(2^p \cdot n - m)$ не має розв'язків в натуральних числах. Іншими словами, кожен внутрішній член послідовності не може бути кратним 3.

Послідовності, які починаються з чисел виду $3n$, будемо називати *зовнішніми*, а всі інші – *внутрішніми*. Таким чином, зовнішні послідовності являються в певному сенсі «максимальними», оскільки вони не входять в якості підпослідовностей ні в які інші послідовності.

Теорема 1. Для довільного n_0 в послідовності (2) не існує скінченного числа k такого, що $n_k = n_0 > 1$.

Це означатиме, що в послідовностях Коллатца відсутні "внутрішні цикли", тобто такі цикли, які починаються і завершуються числом відмінним від 1.

Для доведення скористаємося принципом *від супротивного*: припускаємо, що існує таке число k, що $n_k = n_0 > 1$. Тоді існує зовнішня послідовність виду

$$(3m, a_1, a_2, \ldots, a_q, n_0, 2^{p_1} n_1, n_1, 2^{p_2} n_2, \ldots, n_{k-1}, 2^{p_k} n_0),$$

де $a_q = 2^t n_0$ для деякого $t > 0$. Тоді виконуються рівності $3a_{q-1} + 1 = a_q = 2^t n_0$ і $3n_{k-1} + 1 = 2^{p_k} n_0$, а їх різниця створює нову рівність $3(a_{q-1} - n_{k-1}) = n_0 (2^t - 2^{p_k})$. Звідси отримуємо $n_0 = 3 (a_{q-1} - n_{k-1}) / (2^t - 2^{p_k})$, що суперечить наведеній вище властивості **(5)** послідовностей – відсутність членів (крім, можливо, першого), кратних 3. Таким чином, залишається один можливий варіант $n_k \neq n_0$.

Теорема 2. Функція Коллатца є циклічною тоді і тільки тоді, коли виконуються дві наступні умови:

а) якщо для довільного початкового числа n послідовність $[n]$ містить цикл $[c]$, то $c = 1$;

б) якщо в послідовності $[n]$ для довільного початкового числа n всі її члени попарно різні, то така послідовність є скінченною.

Доведення. Нехай гіпотеза Коллатца вірна і відповідна функція є циклічна з точкою циклу 1. Тоді умова а) виконується згідно визначення циклічної функції. Якщо би в послідовності [n] існувало таке число $c > 1$, що підпослідовність [c] утворювала би цикл, то число 1 входило би як член в [c]. Тоді цикл [1] = (1, 4, 2, 1) повинен міститися в [c]. Звідси отримуємо, що $c = 1$. Умова б) теж випливає безпосередньо з визначення циклічної функції: для кожного початкового числа n існує скінченне число k (довжина буфера циклу) таке, що в послідовності [n] до k-го члена включно всі члени попарно різні.

Далі, виконання умов а) і б) гарантують саму умову визначення циклічної функції.

Таким чином, для повного доведення гіпотези Коллатца потрібно довести виконання умови б), що виходить за межі цієї роботи.

4. Циклічні функції, породжені гіпотезою Коллатца

Функція в множині чисел обернених до натуральних

Нехай функція Коллатца K замінена на наступну функцію:

$$K'\colon \{1/n \mid n>0\} \to \{1/n \mid n>0\},\ \text{де } K'(1/n) = h'(1/n) = 2/n \text{ для } n \in \mathrm{N}_2 \text{ і}$$
$$K'(1/n) = g'(1/n) = 1/(3n+1) \text{ для } n \in \mathrm{N}_1.$$

Стверджується, *що функція K' є циклічною в точці 1 тоді і тільки тоді, коли такою є функція Коллатца*.

Дійсно, якщо для довільного натурального числа n_0 функція K породжує послідовність $\mathbf{(}n_0, 2^{p_1}\cdot n_1, n_1, 2^{p_2}\cdot n_2, \ldots, 2^{p_k}\cdot n_k = 2^{p_k}\cdot 1, 1\mathbf{)}$ то функція K' відповідно породжує послідовність

$$\mathbf{(}1/n_0, 1/(2^{p_1}\cdot n_1), 1/n_1, 1/(2^{p_2}\cdot n_2), 1/n_2, \ldots, 1/(2^{p_k}\cdot n_k), 1/2^{p_k}, 1\mathbf{)}.$$

Таким чином, якщо функція Коллатца на довільному числі n_0 досягає точки циклу 1, то і функція K' на оберненому до n_0 числі досягає точки 1, при чому за одну і ту же кількість кроків k. Перехід від функції K' до K відтворюється зворотнім чином.

Відмітимо також той факт, що точкою циклу (при умові справедливості гіпотези Коллатца!) для функції K є *найменше* натуральне

число в множині **N**, а для функції K' – відповідно *найбільше* число в множині $\{1/n \mid n > 0\}$.

Функція в множині від'ємних цілих чисел

Розглянемо функцію K'': $\{n \mid n < 0\} \rightarrow \{n \mid n < 0\}$, де $K''(n) = h''(n) = n/2$ для $n \in N_2$ і $K''(n) = g''(n) = 3n - 1$ для $n \in N_1$.

Стверджується, *що функція K'' є циклічною в точці* -1 *тоді і тільки тоді, коли функція Коллатца є циклічною в точці* 1.

Як в попередньому випадку послідовності

$$(n_0,\ 2^{p_1} \cdot n_1,\ n_1,\ 2^{p_2} \cdot n_2, \ldots,\ 2^{p_k} \cdot n_k = 2^{p_k} \cdot 1,\ 1),$$

де всі числа n_0, n_1, n_2, . . ., n_k додатні, ставимо у відповідність таку же послідовність, всі члени якої беруться зі знаком "-".

Основна відмінність функції K'' від K полягає в тому, що функція g'' від'ємне значення $3n$ *зменшує* на 1, а функція g (як складова функції K) відповідно *збільшує* $3n$ на 1. Така особливість цих функцій забезпечує рівність по абсолютній величині відповідних членів послідовностей, породжених функціями K і K''.

Функція в множині від'ємних обернених цілих чисел

Визначимо функцію K''': $\{1/n \mid n < 0\} \rightarrow \{1/n \mid n < 0\}$, де $K'''(1/n) = h'''(1/n) = 2/n$ для $n \in N_2$ і $K'''(1/n) = g'''(1/n) = 1/(3n - 1)$ для $n \in N_1$.

Стверджується, що функція K''' є циклічною в точці -1 тоді і тільки тоді, коли циклічною в точці 1 є функція Коллатца. Справедливість цього твердження випливає з порівняння послідовностей, породжених цими функціями.

5. Скінченно-збіжні функції, породжені послідовностями Гудстейна

В 1944р. англійський математик-логік Рубен Гудстейн висунув твердження [2] математичної логіки про натуральні числа, яке отримало великий резонанс серед спеціалістів математичної логіки в галузі досліджень теорем Геделя про неповноту елементарної арифметики (арифметики першого порядку – арифметики Пеано) [3, 4, 5]. Це

твердження, отримавши назву ”Теорема Гудстейна”, зводиться до наступної властивості запропонованої множини послідовностей натуральних чисел – *всі такі послідовності скінченні та закінчуються нулем*. Процедура побудови таких послідовностей наступна (наводимо спрощену процедуру):

1) довільне натуральне число $n > 0$ розкладається за степенями з основою 2, наприклад, $11 = 2^3 + 2 + 1$;
2) далі, таке зображення числа n трансформується в нове зображення шляхом заміни основи (на першому кроці це – 2) на основу, збільшену на 1, а отримане число зменшується на 1, наприклад, $11 = 2^3 + 2 + 1 \Rightarrow 30 = (3^3 + 3 + 1) - 1$.

Теорема Гудстейна стверджує, що довільне натуральнее число, починаючи з кроку 1), за скінченну кількість раз застосувань кроків 2) перетвориться в нуль. Механізм такого перетворення (за спрощеною процедурою) можна пояснити наступним чином. Нехай на якомусь кроці процедури поточне число n має вигляд $n = b_k a^k + \ldots + b_1 a + b_0$, де a – основа розкладу числа і всі коефіцієнти b_i менші a. Тоді після b_0 раз застосувань процедури отримаємо число $m = b_k (a + b_0)^k + \ldots + b_1 (a + b_0)$, а через ще один крок процедури - відповідно число

$$d = b_k (a + b_0 + 1)^k + \ldots + b_1 (a + b_0 + 1) - 1 =$$
$$= b_k (a + b_0 + 1)^k + \ldots + (b_1 - 1)(a + b_0 + 1) + (a + b_0),$$

розкладене за основою $a + b_0 + 1$. Тепер вільний член в такому розкладі числа $(a + b_0)$ буде менший за його основу. Далі знову через $(a + b_0)$ кроків отримаємо число виду

$$b_k (2(a + b_0) + 1)^k + \ldots + (b_1 - 1)(2(a + b_0) + 1).$$

Таким чином, зі зростанням основи розкладу поточного числа зменшується кількість членів такого розкладу, що приведе до утворення числа виду $b_k p^k$. Наприклад,

$5 = 2^2 + 1 \Rightarrow 3^2 + 1 - 1 = 3^2 \Rightarrow 4^2 - 1 = 3 \cdot 4^1 + 3 \Rightarrow$ (через 3 кроки) $\Rightarrow 3 \cdot (4+3) \Rightarrow$
$3 \cdot (4+3+1)^1 - 1 = 2(4+3+1)^1 + (4+3) \Rightarrow$ (через (4+3) кроки) $\Rightarrow 2(2(4+3)+1)^1$
$\Rightarrow 2(2(4+3)+2)^1 - 1 = (2(4+3)+2)^1 + (2(4+3)+1) \Rightarrow$ (через 2(4+3)+1 кроки) $\Rightarrow$
$(4(4+3)+3)^1 = 31 \Rightarrow$ (через 31 крок) $\Rightarrow 0$.

З цього прикладу видно, що стабілізація зростання поточного числа 31 відбулася, коли поточна основа розкладу числа зросла до 4(4+3)+3. Далі ця основа в розкладі числа буде входити з нулевою степенню.

Теорема Гудстейна насправді дає більш загальне твердження. Відповідно сама процедура узагальнюється: на кроці 1) за основою 2 розкладаються всі показники степенів 2, а показники останніх знову розкладаються за тією ж основою і т.д. В результаті при такому максимальному розкладі початкового числа в його записі будуть задіяні тільки числа 2 і 1. Далі, на кроці 2) збільшення основи на 1 відбувається на всіх рівнях степеневої ієрархії, наприклад, $n = 2^{2+1} + 2 + 1 \Rightarrow m = 3^{3+1} + 3 + 1$.

Відомо, що якщо крок 1) починати з довільного наперед фіксованого чиала *a*, то описана вище процедура побудови послідовностей все одно буде приводити до числа 0. Більше того, якщо в описаній вище процедурі (як в спрощеному, так і в узагальному варіантах) число 1 замінити на деяке фіксоване число p, то все одно через скінченну кількість кроків відповідна послідовність буде закінчуватися нулем.

З так описаної процедури (спрощеної чи узагальненої) випливає її детермінованість (кожен крок визначається однозначно), що дозволяє припустити наявність деякої функції, яка породжує такі послідовності. Із теореми Гудстейна випливає, що така функція буде скінченно-збіжною функцією з точкою збіжності 0. Оскільки послідовності Гудстейна починаються з чисел $n > 0$, то таку функцію можна перетворити в циклічну функцію довизначивши її для 0 рівним деякому числу (вибір може бути довільним!) відмінному від 0. Якщо порівняти таку циклічну функцію з функцією Коллатца (в припущенні, що його гіпотеза вірна), то виявляється принципіальна різниця між ними, а саме: довжина циклу для функції Коллатца фіксована і рівна 3 для кожного початкового числа, а для циклічної функції Гудстейна довжина циклів не є фіксованою і залежить від початкового числа.

Якщо замість початкової степеневої основи (в наведеній вище процедурі основа рівна 2) брати довільне число $a > 1$, а замість рівня степеневої ієрархії – число $k > 0$ (в спрощеному варіанті процедури таке число рівне 1), то можна з кожною так зміненою процедурою побудови послідовностей асоціювати деяку функцію G_a^k. Якщо рівень степеневої

ієрархії не обмежувати числом *k,* то відповідну функцію будемо записувати як G_a^*. Таким чином, можна розглядати цілий клас скінченно-збіжних (а, відповідно, і циклічних) функцій, навіяний ідеями Гудстейна. Стандартна процедура побудови послідовностей Гудстейна, яка наводиться в його теоремі, продукує функцію G_2^*. Із цієї теореми та результатів роботи [3] випливає, що довести скінченно-збіжність функції G_2^* засобами елементарної арифметики неможливо (Гудстейн використовує механізм трансфінітної індукції, що виходить за межі арифметики Пеано). Але для функцій виду G_a^k доведення їх циклічності можна, згідно з нашими припущеннями, обійтися засобами аксіом Пеано, зокрема методом звичайної математичної індукції.

Характерною властивістю функцій цього класу є те, що згідно процедурам побудови послідовностей, останні досить стрімко (і чим більше число *k*, тим стрімкіше) зростають до деякого максимуму (залежить від початкового числа), а потім дуже повільно за рахунок зменшення на 1 спадають до 0. Таке максимальне число в послідовності досягається тоді, коли поточна основа розкладу числа, зростаючи на 1, зрівнюється з самим числом. Після такого зрівняння подальше збільшення основи не приводить до зростання самого числа. На наш погляд, ключова складність Теореми Гудстейна полягає якраз в існуванні такого максимального члена послідовності для кожного початкового натурального числа.

Список використаних джерел

1. Jeffrey C.Lagarias. The Ultimate Challenge: The 3x+1 Problem. American Mathematical Society. 2010, – 344p.
2. Goodstein R. On the Restricted Ordinal Theorem. Journal of Symbolic. 1944, T.9: 33-41 .
3. Kirby L., Paris J. Accessible independence results for Peano arithmetic. Bulletin London Mathematical Society. 1982, T.14: 285-293.
4. В.В. Целищев, А.В. Бессонов. Является ли теорема Гудстейна Геделевым предложением. Проблемы логики и методологии науки. НГУ: УДК 160.1 DOI: 10.15372/PS20170202.
5. Пенроуз Р. Теорема Гудстейна и математическое мышление// Пенроуз Р. Большое, малое и человеческий разум. – М.:Мир, 2004, – с.180-184.

Дуцяк Ігор Зенонович, д.філос.н, професор

ПРО КОРЕКТНІСТЬ ФОРМАЛІЗАЦІЇ ПРАВОВОГО ПРИНЦИПУ «НЕЗАБОРОНЕНЕ ДОЗВОЛЕНО» В ДЕОНТИЧНІЙ ЛОГІЦІ

У дослідженні обґрунтовано засади деонтичної логіки, узгоджені з правом. В основу побудованої деонтичної системи покладено такі принципи: 1) є три модальні оператори (обов'язково, заборонено, дозволено); 2) всі вони є взаємонесумісні; 3) дозвіл виконання дії сумісний з дозволом її невиконання. Деонтиччні вислови з модальністю «дозволено» уміщуються в деонтичну логіку з асерторичної логіки. Для цього асерторичні вислови трансформують в деонтичні.

Ключові слова: правові норми, деонтична логіка, деонтичні оператори, відносини між деонтичними модальностями, принцип «незаборонене – дозволене».

In the study substantiates the principles of deontic logic, consistent with law. The constructed deontic system is based on the following principles: 1) there are three modal operators (obligatory, forbidden, permitted); 2) they are all mutually incompatible; 3) the permission to perform an action is compatible with the permission not to perform it. Deontic expressions with the modality "permitted" are placed in deontic logic from asertoric logic. For this, asertoric sentences are transformed into deontic ones.

Key words: legal norms, deontic logic, deontic operators, relations between deontic modalities, the principle "not forbidden - allowed".

Однією з умов застосовности формальних деонтичних систем в юридичній практиці є коректність формалізації відносин між видами правничих норм (зобов'язувальні, заборонювальні, дозволювальні). Водночас є підстави для думки, що в сучасних формальних деонтичних системах відносини між деонтичними операторами можуть бути не завжди коректно узгоджені з правничим тлумаченням відносин між нормами. Для прикладу, в багатьох запропонованих деонтичних системах використовують крім операторів «обов'язково», «заборонено» і «дозволено», оператор «байдуже» (в тому сенсі, що дія може виконуватись, а може не

виконуватись). Водночас у правничому регулюванні суспільних відносин саме можливість вибору (виконувати чи не виконувати якусь дію) тлумачать як дозволене. Зважаючи на цю та інші невідповідності між значенням термінів у правничих і логічних деонтичних системах, важливо виявити неузгодженість логічних відносин з модельованими ними правничими відносинами та увідповіднити їх. Зазначена проблема стала предметом дослідження, результати якого містяться в цьому викладі.

На сьогоднішній день дослідниками запропоновано велику кількість деонтичних систем, з багатьма з яких можна ознайомитися в працях [1–10]. У цій публікації обмежимося лише викладом головних засад деонтичної системи, яка була б узгодженою з відносинами між видами норм («обов'язково», «заборонено» і «дозволено») в галузі права. Виконаємо такі пізнавальні дії: 1) стисло викладемо погляди правників на відносини між нормами «обов'язково», «заборонено» і «дозволено»; 2) відтворимо відносини між видами правових норм у вигляді формальних відносин між деонтичними операторами.

Обґрунтування засад побудови формальної деонтичної системи, узгодженої з правовою системою

За основу для актуалізації відносин між правовими нормами візьмемо підходи Ганса Кельзена [11]. З-посеред особливостей правових норм головними виокремимо такі:

1. Будь-яка правова норма є повинністю в тому сенсі, що кожна норма нав'язана нормодавцем адресатам цієї норми. Нормодавець нав'язує, щоб якась діяльність була обов'язкова, якась – заборонена і якась – дозволена. У цьому сенсі нормодавець примушує і до обов'язковости чогось, і до заборони чогось, і до дозволености чогось.

2. На відміну від буття природних фактів (вислови про згадане буття ми оцінюємо за ознакою фактичної істинности – *І. Д.*), специфічне існування норми треба розуміти як її «чинність» (чинна норма – це норма, згідно з якою повинна відбуватись діяльність в якихось конкретних умовах, в якомусь конкретному просторі і періоді). На підставі цього можемо прийняти, що в деонтичній логіці треба діяти подібно до того, як діють в

асерторичній логіці. В асерторичній логіці висловам присвоюють значення «фактично істинно» і «фактично хибно» (на підставі зіставлення змісту цих висловів з дійсністю) і далі на цій підставі з'ясовують значення логічної істинности. Подібно до цього, в деонтичній логіці треба висловам з деонтичними операторами (вони не є ані істинними, ані хибними) треба присвоїти значення «чинне» (тобто таке, що міститься у вигляді правової норми в модельованій правовій системі) або «нечинне». Після цього можна виконувати операції подібні до тих, які виконують в асерторичній логіці, – з'ясовувати випадки несумісності норм та дедукування з явно зафіксованих норм тих, які не є явними, однак логічно необхідними.

3. Є два способи правового регулювання: позитивне (за допомогою явно сформульованих норм) і негативне (за допомогою неявно прийнятих норм у разі, коли норми явно не сформульовані).

4. «Коли норма дозволяє людині чинити певну дію, що в іншому випадку забороняється», то «саме ця норма обмежує сферу чинності тієї норми, яка забороняє дію» [Там само, С. 26]. До щойно вміщеної думки Ганса Кельзена доцільно додати подібне щодо дозволу на невиконання обов'язкових дій – дозвіл невиконувати дію, яка в іншому випадку є обов'язковою, обмежує сферу дії зобов'язувальної норми[*]. Звідси можна виснувати, що дозвіл може бути лише там, де є примус (зобов'язувальний чи заборонювальний). Наприклад, якщо в концентраційних трудових таборах кожна людина зобов'язана виконувати трудові дії, то в окремих випадках хтось може отримати дозвіл на, скажімо, тимчасове невиконання трудових дій.

[*] Оскільки ми дозволяємо не тільки виконувати дію у разі її заборони в загальному випадку, але й не виконувати дію у разі її обов'язковости в загальному випадку, є підстави не погодитись з дещо звуженим підходом Г. Кельзена: «Поведінка, яка не забороняється з правової точки зору, водночас дозволяється (в цьому негативному розумінні) з правової точки зору. Оскільки певна людська поведінка або забороняється, або не забороняється (а оскільки не забороняється, то має розглядатись як дозволена правовим порядком), то будь-яка поведінка людей, що перебувають у правовому полі, може розглядатись як така, що регулюється цим правопорядком, – у позитивному чи негативному розумінні» [Там само, С. 55]. З огляду на згадане на початку цієї примітки, цитований текст треба узагальнити замінивши слово «заборона» на «примус» – у такому разі йтиметься і про дозвіл на виконання забороненого, і про дозвіл на невиконання обов'язкового.

5. Невнормовані дії прийнято тлумачити як негативно дозволені: «Нормативний порядок регулює людську поведінку негативним способом тоді, коли ця поведінка не заборонена порядком, хоча її й не дозволено нормою, яка обмежує сферу чинности забороняючої норми. Отже, поведінка в цьому випадку дозволяється лише в негативному значенні» [Там само]. Цю думку висловлено в іншому місці таким чином: «Поведінка, яка не забороняється з правової точки зору, водночас дозволяється (в цьому негативному розумінні) з правової точки зору» [Там само, С. 55].

6. Позитивні дозволи не є самостійними нормами, оскільки ними обмежують дію примусових норм (отже будь-яку дозволювальну норму, зафіксовану в явному вигляді нормативним актом, треба тлумачити як частину примусової норми). Згідно з цими міркуваннями Г. Кельзена, можна виокремити три види норм: 1) позитивні, тобто сформульовані в явному вигляді нормами права, зобов'язувальні норми, 2) позитивні заборонювальні норми і 3) негативні дозволювальні норми, тобто не сформуловані в явному вигляді, але прийняті за умовчанням як дозволені. Всі ці види норм є взаємонесумісними згідно з такими обставинами: оскільки примусові норми (зобов'язувальні) не дають змоги вибору між варіантами виконувати дію чи ні (як це є згідно з дозволювальною нормою), то кожна з примусових норм не може бути сумісною з дозволювальною нормою. Зобов'язувальна і заборонювальна норми є несумісними, оскільки онтологічно неможливо водночас примушувати виконувати якусь дію і водночас не виконувати її.

7. Згідно з попереднім пунктом невнормованих дій у суспільній практиці нема. Явно сформульовані норми – це примусові норми (як було зазначено, явно сформульовані дозволи не є самостійними нормами – це лише межі дії примусових норм). Дії, щодо яких нема явно сформульованого примусу є внормовані негативно як дозволені.

Перш ніж формулювати відносини між правовими нормами в символьному вигляді зафіксуймо прийняті засади побудови деонтичної системи. Порівняймо три типи речень: стверджувальні, питальні і спонукальні. Стверджувальне речення може відрізнятися від питального тільки знаком пунктуації в кінці речення. Наприклад, «Машина стоїть.» і «Машина стоїть?». Отже є змістова частина речення (пропозиція) і знак,

яким позначають в одному випадку, що ми подали вислів як істинний (тоді він є об'єктом аналізу асерторичної логіки), а в іншому випадку як запит на отримання знань (тоді, в разі знака питання, він є об'єктом аналізу еротетичної логіки). Це стосується також деонтичної логіки. Отже, вживаючи в послідовності символів символ *a*, позначатимемо ним не стверджувальний вислів (який може бути істинним чи хибним), а лише пропозицію, змістом якої є називання якоїсь дії, діяльности. Коли перед згаданим символом записуємо один з операторів (*O* – обов'язково, *F* – заборонено, *P* – дозволено), то ними позначаємо, що отримуємо відповідну норму (обов'язковою, забороненою чи дозволеною є ця дія чи діяльність). Булевими функціями позначатимемо чинність чи нечинність норми (тобто чи є така норма в модельованій правничій системі, як про це було зазначено в пункті 2 раніше поданого переліку). Фомулами вважатимемо тільки ті вирази, які містять деонтичний оператор перед символом пропозиції, якою позначено якусь дію чи діяльність (*Oa, Fa, Pa*) а також двоаргументні булеві функції. Згідно з зазначеним сформулюємо відносини між правовими нормами у символьному вигляді.

1. Відносини між нормами з однаковою модальністю однак різними діями: виконання якоїсь дії (*a*) і невиконання цієї дії (¬*a*). У цих формулах виразно проявлються відмінності між сильними і слабкою деонтичними модальностями.

1.1. Несумість обов'язковости виконувати якусь дію з обов'язковістю не виконувати цю дію:

$$\neg (Oa \wedge O\neg a). \tag{1}$$

1.2. Несумість заборони виконувати якусь дію з забороною не виконувати цю дію:

$$\neg (Fa \wedge F\neg a). \tag{2}$$

1.3. Сумісність дозволу виконувати якусь дію з дозволом не виконувати цю дію. Цю норму нема змоги записати подібним чином як попередні, оскільки вираз ($Pa \wedge P\neg a$) є суперечністю. Тому, аби мати змогу використовувати формальний апарат логіки висловів, запишемо це відношення у вигляді тотожности:

$$Pa =_{\text{Def}} P\neg a. \quad (3)$$

2. Взаємна несумісність норм.

2.1. Обов'язкова діяльність несумісна з дозволеною діяльністю.

$$\neg(Oa \wedge Pa). \quad (4)$$

2.2. Заборонена діяльність несумісна з дозволеною діяльністю.

$$\neg(Fa \wedge Pa). \quad (5)$$

2.3. Обов'язок виконувати якусь дію несумісний із забороною виконувати цю дію.

$$\neg(Oa \wedge Fa). \quad (6)$$

Якщо побудувати таблицю істинности кон'юнкції формул 4-6, то виявиться, що можливим є також варіант, коли водночас нечинною є жодна з норм Oa, Fa, і Pa. Для усунення цього варіанту до аксіом 4-6 достатньо додати формулу $(Oa \vee (Fa \vee Pa))$ або будь-яку еквівалентну їй, наприклад $\neg(\neg Oa \wedge (\neg Fa \wedge \neg Pa))$.

Водночас записані аксіоми можуть мати інший (але тотожний наведеному) формальний запис. Це може бути, скажімо, диз'юнктивна нормальна форма, яку (прийнявши для спрощення запису скорочення $Oa \equiv o$; $Fa \equiv f$; $Pa \equiv p$) запишемо у вигляді

$$((o \wedge \neg f) \wedge \neg p) \vee ((\neg o \wedge f) \wedge \neg p) \vee((o \wedge f) \wedge \neg p).$$

Те саме можна записати відповідною запереченою кон'юктивною нормальною формою, тобто $((o \veebar f) \veebar p) \wedge \neg((o \wedge f) \wedge p)$, де символом $\veebar$ позначено сильну диз'юнкцію, чи довільною іншою еквівалентною формулою.

3. Той факт, що з трьох висловів чинним може бути тільки один, зручно формалізувати геометричною аналогією у вигляді трикутника і тіла, яке може перебувати тільки в одній з його вершин (чинною може бути тільки одна з трьох норм). У такому разі маємо два види правил:

3.1. Відносини між деонтичними операторами згідно з принципом: «якщо ми не перебуваємо в якійсь конкретній вершині трикутника, то ми перебуваємо в одній з решти двох»:

$$(\neg Oa \leftrightarrow (Fa \dot{\vee} Pa)); \quad (7)$$

$$(\neg Fa \leftrightarrow (Oa \dot{\vee} Pa)); \quad (8)$$

$$(\neg Pa \leftrightarrow (Fa \dot{\vee} Oa)). \quad (9)$$

3.2. Відносини між деонтичними операторами згідно з принципом: «якщо ми перебуваємо в якійсь конкретній вершині трикутника, то ми не перебуваємо в жодній з решти двох» (кожні дві норми можуть бути водночас нечинними):

$$(Oa \leftrightarrow (\neg Fa \wedge \neg Pa)); \quad (10)$$

$$(Fa \leftrightarrow (\neg Oa \wedge \neg Pa)); \quad (11)$$

$$(Pa \leftrightarrow (\neg Fa \wedge \neg Oa)). \quad (12)$$

4. На підставі згаданого принципу можна сформулювати також інші зв’язки між деонтичними операторами. Зокрема, можна додати такі принципи:

4.1. Дія не дозволена у двох випадках – коли вона обов'язкова і коли вона заборонена, звідси маємо (вислів Pa не чинний у двох випадках):

$$Oa \rightarrow \neg Pa;$$

$$Fa \rightarrow \neg Pa.$$

4.2. Дія не заборонена у двох випадках – коли вона обов'язкова і коли вона дозволена, звідси маємо:

$$Oa \rightarrow \neg Fa;$$

$$Pa \rightarrow \neg Fa.$$

4.3. Дія не обов'язкова у двох випадках – коли вона дозволена і коли вона заборонена, звідси маємо:

$$Pa \rightarrow \neg Oa;$$

$$Fa \rightarrow \neg Oa.$$

Щойно було записано низку відносин між висловами з деонтичними операторами для випадків, коли нормується якась дія (a). До них треба

додати правила, які стосуються висловів про відсутність дії ($\neg a$). Можна стверджувати, що всі подані щойно відносини щодо виконання дії будуть чинними також для відносин між нормами щодо невиконання дії. Подібно до того, як ми можемо примушувати до виконання дії, забороняти виконання дії чи дозволяти виконання дії, ми можемо примушувати до виконання бездіяльности, забороняти виконання бездіяльности чи дозволяти виконання бездіяльности. У такому разі у відносинах 7-12 можна поміняти a на $\neg a$:

$$(O\neg a \leftrightarrow (\neg F\neg a \wedge \neg P\neg a));$$
$$(F\neg a \leftrightarrow (\neg O\neg a \wedge \neg P\neg a));$$
$$(P\neg a \leftrightarrow (\neg F\neg a \wedge \neg O\neg a));$$
$$(\neg O\neg a \leftrightarrow (F\neg a \dot{\vee} P\neg a));$$
$$(\neg F\neg a \leftrightarrow (O\neg a \dot{\vee} P\neg a));$$
$$(\neg P\neg a \leftrightarrow (F\neg a \dot{\vee} O\neg a)).$$

Водночас щойно записані відносини легко отримати, взявши до уваги виознаку 3, тобто $Pa =_{\text{Def}} P\neg a$, і прийнявши еквівалентності:

$$Fa \leftrightarrow O\neg a; \quad (13)$$
$$Oa \leftrightarrow F\neg a. \quad (14)$$

У такому разі всі відносини між деонтичними модальностями можна подати графічно у вигляді трикутника (Рис. 1).

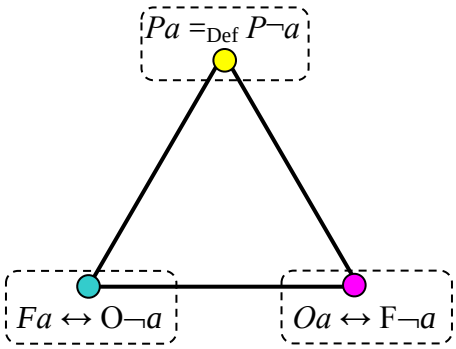


Рисунок 1. Трикутник відносин між деонтичними модальностями (лініями позначено несумісність)

Згідно з викладеними міркуваннями для формального запису правової засади «незаборонене – дозволено» потрібно утворити асерторично-деонтичну систему, в якій були б і стверджувальні вислови, і деонтичні вислови. Нехай маємо вислів про існування якоїсь дії чи діяльности. Це асерторичний вислів, який мовець подає як істинний. Оскільки, є пропозиція (*a*), зміст якої містить інформацію про виконання якоїсь дії, то в асерторичній логіці поставимо перед нею оператор істинности *T* (кожен стверджувальний вислів актом ствердження, зокрема крапкою в кінці речення, ми в усіх випадках подаємо як істинний). Отримаємо запис стверджувального вислову про виконання дії *a* у вигляді *Ta*. Якщо в правовій системі відсутня явно зафіксована примусова норма (обов'язковість чи заборона виконання цієї дії), то її, як введену негативною правовою нормою «дозволено» треба ввести в деонтичну логічну систему. Для цього у стверджувальному вислові треба вилучити оператор істинности (який ми для спрощення ніколи не позначаємо в асерторичній логіці) і замість нього поставити опеатор «дозволено» (*Pa*). Таким чином вислів асерторичної логіки перетворюємо у вислів деонтичої логіки (*Ta* ⇨ *Pa*), виражаючи засаду «незаборонене – дозволено». Треба зауважити, що для практичного використання деонтичної формальної системи потрібно поєднувати її з асерторичною і з інших міркувань – адже потрібно позначити, що якась конкретна дія виконана або ні, зіставити її з правовою нормою (формально записаною деонтичним висловом), і тоді робити висновок (у разі співпадіння виконуваного з повинним) – правова дія не потрібна, а в разі неспівпадіння – повинна виконуватись санкція.

Висновки. Перш ніж будувати універсальну деонтичну систему, яка охоплюватиме як окремі сегменти моделі регулювання в галузях права, моралі та ін., доцільно розробити формальні деонтичні системи, які максимально адекватно описують кожен з цих специфічних сегментів.

Список використаних джерел

1. Deontic logic: introductory and systematic readings / Risto Hilpinen. Dordrecht, Holland : D. Reidel Publishing Company, 1981. 197 p.
2. New studies in deontic logic: Norms, Actions, and the Foundations of Ethics / Risto Hilpinen. Dordrecht, Holland : D. Reidel Publishing Company, 1981. 263 p.
3. Fred Feldman. Doing the best we can: An Essay in Informal Deontic Logic. Dordrecht , Holland : D. Reidel Publishing Company, 1986. 255 p.
4. Defeasible deontic logic / Donald Nute. Dordrecht: Springer Science+Business Media, B.V., 1997. 360 p.
5. Lambèr M.M. Royakkers. Extending deontic logic for the formalisation of legal rules. Eindhoven : Springer Science+Business Media, B.V., 1998. 198 p.
6. John F. Horty. Agency and deontic logic. New York : Oxford University Press, 2001. 205 p.
7. Handbook of Philosophical Logic / Dov M. Gabbay, F. Guenthner. Netherlands: Kluwer Academic Publishers, 2002. 366 p.
8. Handbook of the History of Logic. Vol. 7. Logic and the Modalities in the Twentieth Century / Dov M. Gabbay, John Woods. Netherlands : Elsevier, 2006. 733 p.
9. Handbook of Deontic Logic and Normative Systems / Dov Gabbay, John Horty, Xavier Parent, Ron van der Meyden, Leendert van der Torre.: Lightning Source, Milton Keynes, 2013. 648 p.
10. Pablo E. Navarro, Jorge L. Rodríguez. Deontic Logic and Legal Systems. New York : Cambridge University Press, 2014. 288 p.
11. Кельзен Г. Чисте правознавство. З додатком: Проблема справедливості. Київ: Юніверс, 2004. 496 с.

Зубенко Віталій Володимирович, к.ф.-м.н, доцент

ПРО КОНЦЕПЦІЮ АЛГОРИТМУ В КУРСАХ ТЕОРІЇ АЛГОРИТМІВ ТА ПРОГРАМУВАННЯ

Робота присвячена математичному уточненню загального поняття алгоритму. Ми прагнемо відійти від прийнятої в традиційних курсах теорії алгоритмів та програмування практики визначати якийсь конкретний клас алгоритмів (машини Тюрінга, алгоритми Маркова, регістрові машини тощо) і, посилаючись на його універсальність, пов'язувати з ним загальне поняття алгоритму. Такий підхід, оснований на моделюванні, прийнятний і корисний в багатьох відношеннях, на жаль, не дозволяє підійти ні до уточнення предмету інформатики, ні до інтегрування чисельних моделей обчислень, які використовуються в теорії алгоритмів та програмуванні [1]. Наш підхід до формалізації поняття алгоритму протилежний і базується на уточненні більш загального поняття обчислювальної процедури та звуженні його до формального поняття алгоритму. Розглядається місце основних математичних моделей алгоритмів та обчислень в межах процедурної концепції алгоритму.

Ключові слова: процедура, алгоритм, обчислювальна процедура, розширена процедура, процедурний алгоритм.

The work is devoted to mathematical clarification of the general concept of algorithm. We strive to move away from traditional algorithm theory and programming practices to define a specific class of algorithms (Turing machines, Markov algorithms, register machines, etc.) and, referring to its universality, relate the general concept of the algorithm to it. This approach, based on modeling, is acceptable and useful in many respects, unfortunately, does not allow to approach either the refinement of the subject of computer science or the integration of numerical computational models used in the theory of algorithms and programming [1]. Our approach to the formalization of the concept of algorithm is opposite and is based on the clarification of a more general concept of computational procedure and narrowing it to the formal concept of algorithm. The place of the basic mathematical models of algorithms and calculations within the procedural concept of algorithm is considered.

Keywords: procedure, algorithm, computational procedure, extended procedure, procedural algorithm.

1. Як відомо, термін алгоритм походить від імені перського математика IX ст. Аль-Хорезмі, який вперше в систематичному вигляді описав арифметичні операції в 10-й позиційній системі і сформулював правила для їх виконання в тому вигляді, в якому ми з вами сьогодні ними користуємося і називаємо алгоритмами додавання, множення тощо. Пізніше цим терміном стали називати усі чітко сформульовані правила для отримання потрібних результатів, виходячи з конкретних вхідних даних з певної сукупності, за допомогою строго визначеної послідовності дії. Ця послідовність отримала назву обчислення за алгоритмом або на програмістському жаргоні – роботою алгоритму.

Насправді, термін алгоритм є частковим випадком більш широкого поняття – процедури, яке походить від французьких слів procedure (процедура, операція, процес) та procedere (діяти в часі, рухатися вперед). Кому не відомі медичні, юридичні, танцювальні, музичні, ігрові чи кулінарні процедури (останні в побуті називають кулінарними рецептами) або функції-процедури мов програмування.

Процедури, як і алгоритми, є інструкціями для виконання дій з певною метою. При цьому вони можуть не повертати конкретний результат. У цьому випадку сенсом роботи процедури є сам процес обчислення, як це має місце, наприклад, при виконанні танцю чи музичного твору. Основна відмінність між процедурами та алгоритмами полягає у вимогах до способу їхнього подання. Як правило, алгоритми потребують формального чи формалізованого[1], обов'язково скінченного за розміром, опису і якщо це фрагмент природньої мови, то він має чітко й однозначно трактуватися. Прикладами такої точності опису можуть слугувати згадані вище арифметичні правила чи запис шахової партії. Процедури часто описують природною мовою, конструкції якої можуть допускати різну інтерпретацію і бути не повними. Наприклад, у різних господарок за одним і тим же рецептом борщу результати на виході (на смак) будуть, як правило, різні, а от річницю держави України 2091-1991 за алгоритмом віднімання вони отримають однакову – 100.

[1] Формалізована мова - це мова з елелементами формальної мови.

Якщо поглянути на процедури історично, то потреба в діях та їхньому описі відображена вже в перших природніх мовах. Найпростіші з дій – будемо називати їх далі елементарними - представлені в мовах дієсловами, які можуть супроводжуватися супутніми конструкціями, що уточнюють їхню дію. Наприклад, зробити крок повільніше або в певному напрямку тощо. Більш складні дії складаються з сукупності елементарних операцій, виконуваних в певній послідовності. Вони називаються *складеними* і подаються одним або кількома реченнями. Складена дія може й не закінчуватись, а продовжуватися нескінченно, наприклад, коли вона “зациклюється” як при наказі крокувати по колу.

З розвитком абстрактного мислення, а з ним і природних мов, з’явилася можливість описувати дії, не прив’язуючись жорстко до контексту їхнього виконання, а навпаки – абстрагуючись від нього. Насамперед, це стосується конкретних значень вхідних даних. У цьому випадку останні подаються в процедурах у вигляді *параметрів* – імен вхідних даних та типом їхніх можливих значень, а не самими значеннями. Подібні параметризовані процедури описують цілий клас однотипних дій, що відповідає усім можливим варіантам значень параметрів. Щоб виконати дію, яку описує параметризована процедура, потрібно актуалізувати її параметри, тобто надати їм конкретні значення. Параметризуватися можуть і цілі внутрішні фрагменти процедур з наступною їхньою актуалізацією в процесі виконання. Параметри можуть бути присутні в процедурах неявно як у згадуваних вже арифметичних правилах Аль-Хорезмі. Кожне з них має два вхідні параметри: ”перше число” та ”друге число”, які представляють десяткові числа.

Сформулюємо тепер вимоги до дефініцій процедур. Вони є необхідними для розуміння конкретної процедури та її дії.

1) Якщо в дефініції є параметри, то обов’язково мають бути означені всі їхні можливі значення.
2) Повинна бути описана кожна елементарна дія процедури.
3) Повинен бути описаний порядок виконання елементарних дій для кожного конкретного варіанту вхідних даних.

4) Необхідно задати механізм завершення роботи процедури і визначення результату, якщо він є.

Якщо тепер до цих чотирьох вимог додати вимогу 5) про те, що подання процедури має бути вичерпним і скінченним, то отримаємо інтуїтивне означення алгоритму.

Наведеного означення, цілком достатньо, якщо мова йде про окремо взяті процедури чи алгоритми та проведення за ними обчислень. Але в середині XIX ст. виникли дві обставини, які докорінно змінили ситуацію і вимагали розгляду вже не окремих алгоритмів, а їхніх сукупностей і навіть всю їхню можливу сукупність.

В 1834-37 рр. англійський математик Ч.Бебідж запропонував проект Аналітичної машини, яка за задумом мала самостійно проводити числові обчислення за спеціальними програмами-алгоритмами, поданими у певній мові (мові програмування). Машина не була побудована, але вона привела до загальної ідеї автомата з програмним керуванням (АПК), яка незабаром знайшла своє теоретичне обґрунтування і практичну реалізацію. В моделях АПК мови програмування мають бути формальними для того, щоб їхні програми можна було однозначно трактувати і автоматично виконувати.

В зв'язку з автоматами з програмним керуванням виникає питання про їхні обчислювальні можливості. Очевидно, що відповідь на нього залежить від класу програм, які здатен виконувати конкретний АПК. Більше того, природно виникає і загальне питання чи існує модель універсального АПК, здатного реалізувати дії за будь-яким алгоритмом, не обов'язково числовим. Щоб дати на нього відповідь, потрібно точно сформулювати, що таке алгоритм. Іншими словами, коли мова йде про конкретний алгоритм і його виконання, то спеціальних вимог до його подання немає, хіба що він має бути зрозумілим для виконавця. А от створити мову програмування для універсального числового АПК означає вже дати формальний опис усіх можливих алгоритмів.

