

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ  
Дніпровський національний університет імені Олеся Гончара  
МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ  
Дніпровський національний університет імені Олеся Гончара

Кваліфікаційна наукова  
праця на правах рукопису

**ФОРКЕРТ ПАВЛО ПАВЛОВИЧ**

УДК 004.42

**ДИСЕРТАЦІЯ**  
**ДОСЛІДЖЕННЯ ВИКОРИСТАННЯ РАНТАЙМУ GO ЯК ПЛАТФОРМИ**  
**ДЛЯ ПОБУДОВИ НОВИХ МОВ ПРОГРАМУВАННЯ**

12 Інформаційні технології  
121 Інженерія програмного забезпечення

Подається на здобуття ступеня доктора філософії. Дисертація містить результати власних досліджень. Використання ідей, результатів та текстів інших авторів мають посилання на відповідне джерело

\_\_\_\_\_ П. П. Форкерт

Науковий керівник:  
Іванченко Марина Геннадіївна  
кандидат технічних наук, доцент

Дніпро – 2026

## АНОТАЦІЯ

*Форкерт П. П.* Дослідження використання рантайму Go як платформи для побудови нових мов програмування. — Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття ступеня доктора філософії за спеціальністю 121 Інженерія програмного забезпечення — Дніпровський національний університет імені Олеся Гончара, Дніпро, 2026.

Сучасна практика створення нових мов програмування дедалі рідше спирається на повну побудову компілятора, рантайму та інструментальної екосистеми з нуля. Натомість домінує підхід повторного використання вже наявних платформ, який дає змогу зменшити вартість реалізації мови, скоротити час експериментування з новими мовними конструкціями та швидше довести нову мову або діалект до практичного застосування. Найчастіше в такому контексті досліджуються JVM, LLVM, GraalVM та інші спеціалізовані платформи. Водночас придатність Go як основи для побудови нових мов програмування вивчена значно менше, хоча вона має важливі інженерні переваги: швидкий компілятор, зрілий рантайм, сучасний збирач сміття, підтримку кроскомпіляції, велику стандартну бібліотеку та розвинені засоби аналізу і генерації коду.

У дисертаційній роботі досліджено зручність і доцільність використання рантайму Go як платформи для побудови нових мов програмування, діалектів та предметно-орієнтованих мов через транспіляцію у звичайний Go-код. Основну увагу зосереджено на тому, які класи мовних розширень можуть бути легко і ефективно реалізовані поверх Go без модифікації офіційного компілятора, які архітектурні принципи повинен мати відповідний транспілятор, які обмеження накладає сама хост-платформа та за яких умов такий підхід є практично виправданим. Як експериментальне підтвердження запропонованого підходу розроблено діалект GoNext, який використано як демонстраційний майданчик для реалізації низки мовних розширень.

У роботі показано, що Go доцільно розглядати не як класичну багатомовну віртуальну машину і не як низькорівневий компіляторний фреймворк, а як хост-платформу особливого типу, придатну насамперед для транспільованих мов, що прагнуть зберегти сумісність із Go-екосистемою. Запропоновано систему критеріїв оцінювання такої придатності, яка охоплює виконувальні властивості, виразні можливості хост-мови, інтероперабельність, інструментальну підтримку транспіляції та вартість реалізації мовних конструкцій. Також розроблено узагальнену архітектуру транспілятора до Go та класифікацію перетворень мовних розширень на синтаксичні, структурні й репрезентаційні.

У **першому розділі** виконано аналіз сучасних підходів до реалізації мов програмування на базі наявних платформ, зокрема JVM, LLVM та Truffle/GraalVM, і показано, що повторне використання рантаймів, компіляторних бекендів і віртуальних машин є нині домінантною тенденцією при створенні нових мов. Розглянуто роль транспіляції як самостійного механізму побудови мов, особливо доцільного тоді, коли хост-платформа не надає стабільного низькорівневого API чи байткоду, але має стабільну мову, компілятор і потужні засоби аналізу та генерації коду. Проаналізовано інженерні властивості Go, релевантні для її використання як хост-платформи, а також її інструментальну екосистему (`go/parser`, `go/ast`, `go/types`, `go/printer`, `golang.org/x/tools` тощо). Окремо розглянуто наукові й практичні роботи, дотичні до використання Go як платформи для інших мов, і встановлено, що така практика вже існує, але майже не супроводжується системним науковим аналізом.

У **другому розділі** сформульовано теоретичні засади використання Go як цілі транспіляції: розроблено п'ятикомпонентну систему критеріїв оцінювання, виокремлено класи мов, для яких Go є доцільною ціллю, і навпаки, для яких вона малоприматна, а також запропоновано загальну архітектуру транспілятора. Описано підходи до часткової типізації, нормалізації високорівневих конструкцій до обмеженого набору внутрішніх

форм, трансформації AST у вузли `go/ast` та забезпечення інтероперабельності. Сформульовано класифікацію трансформацій мовних розширень і набір принципів проєктування нових мов на базі Go, серед яких — мінімальна семантична відстань до Go, перевага локальних трансформацій над глобальними, максимальне використання офіційних інструментів Go та пріоритет інтероперабельності над теоретично «ідеальною» реалізацією окремої мовної конструкції.

У **третьому розділі** описано експериментальну реалізацію діалекту GoNext і продемонстровано практичну реалізацію повнофункціональних enum-типів, зіставлення із взірцем, значень параметрів за замовчуванням, іменованих аргументів, методів розширення, узагальнених методів, коротких анонімних функцій і універсального синтаксису виклику функцій. Кожне розширення співвіднесено із запропонованою класифікацією трансформацій: enum-типи проілюстровано як приклад репрезентаційної трансформації, `match` — як структурної, а ергономічні розширення — переважно як синтаксичні або структурні. Показано, що частина конструкцій потребує не лише локальних синтаксичних переписувань, а й часткового семантичного аналізу — знання сигнатур функцій, областей видимості та типів оточуючого контексту, — а також що окремі розширення утворюють змістовні комбінації, при цьому зберігаючи високу інтероперабельність зі звичайним Go, оскільки після трансформації здебільшого зникають, залишаючи звичайні Go-функції, типи та виклики у результуючому коді.

У **четвертому розділі** виконано комплексне оцінювання запропонованого підходу за всіма групами критеріїв — виконувальними властивостями, виразністю, інтероперабельністю, доступністю інструментів та вартістю реалізації, — визначено його межі застосовності та проведено порівняння Go з альтернативними платформами. Встановлено, що найбільший вигравш цей підхід дає для діалектів і надмножин Go, предметно-орієнтованих мов серверно-інфраструктурного профілю та нових мов, свідомо спроектованих з орієнтацією на трансформацію у Go, тоді як для мов

із радикально іншою моделлю виконання, ручним керуванням пам'яттю або потребою в окремому байткодi він є малодоцільним. Сформульовано практичні рекомендації щодо проєктування нових мов і транспіляторів, наведено методику прийняття рішення про вибір Go як цільової платформи та узагальнено результати на ширший клас задач, зокрема транспіляцію інших мов у Go.

**Наукова новизна** дисертаційної роботи полягає в такому:

1. Вперше системно обґрунтовано доцільність розгляду рантайму Go як хост-платформи особливого типу для побудови нових мов програмування — не як багатомовної віртуальної машини і не як низькорівневого компіляторного фреймворку, а як цілі транспіляції для мов із високою інтероперабельністю з екосистемою Go.
2. Удосконалено підхід до оцінювання придатності Go як цілі транспіляції за рахунок комплексного врахування виконувальних властивостей, виразних можливостей хост-мови, інтероперабельності, інструментальної підтримки транспіляції та вартості реалізації мовних конструкцій.
3. Удосконалено архітектурний підхід до побудови мовних розширень у формі надмножини Go з поетапною трансформацією у звичайний Go-код, який поєднує нормалізацію конструкцій, частковий семантичний аналіз і перетворення у вузли `go/ast`, що дає змогу коректно реалізовувати конструкції, залежні від сигнатур функцій, областей видимості, контекстного відновлення типів і правил уніфікації типів Go, із збереженням двосторонньої інтероперабельності з екосистемою Go.
4. Вперше в межах єдиного транспіляційного підходу показано можливість реалізації комплексу мовних розширень поверх Go-рантайму без модифікації офіційного компілятора Go, зокрема повнофункціональних `enum`-типів, зіставлення із взірцем, іменованих аргументів, параметрів за замовчуванням, методів розширення,

узагальнених методів, коротких анонімних функцій та універсального синтаксису виклику функцій.

**Практичне значення роботи** полягає в тому, що запропонований підхід дає змогу суттєво зменшити витрати на створення нових мов програмування, діалектів і предметно-орієнтованих мов за рахунок повторного використання компілятора, рантайму, системи збірки та бібліотек Go. Результати роботи можуть бути використані для проєктування нових серверних, інтеграційних, інфраструктурних і конфігураційних мов, для створення навчальних і дослідницьких транспіляторів, а також для еволюційного розширення мови Go без втрати сумісності з уже наявним кодом і бібліотеками Go. Експериментальна реалізація GoNext підтверджує практичну здійсненність запропонованих рішень і може слугувати основою для подальших досліджень та прикладних розробок.

**Ключові слова:** Go, рантайм Go, мова програмування, транспіляція, компіляція, програмне забезпечення, алгоритм, критерії, система, архітектура, модель, інформаційні технології, хмарні технології, формальна мова, граматики.

## ABSTRACT

*Forkert P. P.* Investigating the use of the Go runtime as a platform for developing new programming languages. — Qualifying scientific work submitted as a manuscript.

Dissertation for the degree of Doctor of Philosophy in Specialty 121 Software Engineering — Oles Honchar Dnipro National University, Dnipro, 2026.

Modern practice in the design of new programming languages increasingly relies not on building a compiler, runtime, and tooling ecosystem entirely from scratch, but on reusing existing platforms. This approach reduces implementation cost, shortens the time needed to experiment with new language constructs, and makes it possible to bring a new language or dialect to practical use much faster. In this context, JVM, LLVM, GraalVM, and other specialized platforms have been studied extensively. By contrast, Go as a foundation for building new programming languages has received much less attention, despite its important engineering advantages: a fast compiler, a mature runtime, a modern garbage collector, cross-compilation support, a large standard library, and well-developed tools for code analysis and generation.

This dissertation investigates the convenience and feasibility of using the Go runtime as a platform for building new programming languages, dialects, and domain-specific languages through transpilation into ordinary Go code. The main focus is on which classes of language extensions can be implemented easily and efficiently on top of Go without modifying the official compiler, what architectural principles such a transpiler should follow, what limitations are imposed by the host platform itself, and under which conditions this approach is practically justified. As an experimental validation of the proposed approach, the GoNext dialect was developed and used as a demonstration platform for implementing a set of language extensions.

The work shows that Go should be treated neither as a classical multilingual virtual machine nor as a low-level compiler framework, but as a special kind of host platform, primarily suitable for transpiled languages that aim to preserve

compatibility with the Go ecosystem. A system of evaluation criteria for such suitability is proposed; it covers execution-related properties, the expressive capabilities of the host language, interoperability, tooling support for transpilation, and the cost of implementing language constructs. A generalized architecture of a transpiler to Go is also developed, together with a classification of transformations for language extensions into syntactic, structural, and representational ones.

**Section 1** analyzes current approaches to implementing programming languages on top of existing platforms, in particular the JVM, LLVM, and Truffle/GraalVM, and shows that the reuse of runtimes, compiler backends, and virtual machines is currently the dominant trend in creating new languages. It examines the role of transpilation as an independent mechanism for language construction, which is especially appropriate when the host platform provides no stable low-level API or bytecode but does offer a stable language, a compiler, and powerful tools for code analysis and generation. The engineering properties of Go relevant to its use as a host platform are analyzed, along with its tooling ecosystem (`go/parser`, `go/ast`, `go/types`, `go/printer`, `golang.org/x/tools`, and others). The scientific and practical works related to using Go as a platform for other languages are reviewed separately, and it is established that such practice already exists but is almost never accompanied by systematic scientific analysis.

**Section 2** formulates the theoretical foundations of using Go as a transpilation target: it develops a five-component system of evaluation criteria, identifies the classes of languages for which Go is an appropriate target and, conversely, those for which it is poorly suited, and proposes a generalized transpiler architecture. Approaches to partial typing, normalization of high-level constructs into a limited set of internal forms, transformation of the AST into `go/ast` nodes, and interoperability support are described. A classification of transformations for language extensions is formulated, together with a set of principles for designing new languages on top of Go, among which are minimal semantic distance to Go, preference for local transformations over global ones, maximal use of the official

Go tools, and the priority of interoperability over a theoretically “ideal” implementation of an individual language construct.

**Section 3** describes the experimental implementation of the GoNext dialect and demonstrates the practical implementation of full-featured enum types, pattern matching, default parameter values, named arguments, extension methods, generic methods, short anonymous functions, and uniform function call syntax. Each extension is mapped onto the proposed classification of transformations: enum types are presented as an example of a representational transformation, `match` as a structural one, and the ergonomic extensions predominantly as syntactic or structural ones. It is shown that some constructs require not only local syntactic rewrites but also partial semantic analysis — knowledge of function signatures, scopes, and the types of the surrounding context — and that individual extensions form meaningful combinations while preserving high interoperability with ordinary Go, since after transformation they mostly disappear, leaving ordinary Go functions, types, and calls in the resulting code.

**Section 4** provides a comprehensive evaluation of the proposed approach across all groups of criteria — execution-related properties, expressiveness, interoperability, tooling availability, and implementation cost — determines its applicability limits, and compares Go with alternative platforms. It is established that the approach yields the greatest benefit for Go dialects and supersets, domain-specific languages of a server-infrastructure profile, and new languages deliberately designed with transpilation into Go in mind, whereas for languages with a radically different execution model, manual memory management, or a need for separate bytecode it is rarely justified. Practical recommendations for designing new languages and transpilers are formulated, a decision-making methodology for choosing Go as the target platform is presented, and the results are generalized to a broader class of tasks, specifically the transpilation of other languages into Go.

**The scientific novelty** of the dissertation is as follows:

1. For the first time, the feasibility of considering the Go runtime as a special kind of host platform for building new programming languages has been

systematically substantiated, with Go understood not as a multilingual virtual machine or a low-level compiler framework, but as a transpilation target for languages with high interoperability with the Go ecosystem.

2. The approach to assessing the suitability of Go as a transpilation target has been improved by comprehensively taking into account execution-related properties, the expressive capabilities of the host language, interoperability, tooling support for transpilation, and the cost of implementing language constructs.
3. The architectural approach to constructing language extensions in the form of a superset of Go with staged transformation into ordinary Go code has been improved; it combines construct normalization, partial semantic analysis, and transformation into `go/ast` nodes, which makes it possible to correctly implement constructs that depend on function signatures, scopes, contextual type reconstruction, and Go type-unification rules, while preserving bidirectional interoperability with the Go ecosystem.
4. For the first time, within a unified transpilation-based approach, the possibility of implementing a set of language extensions on top of the Go runtime without modifying the official Go compiler has been demonstrated, in particular full-featured enum types, pattern matching, named arguments, default parameter values, extension methods, generic methods, short anonymous functions, and uniform function call syntax.

**The practical significance of the work** lies in the fact that the proposed approach makes it possible to substantially reduce the cost of creating new programming languages, dialects, and domain-specific languages through the reuse of the Go compiler, runtime, build system, and libraries. The results of the work can be used for designing new server-side, integration, infrastructure, and configuration languages, for creating educational and research transpilers, and for the evolutionary extension of the Go language without losing compatibility with existing Go code and libraries. The experimental implementation of GoNext

confirms the practical feasibility of the proposed solutions and can serve as a basis for further research and applied developments.

**Keywords:** Go, Go runtime, programming language, transpilation, compilation, software, algorithm, criteria, system, architecture, model, information technologies, cloud technologies, formal language, grammar.

## Список опублікованих праць за темою дисертації

### *Статті у наукових фахових виданнях України:*

1. Forkert P. P., Sydorova M. G. Integrating full-featured enums into Go programming language. *Актуальні проблеми автоматизації та інформаційних технологій*, т. 27, Дніпро, 2023, с. 3-16. DOI: <http://dx.doi.org/10.15421/432301> (особистий внесок Форкерта П.П.: провів аналіз підходів до реалізації мов програмування та можливостей використання транспіляції в Go, сформулював вимоги до повнофункціональних епит-типів у діалекті GoNext, запропонував синтаксис їх оголошення, дослідив та порівняв кілька варіантів представлення епит-типів у Go-кодi, зокрема з урахуванням узагальнень, вказівників, роботи збирача сміття та накладних витрат, розробив підхід до генерації конструкторів, Match-методів і допоміжних методів доступу, а також проаналізував сумісність запропонованого рішення з екосистемою Go; Іванченко М.Г.: постановка завдання, узагальнення отриманих результатів).
2. Forkert P. P., Ivanchenko M. G. Implementing named arguments in go programming language dialect. *Актуальні проблеми автоматизації та інформаційних технологій*, т. 29, Дніпро, 2025, с. 3-11. DOI: <https://dx.doi.org/10.15421/432501>. (особистий внесок Форкерта П.П.: провів порівняльний аналіз підходів до реалізації іменованих аргументів у сучасних мовах програмування; обґрунтував вибір call-site-підходу, за якого іменовані аргументи не потребують змін у визначеннях функцій, запропонував синтаксис іменованих аргументів для GoNext, дослідив його сумісність із граматикою Go та розробив алгоритм транспіляції, що передбачає визначення сигнатури функції, перевірку помилкових і дубльованих імен параметрів, окремо проаналізував інтеграцію іменованих аргументів із механізмом значень параметрів за

замовчуванням; Іванченко М.Г.: постановка мети дослідження, контроль та узагальнення отриманих результатів).

3. Forkert P. P., Ivanchenko M. G. Implementing extension methods and generic methods in Go programming language dialect. *Системні технології*, т. 163, Дніпро, 2026, с. 111-121. DOI: <https://doi.org/10.34185/1562-9945-2-163-2026-10>. (особистий внесок Форкєрта П.П.: провів аналіз реалізацій методів розширення в сучасних мовах програмування та порівняв їх з універсальним синтаксисом виклику функцій, сформулював вимоги до реалізації методів розширення в GoNext з урахуванням мінімальних накладних витрат і повної сумісності зі стандартним Go, запропонував синтаксис оголошення методів розширення через модифікатор *extension*, механізм *import extension* для використання функцій наявних Go-бібліотек як методів розширення, а також алгоритм переписування викликів виду *value.Method(args)* у звичайні виклики функцій, дослідив застосування цього підходу для підтримки узагальнених методів, які відсутні у стандартному Go, та проаналізував інтероперабельність запропонованого рішення з існуючим кодом на Go; Іванченко М.Г.: постановка задачі, аналіз результатів).

**Наукові праці, які засвідчують апробацію матеріалів дисертації:**

1. Forkert P. P., Sydorova M. G., Honcharova Yu. S. MODERN ARCHITECTURE OF DYNAMICALLY TYPED PROGRAMMING LANGUAGE VMS. *Тези доповідей IV Всеукраїнської науково-практичної конференції молодих науковців та здобувачів вищої освіти «Сучасні науково-технічні дослідження у контексті мовного простору»*, Дніпро, 11 травня 2023, с. 188-190. URL: <https://www.confcontact.com/2023-suchasni-ntd/3-Forkert-Sydorova-Honcharova.pdf> (особистий внесок Форкєрта П.П.: провів аналіз

*архітектур сучасних віртуальних машин динамічно типізованих мов програмування, зокрема V8, SpiderMonkey та JavaScriptCore, узагальнив спільну для них багаторівневу схему виконання з інтерпретатором і JIT-компіляторами; Іванченко М.Г.: постановка завдання, узагальнення результатів; Гончарова Ю.С.: аналіз формулювань).*

2. Forkert P. P., Sydorova M. G. ADVANTAGES OF GOLANG AS A FOUNDATION FOR NEW PROGRAMMING LANGUAGES. *Тези доповідей XXI міжнародної науково-практичної конференції «МАТЕМАТИЧНЕ ТА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ІНТЕЛЕКТУАЛЬНИХ СИСТЕМ (МПЗІС-2023)», Дніпро, 22-24 листопада 2023, с. 7-8. URL: <https://mpzis.dnu.dp.ua/wp-content/uploads/2023/11/mpzis-2023.pdf> (особистий внесок Форкєрта П.П.: проаналізував переваги Go як основи для побудови нових мов програмування, зокрема продуктивність рантайму, підтримку конкурентності, збирач сміття, кроскомпіляцію та екосистему бібліотек, обґрунтував транспіляцію як основний спосіб використання Go-рантайму гостьовими мовами; Іванченко М.Г.: постановка завдання, узагальнення результатів).*
3. Forkert P. P., Sydorova M. G. CHALLENGES OF USING GOLANG AS A FOUNDATION FOR NEW PROGRAMMING LANGUAGES. *Тези доповідей VI Всеукраїнської науково-практичної інтернет-конференція студентів, аспірантів та молодих вчених «Сучасні інформаційні системи та технології», Хмельницький, 30 листопада 2023, с. 55-56. URL: <https://kntu.net.ua/ukr/content/download/110490/623634/file/CICT2023.pdf> (особистий внесок Форкєрта П.П.: систематизував обмеження використання Go як хост-платформи, зокрема відсутність стабільного API рантайму та готових інструментів транспіляції,*