Друга і вирішальна обставина пов'язана зі створенням у XIX ст. формальних числень і аксіоматизацією математичних теорій, зокрема, арифметики (аксіоматика Пеано). З появою точного означення арифметичної теореми та її доведення стало можливим запитати не тільки

чи буде довільне арифметичне твердження теоремою чи ні, а й і чи існує алгоритм для відповіді на нього. На початку XX ст. в арифметиці накопичилася велика купа відкритих проблем-тверджень, про які не було відомо чи є вони теоремами чи ні. Враховуючи, що частина з них були надскладними і перебували в статусі відкритих десятиліттями і навіть століттями, то виникали серйозні підстави вважати, що загального алгоритму для перевірки чи буде довільна арифметична формула теоремою чи ні, не існує. Але щоб строго це довести, потрібно мати математичну модель поняття алгоритму.

Таким чином, на початку XX ст. перед математичною спільнотою постала нагальна проблема математичного уточнення поняття алгоритму.[2] В 30-х роках XX ст. ця проблема була розв'язана в роботах А.Чорча, К.Гьоделя, А.Тьюрінга, Е.Поста, С.Кліні, які запропонували перші формальні моделі алгоритмів і алгоритмічно обчислювальних функцій і заклали тим самим підвалини сучасної теорії алгоритмів (ТА).

Зазначимо, що список моделей алгоритмів постійно поповнюється і налічує вже не один їх десяток, а якщо враховувати й формальні моделі мов програмування, то їхнє число буде ще більше. Сама ж теорія алгоритмів давно вийшла за межі проблематики математичної логіки і стала окремою математичною дисципліною.

2. Як не парадоксально, не дивлячись на фундаментальні результати, сучасна ТА все ще не має загального формального означення свого предмету – алгоритму. Конкретні модельні приклади, які б вони не були цікаві самі по собі, не можуть замінити собою відсутність загального поняття. Таке уточнення мало б концептуально упорядкувати й об'єднати

---

[2] Нагадаємо принагідно, що серед 23 знаменитих відкритих проблем, проголошених Д.Гільбертом в 1900 році як особливо важливими для подальшого розвитку математики, 10-та була алгоритмічною і стосувалася пошуку алгоритму для розв'язку діофантових рівнянь в цілих числах. Вочевидь, формулюючи дану проблему, Д.Гільберт вважав, що такий алгоритм існує, адже алгоритмічно нерозв'язні проблеми на той час ще не були відомі. Але з'ясувалося, що такого алгоритму немає. (Ю.Матиясевич, 1970).

всі конкретні моделі алгоритмів, показати їхнє місце в загальній картині.[3] Це тим більш актуально, що в зв'язку з бурхливим розвитком ІТ-технологій виникають нові моделі та парадигми обчислень і вони теж чекають на своє місце та статус в ТА.

Причина відсутності загального означення алгоритму – на поверхні. Моделі алгоритмів нерозривно пов'язані з конкретними мовами їхнього подання, а скільки моделей – стільки й мов. Щоб обійти цю першопричину, спробуємо піти іншим шляхом і спочатку математично уточнити більш загальне поняття процедури (причому зробити це на максимально можливому рівні абстракції), а потім вже, відштовхуючись від нього, повернутися до алгоритмів. Як виявилося, математично уточнити поняття процедури простіше, адже в них, на відміну від алгоритмів, не йдеться про якісь обмеження щодо подання [2].

Якщо взяти обчислення за процедурами, то вони розгортаються в часі і супроводжується виконанням певних дій. В класичній ТА час T – *дискретний*, його моменти ототожнюють з натуральними числами 0,1,2… і говорять про нульовий крок обчислення, перший, другий, тощо. Дії в обчисленнях відбуваються над *станами* – абстрактними не структурованими об'єктами з певної сукупності S. Стани представляють в процедурах інформаційну складову обчислень. Серед станів виділяють підмножину $S_0 \subseteq S$ *вхідних* станів. Трійка $\Pi = (T, S, S_0)$ називається *обчислювальним простором,* а упорядковані пари $(t, s) \in T \times S$ – його *конфігураціями.* Кожному обчислювальному простору відповідає сукупність скінченних і нескінченних *процесів* – відображень вигляду $p : T \to S$. Процес p будемо подавати послідовностю станів $s_0, s_1, \ldots$, де $s_i = p(i), i \geq 0$ і зображати словом $s_0 s_1 \ldots$ в алфавіті S. Позначимо S^+ та S^∞ – сукупності усіх непорожніх відповідно скінченних та нескінченних слів в алфавіті S.

[3] В певному сенсі існуючі моделі алгоритмів об'єднані, адже задовільняють властивостям 1)-5) і обчислюють один і той же клас числових функцій – клас ЧРФ. Але мова йде про їхнє понятійне, родове об'єднання, а не на рівні семантики.

Ми хочемо серед усіх процесів виділити сукупність обчислювальних процесів. Дії на станах будемо розглядати як певні багато значні операції на них. Зафіксуємо певну сукупність $\Delta = \{ f_i : S \to S, i = \overline{1,n} \}$ *елементарних операцій* на станах. Центральним елементом процедур є *функція керування* або *переходів* $\delta : T \times S \to \Delta$, яка регулює порядок застосування операцій в обчисленнях. Вона пов'язує з кожною конфігурацією обчислення одну або кілька можливих елементарних операцій на її стані. Функція керування теж може бути багато значною.

Обчислювальною процедурою у просторі Π із сукупністю елементарних операцій Δ і функцією керування δ називається трійка $P = < \Pi, \Delta, \delta >$.

Кожна обчислювальна процедура породжує певну сукупність обчислювальних процесів у просторі Π. Домовимося обчислювальні процеси називати *обчисленнями*. *Обчислення* $p : T \to S$ за процедурою P з початковим станом s визначається рекурентно:: $p_0 = s$ і для всіх $t > 0$ $p_t = f_t(p_{t-1})$, де f_t – одне зі значень функції керування δ на конфігурації (t, p_{t-1}).

Функція керування за поточним моментом часу t і попереднім станом p_{t-1} визначає сукупність елементарних операцій, які можуть бути застосовані до цього стану для отримання стану p_t. Будемо говорити, що обчислення p у момент часу t *переходить* від попереднього стану p_{t-1} до наступного стану p_t. Варіантів таких переходів може бути декілька, оскільки функція керування, як і самі елементарні операції, є багатозначними. Насправді, їх може бути: а) два і більше (*недетерміновані* процедури); б) рівно один (*детерміновані* процедури); в) жодного (функція керування чи елементарна операція не визначені).

Зазначимо, що з кожним обчисленням за процедурою пов'язана відповідна послідовність конфігурацій, а саме $(0, s), \ldots, (t, p(t)), \ldots$, а також послідовність операцій $f_1, \ldots, f_t, \ldots$.

Будемо називати процедури, функція керування яких реально залежить: а) тільки від часової компоненти *темпоральними*, б) тільки від стану – *автоматними* і в) коли таке керування залежить від обох компонент конфігурацій – *змішаними*

Тепер щодо завершення обчислень і визначення їхніх результатів. Основних механізмів завершення обчислень – два. Перший – коли в процесі обчислення функція керування чи елементарна операція на поточній конфігурації не визначена. Тоді обчислення припиняється і стан останньої конфігурації проголошується результатом обчислення.

Другий механізм пов'язаний з виділенням апріорі серед усіх станів процедури P підмножини *вихідних* $S_f \subseteq S$ станів. Будемо визначати обчислювальні процедури з вихідними станами як четвірки $P = < \Pi, \Delta, \delta, S_f >$. В таких процедурах обчислення закінчується, коли воно перший раз попадає у вихідний стан, який, як і в попередньому випадку, проголошується *результатом* обчислення. В обох випадках результат обчислень на вході s позначається $P^*(s)$. Він є одно елементним у випадку детермінованих процедур.

Якщо в повному обчисленні зняти умову, щоб усі проміжні стани були не вихідними, то таке обчислення називається *гіперрезультативним*. Його результат позначається $P^{**}(s)$. Про відповідності $P^*: S_0 \to S$, $P^*: S_0 \to S_f$ та $P^{**}: S_0 \to S_f$ говорять, що їх *обчислює* процедура P.

Часто цікавлять не самі відповідності, які обчислює процедура, а їхні проекції на складові станів. Нехай B – довільна сукупність елементів, які назвемо *складовими* станів процедури, а відображення вигляду $pr: S \to B$ – *функцією-проекцією* станів. Покладемо $B_0 = pr(S_0)$ та $B_f = pr(S_f)$. Про відповідності вигляду $R^*: B_0 \to B_f$ та $R^{**}: B_0 \to B_f$, такі що $R^*(b) = pr(P^*(pr^{-1}(b)))$ та $R^{**}(b) = pr(P^{**}(pr^{-1}(b)))$, будемо говорити як про *відповідності-проекції*, які обчислює процедура P.

Для нескінченних обчислень теж може бути передбачено механізм для встановлення результату. Одним з них може бути, наприклад, визначення на станах певного часткового порядку і тоді за результат можна взяти найменшу верхню границю станів обчислення у разі її існування[4]. Але це тема вже для окремого обговорення.

Скажемо також про варіанти закінчення обчислень в темпоральних процедурах. Вони можуть закінчуватись (примусово) по досягненню певного заданого моменту часу, вичерпання часового ліміту на обчислення, тощо.

Процедури P та Q називаються *еквівалентними* $(P \cong Q)$, якщо відповідності P^* та Q^*, які вони обчислюють, збігаються. Стан процедури назвемо *допустимим*, якщо він зустрічається серед станів деякого її результативного обчислення. Для відповідностей, які обчислює процедура, стани за межами підмножини її допустимих станів S_{acc} не важливі. Процедура, усі стани якої допустимі, називається *приведеною.*

Багато прикладів обчислювальних процедур можна знайти вже в шкільній алгебрі та геометрії. Так, задачі пошуку розв'язку лінійних рівнянь та їх систем, розв'язок квадратних рівнянь, задачі пошуку за допомогою циркуля й лінійки середини відрізка, найбільшої спільної міри двох відрізків розв'язуються за допомогою обчислювальних процедур. Ці процедури, за винятком останньої, є темпоральними. Алгоритм Евкліда для пошуку найбільшої спільної міри двох відрізків − автоматний. Класичними прикладами автоматних процедур є також скінченні й магазинні автомати, машини Тюрінга, ігрові числення тощо. Але в останніх часові фактори можуть впливати на закінчення обчислень.

В деяких випадках в процесі обчислень доводиться враховувати не тільки поточний стан, а й попередні. Це стосується, як функції керування, так і елементарних операцій. Нехай $\Pi = (T, S, S_0)$ – довільний обчислювальний простір. Новими його конфігураціями покладемо пари

[4] Такі процедури може бути цікавими у світлі апроксимаційної платформи Д.Скотта [3], яка сама по собі є не процедурною.

декартового добутку $T \times S^+$. Розширимо функцію керування та елементарні операцій на множину скінченних послідовностей станів: $\delta : T \times S^+ \to \Delta$, $\Delta = \{ f_i : S^+ \to S, i = \overline{1,n} \}$. Трійку $P = < \Pi, \Delta, \delta >$ з такими з такими розширеними функцією керування та елементарними операціями будемо називати *розширеною процедурою* у просторі Π. Обчислення $p : T \to S$ з початковим станом s за розширеною процедурою P визначається як послідовність станів $p_0, p_1, ...$ така, що: $p_0 = s$ і для всіх $t > 0$ $p_t = f_t(p_0 p_1 ... p_{t-1})$, де f_t – одне з значень функції керування δ на конфігурації $(t, p_0 p_1 ... p_{t-1})$.

Зазначимо, що функцію керування δ тепер можна визначати як унарну тільки на послідовностях станів, тому що часовий компонент конфігурації обчислень неявно представлений довжиною цих послідовностей. Розширену процедура P назвемо *автоматною*, якщо значення її функції керування та всіх елементарних операцій залежать тільки від останнього стану обчислення, тобто $\delta(p_0 p_1 ... p_{t-1}) = \delta(p_{t-1})$, $f_i(p_0 p_1 ... p_{t-1}) = f_i(p_{t-1})$.

Процедури й розширені процедури є підкласом динамічних дискретних систем [4]. Якщо нівелювати вплив функції керування на обчислення (випадок, коли на всіх елементах своєї області визначення ця функція повертає кожну з елементарних операцій), то отримаємо клас розмічених транзиційних систем [5].

Далі під процедурами будемо розуміти саме розширені процедури, а звичайні процедури, в разі необхідності, будемо називати *простими*. Останні ми виділили окремо ще й тому, що, як побачимо далі, більшість існуючих моделей алгоритмів належать саме до простих процедур і навіть до простих автоматних процедур. Зазначимо також, що кожну розширену процедуру можна промоделювати простою процедурою, якщо перейти від станів до "узагальнених" станів − їхніх послідовностей і використати функцію-проекцію, яка за узагальненим станом повертає останній його елемент.

Тепер, щоб отримати математичного означення алгоритму, залишилося формально виокремити алгоритми серед всіх обчислювальних процедур.

Нагадаємо, що ознакою *ядерної еквівалентності* двох елементів з області визначення довільної функції є збіг її значень на цих елементах. Для функцій керування процедур ці елементи є конфігураціями обчислювального простору, а значеннями – елементарні операції. Індексом відношення еквівалентності називається потужність множини її фактор-класів. Ядерна еквівалентність функцій керування процедур має скінченний індекс. Множина називається *конструктивною*, якщо вона розв'язна або перелічна[5]. Функції з конструктивним графіком самі називаються *конструктивними*. Як відомо, конструктивність графіку функції є необхідною і достатньою умовою алгоритмічної обчислювальності функцій.

Для алгоритмів, на відміну від процедур, будемо вимагати *конструктивності* функції керування. Таким чином, прийшли до наступного означення алгоритму:

А*лгоритм – це обчислювальна процедура з конструктивною функцією керування*.

Враховуючи, що конструктивність функції керування рівнозначна конструктивності її ядерної еквівалентності, то маємо ще одне означення алгоритму:

А*лгоритм – це обчислювальна процедури, ядерна еквівалентність функції керування якої є конструктивною*.

Як бачимо, виокремлення алгоритмів вимагає наявності певної моделі конструктивності множин. Здавалося б маємо зачароване коло – загальне поняття потребує визначення самого себе. Насправді, тут не виникає зачароване коло, тому що мова не йде про загальну конструктивність множин, а тільки про окремий її тип чи модель. Якщо цей тип чи модель не фіксуються, то маємо абстрактне означення алгоритму, якщо ж це зробити, то отримаємо означення певного конкретного класу однотипних алгоритмів. Важливо, що фіксація вже дуже простих

[5] Множина є розв'язною (перелічною), якщо її характеристична (часткова характеристична) функція є алгоритмічно обчислювальною.

еквівалентностей на станах обчислювальних просторів дозволяє отримати широкий спектр універсальних в сенсі тези Чорча моделей алгоритмів і відкриває шлях для класифікації, упорядкування та уніфікації цих моделей.

Що до останнього, то подання алгоритмів можна задавати у вигляді скінченної таблиці з двома стовпчиками: «клас ядерної еквівалентності / значення функції». В такій таблиці можна знайти усі елементи алгоритму як процедури. Щоб відрізняти алгоритми-процедури від інших моделей алгоритмів, будемо називати їх ще *процедурними алгоритмами* або *𝒫-алгоритмами.*

Як процедури, алгоритми можуть бути детермінованими й недетермінованими. Функції (відповідності), які вони обчислюють, називаються *алгоритмічно обчислювальними.*

Наведемо основні властивості алгоритмів, які безпосередньо випливають з їхнього формального означення.

Масовість. *𝒫*-алгоритм може бути застосований до будь-якого вхідного стану обчислювального простору.

Дискретність. Обчислення за *𝒫*-алгоритмом відбуваються в дискретному часовому просторі, елементи якого разом зі станами можуть впливати на вибір перетворень поточного стану.

Елементарність. В кожний момент часу в обчисленні виконується одна елементарна операція з фіксованої сукупності таких операцій.

Визначеність. Порядок застосування елементарних операцій в обчисленні не довільний, а визначається функцією керування за поточною конфігурацією обчислення.

Результативність. У кожному *𝒫*-алгоритмі є механізм завершення обчислень.

Фінітність. Скінченність подання станів та функції керування *𝒫*-алгоритму. Це наслідок конструктивності функції керування.

Відносність. Означення алгоритму не абсолютне, а відносне. Ця відносність пов’язана як з залежністю його від моделі конструктивності

множин, так і з тим, що у деяких випадках окремі чи всі символи елементарних операцій можуть параметризуватися з наступною їх актуалізацією. Більше того, деякі з них можуть і не актуалізуватися перед застосуванням алгоритму. Тоді вони називаються *оракулами*, а відповідні алгоритми – *алгоритмами з оракулами*.

Як бачимо, $\mathcal{P}$-алгоритмам притаманні всі інтуїтивні властивості алгоритмів. Відмінність в тому, що в нашому випадку вони є наслідком формального означення алгоритму, а в інтуїтивній ТА – фіксуються апріорі. Проте з'явилися й нові специфічні риси – відносність та можлива залежність функції керування $\mathcal{P}$-алгоритму одночасно від обох компонентів обчислюваного простору – часового та стану (змішані алгоритми). Усі класичні алгоритми є автоматними і жорстко прив'язані до свого способу подання функції керування.

3. В теорії алгоритмів розрізнюють власне алгоритмічні системи і функціональні алгоритмічні системи. Функціональні певним чином (як правило, алгебраїчно) описують класи алгоритмічно обчислювальних функцій. Далі під алгоритмічними буде розуміти саме перші, тобто процедурні системи.

Щоб отримати конкретний клас алгоритмів необхідно вибрати його обчислювальний простір, сукупність базових операцій на станах, визначити вхідні стани, а за необхідності і вихідні та функцію-проекцію, але головне – вибрати тип ядерної еквівалентності функції керування. З вибором останнього і пов'язана, в першу чергу, специфіка кожного з існуючих класів алгоритмічних систем – машин Тюринга, алгоритмів Маркова, регістрових машин, тощо.

Визначимо місце цих та інших традиційних алгоритмічних систем в системі процедурних алгоритмів. Якщо звернутися до цих систем, то можна помітити таку закономірність: 1) вони, як правило, автоматні, тобто переходи в обчисленнях залежать тільки від стану обчислення і ніяким чином не залежать від кроку (моменту часу) обчислення і 2) для цих переходів визначальним є не весь поточний стан, а тільки вміст певного

обмеженого його структурного фрагменту. Збіг цих вмістів лежить в основі ядерної еквівалентності відповідних функції керування.

1) *Машини Тюрінга (МТ).* Станами $\mathcal{P}$-алгоритму для машини Тюрінга виступають пари (внутрішній стан МТ, слово на стрічці з виділеною доступною коміркою). В початкових станах внутрішній стан МТ є фіксованим початковим, в заключних – заключним. Функція-проекція *pr* повертає вміст стрічки стану. Два стани $\mathcal{P}$-алгоритму еквівалентні, якщо внутрішні стани МТ та вмісти доступних комірок на стрічках збігаються. Функція керування не залежить від часового компонента, тому $\mathcal{P}$-алгоритм – автоматний.

2) *Регістрові машини (РМ)*. Стани $\mathcal{P}$-алгоритму для регістрової машини складає сукупність всіх її регістрів, серед яких є лічильник команд *PC*. Останній містить адресу команди, яка буде виконуватись на наступному кроці обчислення РМ. В початковому стані $PC=1$, в заключних станах $PC>n$, де n – адреса останньої машинної команди в програмі РМ. Функція-проекція *pr* повертає вміст нульового регістру стану. Стани $\mathcal{P}$-алгоритму еквівалентні, якщо в них значення лічильника команд *PC* збігаються. Сам $\mathcal{P}$-алгоритм, як і у випадку МТ – автоматний.

3) *Системи Поста (СП).* Станами $\mathcal{P}$-алгоритму для системи Поста виступають усі слова об'єднаного алфавіту, вихідними станами – слова термінального алфавіту. Вхідні стани задають аксіоми. В $\mathcal{P}$-алгоритмі використовуються гіперобчислення. Для кожного правила виведення розглянемо множину всіх станів, до яких воно застосовне і накладемо усі ці множини одночасно одна на одну. Отримане розбиття скінченного індексу множини всіх станів індукує потрібну еквівалентність. $\mathcal{P}$-алгоритм, як і в попередніх прикладах – автоматний.

4) *Алгоритми Маркова (АМ).* Стани і вихідні стани $\mathcal{P}$-алгоритму для алгоритму Маркова – ті ж, що і в системах Поста. Тільки в заключних станах може бути одне входження символу "." Крапка не є термінальним символом – вона є обов'язковим префіксом правої частини заключних правил виведення. На словах з крапкою функція переходу табличного алгоритму не визначена, тому на них $\mathcal{P}$-алгоритм припиняє роботу. Вхідними є всі слова в термінальному алфавіті. Функція-проекція *pr* вилучає крапку зі слова і залишає слово без змін, якщо крапки в ньому немає. Як і у випадку систем Поста, для кожного правила виведення розглянемо множину всіх слів, до яких воно застосовне в АМ. На відміну від систем Поста, ці множини попарно не перетинаються і складають розбиття множини станів, яке і індукує потрібну еквівалентність. $\mathcal{P}$-алгоритм – автоматний.

5) λ – *числення.* Станами відповідного *T*-алгоритму виступають усі λ – вирази. Початковими станами є вирази-застосування. Правило редукції є єдиною частковою операцією алгоритму. Два стани еквівалентні, якщо до них застосовне правило редукції. Функція керування не визначена на станах, до яких правило редукції не застосовне. $\mathcal{P}$-алгоритм – автоматний.

6) *Арифметика Пеано.* Розглядається формальна арифметика з аксіоматикою Пеано. Станами відповідного розширеного $\mathcal{P}$-алгоритму виступають арифметичні формули. Початковими станами – аксіоми формальної арифметики, вихідними – теореми. В алгоритмі використовуються гіперобчислення. Два стани еквівалентні, якщо до них застосовне, принаймні, одне з правил виведення. Розширений $\mathcal{P}$-алгоритм – не автоматний.

7) *Граф-схеми Калужніна (ГС).* Занумеруємо всі вузли ГС числами від 0 до *n,* де 0 – номер початкового вузла. Станами $\mathcal{P}$-алгоритму для граф-схеми Калужніна виступають пари (номер вершини ГС, елемент предметної області). У початковому стані перша компонента 0, у

заключному – номер заключного вузла ГС. Проективна функція повертає елемент предметної області стану. Два стани алгоритму еквівалентні, якщо їхні перші компоненти збігаються і а) ця компонента є номером вузла-перетворювача або б) ця компонента є вузлом-предикатом і значення цього предикату на елементах предметної області станів збігаються. Розширений $\mathcal{P}$-алгоритм – автоматний.

8) *Дискретні перетворювачі Глушкова (ДП).* Станами $\mathcal{P}$-алгоритму для дискретного перетворювача виступають трійки (стан операційного автомата, стан керуючого автомата, вхідний сигнал). В початкових станах *T*-алгоритму внутрішній стан керуючого автомату – початковий, стан операційного автомату – довільний, початковий вхідний сигнал функціонально залежить від початкового стану операційного автомата. В заключних станах – стан керуючого автомата – заключний. Функція-проекція *pr* повертає стан операційного автомата. Два стани $\mathcal{P}$-алгоритму еквівалентні, якщо стани керуючого автомата в них та вхідні сигнали збігаються. Розширений $\mathcal{P}$-алгоритм – автоматний.

Всі еквівалентності в 1)-8) мають скінченний індекс і збігаються з ядерною еквівалентністю функцій керування відповідних $\mathcal{P}$-алгоритмів. Вони є розв'язними в традиційному сенсі для 1)-6). Для 7) їхня конструктивність залежить від алгоритмічної обчислювальності предикатів ГС. Для 8) – від алгоритмічності функції формування вхідного сигналу.

Не складно впевнитися, що для кожної з наведених вище алгоритмічних систем обчислення за нею та її $\mathcal{P}$-алгоритмом на однакових (з урахуванням проекції) початкових станах будуть розгортатися синхронно з застосуванням на кожному кроці одних і тих же операцій і закінчуватися з однаковими (з урахування проекції) результатами. Це означає, що ці алгоритмічні системи та відповідні $\mathcal{P}$-алгоритми є еквівалентними.

Як бачимо, процедурні алгоритми дозволять розглядати традиційні

моделі алгоритмів з єдиних позицій з урахуванням їхнього місця в системі цих алгоритмів.

Загальні властивості процедурних алгоритмів мають свою інтерпретацію в усіх конкретних алгоритмічних системах, зокрема в розглянутих пунктах 1)-8). Деякі теоретичні результати, що стосуються властивостей обчислювальних процедур та алгоритмів, можна знайти в [6].

Висновки. Розглянута процедурна платформа для теорії алгоритмів. Запропоновано поняття процедурного алгоритму у якості загального формального означення алгоритму. Показано, що запропонована концепція алгоритму може бути теоретичною і методичною основою для вивчення традиційних моделей алгоритмів та алгоритмічних обчислень і бути корисною в курсах з теорії алгоритмів та програмування.

Список використаних джерел

1. Глібовець М.М. Моделі обчислень у програмній інженерії. /О.В.Кирієнко, В.С.Проценко. – К.: Видавничий дім «Києво-Могилянська академія», 2019. – 209 с.
2. Зубенко В. В. Програмування: Навчальний посібник / Л.Л. Омельчук. – К.: ВПЦ «Київський університет», 2011. – 625 с.
3. Скотт Д. Теория решеток, типы данных и семантика. – В кн.: Данные в языках программирования. – М.: Мир, 1982. – С. 25-53.
4. Капитонова Ю.В. Математическая теория проектирования вычислительніх систем. / А.А. Летичевский. – М.: Наука, 1988. – 295 с.
5. Arnold A. Finite Transition System. – Paris: Prentice Hall. – pp. 178 .
6. Зубенко В.В. Темпоральні процедури та алгоритми. // Проблеми програмування, 2006. – № 2-3. – С. 53-59.

Іванов Євген В'ячеславович, к.ф.-м.н., доцент

ПРО ПРИНЦИПИ ІНДУКЦІЇ ЗІ СЛАБКИМ БАЗИСОМ
ON INDUCTION PRINCIPLES WITH WEAK BASIS

У роботі пропонуються модифікації принципів відкритої та лінійно впорядкованої індукції, що можуть бути корисними для доведення властивостей безпечності для дискретно-неперервних (гібридних) систем у системах інтерактивного доведення теорем, таких, як Isabelle, Coq та ін.

Ключові слова: математична індукція, частково впорядкована множина, принцип відкритої індукції.

We propose modifications of the open induction principle and the principle of linearly ordered induction which may be useful for proving safety properties of discrete-continuous (hybrid) systems in interactive proof assistants such as Isabelle, Coq, etc.

*Key words**:** mathematical induction, ordered set, open induction.*

We consider the question of applicability of induction proof principles to the problem of proving a property of elements of a poset with subsets which are not bounded above and below and propose a modification of Raoult's open induction principle [1] suitable for a class of such cases which may be useful for computer science applications such as formalization of discrete-continuous (hybrid) systems in computer proof assistants (e.g. Isabelle, Coq, etc.) and semantics of specification languages for such systems.

The soundness of the ordinary weak induction principle (which is useful for proving that a given predicate P is true on all natural numbers) depends (in particular) on the fact that the set of natural numbers has a least element, and the basis of the induction explicitly refers to this element. Other examples of induction principles which are useful for proving that a predicate is true on all elements of a partially ordered set (poset) for different classes of posets include well-founded induction, Noetherian induction, open induction [1], real induction [2], linearly ordered induction [2]. In some of them the basis of the induction is not formulated explicitly, but can be separated from their conditions. For example,

in the premise of Noetherian induction principle for Noetherian posets $\forall y(\forall x(y < x \Rightarrow P(x)) \Rightarrow P(y))$ one can call the condition that $P(x)$ is true for each maximal element x the induction basis (informally, for such x the induction hypothesis is not useful for showing that $P(x)$ holds). When one needs to adapt such principles to posets with subsets which are not bounded above and below, an issue of formulation of an induction basis arises. For example, the principle of linearly ordered induction proposed in [2] for Dedekind complete total orders (which may have no minimal or maximal elements) deals with this issue, basically, by requiring that the truth domain of a predicate includes $\{x \mid x \leq a\}$ for some element a. In the talk we describe an alternative approach based on replacing such a condition with a weaker condition of coinitiality or cofinality.

A poset has the least-upper-bound property for chains, if every nonempty upper bounded chain has a supremum in $(X, \leq)$. We discuss adaptations of the open induction principle to posets with the least-upper-bound property and their first-order reformulations in the signature of posets extended with a predicate symbol for the predicate P for special classes of posets which are of interest for formal methods and discuss their potential applications.

Список використаних джерел / References

1. Raoult J.-C. Proving open properties by induction / J.-C. Raoult // Information Processing Letters 29. – 1988. – P. 19-23.
2. Clark P. The instructor's guide to real induction / P. Clark // Mathematics Magazine 92. – 2019. – P. 136-150.

Кохан Ярослав Олексійович, к. філос. н.

СИСТЕМА АЛЬТЕРНАТИВ ЯК ТЕОРЕТИЧНИЙ ОБ'ЄКТ ЛОГІКИ

Системою альтернатив на заданій предметній області D ми називаємо будь-яку множину n-місних предикатів, один і тільки один з яких здійснюється на будь-якій n-ці предметів з D. У роботі будуються начала теорії систем альтернатив. Порівнюються пари множин всіх систем альтернатив, у котрі входять предикати, між якими задане конкретне відношення. Встановлюється зв'язок між поняттями системи альтернатив та класифікації, досліджується відношення ортогональності між системами альтернатив, описуються багатовимірні класифікації. Обговорюються можливі застосування теорії систем альтернатив у логіці, математиці, філософії та психології.

Ключові слова: предикат, система альтернатив, класифікація, ортогональність, вибір.

We call a system of alternatives on a given domain D any set of n-ary predicates, one and only one of those satisfies for every n-tuple of individuals from D. The fundamentals of the system of alternatives theory are build in the paper. Next we compare pairs of sets consisting from systems of alternatives that include predicates between which a particular relation is established. Next the connection between the concepts of a system of alternatives and classification is established, the orthogonality relation between systems of alternatives is investigated, multidimensional classifications are introduced and described. Possible applications of the systems of alternatives theory in logic, mathematics, philosophy and psychology are discussed.

Keywords: predicate, system of alternatives, classification, orthogonality, choice.

Постановка проблеми

Мотивація. Опис тієї чи іншої частини дійсності ми здійснюємо за допомогою предикатів. При цьому, вводячи всякий новий предикат, ми неявно передбачаємо, що він відповідає деякій можливості, деякому можливому стану справ, і крім нього існує лише обмежена кількість інших можливостей, одна з яких має справдитися, якщо не справдилася описувана

даним предикатом; ці можливості описуватимуться якимись іншими предикатами. Тим самим предикати природно групуються в *системи альтернатив*, у кожній з яких для всякого набору аргументів здійснюється одна і тільки одна альтернатива-предикат. Так, для пар дійсних чисел існує три альтернативи: перше з чисел або більше за друге, або менше, або рівне йому (є тим самим числом); відтак, якщо ми введемо предикат $<^{(2)}$ *Менше* на області дійсних чисел, ми повинні потурбуватися і про введення альтернативних до нього предикатів $>^{(2)}$ *Більше* та $=^{(2)}$ *Дорівнює*. Зробити це можна двома способами: (a) явно задавши ці предикати і явно описавши відношення між ними або (b) задавши такі постулати (позалогічні аксіоми), на основі яких можна було б означити відповідні альтернативні предикати (насправді тільки один, бо другий доведеться також ввести як початковий), а відношення між ними довести як (позалогічні) теореми. Аксіоматичний спосіб дослідження (b) достатньо розроблений в логіці і складає предмет цілих розділів цієї науки, а саме, логічного синтаксису та теорії доведень. Безпосереднє ж задання систем альтернатив досі перебувало поза рамками теоретичного дослідження. Пропонована робота компенсує цю прогалину в корпусі логічних теорій.

Позначення й термінологія. За основу всякого логіко-предикатного розгляду ми беремо три множини: множину $\mathbb{B} = \{\mathbf{1}, \mathbf{0}\}$ значень істинності, де **1** - це істина, а **0** - це лож (хиба), *предметну область*, або *домен*, D, тобто, множину довільних об'єктів, які ми називатимемо (*логічними*) *предметами*, та непорожню скінченну множину Ω заданих на D предикатів (ми обмежуємо теорію скінченним випадком, виходячи з практичних міркувань; див. останній розділ, присвячений застосуванням).

Окремі предмети ми позначаємо *вільними предметними змінними* 'a_i', де $i \in \mathbb{N}$, і 'a_0' скорочується до 'a'. На всі або деякі предмети ми *посилаємося* (при квантифікації має місце саме узагальнене посилання, а не позначення) за допомогою *зв'язаних предметних змінних* 'x_i', де $i \in \mathbb{N}$, і 'x_0' скорочується до 'x'.

В логіці існує два різних розуміння терміна «предикат» (а також деякі інші розуміння, які не набули поширення, тому ми їх не обговорюємо). Під предикатом, або, точніше, *n-місним* (або *n-арним*) *предикатом*, (*заданим*) *на*

D, розуміють або n-місне відношення, задане на D, або однозначну функцію типу $f\colon \mathrm{D} \mapsto \mathbb{B}$. Позначатимемо предикати-відношення *вільними предикатними змінними* ‘$P_i^{(n)}$’, ‘$R_i^{(n)}$’, ‘$S_i^{(n)}$’, де $i \in \mathbb{N}$, і ‘$P_0^{(n)}$’, ‘$R_0^{(n)}$’, ‘$S_0^{(n)}$’ скорочуються до ‘$P^{(n)}$’, ‘$R^{(n)}$’, ‘$S^{(n)}$’ відповідно, а предикати-функції - *вільними предикатними змінними* ‘$F_i^{(n)}$’, ‘$G_i^{(n)}$’, ‘$H_i^{(n)}$’, де $i \in \mathbb{N}$, і ‘$F_0^{(n)}$’, ‘$G_0^{(n)}$’, ‘$H_0^{(n)}$’ скорочуються до ‘$F^{(n)}$’, ‘$G^{(n)}$’, ‘$H^{(n)}$’ відповідно; квантифікацію по предикатах ми не використовуватимемо, тому *зв’язані* предикатні змінні нам не знадобляться. Той факт або умова, що деякі $a_1, ..., a_n \in \mathrm{D}$ перебувають у відношенні $P_i^{(n)}$, ми традиційно записуємо у вигляді формули ‘$P_i(a_1, ..., a_n)$’, а факт або умова, що функція $F_j^{(n)}$ набуває на тих самих аргументах $a_1, ..., a_n$ значення $\boldsymbol{i} \in \mathbb{B}$, ми так само традиційно записуємо у вигляді формули ‘$F_j(a_1, ..., a_n) = \boldsymbol{i}$’ (арності предикатів при цьому не пишуться).

Слід звернути особливу увагу на те, що використання значень істинності **1** та **0** для опису предикатів не означає, що ми опиняємося в області семантики. Навпаки: значення істинності є такими ж об’єктами онтології (позначуваної дійсності, світу денотатів), як і будь-які предмети з D та будь-які предикати з Ω, а прирівнювання значення $F(a_1, ..., a_n)$ будь-якого предиката $F^{(n)}$ до значення істинності **1** або **0** належить єдиному світу денотатів так само, як це відбувається в математиці з будь-якими рівняннями ‘$f(a_1, ..., a_n) = a_0$’. Відтак, вся теорія предикатів лежить в області онтології. При цьому предикати мають властивості, аналогічні властивостям формул у семантиці (в теорії моделей). В цій роботі ми використовуємо такі властивості предикатів, як тотожна істинність, тотожна хибність, нейтральність, здійсненність та не здійсненність на D та на D^n. Ми називаємо предикат $P^{(n)}$, відповідно $F^{(n)}$, *тотожно-істинним на* D^n (відповідно, *тотожно-хибним на* D^n; *нейтральним на* D^n) я. я. (якщо і тільки якщо) на будь-якій (відповідно, на жодній; на деякій, але не кожній) n-ці $\langle a_1, ..., a_n \rangle \in \mathrm{D}^n$ має місце факт $P(a_1, ..., a_n)$, відповідно, факт $F(a_1, ..., a_n) = \mathbf{1}$. Ми називаємо предикат $P^{(n)}$, відповідно $F^{(n)}$, *здійсненним на* D^n (відповідно, *не здійсненним на* D^n) я. я. знайдеться (відповідно, не знайдеться) така n-ка $\langle a_1, ..., a_n \rangle \in \mathrm{D}^n$, на якій має місце факт $P(a_1, ..., a_n)$, відповідно, факт

$F(a_1, ..., a_n) = \mathbf{1}$. Зрозуміло, що якщо підставити в ці означення ‘D’ замість ‘D^n’ і «$a_1, ..., a_n \in D$» замість «$<a_1, ..., a_n> \in D^n$», отримаємо коректні означення понять тотожної істинності, тотожної хибності, нейтральності, здійсненності та не здійсненності на D, еквівалентні тільки-но введеним поняттям для D^n. Тому надалі ми користуємося формулюваннями обох наведених типів.

Як відомо, обидва трактування поняття предиката - як відношення та як функції - *еквівалентні* в тому сенсі, що на будь-якій даній D, по-перше, для всіх $a_1, ..., a_n \in D$ і всякого предиката $P^{(n)}$ знайдеться такий предикат $F^{(n)}$, що мають місце еквіваленції

$$P(a_1, ..., a_n) \leftrightarrow F(a_1, ..., a_n) = \mathbf{1} \tag{1}$$

і

$$\neg P(a_1, ..., a_n) \leftrightarrow F(a_1, ..., a_n) = \mathbf{0}, \tag{2}$$

і, по-друге, для всякого предиката $F^{(n)}$ знайдеться такий предикат $P^{(n)}$, що мають місце еквіваленції (1) і (2). В силу такої еквівалентності ми надалі описуватимемо заради простоти лише предикати-відношення; все сказане про них легко перенести і на предикати-функції. Для обох понять предиката ми сформулюємо лише означення основного поняття даної роботи - поняття системи альтернатив.

Екстенсіональний та інтенсіональний підходи. Відношення між предикатами та їхніми умовами істинності визначаються тим очевидним фактом, що тотожні предикати суть еквівалентні, що символізується формулою

$$P^{(n)} = R^{(n)} \rightarrow \forall x_1 ... \forall x_n \, (P(x_1, ..., x_n) \leftrightarrow R(x_1, ..., x_n)). \tag{3}$$

В сучасній логіці практично завжди приймають *екстенсіональну*, або *об’ємніcну*, трактовку поняття предиката. Це означає, що предикати трактуються винятково як об’єми, множини, а отже, еквівалентні предикати ототожнюються. Це можна виразити за допомогою формули

$$\forall x_1 ... \forall x_n \, (P(x_1, ..., x_n) \leftrightarrow R(x_1, ..., x_n)) \rightarrow P^{(n)} = R^{(n)}, \tag{4}$$

конверсної до формули (3). Екстенсіональний підхід до предикатів достатній для застосувань у математиці, але є хибним в загальному випадку, що можна показати навіть на математичних прикладах. Скажімо, відношення *Бути перетином всіх бісектрис трикутника ...* та *Бути перетином всіх медіан трикутника ...* очевидно не тотожні, однак здійснюються на одних і тих самих парах <точка, трикутник>, яку б D ми не вибрали. Тим більше, навіть для не еквівалентних предикатів завжди можна так вибрати предметну область D, що на ній вони стануть еквівалентними. Відтак, в загальному випадку слід відрізняти всякий предикат від його графіка - так само, як це має місце для функцій загального виду. Такий не екстенсіональний підхід до предикатів і таке їхнє трактування називатимемо *інтенсіональним*, або *змістовним*. За нього формула (4) відкидається як хибна. Надалі ми приймаємо саме інтенсіональний підхід; за нього, приміром, ми не ототожнюємо предикати *Тварина, що має стать* та *Той, хто є самцем або самицею* на всякій D, яка містить тварин лише належних до видів зі статевим розмноженням, і не містить гермафродитів.

Тим не менше, випадки, коли екстенсіональний підхід застосовний, мають самостійний інтерес. Тому нижче ми відзначатимемо їх окремо.

Означення поняття системи альтернатив. Будь-яка множина $\mathbb{A}$ n-місних предикатів, означених на даній D, тобто, будь-яка підмножина множини Ω називається *системою альтернатив на* D, я. я. здійснюються наступні дві умови:

$$\text{(i)}\ \forall x_1 \ldots \forall x_n \left(\bigvee_{P(n)\in \mathbb{A}} P(x_1, \ldots, x_n)\right)$$

(словами: *диз'юнкція всіх предикатів з* $\mathbb{A}$ *тотожно-істинна на* D);

$$\text{(ii)}\ (P^{(n)} \neq R^{(n)})_{P(n),\, R(n)\in \mathbb{A}} \rightarrow \forall x_1 \ldots \forall x_n \neg (P(x_1, \ldots, x_n) \wedge R(x_1, \ldots, x_n))$$

(словами: *всі предикати з* $\mathbb{A}$ *попарно несумісні на* D).

Якщо задати деякий перерахунок елементів множини $\mathbb{A}$ індексами i з якоїсь довільної множини I, ці самі умови для випадку предикатів-функцій можна сформулювати наступним чином:

(i) $\forall x_1 \ldots \forall x_n \, (\bigvee_{i \in I} F_i(x_1, \ldots, x_n) = \mathbf{1})$;

(ii) $\forall x_1 \ldots \forall x_n \, (\bigwedge_{i \in \{j, k\} \subseteq I, j \neq k} F_i(x_1, \ldots, x_n) = \mathbf{0})$.

Предикати-елементи всякої системи альтернатив $\mathbb{A}$ називаються *альтернативами з* (або *належними*) $\mathbb{A}$, а також *альтернативами* один відносно одного. Кількість альтернатив з $\mathbb{A}$ - це потужність $\|\mathbb{A}\|$ множини $\mathbb{A}$.

Елементи теорії систем альтернатив

Основні властивості систем альтернатив. Найперша властивість систем альтернатив виражається наступною умовою: які б не були система альтернатив $\mathbb{A}$ та предикат $P^{(n)} \in \mathbb{A}$, має місце еквіваленція

$$\forall x_1 \ldots \forall x_n \, (\neg P(x_1, \ldots, x_n) \leftrightarrow \bigvee_{R(n) \in \mathbb{A} \setminus \{P(n)\}} R(x_1, \ldots, x_n)). \qquad (5)$$

Доведення. Всякий предикат $P^{(n)} \in \Omega$ розбиває область D^n на дві підобласті: *область істинності* $\mathrm{dom}_1 P^{(n)}$ предиката $P^{(n)}$ та *область* $\mathrm{dom}_0 P^{(n)}$ його *хибності*:

$$\mathrm{D}^n = \mathrm{dom}_1 P^{(n)} \cup \mathrm{dom}_0 P^{(n)},$$
$$\varnothing = \mathrm{dom}_1 P^{(n)} \cap \mathrm{dom}_0 P^{(n)}.$$

З пункту (i) означення поняття системи альтернатив та очевидного факту

$$\forall x_1 \ldots \forall x_n \, (\bigvee_{R(n) \in \mathbb{A}} R(x_1, \ldots, x_n) \leftrightarrow (P(x_1, \ldots, x_n) \vee \bigvee_{R(n) \in \mathbb{A} \setminus \{P(n)\}} R(x_1, \ldots, x_n)))$$

безпосередньо випливає, що предикат, заданий квазіформулою ‘$\bigvee_{R(n) \in \mathbb{A} \setminus \{P(n)\}} R(x_1, \ldots, x_n)$’, є тотожно-істинним на $\mathrm{dom}_0 P^{(n)}$; з пункту ж (ii) означення поняття системи альтернатив випливає, що всі $R^{(n)}$ з $\mathbb{A} \setminus \{P^{(n)}\}$ тотожно-хибні на $\mathrm{dom}_1 P^{(n)}$, а отже, тотожно-хибна на $\mathrm{dom}_1 P^{(n)}$ і їхня диз’юнкція. Відтак,

$$\mathrm{dom}_1 P^{(n)} = \mathrm{dom}_0 \bigvee_{R(n) \in \mathbb{A} \setminus \{P(n)\}} R^{(n)}$$

і

$$\mathrm{dom}_0 P^{(n)} = \mathrm{dom}_1 \bigvee_{R(n) \in \mathbb{A} \setminus \{P(n)\}} R^{(n)},$$

що й доводить формулу (5). □

Який вигляд матиме найменша за кількістю елементів система альтернатив? З означення поняття системи альтернатив випливає, що це буде множина, яка складається з одного єдиного предиката. А точніше, маємо

ТВЕРДЖЕННЯ 1. *Всяка підмножина множини* Ω*, яка складається з одного єдиного тотожно-істинного на* D *предиката, є системою альтернатив на* D*.*

Доведення. Справді, розглянемо множину $\{P^{(n)}\}$, яка складається з одного єдиного предиката $P^{(n)}$, тотожно-істинного на D. Допущенно говорити про диз'юнкцію з одної єдиної конституенти. Диз'юнкція, єдиною конституентою якої є тотожно-істинний на D предикат, сама буде тотожно-істинною на D; відтак, множина $\{P^{(n)}\}$ задовольняє пункту (i) означення поняття системи альтернатив. З другого боку, не існує жодного предиката $R^{(n)}$ такого, що

$$R^{(n)} \in \{P^{(n)}\} \wedge R^{(n)} \neq P^{(n)} \wedge \exists x_1 \ldots \exists x_n \, (R(x_1, \ldots, x_n) \wedge P(x_1, \ldots, x_n)),$$

а отже, для всіх $R^{(n)} \in \{P^{(n)}\}$ вірно, що

$$R^{(n)} \neq P^{(n)} \rightarrow \forall x_1 \ldots \forall x_n \neg \, (R(x_1, \ldots, x_n) \wedge P(x_1, \ldots, x_n));$$

а це якраз означає, що множина $\{P^{(n)}\}$ задовольняє пункту (ii) означення поняття системи альтернатив. □

Назвемо всяку одноелементну систему альтернатив *виродженою*. З твердження 1 безпосередньо випливає наступне

ТВЕРДЖЕННЯ 2. *На всякій непорожній D існує вироджена система альтернатив.*

Точніше, на всякій даній D існує стільки вироджених систем альтернатив, скільки є тотожно-істинних на D елементів множини Ω.

Зрозуміло, що такі системи альтернатив теоретично не цікаві - інтерес представляють лише системи, в яких є різні елементи, тобто, потужність яких не менша за 2.

Нехай дано множини A і B такі, що $A \subset B$. Казатимемо, що A є *звуженням* B, а B - *розширенням* A, і що A *розширюється до* B, а B - *звужується до* A. Якщо при цьому $B = A \cup \{a_1, ..., a_n\}$, казатимемо, що B є *поповненням* множини A *за допомогою* об'єктів $a_1, ..., a_n$. Маємо

ТВЕРДЖЕННЯ 3. *Поповнення довільної системи альтернатив на* D *за допомогою предиката* $P^{(n)}$ *самé є системою альтернатив я. я.* $P^{(n)}$ *не здійсненний на* D.

Доведення. Нехай дано систему альтернатив $\mathbb{A}$ на D, що складається з n-місних предикатів, і предикат $P^{(n)}$ такий, що $P^{(n)} \in \Omega \setminus \mathbb{A}$.

Достатність. Припустимо, що $P(a_1, ..., a_n)$, тобто, $P^{(n)}$ здійснюється на $<a_1, ..., a_n>$. Згідно з пунктом (i) означення поняття системи альтернатив, хоча б один елемент $\mathbb{A}$ здійснюється на $<a_1, ..., a_n>$. Позначимо цей елемент через '$R^{(n)}$'. В цьому разі, маємо $P(a_1, ..., a_n) \wedge R(a_1, ..., a_n)$, що несумісно з пунктом (ii) означення поняття системи альтернатив. Отже, множина $\mathbb{A} \cup \{P^{(n)}\}$ не є системою альтернатив.

Необхідність. Припустимо, що $\forall x_1 ... \forall x_n \neg P(x_1, ..., x_n)$, тобто, $P^{(n)}$ не здійсненний на D. В такому разі, $\mathbb{A} \cup \{P^{(n)}\}$ є системою альтернатив, оскільки істинні наступні дві умови:

$$\forall x_1 ... \forall x_n \, (\bigvee_{R(n) \in \mathbb{A}} R(x_1, ..., x_n) \vee P(x_1, ..., x_n)),$$

$$R^{(n)} \in \mathbb{A} \rightarrow \forall x_1 ... \forall x_n \neg (R(x_1, ..., x_n) \wedge P(x_1, ..., x_n)),$$

які відповідають пунктам (i) та (ii) означення поняття системи альтернатив. □

Таким чином, всяке поповнення будь-якої системи альтернатив тотожно-істинними і/або нейтральними на D предикатами самé не є системою альтернатив.

Назвемо систему альтернатив на D *нетривіальною*, я. я. всі її елементи здійсненні на D. Нетривіальні системи альтернатив є єдино важливими з *практичного* погляду. Доведемо ряд тверджень, які стосуються властивості нетривіальності. Перш за все, зафіксуємо очевидне

ТВЕРДЖЕННЯ 4. *Будь-яка система альтернатив на даній* D:

a) *не може містити більш ніж один тотожно-істинний на* D *предикат*;

b) *містить або тотожно-істинні, або нейтральні на* D *предикати, але не ті й другі разом*;

c) *не може містити точно один нейтральний на* D *предикат*.

Всі пункти даного твердження безпосередньо випливають з умови (ii) означення поняття системи альтернатив. Далі, з твердження 2 та означення поняття нетривіальності безпосередньо випливає

ТВЕРДЖЕННЯ 5. *Якщо* A *і* B *суть системи альтернатив, і* $A \subset B$, *то* B *не є нетривіальною*.

Також з твердження 2 випливає, що можна утворювати нові системи альтернатив не тільки додаванням до деякої заданої $\mathbb{A}$ не здійсненних предикатів, але й їх вилученням з $\mathbb{A}$. Такий процес поетапного вилучення предикатів з $\mathbb{A}$ обов'язково буде скінченним. А саме, маємо

ТВЕРДЖЕННЯ 6. *Для всякої системи альтернатив* $\mathbb{A}$ *на* D, *яка не є нетривіальною, існує така система альтернатив* $\mathbb{A}_0$ *на* D, *що* $\mathbb{A}_0 \subset \mathbb{A}$ *і* $\mathbb{A}_0$ *нетривіальна*.

Доведення. З пункту (i) означення поняття системи альтернатив випливає, що всяка система альтернатив містить здійсненні на D предикати. За умовою, $\mathbb{A}$ не є нетривіальною, отже, містить також і не здійсненні на D предикати. Відтак, можна розбити $\mathbb{A}$ на дві не перетинні підмножини $\mathbb{A}_{(1)}$ здійсненних та $\mathbb{A}_{(0)}$ не здійсненних на D предикатів. Залишається показати, що $\mathbb{A}_{(1)}$ є системою альтернатив. В силу твердження 4, $\mathbb{A}_{(1)}$ складається або з одного єдиного тотожно-істинного на D предиката, або більш ніж з одного нейтрального на D предиката і не містить тотожно-істинних на D предикатів. У першому з цих випадків, в силу твердження 1, $\mathbb{A}_{(1)}$ є виродженою системою альтернатив. Розгляньмо другий випадок. Для будь-якої множини M предикатів з Ω вірно, що

$$\forall x_1 \ldots \forall x_n \left(\left(\bigvee_{P(n) \in M} P(x_1, \ldots, x_n) \vee \bigvee_{R(n) \in \mathbb{A}_{(0)}} R(x_1, \ldots, x_n) \right) \right.$$

$$\leftrightarrow \bigvee_{P(n) \in \mathrm{M}} P(x_1, ..., x_n)),$$

відтак, це вірно і для M = $\mathbb{A}_{(1)}$. Оскільки ж диз'юнкція всіх предикатів з $\mathbb{A}$ в силу пункту (i) означення поняття системи альтернатив тотожно-істинна на D, це разом означає, що й диз'юнкція всіх предикатів з $\mathbb{A}_{(1)}$ також тотожно-істинна на D, а отже, $\mathbb{A}_{(1)}$ задовольняє умові (i) означення поняття системи альтернатив. Також $\mathbb{A}_{(1)}$ задовольняє умові (ii) означення поняття системи альтернатив, оскільки ця умова стосується будь-яких пар елементів $\mathbb{A}$, а отже й будь-яких пар елементів $\mathbb{A}_{(1)}$ в силу того, що $\mathbb{A}_{(1)} \subseteq \mathbb{A}$. Остаточно: $\mathbb{A}_{(1)}$ є системою альтернатив і містить тільки здійсненні на D предикати, відтак, вона і є шуканою $\mathbb{A}_0$. □

Назвемо *мінімальною* всяку таку систему альтернатив на D, будь-яка власна підмножина якої не є системою альтернатив. Вироджені системи альтернатив очевидно є мінімальними, отже, мінімальні системи альтернатив існують на будь-якій непорожній D за умови, що відповідна Ω містить бодай один тотожно-істинний на даному D предикат. Більше того, маємо

ТВЕРДЖЕННЯ 7. a) *Всяка система альтернатив на* D *або мінімальна, або може бути звужена до мінімальної на* D *системи альтернатив;* b) *всяка мінімальна система альтернатив або вироджена, або нетривіальна.*

Доведення. Розглянемо не мінімальну $\mathbb{A}$ на D. В силу твердження 4, вона або містить один-єдиний тотожно-істинний на D предикат $P^{(n)}$, або не містить. В першому випадку множина $\{P^{(n)}\}$ буде шуканою мінімальною системою альтернатив. У другому випадку, в силу твердження 4, $\mathbb{A}$ міститиме більш ніж один нейтральний предикат. Виділимо множину $\mathbb{A}_{(1)}$ всіх нейтральних предикатів з $\mathbb{A}$, як це було здійснено в доведенні твердження 6. Там же було доведено, що $\mathbb{A}_{(1)}$ є нетривіальною системою альтернатив. Якщо вилучити з неї будь-який предикат $R^{(n)}$, то диз'юнкція предикатів, що лишилися, в силу формули (5), буде тотожно-хибною на множині $\mathrm{dom}_1 R^{(n)}$ істинності предиката $R^{(n)}$, отже, не буде тотожно-істинною

на D (оскільки $\mathrm{dom}_1 R^{(n)} \subseteq \mathrm{D}$), тобто, не задовольнятиме умові (i) означення поняття системи альтернатив, тож не буде системою альтернатив. Звідси, $\mathbb{A}_{(1)}$ мінімальна. Це і доводить обидва пункти a), b) твердження. □

Порівняння множин систем альтернатив. Нехай $\{\mathbb{A}_P\}$ – множина всіх систем альтернатив на D, в які входить предикат $P^{(n)}$ з Ω. Постає питання: якщо між предикатами $P^{(n)}, R^{(n)} \in \Omega$ має місце якесь відоме відношення $*^{(2)}$, як співвідноситимуться відповідні їм множини $\{\mathbb{A}_P\}$ і $\{\mathbb{A}_R\}$?

1. *Еквівалентність*. Нехай

$$\forall x_1 \ldots \forall x_n \, (P(x_1, \ldots, x_n) \leftrightarrow R(x_1, \ldots, x_n)),$$

тобто, $P^{(n)}$ і $R^{(n)}$ еквівалентні (символічно: $P^{(n)} \equiv R^{(n)}$).

В цьому випадку заміна в будь-якій системі альтернатив $\mathbb{A}_P$ ($\mathbb{A}_P \subseteq \Omega$) предиката $P^{(n)}$ на предикат $R^{(n)}$ перетворить її на систему альтернатив з $\{\mathbb{A}_R\}$. В силу пункту (ii) означення поняття системи альтернатив, еквівалентні $P^{(n)}$ і $R^{(n)}$ не можуть входити в одні й ті самі системи альтернатив, отже, заміна предиката $P^{(n)}$ на $R^{(n)}$ всюди в $\{\mathbb{A}_P\}$ встановлює між $\{\mathbb{A}_P\}$ і $\{\mathbb{A}_R\}$ взаємно-однозначну відповідність, яку можна описати наступним чином. Нехай дано відображення $\varphi_{P,R}\colon \Omega \mapsto \Omega$ і $\psi_{P,R}\colon 2^{\Omega} \mapsto 2^{\Omega}$, задані наступним чином:

$$\varphi_{P,R}(S^{(n)}) = \begin{cases} R^{(n)}, \text{ якщо } S^{(n)} = P^{(n)}; \\ P^{(n)}, \text{ якщо } S^{(n)} = R^{(n)}; \\ S^{(n)}, \text{ якщо } S^{(n)} \neq P^{(n)} \text{ і } S^{(n)} \neq R^{(n)} \end{cases}$$

і

$$\psi_{P,R}(\mathrm{M}) = \begin{cases} \varnothing, \text{ якщо } \mathrm{M} = \varnothing; \\ \varphi_{P,R}[\mathrm{M}], \text{ якщо } \mathrm{M} \neq \varnothing; \end{cases}$$

тут $\mathrm{M} \subseteq \Omega$, а $\varphi_{P,R}[\mathrm{M}]$ - це $\varphi_{P,R}$-образ множини M. Згідно з цими означеннями, відображення $\varphi_{P,R}^{(1)}$ замінює місцями предикати $P^{(n)}$ і $R^{(n)}$, залишаючи всі інші предикати з Ω без змін, у той час як відображення $\psi_{P,R}^{(1)}$ ставить кожній непорожній множині $\mathrm{M} \subseteq \Omega$ у відповідність її образ $\varphi_{P,R}[\mathrm{M}]$ при відображенні $\varphi_{P,R}^{(1)}$. Звідси видно, що $\psi_{P,R}(\{\mathbb{A}_P\}) = \{\mathbb{A}_R\}$ і $\psi_{P,R}(\{\mathbb{A}_R\}) = \{\mathbb{A}_P\}$, що й доводить наявність взаємно-однозначної відповідності між $\{\mathbb{A}_P\}$ і

$\{\mathbb{A}_R\}$. Назвемо таку взаємно-однозначну відповідність *синонімією* і позначимо її через '$\simeq$'. Відтак остаточно маємо: множини $\{\mathbb{A}_P\}$ і $\{\mathbb{A}_R\}$ еквівалентних $P^{(n)}$ і $R^{(n)}$ синонімічні, символічно:

$$P^{(n)} \equiv R^{(n)} \to \{\mathbb{A}_P\} \simeq \{\mathbb{A}_R\}.$$

1ex. За екстенсіонального підходу еквівалентність $P^{(n)}$ і $R^{(n)}$ призводить, в силу (4), до рівності відповідних їм множин систем альтернатив:

$$P^{(n)} \equiv R^{(n)} \to \{\mathbb{A}_P\} = \{\mathbb{A}_R\}.$$

Синонімію множин $\{\mathbb{A}_P\}$ і $\{\mathbb{A}_R\}$ можна описати й більш загальним способом. Для цього введемо одне єдине тримісне відображення $\theta^{(3)}$ на довільних множинах, задане наступним чином:

$$\theta\,(\mathrm{B}, \mathrm{C}, \mathrm{D}) = \begin{cases} (\mathrm{D} \setminus \mathrm{B}) \cup \mathrm{C}, \text{ якщо } \mathrm{B} \subseteq \mathrm{D}; \\ \mathrm{D}, \text{ якщо } \mathrm{B} \not\subseteq \mathrm{D}. \end{cases}$$

Процедурно $\theta^{(3)}$ полягає в тому, що ми замінюємо в кожній множині D деяку її підмножину B на довільну множину C. У випадку пункту 1 розглянемо підстановки аргументів $\mathrm{B} = \{P^{(n)}\}$, $\mathrm{C} = \{R^{(n)}\}$, $\mathrm{D} = \mathbb{A}_P$ та $\mathrm{B} = \{R^{(n)}\}$, $\mathrm{C} = \{P^{(n)}\}$, $\mathrm{D} = \mathbb{A}_R$ для кожної $\mathbb{A}_P$ і кожної $\mathbb{A}_R$; отримаємо $\theta\,(\{P^{(n)}\}, \{R^{(n)}\}, \mathbb{A}_P) = \mathbb{A}_{R\theta}$ та $\theta\,(\{R^{(n)}\}, \{P^{(n)}\}, \mathbb{A}_R) = \mathbb{A}_{P\theta}$ для деяких $\mathbb{A}_{R\theta}$ і $\mathbb{A}_{P\theta}$ відповідно. Оскільки всі $\mathbb{A}_P$ і $\mathbb{A}_R$ (включно з усіма $\mathbb{A}_{R\theta}$ і $\mathbb{A}_{P\theta}$) складаються з одних і тих самих елементів, за винятком хіба елементів $P^{(n)}$ та $R^{(n)}$, це й доводить наявність взаємно-однозначної відповідності між $\{\mathbb{A}_P\}$ і $\{\mathbb{A}_R\}$, тобто, синонімічність цих двох множин. У випадку пункту 1ex в силу (4) додатково маємо $P^{(n)} = R^{(n)}$, а тому й $\{\mathbb{A}_P\} = \{\mathbb{A}_R\}$.

Введемо для зручності додаткову термінологію. Казатимемо, що предикат $P^{(n)}$ (відповідно, система альтернатив $\mathbb{A}$ така, що $P^{(n)} \in \mathbb{A}$) *порушує* (*не порушує*) *межі* деякого іншого предиката $R^{(n)}$, я. я. здійснюються обидві наступні умови (не здійснюється хоча б одна з двох наступних умов):

$$\exists x_1 \ldots \exists x_n\,(P(x_1, \ldots, x_n) \wedge R(x_1, \ldots, x_n)),$$

$$\exists x_1 \ldots \exists x_n\,(P(x_1, \ldots, x_n) \wedge \neg R(x_1, \ldots, x_n)).$$

2. *Підпорядкування/Зумовлення*. Нехай

$$\forall x_1 \ldots \forall x_n \, (P(x_1, \ldots, x_n) \rightarrow R(x_1, \ldots, x_n)),$$

тобто, $P^{(n)}$ *підпорядковується* $R^{(n)}$ або, що те саме, *зумовлює* (імплікує) $R^{(n)}$ (символічно: $P^{(n)} \vDash R^{(n)}$).

В цьому випадку, якщо множина Ω достатньо багата, знайдуться предикати $S_1^{(n)}, \ldots, S_k^{(n)} \in \mathbb{A}_P$ такі, що $(P^{(n)} \vee S_1^{(n)} \vee \ldots \vee S_k^{(n)}) \equiv R^{(n)}$, тобто,

$$\forall x_1 \ldots \forall x_n \, (((P(x_1, \ldots, x_n) \vee S_1(x_1, \ldots, x_n) \vee \ldots \vee S_k(x_1, \ldots, x_n)) \leftrightarrow R(x_1, \ldots, x_n)).$$

При цьому можуть трапитися два випадки: коли предикат $S_1^{(n)} \vee \ldots \vee S_k^{(n)}$ здійсненний та, відповідно, не здійсненний на D. У кожному з них однаково всяка $\mathbb{A}_{P, S1, \ldots, Sk} \in \{\mathbb{A}_P\}$ однозначно визначає деяку $\mathbb{A}_{R\theta} = \theta(\{P^{(n)}, S_1^{(n)}, \ldots, S_k^{(n)}\}, \{R^{(n)}\}, \mathbb{A}_{P, S1, \ldots, Sk}) \in \{\mathbb{A}_R\}$. Якщо при цьому $k > 1$, то й наборів $S_1^{(n)}, \ldots, S_k^{(n)}$ може виявитися більше одного, відтак, описана відповідність в загальному випадку багато-однозначна (не взаємно-однозначна). Оскільки ж можуть існувати такі $\mathbb{A}_P$, які порушують межі предиката $R^{(n)}$, не для кожних $P^{(n)}$ і $R^{(n)}$ всяка $\mathbb{A}_P$ є такою $\mathbb{A}_{P, S1, \ldots, Sk}$, як описано вище, відтак, описана в цьому абзаці відповідність може бути частковою. Казатимемо, що довільна множина M_1 *частково накладається на* довільну множину M_2, я. я. деяка підмножина M_0 множини M_1 може бути відображена на M_2, і позначатимемо це наступним чином: '$M_1 \subsetneq M_2$'. Звідси остаточно маємо: якщо $P^{(n)}$ зумовлює $R^{(n)}$, а в Ω існують такі предикати $S_1^{(n)}, \ldots, S_k^{(n)}$, що $(P^{(n)} \vee S_1^{(n)} \vee \ldots \vee S_k^{(n)}) \equiv R^{(n)}$, то $\{\mathbb{A}_P\}$ частково накладається на $\{\mathbb{A}_R\}$, символічно:

$$P^{(n)} \vDash R^{(n)} \wedge \forall x_1 \ldots \forall x_n \, (((P(x_1, \ldots, x_n) \vee S_1(x_1, \ldots, x_n) \vee \ldots \vee S_k(x_1, \ldots, x_n)) \leftrightarrow R(x_1, \ldots, x_n)) \rightarrow \{\mathbb{A}_P\} \subsetneq \{\mathbb{A}_R\}.$$

3. *Суперечливість/Контрадикторність/Альтернативність.* Нехай

$$\forall x_1 \ldots \forall x_n \, (P(x_1, \ldots, x_n) \leftrightarrow \neg R(x_1, \ldots, x_n)),$$

тобто, $P^{(n)}$ і $R^{(n)}$ взаємно суперечливі (контрадикторні) або, що те саме, *альтернативні* (символічно: $P^{(n)} \bowtie R^{(n)}$).

В цьому випадку множина $\{P^{(n)}, R^{(n)}\}$ є системою альтернатив, а тому $\{\mathbb{A}_P\}$ і $\{\mathbb{A}_R\}$ мають непорожній перетин, який складається з фіксованої кількості елементів виду $\mathbb{A}_{P,R}{}^{<i1, \ldots, ir>} = \{P^{(n)}, R^{(n)}, IF_{i}1^{(n)}, \ldots, IF_{i}r^{(n)}\}$, де $1 \leq r \leq m$, $1 \leq i_r \leq m$, а $IF_1{}^{(n)}, \ldots, IF_m{}^{(n)}$ – це всі тотожно-хибні на D предикати (один з яких може збігатися з якимось із предикатів $P^{(n)}$, $R^{(n)}$, якщо останній тотожно-хибний на D; також ми допускаємо випадок, коли $r = m = 0$ і список $<IF_1{}^{(n)}, \ldots, IF_m{}^{(n)}>$ порожній, тобто, коли в Ω не входять ніякі тотожно-хибні предикати, можливо, за винятком предиката $P^{(n)}$ або $R^{(n)}$); звідси $\|\{\mathbb{A}_P\} \cap \{\mathbb{A}_R\}\| = 2^m$. Далі, очевидно, що всяка $\mathbb{A}_P$ має вигляд $\{P^{(n)}, S_1{}^{(n)}, \ldots, S_k{}^{(n)}, IF_{i}1^{(n)}, \ldots, IF_{i}r^{(n)}\}$, а всяка $\mathbb{A}_R$ – вигляд $\{R^{(n)}, T_1{}^{(n)}, \ldots, T_l{}^{(n)}, IF_{i}1^{(n)}, \ldots, IF_{i}r^{(n)}\}$, де $(S_1{}^{(n)} \vee \ldots \vee S_k{}^{(n)}) \equiv R^{(n)}$ і $(T_1{}^{(n)} \vee \ldots \vee T_l{}^{(n)}) \equiv P^{(n)}$. Останнє означає, що існують два відображення $\phi_1{}^{(1)}$ та $\phi_2{}^{(1)}$ множин $\{\mathbb{A}_P\}$ і $\{\mathbb{A}_R\}$ в їхній перетин, які задаються як наступні підстановки:

$$\phi_1(\mathbb{A}_P) = \theta\,(\{S_1{}^{(n)}, \ldots, S_k{}^{(n)}\}, \{R^{(n)}\}, \mathbb{A}_P),$$
$$\phi_1(\mathbb{A}_R) = \theta\,(\{T_1{}^{(n)}, \ldots, T_l{}^{(n)}\}, \{P^{(n)}\}, \mathbb{A}_R).$$

У випадку, коли існують відображення $\phi_1{}^{(1)}$ та $\phi_2{}^{(1)}$, називатимемо перетин $\{\mathbb{A}_P\} \cap \{\mathbb{A}_R\}$ множин $\{\mathbb{A}_P\}$ і $\{\mathbb{A}_R\}$ їхнім *спільним коренем*. Відношення «множини $\{\mathbb{A}_P\}$ і $\{\mathbb{A}_R\}$ мають спільний корінь» позначатимемо як '$\{\mathbb{A}_P\} \Bbbk \{\mathbb{A}_R\}$'. Відтак, остаточно маємо: якщо предикати $P^{(n)}$ і $R^{(n)}$ взаємно суперечливі, або, що те саме, контрадикторні/альтернативні, то відповідні їм множини систем альтернатив $\{\mathbb{A}_P\}$ і $\{\mathbb{A}_R\}$ мають спільний корінь, символічно:

$$P^{(n)} \bowtie R^{(n)} \rightarrow \{\mathbb{A}_P\} \Bbbk \{\mathbb{A}_R\}.$$

Зех. За екстенсіонального підходу, перетин множин $\{\mathbb{A}_P\}$ і $\{\mathbb{A}_R\}$ складається або з двох елементів: $\mathbb{A}_1 = \{P^{(n)}\}$ та $\mathbb{A}_2 = \{P^{(n)}, R^{(n)}\}$ – якщо $P^{(n)}$ тотожно-істинний на D – або тільки з одного елемента $\mathbb{A}_2$ – якщо $P^{(n)}$ і $R^{(n)}$ нейтральні, а тотожно-хибних предикатів немає в Ω – або з трьох елементів: $\mathbb{A}_1$, $\mathbb{A}_2$ та $\mathbb{A}_3 = \{P^{(n)}, R^{(n)}, IF^{(n)}\}$, де $IF^{(n)}$ – єдиний тотожно-хибний на D предикат, – якщо $P^{(n)}$ і $R^{(n)}$ обидва здійсненні на D. Всі інші міркування

залишаються в силі, лише замість списків $IF_i1^{(n)}$, ..., $IF_ir^{(n)}$ і $IF_1^{(n)}$, ..., $IF_m^{(n)}$ слід розглядати один єдиний предикат $IF^{(n)}$.

Всі інші випадки відношень між предикатами: несумісність (контрарність) $P^{(n)}$ і $R^{(n)}$, їхня підконтрарність (лож-несумісність), їхня незалежність і, нарешті, сумісність – будуть розглянуті в наступних публікаціях.

Відношення між системами альтернатив. В основу аналізу відношень між системами альтернатив ми покладемо наступний підхід. Нехай дано предикат $S^{(n)} \in \Omega$ множину індексів I та таку множину предикатів $\mathrm{CL}_S = \left\{S_i^{(n)}\right\}_{i \in I}$, яка є системою альтернатив на $\mathrm{dom}_1 S^{(n)}$; називатимемо множину CL_S *класифікацією* предиката $S^{(n)}$. Оскільки на всякій предметній області D можна встановити тотожно-істинний предикат (за інтенсіонального підходу – навіть не один), це означає, що за деякої Ω всяка система альтернатив на D є класифікацією деякого предиката. Наш підхід полягатиме в тому, щоб виясняти, чи передбачає те чи те відношення між заданими системами альтернатив $\mathbb{A}_{[P]} = \mathrm{CL}_P$ і $\mathbb{A}_{[R]} = \mathrm{CL}_R$, котрі суть класифікації заданих предикатів $P^{(n)}$ і $R^{(n)}$, наявність певного відношення між самими предикатами $P^{(n)}$ і $R^{(n)}$. Опишемо одну таку ситуацію.

1. *Ортогональність і багатовимірні класифікації*. Нехай дано множину індексів I та множину $\mathrm{U} = \{\mathbb{A}_i\}_{i \in I}$ систем альтернатив, що складаються з n-місних предикатів, на одній і тій самій D. Називатимемо всі $\mathbb{A}_i$ *ортогональними в сукупності*, я. я. всяка кон'юнкція $\bigwedge_{i \in I} P_i(a_1, ..., a_n)$ така, що $P_i^{(n)} \in \mathbb{A}_i$, істинна хоча б на якихось $a_1, ..., a_n \in \mathrm{D}$. З такого означення негайно випливає, що всі елементи U нетривіальні. Ті самі $\mathbb{A}_i$ будуть *ортогональними попарно*, я. я. всяка кон'юнкція $P_i(a_1, ..., a_n) \wedge P_j(a_1, ..., a_n)$ така, що $P_i^{(n)} \in \mathbb{A}_i \wedge P_j^{(n)} \in \mathbb{A}_j$ $(i, j \in I)$, істинна хоча б на якихось $a_1, ..., a_n \in \mathrm{D}$. Зрозуміло, що якщо всі $\mathbb{A}_i$ суть ортогональні в сукупності, то вони й ортогональні попарно. Зворотне, однак, не має місця. Для доведення розглянемо приклад трьох наступних систем альтернатив: $\mathbb{A}_P = \{P^{(1)}, \neg P^{(1)}\}$, $\mathbb{A}_R = \{R^{(1)}, \neg R^{(1)}\}$, $\mathbb{A}_S = \{S^{(1)}, \neg S^{(1)}\}$, $\mathrm{U} = \{\mathbb{A}_P, \mathbb{A}_R, \mathbb{A}_S\}$. В такому разі, якщо

підібрати область D й предикати $P^{(1)}$, $R^{(1)}$, $S^{(1)}$ так, щоб *не* здійснювалися всього дві альтернативи: ($P^{(1)} \wedge \neg R^{(1)} \wedge \neg S^{(1)}$) та ($\neg P^{(1)} \wedge R^{(1)} \wedge \neg S^{(1)}$) – системи альтернатив $\mathbb{A}_P$, $\mathbb{A}_R$ і $\mathbb{A}_S$ будуть ортогональними попарно, але не в сукупності. Це видно з наступних двох таблиць, в першій з яких дані номери всім альтернативам, які мають місце між двома довільними незалежними одномісними предикатами, а у другій показані альтернативи для введених вище у прикладі предикатів $P^{(1)}$, $R^{(1)}$ та $S^{(1)}$:

Таблиця 1. Альтернативи для незалежних предикатів

	$F(x)$	$G(x)$
1)	**1**	**1**
2)	**1**	**0**
3)	**0**	**1**
4)	**0**	**0**

Таблиця 2. Альтернативи для залежних предикатів

$P(x)$	$R(x)$	$S(x)$	$P(x)\wedge R(x)$	$P(x)\wedge S(x)$	$R(x)\wedge S(x)$
1	**1**	**1**	1)	1)	1)
1	**1**	**0**	1)	2)	2)
1	**0**	**1**	2)	1)	3)
0	**1**	**1**	3)	3)	1)
0	**0**	**1**	4)	3)	3)
0	**0**	**0**	4)	4)	4)

Поняття ортогональності тісно пов'язане з поняттям класифікації. *n-вимірною класифікацією* $\mathrm{Cl}^n(\mathrm{T}, S^{(m)})$ предиката $S^{(m)}$ (а також його області істинності $\mathrm{dom}_1 S^{(m)}$) *на* D *з базисом* T назвемо всяку таку систему альтернатив $\mathbb{A}$ на D, для якої існують множина індексів I, множина предикатів $\mathrm{T} = \{S_i^{(m)}\}_{i \in I}$ і множина систем альтернатив $\mathrm{U} = \{\mathbb{A}_i\}_{i \in I}$ такі, що

- $\|\mathrm{T}\| = n$ (кардинал множини T, а отже, і множин I та U дорівнює n);
- $\mathbb{A}$ є класифікацією $S^{(m)}$;
- $\mathbb{A}_i$ є класифікацією $S_i^{(m)}$ для будь-якого $i \in I$;
- кожен $P^{(m)} \in \mathbb{A}$ для всіх $a_1, ..., a_m \in \mathrm{D}$ еквівалентний на D деякій кон'юнкції $\bigwedge_{i \in I} R_i(a_1, ..., a_m)$, де $R_i^{(m)} \in \mathbb{A}_i$ для кожного $i \in I$;
- для всіх $a_1, ..., a_m \in \mathrm{D}$ всяка кон'юнкція $\bigwedge_{i \in I} R_i(a_1, ..., a_m)$, де $R_i^{(m)} \in \mathbb{A}_i$ для кожного $i \in I$, еквівалентна деякому $P^{(m)} \in \mathbb{A}$.

Елементи множини T називатимемо *вимірами* або *осями* n-вимірної класифікації $\mathbb{A} = \mathrm{Cl}^n(\mathrm{T}, S^{(m)})$. Класифікації $\mathbb{A}_i$ вимірів $S_i^{(m)}$ всякої n-вимірної класифікації $\mathbb{A}$ називатимемо *розгортками* цієї n-вимірної класифікації $\mathbb{A}$. Основну властивість n-вимірних класифікацій встановлює наступне

ТВЕРДЖЕННЯ 8. *Всяка n-вимірна класифікація є нетривіальною системою альтернатив, я. я. множина її розгорток ортогональна в сукупності.*

Доведення тривіальне. За $n = 1$ твердження 8 вироджується в тавтологію.

Розглянемо довільну *одновимірну* класифікацію $\mathbb{A} = \mathrm{Cl}^1(\mathrm{T}, S^{(m)})$. Оскільки в цьому разі, в силу $\|\mathrm{T}\| = 1$, ми можемо покласти $\mathrm{T} = \{S^{(m)}\}$, звідки $\mathrm{U} = \{\mathbb{A}\}$ і $\mathbb{A} = \{S_i^{(m)}\}_{i \in I}$ для деякої I, а отже, $\mathbb{A} = \mathrm{CL}_S$. Це означає, що всяка одновимірна класифікація деякого предиката – це класифікація CL цього предиката в сенсі означення з початку даного пункту.

Поглянемо на відношення ортогональності в сукупності, або *сукупної ортогональності*, через призму заявленого підходу до аналізу відношень між системами альтернатив. Існує явний паралелізм між протиставленням відношень сукупної та попарної ортогональності між системами альтернатив

з одного боку, та протиставленням відношень сукупної та попарної незалежності предикатів. Зокрема, сукупна (попарна) ортогональність класифікацій $\mathbb{A}_i$ предикатів $S_i^{(m)}$ з будь-якої заданої множини $T = \{S_i^{(m)}\}_{i \in I}$ передбачає сукупну (попарну) незалежність самих $S_i^{(m)}$. Це безпосередньо випливає з означення поняття ортогональності. З цього ж означення випливає і зворотне: класифікації незалежних предикатів суть ортогональні. Звідси маємо

ТВЕРДЖЕННЯ 9. *Класифікації $\mathbb{A}_i$ предикатів $S_i^{(m)}$ з будь-якої заданої множини* $T = \{S_i^{(m)}\}_{i \in I}$ *за будь-якої даної I сукупно (попарно) ортогональні, я. я. предикати $S_i^{(m)}$ сукупно (попарно) незалежні на* D.

Розглянемо відношення сукупної та попарної ортогональності на парах систем альтернатив. В цьому разі обидва відношення: сукупне та попарне – виявляться тотожними, тож стане можливим казати про єдине бінарне відношення ортогональності. Воно очевидно буде іррефлексивним, симетричним і не транзитивним (оскільки, якщо $\mathbb{A}_1$ ортогональна $\mathbb{A}_2$, то $\mathbb{A}_2$ ортогональна $\mathbb{A}_1$, але $\mathbb{A}_1$ не ортогональна $\mathbb{A}_1$). Таким чином, *бінарне відношення ортогональності між системами альтернатив має ті самі властивості, що й бінарне відношення несумісності предикатів* – при тому, що предикати, класифікаціями яких є взаємно ортогональні системи альтернатив, самі є незалежними, а не несумісними.

Інші бінарні відношення між системами альтернатив: деталізація, перейменування та зсув – будуть розглянуті в наступних публікаціях.

Застосування теорії систем альтернатив

Питання: «А де це можна застосувати?» стандартно виникає до кожної нової теорії (навіть, якщо відповідь очевидна або вже дана). Враховуючи обмежений обсяг роботи, не обговорюватимемо можливі застосування предметно, а лише вкажемо на деякі з них.

Внутрішньологічні застосування. В логіці теорія систем альтернатив є частиною алгебраїчної теорії предикатів і відповідає баченню Жака Ербрана, який розглядав логіку як теорію відношень. Нижче вказуємо на три більш спеціальні області застосування.

1. *Аналіз предикатів*. Предикат завжди можна зобразити деякою логічною функцією від інших раніше введених предикатів. Автор називає таке зображення *аналізом* даного предиката. В [2] описано загальний підхід до побудови аналізів предикатів, вводяться поняття регулярного аналізу та розкладання предиката на альтернативи і представлена теорема про те, що по будь-якому регулярному аналізу даного предиката можна побудувати його розкладання на альтернативи. Її доведення буде наведено в наступних публікаціях.

2. *Описи стану*. Техніка описів стану Рудольфа Карнапа [1, с. 38] застосовна лише у випадках, в яких всі атомарні речення теоретичної системи попарно незалежні. Але це далеко не завжди так, в тому числі не так у випадках, які наводив сам Карнап в [1]. Використовуючи теорію систем альтернатив, можна описати узагальнення техніки описів стану на загальний випадок довільної множини предикатів, а отже й атомів. Загальна схема побудови відповідної теорії (яка виявляється значно складнішою, ніж теорія Карнапа), описана в [3]. Повна її побудова буде здійснена в подальших публікаціях.

3. *Багатозначні логіки*. Очевидно, що для всякого предиката $F^{(n)}$ і всіх $a_1, ..., a_n \in \mathrm{D}$ умови $F(a_1, ..., a_n) = \mathbf{1}$ та $F(a_1, ..., a_n) = \mathbf{0}$ складають мінімальну (в наших термінах) систему альтернатив на D. Якщо ж замінити множину значень істинності $\mathbb{B} = \{\mathbf{1}, \mathbf{0}\}$ на множину з k значень $\mathbb{B}_k = \{\boldsymbol{i}_1, ..., \boldsymbol{i}_k\}$, де $k > 2$, отримаємо знову мінімальну, але більшу систему альтернатив. Таким чином, вибір системи альтернатив лежить в основі вибору логіки. Це питання досі не досліджувалося.

Математичні застосування. Відзначимо два очевидні випадки.

4. *R-функції*. З функціями k-значних логік тісно пов’язані введені Володимиром Рвачовим *R*-функції, тобто, функції, з областями визначення і значень кожної з яких пов’язана одна й та сама система альтернатив $\mathbb{A}$, і належність значення всякої *R*-функції до конкретної альтернативи з $\mathbb{A}$ визначається винятково альтернативами, до яких належать аргументи даної *R*-функції [4, с. 47]. Досі *R*-функції описувалися саме за допомогою зіставлення їм функцій вибраної k-значної логіки. Постає питання про

застосування в цих випадках апарату теорії систем альтернатив. Це питання досі не досліджувалося.

5. *Теорія імовірностей*. Ця теорія очевидно пов'язана і з багатозначною логікою, і з системами альтернатив. Справді, простір елементарних подій X природно описується мінімальною системою альтернатив, альтернативи з якої можна зобразити рівностями виду $\boldsymbol{P}(a) = \boldsymbol{i}$, де a – елементарна подія, $\boldsymbol{P}$ – функція абсолютної імовірності, а $\boldsymbol{i} \in \mathbb{B}_X$, де $\mathbb{B}_X$ – множина числових значень імовірності для елементарних подій з X. $\mathbb{B}_X$ також можна розглядати як множину значень багатозначної логіки. Наскільки відомо автору, завершених логічних досліджень цього питання досі немає.

Філософські застосування. Відзначимо одне важливе застосування.

6. *Класифікації, типології, покриття*. У філософії та гуманітаристиці класифікації часто розглядаються як метод дослідження. Середньовічна логіка та деякі області досліджень у сучасній психології суцільно складаються з самих класифікацій. Не претендуючи на методологічний аналіз (тут краще звернутися до робіт Карла Поппера, який добре показав, що справжнім науковим методом є пошук причинно-наслідкових зв'язків), теорія систем альтернатив може прояснити самé питання про те, щó вважати класифікацією. Проблема полягає в тому, що часто класифікаціями називають набори одномісних предикатів, які не є альтернативами, бо перетинаються, і/або не утворюють систему альтернатив, бо перелічені не всі. Особливо цим грішать математики, які називають класифікаціями будь-які розподіли і протиставлення взагалі, хоча для багатьох випадків годяться лише більш загальні поняття *типології* та *покриття*. Ці два поняття та їхній зв'язок із поняттям класифікації будуть описані в наступних публікаціях.