- обмеженість узагальнень, і запропонував шляхи їх подолання; Іванченко М.Г.: постановка завдання, узагальнення результатів).*
4. Forkert P. P., Sydorova M. G. IMPROVING ENUMS IN GO PROGRAMMING LANGUAGE DIALECT. *Тези доповідей VI Міжнародної науково-практичної конференції молодих вчених та студентів «ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ І ПЕРЕДОВІ ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ (SOFT TECH-2024)», Київ, 21-23 травня 2024, с. 148-150. URL: <https://drive.google.com/file/d/18bZ9QBure7U08rbqiHmxWglTrK1C9D4L/view> (особистий внесок Форкерта П.П.: запропонував для GoNext синтаксис зіставлення із взірцем match, адаптований до стилю оператора switch у Go, із захисними умовами та перевіркою вичерпності, а також схему його транспіляції у виклики згенерованих Match-методів enum-типів; Іванченко М.Г.: постановка завдання, аналіз результатів).*
  5. Форкерт П. П., Іванченко М. Г. УНІВЕРСАЛЬНИЙ СИНТАКСИС ВИКЛИКУ ФУНКЦІЙ В ДІАЛЕКТІ МОВИ ПРОГРАМУВАННЯ GO. *Збірник наукових праць IV Міжнародної науково-практичної конференції «ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ В ОСВІТІ ТА НАУЦІ», Запоріжжя, 20 травня 2025, с. 561-565 (особистий внесок Форкерта П.П.: проаналізував реалізації універсального синтаксису виклику функцій у сучасних мовах програмування, запропонував варіант його інтеграції в GoNext зі збереженням зворотної сумісності з наявним Go-кодом та правила розв'язання неоднозначностей під час пошуку функцій; Іванченко М.Г.: постановка завдання, узагальнення результатів).*
  6. Форкерт П. П., Іванченко М. Г. КОРОТКИЙ СИНТАКСИС ДЛЯ АНОНІМНИХ ФУНКЦІЙ В ДІАЛЕКТІ МОВИ ПРОГРАМУВАННЯ GO. *Тези доповідей VI Міжнародної науково-практичної інтернет конференції «ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ: МОДЕЛІ, АЛГОРИТМИ,*

*СИСТЕМИ (ITMAS – 2025)», Миколаїв, 16-17 листопада 2025, с. 531-533. URL: <https://itconf.nuos.edu.ua/2025/publications/short-syntax-for-anonymous-functions-in-go-programming-language-dialect/> (особистий внесок Форкерта П.П.: запропонував розширення граматики GoNext короткими літералами анонімних функцій та підхід до їх транпіляції з відновленням пропущених типів шляхом уніфікації з очікуваним типом функцій; Іванченко М.Г.: постановка завдання, контроль результатів).*

7. Forkert P. P., Ivanchenko M. G. IMPLEMENTING DEFAULT PARAMETER VALUES IN GO PROGRAMMING LANGUAGE DIALECT. *Тези доповідей XXIII міжнародної науково-практичної конференції «МАТЕМАТИЧНЕ ТА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ІНТЕЛЕКТУАЛЬНИХ СИСТЕМ (МПЗІС-2025)», Дніпро, 19-21 листопада 2025, с. 8-9. URL: <https://mpzis.dnu.dp.ua/wp-content/uploads/2025/11/%D0%9C%D0%9F%D0%97%D0%86%D0%A1-2025.pdf> (особистий внесок Форкерта П.П.: запропонував синтаксис значень параметрів за замовчуванням для GoNext та алгоритм їх транпіляції через генерацію допоміжних функцій з опціональними параметрами, що коректно обробляє значення, залежні від області видимості означення функції; Іванченко М.Г.: постановка завдання, аналіз результатів).*
8. Forkert P. P., Ivanchenko M. G. A GENERALIZED TRANSPILER ARCHITECTURE FOR LANGUAGES TARGETING GO. *Тези доповідей XXVI всеукраїнської науково-технічної конференції молодих вчених, аспірантів та студентів «СТАН, ДОСЯГНЕННЯ І ПЕРСПЕКТИВИ ІНФОРМАЦІЙНИХ СИСТЕМ І ТЕХНОЛОГІЙ», Одеса, 16-17 квітня 2026, с. 143-145. URL: [https://drive.google.com/file/d/1QEcwdAJ9J4nzSCRZZ5X\\_eW5JFQE7a5vh/view](https://drive.google.com/file/d/1QEcwdAJ9J4nzSCRZZ5X_eW5JFQE7a5vh/view) (особистий внесок Форкерта П.П.: описав узагальнену багатоетапну архітектуру транпілятора для мов, що компілюються*

*в Go, виокремив нормалізацію конструкцій як ключовий етап, що забезпечує модульність; Іванченко М.Г.: постановка завдання, узагальнення результатів).*

## ЗМІСТ

ВСТУП .....	21
РОЗДІЛ 1. АНАЛІЗ СУЧАСНОГО СТАНУ ДОСЛІДЖЕНЬ У ГАЛУЗІ РЕАЛІЗАЦІЇ МОВ ПРОГРАМУВАННЯ НА БАЗІ НАЯВНИХ ПЛАТФОРМ .....	29
1.1. Підходи до реалізації мов програмування.....	29
1.2. Рантайм Go та його інженерні властивості як потенційна хост-платформа .....	32
1.3. Транспіляція як механізм побудови мов поверх існуючих екосистем.....	35
1.4. Наукові та практичні роботи, дотичні до використання Go як платформи для мов.....	37
1.5. Інструментальна екосистема Go для побудови транспіляторів і мовних процесорів .....	39
1.6. Висновки до розділу .....	41
РОЗДІЛ 2. ТЕОРЕТИЧНІ ЗАСАДИ ТА АРХІТЕКТУРА ВИКОРИСТАННЯ GO ЯК ПЛАТФОРМИ ДЛЯ НОВИХ МОВ .....	43
2.1. Постановка задачі та критерії оцінювання хост-платформи .....	43
2.2. Класи мов, для яких Go є доцільною ціллю транспіляції .....	45
2.3. Загальна архітектура транспілятора до Go .....	47
2.4. Парсинг і побудова внутрішнього представлення.....	49
2.5. Семантичний аналіз і часткова типізація .....	50
2.6. Моделі трансформації мовних розширень у Go .....	52
2.7. Інтероперабельність як архітектурний принцип .....	54
2.8. Принципи проєктування нових мов на базі Go.....	55
2.9. Висновки до розділу .....	57

РОЗДІЛ 3. ЕКСПЕРИМЕНТАЛЬНА РЕАЛІЗАЦІЯ ПІДХОДУ У ВИГЛЯДІ ДІАЛЕКТУ GONEXT .....	59
3.1. Призначення GoNext у межах дослідження .....	59
3.2. Загальна архітектура GoNext-транспілятора.....	60
3.3. Принципи відбору мовних розширень для GoNext.....	62
3.4. Повнофункціональні enum-типи як приклад репрезентаційної трансформації .....	63
3.5. Конструкція <code>match</code> як приклад структурної трансформації .....	67
3.6. Значення параметрів за замовчуванням.....	69
3.7. Іменовані аргументи .....	71
3.8. Методи розширення та універсальний синтаксис виклику функцій .....	73
3.9. Узагальнені методи як похідна обраної реалізації методів розширення .	74
3.10. Короткий синтаксис для анонімних функцій.....	76
3.11. Взаємодія реалізованих мовних розширень.....	77
3.12. Інтероперабельність GoNext зі звичайним Go .....	79
3.13. Якість згенерованого коду та налагоджуваність .....	80
3.14. Узагальнення результатів GoNext на ширший клас мов програмування .....	81
3.15. Висновки до розділу .....	83
РОЗДІЛ 4. ОЦІНЮВАННЯ ДОЦІЛЬНОСТІ ПІДХОДУ ТА УЗАГАЛЬНЕННЯ НА ІНШІ МОВИ Й ПРЕДМЕТНІ ОБЛАСТІ .....	86
4.1. Логіка оцінювання результатів .....	86
4.2. Оцінювання за виконувальними властивостями .....	86
4.3. Оцінювання за виразними властивостями хост-мови .....	88
4.4. Оцінювання за інтероперабельністю .....	91

4.5. Оцінювання за наявністю інструментів для транспіляції та вартістю реалізації.....	93
4.6. Порівняння Go з альтернативними платформами .....	94
4.7. Go як цільова платформа для широкого класу мов програмування .....	95
4.8. Як писати парсери та компілятори нових мов на Go .....	96
4.9. Обмеження та ризики запропонованого підходу.....	97
4.10. Практичні рекомендації щодо застосування Go як платформи для нових мов.....	98
4.11. Типові сценарії практичного застосування підходу.....	98
4.12. Методика прийняття рішення про вибір Go як цільової платформи.....	99
4.13. Перспективні напрями подальших досліджень .....	102
4.14. Висновки до розділу .....	102
ВИСНОВКИ.....	104
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	107
ДОДАТКИ.....	117
Додаток А.....	117
Додаток Б .....	123
Додаток В.....	203

## ВСТУП

**Обґрунтування вибору теми дослідження.** У сучасній інженерії програмного забезпечення питання створення нових мов програмування постає не лише як суто теоретична проблема мовознавства програмування, а й як практична інженерна задача. Кожна значуща предметна область формує власні вимоги до виразності коду, безпечності, зручності абстрагування, супроводу, засобів інтеграції та продуктивності виконання. Загальноживані мови не завжди надають той баланс властивостей, який є бажаним у конкретному контексті. У результаті з'являються як повністю нові мови [57], так і діалекти, надмножини [66, 68], внутрішні та зовнішні предметно-орієнтовані мови [48], а також транспілятори [14, 83], що дають змогу розширювати наявні екосистеми без розроблення повноцінного власного бекенду машинного коду.

Класичний підхід до створення мови програмування передбачає повний цикл робіт: формалізацію граматики, побудову синтаксичного й семантичного аналізу, проектування проміжного представлення, реалізацію оптимізацій, генерацію машинного коду, розроблення рантайму, збирача сміття, системи завантаження модулів, засобів налагодження, збірки і розгортання. Такий шлях дає максимальний контроль, однак є надзвичайно дорогим за часом і складністю. Саме тому значна частина сучасних мов створюється не «з нуля», а на базі вже наявних платформ: Java Virtual Machine [23], GraalVM/Truffle [63, 90], LLVM [46], MLIR [3, 47], JavaScript-рушіїв [41], .NET CLR [71] та інших. Це зменшує бар'єр входу в проектування мов і дає змогу сконцентруватися на фронтенді мови, її семантиці та користувацькому досвіді.

На цьому тлі мова Go посідає особливе місце. З одного боку, вона не була спроектована як хост-платформа для гостей мов у тому вигляді, у якому такими платформами є JVM та LLVM. Вона не надає стабільного низькорівневого API до свого рантайму, не пропонує офіційного байткоду як цілі для зовнішніх компіляторів і не формує окремого проміжного

представлення, призначеного для сторонніх мов. З іншого боку, Go володіє набором інженерних властивостей, які роблять її надзвичайно привабливою як практичну основу для мовних експериментів: ефективний компілятор, стабільний інструментарій, швидка збірка, кроскомпіляція, розвинута стандартна бібліотека, велика екосистема пакетів, конкурентний рантайм з легковаговими горутинами, сучасний збирач сміття, строгі правила сумісності та зручні бібліотеки для аналізу й генерації власне Go-коду [13].

Проблема полягає в тому, що придатність Go як платформи для реалізації нових мов програмування досліджена значно менше, ніж придатність JVM або LLVM. У науковій і практичній літературі Go переважно розглядається як самостійна мова та інженерне середовище для розроблення розподілених і хмарних систем. Натомість питання використання Go-рантайму як цілі для транспіляції, як носія семантики для гостьової мови, як бази для реалізації нових мовних конструкцій і як компромісу між простотою та виразністю лишається порівняно малодослідженим. Особливо актуально це в умовах, коли багато мовних новацій потрібні не для «революційної» нової мови, а для еволюційного розширення вже наявної мови, її діалекту або предметно-орієнтованого надбудовного шару.

У цій дисертаційній роботі досліджується саме така постановка задачі: чи є рантайм Go зручним і доцільним фундаментом для побудови нових мов програмування, які саме класи мовних розширень можна ефективно реалізувати через транспіляцію в Go, які обмеження накладає така платформа, як слід архітектурно будувати транспілятор, та за яких умов запропонований підхід є практично виправданим [28, 29]. Для перевірки цих положень використовується діалект GoNext, створений як експериментальний майданчик для дослідження мовних розширень і механізмів їх ефективної трансляції у стандартний Go-код.

Актуальність теми визначається сукупністю таких факторів:

- зростанням інтересу до нових сучасних мов програмування як засобу підвищення продуктивності розробників [62];
- потребою в дешевших методах створення нових мов, які не вимагають повної реалізації власного рантайму [54];
- поширеністю Go у хмарній, серверній та інфраструктурній розробці, де нові DSL і спеціалізовані мови можуть приносити безпосередню практичну користь [87];
- наявністю у Go потужної стандартної й зовнішньої екосистеми інструментів для аналізу коду, роботи з AST, форматування та генерації коду;
- потребою теоретично й практично окреслити межі застосовності транспіляції в Go як альтернативи компіляції в байткод або машинний код.

Таким чином, актуальною науково-технічною задачею є розроблення та обґрунтування методології використання рантайму Go як платформи для побудови нових мов програмування, включно з діалектами, предметно-орієнтованими мовами та транспільованими надмножинами існуючих мов.

**Мета та завдання дослідження.** Метою дисертаційної роботи є дослідження зручності та доцільності використання рантайму Go як платформи для побудови нових мов програмування, а також розроблення й експериментальна перевірка архітектурних, алгоритмічних і програмних засобів, що забезпечують реалізацію таких мов через транспіляцію у Go.

Для досягнення поставленої мети необхідно розв'язати такі основні завдання:

1. Провести аналіз сучасних підходів до реалізації мов програмування, включно з компіляцією безпосередньо в машинний код, використанням віртуальних машин, проміжних представлень і транспіляції.

2. Дослідити особливості Go як інженерної платформи: самої мови, її рантайму, системи збірки, бібліотек та інструментів для статичного аналізу і генерації коду.
3. Визначити критерії, за якими можна оцінювати придатність рантайму Go для реалізації нових мов програмування.
4. Розробити загальну архітектуру транспілятора, що перетворює код гостьової мови в код на Go із збереженням семантики та максимальною сумісністю з екосистемою Go.
5. Дослідити способи реалізації мовних розширень, які не підтримуються стандартною мовою Go, але можуть бути подані через коректні й ефективні трансформації в її код.
6. Створити експериментальний діалект GoNext як доказ можливості додавання нових мовних конструкцій без модифікації офіційного компілятора Go.
7. Реалізувати та проаналізувати механізми транспіляції для повнофункціональних enum-типів, зіставлення із взірцем, іменованих аргументів, значень параметрів за замовчуванням, методів розширення, узагальнених методів, коротких анонімних функцій та універсального синтаксису виклику функцій.
8. Оцінити зручність, виразність, інтероперабельність і накладні витрати запропонованого підходу.
9. Узагальнити результати на ширший клас задач: транспіляцію інших існуючих мов у Go, створення нових предметно-орієнтованих мов на базі Go та побудову транспіляторів, які використовують екосистему Go.

**Об'єктом дослідження** є процеси проектування та реалізації нових мов програмування і мовних розширень на базі наявних програмних платформ.

**Предметом дослідження** є методи, моделі та програмні засоби використання рантайму Go як платформи для побудови нових мов програмування або діалектів існуючих.

**Методи дослідження:** у роботі використано методи теорії мов програмування, компіляторобудування, інженерії програмного забезпечення, статичного аналізу програм, типових трансформацій абстрактних синтаксичних дерев, порівняльного аналізу програмних платформ, а також методи дизайну й архітектурного проектування програмних систем.

Для аналізу предметної області застосовано порівняння наукових праць і інженерних платформ, зокрема робіт про LLVM, JVM, Truffle, транспіляцію та сучасні механізми мовних розширень. Для перевірки запропонованих підходів створено експериментальну імплементацію діалекту GoNext і аналіз коду, що ним генерується.

**Наукова новизна** дисертаційної роботи полягає в такому:

1. Вперше системно обґрунтовано доцільність розгляду рантайму Go як хост-платформи особливого типу для побудови нових мов програмування — не як багатомовної віртуальної машини і не як низькорівневого компіляторного фреймворку, а як цілі транспіляції для мов із високою інтеперабельністю з екосистемою Go.
2. Удосконалено підхід до оцінювання придатності Go як цілі транспіляції за рахунок комплексного врахування виконувальних властивостей, виразних можливостей хост-мови, інтеперабельності, інструментальної підтримки транспіляції та вартості реалізації мовних конструкцій.
3. Удосконалено архітектурний підхід до побудови мовних розширень у формі надмножини Go з поетапною трансформацією у звичайний Go-код, який поєднує нормалізацію конструкцій, частковий семантичний аналіз і перетворення у вузли `go/ast`, що дає змогу коректно реалізовувати конструкції, залежні від сигнатур функцій, областей видимості, контекстного відновлення типів і правил уніфікації типів Go, із збереженням двосторонньої інтеперабельності з екосистемою Go.
4. Вперше в межах єдиного транспіляційного підходу показано можливість реалізації комплексу мовних розширень поверх Go-

рантайму без модифікації офіційного компілятора Go, зокрема повнофункціональних enum-типів, зіставлення із взірцем, іменованих аргументів, параметрів за замовчуванням, методів розширення, узагальнених методів, коротких анонімних функцій та універсального синтаксису виклику функцій.

**Практичне значення** одержаних результатів полягає у тому, що запропоновані підходи дають змогу:

- істотно зменшити вартість створення нової мови програмування або її діалекту завдяки повторному використанню компілятора, рантайму та бібліотек Go;
- розробляти мовні розширення з мінімальним ризиком втрати сумісності з уже наявним кодом і пакетами на Go;
- створювати предметно-орієнтовані мови та генератори коду для окремих галузей, не розробляючи з нуля власні компілятор, рантайм, систему збірки та супровідні інструменти;
- швидко експериментувати з новими мовними розширеннями й оцінювати їх доцільність до потенційного впровадження в більших системах;
- спиратися на зрілу екосистему Go для синтаксичного розбору, семантичного аналізу, форматування та генерації коду;
- отримувати виконувані програми, що мають властивості звичайних Go-застосунків: статичну збірку, кроскомпіляцію, доступ до стандартної бібліотеки й звичні механізми розгортання.

Результати роботи можуть бути використані під час створення нових DSL для конфігурації та оркестрації систем, мов опису інтеграційних процесів, серверних мов програмування, експериментальних надмножин Go, навчальних компіляторів, а також у дослідницьких проєктах, що вивчають межі еволюційного розширення існуючих мов програмування.

**Особистий внесок здобувача.** Усі основні результати дисертаційної роботи, включно з аналізом предметної області, формулюванням критеріїв оцінювання, проєктуванням архітектури транспілятора, розробленням концепції GoNext, постановкою та аналізом експериментів, а також підготовкою тексту статей і узагальненням результатів, отримано автором особисто. Ідеї окремих публікацій розвивалися послідовно: від загального обґрунтування використання Go як хост-платформи до реалізації конкретних мовних розширень і формування цілісної моделі побудови нових мов на базі рантайму Go.

**Апробація результатів дисертації.** Основні положення та результати дисертаційної роботи доповідалися й обговорювалися на XXI та XXIII Міжнародних науково-практичних конференціях «Математичне та програмне забезпечення інтелектуальних систем (MPZIS)» (м. Дніпро, 2023, 2025), IV Всеукраїнській науково-практичній конференції молодих науковців та здобувачів вищої освіти «Сучасні науково-технічні дослідження у контексті мовного простору» (м. Дніпро, 2023), VI Всеукраїнській науково-практичній інтернет-конференції студентів, аспірантів та молодих вчених «Сучасні інформаційні системи та технології» (м. Хмельницький, 2023), VI Міжнародній науково-практичній конференції молодих вчених та студентів «Інженерія програмного забезпечення і передові інформаційні технології (SOFT TECH)» (м. Київ, 2024), IV Міжнародній науково-практичній конференції «Інформаційні технології в освіті та науці» (м. Запоріжжя, 2025), VI Міжнародній науково-практичній інтернет конференції «Інформаційні технології: моделі, алгоритми, системи (ITMAS)» (м. Миколаїв, 2025), XXVI Всеукраїнській науково-технічній конференції молодих вчених, аспірантів та студентів «СТАН, ДОСЯГНЕННЯ І ПЕРСПЕКТИВИ ІНФОРМАЦІЙНИХ СИСТЕМ І ТЕХНОЛОГІЙ» (м. Одеса, 2026), наукових конференціях за підсумками науково-дослідної роботи Дніпровського національного університету імені Олеся Гончара (м. Дніпро, 2023, 2024, 2025).

**Публікації.** Основні положення й результати дисертаційної роботи опубліковано у 11 роботах: 3 статті у наукових фахових виданнях України категорії Б та 8 тез доповідей у збірниках матеріалів наукових конференцій.