Психологічні застосування. Із точки зору психології системи альтернатив пов'язані з практичним вибором, який люди (та інші тварини, наділені психікою) здійснюють на практиці. При цьому доводиться мати справу із системами альтернатив, котрі можуть змінюватися в часі. Відзначимо дві теоретичні проблеми в цій області.

7. *Вибір альтернативи*. Перша проблема полягає в тому, щоб встановити, чи впливає на характеристики вибору потужність системи

альтернатив. Інакше кажучи: якщо альтернатив мало або багато, якщо вони додаються чи зникають – як це впливає на психологічну легкість/складність вибору, на категоричність судження суб'єкта вибору щодо окремих альтернатив, на формування когнітивного дисонансу після здійснення вибору тощо. Друга проблема пов'язана з ранжуванням альтернатив за привабливістю і полягає у встановленні всіх можливих зв'язків між альтернативами, які визначають це ранжування (і призводять в окремих випадках до, приміром, суперечливих, колових ранжувань або до зміни порядку альтернатив у ранжуванні при появі нової або вилученні наявної альтернативи). Всі ці проблеми потребують як теоретичного дослідження, так і постановки експериментів (що вже частково здійснюється різними авторами).

Список використаних джерел

1. Карнап Р. *Значение и необходимость. Исследование по семантике и модальной логике*. Пер. с англ. 2-е изд. Москва: Издательство ЛКИ, 2007. 384 с.
2. Кохан Я. А. Теоретико-модельный анализ предикатов. *Седьмые Смирновские чтения: материалы Междунар. науч. конф.* (Москва, 22–24 июня 2011 г.). Москва: Современные тетради, 2011. С. 20–22.
3. Кохан Я. А. Фактуальные альтернативы вместо описаний состояния. *Восьмые Смирновские чтения: материалы Междунар. науч. конф.* (Москва, 19–21 июня 2013 г.). Москва: Современные тетради, 2013. С. 54–56.
4. Рвачев В. Л. *Методы алгебры логики в математической физике*. Киев: Наукова думка, 1974. 260 с.

РОЗДІЛ 2. ПРАКТИЧНІ АСПЕКТИ ВИКЛАДАННЯ КУРСІВ ПРОГРАМУВАННЯ

Дорошенко Анатолій Юхимович, д.ф.-м.н., професор
Іваненко Павло Андрійович, к.ф.-м.н., науковий співробітник
Яценко Олена Анатоліївна, к.ф.-м.н., старший науковий співробітник

ВЕРИФІКАЦІЯ ПРОГРАМНИХ ТРАНСФОРМАЦІЙ ДЛЯ АВТОТЮНІНГУ ПАРАЛЕЛЬНИХ ПРОГРАМ

Дисципліна «Верифікація та валідація програмних систем» охоплює кілька методів перевірки програмного забезпечення, зокрема, формальних методів доведення правильності програм. В даній роботі авторами запропоновано підхід до перевірки коректності оптимізаційних перетворень паралельних програм, що виконуються автотюнером, з використанням техніки переписувальних правил.

Особливість підходу полягає у тому, що цю перевірку у важливих часткових випадках можна виконати автоматично за вихідним кодом програми. Підхід проілюстровано на прикладі методу вибору оптимальної стратегії обходу індексованих структур даних для програм з ітеративною схемою обчислень. Метод застосовано для оптимізації паралельного алгоритму складної прикладної задачі короткотермінового метеорологічного прогнозування.

Ключові слова: автоматизація оптимізації паралельних програм, автотюнінг, алгебро-динамічні моделі, техніка переписувальних правил, перевірка коректності перетворень

The discipline "Verification and Validation of Software Systems" comprises several methods of software verification including formal ones. In this study, the authors propose an approach to verification of correctness of optimization transformations of parallel programs performed by an auto-tuner using the rewriting rules technique.

The feature of the approach is that in some important partial cases, such verification can be done automatically based on the source code of a program. The approach is illustrated on the example of the method for choosing the optimal strategy for indexed data structures traversal for programs with iterative computing scheme. The method is applied for the optimization of the short-term

meteorological forecasting program.

Keywords: automation of parallel program optimization, auto-tuning, algebra-dynamic models, rewriting rules technique, verification of transformation correctness.

Для розв'язання складних науково-технічних задач необхідні значні обчислювальні потужності, і їх раціональне використання завжди було однією з головних проблем під час розробки прикладних програм. Відомий вислів "ефективні алгоритми завжди кращі за суперкомп'ютери" дуже влучно формулює проблематику паралельних обчислень: наявність потужного обчислювача ще не гарантує успішного виконання обчислень в прийнятних часових межах. Ефективна паралельна програма, окрім вдалої декомпозиції задачі на незалежні підзадачі, повинна мінімізувати витрати на синхронізацію обчислень і пересилання даних, що дуже складно зробити без врахування архітектури середовища виконання. Такі оптимізації зазвичай залежать від апаратної платформи, і програма, яка була оптимізована для однієї платформи, вірогідно не буде оптимальною на іншій платформі. Для досягнення максимальної ефективності програми необхідне її додаткове налаштування під обчислювальне середовище, в якому вона буде виконуватися. Сучасна методологія самоналаштування (автотюнінгу) [1, 2] дозволяє автоматизувати цю процедуру. Ідея автотюнінгу полягає в емпіричному оцінюванні кількох варіантів програми й вибору найкращого. Традиційно підбір виконує окрема програма-тюнер, вона ж відповідає за генерацію різних модифікацій вихідної програми. Основними критеріями оцінки, зазвичай, є швидкодія й точність отриманих результатів.

У даній роботі запропоноване програмне рішення, що дозволяє тюнеру автоматично генерувати нові коректні версії програм, спираючись на експертне знання розробника, оформлене у вигляді метаданих у вихідному коді. Проілюстрована автоматизація верифікації коректності перетворень вихідного коду програми на прикладі методу вибору оптимальної стратегії обходу індексованих структур даних для покращення швидкодії програми, що оптимізується. Ця перевірка ґрунтується на властивостях, сформульованих у термінах дискретних динамічних систем для паралельних програм.

Властивості програм

Виконання будь-якої програми можна змоделювати з використанням теорії дискретних динамічних систем (ДДС) [3]. ДДС задається як трійка , де S – простір станів; – множина початкових станів; – бінарне відношення переходів у просторі станів. Система може перейти із стану s_i в стан s_j, якщо $(s_i, s_j) \in d$. Формальна модель автотюнінгу S^{tune} як еволюційне розширення моделі дискретних динамічних систем розроблена в [4]. Вона базується на ДДС для мультипотокових програм S^{mt}, яка в свою чергу побудована на моделі гпослідовних програм S^{ser}. Програми у цих моделях подані у вигляді виразів алгебри Глушкова (АГ) [3], які інтерпретуються як функції ДДС.

Визначимо деякі властивості програм, які допоможуть проаналізувати коректність й ефективність оптимізаційних перетворень, що виконує тюнер. Під *коректністю* перетворень будемо розуміти те, що вихідна й перетворена програми повертають однаковий результат. Формально цю умову можна сформулювати таким чином. Нехай задана підмножина $V_R \subset V$ результуючих змінних. Тоді дві програми P^1 й P^2 будемо називати еквівалентними за результатом, якщо для однакових початкових даних b_0 програми одночасно приходять або не приходять в фінальні стани $s_f^1 = (b^1, \varepsilon, \varnothing)$ й $s_f^2 = (b^2, \varepsilon, \varnothing)$, причому ці фінальні стани пам'яті співпадають за результуючими змінними $b^1_{V_R} = b^2_{V_R}$.

Введемо поняття Δ-коректності для класу задач, які дозволяють поступитися точністю результатів обчислень заради пришвидшення швидкодії програми. Дві програми P^1 й P^2 будемо називати еквівалентними за результатом з Δ-точністю, якщо для однакових початкових даних b_0 програми одночасно приходять або не приходять в фінальні стани $s_f^1 = (b^1, \varepsilon, \varnothing)$ й $s_f^2 = (b^2, \varepsilon, \varnothing)$, причому ці фінальні стани пам'яті відхиляються не більше ніж на величину Δ: $r(b^1_{V_R}, b^2_{V_R}) \le \Delta$.

Еквівалентність за результатом – достатньо загальна властивість, проте реалізувати перевірку цієї властивості в загальному випадку

практично неможливо, оскільки для цього необхідний аналіз усіх можливих шляхів виконання програми в залежності від вхідних даних й варіантів виконання різних потоків. Тому необхідне визначення більш часткових властивостей для опису коректності перетворень, які можуть бути практично перевірені.

Спочатку визначимо деякі властивості операторів АГ. Для кожного оператора $y \in Y$ визначені множини $R(y) \subset V$ змінних, від яких залежить результат застосування оператора, й $W(y) \subset V$ змінних, які змінюють значення в результаті виконання оператора. Два оператора $y_1, y_2 \in Y$ будемо називати залежними (за даними), якщо $R(y_1) \cap W(y_2) \neq \varnothing$ або $W(y_1) \cap R(y_2) \neq \varnothing$, або $W(y_1) \cap W(y_2) \neq \varnothing$. Два оператори є перестановочними або, комутативними (відносно операції композиції) якщо $y_1 y_2 = y_2 y_1$. Очевидним є те, що незалежні оператори є перестановочними, проте обернене твердження не є істинним – наприклад, дві копії одного оператора завжди перестановочні, але вони залежні якщо $W(y) \neq \varnothing$.

Тепер визначимо такі властивості програм*: безтупиковість, безконфліктність* й *еквівалентність за операторами*. Програму P будемо називати *безтупиковою*, якщо при її виконанні не з'являються тупикові стани (стан коли неможливе подальше виконання залишку програми у ДДС). Очевидно що усі послідовні програми в моделі S^{ser} є безтупиковими.

В моделі будемо називати конфліктним переходом ситуацію, коли існують залежні оператори, які виконуються одночасно у різних потоках. Програму P будемо називати безконфліктною, якщо при її виконанні не з'являються конфліктні переходи. Для безконфліктних програм можна не використовувати функцію merge [3, 4]. Замість об'єднання результатів одночасного виконання декількох операторів можна виконати ці оператори послідовно у будь-якій послідовності.

Якщо відома послідовність виконання програми (послідовність переходів ДДС), можна визначити історію використання базових операторів. Для цього додамо до стану ДДС ще одну компоненту $h \in Y$. В початковому стані $h = \varepsilon$. Також модифікуємо правило переходу:

$(b, \gamma R, F, h) \to (\gamma(b), R, F, h\gamma)$. При одночасному виконанні декількох таких правил у різних потоках будемо додавати відповідні оператори y у довільному порядку. Оператор h у фінальному стані будемо називати історією використання операторів.

Дві історії h_1, h_2 будемо вважати еквівалентними (відносно визначеної системи рівності операторів АГ) якщо h_2 можна отримати з h_1 послідовним застосуванням операції рівності до операторів історії. Зокрема, система рівності завжди включає усі відношення які визначають комутативність операторів. Тобто h_2 також можна отримати із h_1 послідовними перестановками комутативних пар операторів. Дві програми P^1 й P^2 будемо називати еквівалентними за операторами, якщо для однакових вхідних даних b_0 вони породжують еквівалентні історії виконання операторів.

Тепер можна сформулювати наступне твердження.

Лема 1. Якщо дві програми P^1 й P^2 безтупикові, безконфліктні й еквівалентні за операторами, то вони еквівалентні за результатом.

Доведення. Безтупикові програми не переходять у тупикові стани, отже, вони або переходять в фінальний стан, або не зупиняються взагалі. В останньому випадку історія операторів буде нескінченно, а тому, якщо така ситуація має місце для P^1, то вона буде виконуватися й для P^2. Отже, перша частина визначення еквівалентності за результатом доведена. Перевіримо тепер, що на однакових вихідних даних програми обчислюють однаковий результат. Обчислення програми можна подати у вигляді виконання деякого комбінованого оператора y_f над вихідним станом пам'яті $b_0 : b_f = y_f b_0$. Виходячи з побудови ДДС й S^{mt}, комбінований оператор є послідовною композицією операторів, які виконуються на кожному переході ДДС. Для системи кожний перехід поєднує множину переходів окремих потоків; проте, для безконфліктних програм перехід системи в цілому еквівалентний послідовній композиції переходів окремих потоків, які розглядаються у довільному порядку.

Таким чином, для безконфліктних програм комбінований оператор співпадає з історією виконання операторів, тобто $y_f = h$. З визначення еквівалентності за операторами виплаває, що $\forall b \in B,\ (h^1 b)_{V_R} = (h^2 b)_{V_R}$, тобто історії операторів діють однаково на результуючі дані. Тому $b^1_{V_R} = (h^1 b_0)_{V_R} = (h^2 b_0)_{V_R} = b^2_{V_R}$, що й треба було довести.

Отже, перевірка еквівалентності за результатом зводиться до перевірки цих трьох властивостей, що є цілком можливим для багатьох задач.

Перевірка властивостей

Властивості безтупиковості, безконфліктності й еквівалентності за операторами, які було визначено у попередньому розділі, в загальному випадку важко перевірити. Проте, для окремих часткових випадків, можна вказати достатні умови для забезпечення цих властивостей, які легше перевірити. Розглянемо деякі такі умови.

Умова еквівалентності за операторами практично означає, що деякі комутативні оператори програми були переставлені в історії виконання. Вимогу комутативності для перестановки операторів в межах послідовної композиції будемо позначати як *CommutativeOperators*. У випадках, коли окремі ітерації циклу виконуються в іншій послідовності, для еквівалентності за операторами необхідно, щоб комутативними були окремі ітерації (будемо позначати цю умову як *CommutativeIterations*) та щоб була взаємно однозначна відповідність між ітераціями у обох програмах (вихідній й перетвореній). Останню умову позначимо *IdenticalIterations*.

Перевірка умов комутативності – достатньо складна задача у загальному випадку, оскільки потребує аналізу властивостей усіх операторів, викликів функцій й можливих варіантів виконання програми. У багатьох випадках комутативність можна встановити на основі незалежності операторів циклу (для умови *CommutativeIterations*), що значно легше зробити з використанням техніки переписувальних правил. В інших випадках будемо виконувати перевірку результатів програми.

Наприклад, для перевірки властивості *CommutativeIterations* деякого циклу програми можна згенерувати програму з оберненим порядком ітерацій. Якщо у вихідній програмі ітерації були перестановочними, то модифікована програма повинна обчислити той самий результат. Очевидно, що збіг результатів тесту ще не гарантує комутативності ітерацій, оскільки можлива ситуація, коли на деякому іншому наборі даних результати обчислень будуть відрізнятися. Для зменшення вірогідності такої ситуації можна використовувати тести з більш складними трансформаціями. Наприклад, розгорнути дві ітерації циклу в одну й перемішати оператори цих ітерацій.

Важливою перевагою використання фреймворку TermWare й техніки переписувальних правил [5] є можливість автоматизувати створення великого набору різних тестів для кожної конкретної програми. Цей фреймворк дозволяє для деякої предметної області визначити відповідність між операторами й предикатами АГ й програмним кодом деякої мови програмування. Така відповідність дає змогу використовувати переписувальні правила TermWare для автоматичного переходу між двома рівнями подання програми.

Умову безконфліктності можна сформулювати таким чином: будь-які два залежні оператори, що можуть виконуватися у різних потоках, повинні бути захищеними відповідними критичними секціями. Далі будемо позначати цю умову як *AllNeededCS*. Безпосередня перевірка цієї умови вимагає виконання повного аналізу залежностей програми і є складною задачею. Проте, в деяких часткових випадках перевірка значно спрощується. Наприклад, незалежними будуть оператори, що змінюють лише локальні змінні для потоку.

Перейдемо до розгляду властивості безтупиковості й почнемо з випадку, коли в програмі відсутні оператори *wait*. Цю умову легко перевірити за допомогою системи правил TermWare:

$$wait(\$x) \rightarrow NIL[error()].$$

Без операторів *wait* потенційні тупикові ситуації можуть з’явитися лише через використання операторів *lock*. В цьому випадку залежність між критичними секціями можна зобразити у вигляді орієнтованого графа:

вершинами будуть усі критичні секції програми cs_i, а ребро з вершини cs_i в додається у випадку, коли деякий потік T_k може виконати оператор $lock(cs_j)$ з критичної секції. У такій інтерпретації безтупиковість програми еквівалентна властивості ациклічності (позначимо як *NoCyclicCriticalSections*) побудованого графа.

Частковим випадком є ситуація, коли програма не містить критичних вкладених секцій (умова *NoEmbeddedCriticalSections*) – граф критичних секцій буде ациклічним, оскільки в ньому взагалі не буде ребер. Цю умову досить легко автоматично перевірити з використанням переписувальних правил TermWare:

1. *Method*(\$*head*, \$*body*) → *Method*(\$*head*, [*m*0 : \$*body*], *_Mark*);
2. *m*0 : [*lock*(\$*cs*) : \$*x*]] → [*lock*(\$*cs*) : [*m*1(\$*cs*) : \$*x*]];
3. *m*1(\$*cs*) : [*unlock*(\$*cs*) : \$*x*]] → *unlock*(\$*cs*) : [*m*0 : \$*x*]];
4. *m*1(\$*cs*) : [*lock*(\$*cs*1) : \$*x*]] → *NIL*[*error*()];
5. *m*0 : [\$*x* : \$*y*]] → [\$*x*: [*m*0 : \$*y*]];
6. *m*1(\$*cs*) : [\$*x* : \$*y*]] → [\$*x* : [*m*1(\$*cs*) : \$*y*]].

Тут правило 1 додає спеціальний маркувальний терм m_0 в початок кожного методу. Цей терм використовується під час обходу всіх операторів методу. Правило 2 замінює маркер m_0 на m_1, якщо при обході зустрічається критична секція. Правило 3 виконує зворотне перетворення, якщо маркер пересувається до точки завершення критичної секції й нові секції не зустрічаються (виконується властивість *NoEmbeddedCriticalSections*). Якщо ж умова *NoEmbeddedCriticalSections* порушується – система правил сигналізує про помилку за рахунок правила 4. Правила 5 й 6 забезпечують пересування маркерів m_0 й m_1 між операторами методу.

Звісно, порушення умови *NoEmbeddedCriticalSections* ще не вказує на те, що у програмі є помилка, яка призводить до тупика. В таких випадках необхідно перевіряти більш загальну вимогу *NoCyclicCriticalSections*. Варто зауважити, що багато прикладних паралельних програм задовольняють властивості *NoEmbeddedCriticalSections*, оскільки такий стиль використання синхронізації суттєво простіший для розуміння.

Перевірка безтупиковості суттєво ускладнюється, якщо в програмі використовуються оператори *wait* – у цьому випадку для кожного оператора $wait(ev_i)$ має бути присутнім відповідний оператор $signal(ev_i)$. Це дуже важко перевірити для усіх можливих варіантів виконання програми.

Тому замість загальних умов безтупиковості при використанні операторів *wait* будемо формулювати й розглядати деякі шаблони їх використання, для яких гарантується коректність.

Наприклад, оператор бар'єрного очікування $Barrier(ts_i)$, який задає точку в програмі, у якій кожен потік очікує усі інші або деяку частину, може бути заданий таким чином:

$$Barrier(k, b_i) = lock(cs\_b_i);\ inc(count\_b_i);$$
$$if\ (count\_b_i < k,\ unlock(cs\_b_i); wait(ev\_b_i),$$
$$assign(count\_b_i, 0);\ signall\_all(ev\_b_i);\ unlock(cs\_b_i)).$$

Як видно з визначення, цей складений оператор використовує додаткову змінну, критичну секцію й точку синхронізації. У додатковій змінній зберігається кількість потоків, які вже очікують за бар'єром. Якщо їх менше за параметр, то поточний потік також буде очікувати. Як тільки потоків збирається штук, усі вони вивільнюються й у змінну записується 0.

Оператор бар'єру не буде спричиняти тупики, якщо загальна кількість потоків, де він використовується, дорівнює й у кожному потоці виконується однакова кількість операторів. У багатьох сучасних мовах з підтримкою багатопотоковості є безпосередня реалізація оператора бар'єру, тому немає необхідності реалізовувати його через базові оператори *wait* та *signal*. Проте, сформульована умова безтупиковості програми з бар'єрами в цьому випадку також залишається істинною.

Метод вибору оптимальної стратегії обходу індексованих структур даних та його програмна реалізація

Розглянемо метод оптимізації використання процесорного кешу програмами з ітеративною схемою обчислень. Як відомо, сучасні процесори мають декілька рівнів кешів з різною швидкістю доступу. Розміри цих кешів зазвичай відрізняються на порядок, тому швидкодія програми суттєво

залежить від частоти переміщення даних між цими рівнями. Запропонований метод автоматично генерує й оцінює швидкодію інверсного напрямку ітерування за даними й, у першу чергу, орієнтований на задачі, що працюють з великими багатомірними даними. Для одномірних масивів послідовне зчитування елементів масиву у більшості випадків автоматично буде найшвидшим за рахунок апаратної оптимізації – доступ до сусідніх елементів значно швидший, якщо вони вміщуються в одну кеш-лінію. Проте, якщо програма працює з багатовимірними масивами або з їх поданням у одновимірному масиві, частіше за все сусідні з точки зору ітерацій циклу елементи не будуть потрапляти до однієї кеш-лінії. Тому варто оцінити усі комбінації ітерування – їх кількість дорівнює 2^d, де d – кількість вкладених циклів.

Позначимо вихідний варіант програми через *P*1, а її трансформовану варіацію зі зміненим напрямком ітерування довільного циклу – через *ReversedCycleP*1. Перевірка властивості комутативності ітерацій *CommutativeIterations* в загальному вигляді є складною, тому розглянемо декілька часткових випадків, для яких така верифікація можлива.

Введемо нову властивість, яка характеризує обчислення, що виконуються в межах ітерацій одного циклу. Будемо казати, що ітерації циклу задовольняють вимогам *AllLocalVariables*, якщо внутрішні оператори використовують лише внутрішні змінні. Ця властивість перевіряється системою переписувальних правил TermWare, яка дозволяє використовувати лише одну зовнішню змінну – лічильник циклу.

Теорема 1. Ітерації циклу є комутативними (властивість *CommutativeIterations*), якщо вони задовольняють властивості *AllLocalVariables*.

Доведення досить очевидне, оскільки властивість *AllLocalVariables* виключає будь-яку залежність за даними між різними ітераціями циклу.

Зауважимо, що властивість *AllLocalVariables* не має великої практичної цінності – оператори циклу не використовують індексованих структур (одно/багатовимірні масиви) й зміна напрямку ітерування не впливає на ефективність використання кешів, тому швидкодія програми не буде залежати від виконуваної трансформації. Тому розглянемо більш

універсальний випадок, коли оператори циклу або читають, або записують значення у зовнішні змінні, але ніколи не роблять це одночасно. Така властивість, яку далі будемо позначати як *ReadOrWriteAccessToVariables*, виключає рекурентну схему обчислень. Ця властивість, як і *AllLocalVariables*, виключає залежність за даними між ітераціями циклу, з чого випливає виконання умови їх комутативності *CommutativeIterations*. Типовим прикладом такої схеми обчислень є простий послідовний алгоритм множення матриць – кожна ітерація циклу зчитує відповідні значення з вихідних матриць й записує результат у іншу змінну.

Сформулюємо ще одну властивість – властивість "локальності" операторів циклу у тому сенсі, що вони виконуються в одному потоці й не залежать від даних з інших потоків. Будемо позначати таку властивість циклів через *OnlySequentialCalculations.* Цю властивість легко перевірити системою переписувальних правил, яка забороняє використання операторів для багатопотокової взаємодії *call_thread, wait, signal, signal_all, lock* та *unlock.*

Теорема 2. Програми *P*1 й *ReversedCycleP*1 еквівалентні за результатом, якщо ітерації трансформованих циклів задовольняють властивостям *OnlySequentialCalculations* й *ReadOrWriteAccessToVariables*.

Доведення. Для доведення теореми згідно з лемою 1 необхідно перевірити властивості безтупиковості, безконфліктності й еквівалентності за операторами програм *P*1 й *ReversedCycleP*1.

Безтупиковість обох програм випливає з властивості *OnlySequentialCalculations* й характеру трансформації – змінюється лише напрям ітерування циклу(ів).

Відсутність взаємодії між потоками гарантує відсутність конфліктних переходів у *P*1 й *ReversedCycleP*1, а тому й безконфліктність програм.

Для доведення еквівалентності за операторами необхідно, щоб виконувалися умови *CommutativeIterations* й *IdenticalIterations*. Умова *IdenticalIterations* гарантується самою трансформацією – змінюється лише порядок ітерацій циклу, а не їх кількість. Властивість *CommutativeIterations* (комутативність ітерацій циклу) випливає з умови *ReadOrWriteAccessToVariables.*

Застосування методу вибору оптимальної стратегії обходу індексованих структур даних на практиці можна звести до розмічення циклів у вихідному коді програми прагмами-коментарями [6]. За цими мітками автотюнер генерує систему переписувальних правил, яка розвертає ітерації циклу у зворотному напрямку. Наприклад, у мові Java для циклів типу:

```
// bidirectionalCycle
for (int loopVar = 0; loopVar < 42; loopVar++)
{
  ...
}
```

відповідне переписувальне правило виглядає таким чином:

ForStatement (
 LoopHead (
  ForInit(*VariableDeclarator*(*Id*(“*loopVar*”), *Literal*($*initialValue*))),
  RelationalExpression(*Identifier*(“*loopVar*”), “<”, *Literal*($*endValue*)),
  ForUpdate(*StatementExpression*(*Identifier*(“*loopVar* "), “++”))),
  Block(.....)) →
 ForStatement(
  LoopHead(*ForInit*(*VariableDeclarator*(*Id*(“*loopVar*”),
Literal($*endValue*))),
   RelationalExpression(*Identifier*(“*loopVar*”), “>”,
Literal($*initialValue*)),
   ForUpdate(*StatementExpression*(*Identifier*(“*loopVar*”), “--”))),
   Block(.....)).

Це правило, використовуючи пропозиційні змінні, міняє місцями початкове і фінальне значення лічильника циклу й замінює оператор зміни значення цього лічильника на кожній ітерації циклу.

Приклад застосування

Запропонований метод було використано під час оптимізації паралельного алгоритму задачі короткотермінового метеорологічного

прогнозування [7]. Ця задача була обрана для практичного розгляду, оскільки вона має високу обчислювальну складність і природну необхідність у максимально швидкому й точному результаті. Її програмна реалізація поєднує геометричне та операторне розщеплення, а схема обчислень є ітеративною (див. рисунок 1). На вхід першої ітерації t_0 подаються поточні значення метеорологічних величин у області, що розглядається, наприклад, напрям й швидкість вітру, вологість, температура повітря, тощо. Результатом обчислення кожної ітерації є прогноз стану системи через деякий час Δt, який подається на вхід наступній ітерації t_1. Чим більший часовий крок, тим більша прогностична похибка, тому рекомендовано використання часового кроку від десятків до сотень секунд. Для зберігання даних використовуються чотиривимірні масиви. Повний опис математичної моделі, декомпозиції області обчислень та методу розщеплення наведено в [8]. Паралельна чисельна реалізація цієї моделі здійснюється відповідно до трирівневого алгоритму, який передбачає розпаралелювання за рівняннями, просторовими напрямками та підобластями [7]. Залежність часу виконання від кількості підобластей наведено на рисунку 2.

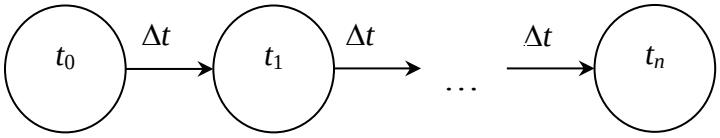


Рисунок 1. Загальна схема обчислень у задачі метеорологічного прогнозування

Чисельний експеримент проводився на гомогенній ЕОМ зі спільною пам’яттю та 24-ма задіяними процесорами. Зміна лише напрямку ітерування даними в результаті дала покращення загального часу виконання обчислень на 13 %, що у сукупності з геометричною декомпозицією вхідних даних й паралельною реалізацією операторного розщеплення дозволило досягти

мультипроцесорного прискорення $W(24) = 18.36$ й загальної ефективності $E(24) = 0.76$, що є гарним показником.

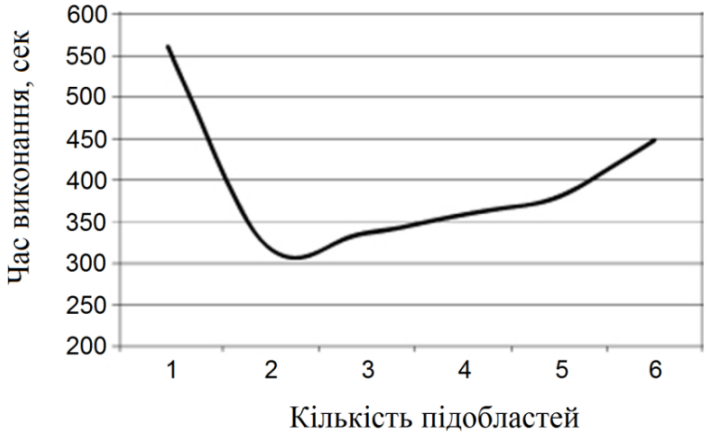


Рисунок 2. Залежність часу виконання паралельної програми метеорологічного прогнозування від кількості підобластей розбиття

Висновки. Запропоновано підхід до перевірки коректності оптимізаційних перетворень паралельних програм, що виконуються автотюнером, з використанням техніки переписувальних правил. Гнучкість підходу полягає у тому, що цю перевірку у часткових випадках можна виконати автоматично за вихідним кодом програми. Підхід проілюстровано на прикладі методу вибору оптимальної стратегії обходу індексованих структур даних для програм з ітеративною схемою обчислень. Метод застосовано для оптимізації паралельного алгоритму складної прикладної задачі короткотермінового метеорологічного прогнозування. Результати практичного експерименту продемонстрували підвищення ефективності оптимізованої програми в сенсі часу виконання та підтвердили доцільність застосування методу.

Список використаних джерел

1. Software automatic tuning: from concepts to state-of-the-art results / Naono K., Teranishi K., Cavazos J., Suda R. – Berlin: Springer, 2010. – 377 p.
2. Durillo J. From single- to multi-objective auto-tuning of programs: Advantages and implications / J. Durillo, T. Fahringer // Scientific programming – automatic application tuning for HPC architectures, 2014, Vol. 22. – N 4. – P. 285-297.
3. Методы алгебраического программирования. Формальные методы разработки параллельных программ / [Ф. И. Андон, А. Е. Дорошенко, К. А. Жереб и др.]. – Киев: Наукова думка, 2017. – 440 с.
4. Иваненко П. А. Метод автоматической генерации автотьюнеров для параллельных программ / П. А. Иваненко, А. Ю. Дорошенко // Кибернетика и системный анализ, 2014. – №3. – С. 75-83.
5. Дорошенко А. Е. Система символьных вычислений для программирования динамических приложений / А. Е. Дорошенко, Р. С. Шевченко // Проблемы программирования, 2005. – №4. – С. 718-727.
6. Ivanenko P. TuningGenie: auto-tuning framework based on rewriting rules / P. Ivanenko, A. Doroshenko, K. Zhereb // Proc. 10th International Conference “ICT in Education, Research, and Industrial Applications” (ICTERI 2014), Revised Selected Papers, Kherson, Ukraine (9-12 June 2014). – Berlin: Springer, 2014. – P. 139-158.
7. Автоматизоване проектування програм для розв’язання задачі метеорологічного прогнозування / А. Ю. Дорошенко, П. А. Іваненко, О. М. Овдій, О. А. Яценко // Проблеми програмування, 2016. – №1. – С. 102-115.
8. Черниш Р. І. Модифіковане адитивно-усереднене розщеплення, його паралельна реалізація та застосування до задач метеорології : автореф. дис. на здобуття наук. ступеня канд. фіз.-мат. наук : спец. 10.05.02 / Черниш Руслан Іванович – Київ, 2010. – 20 с.

Волохов Віктор Миколайович, к.ф.-м.н, доцент

ТЕОРЕТИЧНІ ТА ПРАКТИЧНІ АСПЕКТИ РОЗРОБКИ МОВНИХ ПРОЦЕСОРІВ

Надання студентам теоретичних знань та практичних навичок щодо побудови мовних процесорів (трансляторів та інтерпретаторів) – основна мета курсу «Системне програмування». У роботі викладення матеріалу починається з теоретичних результатів з подальшим переходом до аналізу практичних підходів у розробці компонент мовного процесора.

Ключові слова: мовний процесор, системне програмування.

Providing students with theoretical knowledge and practical skills in building language processors (translators and interpreters) is the main goal of the course "System Programming". The presentation of the material begins with theoretical results with the subsequent transition to the analysis of practical approaches in the development of components of the speech processor.

Keywords: language processor, system programming.

При вивченні мов програмування, як правило, виділяють три аспекти: прагматичний, семантичний та синтаксичний [1].

Прагматичний аспект (прагматика мови програмування) визначає клас задач, на рішення яких орієнтується мова програмування. Як правило, прагматичний аспект менш формалізований в порівнянні з семантичним та синтаксичним аспектами [2].

З урахуванням прагматичного аспекту мови програмування можна поділити на процедурні та непроцедурні. Процедурні мови програмування орієнтовані перш за все на опис (визначення) алгоритмів, тобто по суті використовуються для побудови процедур обробки даних. До таких мов ми відносимо всім відомі мови програмування, такі як Pascal [3], C++[4], Java [5] тощо.

Непроцедурні мови програмування на відміну від процедурних неявно визначають процедури обробки даних. Частіше всього, такі мови використовуються для побудови завдань на обробку даних. До

непроцедурних мов програмування ми відносимо командні мови операційних систем, мови управління в пакетах прикладних програм тощо.

Як процедурні, так і непроцедурні мови програмування можуть орієнтуватися як на декілька класів задач, так і конкретну предметну область. В першому випадку ми будемо говорити про універсальні мови програмування (Pascal, C++, Java), в другому – про спеціалізовані мови програмування (Snobol [6], Lisp [7]).

Семантичний аспект (семантика мови програмування) визначається шляхом конкретизації базових функцій обробки даних, набору конструкцій управління та методами побудови більш "складних" програм на основі "простих". Наприклад, визначивши як базовий тип даних "рядок" ми повинні запропонувати "традиційний" набір функцій обробки таких даних: порівняння рядків, виділення частини рядка, конкатенацію рядків та ін [2].

Семантика мови програмування має бути визначена формально, інакше в подальшому неможливо буде побудувати відповідний мовний процесор. На сьогодні існують два основних напрямки визначення семантики мов програмування: методи денотаційної семантики та методи операційної семантики. Методи денотаційної семантики базуються на класах алгебр, методи операційної семантики базуються на синтаксичних структурах програм.

Синтаксичний аспект (синтаксис мови програмування) визначає набір синтаксичних конструкцій мови програмування, які використовуються для нотації (запису) семантичних одиниць в програмі. Про синтаксис мови програмування можна сказати як про форму, яка є суть похідною від семантики. Для визначення (опису) синтаксису мови програмування використовуються як механізми, що орієнтовані на синтез, так і механізми, орієнтовані на аналіз. Задачі аналізу та синтезу синтаксичних структур програм – це дуальні задачі. Їх конкретизацію ми будемо розглядати в наступних розділах [8].

Виходячи з вищенаведеного, щоб побудувати мову програмування потрібно:

- визначити клас (класи) задач, на розв'язок яких орієнтована мова програмування;

- виділити базові типи даних та функції їх обробки, указати конструкції управління в програмах. Побудувати механізми конструювання більш складних програм та структур даних на основі більш простих одиниць;
- визначити синтаксис мови програмування.

Мовний процесор призначений для обробки програм відповідної мови програмування. З точки зору прагматики, мовні процесори діляться на транслятори та інтерпретатори.

Мовний процесор типу транслятор (транслятор) [8] – це програмний комплекс, котрий на вході отримує текст програми на вхідній мові, а на виході видає версію програми на вихідній мові, що називається об'єктною мовою. В більшості випадків як об'єктна мова виступає мова команд деякої обчислювальної машини. Серед трансляторів можна виділити дві програмні системи:

- компілятори – транслятори з мов програмування високого рівня;
- асемблери – транслятори машинно-орієнтованих мов програмування.

Мовний процесор типу інтерпретатор (інтерпретатор) – це програмний комплекс, котрий на вході отримує текст програми на вхідній мові та вхідні дані, які в подальшому обробляються програмою, а на виході видає результати обчислень (вихідні дані).

Оскільки транслятори та інтерпретатори реалізують мови програмування, вони мають спільні риси: їх структура досить схожа, в основу їх реалізації покладено спільні теоретичні результати та практичні практичні підходи до реалізації.

Оскільки основна мета курсу «Системне програмування» це розробка лексичних аналізаторів мов програмування.

Термінологічні позначення, якими ми будемо у подальшому користуватися:

- Σ - основний алфавіт – скінчена множина символів;

- Σ* - множина слів в алфавіті Σ. В цю множину входить ε-слово. Його властивості: $w\varepsilon = \varepsilon w = w$, тобто $|\varepsilon| = 0$.

- $\Sigma^{+} = \Sigma^{*} \setminus \{\varepsilon\}$

- $\Sigma^{*\mathbf{k}}$ - множина слів в алфавіті Σ, довжина яких не більше k.

- $\Sigma^{+\mathbf{k}} - \Sigma^{*\mathbf{k}} \setminus \{\varepsilon\}$.

Словарну множину $L \subseteq \Sigma^{*}$ будемо називати мовою (на самому високому рівні абстракції).

Призначення основних компонентів транслятора (рис.1):

1. *Лексичний аналізатор*.

Вхід: вхідний текст (послідовність літер) програми.

Вихід: послідовність лексем програми.

Лексема – це ланцюжок літер, що має певний семантичний зміст. Всі лексеми мови програмування (їх кількість, як правило, нескінчена) можна розбити на скінчену множину класів. Для більшості мов програмування актуальні наступні класи лексем:

- зарезервовані слова;
- ідентифікатори;
- числові константи (цілі та дійсні числа);
- літерні константи;
- рядкові константи;
- коди операцій;
- коментарі. Коментарі безпосередньо не несуть інформації щодо структури програми. В подальшому вони не використовуються, тобто не передаються синтаксичному аналізатору.
- дужки та інші елементи програми.

2. *Синтаксичний аналізатор*.

Вхід: послідовність лексем програми.

Вихід: - “Так” + синтаксична структура (синтаксичний терм) програми,

- “Ні” + синтаксичні помилки в програмі.

3. Семантичний аналізатор.

Вхід: Синтаксичний терм програми.

Вихід: - “Так” + семантична структура (семантичний терм) програми,

- “Ні” + семантичні помилки в програмі.

4. *Оптимізація проміжного коду.*

Вхід: семантичний терм програми.

Вихід: оптимізований семантичний терм програми.

Оптимізація – це еквівалентне перетворення програми на основі певних критеріїв. Серед критеріїв оптимізації можна виділити оптимізацію по пам'яті та оптимізацію по швидкості виконання результуючої програми. В залежності від підходів по оптимізації програми можна розглядати машиннозалежні та машиннонезалежні методи оптимізації. На відміну від машиннонезалежних методів машиннозалежні методи оптимізації враховують архітектурні особливості ЕОМ, наприклад, наявність апаратного стека, наявність вільних регістрів тощо.

5. *Генерація об'єктного коду.*

Вхід: семантичний терм програми.

Вихід: результуючий (об'єктний) код програми.

Структура транслятора

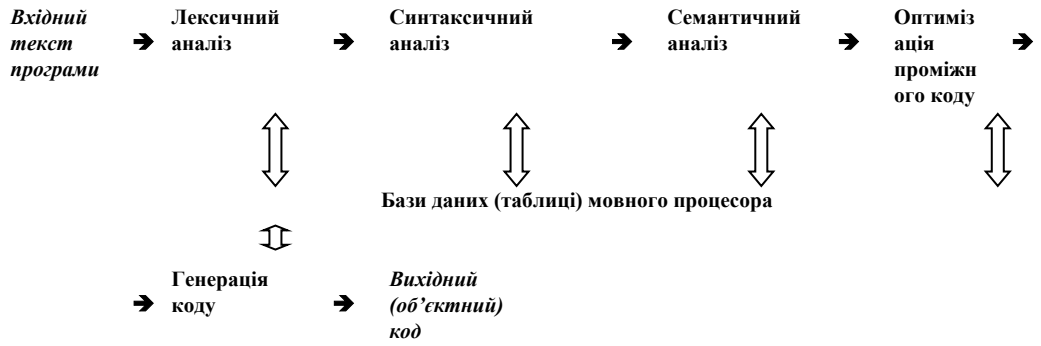


Рисунок 1. Блочна схема мовного процесора типу транслятор

Лексичний аналіз в мовних процесорах

Призначення: перетворення вхідного тексту програми з формату зовнішнього представлення в машинноорієнтований формат – послідовність лексем.

Лексема [2] – це ланцюжок літер (елементарний об’єкт програми), що несе певний семантичний зміст. В подальшому кожну лексему будемо представляти як пару

(<клас_лексеми, ім’я_лексеми>)

В більшості мов програмування для визначення класів лексем достатньо скінчених автоматів.

Скінчені автомати

Означення: Недетермінований скінчений автомат – це п’ятірка

$$M = < Q, \Sigma, \delta, q_0, F>,$$

де

- $Q = \{q_0, q_1, .. , q_{n-1}\}$ – скінчена множина станів автомата;
- $\Sigma = \{a_1, a_2, .., a_m\}$ – скінчена множина вхідних символів (вхідний алфавіт);
- $q_0 \in Q$ – початковий стан автомата;
- δ – відображення множини $Q^*\Sigma$ в множину $P(Q)$. Відображення δ як правило називають *функцією переходів*;
- $F \subseteq Q$ – множина заключних станів. Елементи з F називають *заключними* або *фінальними* станами.