**Зв'язок роботи з науковими програмами, планами, темами.** Дисертаційна робота виконувалася відповідно до індивідуального плану підготовки здобувача ступеня доктора філософії кафедри інженерії програмного забезпечення та інформаційних технологій Дніпровського національного університету імені Олеся Гончара та пов'язана з дослідженнями в галузі інженерії програмного забезпечення, технологій компіляції, мов програмування та програмних платформ. Дослідження здійснювалось в рамках науково-дослідної роботи № ФПМ-2-22 «Розроблення програмного забезпечення аналізу та кластеризації часових рядів» 2022-2024 рр. номер держреєстрації 0122U001465 та № 58 – ФПМ-2-25 «Розроблення інформаційної технології обробки статистичних даних» 2025-27рр. номер держреєстрації 0125U002280.

**Структура та обсяг дисертації.** Дисертаційна робота складається зі вступу, чотирьох розділів, висновків та списку використаних джерел, що містить 92 найменування на 9 сторінках. Загальний обсяг дисертації – 204 сторінки, обсяг основного тексту – 116 сторінок.

# РОЗДІЛ 1. АНАЛІЗ СУЧАСНОГО СТАНУ ДОСЛІДЖЕНЬ У ГАЛУЗІ РЕАЛІЗАЦІЇ МОВ ПРОГРАМУВАННЯ НА БАЗІ НАЯВНИХ ПЛАТФОРМ

## 1.1. Підходи до реалізації мов програмування

У процесі історичного розвитку реалізацій мов програмування сформувалися два полярні підходи. З одного боку, це самодостатні компілятори та середовища виконання, які будують власну технологічну інфраструктуру, включно з генерацією машинного коду. З іншого боку, це мови, що значною мірою спираються на вже існуючі платформи, використовуючи їхні засоби типізації, виконання програм, збірки, керування пам'яттю та оптимізації [6, 18]. У сучасних умовах саме другий підхід став особливо продуктивним, адже дає змогу істотно скоротити час від задуму мови до її практичної придатності.

Найбільш «чистим» варіантом першого підходу є написання компілятора і рантайму з нуля. Такі системи забезпечують повний контроль над представленням даних, схемою викликів, моделлю пам'яті, системою типів і можливими оптимізаціями. Цей шлях був типовим для ранніх мов системного рівня і досі лишається актуальним, коли нова мова суттєво відрізняється від наявних платформ або орієнтується на специфічні апаратні властивості. Проте ці переваги досягаються високою ціною: кожний елемент інфраструктури необхідно спроектувати, реалізувати, протестувати та супроводжувати. Для академічних або прикладних проєктів із обмеженими ресурсами це часто є неприйнятним.

Саме тому широкого поширення набули проміжні та платформно-орієнтовані підходи. Класичним прикладом є JVM, яка була спроектована для мови Java, але згодом стала основою для великої кількості мов різних парадигм [51]. Схожим чином LLVM сформував інший тип екосистеми: не віртуальну машину з байткодом і JIT-компіляцією, а інфраструктуру низькорівневого проміжного представлення та оптимізувальних проходів

[11], на яку спирається багато компіляторів сучасних мов програмування. У більш пізніх роботах суттєву роль почали відігравати також платформи для реалізації мов через інтерпретацію та часткове обчислення, таких як Truffle у GraalVM [81].

У праці Крістофера Латтнера і Вікрама Адве про LLVM [45] показано, що важливим чинником успіху платформи для мов є наявність спільного низькорівневого представлення, яке одночасно достатньо абстрактне для вираження високорівневих конструкцій і достатньо формалізоване для агресивних оптимізацій. LLVM IR у SSA-формі став саме таким чинником. Завдяки йому багато мов, зокрема Rust, Swift, Zig, Julia та інші, змогли повторно використати вже готові бекенди і зосередитися на побудові фронтенду, систем типів та власної семантики. Такий досвід важливий для цієї дисертації не тому, що Go прагне бути аналогом LLVM, а тому, що він демонструє загальний принцип: платформа стає цінною тоді, коли значно зменшує обсяг інфраструктури, яку потрібно реалізовувати щоразу заново.

Іншу модель представляє собою JVM, де гостьова мова компілюється у байткод, а виконання забезпечується віртуальною машиною з потужним рантаймом, збирачем сміття, потоками, профілюванням та JIT-компіляцією. Перевагою такого підходу є багатий набір готових механізмів виконання. Недоліком є залежність від конкретної віртуальної машини, її моделі об'єктів і викликів, а також від властивостей старту, споживання пам'яті й дистрибуції програм. Мови сімейства Scala, Kotlin, Clojure або JRuby доводять ефективність такого підходу, однак також демонструють, що особливості хост-платформи істотно впливають на дизайн та можливості гостьової мови [51].

У роботах, присвячених Truffle, мова реалізується як інтерпретатор AST, який оптимізується самою платформою виконання. Така архітектура особливо цікава тим, що зменшує поріг побудови високопродуктивної мови: автор мови описує семантику на рівні вузлів дерева, а оптимізації частково бере на себе хост-платформа. Це підтверджує важливу тезу: придатність

базової платформи полягає не лише у швидкості виконання кінцевого коду, а й у тому, наскільки сильно вона зменшує інженерну складність реалізації нової мови.

Сучасний стан досліджень показує, що крім компіляції у байткод або проміжне представлення дедалі важливішим стає підхід транспіляції, тобто переводу коду на одній мові програмування у код іншої мови програмування. У огляді André Freitas і співавторів [32] показано, що зараз транспілятори використовуються в дуже широкому спектрі задач: для оптимізації, портування коду, фронтенд-розробки, забезпечення сумісності між платформами, автоматичних перетворень коду, програмної генерації та ін. Це означає, що транспіляція вже давно перестала бути «другорядною» технікою; навпаки, вона стала самостійним інструментом розроблення мов і програмних платформ.

Для цієї роботи особливо важливим є те, що транспіляція дає змогу обійти відсутність стабільного низькорівневого API рантайму. Якщо платформа не надає окремого байткоду або вбудованого механізму для зовнішніх компіляторів, але має стабільну мову, компілятор і бібліотеки для роботи з її кодом, то саме транспіляція стає природним засобом роботи з такою платформою. Цей механізм добре відомий, наприклад, у JavaScript-екосистемі [56], але він може бути застосований і до Go, хоча наслідки такого кроку відрізняються через особливості самої мови та її інструментарію.

Отже, аналіз базових підходів до реалізації мов програмування дозволяє виділити ключову ідею дисертації: існує проміжний клас платформ, які не є ані повноцінними віртуальними машинами для багатьох мов, ані чистими низькорівневими компіляторними проектами, проте можуть ефективно використовуватися як ціль для транспіляції, якщо хост-мова достатньо стабільна, виразна та має розвинені засоби аналізу й генерації коду. Саме до такого класу пропонується віднести Go.

## 1.2. Рантайм Go та його інженерні властивості як потенційна хост-платформа

У статті Russ Cox, Robert Griesemer, Rob Pike, Ian Lance Taylor і Ken Thompson [13] про Go як мову й інженерне середовище підкреслюється, що успіх Go значною мірою зумовлений не поодинокими «революційними» мовними новаціями, а цілісністю інженерного середовища: швидкою збіркою, суворою сумісністю коду між версіями Go, прозорою системою пакетів, зручними інструментами та орієнтацією на практичну розробку великих систем. Для нашого дослідження це спостереження є принциповим. Якщо мова використовується як база для створення інших мов, вирішальними стають не лише її синтаксичні чи семантичні особливості, а насамперед те, наскільки вона забезпечує стабільний фундамент для створення надбудов.

Go має низку характеристик, які відразу роблять її привабливою як платформу транспіляції.

По-перше, це ефективний компілятор із дуже швидким циклом збірки. Для автора нової мови це означає, що транспіляція в Go не породжує надмірного навантаження на загальний час ітерації «редагування-компіляція-запуск». Якщо проміжна хост-мова компілюється повільно, переваги повторного використання її рантайму можуть нівелюватися поганим користувацьким досвідом. Go, навпаки, традиційно оптимізований саме під швидкий зворотний зв'язок під час розробки [39].

По-друге, екосистема Go дає змогу компілювати код у бінарні файли та підтримує повноцінну кроскомпіляцію [16]. Для нової мови це суттєва перевага, оскільки кожна програма, скомпільована через транспіляцію у Go, а потім у бінарний файл, автоматично набуває властивостей звичайної Go-програми. У результаті автору гостьової мови не потрібно окремо проектувати схему дистрибуції, збірки чи платформної підтримки.

По-третє, Go-рантайм надає легковагові конкурентні примітиви у формі горутин і каналів, а також містить сучасний збирач сміття з малими паузами [17, 75]. Це означає, що нова мова, якщо її семантика узгоджується з

моделлю Go, може отримати потужну інфраструктуру для управління пам'яттю та конкурентністю практично «безкоштовно». У випадку мов, орієнтованих на серверні, мережеві, інтеграційні та оркестраційні задачі, ця властивість має не просто технічну, а безпосередню прикладну цінність.

По-четверте, Go має потужну стандартну бібліотеку й надзвичайно велику екосистему пакетів. Для нової мови доступ до цієї екосистеми означає різке зменшення порогу практичного використання. Одна з головних проблем молодих мов полягає не в тому, що вони не можуть виразити певну конструкцію, а в тому, що навколо них немає бібліотек для роботи з HTTP, GRPC, JSON, файловими системами, конкурентністю, криптографією, контейнерами, базами даних, конфігурацією та тестуванням. Якщо ж нова мова транспілюється в Go з високим рівнем сумісності, вона потенційно може використовувати весь цей інструментарій з першого дня.

По-п'яте, Go відзначається дуже великим стандартним набором бібліотек для аналізу та генерації власного коду. Пакети `go/parser`, `go/ast`, `go/token`, `go/types`, `go/printer`, `go/format`, а також бібліотеки з `golang.org/x/tools`, зокрема `go/packages`, `go/ssa`, `go/ast/astutil`, створюють для автора транспілятора майже унікально сприятливе середовище. Фактично Go надає офіційні та напівофіційні інструменти для тих фаз компілятора, які в багатьох мовах доводиться реалізовувати значною мірою самостійно. Це особливо важливо в контексті даної роботи, оскільки транспілятор на практиці часто має не просто синтаксично перетворити код, а також частково виконати семантичний аналіз, наприклад перевірку типів, переупорядкування аргументів, пошук функцій за сигнатурою, виведення контексту та типів для коротких функціональних літералів, а в деяких випадках і побудову власних внутрішніх проміжних представлень.