Якщо M – скінчений автомат, то пара $(q, w) \in Q_*\Sigma^*$ називається *конфігурацією автомата M*. Оскільки скінчений автомат – це дискретний пристрій, він працює по тактам. Такт скінченого автомата M задається бінарним відношенням $|=$, яке визначається на конфігураціях:

$(q_1,aw) |= (q_2,w)$, якщо $\delta(q_1, a)$ містить q_2 та для всіх $w \in \Sigma^*$.

Означення. Скінчений автомат M розпізнає (допускає) ланцюжок w, якщо

$$(q_0, w) |=^* (q, \varepsilon) \text{ для деякого } q \in F,$$

де $|=^*$ – рефлексивно-транзитивне замикання бінарного відношення $|=$.

Означення. Мова, яку допускає автомат M (розпізнає автомат M)

$$L(M)=\{ w \mid w \in \Sigma^* \text{ та } (q_0, w) |=^* (q, \varepsilon) , \ q \in F \}.$$

На практиці, при визначенні скінченого автомата M, використовують декілька способів визначення функції δ, наприклад:

- це табличне визначення δ;
- діаграма проходів скінченого автомата.

З практичної точки зору, при дослідженні скінчених автоматів ми будемо розглядати лише скінчені автомати без ε-переходів, тобто не розглядатимемо значення функції $\delta(q_i, \varepsilon)$, $q_i \in Q$.

Табличне визначення функції δ – це таблиця $M(q_i,a_j)$, де $a_j \in \Sigma$, $q_i \in Q$, тобто

$$M(q_i,a_j) = \{ q_k \mid , q_k \in \delta(q_i,a_j) \}.$$

Діаграма переходів скінченого автомата M – це орієнтований граф G(V, P), де V – множина вершин графа, а P – множина орієнтованих дуг, причому з вершини q_i у вершину q_j веде дуга позначена a_k , коли $q_j \in \delta(q_i,a_k)$.

Побудуємо діаграму переходів скінченого автомата M, який розпізнає множину цілочислових констант мови C- (Рис. 2).

В подальшому, на діаграмі переходів скінченого автомата M елементи з множини заключних станів будемо позначити так: q_i.

З побудованого прикладу видно, що приведений автомат не повністю визначений, тобто існують пари (q_i, a_k) для яких $\delta(q_i, a_k)$ – невизначено.

Означення. Скінчений автомат M називається детермінованим, якщо $\delta(q_i, a_k)$ містить не більше одного стану для любого $q_i \in Q$ та $a_k \in \Sigma$.

Твердження: Для довільного недетермінованого скінченого автомата M можна побудувати еквівалентний йому детермінований скінчений автомат M_1, такий що

$$L(M) = L(M_1).$$

Доведення: Нехай M – недетермінований скінчений автомат, такий що

$$M=< Q, \Sigma, \delta, q_0, F>.$$

Детермінований автомат $M_1=< Q_1, \Sigma, \delta_1, q_{01}, F_1>$ побудуємо таким чином:

1. $Q_1 = P(Q)$, тобто імена станів автомата M1 – це підмножини множини Q.

2. $q_{01} = \{ q_0 \}$, $\{ q_0 \} \in P(Q)$.

3. F_1 складається з усіх таких підмножин $S \in P(Q)$, таких що $S \cap F \neq \varnothing$.

4. $\delta_1(S, a) = \{q \mid q \in \delta(q_i, a), q_i \in S \}$.

Доведемо індукцією по i, що $(S,w) \mid=^i (S_1,\varepsilon)$, тоді і тільки тоді, коли

$$S_1 = \{ q \mid (q_i, w)) \mid=^i (q, \varepsilon), \text{ для } q_i \in S \}.$$

Зокрема, $(\{q_0\}, w) \mid=^* (S_1, \varepsilon)$, для деякого $S_1 \in F_1$, тоді і тільки тоді, коли

$(q_0, w) \mid=^* (q, \varepsilon)$, $q \in F$. Таким чином, $L(M) = L(M_1)$.

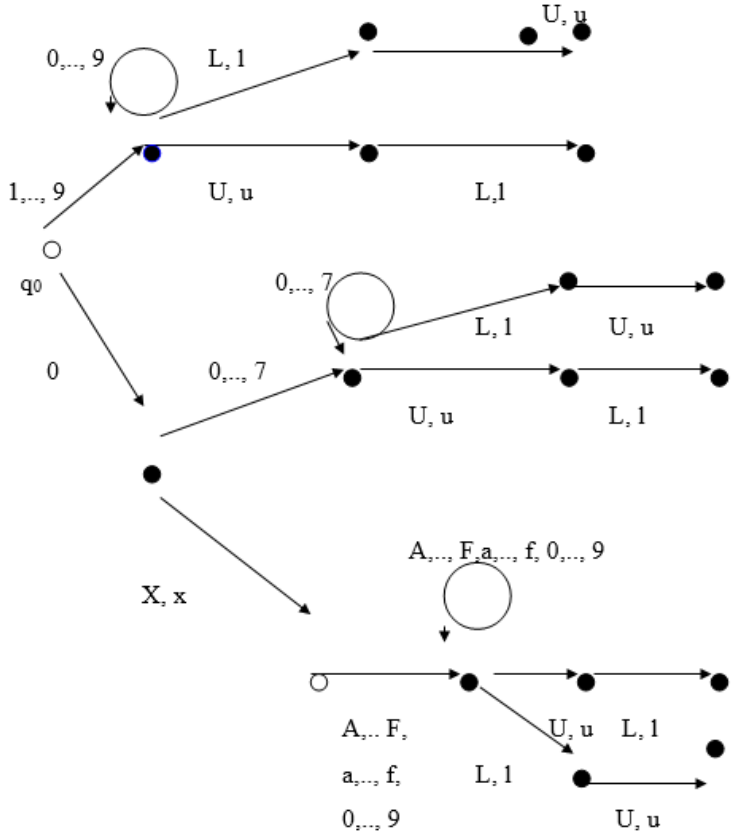


Рисунок 2. Діаграма переходів скінченого автомата

Побудований нами автомат M має дві властивості: він детермінований та повністю визначений. До того ж кількість станів цього автомата $2^n - 1$.

Приклад. Алгоритм. Побудова мінімального скінченого автомата.

П1. Побудувати скінчений автомат без тупикових станів.

П2. Побудувати скінчений автомат без недосяжних станів.

П3. Знайти множини еквівалентних станів та побудувати найменший (мінімальний) автомат.

Ознайомившись з деякими результатами теорії скінчених автоматів, спробуємо уяснити, які мови (словарні множини) є скінчено автоматними.

Твердження: Скінчено автоматними є наступні множини:

- порожня словарна множина – $\varnothing$;
- словарна множина, що складається з одного ε-слова – $\{\varepsilon\}$;
- множина $\{a\}$, $a \in \Sigma$.

Доведення: в кожному випадку нам доведеться конструктивно побудувати відповідний скінчений автомат:

1. Довільний скінчений автомат з пустою множиною заключних станів (а мінімальний – з пустою множиною станів) допускає $\varnothing$;

2. Розглянемо автомат $M = <\{q_0\}, \Sigma, q0, \delta, \{q0\}>$, у якому δ не визначено ні для яких $a \in \Sigma$.

Тоді $L(M) = \{\varepsilon\}$.

3. Розглянемо автомат $M = <\{q_0, q_1\}, \Sigma, q_0, \delta, \{q_1\}>$, у якому функція δ визначена лише для пари (q_0, a), а саме: $\delta(q_0, a) = q_1$.

Тоді $L(M) = \{a\}$.

Твердження: Якщо $M_1 = <Q_1, \Sigma, q_{01}, \delta_1, F_1>$ та $M_2 = <Q_2, \Sigma, q_{02}, \delta_2, F_2>$, що визначають відповідно мови $L(M_1)$ та $L(M_2)$, то скінчено автоматними мовами будуть:

- $L(M_1) \cup L(M_2)$;
- $L(M_1) * L(M_2)$;
- $\{L(M_1)\} = \{\varepsilon\} \cup L(M_1) \cup L(M_1)* L(M_1) \cup \ldots$,

де:

$L(M_1) \cup L(M_2) = \{ w \mid w \in L(M_1)$ або $w \in L(M_2) \}$,

$L(M_1) * L(M_2) = \{w=xy \mid x \in L(M_1),\ y \in L(M_2)\}$.

Доведення:

1. Побудуємо автомат $M = <Q, \Sigma, q_0, \delta, F>$, такий що $L(M) = L(M_1) \cup L(M_2)$.

- $Q = Q_1 \cup Q_2 \cup \{q_0\}$, де q_0 – новий стан ($q_0 \notin Q_1 \cup Q_2$);
- Функцію δ визначимо таким чином:

$$\delta(q,a) = \delta_1(q,a),\ q \in Q_1,\ a \in \Sigma;$$
$$\delta(q,a) = \delta_2(q,a),\ q \in Q_2,\ a \in \Sigma;$$
$$\delta(q_0,a) = \delta 1(q_{01},a) \cup \delta 2(q_{02},a),\ q_{01} \in Q_1,\ q_{02} \in Q_2,\ a \in \Sigma.$$

- Множина заключних станів

$$F = \begin{Bmatrix} F_1 \cup F_2, \text{якщо } \varepsilon \notin (L \cup L_2) \\ F_1 \cup F_2 \cup \{q_0\}, \text{ якщо } \varepsilon \in (L_1 \cup L_2) \end{Bmatrix}.$$

Побудований в цьому випадку автомат взагалі недетермінований. Індукцією по $i >= 1$ покажемо, що $(q_0,w) |=^i (q,\varepsilon)$ можливо тоді і тільки тоді, коли $(q_{01},w) |=^i (q,\varepsilon)$, $q \in F_1$ або $(q_{02},w) |=^i (q,\varepsilon)$, $q \in F_2$.

2. Побудуємо автомат $M = <Q, \Sigma, q_0, \delta, F>$, такий що $L(M) = L(M_1) * L(M_2)$:

- $Q = Q_1 \cup Q_2$;
- $q_0 = q_{01}$;
- Функцію δ визначимо таким чином:

$$\delta(q,a) = \delta_1(q,a),\ q \in Q_1 \backslash F_1,\ a \in \Sigma;$$
$$\delta(q,a) = \delta_2(q,a),\ q \in Q_2,\ a \in \Sigma;$$
$$\delta(q,a) = \delta_1(q,a) \cup \delta_2(q_{02},a),\ q \in F_1,\ q_{02} \in Q_2, a \in \Sigma;$$

- Множина заключних станів

$$F = \begin{Bmatrix} F_2, \text{якщо } \varepsilon \notin L_2 \\ F_1 \cup F_2, \text{ якщо } \varepsilon \in L_2 \end{Bmatrix}.$$

3. Побудуємо автомат $M = <Q, \Sigma, q_0, \delta, F>$, такий що $L(M) = \{L(M_1)\}$:

- $Q = Q_1 \cup \{q_0\}$, де q_0 – новий стан ($q_0 \notin Q1$);

- Функцію δ визначимо таким чином:

$$\delta(q,a) = \delta_1(q,a),\ q \in Q_1 \backslash F_1,\ a \in \Sigma;$$
$$\delta(q_0,a) = \delta_1(q_{01},a)\ ,\ q_{01} \in Q_1,\ a \in \Sigma;$$
$$\delta(q,a) = \delta_1(q,a) \cup \delta_1(q_{01},a),\ q \in F_1,\ a \in \Sigma;$$

- Множина заключних станів $F = F_1 \cup \{q_0\}$.

Формальне визначення класу лексем мови програмування можна виконати одним з нижченаведених способів:

– За допомогою праволінійних граматик;

– За допомогою скінчених автоматів;

– За допомогою регулярних виразів;

– Перерахувати лексеми даного класу як скінчену множину елементів.

Перші три способи визначення класів лексем за своєю потужністю еквівалентні. Якщо деякий клас лексем мови програмування скінчена множина, то одним з тривіальних способів визначення лексем цього класу є їх перерахування. Наприклад, клас однолітерних кодів операцій мови програмування *C* можна визначити як скінчену множину

$$L_0 = \{ +, -, /, *, \ldots.\}.$$

Сформулюємо фундаментальне твердження теорії граматик та автоматів: клас мов, які розпізнаються скінченими автоматами, співпадає з класом мов визначених праволінійними граматиками та регулярними виразами та навпаки. Відмітимо, що аналіз досвіду використання перерахованих засобів визначення класів лексем показує, що скінчені автомати знайшли широке використання при розробці лексичних аналізаторів для конкретних мов програмування, а регулярні вирази та праволінійні граматики широко використовуються в системах автоматизації побудови мовних процесорів як засоби високого рівня денотативності опису класів лексем [1].

Розглянемо можливі варіанти використання різних підходів визначення класів лексем мов програмування та їх реалізацію в мовних процесорах. Продемонструємо два підходи розробки програмних модулів,

які розпізнають множину ланцюжків (лексем), що допускає скінчений автомат, діаграма переходів котрого зображена на Рис. 1.

Скінчений автомат має:

- множину станів $Q=\{q_0, q_1, q_2, q_3, q_4, q_5, .., q_{17}\}$;
- вхідний алфавіт Σ={0,1,..9, A, B,..F, a, b,..f, X, x, L, l, U, u};
- початковий стан q_0;
- множину заключних станів $F= Q\setminus\{ q_0, q_{12}\}$;
- відображення δ визначено діаграмою переходів.

Програму, яка розпізнає множину лексем, можна реалізувати двома способами:

- *перший спосіб*: створимо програмний модуль, роботою котрого керує таблиця управління скінченого автомата $M(q_i, a_j)$, яка визначається в програмі явно;
- *другий спосіб*: розробимо програмний модуль з управлінням за номером поточного стану скінченого автомата та поточною вхідною літерою;
- *третій спосіб*: скористатися інструментальною системою автоматизації розробки мовних процесорів.

Очевидною перевагою програмної реалізації скінчених автоматів першим способом є простота визначення інформаційних масивів даних: alphabet – вхідний алфавіт автомата, sigma – таблиця переходів автомата, finish – масив імен заключних станів. Серед недоліків реалізації наведеної вище програми яскраво виділяється один – досить великі затрати пам'яті для таблиці sigma. Очевидно, що матриця sigma сильно розріджена, тобто більшість її елементів має значення ERROR. В такому випадку при реалізації скінчених автоматів, кількість станів яких вимірюється десятками, та як вхідний алфавіт розглядається вся кодова таблиця ЕОМ (255 символів), то затрати оперативної пам'яті будуть занадто великі.

На сьогодні багато компаній розробляють використовують інструментальні програмні системи, які забезпечують автоматизацію побудови програмних систем. Такий підхід має ряд суттєвих переваг:

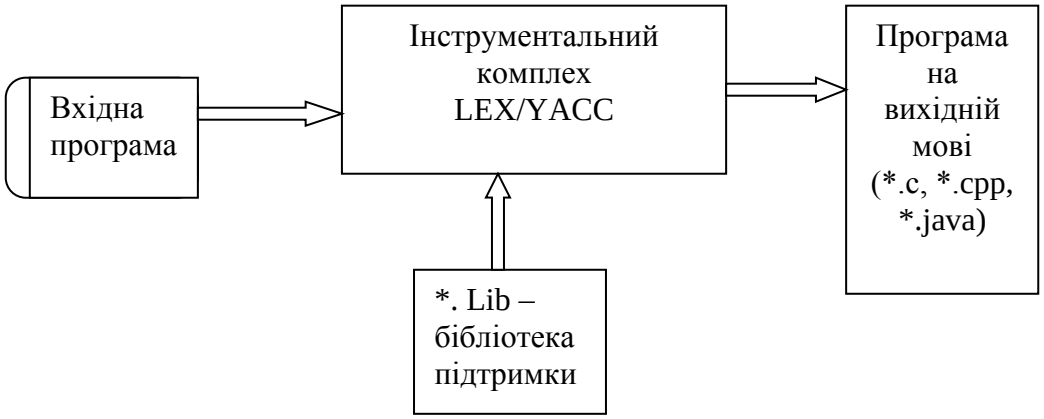
- в основу інструментальної системи покладають теоретичні дослідження відповідної предметної області;

- теоретичне обгрунтування алгоритмів забезпесує коректність результату;
- є можливість масово примінити інструментальний комплекс для вирішення поставлених задач не використовуючи багато часу на реалізацію проекту.

Спеціалісти з розробки мовних процесорів пішли аналогічним шляхом. Було розроблено ряд інструментальних систем автоматизації побудови мовних процесорів, зокрема LEX/YACC, FLEX, BIZON, JCC.

Складовими цих інструментальних систем є:

- мова для опису лексичної, синтаксичної та семантичної компоненти мови програмування;
- віртуальна машина, яка маніпулює (обробляє) опис компонент мови програмування та генерує відповідний програмний модуль на вихідній мові – HOST-мові;
- шаблони для подальшої побудови модулів (компонент) мовного процесора;
- бібліотеки інструментального комплексу.



Вхідна програма – описує на вхідній мові інструментального пакету компоненти мовного процесора. Вхідна програма, як правило, складається з розділів, які відділяються один від одного спецсимволами, наприклад ‘%%’.

Як приклад, наведемо фрагмент програми на вхідній мові інструментального комплексу LEX:

```
/***************** Секція  означень ****************/
/*<ідентифікатор> <регулярний вираз> */
NODELIM     [^" "\t\n]
/*********** Секція правил ********************
<регулярний вираз> { /* код на HOST-мові */ }
%%
{NODELIM}+  { w++; c+=yyleng;  /* Слово */  }
\n          { l++;                }
.           { c++;                }
%%
/****************** Секція програм ***************/
int main() {
        int l=w=c=0;
        yylex(); return 0; }

int yywrap() { /* Викликається при досягненні кінця вхідного файла
*/ };
printf("Lines - %d Words - %d Chars - %d\n", l, w, c );
return ;
}
/*********** Кінець програми wc.lex ****************/
```

Віртуальна машина :

- використовує секцію скорочень по принципу #define у мові програмування С.
- на основі кожного виразу секції правил будує скінчений автомат, який розпізнає множину слів, що позначаються регулярним виразом;
- об’єднує побудовані автомати, детермінує та мінімізує отриманий мінімальний автомат. Для фінальних станів побудованих раніше автоматів генерує код на HOST-мові.
- отриманий результат поміщає у шаблон лексичного аналізатора.
- у кінець шаблону лексичного аналізатора дописує код з розділу «секція підпрограм».

Результат роботи інструментального комплексу LEX – програмний модуль (текстовий файл) згенеровагого лексичного аналізатора.

Ми розглянули теоретичне підгрунття та практичні підходи щодо розробки лише одного блоку мовного процесора. Зазвичай, розробка решти модулів вимагає теоретичних знань та практичного досвіду у наступних напрямках досліджень:

- розробка синтаксичного аналізатора з оцінкою у часі *O(n)*, де *n* – довжина вхідної програми. Тут розглядаються підкласи контестно-вільних граматик, як-то LL(k) – та LR(1) – граматики, які забезпечують аналіз вхідної програми за час пропорційний *O(n).* Як інструменти програмної реалізації розглядаються магазинні автомати побудовані на основі вище означених класів граматик.
- розробка семантичного аналізатора, який взаємодіє з синтаксичним аналізатором та забезпечує побудову семантичного терма програми.
- Розробка генератора вихідного коду - об’єктного коду ЕОМ або коду віртуальної машини.

Список використаних джерел

1. Теоретичні основи програмування: [навчальний посібник] / М. С. Нікітченко. – Ніжин: Видавництво НДУ імені Миколи Гоголя, 2010. – 121 с.
2. Волохов В.М. Методичні рекомендації до лабораторного практикуму побудови мовних процесорів з дисципліни «Системне програмування» – Київ: 2013. – 53 с.
3. PascalABC.NET [Електронний ресурс] – Режим доступу до ресурсу: http://pascalabc.net/.
4. Документація Microsoft C++ [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/cpp/?view=msvc-170.
5. JAVA [Електронний ресурс] – Режим доступу до ресурсу: https://www.java.com/ru/.
6. The SNOBOL 4 programming language [Електронний ресурс] – Режим доступу до ресурсу: https://www.snobol4.org/docs/books.html.
7. Common Lisp Books [Електронний ресурс] – Режим доступу до ресурсу: https://lisp-lang.org/books/.
8. Мови програмування [Електронний ресурс] – Режим доступу до ресурсу: https://csc-knu.github.io/sys-prog/lectures/md/01.html.

Кузенко Володимир Федорович, к.ф.-м.н, доцент

ПАРАДИГМИ ПРОГРАМУВАННЯ ТА ОСОБЛИВОСТІ МАНІПУЛЮВАННЯ ДАНИМИ

Досліджується можливість уточнення та класифікації змінних в імперативних програмах, базуючись на понятті іменованих елементів та виокремленні двох різних рівнів абстракції. Функціональні програми порівняно з імперативними часто виглядають значно простішими. Змінні можуть не використовуються взагалі, а компонування (програм як функцій) може забезпечуватись відомою з шкільної математики суперпозицію функцій. Проте ситуація ускладнюється для функцій не всюди визначених чи з так званими побічними ефектами. Відповідні засоби компонування досліджуються, використовуючи методичне зіставлення монадних операцій зі звичайними. Виявляються моменти взаємопроникнення імперативного та функціонального стилів.

Ключові слова: програмні змінні, програмні функцій, композиції, аплікації, монади.

The possibility of specifying and classifying variables in imperative programs is investigated, based on the concept of named elements and the separation of two different levels of abstraction. Functional programs often look much simpler than imperative ones. Variables may not be used at all, and the composition (of programs as functions) can be provided by a superposition of functions known from school mathematics. However, the situation is complicated by functions that are not defined everywhere or with so-called side effects. Appropriate composition tools are investigated using a methodical comparison of monad operations with usual ones. There are moments of interpenetration of imperative and functional styles.

Keywords: program variables, program functions, compositions, applications, monads.

Обробка різноманітних даних є квінтесенцією програмування. Тому питання подання та використання даних відіграє особливо важливу роль незалежно від застосування тієї чи іншої парадигми чи мови програмування.

До того ж варто додати, що стрімкий розвиток комп'ютеризації супроводжується постійним зростанням зацікавленості не тільки до можливостей програм з точки зору користувача, але й до розуміння того, як програми розробляти. Сьогодні вже не підлягає сумніву доцільність впровадження у шкільну програму предмету «Інформатика». Школярі успішно програмують, деякі з них пропонують цікаві, практично важливі і конкурентноспроможні програмні продукти, ґрунтуючись по суті лише на інтуїтивному розумінні маніпулювання даними як даними імперативними, що можуть змінюватись. При цьому поряд з інтуїтивним розумінням функцій та їх обчислень найчастіше залучаються до розгляду так звані програмні змінні та присвоювання значень таким змінним.

Незважаючи на прагматичну корисність такого підходу для набуття програмістських навичок, особливо для обчислювальних задач, він не тільки потребує подальших уточнень, але й може виявитись не завжди придатним з огляду на різноманіття як програмістських задач, так і наявних парадигм та стилів програмування. Навіть у студентів старший курсів нерідко виникають принципові труднощі при вивченні особливостей існуючих парадигм програмування, що лише підкреслює актуальність експлікативного програмування [1].

До експлікації змінних в імперативному програмуванні

Запропонований В.Н.Редьком підхід [2] розглядати програмні дані як іменні дані, а програми як іменні функції надав поштовх не тільки у дослідженні різноманітних програмних алгебр, але й дозволив на основі іменних даних найбільш повно і природно надати експлікацію таких ключових понять імперативного програмування як програмні змінні та константи [3,4] .

Нагадаємо означення іменних даних (від запропонованого в [2] воно суттєво не відрізняється).

Нехай V – деяка множина (абстрактних) імен, S – деяка множина (базових) значень. Тоді множина іменних даних $D_{V,S}$ є найменшою (щодо включення) множиною серед таких множин D, які задовольняють наступним трьом умовам:

1) $V \cup S \subset D$; $\varnothing \in D$ ($\varnothing$ – порожня множина);

2) нехай $v \in V$ і $d \in D \setminus (V \setminus S)$, але $d \neq \varnothing$. Тоді (впорядкована) пара (v,d) належить множині D в тому і тільки в тому випадку, коли ім'я v не використовується в d. Про таку пару $(v,d) \in D$ надалі будемо говорити як про *іменований елемент*, вважаючи при цьому, що (абстрактне) ім'я v *іменує* (денотат) d, тобто, що між v і d встановлене *відношення іменування*;

3) нехай $d_1 \in D$, $d_2 \in D$, ... , $d_n \in D$ (n – натуральне число). Тоді множина $M=\{d_1, d_2, ..., d_n\}$ буде належати D в тому і тільки в тому випадку, коли в M не використовуються одні і ті ж самі імена для іменування різних денотатів. (У композиційному програмуванні така вимога щодо використання імен для іменування денотатів складає один з основних принципів – принцип іменування).

Уведене "попутно" поняття іменованого елемента дозволяє підійти до уточнення програмних змінних, розглядаючи іменований елемент і програмну змінну як спряжені між собою об'єкти такі, що відповідають різним рівням абстракції. Програмна змінна відповідає більш низькому рівню абстракції або ж, відповідно, більш високому рівню конкретизації порівняно з іменованим елементом (v,d), а саме коли у тексті програми для (абстрактного) імені v доводиться фіксувати деяке *синтаксичне позначення* (як синтаксичний вираз). Таке синтаксичне позначення виступає *іменем програмної змінної* (рис. 1). Іншими словами, "перехід" від іменованих елементів до програмних змінних фактично пов'язується із долученням до розгляду поряд із семантичним аспектом ще й синтаксичного.

Підкреслимо наявність деякої паралелі між такою концепцією програмних змінних та пропозицією у відомому теоретичному, історично узагальнюючому мовному проекті Алгол-68 залучати до розгляду разом з іменами та значеннями програмних змінних ще й так звані дескриптори змінних – своєрідні аналоги абстрактних імен.

Зазначимо також, що проблема реалізації мов програмування по суті потребує подальшої конкретизації абстрактних імен (дескрипторів). Найчастіше абстрактні імена чи дескриптори пов’язують із так званими адресами (пам’яті), говорячи про абстрактні чи реальні адреси “комірок пам’яті ” для збереження значень змінних.

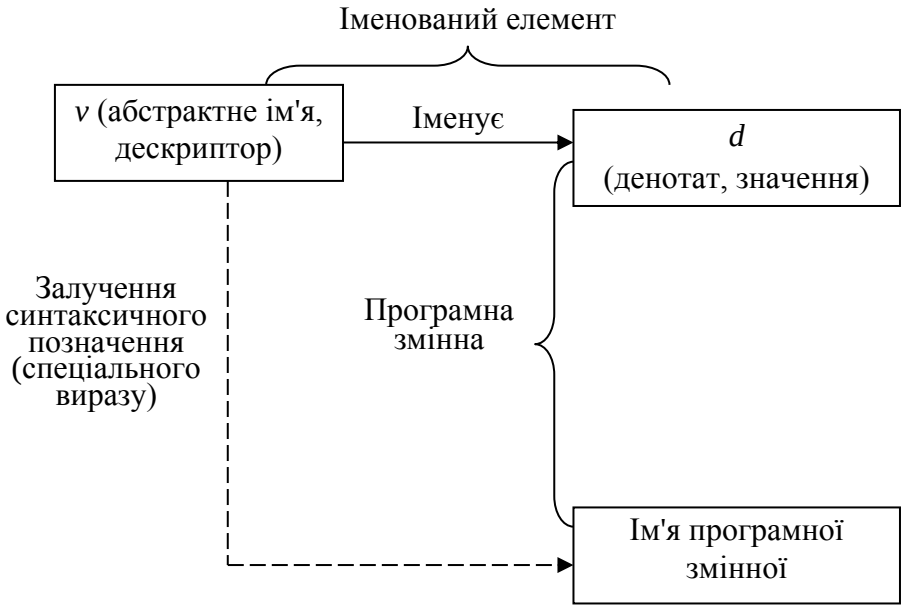


Рисунок 1. Іменований елемент та програмна змінна

Зазначимо також, що проблема реалізації мов програмування по суті потребує подальшої конкретизації абстрактних імен (дескрипторів). Найчастіше абстрактні імена чи дескриптори пов’язують із так званими адресами (пам’яті), говорячи про абстрактні чи реальні адреси “комірок пам’яті” для збереження значень змінних.

Зображенням на рис. 1 суцільної стрілки, що веде від блока з абстрактним іменем *v* до блока з денотатом *d*, підкреслюється наявність (у відповідності із принципом іменування) між іменами та їхніми денотатами функціональної залежності, яка забезпечує можливість "доступу" до денотатів за заданими іменами. При побудові програмних алгебр функцію, що забезпечує обчислення за абстрактним іменем його денотату (її найчастіше називають розіменуванням і позначають через PIM) природно вибирати як одну з базових. Формально функцію розіменування PIM можна подати у вигляді PIM(*x*, *D*), де аргумент *x* – це (абстрактне) ім'я, а *D* – іменне дане, саме із якого за іменем *x* вибирається іменований цим іменем денотат.

Абстрактні імена, використання яких покликане "втягувати" у програмні обчислення відповідні цим іменам денотати, можуть у деяких випадках і самі обчислюватися, тобто під час програмних обчислень абстрактні імена можуть "вироблятися" деякими функціями. Більш того, класифікація абстрактних імен, що використовуються в програмі, на необчислювані та обчислювані відповідає традиційній класифікації програмних змінних на прості та складені (або складні).

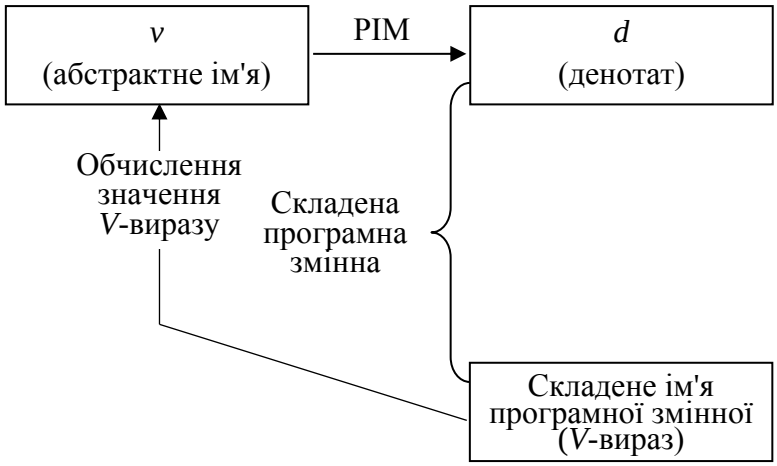


Рисунок 2. Складена програмна змінна та обчислюваність абстрактих імен

Відповідні складені імена природно розглядати як спеціальні *вирази*, значеннями яких (при програмних обчисленнях) є абстрактні імена (рис. 2). Наприклад, у [5] для таких виразів запропоновано термін *V*-вирази. Доречно нагадати також про третю компоненту (поряд з іменем та денотатом) з відомого трикутника Фреге – *зміст імені* і про те, що "можливо розуміти зміст імені, нічого не знаючи про його денотат" [6]. Наведемо кілька ілюструючих згадане поняття прикладів: “друга зліва книга на третій поличці знизу”, “прем’єр-міністр України у лютому 2021 року”, “рік народження нинішнього президента України”. Саме зміст імені може подаватись із використанням згаданих вище виразів (*V*-виразів), і у

програмування важливо забезпечити можливість проведення обчислень у відповідності до структури таких виразів.

Зауважимо також, що обчислюваність абстрактних імен є істотною обставиною для експлікації маніпулювання елементами масивів, динамічними змінними, посиланнями, окремими компонентами різноманітних структурованих даних тощо.

У випадку необчислюваних абстрактних імен природним є вибір синтаксичних позначень постійними й унікальними, а отже матимемо змінні з простими іменами або ж прості змінні.

Формально, у випадку простої змінної (рис. 3) можна ототожнювати абстрактне ім’я, у даному прикладі *v*, з обраним для нього синтаксичним позначенням, у даному прикладі *id*. Те, що позначення вибирається постійним та унікальним, гарантує непорушність принципу іменування, і, отже, є підставою введення спеціального варіанту розіменування – функції $РІМП_{Id}(D)$ [4], яка забезпечує обчислення денотатів за іменами простих змінних, як кажуть, “напряму”.

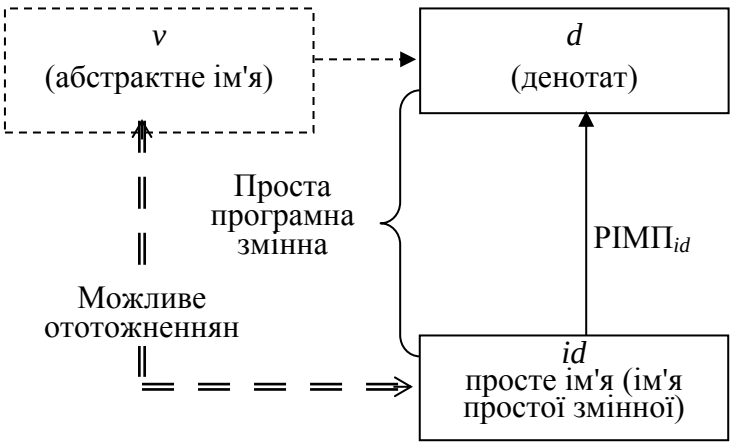


Рисунок 3. Проста програмна змінна

Більш повному і докладному дослідженню імперативних засобів у мовах програмування високого рівня, ролі розіменувань РІМ та РІМП присвячені роботи [3,4]. Зокрема, в [4] аргументується узгодженість запропонованих уточнень програмних констант і змінних із трактуванням констант і змінних, яке є загальноприйнятим у математиці. (Нагадаємо, що в [6] щодо змінних, на відміну від констант, лише припускається можливість іменем іменувати не обов'язково один і той самий денотат).

Зауважимо також, що математиці притаманне абстрагування від способів отримання денотатів за іменами змінних. Цей підхід часто застосовується і у програмуванні. Так, наприклад, традиційні програмні вирази (арифметичні, логічні тощо) виглядають як звичайні математичні формули, хіба що з урахуванням потреби подання виразів у лінійній нотації. Більш того, спостерігається тенденція поширення такого підходу. Яскравим прикладом цього може слугувати відмова використовувати у виразах операцію (*розіменування*!) "^" при переході від (*спеціально створеної для університетської освіти*!) мови *Pascal* до реалізації *Pascal* під назвою "нова об'єктна модель *Pascal*".

Проте приховування розіменувань разом із позитивним моментом, а саме можливістю для пояснення семантики виразів спиратись на традиційні обчислення за формулами, має й негативний – завуальовуються певні семантичні залежності між операторами і виразами. Дійсно, при прихованих розіменуваннях вирази синтаксично виступають як складові частини операторів. Цю підпорядкованість іноді абсолютизують, переносячи її на семантичний рівень. Саме з цих позицій можна, наприклад, оцінювати пропозиції відокремити область виразів від області операторів і вивчати ці області ізольовано [7]. Конкретизація ж обчислень денотатів за їх іменами спряжена із виявленням *залежності виразів від операторів*, точніше від іменних даних, що виробляються операторами. Саме при уточненні розіменування у вигляді функцій РІМ(x,D) чи РІМП$_{Id}$(D) така залежність чітко простежується з огляду на наявність аргумента (іменного даного) D.У багатьох же підходах до уточнення розіменувань ця обставина залишається поза увагою.

Функціональна парадигма. Обчислення та компонування функцій

Функціональному програмуванню і, зокрема, мові *Haskell*, яка вважається стандартом для відповідної парадигми, притаманні засоби, що є не тільки простими за формою, але й вбачаються звичними та зрозумілими.

Особливо це стосується так званих запитів як формальних виразів, призначених для обчислень власне комп'ютером. Так, вирази на зразок *f*(5) (у Haskell, до речі, допускається й більш коротка бездужкова форма *f* 5) або ж *g*(*f*(5)) не тільки узгоджуються зі звичною ще з шкільної математики нотацією, але й можуть тлумачитись, ґрунтуючись на звичних зі школи поняттях: понятті *функції* та понятті *суперпозиції* функцій. При тому часто підкреслюють, що на відміну від імперативного функціональний стиль програмування *не потребує використання змінних* і, як наслідок, дозволяє позбутись багатьох проблем, притаманних імперативному стилю [7].

Не набагато складнішими можуть у *Haskell* виглядати і визначення функцій. Повністю зрозумілими є визначення із використанням рівнянь на зразок

$$my\_inc\ x = x+1.$$

Однак можливість використовувати рівняння, що мають так званий рекурсивний вигляд, піднімає цілу низку питань, отримання відповідей на які спряжене із серйозними математичними дослідженнями [8, 9]. Зокрема, це стосується питань існування розв'язку, його єдиності та власне пошуку такого розв'язку для того чи іншого рівняння.

Знову повертаючись до запитів, варто усвідомлювати, що їх простота і зрозумілість декларуються щодо функцій, які вважаються *всюди визначеними та без побічних ефектів*. Ситуація кардинально ускладнюється при потребі використовувати і компонувати функції, які не задовольняють цим обмеженням. У такому випадку доводиться залучати монади (клас типів Monad) і спряжені з ними поняття монадних значень, монадних функцій тощо. При тому компонування монадних функцій із використанням суперпозиції виявляється неможливим з огляду на вимоги щодо сумісності типів, а компонування із використанням прописаної у класі типів *Monad* операції *bind* (">> =") виглядає занадто незвичним і важко сприймається.

Однак є можливість провести паралель між компонуванням функцій звичайних, тобто всюди визначених і без побічних ефектів, та компонуванням функцій монадних. Така можливість з'являється, якщо при компонуванні функцій звичайних відштовхуватись не від суперпозиції, а від операцій *композиції* (зазвичай пропонуються два варіанти композиції, для яких використовуються позначення "." і ">.>") та *аплікації* (для аплікації пропонуються також два варіанти: "$" і "> $ >"), означення яких ґрунтується на співвідношеннях:

$$(g.f)(x) = g(f\ x) = (f >.> g)(x)\ ; \qquad (1)$$

$$f\ \$\ x = f\ x = x >\$> f\ . \qquad (2)$$

Зауважимо, по-перше, що з (1) випливає, що обидва варіанти композиції (".") та ">.>") пов'язані із суперпозицією функцій і, по-друге, що у *Haskell*-програмах частіше можна зустріти композицію "." і аплікацію "*$*". При тому використання аплікації *$* часто спряжене з намаганням елімінувати деякі пари дужок у випадку складних виразів.

Повертаючись до порівняння засобів компонування функцій звичайних і функцій монадних, можна зазначити, що звичайній *композиції* (точніше, варіанту композиції ">.>") ставиться у відповідність *монадна композиція* "> = >". Важливо при тому зауважити, що наявність для звичайної композиції таких властивостей як асоціативність та існування одиниці дозволяє не тільки усвідомити мотиви уведення так званих *монадних законів*, але й забезпечити їх прозорість:

1) $return >=> f = f$;
2) $f >=> return = f$;
3) $(f >=> g) >=> h = f >=> (g >=> h)$.

Дана версія монадних законів (версія в термінах монадної композиції "> = >") підтверджує природність зіставлення композиції ">.>" з композицією монадною">=>" та дає зрозуміти, чому її називають *приємною*

версією на відміну від наступної версії законів “прямої дії” в термінах операції *bind* (>> =) з класу типів *Monad*:

1) *return x >> = f = f x*;
2) *mv >> = return = mv*;
3) (*mv >> = f*) *>> = g = mv >> =* (*x -> (f x >> = g*)).

Монадна композиція “> = >” визначається через bind (>> =) наступним чином:

$$fm > = > gm = \backslash x \text{ -> } ((fm\ x) >> = gm),$$

або

$$(fm > = > gm)\ x = (fm\ x) >> = gm \qquad (3)$$

(тут *fm* і *gm* – функції монадні).

Проаналізуємо операцію *bind* (>> =) з огляду на природність зіставлення композиції монадної “> = >” з композицію звичайною “>.>”.

Нехай *f* і *g* – функції звичайні, тоді з (1),(2) можна отримати:

$$(f >.> g)\ x = (g.f)\ x = g(f\ x) = (f\ x) > \$ > g. \qquad (4)$$

Із зіставлення (3) та (4) випливає, що операції (>> =) над монадними функціями можна поставити у відповідність операцію аплікації (варіант > *$* >), тобто, іншими словами, операцію >> = можна було б назвати *монадною аплікацією*. Наведемо для порівняння між собою типи аплікації (> *$* >) та bind (>> =) відповідно:

$$(> \$ >) :: b \text{ -> } (b \text{ -> } c) \text{ -> } c\ ;$$

$$(>>=) :: m\ b \text{ -> } (b \text{ -> } m\ c) \text{ -> } m\ c\ .$$

Розглянемо тепер чи не найважливіший варіант компонування монадних функцій – варіант, пов'язаний із конвеєрними (послідовними) обчисленнями на зразок:

$$hm = fm\ 1 > = > fm\ 2 > = >\ \text{fm}\ 3,$$

або

$$hm\ x = fm\ 1\ x >> = fm\ 2 >> = fm\ 3.$$

У такому випадку можна скористатись так званою *do-нотацією* (як спеціальним синтаксичним цукром *Haskell*):

$$\begin{aligned} hm\ x &= do \\ y &<- fm\ 1\ x \\ z &<- fm\ 2\ y \\ &fm\ 3\ z\ . \end{aligned} \qquad (5)$$

Як можна бачити, у do-нотації по суті використовується стиль, характерний *імперативному програмуванню*. Але важливо при тому усвідомлювати, що для do-нотацій існують формальні правила перетворення (трансляції) такого типу визначень у звичайну (“безцукрову”) форму. Застосовуючи, зокрема, ці правила до (5), можна отримане наступне рівняння для функції hm :

$$hm\ x = fm\ 1\ \text{x} >> = (\backslash y\ ->\ (fm\ 2\ y >> = (\backslash \mathbf{z}\ ->\ (fmz\ z))))$$

або, враховуючи пріоритети операцій,

$$hm\ x = fm\ 1\ \underline{x} >> = \backslash y\ ->\ fm\ 2\ y >> = \backslash z\ ->\ fmz\ z.$$

Висновки. Досліджено використання, можливість уточнення, класифікацію програмних змінних, основні засоби компонування програм

стосовно імперативного та функцінального стилів програмування та виявлено окремі моменти взаємопроникнення цих стилів.

Список використаних джерел

1. Редько В.Н. Экспликативное программирование: ретроспективы и перспективы // Тр. 1-й Междун.научн.-практ. конф. по программированию, К. – 1998. – С. 3-24.
2. Редько В.Н. Семантические структуры программ // Программирование. – 1981. – 1. С. 3-19.
3. Кузенко В.Ф. Отношения именования и программные алгебры языков высокого уровня // Программирование. – 1994. – 2. – С. 79-84.
4. Кузенко В.Ф. Про імперативні засоби в мовах програмування високого рівня // ВісникКиївського університету. Серія: фізико-математичні науки, 2002. – №3. – С. 207-214.
5. Агафонов В.Н. О семантике переменных в паскалеобразных языках // Теоретические основы компиляции. – Новосибирск. – 1980. – С.17-23.
6. Черч А. Введение в математическую логику. – Москва: Изд-во иностр. литературы, 1960. – 484 с.
7. Backus J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs // Comm. ACM.– 1978. – v.21. – 8. – p.613-641.
8. БарендрегтХ. Ламбда-исчисление. Его синтаксис и семантика. – Москва: Мир, 1985. – 606 с.
9. Филд А., Харрисон П. Функциональное программирование = Functional Programming. – Москва: Мир, 1993. – 637 с.

Кузенко Володимир Федорович, к.ф.-м.н, доцент

ТЕХНОЛОГІЇ ВЕБ-ПРОГРАМУВАННЯ НА ПЛАТФОРМІ JAVA ТА ВИКОРИСТАННЯ ФРЕЙМВОРКІВ

Визначаються основні архітектурні засади та надаються основні фрагменти програмної реалізації web-фреймворку Miniature. Відповідний проєкт може розглядатись як патерн для Java web-фреймворків, що грунтуються на використанні архітектурного шаблону Model-View-Controller (MVC), та використовуватись у навчальному процесі при вивченні тематики, повязаної із веб-програмуванням на платформі Java. Відштовхуючись від запропонованого проєкту, проводиться порівняльний аналіз з деякими промисловими потужними web-фреймворками, і такий аналіз забезпечує можливість більш швидкої адаптації до їх практичного використання.

Ключові слова: Java-сервлет, web-проєкт, web-фреймворк, архітектура Model-View-Controller.

The basic architectural principles are defined and the main fragments of the software implementation of the Miniature web-framework are provided. The project can be considered as a pattern for Java web-frameworks based on the use of the architectural template Model-View-Controller (MVC), and used in the educational process in the study of topics related to web programming on the Java platform. Based on the proposed project, a comparative analysis is carried out with some powerful industrial web-frameworks, and such analysis provides the possibility of faster adaptation to their practical use.

Keywords: Java-servlet, web project, web framework, Model-View-Controller architecture.

Web-проект може розглядатись як *мовний процесор*, що отримує на вході рядок-запит, структура якого визначається специфікою *web* (це, найчастіше, *http-get* чи *http-post* запити), та виробляє результат, який по суті також може розглядатись як послідовність символів – *html*-код *web*-сторінки, яка відображатиметься у браузері користувача. Врахування потреби відображати отримувану відповідь у *web*-браузері обумовлює популярність у *web*-проектах саме *MVC*-архітектури, причому з орієнтацією

на використання в якості *View* як безпосередньо *html*-сторінок, так і "збагачених" *html*-сторінок, які потребують додаткового процесування в *html*-код. Такими "збагаченнями" найчастіше виступають сторінки *ASPX, Razor* (обидва типи сторінок з "двигунами" від *Microsoft*), *JSP, Velocity, FreeMarker* та ін.

Загалом під фреймворком для деякого *класу задач* стосовно розробки програмних систем (ПС) розуміють "шаблонний" програмний проект або, простіше, *програму-шаблон*, яка, по-перше, є потенційно готовою до запуску на виконання (завдяки здійсненій у ній реалізації *інваріантного ядра*, що відповідає обумовленому класу ПС) та, по-друге, дозволяє для кожної конкретної задачі (з обумовленого класу) отримувати рішення (як готову програмну систему) шляхом *налаштування* шаблонного проекту та можливого додавання деяких "архітектурно узгоджених" модулів.

У цій роботі переслідуються дві основні цілі:

- викласти основні засади реалізації запропонованого автором *web*-фреймворку *Miniature*, який можна розглядати не тільки як ще один приклад *web*-фреймворку, але і як *патерн* по відношенню до багатьох відомих *Java MVC web*-фреймворків*;*
- продемонструвати "патерновість" фреймворку *Miniature* на підставі аналізу конкретних популярних промислових *Java MVC web*-фреймворків та порівняння їх архітектури з архітектурою фреймворку *Miniature*.

Насамперед зробимо кілька зауважень.

По-перше, зазначимо, що *Miniature* є *web*-фреймворком, тобто цим окреслюється його цільове спрямування на розробку виключно *web*-проектів. (Саме *web*-проекти складають той клас задач-проектів, на який орієнтований *Miniature* як фреймворк).

По-друге, зауважимо, що архітектурним підґрунтям *Miniature* є відомий *MVC*-шаблон.

Третє зауваження стосується обраної платформи – платформи *Java*. Такий вибір не тільки дозволяє в якості *View* стосовно архітектурного шаблону *MVC* використовувати *JSP* як надзвичайно потужний і популярний

технологічний варіант "розширених" *HTML*-сторінок, а й гармонійно поєднати такі сторінки ще з однією ефективною у *web*-проектуванні технологією – технологією *JavaServlet*. Саме використання зазначених технологічних засад, а також технічніх прийомів з [1] дозволило реалізації запропонованого фреймворку надати прозорого та «*мініатюрного*» вигляду.

Web-фреймворком *Miniature* забезпечується реалізація наступних двох послідовно виконуваних кроків (функцій):

1) аналіз вхідного *web*-запиту (найпростіше такий запит можна тлумачити як запит деякого *web*-ресурсу);
2) диспетчеризація запиту – передача отриманого запиту одному з об'єктів-обробників.

Саме на основі цієї функціональності визначається *інваріантне ядро* фреймворку *Miniature*.

Деталізуючи диспетчеризацію, варто додатково відзначити, що фреймворк "бере на себе" як створення необхідних об'єктів-обробників, так і їх "запуск" шляхом виклику наперед обумовленого методу *execute*. Така обумовленість спряжена з вимогою "архітектурної узгодженості" класів-обробників, а саме з вимогою забезпечувати у класах-обробниках реалізацію фіксованого інтерфейсу *Action* з єдиним методом *execute*.

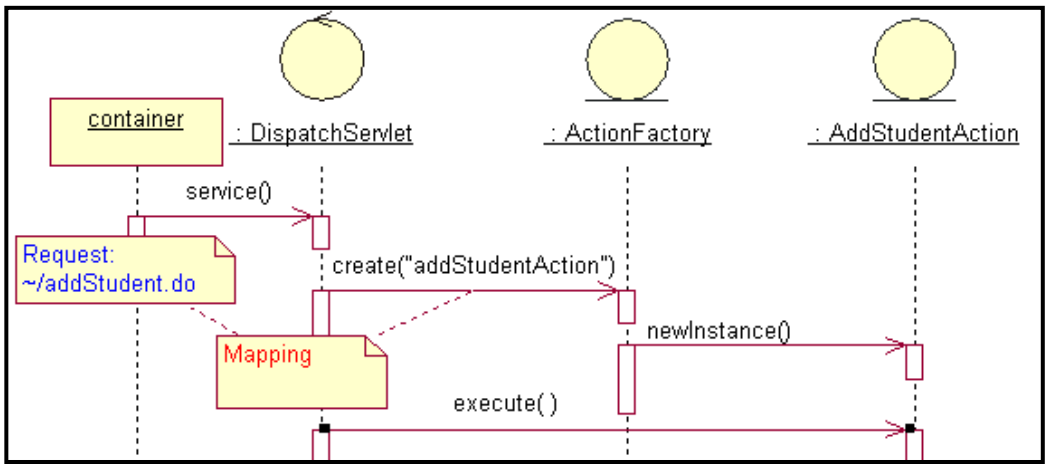


Рисунок 1. Обробка запиту "addStudent"

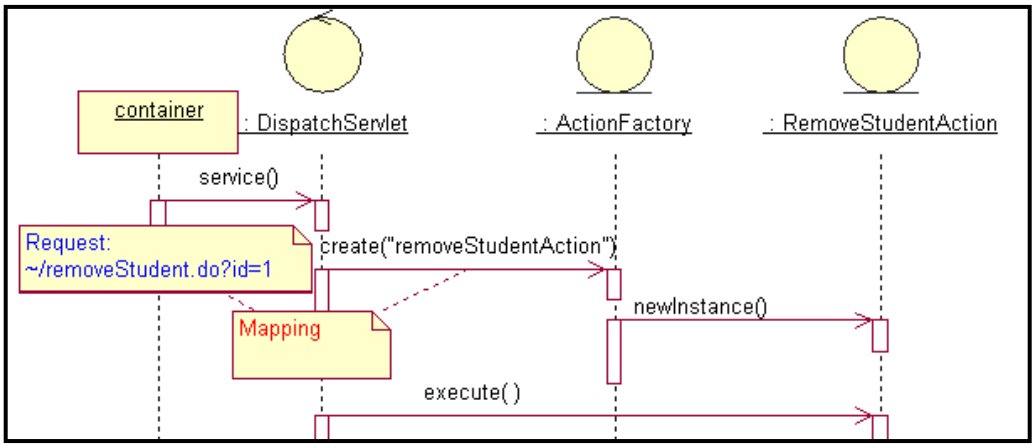


Рисунок 2. Обробка запиту "removeStudent"

Наведені (рис. 1 та рис. 2) дві діаграми послідовності *UML* (вони стосуються *web*-проекту "Студенти" як конкретного прикладу проекта із пітримкою типової *CRUD*-функціональності: *Create-Read-Update-Delete*) ілюструють диспетчеризацію для наступних двох варіантів запитів: запитів щодо створення нових даних про студентів (клас-обробник таких запитів – *AddStudentAction*) та запитів щодо вилучення даних про студентів (клас-обробник таких запитів – *RemoveStudentAction.* (В *UML*-коментарі наведено конкретний запит на вилучення даних про студента із заданим *id*).

В обох діаграмах фігурує об'єкт-фабрика (об'єкт типу *ActionFactory*), який залежно від варіанту *web*-запиту створює необхідний об'єкт-обробник (*Action*-об'єкт). Так для зазначених двох варіантів запитів створюються відповідно об'єкт типу *AddStudentAction* чи об'єкт типу *RemoveStudentAction*. Використання об'єкта-фабрики тут лише акцентує увагу на факті динамічного створення *Action*-об'єктів, проте сам фабричний об'єкт можна й елімінувати. У такому випадку реалізація методу *service* (він, нагадаємо, при обробці *web*-запитів є ключовим у *Java*-класі *HttpServlet*) набуває у класі *DispatchServlet* зовсім лаконічного вигляду (рис. 3).

Диспетчеризація запитів базується на використанні "таблиці" *map* (типу *HashMap*). Така таблиця визначає відображення (*mapping*) рядків

keyStringFromRequest (з *URL*-адрес – запитів) в *Action*-класи: *String actionNameFromMapping* = *getMap*(*keyStringFromRequest* (див. рис. 3).

```
public void service(HttpServletRequest request,
                    HttpServletResponse response)
 throws ServletException {
  try {
   String keyStringFromRequest = getKeyString(request);
   String actionNameFromMapping = map.get(keyStringFromRequest);
   Action action = (Action) (Class.forName
                     (actionNameFromMapping).newInstance());
   String urlNext = action.execute(request, response);
   if (urlNext != null)
           getServletContext().getRequestDispatcher(urlNext).
                    forward(request, response);
  } catch (Exception e) {
                           e.printStackTrace();
                       }
 }
```

Рисунок 3. Реалізація основного сервлетного методу service

```
<?xml version="1.0" encoding="UTF-8" ?>
  <mapping>
      path="/index"
      type="com.ttp.InitAction">
   </action>
   <action
      path="/addStunentAction"
      type="ttp.AddStudentAction">
   </action>
   <action
      path="/removeStunentAction"
      type="ttp.RemoveStudentAction">
   </action>
   <action
    . . .
  </mapping>
```

Рисунок 4. Конфігураційний файл фреймворку – файл minia.xml

Формується "таблиця" *map* на стадії *налаштування фреймворку* за *mapping*-даними зі спеціального *конфігураційного файлу minia.xml*. Власне формування "таблиці" *map* повністю вичерпує задачу налаштування *Miniature* (рис. 4).

Наведемо склад фреймворку Miniature:

- *єдиний сервлетний (диспетчерський) клас* – `class DispatchServlet extends HttpServlet` (його і тільки його треба прописувати у стандартному для *web*-проектів конфігураційному файлі *web.xml*).
- *інтерфейс* `Action` (рис. 5) з єдиним методом `execute`.

```
public interface Action {
  public String execute(HttpServletRequest request,
                        HttpServletResponse response);
}
```

Рисунок 5. Інтерфейс *Action*

До складу *web*-проектів, заснованих на фреймворку *Miniature*, також мають додаватись:

- конфігураційний файл фреймворку *Miniature* – *minia.xml* (із метою *mapping*-налаштування);
- необхідні файли класів-обробників (нагадаємо, що такі класи мають реалізовувати інтерфейс *Action*) та відповідні їм (згідно з принципами *MVC*-архітектури) *view*-файли (найчастіше такими виступаються *JSP*- чи навіть *HTML*-файли). (На рис. 6 наведено приклад класу *InitAction* з використаним *view* – *studentList.jsp* в якості *urlNext* згідно з реалізацією диспетчерського сервлету (див. рис. 3).

Отже, основними архітектурними особливостями *web*-проектів, що будуються на основі фреймворку *Miniature*, є наступні три:

1) використовується *єдиний сервлетний клас ttp.DispatchServlet*, призначений для диспетчеризації запитів;
2) власте диспетчеризація базується на *відображенні* (*mapping*) *URL-шлях – обробник* (*ActionClass*), а її *налаштування* повністю

визначається *конфігураційним файлом фреймворку minia.xml*; *класи-обробники* (*Action*-класи) мають реалізовувати визначений у фреймворку інтерфейс *ttp.Action* з єдиним методом *execute* (призначеним для "запуску" обробки запиту).

```
public class InitAction implements Action {
     public String execute  (HttpServletRequest request,
       HttpServletResponse response) {
          return "/studentList.jsp";
     }
}
```

Рисунок 6. Клас *InitAction*

Важливо, що зазначені архітектурні засади притаманні багатьом іншим фреймворкам, зокрема, таким промисловим фреймворкам як *Spring*, *Struts*, *WebWork*, *Stripes*, *Struts* 2 та ін. Отже, з огляду на простоту фреймворку *Miniature* його можна розглядати як варіант *шаблона* (*патерна*) для багатьох популярних *Java web*-фреймворків.

На переконання у такому висновку проведемо аналіз на "відповідність" патерну для двох чи не найпопулярніших *Java web*-фреймворків *Spring* [2] та *Struts* [3].

1. Фреймворк *Spring* **:**
   1) використовується *єдиний диспетчерський сервлетний клас* як *Front Controller – org.springframework.web.servlet.DispatcherServlet;*
   2) ім'я конфігураційного файлу фреймворку залежить від імені сервлетного об'єкта, скажімо, для об'єкта *dispatcherServlet* це буде *dispatcherServlet-servlet.xml*. Проте важливішим є те, що в ньому містяться дані *mapping*-налаштування;
   3) *класи-обробники* мають реалізовувати визначений у фреймворку спеціальний інтерфейс *org.springframework.web.servlet.mvc.Controller.*
2. Фреймворк *Struts*:
   1) використовується єдиний диспетчерський сервлетний клас – *org.apache.struts.action.ActionServlet*;

2) конфігураційний файл фреймворку має назву *struts-config.xml* та містить дані *mapping*-налаштування;
3) класи-обробники мають успадковуватись від абстрактного класу *org.apache.struts.action.Action* з єдиним методом *execute*.

Зрозуміло, що навчальний фреймворк *Miniature* розроблявся, абстрагуючись від деяких практично важливих проблем *web*-проектування. Зокрема, осторонь залишилися проблеми, пов'язані із валідацією даних, локалізацією, інтернаціоналізацією. Варто також зазначити, що у тому ж *Spring*, одному з найпотужніших існуючих фреймворків за кількістю готових класів, запропоновано чотири класи для різних варіантів реалізації *Handler Mapping*, десять варіантів *Controller*-класів із реалізованим інтерфейсом *Controller*, подібним *Action* тощо.

Висновки. У роботі надано архітектурні засади та практично повністю подана реалізація складових частин запропонованого навчального *web*-фреймворку *Miniature*, демонструючи тим самим підхід до реалізації власних *web*-фреймворків.

Web-фреймворк *Miniature* можна розглядати як варіант шаблону (патерну) *Java MVC web*-фреймворків, і, отже, ознайомлення з ним дозволяє швидко адаптовуватись до практичного застосування цілої низки популярних промислових *web*-фреймворків таких, як *Spring, Struts, WebWork, Stripes* та ін.

Список використаних джерел

1. Miller R.W. Introduction to Java Servlet technology [Електронний ресурс] // Режим доступу до ресурсу: http://www. ibm.com/ developerworks/java/tutorials/j-intserv/.
2. What Spring can do [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://spring.io/.
3. Apache Struts [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: https://struts.apache.org/.

Омельчук Людмила Леонідівна, к.ф.-м.н., доцент

ДИСЦИПЛІНА «ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ» ТА ЇЇ МІСЦЕ В СТРУКТУРНО-ЛОГІЧНІЙ СХЕМІ ОСВІТНЬО-ПРОФЕСІЙНОЇ ПРОГРАМИ «ІНФОРМАТИКА»

Роботу присвячено концепції викладання обов'язкової навчальної дисципліни «Об'єктно-орієнтоване програмування» для студентів факультету комп'ютерних наук та кібернетики що навчаються на першому (бакалаврському) рівні вищої освіти за освітньо-професійною програмою «Інформатика».

Особливістю викладання дисципліни «Об'єктно-орієнтоване програмування» є використання проблемно орієнтованого підходу до навчання при якому студенти опановують дисципліну в процесі вирішення проблем, що виникають при написанні лабораторних робіт (проектів) та виконання домашніх завдань. Такий спосіб навчання дає змогу набути знання методів об'єктно-орієнтованого аналізу та проектування програмних систем, знання парадигм програмування, принципів парадигми об'єктно-орієнтованого програмування, знання шаблонів та принципів об'єктно-орієнтованого проектування та опанувати практичні навички об'єктно-орієнтованго аналізу, проектування та програмування.

Ключові слова: навчальна дисципліна, об'єктно-орієнтований аналіз, об'єктно-орієнтований проектування, об'єктно-орієнтоване програмування, шаблони проектування.

These papers are devoted to the description of the principles of teaching the compulsory subject "Object-Oriented Programming" for students of the Faculty of Computer Science and Cybernetics studying at the first (bachelor's) level of higher education in the educational-professional program "Informatics".

The peculiarity of teaching this subject is the use of problem-oriented approach to learning. In this approach, students master the subject in the process of solving problems that arise when writing laboratory work (projects) and homework. This method of teaching allows you to acquire knowledge of methods of object-oriented analysis and design of software systems, knowledge of paradigms, patterns and principles of object-oriented programming and master the practical skills of object-oriented analysis, design and programming.

Keywords: discipline, object-oriented analysis, object-oriented design, object-oriented programming, design templates.

Розробка програмного забезпечення в Україні є індустрією, що динамічно розвивається. Результати “Аналізу ІТ-освіти у вишах України” [1], розробленого експертами Офісу ефективного регулювання BRDO в лютому 2021 року свідчать, що наразі попит на нових ІТ-фахівців в Україні складає 30-50 тисяч осіб на рік. У свою чергу, заклади вищої освіти щороку випускають у середньому 16,2 тисяч бакалаврів ІТ-спеціальностей [1].

Разом з тим 44% випускників українських закладів вищої освіти після отримання диплому працюють за фахом [2], інші не відповідають рівню своєї кваліфікації.

Отже, проблема підвищення якості професійної підготовки ІТ-фахівців, кваліфікація та рівеь фахової підготовки яких відповідали б сучасним потребам ІТ-ринку та світовим вимогам є актуальною.

Місце дисципліни «Об’єктно-орієнтоване програмування» в структурно-логічній схемі освітньої програми

Одним з ключових елементів професійної підготовки ІТ-фахівців є дисципліни, основним завданням яких є практична та професійна підготовка здобувачів до роботи за спеціальністю. Ключову роль в такій підготовці відіграють дисципліни програміського спрямування.

У випадку освітньо-професійної програми «Інформатика» першого (бакалаврського) рівня вищої освіти за спеціальністю 122 «Комп’ютерні науки» [3], яка реалізується на факультеті комп’ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка цикл дисциплін програміського спрямування розпочинає у першому та другому семестрах обов’язкова фахова дисципліна «Програмування».

З метою покращення та контролю за актуальністю матеріалу дисципліни лектором щорічно здійснюються два анонімних опитування студентів, що опановують дисципліну «Об’єктно-орієнтоване програмування» [4, 5]: вхідне (посилання на Google форму оприлюднюється на першому занятті) та вихідне (посилання на Google форму

оприлюднюється на після завершення навчальних занять, але до проведення іспиту).

Основною методичною проблемою дисципліни «Програмування» є велика відмінність у рівні програміських навичок першокурсників за освітньою програмою. На програму щорічно вступає 80-125 першокурсники. Найнижчий прохідний рівень вступників, зарахованих на загальних засадах на бюджетну форму навчання за освітньою програмою «Інформатика» у 2020 році склав 191,8 бали, у 2019 році – 189,4 бали, а у 2018 році – 189,3 бали з 200 можливих [6]. Навіть при такому високому рівні підготовки вступників серед першокурсників існує суттєве розшарування за рівнем підготовки з програмування. Так, традиційно на цю освітню програму певна частина здобувачів приходять з достатньо великим досвідом програмування, учасники олімпіад, і одночасно з тим достатньо велика частка першокурсників вперше стикаються з цією дисципліною в саме під час вивчення дисципліни «Програмування» в університеті. Так, відповідно до результатів вхідного опитування студентів другого курсу у 2020/2021 н.р. дисципліни «Об’єктно-орієнтоване програмування», серед 115 респондентів 3,5% написали свою першу програму в 1-4 класі школи, 43,5% – в 5-9 класі, 31,3% – 10-11 клас і 21,7% – на 1 курсі. (див. рис. 1):

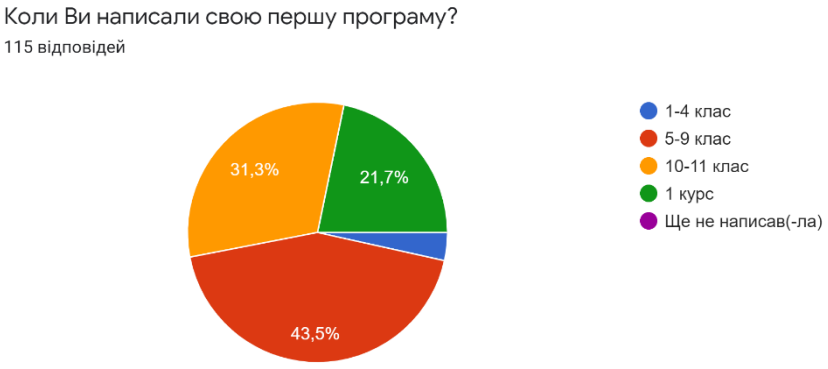


Рисунок 1. Результати опитування студентів до початку вивчення дисципліни «Об'єктно-орієнтоване програмування»

Тому, важливою задачею, вирішення якої покладається на дисципліну «Програмування», що викладається на першому курсі є опанування основами програмування здобувачами, які вперше зіштовхуються з цією дисципліною та розвиток програміських навичок у більш досвідчених студентів.

На другому курсі цикл програміських дисциплін за освітньою програмою в третьому семестрі продовжується обов'язковими дисциплінами «Об'єктно-орієнтоване програмування» та «Архітектура обчислювальних систем та комп'ютерні мережі», а в четвертому семестрі обов'язковими дисциплінами «Інструментальні середовища та технології програмування» та «Бази даних та інформаційні системи». Крім того, з переліку вибіркових дисциплін студенти на четвертому семестрі навчання можуть обрати вибіркові дисципліни «Алгоритміка» чи «Прикладні алгоритми». Усі перелічені дисципліни покликані розвивати навички професійної роботи в ІТ-галузі за єдиною схемою та логічно доповнюють одна одну. Так, дисципліна «Об'єктно-орієнтоване програмування», базуючись на отриманих в межах дисципліни «Програмування» знаннях та уміннях, що включають в себе початкові знання з парадигми об'єктно-орієнтованого програмування, дозволяє студентів поглиблено опанувати одну із найбільш поширених у практичному програмуванні парадигм програмування – об'єктно-орієнтоване програмування. Водночас дисципліна «Архітектура обчислювальних систем та комп'ютерні мережі» надає здобувачам знання технічних аспектів ІТ-галузі. Після завершення дисциплін третього семестру в межах дисциплін четвертого семестру «Інструментальні середовища та технології програмування» та «Бази даних та інформаційні системи» студенти мають можливість навчитися використовувати технології програмування та бази даних для розробки проектів максимально наближених до реальних потреб роботодавців. При цьому, в межах дисципліни «Інструментальні середовища та технології програмування» увага слухачів акцентується на вивчених в «Об'єктно-орієнтованому програмування» парадигмах, принципах проектування та програмування на

реалізації опанованих шаблонів проектування в використовуємих технологіях програмування.

Таким чином, дисципліна «Об'єктно-орієнтоване програмування» є невід'ємною компонентою програми підготовки здобувачів першого (бакалаврського) рівня вищої освіти в галузі знань 12 «Інформаційні технології» за спеціальністю 122 «Комп'ютерні науки» та органічно вписується в структурно логічну схему освітньої програми.

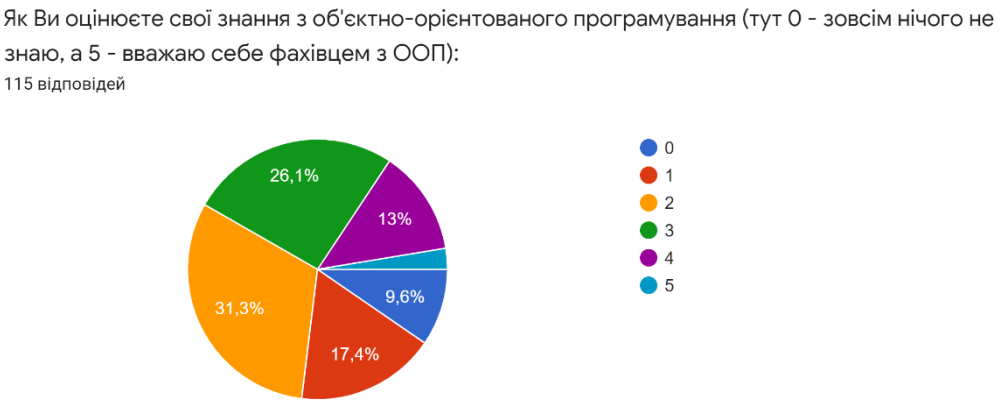


Рисунок 2. Результати опитування студентів до початку вивчення дисципліни «Об'єктно-орієнтоване програмування»

Відповідно до результатів вхідного опитування 115 студентів другого курсу у 2020/2021 н.р. з дисципліни «Об'єктно-орієнтоване програмування», з них, оцінили свої знання парадигми об'єктно-орієнтованого програмування (до вивчення дисципліни), як 0 (зовсім нічого не знаю) - 9,6%, 1 – 17,4%, 2 – 31,3%, 3 – 26,1%, 4 – 13% та 5 (вважаю себе фахівцем з об'єктно-орієнтованого програмування) – 2,6% опитаних (див. рис. 2).

Для задоволення потреб здобувачів при вивченні дисципліни важливим є аналіз їх очікувань перед початком її вивчення. Відповідно до результатів вхідного опитування 115 респондентів з дисципліни «Об'єктно-орієнтоване програмування» на відкрите запитання про очікування студентів від вивчення дисципліни "Об'єктно-орієнтоване програмування" було

отримано 109 відповідей. Серед цих відповідей варто відмітити наступні очікування (формулювання авторів збережено):

- *«Корисна та сучасна інформація, яка нам допоможе в подальшому, розвиваючі корисні навички для програмування».*
- *«Отримання бази для проектування та створення програм використовуючи ідеї об'єктно-орієнтованого програмування».*
- *«Сподіваюсь покращити свої навички проектування програмного забезпечення. Сподіваюсь отримати гарну практику програмування».*
- *«Покращити знання з парадигми ООП, виконати низку лабораторних, які можна буде написати настільки добре, щоб пізніше використовувати на співбесідах».*
- *«Розвиток в області програмування мовою C#, знайомство з технологіями платформи .NET Core 3.1 (WPF, ASP.NET). Знайомство з паттернами проектування».*
- *«Дізнатися більше про проектування та архітектуру ПЗ. Написати проект, який допоможе закріпити свої знання з цього предмету».*
- *«Дізнатися більше про проектування БД».*
- *«Покращити знання мови C#, добре розібратись з ООП, дізнатись про плюси і мінуси такого підходу до програмування, вміти використовувати отриманні знання на практиці».*
- *«Краще зрозуміти як це працює і самій попробувати використовувати принципи ООП в написані програм».*
- *«Отримаю знання і досвід, який потім знадобиться при написанні проектів, котрі мене цікавлять».*
- *«Отримати хорошу базу, щоб в подальшому можна було самостійно вивчати нові мови та технології».*
- *«Очікую потужне просування у галузі програмування, цікаві лабораторні та новий досвід».*
- *«Дізнатися більше про програмування взагалі, та отримати корисні практично навички».*
- *«Структурована інформація з можливістю застосувати теоретичні знання на практиці».*

Таким чином, від обов’язкової дисципліни «Об’єктно-орієнтоване програмування» студенти в першу чергу очікують набуття практичних навичок проектування та програмування з використанням об’єктно-орієнтованої парадигми програмування. Ці очікування відповідають основним завданням дисципліни та її ролі в структурно-логічній схемі дисциплін прогаміського спрямування освітньої програми «Інформатика» та вимогам роботодавців в галузі ІТ, що висуваються до випускників [3, 4, 7-9].

Структура дисципліни «Об’єктно-орієнтоване програмування»

Метою дисципліни «Об’єктно-орієнтоване програмування» – є засвоєння базових знань з основ об'єктно-орієнтованого програмування, включаючи основні поняття, парадигми та принципи об'єктно-орієнтованого програмування. Оволодіння базовими навичками проектування програмних систем, роботи з найбільш вживаними шаблонами об’єктно-орієнтованого проектування, набуття практичних навичок об’єктно-орієнтованого аналізу, проектування та програмування.

Особливістю викладання дисципліни є використання концепції проблемно орієнтованого навчання при якому студенти опановують дисципліну в процесі вирішення проблем пов’язаних з об’єктно-орієнтованим аналізом, проектуванням та програмуванням, що виникають при написанні лабораторних робіт (проектів) та виконання домашніх завдань. Такий спосіб вивчення дисципліни «Об’єктно-орієнтоване програмування» передбачає набуття як теоретичних знань під час лекцій так і практичних навичок роботи шляхом виконання лабораторних робіт (проектів) та домашніх завдань, а також та закріплення цих знань та навичок під час самостійної роботи.

При цьому, на лекціях студенти вивчають новий теоретичний матеріал, два лекційних заняття присвячені опануванню нового матеріалу у форматі так званого «перевернутого класу», коли самі здобувачі під час самостійної роботи розбирають новий матеріал та на лекції пояснюють його своїм однокурсникам. Під час лабораторних занять відбувається закріплення вивченого на лекціях теоретичного матеріалу та опрацювання за його темами завдань. Завдання для лабораторних занять включають в себе

наступні форми та методи оцінювання: виконання семи домашніх завдань за різними темами дисципліни, три лабораторні роботи та оформлення одного звіту до лабораторних робіт. На кожен вид робіт визначені свої максимальні бали, вимоги та граничні терміни виконання, з якими студенти можуть ознайомитися в робочій навчальній програмі дисципліни, та на першому лекційному і лабораторному занятті. Під час самостійної роботи та на лабораторних заняттях студенти виконують завдання, що отримали на лабораторних заняттях та опрацьовують лекційний матеріал.

Робочою програмою навчальної дисципліни передбачається, що до кінця семестру кожен студент:

• виконає та захистить 3 (три) лабораторні роботи (проекти);

• виконає 7 (сім) домашніх завдань;

• успішно напише 2 (дві) контрольні роботи.

В кінці семестру передбачено іспит за всіма темами семестру.

Семестрова оцінка – 60 балів; із них лабораторні роботи (проекти) та діаграми до них – 30 балів, домашні завдання – 10 балів, контрольні роботи – 20 балів. Екзаменаційна робота – 40 балів. Максимальна підсумкова оцінка –100 балів.

Студент має право на одне перескладання контрольної роботи із можливістю отримання 7 балів. Термін перескладання визначається викладачем.

При виконанні лабораторних робіт студент має право використовувати мови об’єктно-орієнтованого програмування відмінні від C#. Контрольні роботи мають на меті перевірити знання, отримані студентом під час лекційних занять та не залежать від мови написання лабораторної роботи. У разі використання студентом альтернативних мов об’єктно-орієнтованого програмування (відмінних від C#) викладач додає до результату контрольної роботи 10% від набраної студентом кількості балів, але при цьому загальний результат контрольної роботи не повинен перевищувати максимально можливу оцінку з цієї роботи (своєчасне написання роботи – 10 балів, переписування, або несвоєчасне написання – 7 балів).

Принаймні за два тижні до здачі кожної з лабораторних робіт обов'язковим є демонстрація трьох діаграм UML (1 діаграма прецедентів (Use case diagram), 1 діаграма класів (Class diagram) та 1 діаграма послідовності (Sequence diagram)) з можливістю отримання 1 балу за кожен набір діаграм.

Лабораторна робота (проект) №1 включає в себе розробку програми для роботи з електронними таблицями (аналіз та обчислення виразів) за індивідуальними заняттями. Для аналізу виразів студентам пропонується використовувати ANTLR, розглядається на лабораторних заняттях.

Лабораторна робота (проект) №2 включає в себе роботу аналіз XML файлів, його аналіз. Аналіз включає в себе дві задачі: аналіз вмісту документу (пошук інформації за ключовими словами та динамічну генерацію запитів) та трансформацію у файл HTML. Вхідні дані для аналізу та трансформації надаються у вигляді файлу-прикладу *.xml. Трансформація документу в HTML-код виконується на основі XSL-документа *.xsl. Аналіз вмісту документа повинен бути виконаний трьома способами – за допомогою SAX API, DOM API та LINQ to XML. Необхідно реалізувати всі три способи для цього обов'язковим є використання шаблонів проектування. Наприклад, використання шаблону «Стратегія» Strategy. Тематика XML-файлів індивідуальна для кожного студента.

Лабораторна робота (проект) №3 полягає у розробці програми обробки файлів та редагування текстів. Включає в себе розробку файлового менеджера та текстового редактора. Використання шаблонів проектування є обов'язковим.

Метою перших занять з дисципліни є ознайомлення студентів з основними поняттями об'єктно-орієнтованого програмування, основними етапами життєвого циклу програмних систем (77,2% студентів опитаних після проходження курсу оцінили ці теми як важливі для дисципліни).

На подальших заняттях здобувачі ознайомлюються з основними принципами об'єктно-орієнтованого аналізу та моделювання програмних систем. Вивчають мову UML, роблячи основний акцент на діаграмах

прецедентів, класів та послідовності використання. З метою закріплення знань за набуття практичних навичок з цієї теми приклади побудови цих діаграм розглядаються на лабораторному занятті та студенти виконують домашнє завдання з самостійної побудови Use-case діаграми, Class-діаграми та Sequence-діаграми. В подальшому ці типи діаграм здобувачі повинні розробляти на початковому етапі виконання трьох лабораторних робіт, що є обов'язковими при опанування дисципліни «Об'єктно-орієнтоване програмування». 74,4% студентів опитаних після проходження курсу оцінили ці теми як важливі для дисципліни.

Наступна тема, готує студентів до практичного застосування об'єктно-орієнтованого програмування. Для цього на лекціях надається теоретичний матеріал з об'єктно-орієнтованої мови програмування C#, на лабораторних заняттях набуваються навички практичного застосування мови C#, а під час самостійної роботи студенти опрацьовують лекційний матеріал та закріплюють практичні навички шляхом виконання домашніх завдань та лабораторної роботи №1. Зважаючи на те, що під час вивчення на першому курсі дисципліни «Програмування» здобувачі опановували мову C++, вивчення мови C# відбувається на основі її порівняльного аналізу з C++. 85,1% студентів опитаних після проходження курсу оцінили ці теми як важливі для дисципліни.

Розглядаються основні вимоги і рекомендації з написання роду та рефакторинг. З моменту вивчення цієї теми при виконання домашніх завдань, лабораторних та контрольних робіт дотримання студентами усіх розглянутих вимоги до якості коду є обов'язковим. 93,6% студентів опитаних після проходження курсу оцінили ці теми як важливі для дисципліни.

Для виконання першої лабораторної роботи (проекту) під час лабораторних занять викладач знайомить студентів з основами синтаксичного аналізу та обчислення виразів та правилами використання ANTLR в .Net. 89,3% студентів опитаних після проходження курсу оцінили ці теми як важливі для дисципліни.

Для виконання другої лабораторної роботи (проекту) під час лабораторних занять викладач знайомить студентів з основами XML,

трансформацією XML, LINQ to XML. 80,9% студентів опитаних після проходження курсу оцінили ці теми як важливі для дисципліни.

Наступна тема, яка розглядається на лекціях, лабораторних заняттях та самостійній роботі є тема, присвячена вивченню шаблонів проектування, їх призначенню, класифікації та детальному вивченню на конкретних прикладах найбільш розповсюджених шаблонів. Для закріплення розглянутого матеріалу студентам запропоновано в межах лабораторних робіт використовувати визначені шаблони проектування, виконати домашні завдання з реалізації обраних шалонів, підготувати доповідь з визначених викладачем шаблонів проектування програмного забезпечення. 91,5% студентів опитаних після проходження курсу оцінили ці теми як важливі для дисципліни.

Останні лекційні та лабораторні заняття включають вивчення основних антишаблонів та принципів проектування програмного забезпечення S.O.L.I.D. 91,5% студентів опитаних після проходження курсу оцінили ці теми як важливі для дисципліни.

Результати дисципліни «Об'єктно-орієнтоване програмування»

Відповідно до анонімного підсумкового опитування яке було проведено лектором по завершенню вивчення студентами дисципліни «Об'єктно-орієнтоване програмування» 40,4% респондентів повністю погоджуються з твердженням, що ця дисципліна якісно підвищила їх професійну кваліфікацію, 36,2% – в більшій мірі погоджуються з цим твердженням, 14,9% – однаковою мірою погоджуються і не погоджуються, 6,4% – переважно не погоджуються та 2,1% – абсолютно не погоджуються.

38,3% респондентів повністю погоджуються з твердженням, що в ході вивчення дисципліни здобули багато корисних практичних навичок та вмінь, 40,4% – в більшій мірі погоджуються з цим твердженням, 14,9% – однаковою мірою погоджуються і не погоджуються, 4,2% – переважно не погоджуються та 2,1% – абсолютно не погоджуються.

51,1% респондентів повністю погоджуються з твердженням, що в ході вивчення дисципліни здобули багато нових та корисних знань, 29,8% – в більшій мірі погоджуються з цим твердженням, 10,6% – однаковою мірою

погоджуються і не погоджуються, 6,3% – переважно не погоджуються, та 2,1% – абсолютно не погоджуються.

57,4% респондентів повністю погоджуються з твердженням, що вони повністю розуміють роль та значення дисципліни в рамках спеціальності та освітньої програми, 31,9% – в більшій мірі погоджуються з цим твердженням, 8,5% – однаковою мірою погоджуються і не погоджуються, та 2,1% – абсолютно не погоджуються.

Переважна кількість опитаних після прослуховування дисципліни відмітила підвищення рівня знань за усіма темами дисципліни. Зокрема,

За темами «Основні поняттями об’єктно-орієнтованого програмування» та «Основні етапами життєвого циклу програмних систем» в середньому студенти оцінили свої знання до прослуховування курсу на 2,96 бали за п’ятибальною шкалою, а після – на 4,2 бали. Тобто, здобувачі визнали, що дисципліна дозволила покращити рівень їх знань на 1,24 бал.

За темами, присвяченими вивченню мови програмування C# в середньому студенти оцінили свої знання до прослуховування курсу на 2,75 бали за п’ятибальною шкалою, а після – на 4,04 бали. Тобто, здобувачі визнали, що дисципліна дозволила покращити рівень їх знань на 1,29 бала.

За темами, присвяченими рефакторингу та вимогам і рекомендаціям з написання коду в середньому студенти оцінили свої знання до прослуховування курсу на 2,9 бали за п’ятибальною шкалою, а після – на 4,2 бали. Тобто, здобувачі визнали, що дисципліна дозволила покращити рівень їх знань на 1,3 бала.

За темами, присвяченими вивченню шаблонів проектування в середньому студенти оцінили свої знання до прослуховування курсу на 2,2 бали за п’ятибальною шкалою, а після – на 4 бали. Тобто, здобувачі визнали, що дисципліна дозволила покращити рівень їх знань на 1,8 бала.

За темами, присвяченими вивченню антишаблонів проектування в середньому студенти оцінили свої знання до прослуховування курсу на 2,2 бали за п’ятибальною шкалою, а після – на 4,1 бали. Тобто, здобувачі визнали, що дисципліна дозволила покращити рівень їх знань на 1,9 бала.

За темами, присвяченими вивченню принципів проектування S.O.L.I.D. в середньому студенти оцінили свої знання до прослуховування

курсу на 2,2 бали за п’ятибальною шкалою, а після – на 4 бали. Тобто, здобувачі визнали, що дисципліна дозволила покращити рівень їх знань на 1,8 бала.

Таким чином, можна стверджувати, що в загальному дисципліна «Об’єктно-орієнтоване програмування» сприяє досягненню мети та покращує рівень знань і умінь слухачів.

Висновки. Таким чином, використання проблемно орієнтованого підходу до викладання навчальної дисципліни «Об’єктно-орієнтоване програмування» дає змогу набувати знання методів об’єктно-орієнтованого аналізу та проектування програмних систем, знання парадигм програмування, принципів парадигми об’єктно-орієнтованого програмування, знання шаблонів та принципів об’єктно-орієнтованого проектування.