Водночас Go як хост-платформа має і обмеження у можливостях її використання. Головне з них полягає у відсутності стабільного, підтримуваного зовнішнього інтерфейсу до рантайму. Це означає, що нова мова не може напряму «цілитися» у рантайм Go на рівні байткоду або

специфікації виконання. Єдиним стабільним шляхом доступу до рантайму фактично є сама мова Go. Отже, гостьова мова має транслюватися не в абстрактне внутрішнє представлення рантайму Go, а саме у валідний Go-код, який уже компілюється офіційним інструментарієм.

Друге обмеження пов'язане з простотою самої мови Go [22]. Простота є перевагою для інженерії, але водночас накладає межі на вираження деяких складних конструкцій інших мов програмування. Якщо певна функціональність не має прямого або близького відображення у Go, автор гостьової мови змушений або імітувати її через більш громіздкі конструкції, або вводити додатковий рантайм-шар, або відмовлятися від частини семантики. Саме тому оцінювання придатності Go не може бути одновимірним: навіть розвинені засоби для транспіляції ще не гарантують, що будь-яка мовна ідея буде втілена елегантно й ефективно.

Третє обмеження полягає в семантичних відмінностях між гостьовою і хост-мовою. Наприклад, алгебраїчні типи даних, зіставлення із взірцем, методи розширення, іменовані аргументи чи значення параметрів за замовчуванням прямо не підтримуються Go. Їх можна реалізувати як похідні конструкції, але тоді доводиться окремо доводити, що отримане представлення є коректним, сумісним з Go та та може бути транспільоване в ефективний Go код.

Незважаючи на ці обмеження, саме поєднання стабільності, продуктивності, великої та різноманітної екосистеми та наявності інструментів для транспіляції робить Go дуже цікавим кандидатом на роль хост-платформи специфічного типу: не універсального байткодного середовища, а практичної основи для транспільованих мов, орієнтованих на тісну взаємодію з самим Go.

### 1.3. Транспіляція як механізм побудови мов поверх існуючих екосистем

У найзагальнішому розумінні транспіляція є перетворенням коду, написаного однією високорівневою мовою, в еквівалентний код іншою високорівневою мовою. Від традиційної компіляції її відрізняє не лише тип цільового представлення, а й інженерна мета. Транспілятор часто створюється не для максимального усунення зв'язку з хост-платформою, а навпаки, для свідомого використання її переваг.

Огляд Bastidas Fuertes та співавторів [4] показав, що транспіляція використовується в дослідженнях і індустрії не як виняток, а як стабільний метод повторного використання наявних середовищ виконання, фреймворків і платформ. Це важлива відправна точка для даної дисертації: якщо транспіляція стала загальноприйнятим інструментом у багатьох галузях, то питання про транспіляцію саме в Go є не екзотичним, а закономірним продовженням ширшого тренду.

Транспіляційний підхід має кілька переваг. Найочевидніша з них полягає у можливості використати перевірений і надійний компілятор хост-мови. Це особливо важливо тоді, коли хост-мова має сильні гарантії сумісності [35], якісні діагностичні повідомлення, налагоджені ланцюжки збірки та усталені практики розгортання. У такому разі автор нової мови може сконцентруватися на тому, що є її сутністю: синтаксисі, абстракціях, семантичних перевірках і механізмах трансформації.

Друга перевага пов'язана з інтеоперабельністю. Якщо транспільована мова добре узгоджена з хост-мовою, то код двох світів може взаємодіяти без дорогих FFI-шарів [50], маршалінгу даних або окремої моделі пакетів. Для прикладних мов це критично: мова без бібліотек часто не має шансів на прийняття, а мова, яка з першого дня може викликати наявні бібліотеки, набуває реальної практичної цінності [21].

Третя перевага полягає у здатності використовувати транспіляцію як інструмент еволюційного розширення. Якщо повністю нова мова може бути

надто радикальним кроком для команди або екосистеми, то діалект, надмножина чи DSL, що транспілюється до знайомої мови, часто є прийнятнішим. Саме такий підхід було обрано в цій роботі: GoNext не заперечує Go, а використовує його як семантичну основу та точку сумісності.

Разом із перевагами транспіляція має й типові труднощі.

По-перше, не всі мовні розширення однаково добре транспілюються в код іншої мови [8]. Синтаксичний цукор, як-от іменовані аргументи або універсальний синтаксис виклику функцій, часто можна прибрати майже без залишку. Натомість конструкції, пов'язані з новими моделями даних, контролю потоку, типізації чи поведінкою під час виконання, можуть вимагати складних обхідних схем [27, 91].

По-друге, виникає проблема якості згенерованого коду. Якщо транспілятор створює непрозорий, громіздкий або погано оптимізований результат, це погіршує налагодження, читабельність і, зрештою, довіру до інструмента. Для Go ця проблема частково пом'якшується тим, що хост-мова сама надає стандартизовані інструменти форматування і друку і абстрактних синтаксичних дерев і коду Go, тож виглядати проміжний код буде звично для людини, навіть якщо він і буде не ідіоматичним.

По-третє, складні мовні розширення часто потребують не лише синтаксичної, а і семантичної інформації. Це принципове спостереження для цієї дисертації. Наприклад, неможливо коректно реалізувати іменовані аргументи лише через локальне абстрактне синтаксичне дерево без знання сигнатури функції та поточного оточення. Аналогічно короткі функціональні літерали з пропущеними типами потребують хоча б часткової типізації для відновлення відсутньої інформації. Отже, повноцінний транспілятор для серйозних мовних розширень не може обмежуватися простими шаблонними замінами; він має включати елементи аналізу імен, типів і областей видимості [24, 26].

Усе це особливо добре узгоджується з Go, адже саме Go надає багату інфраструктуру для синтаксичного і семантичного аналізу власного коду.

Таким чином, Go є не лише ціллю транспіляції, а й зручним матеріалом для побудови самого транспілятора.

#### **1.4. Наукові та практичні роботи, дотичні до використання Go як платформи для мов**

Порівняно з JVM або LLVM, кількість наукових робіт, які прямо розглядають Go як платформу для побудови інших мов, є помітно меншою. Це саме по собі становить дослідницьку нішу. Утім, низка праць і практичних проєктів дозволяє сформувати достатню вихідну картину.

Насамперед слід відзначити роботу авторів Go над формалізацією та розвитком самої мови. Праці «Featherweight Go» [38] та «Welterweight Go» [40] демонструють, що Go розглядається не лише як інженерний інструмент, а й як об'єкт формального мовознавчого дослідження. У них описано модель узагальнень для мови Go та показано, як поєднання структурної типізації та мономорфізації впливає на дизайн мови. Для дисертації це важливо з двох причин. По-перше, це підтверджує, що навіть відносно прагматична мова Go має достатньо строгі теоретичні основи, щоб слугувати платформою для подальших узагальнень. По-друге, ця робота підказує межі, у яких варто використовувати власні механізми Go, а де доводиться будувати додаткові засоби в гостьовій мові.

У практичній площині відомі кілька проєктів, що прямо або опосередковано використовують Go як платформу для іншої мови. Проєкт Grumpy [88] демонструє можливість транспіляції Python у Go з подальшою компіляцією в машинний код. Хоча цей проєкт має обмеження і був спрямований на сумісність із Python 2.7, він важливий як підтвердження самої можливості: Go може виступати не лише мовою реалізації компілятора, а і цільовою мовою для іншої, суттєво відмінної за семантикою системи.

Проєкт Oden [76] є прикладом іншого типу: це експериментальна статично типізована функціональна мова, побудована для екосистеми Go. Вона засвідчує, що Go можна розглядати як середовище, навколо якого

доцільно будувати мови з іншими парадигмальними акцентами, якщо зберігати орієнтацію на бібліотеки та екосистему Go.

Новіші експериментальні мови, зокрема Borgo [7] — статично типізована мова з Rust-подібним синтаксисом, що компілюється в Go і додає більш виразні засоби на кшталт алгебраїчних типів даних та зіставлення із взірцем, — також показують, що ідея більш виразної мови, яка компілюється в Go, має не лише теоретичний, а й практичний інтерес. Водночас більшість таких проєктів залишаються радше інженерними експериментами або вузькоспеціалізованими рішеннями і не супроводжуються системним науковим аналізом придатності Go як цільової платформи. Ця дисертація спрямована на часткове заповнення цієї прогалини.

Окремо можна відзначити той факт, що Go може бути не тільки зручною цільовою платформою для транспіляції, а й ефективним фундаментом для побудови інтерпретаторів та віртуальних машин для виконання динамічних мов програмування. Одним з прикладів такого виду систем є мова Tengo [84], компілятор у байткод і віртуальна машина для виконання байткоду якого написана на Go.

Публікації автора логічно вписуються у цю нішу і становлять безпосередню передумову дисертаційного дослідження. У ранніх роботах було розглянуто переваги та виклики використання Go як бази для нових мов [28, 29]. Ці праці сформулювали початкову тезу: Go може бути вигідним хостом не попри те, що не є JVM, а саме завдяки іншому набору інженерних властивостей. Далі ця теза була конкретизована на прикладі окремих мовних розширень.

У статті про повнофункціональні enum-типи в GoNext [31] розглянуто питання того, як виразна високорівнева конструкція, поширена в Rust- і Haskell-подібних мовах, може бути представлена у Go. Робота цінна тим, що вона не просто додає нову синтаксичну форму, а аналізує різні варіанти представлення у рантаймі, їхню безпечність у контексті збирача сміття, сумісність із узагальненнями й можливі накладні витрати.

Стаття про окремий синтаксис зіставлення із взірцем для таких епитипів [30] демонструє наступний етап: після додавання нової функціональності виникає потреба у її зручному використанні. Це типовий сценарій розвитку мови: одна функціональність майже завжди тягне за собою набір суміжних інструментів.

Роботи про універсальний синтаксис виклику функцій, методи розширення, узагальнені методи, іменовані аргументи, значення параметрів за замовчуванням і короткі анонімні функції [24, 26, 27, 91, 92] демонструють, що GoNext використовується не як випадковий набір «фіч», а як систематичний експериментальний простір. Через нього перевіряється гіпотеза, що значну частину мовної виразності, якої бракує Go для певних сценаріїв, можна додати через транспіляцію без втрати фундаментальної сумісності з хост-мовою.

Окремо варто відзначити дослідження, присвячені методам розширення у різних мовах. Роботи про C#, Scala та динамічно типізовані мови показують, що механізм «розширення» вже наявних типів є давно визнаним інструментом побудови зручних API [20, 44]. Для нашої теми це важливо, бо однією з перспективних ролей Go як хост-платформи є не тільки створення повністю автономної нової мови, а додавання ергономічних та предметно-орієнтованих засобів поверх уже наявної екосистеми типів і бібліотек.

У підсумку аналіз наукових та практичних робіт дозволяє зробити два висновки. По-перше, Go вже фактично використовується як база для мовних експериментів, хоча ця практика ще недостатньо описана теоретично. По-друге, саме систематизація цих експериментів, формулювання критеріїв, меж і рекомендацій становить актуальну наукову проблему.

## **1.5. Інструментальна екосистема Go для побудови транспіляторів і мовних процесорів**

Окремий аспект літературного огляду стосується не стільки рантайму Go, скільки її компіляторної екосистеми. Для побудови нової мови

недостатньо лише мати цільову платформу виконання; потрібні також засоби для розробки фронтенду. Саме тут Go виявляє одну зі своїх найбільш недооцінених переваг.

Пакет `go/parser` зі стандартної бібліотеки надає повноцінний синтаксичний аналізатор Go-файлів і виразів. Пакет `go/ast` формує стабільну модель абстрактного синтаксичного дерева, а `go/token` визначає лексичні токени та засоби роботи з позиціями у вихідному коді. Пакет `go/types` забезпечує перевірку типів зі зв'язуванням імен, виведенням типів для виразів, інформацією про використання й перевизначення ідентифікаторів, обчисленням констант та побудовою об'єктної моделі. Пакети `go/printer` і `go/format` дають змогу отримувати коректний, форматований Go-код із синтаксичних дерев. Це означає, що великий шмат інфраструктури, необхідної для побудови транспілятора, уже існує в офіційному інструментарії [86].

Також варто відзначити розширений набір інструментів із модуля `golang.org/x/tools`. Пакет `go/packages` працює на рівні пакетів Go і дозволяє парсити і робити їх семантичний аналіз. `go/ast/astutil` містить утиліти для трансформування абстрактних синтаксичних дерев Go. `go/ssa` дає наближене до компіляторного проміжне представлення в SSA-формі, придатне для статичного аналізу, побудови графів викликів та більш глибоких перевірок. Інструменти на зразок `stringer` і `goyacc` показують, що в екосистемі Go давно присутня культура генерації коду та мовних процесорів [89].

Поза стандартними пакетами сформувалася й багата екосистема бібліотек для парсингу: `participle`, `pigeon`, `gocc`, `gogll` та інші. Це означає, що автор нової мови може обрати кілька різних маршрутів реалізації:

- побудувати повністю власний парсер;
- розширити або частково повторно використати модель синтаксису Go;
- використовувати генератори парсерів і лексерів;
- будувати гібридну систему, де ранні стадії граматики власні, а пізні опираються на синтаксичне дерево і типову систему Go.

З цього випливає важливий наслідок. Коли йдеться про доцільність використання певної платформи для створення нових мов, слід оцінювати не лише характеристики виконання, а й доступність інструментів для побудови фронтенду. І саме тут мова Go відрізняється від багатьох інших хост-мов тим, що вона не просто дає компілятор і стандартну бібліотеку; вона надає розробнику готовий набір будівельних блоків для створення токенизаторів, парсерів, аналізаторів, генераторів, трансформерів і статичних перевірок.

Це особливо корисно для діалектів і надмножин Go. Якщо нова мова стилістично й семантично близька до Go, можна повторно використати не лише рантайм Go та його бібліотеки, а й частину компіляторної інфраструктури. Саме тому GoNext у цій дисертації розглядається як показовий приклад. Він дозволяє дослідити не лише абстрактну можливість транспіляції в Go, а й конкретний ефект від повторного використання Go-екосистеми на етапах аналізу, типізації та генерації.

## **1.6. Висновки до розділу**

У першому розділі проаналізовано сучасний стан досліджень, пов'язаних із побудовою нових мов програмування на базі вже наявних платформ. Показано, що в сучасних імплементаціях різних мов програмування домінує тенденція до повторного використання рантаймів, компіляторних бекендів, віртуальних машин та інструментальних екосистем. Класичними й добре дослідженими прикладами таких платформ є JVM, LLVM і Truffle/GraalVM.

На основі аналізу літератури встановлено, що транспіляція є повноцінним і широко застосовуваним методом побудови мов та мовних надбудов. Вона особливо доцільна тоді, коли хост-платформа не надає стабільного низькорівневого API або байткоду для зовнішніх компіляторів, але має стабільну мову, компілятор і потужний набір засобів для аналізу та генерації коду.

Досліджено інженерні властивості Go, що роблять її перспективною хост-платформою: швидку збірку, статичну компіляцію у бінарні файли, кроскомпіляцію, конкурентний рантайм, сучасний збирач сміття, розвинену стандартну бібліотеку, велику екосистему пакетів і потужні засоби аналізу та генерації коду. Водночас зазначено потенційні обмеження підходу: відсутність окремого стабільного інтерфейсу до рантайму, простоту хост-мови та можливу невідповідність конструкцій гостьової мови програмування конструкціям мови Go, що може призводити до громіздкого або неефективного транспільованого коду.

Огляд наукових і практичних робіт показав, що Go вже використовується як основа для мовних експериментів, проте абсолютна більшість подібних проектів є прикладними і не базуються на наукових та академічних працях. Саме тому дослідження критеріїв придатності Go як хост-платформи, архітектури транспілятора і моделей реалізації мовних розширень поверх Go-рантайму наразі є актуальною проблемою.

Отже, літературний огляд підтверджує наукову доцільність подальшого дослідження та слугує підґрунтям для побудови власної моделі використання Go як платформи для нових мов програмування.

## РОЗДІЛ 2. ТЕОРЕТИЧНІ ЗАСАДИ ТА АРХІТЕКТУРА ВИКОРИСТАННЯ GO ЯК ПЛАТФОРМИ ДЛЯ НОВИХ МОВ

### 2.1. Постановка задачі та критерії оцінювання хост- платформи

Щоб досліджувати доцільність використання конкретної мови або рантайму як платформи для побудови інших мов, недостатньо описати її технічні переваги. Необхідно визначити систему критеріїв, яка дозволяє структуровано аналізувати той чи інший аспект мови. Надалі пропонується оцінювати Go як хост-платформу за п'ятьма групами характеристик: виконувальні властивості, виразні властивості хост-мови, інтероперабельність, наявність інструментів для трансляції і економіка реалізації.

Перша група критеріїв охоплює виконувальні властивості платформи. Якщо нова мова або DSL після трансляції має працювати в реальних системах, важливими є:

- продуктивність виконання отриманого коду;
- модель керування пам'яттю;
- наявність конкурентних примітивів;
- ефективність збирача сміття;
- швидкість запуску і зручність розгортання;
- підтримка кроскомпіляції для різних цільових платформ.

Go добре задовольняє ці вимоги в тих сценаріях, де гостьова мова може бути перетворена у ідіоматичний Go без потреби в додатковій інфраструктурі виконання. Однак якщо нова мова потребує, наприклад, складної моделі рефлексії, динамічної модифікації коду під час виконання або спеціалізованого JIT-механізму, то Go як платформа може виявитися незручною.

Друга група критеріїв стосується виразних властивостей хост-мови. Під цим мається на увазі не те, наскільки виразною і лаконічною є мова Go сама по собі, а те, наскільки її семантика дозволяє реалізовувати в ній інші конструкції. Існують три типові випадки:

1. Конструкція гостьової мови має пряме або майже пряме відображення в Go. Тоді транспіляція реалізується як локальна AST-трансформація або як заміна синтаксичного цукру на еквівалентну базову форму.
2. Конструкція гостьової мови не має прямого аналога в Go, але може бути коректно виражена через комбінацію наявних засобів Go. У такому разі з'являються накладні витрати на трансформацію, додаткові допоміжні структури або синтетичні функції.
3. Конструкція гостьової мови суперечить фундаментальним властивостям Go-рантайму або моделі типізації. У такому разі або потрібна окрема інфраструктура виконання, або реалізація стає надто дорогою, або доводиться обмежувати семантику мовної конструкції.

Третя група критеріїв охоплює інтероперабельність. Для прикладних мов програмування дуже важливо не лише те, що вони можуть бути реалізовані, а й те, чи можуть вони використовувати вже наявний код [82]. У контексті Go це означає такі питання:

- чи може гостьова мова імпортувати звичайні Go-пакети;
- чи може згенерований код виглядати достатньо «звичним» для компілятора, інструментів та IDE;
- чи може Go-код використовувати результати, створені гостьовою мовою;
- чи не руйнують нові мовні розширення сумісність на рівні пакетів, типів, викликів та механізму збірки.

Четверта група критеріїв пов'язана з наявністю інструментів для транспіляції [5]. Навіть дуже продуктивний рантайм не є придатним фундаментом для нової мови, якщо у нього відсутні засоби для аналізу,

генерації та перевірки коду. У цьому аспекті Go має суттєву перевагу, оскільки офіційно підтримує пакети для парсингу, AST, типізації, друку та форматування коду. Для побудови транспілятора це означає, що значна частина роботи переноситься з області «створити з нуля» в область «скомпонувати вже наявні засоби».

П'ята група критеріїв стосується економіки реалізації. Під цим мається на увазі не швидкість виконання згенерованого коду, а вартість створення та підтримки самого компілятора. Для академічних, експериментальних і навіть багатьох прикладних мов саме цей критерій є визначальним. Платформа доречна тоді, коли:

- дозволяє швидко реалізувати мовні ідеї;
- мінімізує обсяг коду рантайму;
- не вимагає побудови повного бекенду машинного коду;
- спрощує тестування і налагодження;
- забезпечує передбачувану еволюцію разом із власною екосистемою [12].

Запропонована п'ятикомпонентна модель оцінювання є важливою для подальших розділів. Вона дає змогу не просто стверджувати, що Go «зручний» або «незручний», а розкласти це судження на окремі вимірювані або принаймні якісно порівнювані аспекти.

## **2.2. Класи мов, для яких Go є доцільною ціллю транспіляції**

На основі сформульованих критеріїв можна виокремити класи мов, для яких використання Go як хост-платформи є особливо доцільним.

Перший клас становлять еволюційні розширення самої мови Go. Йдеться про діалекти, надмножини або «метамови», які навмисно зберігають основну семантичну модель Go, але додають відсутні мовні зручності: алгебраїчні типи даних, зіставлення із взірцем, універсальний синтаксис виклику функцій, методи розширення, іменовані аргументи, значення параметрів за замовчуванням, більш короткий функціональний синтаксис

тощо. Саме для цього класу задач Go є майже ідеальною ціллю: інтероперабельність із бібліотеками природна, більшість трансформацій можуть бути виражені як перетворення у стандартний код, а рантайм залишається тим самим.