Разом з тим, основною метою дисципліни є набуття практичних навичок об’єктно-орієнтованого аналізу, проектування та програмування.

При цьому практичні навички і теоретичні знання студенти опановують в процесі вирішення проблем з об’єктно-орієнтованого аналізу проектування та програмування, що виникають при написанні лабораторних робіт (проектів) та виконання домашніх завдань за темами дисципліни.

Список використаних джерел

1. Лебедєв Д. Аналіз ІТ освіти у ВИШах України [Електронний ресурс] / Д. Лебедєв, І. Самоходський // https://brdo.com.ua/. – 2021. – Режим доступу до ресурсу: https://brdo.com.ua/wp-content/uploads/2021/02/Analiz_IT_osvity_u_vyshah_Ukrai-ny_Print.pdf.

2. Експерт: 44% випускників вишів працюють не за фахом [Електронний ресурс] // Укрінформ. – 2021. – Режим доступу до ресурсу: https://www.ukrinform.ua/rubric-society/2737000-ekspert-44-vipusknikiv-visiv-pracuut-ne-za-fahom.html.

3. Київський національний університету імені Тараса Шевченка. Освітньо-професійна програма «Інформатика»: Спеціальність 122 Комп’ютерні науки [Електронний ресурс]. – 2019. – Режим доступу до

ресурсу: http://csc.knu.ua/media/filer_public/09/f4/09f46516-6c61-46a5-ae81-f34b7d673499/opp_122_bac_2019_1.pdf.

4. Бойко Б.І. Об'єктно-орієнтоване програмування. Лабораторний практикум: навчальний посібник / Б.І. Бойко, Л.Л. Омельчук, Н.Г. Русіна. – Київ. – 90 с.

5. Зубенко В.В., Омельчук Л.Л. Програмування. Поглиблений курс – Київ: Видавничо-поліграфічний центр "Київський університет", 2011. – 623 с.

6. Вступна кампанія [Електронний ресурс] // Єдина Державна Електронна База з Питань Освіти. – 2021. – Режим доступу до ресурсу: https://vstup.edbo.gov.ua/.

7. European e-Competence Framework, e-CF [El. resource]. – URL: www. ecompetences.eu.

8. Розробка та впровадження галузевої рамки кваліфікацій в галузі знань «Інформаційні технології» / В. А. Заславський, М. С. Нікітченко, Л. Л. Омельчук, О. М. Ямкова. – Київ: Київський національний університет, 2016. «Добродій» – 88 с. ISBN 978-966-97595-1-1.

9. Дослідження ефективності IT-освіти. [Ел ресурс]: http://itukraine.org.ua/ sites/default/files/prezentaciya_it-obrazovanie.pdf.

Ткаченко Олексій Миколайович, к.т.н., доцент

ВИКЛИКИ І ПЕРСПЕКТИВИ ВИВЧЕННЯ ТЕХНОЛОГІЙ ПРОГРАМУВАННЯ ДЛЯ МОБІЛЬНИХ ПЛАТФОРМ

Дослідження присвячено огляду основних факторів, які впливають на формування сучасного ринку мобільної розробки. Глобальні аналітичні звіти вказують на збереження трендів проникнення мобільних пристроїв в суспільстві, зростання мобільного трафіку, зростання інвестицій та попиту на розробників. Це привносить нові виклики для ІТ-освіти. Зміст навчальних курсів з мобільної розробки повинен враховувати поширеність обох мобільних операційних систем, середовищ розробки, мов програмування та сучасні фреймворки для кросплатформної розробки. Робота пропонує дещо з досвіду викладання курсу «Розробка ПЗ під мобільні платформи» в КНУ імені Тараса Шевченка та результатів опитування студентів.

Ключові слова: програмне, забезпечення, мобільна, платформа, розробка.

In the study is devoted to an overview of the main factors influencing the formation of the modern market of mobile development. Global analytics reports point to continuing trends in mobile penetration, growing mobile traffic, increasing investment and demand for developers. This brings new challenges for IT education. The content of mobile development learning courses should take into account the sharing of both mobile operating systems, development environments, programming languages and modern frameworks for cross-platform development. We offers some of the experience of teaching the course "Software Development for Mobile Platforms" at Taras Shevchenko National University of Kyiv and the results of student surveys.

Keywords: software, mobile, platform, development.

Останні кілька десятків років ми є свідками стрімкої зміни способів роботи з інформацією. За ці роки змінилося немало засобів транслювання масової інформації, комунікації, мультимедійних стандартів тощо. Але чи не основною є зміщення «центру ваги» в системі «людина–джерело інформації». Раніше можна було окреслити цілком визначений перелік

джерел отримання інформації (ЗМІ, бібліотеки, стаціонарні телефони тощо), і якщо користувачам необхідна була деяка інформація, вони повинні були дістатися засобів/інституцій доступу до неї. Поширення мобільних комунікаційних сервісів, швидкісного Інтернету та соціально-орієнтованого Web [1] цю модель по суті перевернула. Нині користувачу достатньо мати в кишені сучасний смартфон і доступ до швидкої мережі передачі даних. Саме він зі своїм цифровим пристроєм тепер знаходиться в центрі системи «людина–інформаційні джерела».

У відомих глобальних звітах [2] очевидною є стрімка позитивна динаміка мобільного трафіку (рис. 1).

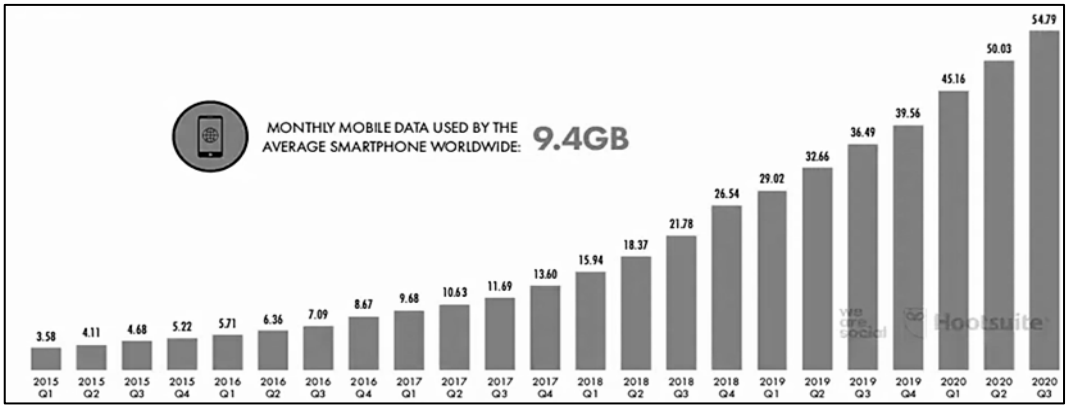


Рисунок 1. Динаміка обсягу мобільного Інтернет-трафіку за 2015-2020 рр.

Рис. 2 демонструє зміни в структурі глобального Інтернет-трафіку в розрізі типів пристроїв [3]. Бачимо стійкий тренд зміщення в бік використання смартфонів.

Ця публікація окреслює фактори, які визначають перспективність розробки для мобільних платформ, пропонує дещо з досвіду викладання відповідних навчальних дисциплін та погляд здобувачів на місце вивчення мобільних ІТ.

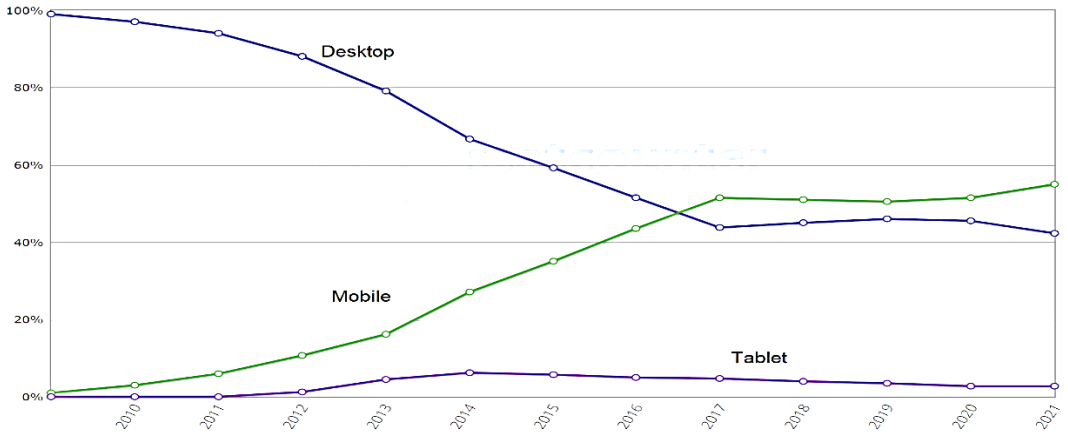


Рисунок 2. Динаміка розподілу Інтернет-трафіку у світі за типами пристроїв

Основна частина

Зміну парадигми у взаємодії людини з інформаційними джерелами автор схильний вважати черговою інформаційною революцією, оскільки це обумовлює якісні зміни у функціонуванні суспільства. Наприклад, зменшення цифрової нерівності, зокрема, через отримання доступу до нових глобальних джерел новин та аналітики, наукових публікацій, навчальних ресурсів; зміна бізнес-моделей компаній, у т.ч. зміна моделі дистрибюції цифрових продуктів, поширення технологій е-банкінгу тощо. Разом з тим, кожна така технологія може нести нові виклики, наприклад, пов'язані з безпекою персональних даних, банківських рахунків, національною безпекою тощо. Зрозуміло, що такі суспільні зміни є фактором зростання інвестицій в ІТ, зокрема, у розробку мобільного програмного забезпечення. За даними Digitalreport [2, ст. 216], у 2020 р. користувачі витратили 143 млрд доларів США за купівлю мобільного ПЗ, що на 20% більше, ніж за попередній період.

Інвестиції у перспективні розробки передбачають забезпечення підготовленими фахівцями, що висуває оновлені вимоги до освітніх програм ІТ-спеціальностей. Виокреслюється суспільна потреба у підготовці фахівців, які готові створювати та підтримувати ІТ-продукти, що ми відносимо до мобільних (смартфони, планшети, смарт-годинники та ін.), розробляти ІТ

для безпеки мобільних даних, а в наступні роки бути готовими створювати і впроваджувати принципово нові технології, які прийдуть на зміну нинішнім мобільним. Статистичні дані [4] вказують, що кількість випускників групи ІТ-спеціальностей в Україні у 2020 р. становила 12347 осіб ОС «Бакалавр», 6182 – ОС «Магістр». Портали вакансій України вказують на щомісячну потребу ІТ-спеціалістів до 10 тис. осіб. Згідно даних порталу Rabota.UA [5], серед усіх вакансій за жовтень 2021 р., пов'язаних з розробкою програмного забезпечення (3684), кількість вакансій розробки для мобільних платформ становить 284, або майже 8%.

У контексті мобільних розробок варто відзначити стабільний уже кілька років розподіл ринку між Android (70-75%) і iOS (25-30%) [6]. Такий розподіл визначає преференції при виборі програмних інструментів, а також зміст відповідних навчальних дисциплін. В останні роки в індустрії розробки мобільного ПЗ сформувалося кілька підходів [7]:

- нативна розробка для ОС Android (мови Java, Kotlin та ін.);
- нативна розробка для ОС iOS (Swift, Objective-C);
- розробка кросплатформного мобільного ПЗ (C#, JavaScript та ін.).

Останнє передбачає використання високофункціональних інструментів у вигляді фреймворків, які переважно базуються на JavaScript. Найбільш популярними залишаються [8] React Native, Flutter, Xamarin, Ionic, PhoneGap та ін. Завдяки таким потужним інструментам знижується вартість і час розробки, а зростання потужності мобільних пристроїв нівелює недоліки кросплатформних застосунків, пов'язані зі швидкодією. З огляду на це, зростає важливість опанування студентами і розробниками відповідних інструментів розробки, серед яких на перше місце виходять мови, які базуються на інтерпретації (JavaScript). Це необхідно враховувати при оновленні змісту відповідних дисциплін.

Освітня програма «Інформатика» для студентів ОС «Бакалавр» Київського національного університету імені Тараса Шевченка містить вибіркові освітні компоненти, пов'язані з мобільними технологіями. Їх викладання забезпечує кафедра теорії та технологіїї програмування.

Зміст дисципліни «Розробка ПЗ під мобільні платформи» оновлюється з урахуванням досвіду закордонних ЗВО, аналізу ринку

вакансій, та опитування студентів. Курс охоплює тематику: принципи роботи мобільних мереж, архітектура мобільних пристроїв та мобільних ОС, інструменти розробки для Android і iOS, програмні засоби створення ПЗ, у т.ч. для проєктування користувацького інтерфейсу, реалізації бізнес-логіки, роботи з файлами і БД, з адресною книгою, API для 2D-графіки і мультимедіа, використання геопросторових сервісів та зв'язку з віддаленими серверами. Курс передбачає неформальну освіту, можливості об'єднання студентів у малі групи для виконання групових проєктів та гнучкий вибір предметних областей для виконання практичних завдань.

У кінці семестру серед студентів традиційно проводиться анкетування з метою вдосконалення навчального курсу у наступному році. У 2021 р. в анкету для студентів дисципліни «Розробка ПЗ під мобільні платформи» було додано блок питань щодо бачення перспективності технолоній мобільної розробки. Всього було опитано 26 студентів.

Понад 60% опитаних розглядають для себе мобільну розробку як можливий професійний напрям діяльності. Серед перспективних мов програмування для мобільних платформ хороші перспективи було відзначено лише для Java, Kotlin, Swift, а також C# меншою мірою (рис. 3). Загалом це узгоджується з популярністю цих мов у світі.

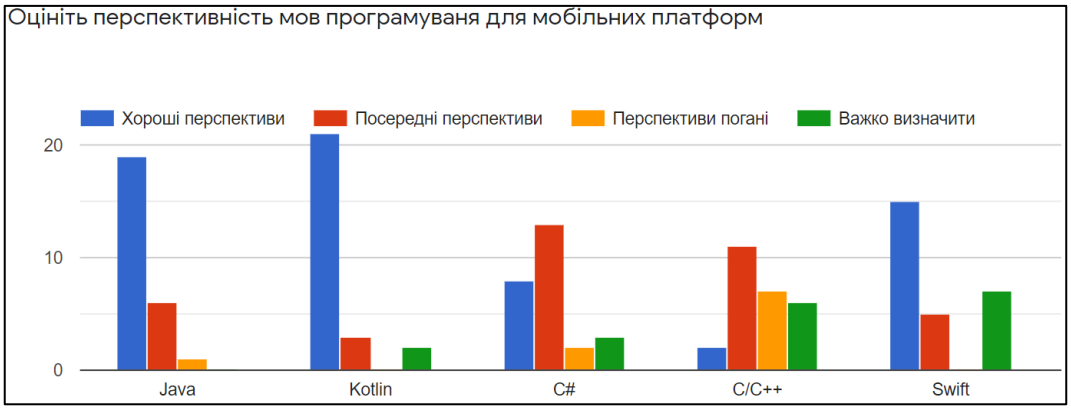


Рисунок 3. Оцінка студентами перспектив мобільних мов програмування (фрагмент діаграми)

Серед факторів, які сприяють подальшому розвитку мобільних технологій, здобувачі вказали: швидкісний Інтернет; зростання часу, який проводить користувач з мобільними пристроями; пандемія; потреба у мобільному спілкуванні; поширення мобільних електронних платежів; достатня потужність мобільних пристроїв, щоб вони взяли на себе частину задач для десктопів/ноутбуків; зростаючий ринок і висока конкуренція; зростаючий попит на мобільні програми; люди звикли до використання мобільних пристроїв і готові періодично купляти нові; смартфони вже є невід'ємним інструментом у повсякденному житті тощо.

Ще одне питання анкети – «Які, на вашу думку, важливі зміни відбудуться в індустрії розробки для мобільних платформ у наступні роки?». Серед відповідей: масовий перехід на мережі нового покоління (5G і наступних); збільшуватимуться розміри екрану, що потягне зміни в ергономіці та інтерактивних можливостях; деякі студенти вважають навпаки, що розміри не будуть збільшуватися з тих ж ергономічних причин; будуть покращуватися характеристики мобільних пристроїв аналогічно до еволюції персональних комп'ютерів останніх років; поширення інструментів одночасної розробки для Web, Android, iOS та настільних ОС; широке використання доповненої реальності; використання штучного інтелекту у мобільному ПЗ; можлива поява нових ОС (імовірно, виробництва КНР); подальше поширення хмарних обчислень і для мобільних пристроїв (та одночасно зменшення важливості розробки саме для мобільних ОС); упровадження нових інженерних рішень (гучкі екрани та подібн.).

Результати анкетування показали прихильність студентів до індивідуальних проєктів (46%), групових (15%) та їх поєднання (31%) (рис. 4).

Практично для всіх видів робіт оцінка складності та трудоємкості має вигляд нормального розподілу.

Серед соціально-орієнтованих навичок, вдосконалити які допомогло навчання на дисципліні, студенти відзначили: планування часу (дедлайни), уміння публічно захищати результати (захист усіх видів робіт), уміння писати звіт з виконання проєкту. Ряд пропозицій (наприклад, прочитати

раніше лекцію про особливості користувацького інтерфейсу для мобільних пристроях) враховано при оновленні дисципліни.

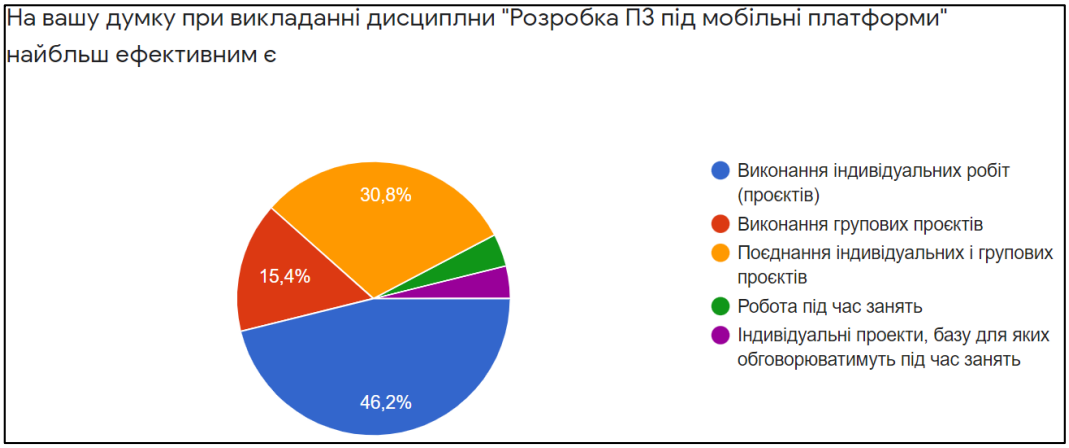


Рисунок 4. Оцінка студентами ефективності використання проєктного підходу на дисципліні «Розробка ПЗ під мобільні платформи»

Висновки. Звіти провідних аналітичних компаній показують збереження трендів зростання попиту на мобільний цифровий контент (у т.ч. ПЗ) та подальшу соціалізацію саме через мобільні комунікації.

Очевидним є стабільний попит на ІТ-фахівців і, зокрема, у галузі розробки для мобільних платформ. Українські ЗВО випускають на ринок щороку майже 20 тис. ІТ-фахівців, але цього недостатньо для задоволення запитів ринку, у т.ч. сегменту мобільних розробок.

Незважаючи на відносно невисокий відсоток ІТ-вакансій для мобільної розробки, більшість студентів вважають ці технології перспективними, а відповідні дисципліни необхідними для вивчення. Бачення студентів загалом збігається зі світовими трендами щодо перспектив окремих платформ, інструментів розробки та руху в бік кросплатформної розробки і використання відповідних фреймворків.

Список використаних джерел

1. O'Reilly, T. What Is Web 2.0. O'Reilly Network. Archived (2005). – [Електронний ресурс]. Режим доступу: https://www.oreilly.com/pub/a/web2/archive/what-is-web-20.html. [30.10.2021].
2. DIGITAL 2021: GLOBAL OVERVIEW REPORT, 2021. – [Електронний ресурс]. Режим доступу: https://datareportal.com/reports/digital-2021-global-overview-report. [30.10.2021]
3. Desktop vs Mobile vs Tablet Market Share Worldwide. [Online]. Available: https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/worldwide/#yearly-2009-2021 [03.11.2021].
4. Вища та фахова передвища освіта в Україні у 2020 році // Державна служба статистики України. – [Електронний ресурс]. Режим доступу: http://www.ukrstat.gov.ua/operativ/operativ2005/osv_rik/osv_u/vysh_osvita/arch_vysh_osvita.htm. [30.10.2021]
5. Портал RABOTA.UA – [Електронний ресурс]. Режим доступу: https://rabota.ua/ [03.11.2021]
6. Mobile Operating System Market Share Worldwide. – [Електронний ресурс]. Режим доступу: https://gs.statcounter.com/os-market-share/mobile/worldwide. [05.05. 2021]
7. Ткаченко О.М. Інструментальні перспективи мобільної розробки / Матеріали IX Міжнародної науково-практичної конференції "Глобальні та регіональні проблеми інформатизації в суспільстві і природокористуванні 2021", м. Київ, 13-14 травня 2021 р., НУБіП України. - К.: НУБіП України, 2021 - С.91-93.
8. Top 14 Best Mobile App Development Frameworks in 2021. – [Електронний ресурс]. Режим доступу: https://www.makeitinua.com/posts/top-12-best-mobile-app-development-frameworks-in-2020. [05.05.2021]

РОЗДІЛ 3. АНАЛІЗ ТА МЕТОДИЧНІ АСПЕКТИ ІНТЕГРАЦІЇ НАВЧАЛЬНИХ КУРСІВ КАФЕДРИ

Омельчук Людмила Леонідівна, к.ф.-м.н, доцент
Русіна Наталія Геннадіївна, к.п.н.

АНАЛІЗ ОСВІТНЬО-ПРОФЕСІЙНОЇ ПРОГРАМИ «ІНФОРМАТИКА» В РОЗРІЗІ ОБОВ'ЯЗКОВИХ ДИСЦИПЛІН ВИКЛАДАННЯ ЯКИХ ЗАБЕЗПЕЧУЄТЬСЯ КАФЕДРОЮ «ТЕОРІЯ ТА ТЕХНОЛОГІЯ ПРОГРАМУВАННЯ»

У роботі наведено порівняльний аналіз обов'язкових дисциплін освітньо-професійної програми «Інформатика» першого (бакалаврського) рівня вищої освіти галузі знань 12 «Інформаційні технології», спеціальності 122 «Комп'ютерні науки», викладання яких забезпечується кафедрою теорії та технології програмування факультету комп'ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка з дисциплінами освітньо-професійних програм того ж рівня та спеціальності інших закладів вищої освіти. Аналіз здійснювався на основі затвердженого стандарту першого (бакалаврського) рівня вищої освіти за спеціальністю 122 «Комп'ютерні науки».

Ключові слова: комп'ютерні науки, інформатика, база даних, освітня програма, теорія та технологія програмування.

We presents a comparative analysis of compulsory disciplines of the educational-professional program "Informatics" of the first (bachelor's) level of higher education in the field of knowledge 12 "Information Technology", specialty 122 "Computer Science". Studied on the Department of Theory and Technology of Programming of Faculty of Computer Science and Cybernetics of Taras Shevchenko National University of Kyiv with disciplines of educational and professional programs of the same level and specialties of other institutions of higher education. The analysis was carried out on the basis of the approved standard of the first (bachelor's) level of higher education in the specialty 122 "Computer Science".

Keywords: computer science, informatics, database, educational program, theory and technology of programming.

Моніторинг та переосмислення освітніх програм за спеціальністю 122 «Комп'ютерні науки», повинен враховувати вимоги Закону України «Про вищу освіту» [1] та Стандарту першого (бакалаврського) рівня вищої освіти за спеціальністю 122 «Комп'ютерні науки», затвердженого Наказом Міністерства освіти і науки України за № 962 від 10.07.2019 році [2]. Важливим елементом на шляху до покращення освітньо-професійної програми (ОПП) «Інформатика» першого (бакалаврського) рівня вищої освіти за спеціальністю 122 «Комп'ютерні науки» [3], що реалізується факультетом комп'ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка є проведення її порівняльного аналізу з аналогічними програмами інших закладів вищої освіти (ЗВО). У зв'язку з вимогами прозорості, що висуваються Законом України «Про вищу освіту» [1] здійснити такий порівняльний аналіз виявляється можливим, адже освітні програми (ОП), що реалізуються різними ЗВО оприлюднюються для вільного доступу на їх офіційних сайтах. Аналіз ОП проведено в розрізі порівняння обов'язкових освітніх компонент ОПП «Інформатика» та забезпечуваних ними програмних результатів навчання й компетентностей з обов'язковими освітніми компонентами інших ОПП. Акцент зроблено на дисциплінах, викладання яких забезпечується кафедрою теорії та технології програмування.

Аналіз останніх досліджень і публікацій

Дослідження ОП у ЗВО України доцільно розглядати в контексті нових стандартів вищої освіти, більшість з яких вже прийняті та затверджені МОН України, а окремі перебувають на стадії обговорення. Проте проблема аналізу саме обов'язкових дисциплін освітньо-професійної програми «Інформатика» першого (бакалаврського) рівня вищої освіти галузі знань 12 «Інформаційні технології», спеціальності 122 «Комп'ютерні науки» є відносно новою та не до кінця дослідженою. В публікаціях [4-6] було проведено порівняння українського стандарту освітньо-професійної підготовки та міжнародного стандарту Computer Science'2013 й надано

пропозиції для покращення українського освітнього стандарту. У роботах [7, 8] описано теоретичне підґрунтя для порівняльного аналізу.

Порівняльний аналіз освітньої програми «Інформатика» з іншими освітніми програмами спеціальності в розрізі обов'язкових дисциплін

Для дослідження авторами відібрано наступні дисципліни з ОПП «Інформатика» [3], що реалізується на факультеті комп'ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка (Інф_КНУТШ):

- Математична логіка;
- Теорія алгоритмів;
- Об'єктно-орієнтоване програмування;
- Інструментальні середовища та технології програмування;
- Системне програмування;
- Теорія програмування;
- Інформаційні технології.

В таблиці 1 представлено розподіл загальних та спеціальних компетентностей за обов'язковими дисциплінами викладення яких забезпечується кафедрою теорії та технології програмування факультету комп'ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка.

Таблиця 1. Розподіл компетентностей за дисциплінами

Компетентності (загальні (ЗК), спеціальні (СК))	Програмні результати навчання (ПРН)
Математична логіка	
СК1, СК3	ПРН1, ПРН5
Теорія алгоритмів	
СК1, СК3	ПРН1, ПРН5
Об'єктно-орієнтоване програмування	
ЗК2, ЗК12, СК8	ПРН9, ПРН15

Інструментальні середовища та технології програмування	
ЗК2, ЗК12, СК8, СК9, СК10	ПРН9, ПРН11, ПРН15
Системне програмування	
ЗК10, ЗК12, СК8, СК12, СК16	ПРН13, ПРН14
Теорія програмування	
ЗК1, СК1, СК8	ПРН1, ПРН5
Інформаційні технології	
ЗК2, ЗК9, СК8, СК9, СК10, СК15, СК16	ПРН9, ПРН11, ПРН15

В зазначеній таблиці, відповідно до ОПП [3] та прийнятого Стандарту [2]:

ЗК2. Здатність застосовувати знання у практичних ситуаціях.

ЗК9. Здатність працювати в команді.

ЗК10. Здатність бути критичним і самокритичним.

ЗК12. Здатність оцінювати та забезпечувати якість виконуваних робіт.

СК1. Здатність до математичного формулювання та досліджування неперервних та дискретних математичних моделей, обґрунтовування вибору методів і підходів для розв’язування теоретичних і прикладних задач у галузі комп’ютерних наук, аналізу та інтерпретування.

СК3. Здатність до логічного мислення, побудови логічних висновків, використання формальних мов і моделей алгоритмічних обчислень, проектування, розроблення й аналізу алгоритмів, оцінювання їх ефективності та складності, розв’язності та нерозв’язності алгоритмічних проблем для адекватного моделювання предметних областей і створення програмних та інформаційних систем.

СК8. Здатність проектувати та розробляти програмне забезпечення із застосуванням різних парадигм програмування: узагальненого, об’єктно-орієнтованого, функціонального, логічного, з відповідними моделями, методами й алгоритмами обчислень, структурами даних і механізмами управління.

СК9. Здатність реалізувати багаторівневу обчислювальну модель на основі архітектури клієнт-сервер, включаючи бази даних, знань і сховища даних, , виконувати розподілену обробку великих наборів даних на кластерах стандартних серверів для забезпечення обчислювальних потреб користувачів, у тому числі на хмарних сервісах.

СК10. Здатність застосовувати методології, технології та інструментальні засоби для управління процесами життєвого циклу інформаційних і програмних систем, продуктів і сервісів інформаційних технологій відповідно до вимог замовника.

СК12. Здатність забезпечити організацію обчислювальних процесів в інформаційних системах різного призначення з урахуванням архітектури, конфігурування, показників результативності функціонування операційних систем і системного програмного забезпечення.

СК15. Здатність до аналізу та функціонального моделювання бізнес-процесів, побудови та практичного застосування функціональних моделей організаційно-економічних і виробничо-технічних систем, методів оцінювання ризиків їх проектування.

СК16. Здатність реалізовувати високопродуктивні обчислення на основі хмарних сервісів і технологій, паралельних і розподілених обчислень при розробці й експлуатації розподілених систем паралельної обробки інформації.

Представлені співвідношення деяких дисциплін, що обов’язковими в Інф_КНУТШ [3] та їх відсотку подібності за компетентностями (таблиці 2) і за програмними результатами навчання (ПРН) (таблиці 3) до дисциплін інших ОПП того ж рівня й спеціальності.

У дослідженні розглянуто наступні ОПП:

Кн_КНУТШ – ОПП «Комп’ютерні науки», факультет інформаційних технологій Київського національного університету імені Тараса Шевченка [9].

Інф_ХНУ – ОПП «Інформатика», факультет математики і інформатики Харківського національного університету імені В.Н. Каразіна (ХНУ імені В.Н. Каразіна) [10];

Кн_ЗНУ – ОПП «Комп’ютерні науки», математичний факультет Запорізького національного університету [11];

Кн_ВНТУ – ОПП «Комп’ютерні науки», факультет інформаційних технологій та комп’ютерної інженерії Вінницького національного технічного університету [12];

Інф_УНУ – ОПП «Інформатика», факультет інформаційних технологій Державного вищого навчального закладу «Ужгородський національний університет» [13].

Таблиця 2. Фрагмент подібності дисциплін Інф_КНУТШ за компетентностями

ОПП	Назва дисципліни	Відсоток подібності
Математична логіка		
Кн_КНУТШ	Дискретні структури	50%
Інф_ХНУ	Елементи математичної логіки, елементарної та дискретної математики	70%
Кн_ЗНУ	Теорія ймовірності та математична статистика	37,5%
Кн_ВНТУ	Дискретна математика	61,11%
Інф_УНУ	Алгоритми і структури даних	66,67%
Теорія алгоритмів		
Інф_ХНУ	Елементи математичної логіки, елементарної та дискретної математики	70%
Кн_ЗНУ	Процедурне програмування	32,14%
Кн_ВНТУ	Теорія алгоритмів	64,28%
Інф_УНУ	Дискретна математика та теорія алгоритмів	64,29%
Кн_КНУТШ	Проектування та аналіз алгоритмів	37,5%
Об’єктно-орієнтоване програмування		
Кн_КНУТШ	Об'єктно-орієнтоване програмування	41,67%
Інф_ХНУ	Шаблони об'єктно-орієнтованого програмування	53,33%
Інф_УНУ	Управління ІТ-проектами	60%

Кн_ВНТУ	Алгоритмізація та програмування	25%
Кн_ЗНУ	Комп’ютерна графіка	45,83%
Інструментальні середовища та технології програмування		
Кн_КНУТШ	Технологія створення програмних продуктів	60%
Кн_ВНТУ	Теорія алгоритмів	17,14%
Інф_УНУ	Технологія програмування та створення програмних продуктів	64,71%
Кн_ЗНУ	Об'єктно-орієнтоване програмування	47,61%
Системне програмування		
Кн_КНУТШ	Проектування та аналіз алгоритмів	45%
Інф_ХНУ	Розробка компіляторів для предметно-орієнт. мов	46,66%
Кн_ЗНУ	Архітектура обчислювальних систем	41,67%
Інф_УНУ	Об'єктно-орієнтоване програмування	29%
Кн_ВНТУ	Крос-платформне програмування	45%
Теорія програмування		
Інф_ХНУ	Математична логіка і мова Prolog	68,75%
Кн_ЗНУ	Процедурне програмування	47,61%
Кн_ВНТУ	Математичні методи дослідження операцій	50%
Інф_ХНУ	Паралельні та розподілені обчислення	29,17%
Кн_КНУТШ	Алгоритмізація та програмування	29,17%
Інформаційні технології		
Кн_КНУТШ	Технологія створення програмних продуктів	51,43%
Інф_ХНУ	Розробка компіляторів для предметно-орієнт. мов	50,79%
Кн_ЗНУ	Об'єктно-орієнтоване програмування	28,57%
Кн_ВНТУ	Організація баз даних та знань	46,75%

Наведене в таблиці 2 співставлення дає наглядне уявлення про співвідношення змісту деяких обов’язкових освітніх компонент у різних ОПП за спільною спеціальністю відносно загальних та фахових компетентностей, визначених стандартом вищої освіти.

Таблиця 3. Фрагмент подібності дисциплін Інф_КНУТШ за ПРН

ОПП	Назва дисципліни	Відсоток подібності
Математична логіка		
Кн_КНУТШ	Проектування та аналіз алгоритмів	75%
Інф_ХНУ	Елементи алгебри та теорії чисел	50%
Кн_ВНТУ	Управління ІТ-проектами	41,67%
Кн_ЗНУ	Алгоритми та структури даних	75%
Теорія алгоритмів		
Кн_КНУТШ	Проектування та аналіз алгоритмів	75%
Інф_ХНУ	Методи оптимізації і дослідження операцій	32,14%
Кн_ВНТУ	Дискретна математика та теорія алгоритмів	100%
Інф_УНУ	Теорія прийняття рішень	35%
Об’єктно-орієнтоване програмування		
Кн_КНУТШ	Об'єктно-орієнтоване програмування	100%
Інф_УНУ	Технологія програмування та створення програмних продуктів	61,11%
Інф_ХНУ	Методи розробки інтерфейсу користувача	37,5%
Кн_ВНТУ	Об’єктно-орієнтоване програмування	37,5%
Інструментальні середовища та технології програмування		
Кн_КНУТШ	Технологія створення програмних продуктів	66,67%
Інф_УНУ	Технологія програмування та створення програмних продуктів	66,68%
Інф_ХНУ	Методи розробки інтерфейсу користувача	29,17%
Кн_ВНТУ	Технології комп’ютерного проектування	33,33%
Кн_ЗНУ	Організація та обробка електронної інформації	41,67%
Системне програмування		
Кн_КНУТШ	Комп'ютерні мережі	83,33%
Інф_ХНУ	Методи розробки інтерфейсу користувача	75%
Кн_ВНТУ	Комп’ютерні мережі та хмарні технології	50%
Кн_ЗНУ	Крос-платформне програмування	37,5%
Інф_УНУ	Об'єктно-орієнтоване програмування	31,25%

Теорія програмування		
Кн_КНУТШ	Алгоритмізація та програмування	50%
Інф_ХНУ	Елементи математичної логіки, елементарної та дискретної математики	50%
Кн_ЗНУ	Алгоритми та структури даних	75%
Кн_ВНТУ	Крос-платформне програмування	37,5%
Кн_КНУТШ	Алгоритмізація та програмування	64,29%
Інф_УНУ	Математичні методи дослідження операцій	37,5%
Інформаційні технології		
Кн_КНУТШ	Проектування інформаційних систем	66,67%
Інф_ХНУ	Інформаційні мережі	33,33%
Кн_ВНТУ	Технології кодування та захисту інформації	41,67%
Інф_УНУ	Комп'ютерні мережі	33,33%
Кн_ЗНУ	Алгоритми та структури даних	29,17%

Наведене в таблиці 3 співставлення дає наглядне уявлення про співвідношення змісту обов'язкових освітніх компонент у різних ОПП за спільною спеціальністю відносно ПРН, визначених стандартом вищої освіти [2].

Висновки. Проведений порівняльний аналіз обов'язкових дисциплін освітньо-професійної програми «Інформатика» першого (бакалаврського) рівня вищої освіти галузі знань 12 «Інформаційні технології», спеціальності 122 «Комп'ютерні науки», викладання яких забезпечується кафедрою теорії та технології програмування факультету комп'ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка з дисциплінами освітньо-професійних програм того ж рівня та спеціальності інших закладів вищої освіти України показав: коректність підібраних загальних та спеціальних компетентностей і результатів навчання до розглянутих дисциплін.

Список використаних джерел

1. Закон України “Про вищу освіту” від 01.07.2014 р. №1556-VII // Відомості Верховної Ради України. – 2014. – № 37-38. – Ст. 2004 [Електронний ресурс]. – 2014. – Режим доступу до ресурсу: https://zakon.rada.gov.ua/laws/show/1556-18#Text
2. Наказ МОН України «Про затвердження стандарт вищої освіти за спеціальністю 122 «Комп’ютерні науки» для першого (бакалаврського) рівня вищої освіти» [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://mon.gov.ua/storage/app/media/vishcha-osvita/zatverdzeni%20standarty/2019/07/12/122-kompyuterni-nauki-bakalavr.pdf.
3. Київський національний університету імені Тараса Шевченка. Освітньо-професійна програма "Інформатика". Спеціальність 122 Комп’ютерні науки [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: http://csc.knu.ua/media/filer_public/09/f4/09f46516-6c61-46a5-ae81-f34b7d673499/opp_122_bac_2019_1.pdf.
4. Омельчук Л.Л. Порівняльний аналіз українського стандарту освітньо-професійної підготовки з інформатики та міжнародних освітніх стандартів / Л.Л. Омельчук // Вісник Київського ун-ту. Серія: фіз.-мат. науки. – 2013. – Вип. 4. – С. 138–149.
5. Омельчук Л.Л. Порівняльний аналіз українського стандарту освітньо-професійної підготовки з інформатики та Computer Science’2013 / Л.Л. Омельчук // Вісник Київського ун-ту. Серія: фіз.-мат. науки. – 2013. – Вип. 2. – С. 216–227.
6. Omelchuk L. Development of the ICT-standard of Higher Education in Ukraine within the Framework of European Requirements / L. Omelchuk , N. Rusina, O. Shyshatska // Proc. 15th Int. Conf. on ICT in Education, Research and Industrial Applications. Integration, Harmonization and Knowledge Transfer (ICTERI 2019). Volume I: Main Conference. – Kherson, Ukraine, June 12-15 (2019). PP. 262-273.
7. Омельчук Л.Л., Русіна Н.Г. Автоматизований аналіз освітньо-професійної програми «Інформатика», що реалізується на факультеті комп’ютерних наук та кібернетики, з програмами інших закладів вищої освіти за

спеціальністю 122 «Комп’ютерні науки» / Л.Л.Омельчук, Н.Г.Русіна // Вісник Київського ун-ту. Серія: фіз.-мат. науки. – 2020. – Вип. 4. – С. 49-62.
8. Омельчук Л.Л., Русіна Н.Г. Аналіз освітньо-професійних програм за спеціальністю 122 Комп’ютерні науки в розрізі програмних результатів навчання/ Л.Л.Омельчук, Н.Г.Русіна // Вісник Київського ун-ту. Серія: фіз.-мат. науки. – 2021. – Вип. 1. – С. 89–101.
9. Освітньо-професійна програма «Комп'ютерні науки». Спеціальність 122 Комп’ютерні науки. Київський національний університету імені Тараса Шевченка. [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: http://fit.univ.kiev.ua/wp-content/uploads/2019/07/ОПП-Компютерні-науки-122.pdf.
10. Освітньо-професійна-програма "Інформатика". Спеціальність 122 Комп’ютерні науки. Харківський національний університет імені В.Н.Каразіна. [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: http://math.univer.kharkov.ua/StandProg/OPPInformBak.pdf.
11. Освітньо-професійна програма «Комп'ютерні науки». Спеціальність 122 Комп’ютерні науки. Запорізький національний університет. [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: https://www.znu.edu.ua/opp2020/bak/math/op_122_komp_nauki_bak.pdf.
12. Освітньо-професійна програма «Комп'ютерні науки». Спеціальність 122 Комп’ютерні науки. Вінницький національний технічний університет. [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: http://vntu.edu.ua/uk/information-forenrollee/progmagbak.html.
13. Освітньо-професійна програма "Інформатика". Спеціальність 122 Комп’ютерні науки. Державний вищий навчальний заклад «Ужгородський національний університет». [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: https://www. uzhnu.edu.ua/ uk/infocentre/15068.

Панченко Тарас Володимирович, к.ф.-м.н., доцент

ХАКАТОН – НОВІТНІЙ ПІДХІД ДО НАВЧАННЯ СТУДЕНТІВ ЧЕРЕЗ ПРАКТИКУ

Розглянуто роль хакатонів як новітнього проєктного підходу до навчання студентів через розв'язання практичних задач. Досліджено переваги та роль хакатонів, а також особливості організації подібних заходів. Наведено тези та цитати учасників, експертів хакатонів та представників ІТ-компаній. Обґрунтовано доцільність та користь від участі у хакатонах.

Ключові слова: хакатон, навчання студентів, практика впровадження, проєктний підхід, ІТ-навчання, навички, м'які навички.

The role of hackathons as a new project-oriented approach to student learning through practical problem solving is considered here. The advantages and role of hackathons, as well as the specifics of the organization of such events are studied. Theses and quotes of participants, hackathon experts and representatives of IT companies are given. The reasonability and benefits of participating in hackathons are explained.

Keywords: hackathon, student education, implementation practice, project-oriented approach, IT education, skills, soft skills.

Хакатони [1] – це новітній підхід до навчання через практику, коли за короткий час вдається не лише засвоїти суттєвий обсяг знань, але й закріпити навички на практиці, "прокачати" свої навички. Ефект від такого навчання посилюється реалістичними (або наближеними до реальності) спеціально підготовленими задачами, які виносяться на цю подію.

Але чи не найголовнішою відмінністю від інших форм навчання є саме формат проведення – у вигляді марафону (тривалого забігу) команди учасників, яким треба досягти найкращого прогресу в розв'язанні поставлених задач та продемонструвати результат у фіналі хакатону. З одного боку, учасники хакатону занурюються в атмосферу продуктивності, що нагадує роботу над реальними проєктами та відтворює атмосферу стартапу, коли проривна ідея, гарно реалізована та якісно подана, може змінити світ. А з іншого – учасники команди вимушені:

- ефективно комунікувати, аргументувати, доводити свою позицію,
- налагоджувати зв’язки,
- планувати та розподіляти роботу,
- управляти своїм часом,
- швидко знаходити нестандартні рішення та впроваджувати їх,
- підготувати презентацію та демонстрацію проєкту,
- ефективно доносити свої думки – як членам команди, так і членам журі у фіналі,
- презентувати рішення та переконати в його перевагах.

Тобто розвивати м’які навички (soft skills) з метою перемогти. Хоч це і не головна ціль хакатону як перегонів. Так, перші хакатони взагалі мали за мету стимулювати розробку нових технологій та продуктових рішень.

Отже, у такій жвавій та часто бурхливій атмосфері, де мотивація учасників зашкалює, народжуються цікаві ідеї та проривні рішення.

На відміну від олімпіад, де зазвичай точне оптимальне рішення задач, що ставляться перед учасниками змагань, відоме наперед – на хакатон виносяться задачі відкритого типу, тобто проблеми, для яких поки що немає ідеального розв’язку (що задовольняв би усім критеріям), або відомі розв’язки задовольняють не всім критеріям. В такий спосіб підживлюється творча складова хакатону як конкурсу чи змагання, надаючи учасникам додаткові можливості для пошуків та самореалізації.

Важливою складовою таких змагань є взаємодія з експертами галузі. По суті, хакатон створює майданчик для ефективного обміну досвідом між більш досвідченими фахівцями галузі (як правило, IT), які виступають в якості менторів команд та допомагають їм досягти максимум результативності протягом хакатону, та учасниками, спраглими до знань. Учасники, які хочуть дійсно здобути нові знання та розвинути навички, об’єднавшись у команди, намагаються розв’язати поставлені задачі у тісній комунікації – більше того, у співпраці – з експертами. Відбувається магія – і у дискусіях народжуються нові ідеї та рішення актуальних проблем. Досвід експертів, змішаний з сміливістю та незашореністю думки більш молодих учасників, справді творить дива.

Значення хакатону в сучасних умовах, де співіснують два великі табори: традиційна вища освіта та освіта, яку все частіше сьогодні пропонують великі ІТ-компанії, важко переоцінити. Хакатон дозволяє поєднати ці два підходи, урізноманітнюючи навчальний процес в ЗВО та даючи можливість доторкнутися до сучасних труднощів, з якими стикаються ІТ-компанії. Ось, що на одному з хакатонів розповів Дмитро Василенко, Senior Software Engineer у GlobalLogic "*Хакатон – це чудова можливість прокачати свої навички. І якраз ті, хто не виграли, досягли найбільшого – адже вони дійсно навчились: дізнались, які методи пропустили, та на що вони не звернули увагу. Це – чудова можливість вивчити щось нове. І дуже добре, що учасники та учасниці вибрали цю можливість навчитись замість "нічого не робити". Мене тішить, що велика кількість мотивованих людей демонструють величезну кількість різних цікавих рішень – це вражає! У мене, як у члена журі – лише позитивні враження від хакатону. Все на високому рівні. І така велика кількість учасників – надихає!*" [2]

Хакатони, з іншого боку, потребують надзвичайно багато "закулісної" роботи з підготовки та проведення, аби подія проходила на високому рівні і забезпечувала умови для продуктивної роботи та якнайкращого розкриття потенціалу учасників. Так, за лаштунками проводиться кількомісячна робота з підготовки та планування до дрібниць, аби вже під час події учасники могли зосередитись лише на роботі, максимізувати концентрацію та відкинути усі супутні питання та проблеми – адже організатори про це подали. Багато хакатонів в Україні проводиться громадською організацією Hackathon Expert. Вона цілеспрямовано займається популяризацією *хакатонів* серед студентів ІТ-спеціальностей ЗВО та просвітництвом серед університетів щодо важливості таких заходів з 2016 року. За цей час було проведено більше 15 хакатонів, в яких встигло взяли участь щонайменше 1500 студентів.

Робота по підготовці до хакатону розпочинається. Оскільки цей етап, як правило, залишається поза увагою, то хотілось би привідкрити завісу щодо цієї важливої складової хакатону – процесу підготовки й планування. Будь-який хакатон розпочинається з пошуку теми хакатону, наприклад

Machine Learning (ML), Computer Vision (CV), Artificial Intelligence (AI), NLP, Cryptocurrency. Після обрання теми хакатону Hackathon Expert розпочинає пошук партнерів, які б підтримали подію. Підтримка буває різною: хтось пропонує своїх кращих працівників у якості менторів та експертів для хакатону, хтось надає хмарні обчислювальні потужності та платформу для навчання моделей, хтось цінні призи, та головне – це безцінний досвід. Досвід, що отримує кожен учасник хакатону за час його проведення. Студенти не тільки намагаються вирішити актуальні проблеми, розв'язувати складні задачі, але й перевірити себе на витривалість. Ми дуже чудово знаємо ціну витривалості. Ні для кого напевно не секрет, що більшість стартапів перестають існувати протягом першого року, так би мовити за півкроку до успіху. Хакатон дає можливість (хоч і опосередковано і дуже стисло в часі) пережити, відчути на собі цей досвід – досвід маленького стартапу, за короткий термін.

Після обрання теми хакатону команда Hackathon Expert розпочинає пошук фахівців з релевантною експертизою, які є амбасадорами та інфлуенсерами в своїй сфері, які б могли допомогти не тільки прокачати hard-skills учасників а й *надихнути, дати поради, котрі були б корисними і після завершення хакатону.* Це дуже тривалий процес, який може зайняти багато часу. Причому, якщо в “до-ковідні” часи присутність учасників і експертів на місці проведення було безальтернативною вимогою, то зараз кордони зникли. Участь онлайн вже є де-факто стандартом, принаймні змішана або виключно онлайн участь дедалі стають все більш поширеними. Та і зручними. Карантин відкрив нові можливості для хакатону і переваги організації заходів онлайн. Ви можете не тільки “дотягнутись” до світової експертизи і залучити фахівців з будь-якого куточку світу на хакатон, але й організувати більш гнучкий графік присутності й зустрічей. Особливо ж цінно, що до панелі експертів все частіше вдається залучити провідних закордонних фахівців. Так, на хакатоні на початку 2022 р. з 10 менторів лише 4 перебували в Україні. Час усіх учасників треба цінувати, і така гнучкість, зрозуміло, покладає більше навантаження та відповідальність на команду організаторів.

Пошук локації. Потрібно знайти підходяще місце, яке б вмістило значну кількість учасників, з можливістю організувати комфортні умови для учасників та зони для їх роботи і відпочинку. Яскравим прикладом пошуку локації був досвід організації та проведення Kyiv Computer Vision Hackathon в центрі Києва в «Креативному кварталі» (Creative Quarter). Було зібрано 50 учасників, 5 досвідчених експертів у галузях комп'ютерного зору, 3D та машинного навчання. Вдалося зібрати учасників з різних частин України, в тому числі студентів провідних навчальних закладів – КНУ імені Тараса Шевченка, НТУУ КПІ імені Ігоря Сікорського, Українського Католицького Університету, й інших ВНЗ. Глибина ідей та рівень готовності проектів, запропонованих учасниками, вразила Віктора Сдобнікова, Артема Чернодуба, Валерія Кригіна, Віктора Сахарчука та Дмитра Яковкіна, які були експертами – знаними фахівцями у своїх галузях. Це явно було помітно з високих оцінок більш ніж половині команд на фінальних презентаціях. Цей хакатон був дійсно ефективним з чудовими результатами, які продемонстрували всі командами – на світ народились нові проєкти, деякі з них у подальшому були втіленими у життя (у співпраці з компаніями-партнерами хакатону), а експерти були задоволені досягнутим прогресом. Разом з тим, карантин і тут змінює правила гри. Тепер на онлайн-хакатонах можна заощадити на оренді.

По-друге, проведення хакатону онлайн розширює географію події. Дає можливість взяти участь абсолютно всім бажаючим незалежно від місця перебування. Збільшуючи таким чином потенційну кількість команд. До речі, саме завдяки проведенню хакатонів онлайн, вже декілька разів поспіль, на хакатоні були представлені команди-учасники з різних країн. Так, з 2020 року на хакатонах Hackathon Expert часто беруть участь команди з Чехії, Німеччини, Італії, інших країн. В свою чергу це мотивує всіх учасників хакатону, оскільки під час таких хакатонів як ніколи відчувається єднання зі світом, обмін вміннями, порівняння рівня знань своїх однолітків за кордоном.

По-третє, час. Коли хакатон триває 2 дні, вирішити питання з тим, де подрімати, відновити сили не стоїть гостро – дехто взагалі не спить, коли мова йде про офлайн хакатон. А що роботи, коли мова йде про 5-денний

хакатон, наприклад? Тоді технічні можливості як організаторів (реалізувати задумане), так і учасників (дістатись фіналу) різко скорочуються.

ГО “Hackathon Expert ” одними з перших в Україні провели свій перший онлайн хакатон. Незважаючи на значний накопичений досвід, онлайн подія стала для організаторів викликом, як і для наших учасників, які до цього також не мали подібного досвіду. Ми, як організатори, доклали максимум зусиль, аби учасники та експерти не відчували незручностей дистанційної роботи і працювали зосереджено та злагоджено.

Проведення Хакатону та участь у хакатоні. Тут краще один раз прийняти участь, а ніж 100 разів чути, дивитись, чи читати про це. Головна ідея хакатону, на нашу думку – дати студентам можливість доторкнутися до важливих задач, над якими працюють провідні ІТ-спеціалісти та ІТ-компанії. Дати можливість студенатам спробувати розкрити свій потенціал, хоч і за короткий час. Ми знаємо багато історій успіху, наприклад, одна з команд, яка прийшла на хакатон з ідеєю, за час хакатону реалізувала її, і зайняла перше місце – далі їм вдалося знайти спонсорів і впровадити ідею та розробку в життя. Вони запустили свій маленький стартап.

Хакатон – це дім, де місце знайдеться кожному: як команді з однієї людини (які, до речі, частенько так показують дуже високі результати!), так і цілком сформованим командам, які злагоджено і цілеспрямовано працюють над досягненням результату.

Після участі у хакатоні ніхто не залишається байдужим. Ось що кажуть учасники [2,3]:

- *“Сенс хакатону – у подоланні труднощів.”*
- *“Ми перепробували різні варіанти, але продовжували шукати. І згодом нам спала на думку ідея, яка дозволила фактично перемогти у хакатоні.”*
- *“Не тільки знання можуть допомогти тобі перемогти, але й твоя креативність та наполегливість. Це було круто!”*
- *“За два вихідних ми дізнались стільки нового та засвоїли на практиці, що у звичайному житті не заставили б себе вивчити і за рік!”*

Валерій Кригін, R&D Software Engineer Apostera, підсумовує: "Вражений, наскільки учасникам вистачає сил і наполегливості два вихідних поспіль, без перерви, "хакатонити"! Продуктивність команд вражає не менше! Цікаво, що для розв'язання задач використаний дуже широкий спектр технологій. Учасники дуже швидко навчались – хакатон заставив мобілізувати знання з різних сфер математики. Команди, які дійшли до кінця, показали силу волі, віру в себе – і це теж дуже важливо! Це – головна перемога!

Як завжди, сподобався творчий підхід студентів. Для них не є шоком, коли задача поставлена не чітко і треба "придумати собі задачу".

Вчергове, дивує, як організаторам вдається зібрати команди з різних міст України, навіть з-за меж України! Як їм вдається все це організувати та підтримувати комунікацію, а також розв'язувати усі питання вчасно" [2,3].

"Хакатони є прекрасною можливістю не лише позмагатись і вирішити практичні кейси, а й зрозуміти свою експертизу, а також зростити її під час події." – резюмувала Ольга Курна, менеджерка GlobalLogic з освітніх програм і співпраці з університетами, на одному з хакатонів [2].

Хакатон – це історія про наполегливість та рух до перемоги. Адже, ми знаємо, що перемагає той, хто витримає на півгодини довше, ніж суперники.

Список використаних джерел

1. Панченко І. В. Що таке хакатон, і чи можна його відправити на карантин? [Електронний ресурс] / І. В. Панченко. – Київ : Ліга.Блоги, 2021. – Режим доступу : https://blog.liga.net/user/ipanchenko/article/chto-takoe-hakaton-i-mojno-li-ego-otpravit-na-karantin. – Назва з екрану.
2. CSC Hackathon 2021 Succeed [Електронний ресурс]. – Режим доступу : https://hackathon.expert/csc-hackathon-2021-succeed/. – Назва з екрану.
3. March AI Hackathon [Електронний ресурс]. – Режим доступу : https://hackathon.expert/march-ai-hackathon/. – Назва з екрану.

Ткаченко Олексій Миколайович, к.т.н, доцент
Омельчук Людмила Леонідівна, к.ф.-м.н, доцент
Шишацька Олена Володимирівна, к.ф.-м.н, асистент

МЕТОДИЧНІ АСПЕКТИ ІНТЕГРАЦІЇ НАВЧАЛЬНИХ КУРСІВ В ГАЛУЗІ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Роботу присвячено методичним питанням впровадження міждисциплінарних зв'язків у навчальний процес закладів вищої освіти, зокрема інтеграційному підходу при виченні дисциплін галузі інформаційних технологій. Продемонстровано застосування такого підходу на прикладі поєднання практичних частин дисциплін «Методи специфікації програм», «Коректність програм та логіки програмування» та «Інструментальні середовища та технології програмування»).

Ключові слова: навчальний процес закладів вищої освіти, дисципліни галузі інформаційних технологій, інтеграційний підхід.

The publication is devoted to methodological issues of implementing of interdisciplinary links into the educational process in universities, in particular, the integration approach in the study of IT oriented learning courses. The applying of proposed approach was demonstrated on the example of a combination of practical parts in courses "Methods of software specification", "Program correctness and programming logic", and "Instrumental environments and programming technologies".

Keywords: educational process in universities, IT oriented learning courses, integration approach.

Актуальність проблеми впровадження принципів міждисциплінарних (міжпредметних) зв'язків у вищій школі зумовлена насамперед забезпеченням якості вищої освіти. Запит суспільства на високо-кваліфікованих фахівців, зокрема у галузі інформаційних технологій, надзвичайно високий.

Наразі попит на нових IT-фахівців в Україні складає 30-50 тисяч осіб на рік (за результатами "Аналізу IT-освіти у вишах України" Офісу ефективного регулювання BRDO в лютому 2021 року [1]). Разом з тим

заклади вищої освіти України щороку випускають у середньому 16,2 тисяч бакалаврів ІТ-спеціальностей. Однак, лише 44% випускників українських закладів вищої освіти після отримання диплому працюють за фахом [2], решта не відповідають рівню своєї кваліфікації. Зазначене свідчить про наявність проблем у професійній підготовці ІТ-фахівців, а саме, у відповідності рівня фахової підготовки випускників сучасним потребам ІТ-ринку.

Згідно опитувань стейкголдерів освітнього процесу, при підготовці фахівців для ІТ-галузі основними проблемами є [1, 2]:

(1) існування невідповідності між очікуваннями роботодавців в галузі ІТ та практичними уміннями і соціальними навичками випускників ІТ-факультетів і, як наслідок, висока потреба у підвищенні рівня зазначених компетентностей у здобувачів вищої освіти;

(2) недостатній зв'язок між теоретичним і практичним матеріалом навчальних дисциплін;

(3) розмитість міждисциплінарних зв'язків;

(4) недостатнє системне розуміння студентами життєвого циклу розробки програмного забезпечення.

Інтеграція навчальних курсів в галузі інформаційних технологій на основі проєктного підходу та гнучких методологій управління повинна сприяти усуненню зазначених проблем.

Формування професійної компетентності фахівців на основі використання міжпредметних зв'язків у навчальному процесі розглядається як важливе завдання сучасної освіти

Слід зазначити, що, незважаючи на численну кількість досліджень (як правило, теоретичних) міжпредметної інтеграції в навчальному процесі закладів вищої освіти, не розроблено конкретні методики впровадження основних положень теорії міжпредметної інтеграції відповідно профілю освіти.

Сучасна освіта передбачає різні шляхи міжпредметної інтеграції, зокрема: створення інтегрованих курсів (адаптація та інтегрування знань декількох наук); створення нових форм занять (заняття з міжпредметними

зв’язками, інтегроване заняття тощо); впровадження навчальних проектів. Останній бачиться найбільш прийнятним та ефективним у навчальних дисциплінах ІТ-галузі.

Перевагами інтегрованого навчання для студентів є чітке розуміння мети кожного предмету в різних контекстах, глибоке розуміння будь-якої теми, завдяки її дослідженню через кілька точок зору, усвідомлення комплексного підходу, через який предмети, навички, ідеї та різні точки зору пов’язані з реальним світом та вдосконалення навичок системного мислення [3].

Зрозуміло, що процес інтеграції вимагає виконання певних умов: об’єкти дослідження однакові або досить близькі (дослідження об’єкту з різних сторін, використовуючи навчальний матеріал різних дисциплін); у навчальних предметах використовуються однакові або близькі методи дослідження предметів та явищ (демонстрація способу пізнання дійсності на прикладах з різних предметів); пізнавані об’єкти та явища підпорядковуються загальним закономірностям (узагальнення навчального матеріалу з різних навчальних дисциплін та пізнання більш складної системи) [3].

Експеримент по реалізації інтеграції навчальних курсів по впровадженню методики проведення інтегрованих курсів (далі проєкт-експеримент) успішно проведено в межах освітньо-професійної програми «Інформатика» бакалаврського рівня вищої освіти, що реалізується на факультеті комп’ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка за спеціальністю 122 «Комп’ютерні науки» в 2020-2021 навчальному році (другий семестр). В рамках спільного проєктного завдання для студентів другого та четвертого років навчання було поєднано практичні частини наступних дисциплін:

- «Методи специфікації програм» (вибіркова дисципліна, 4 рік навчання, викладач: к.ф.-м.н. Шишацька О.В., задіяно 25 студентів);
- «Коректність програм та логіки програмування» (вибіркова дисципліна, 4 рік навчання, викладач: к.т.н. Ткаченко О.М., задіяно 25 студентів);

- «Інструментальні середовища та технології програмування» (обов’язкова дисципліна, 2 рік навчання, викладач: к.ф.-м.н. Омельчук Л.Л., задіяно 8 студентів).

В проєкті-експерименті передбачалося досягнення таких цілей:

(1) посилення практичної складової навчальних дисциплін, задіяних у проєкті-експерименті;

(2) підвищення у студентів рівня розуміння:

- усіх етапів життєвого циклу програмного забезпечення;
- підходів до управління ІТ-проектами;
- міждисциплінарних зв’язків;
- зв’язків між теоретичним і практичним матеріалом;

(3) підвищення рівня професійних та соціальних навичок у студентів;

(4) врахування результатів проєкту-експерименту при перегляді та оновленні освітньої програми та її компонентів.

Для досягнення цілей проєкту-експерименту було поставлено такі задачі:

(1) інтеграція на прикладі теоретично орієнтованих курсів «Методи специфікації програм», «Коректність програм та логіки програмування» та практично орієнтованої курсу «Інструментальні середовища та технології програмування»;

(2) проведення експерименту, в рамках якого для виконання проєктних завдань формуються команди за різними критеріями: володіння технологіями розробки ПЗ, методологіями управління ІТ-проєктом, складом (віковий, соціально-спрямованими вподобаннями, ін.);

(3) розробка критеріїв оцінювання успішності запропонованого підходу до інтеграції дисциплін;

(4) аналіз результатів успішності проєкту-експерименту, формування пропозицій щодо його вдосконалення та впровадження як кейсу на постійній основі;

(5) розробка програмного продукту підтримки інтеграції навчальних дисциплін.

Цільовою аудиторією проєкту-експерименту стали:

- студенти та викладачі бакалаврської освітньої програми «Інформатика» за спеціальністю 122 «Комп'ютерні науки», що

реалізується на факультеті комп'ютерних наук та кібернетики КНУ імені Тараса Шевченка, які задіяні в рамках дисциплін «Методи специфікації програм», «Коректність програм та логіки програмування» та «Інструментальні середовища та технології програмування» (пряма аудиторія);

- IT-компанії, студенти і викладачі інших освітніх програм за IT-спеціальностями (опосередкована аудиторія).

Було визначено наступні показники досягнення цілей:

(1) підвищення рівня професійних та соціальних навичок у студентів, задіяних в проєкті-експерименті (інструмент перевірки – вихідне опитування);

(2) наявність програмних систем, розроблених студентськими командами;

(3) наявність звітів за результатами роботи кожної студентської команди;

(4) наявність звіту викладачів про результати проєкту-експерименту;

(5) оприлюднення результатів проєкту-експерименту.

Проведення проєкту-експерименту

Для студентів четвертого курсу участь в проєкті була обов'язковою. Студентам другого курсу було запропоновано на добровільних засадах долучитися до проєкту-експерименту. При цьому було враховано результати навчання (підсумковий бал більше 80) з обов'язкової дисципліни «Об'єктно-орієнтоване програмування», яку ці студенти вивчали в попередньому семестрі.

На початку семестру студентам четвертого року навчання було запропоновано пройти анкетування, яке включало такі питання:

- ***чи маєте Ви досвід роботи в реальних колективних проектах (не навчальних)*** (*був досвід, не маю досвіду, зараз в проекті);*
- ***запропонуйте декілька варіантів предметних областей або конкретних систем Вашого майбутнього проекту*;**
- ***вкажіть не більше, ніж 3 людини, з якими Ви б хотіли працювати в команді*;**

- ***на Вашу думку, в якій ролі в командній роботі Ви би*** *(почувалися впевнено, НЕ хотіли себе бачити, хотіли "прокачати" себе в проекті / керівник проекту, модератор, фахівець з документації, аналітик, фронтенд, бекенд, тестувальник);*
- ***які компетентності Ви би хотіли розвинути в рамках майбутнього проекту*** (поглибити теоретичні знання, розвинути практичні вміння, програмувати, оформляти документацію, працювати в команді, планувати і організовувати свій час, вміння презентувати результати діяльності, комунікаційні навички, інше);
- ***досвід роботи з якою платформою і/або методологією управління IT-проектами Ви маєте;***
- ***в якій системі управління IT-проектами (методології) Ви б хотіли працювати в проекті;***
- *н****а Вашу думку, використання якої мови для Вас є*** *(близькою, Ви нею володієте на впевненому рівні, не бажаною, Ви не впевнені, що володієте нею на хорошому рівні, бажаною в проекті, бо Ви хочете "прокачати" себе в ній / C#, C++, Java, PHP, Python, інше)*;
- ***ваші пропозиції щодо організації проекту.***

На рис. 1-3 наведено деякі результати вхідного анкетування.

Рисунок 1. Результати анкетування щодо досвіду участі в колективних проектах

Рисунок 2. Результати анкетування щодо бажаних компетентностей

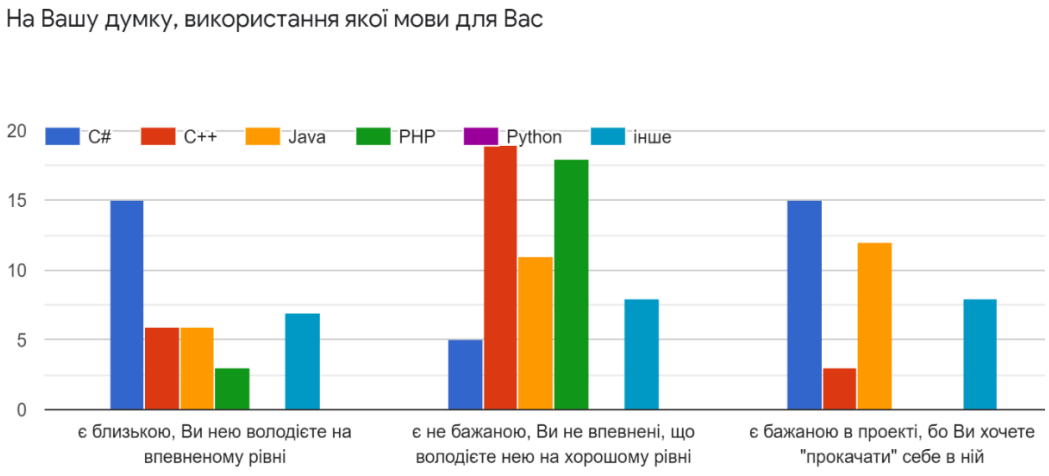


Рисунок 3. Результати анкетування щодо рівня володіння мовами програмування

За результатами анкетування для формування команд було побудовано соціометричний орієнтований граф, який враховував досвід роботи в командних проєктах, ступінь володіння технологіями та вибір колег. З метою експерименту для формування команд застосовувалися різні підходи:

- за роком навчання: команди, що складалися виключно зі студентів четвертого року навчання та команди, що складалися зі студентів четвертого та другого років навчання;
- за соціальними вподобаннями (з урахуванням вибору бажаних партнерів по проєкту): команди з тісними соціальними зв'язками (взаємний вибір) та команди з відсутніми соціальними зв'язками;
- за досвідом участі в реальних колективних проєктах: усі учасники мали досвід, ніхто з учасників не мав досвіду, змішані команда;
- за технологіями: команди зі спільними побажаннями щодо технологій розробки та команди з різними вподобаннями.

В результаті було сформовано вісім команд, з яких сім успішно виконали проєкт. Варто зазначити, що команда, яка не завершила проєкт складалася зі студентів одного року навчання, які мали взаємний вибір (тісні соціальні зв'язки) та однакові технологічні вподобання.

В рамках проєкту-експерименту здійснювалися щотижневі зустрічі учасників студентських команд з викладачами та організовано фінальний публічний захист студентських розробок. Звіти про виконання колективних проєктів, розроблених в межах проєкту-експерименту наведено в збірнику [4].

По завершенню проєкту-експерименту було проведено два вихідних опитування: анонімне та авторизоване. Деякі результати цих опитувань наведено на рис. 4-7.

Рисунок 4. Результати вихідного опитування щодо розвинених компетентностей

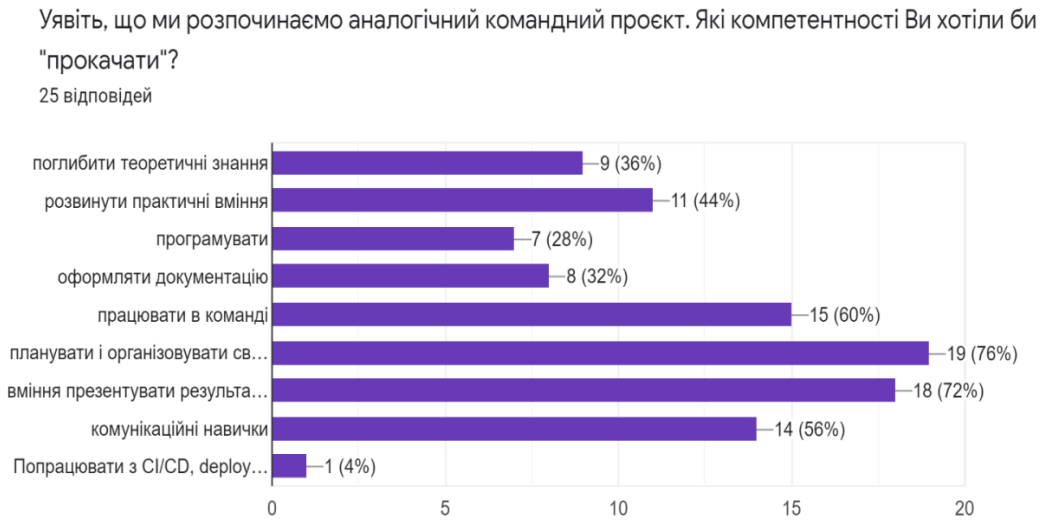


Рисунок 5. Результати вихідного опитування щодо бажаних компетентностей

Рисунок 6. Результати вихідного опитування щодо важливості критеріїв формування команд

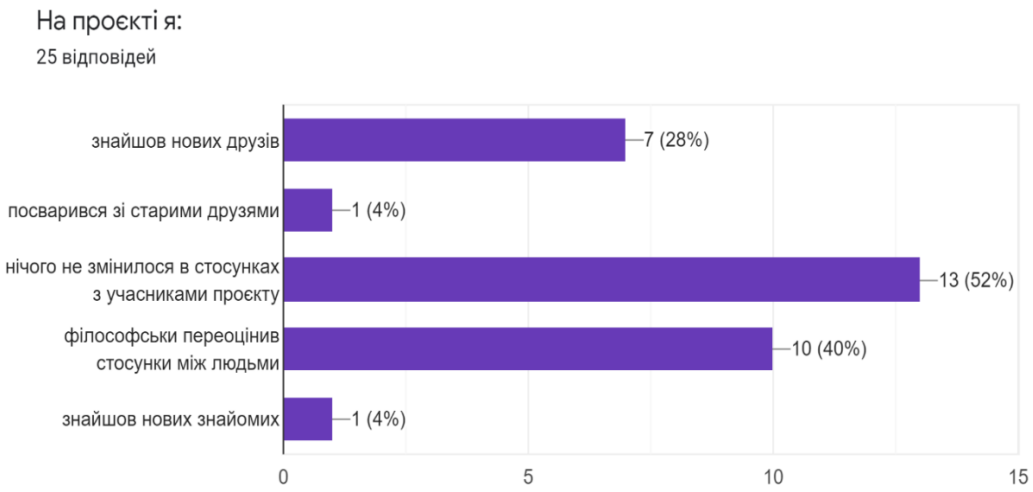


Рисунок 7. Результати вихідного опитування – рефлексія

Висновки. В результаті проведення проєкту-експерименту досягнуто наступні результати:

(1) розроблено програмні продукти підтримки інтеграції навчальних дисциплін;

(2) підвищено рівень професійних та соціальних навичок у студентів, задіяних в проєкті-експерименті;

(3) підвищено у задіяних в проєкті-експерименті студентів рівень розуміння (усіх етапів життєвого циклу програмного забезпечення; підходів до управління ІТ-проектами; міждисциплінарних зв'язків в межах освітньої програми; зв'язків між теоретичним та практичним матеріалом);

(4) посилено практичну складову навчальних дисциплін, задіяних в експерименті;

(5) сформовано пропозиції щодо оновлення робочих програм навчальних дисциплін, задіяних в експерименті, та освітньої програми в цілому – впроваджувати міждисциплінарний проєктний підхід з використанням гнучких методологій управління проєктами.

В оновленій освітньо-професійній програмі «Інформатика» бакалаврського рівня вищої освіти, що реалізується на факультеті комп'ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка за спеціальністю 122 «Комп'ютерні науки» в рамках спільного проєктного завдання для студентів другого та четвертого років навчання планується поєднати практичні частини наступних дисциплін: «Основи управління ІТ-проєктами» (дисципліна вибіркового блоку), «Інформаційні технології та правовий захист» (дисципліна вибіркового блоку), «Розробка ПЗ під мобільні платформи» (дисципліна вибіркового блоку) та «Інструментальні середовища та технології програмування» (обов'язкова дисципліна).

Перспективи дослідження полягають у подальшому розробленні науковометодичного забезпечення освітнього процесу майбутніх фахівців у галузі інформаційних технологій. Зокрема:

(1) розробка та опис методики інтеграції навчальних курсів в галузі інформаційних технологій на основі проектного підходу та гнучких методологій управління;

(2) вироблення рекомендацій щодо впровадження методики інтеграції навчальних курсів в галузі інформаційних технологій на основі проектного підходу та гнучких методологій управління;

(3) поширення досвіду експерименту на інші навчальні дисципліни;

(4) впровадження програмного продукту підтримки інтеграції навчальних дисциплін, розробленого в межах експерименту;

(5) посилення практичної складової теоретикоорієнтованих курсів;

(6) підвищення якості освітньої програми, зокрема, в контексті формування професійних та соціальних навичок у випускників ІТ-спеціальностей.

Список використаних джерел

1. Лебедєв Д. Аналіз ІТ-освіти у ВИШах України / Д. Лебедєв, І. Самоходський, https://brdo.com.ua/wp-content/uploads/2021/02/Analiz_IT_osvity_u_vyshah_Ukrai-ny_Print.pdf, останній доступ: 2021/10/29.
2. Експерт: 44% випускників вишів працюють не за фахом // Укрінформ https://www.ukrinform.ua/rubric-society/2737000-ekspert-44-vipusknikiv-visiv-pracuut-ne-za-fahom.html, останній доступ: 2021/10/29.
3. Навчальні матеріали щодо інтегрованого навчання // https://nus.org.ua/articles/integrovane-navchannya-tematychnyj-i-diyalnisnyj-pidhody-chastyna-2/, останній доступ: 2021/10/29.
4. Програмування: теорія і практика. Збірник матеріалів за результатами ІТ-проєкту міждисциплінарної інтеграції. 2020-2021 навчальний рік / За редакцією Л.Омельчук, О.Ткаченка, О. Шишацької. – Одеса: Видавничий дім «Гельветика», 2021. 161 с.

ВІДОМОСТІ ПРО АВТОРІВ

БАСАРАБ Іван Андрійович
кандидат фізико-математичних наук, старший науковий співробітник
BASARAB Ivan Andriiovych
Ph.D. in Physical and Mathematical Sciences, Senior Researcher
e-mail: basexpot@gmail.com

ВОЛОХОВ Віктор Миколайович
кандидат фізико-математичних наук, доцент, доцент кафедри теорії та технології програмування,
Київський національний університет імені Тараса Шевченка
VOLOKHOV Viktor Mykolaiovych
Ph.D. in Physical and Mathematical Sciences, Associate Professor, Associate Professor of the Department of Theory and Technology of Programming,
Taras Shevchenko National University of Kyiv
e-mail: vvn@uiccnet.com.ua

ГУБСЬКИЙ Богдан Володимирович
доктор економічних наук, кандидат фізико-математичних наук, професор
GUBSKY Bogdan Volodymyrovych
Doctor of Economics, Ph.D. in Physical and Mathematical Sciences, Professor
e-mail: basexpot@gmail.com

ДОРОШЕНКО Анатолій Юхимович
доктор фізико-математичних наук, професор,
професор кафедри інформаційних систем та технологій
НТУУ “КПІ імені ігоря Сікорського”
DOROSHENKO Anatoliy Yukhymovych
Doctor of Physical and Mathematical Sciences, Professor,
professor of the Department of Information Systems and Technologies

NTUU “Igor Sikorsky Kyiv Polytechnic Institute”
e-mail: doroshenkoanatoliy2@gmail.com
ORCID ID 0000-0002-8435-1451

ДУЦЯК Ігор Зенонович
доктор філософських наук, професор, професор кафедри,
Національний університет «Львівська політехніка»
DUTSIAK Ihor Zenonovych
Doctor of Philosophical Sciences, Professor,
Professor of the Department,
Lviv Polytechnic National University
e-mail: Idutsyak@gmail.com
ORCID ID 0000-0001-8751-4001

ЗУБЕНКО Віталій Володимирович
кандидат фізико-математичних наук, доцент кафедри теорії та технології програмування,
Київський національний університет імені Тараса Шевченка
ZUBENKO Vitalii Volodymyrovych
Ph.D. Physical and Mathematical Sciences, Associate Professor, Associate Professor of the Department of Theory and Technology of Programming,
Taras Shevchenko National University of Kyiv
e-mail: zubenko@knu.ua
ORCID ID 0000-0001-9917-0971

ІВАНЕНКО Павло Андрійович
кандидат фізико-математичних наук, науковий співробітник,
Інститут програмних систем НАН України
IVANENKO Pavlo Andriiovych
Ph.D. in Physical and Mathematical Sciences,
research scientist of the Institute of Software Systems of NASU

e-mail: metzgermeister87@gmail.com
ORCID ID 0000-0001-5437-9763

ІВАНОВ Євген В'ячеславович
кандидат фізико-математичних наук, доцент,
Київський національний університет імені Тараса Шевченка
IVANOV Ievgen Vyacheslavovich
Ph.D. in Physical and Mathematical Sciences, Associate Professor,
Taras Shevchenko National University of Kyiv
e-mail: ivanov eugen@gmail.com

КОХАН Ярослав Олексійович
кандидат філософських наук, молодший науковий співробітник,
Інститут філософії ім. Г. С. Сковороди НАНУ, Київ
KOKHAN Yaroslav Olexiyovych
Ph.D. in Philosophical Sciences, junior researcher,
H. S. Skovoroda Institute of Philosophy of the NAS of Ukraine, Kyiv
e-mail: yarkaen@gmail.com
ORCID ID: 0000-0002-8958-772X

КУЗЕНКО Володимир Федорович
кандидат фізико-математичних наук, доцент, доцент кафедри теорії та технології програмування,
Київський національний університет імені Тараса Шевченка
KUZENKO Vladimir Fedorovich
Ph.D. in Physical and Mathematical Sciences, Associate Professor, Associate Professor of the Department of Theory and Technology of Programming,
Taras Shevchenko National University of Kyiv
e-mail: vkuzenko@ukr.net
ORCID ID 0000-0001-7472-6240

НІКІТЧЕНКО Микола Степанович
доктор фізико-математичних наук, професор, завідувач кафедри теорії та технології програмування,
Київський національний університет імені Тараса Шевченка
NIKITCHENKO Mykola Stepanovych
Doctor of Physical and Mathematical Sciences, Professor,
Chairman of the Department of Theory and Technology of Programming,
Taras Shevchenko National University of Kyiv
e-mail: nikitchenko_ms@knu.ua
ORCID ID 0000-0002-4078-1062

ОМЕЛЬЧУК Людмила Леонідівна
кандидат фізико-математичних наук, доцент, доцент кафедри теорії та технології програмування,
Київський національний університет імені Тараса Шевченка
OMELCHUK Liudmyla Leonidivna
Ph.D. in Physical and Mathematical Sciences, Associate Professor, Associate Professor of the Department of Theory and Technology of Programming,
Taras Shevchenko National University of Kyiv
e-mail: l.omelchuk@knu.ua
ORCID ID 0000-0002-2287-1304

ПАНЧЕНКО Тарас Володимирович
кандидат фізико-математичних наук, доцент, доцент кафедри теорії та технології програмування,
Київський національний університет імені Тараса Шевченка,
співзасновник та лідер ГО “Хакатон Експерт”
PANCHENKO Taras Volodymyrovych
Ph.D. in Physical and Mathematical Sciences, Associate Professor at Department of Theory and Technology of Programming,
Taras Shevchenko National University of Kyiv,
co-founder and leader of NGO “Hackathon Expert”

e-mail: taras.panchenko@gmail.com
ORCID ID 0000-0003-0412-1945

РУСІНА Наталія Геннадіївна
кандидат педагогічних наук, доцент кафедри теорії та технології програмування,
Київський національний університет імені Тараса Шевченка
RUSINA Nataliia Hennadiivna
Ph.D. in Pedagogical Sciences, Associate Professor of the Department of Theory and Technology of Programming,
Taras Shevchenko National University of Kyiv
e-mail: rusina@knu.ua
ORCID ID 0000-0002-5595-9548

ТКАЧЕНКО Олексій Миколайович
кандидат технічних наук, доцент, доцент кафедри теорії та технології програмування,
Київський національний університет імені Тараса Шевченка
TKACHENKO Oleksii Mykolaiovych
Ph.D. in Computer Science, Associate Professor, Associate Professor of the Department of Theory and Technology of Programming,
Taras Shevchenko National University of Kyiv
e-mail: otkachenko@knu.ua
ORCID ID 0000-0002-9514-516X

ШИШАЦЬКА Олена Володимирівна
кандидат фізико-математичних наук, асистент кафедри теорії та технології програмування,
Київський національний університет імені Тараса Шевченка
SHISHATSKA Olena Volodymyrivna

Ph.D. in Physical and Mathematical Sciences, Assistant of the Department of Theory and Technology of Programming,
Taras Shevchenko National University of Kyiv
e-mail: shyshatska@knu.ua
ORCID ID 0000-0001-8791-8989

ШКІЛЬНЯК Степан Степанович
доктор фізико-математичних наук, професор, професор кафедри теорії та технології програмування,
Київський національний університет імені Тараса Шевченка
SHKILNYAK Stepan Stepanovych
Doctor of Physical and Mathematical Sciences, Professor,
professor of the Department of Information Systems and Technologies
Taras Shevchenko National University of Kyiv
e-mail: ss.sh@knu.ua
ORCID ID 0000-0001-8624-5778

ЯЦЕНКО Олена Анатоліївна
кандидат фізико-математичних наук, старший науковий співробітник
Інститут програмних систем НАН України
YATSENKO Olena Anatoliivna
Ph.D. in Physical and Mathematical Sciences,
senior research scientist of the Institute of Software Systems of NASU
e-mail: oayat@ukr.net
ORCID ID 0000-0002-4700-6704

Наукове видання

МАТЕМАТИЧНА ЛОГІКА ТА ПРОГРАМУВАННЯ. ДОСВІД ВИКЛАДАННЯ

МОНОГРАФІЯ

До 50-річчя кафедри теорії та технології програмування
факультету комп'ютерних наук та кібернетики
Київського національного університету імені Тараса Шевченка

Технічний редактор: Сімянчук І.Ю.

Підписано до друку 22.02.2022 р. Формат 84x108/8.
Папір офсетний. Цифровий друк.
Ум. друк. арк. 18,82. Наклад 300,
Замовлення № 0621-193
Віддруковано з готового оригінал-макета.

Видавництво: Видавничий дім «Гельветика»
69002, Україна, м. Запоріжжя, вул. Олександрівська, 84, оф. 414
Тел.: +38 (048) 709 38 69, +38 (095) 934 48 28, +38 (097) 723 06 08
E-mail: mailbox@helvetica.ua
Свідоцтво суб'єкта видавничої справи
ДК № 6424 від 04.10.2018 р.

Друкарня: Типографія «Айс Принт»
Тел: +38 (099) 192-00-33, +38 (048) 706-92-82
Site: www.ice-print.com.ua
E-mail:info@ice-print.com.ua