Другий клас становлять предметно-орієнтовані мови для серверної та інфраструктурної розробки. Це можуть бути мови опису workflow, конфігураційних сценаріїв, політик доступу, систем інтеграції, декларативних правил обробки подій, вбудованих запитів або тестових сценаріїв [9, 73]. Потреба в таких мовах закономірно виникає під час побудови інформаційних систем: сучасна інформаційна система зазвичай поєднує велику кількість сервісів, інтеграцій і конфігураційних сценаріїв, які зручніше описувати спеціалізованими декларативними засобами, ніж кодом мовою загального призначення. Якщо такі мови мають бути виконуваними, масштабованими, кросплатформними й добре інтегрованими з мережевими сервісами, Go як платформа може бути виграним вибором. Розробник DSL може створити фронтенд із високим рівнем декларативності, а виконання делегувати вже готовому рантайму Go.

Третій клас — нові мови, семантично близькі до системного чи серверного програмування, але зі зручнішими засобами моделювання даних, обробки помилок або композиції функцій. Такі мови можуть бути спроектовані як «Go-подібні, але більш виразні». Вони не повинні повністю відриватися від екосистеми Go, а навпаки, мають використовувати її як джерело бібліотек і спосіб доставки коду в продакшн-середовище.

Четвертий клас — навчальні та дослідницькі мови. Для них надзвичайно важливо зменшити інфраструктурну складність реалізації, щоб основні зусилля були зосереджені на самій дослідницькій ідеї. Якщо нова мовна конструкція може бути перевірена через транспіляцію в Go, це дозволяє досліднику не витратити роки на побудову власного машинного бекенду [49, 52].

Водночас є й класи мов, для яких Go менш придатний. До них належать:

- мови з радикально динамічною семантикою, які потребують розвиненої рефлексії над середовищем виконання;
- мови, що орієнтуються на агресивні специфічні оптимізації компілятора;
- мови з повністю іншою моделлю керування пам'яттю, наприклад із ручним контролем пам'яті або з нетиповими правилами життя об'єктів;
- мови, яким потрібен стандартний байткод або окремий багатоцільовий бекенд.

Таким чином, питання доцільності транспіляції в Go має не абсолютний, а класово-залежний характер. Саме це відрізняє запропонований у роботі підхід від спрощеного твердження «Go підходить для всього». Дисертація стверджує не універсальну, а контекстну придатність Go.

### **2.3. Загальна архітектура транспілятора до Go**

На основі проведеного аналізу пропонується архітектура транспілятора, орієнтована на реалізацію мов або мовних розширень, що транспілюються до Go. У загальному випадку [2] такий транспілятор складається з таких основних етапів:

1. Лексичний аналіз і синтаксичний розбір гостьової мови.
2. Побудова внутрішнього абстрактного синтаксичного дерева.
3. Зв'язування імен, пакетів і просторів видимості.
4. Часткова або повна семантична перевірка типів.
5. Нормалізація високорівневих мовних конструкцій до внутрішньої проміжної форми.
6. Перетворення внутрішнього представлення в модель вузлів `go/ast`.
7. Постобробка, додавання імпортів, синтетичних допоміжних оголошень, перевірка коректності.
8. Генерація відформатованого Go-коду.

9. Делегування згенерованих файлів стандартному ланцюжку `go build`, `go test`, `go run` або іншим офіційним інструментам.

Слід підкреслити, що ця архітектура є загальною і не зводиться лише до GoNext [25]. Вона може бути застосована до будь-якої мови, що обирає Go як ціль. Разом із тим конкретне наповнення етапів залежатиме від того, чи є гостьова мова близькою до Go, чи має власну незалежну граматику та семантику.

Для діалектів і надмножин Go існують дві характерні стратегії.

Перша стратегія — мінімальне розширення Go-граматики. У цьому разі синтаксис максимально наближений до Go, а нові конструкції вбудовуються як локальні розширення. Перевага підходу полягає в тому, що значну частину наявної інфраструктури можна використовувати повторно. Наприклад, внутрішнє AST може бути побудоване так, щоб максимально легко перетворюватися на `go/ast`, а частини коду, які не містять розширень, фактично проходили транспілятор транзитом.

Друга стратегія — побудова повністю власного фронтенду з подальшою трансформацією у Go. Вона доцільна тоді, коли нова мова значно відрізняється від Go синтаксично або семантично. У такому випадку Go виступає лише цільовою мовою, а не джерелом синтаксичної структури. Проте навіть тут пакети для типізації, генерації імпортів, форматування й аналізу пакетів можуть бути корисними на фінальних стадіях.

У запропонованій архітектурі особливе місце посідає етап нормалізації. Його призначення полягає в тому, щоб перетворити розмаїття зовнішніх мовних конструкцій на обмежену кількість внутрішніх форм. Наприклад:

- іменовані аргументи нормалізуються до впорядкованого списку позиційних аргументів;
- значення параметрів за замовчуванням нормалізуються до синтетичних допоміжних функцій;
- методи розширення нормалізуються до звичайних функцій із додатковим механізмом пошуку на місцях виклику;

- match-конструкції нормалізуються до викликів сгенерованих функцій розбору варіантів або до розгорнутого умовного розгалуження;
- enum-типи нормалізуються до певної обраної схеми представлення у рантаймі.

Виграш від такого поділу полягає в тому, що реалізація транспілятора набуває модульності. Кожна мовна конструкція не мусить знати, як друкувати Go-код на найнижчому рівні; їй достатньо перевести себе в одну з канонічних внутрішніх форм.

## 2.4. Парсинг і побудова внутрішнього представлення

У задачі побудови мов поверх Go питання парсингу залежить від «відстані» між гостьовою і хост-мовою. Якщо мова схожа на Go, часто доцільно зберігати структуру, подібну до `go/ast`, і додавати власні вузли лише для нових конструкцій. Це спрощує подальшу трансформацію, оскільки звичні конструкції — оголошення, виклики, блоки, вирази, типові літерали, імпорти — уже мають усталену інтерпретацію.

Для мов із самостійною граматикою доцільно будувати окреме внутрішнє абстрактне синтаксичне дерево, не прив'язане до структури `go/ast`. Таке представлення має задовольняти кільком вимогам, продиктованим специфікою Go-орієнтованої транспіляції. По-перше, воно повинно підтримувати окремі етапи зв'язування імен і перевірки типів, оскільки без них неможливо коректно розв'язати виклики функцій, що залежать від контексту (наприклад, іменовані аргументи чи методи розширення). По-друге, AST має бути придатним як до локальних, так і до глобальних трансформацій, бо перетворення мовних конструкцій нерідко вимагає доступу до інформації з інших вузлів чи навіть інших файлів. По-третє, дерево має допускати усунення синтаксичного цукру без втрати семантики — інакше пізніші етапи трансформації втрачають контекст, потрібний для генерації коректного Go-коду. Нарешті, представлення має допускати

систематичне зведення до вузлів `go/ast`, тобто кожен вузол дерева повинен мати визначений шлях трансляції до відповідного вузла `go/ast`.

У практиці побудови транспіляторів на Go можливі декілька підходів до синтаксичного аналізу.

Перший — рекурсивний спуск із власним лексером. Це найбільш контрольований та зручний варіант для мов із великою кількістю контекстно-залежних рішень.

Другий — генерація парсера на базі граматики, наприклад через `go yacc`, `parser`, `goc`, `gog11` або інші бібліотеки. Такий шлях доцільний, коли потрібно швидко стабілізувати формальну граматику й спиратися на стандартні класи парсерів.

Третій — часткове повторне використання інфраструктури Go, якщо нова мова є її надмножиною. У цьому випадку можна зберігати окремі вузли абстрактного синтаксичного дерева для розширень і транслювати решту без глибоких змін.

У межах цієї роботи ключовою є не конкретна техніка реалізації, а принцип: транспілятор у Go має оперувати не лише текстом, а структурою програми. Просте текстове переписування швидко вичерпує себе на серйозних мовних розширеннях. Тому центральним об'єктом стає AST, над яким виконуються семантичні перетворення.

## **2.5. Семантичний аналіз і часткова типізація**

Однією з головних відмінностей «серйозного» транспілятора від простого препроцесора є потреба у семантичному аналізі. У багатьох випадках неможливо правильно згенерувати результуючий Go-код, не знаючи, який саме об'єкт мається на увазі, яким є його тип, які функції, змінні та константи доступні в поточній області видимості, як працюють узагальнення та ін.

У контексті Go-орієнтованих мов семантичний аналіз зазвичай повинен розв'язувати щонайменше такі задачі:

- побудова таблиць символів для пакетів, типів, функцій, методів і локальних змінних;
- перевірка коректності використання ідентифікаторів та імпортів;
- зіставлення місць виклику з конкретними сигнатурами функцій;
- уніфікація типів там, де трансформація залежить від правил Go;
- відновлення пропущених частин типів із контексту;
- виявлення неоднозначностей, які слід або заборонити, або розв'язати детерміновано.

Показовими є кілька конкретних сценаріїв.

Іменовані аргументи не можна коректно транспілювати, якщо невідомий порядок параметрів функції. Це означає, що компілятор мусить знати точну сигнатуру цілі виклику, навіть якщо вона імпортована з іншого пакета. Отже, потрібно принаймні частково завантажувати зовнішні пакети або їхні сигнатури типів [27].

Короткі анонімні функції без явно вказаних типів параметрів неможливо коректно перетворити без знання очікуваного типу функції в поточному контексті. Це вимагає аналізу функцій вищого порядку, локального виведення типів і врахування узагальнень [91].

Методи розширення та виклики у формі універсального синтаксису виклику функцій вимагають алгоритму пошуку функцій за іменем та уніфікованою сигнатурою першого параметра. Це вже не просто локальна синтаксична трансформація, а модифікація правил резолюції виклику [24].

Саме тому запропонована в роботі архітектура спирається на частковий семантичний аналіз ще до фінальної трансформації. Це означає, що не всі семантичні задачі гостьової мови повинні бути остаточно вирішені на цій стадії; достатньо зібрати ту інформацію, без якої неможливо згенерувати коректний Go-код. У багатьох випадках повну фінальну перевірку частини інваріантів можна делегувати вже самому Go-компілятору після генерації.

Цей підхід має важливу інженерну перевагу: транспілятор не дублює повністю весь хост-компілятор, а реалізує лише той семантичний мінімум, який потрібен для коректної трансформації. Саме це є одним із проявів доцільності використання Go як хост-платформи.

## **2.6. Моделі трансформації мовних розширень у Go**

На основі аналізу можливостей, реалізованих у GoNext, запропонована наступна класифікація перетворень, корисна не лише для цього конкретного діалекту, а й для ширшого кола мовних систем.

### **2.6.1. Синтаксична трансформація**

До цього класу належать конструкції, які існують лише на синтаксичному рівні гостьової мови і не потребують окремого представлення у рантаймі. Прикладами є іменовані аргументи, універсальний синтаксис виклику функцій у простій формі та незначні варіації синтаксису анонімних функцій [27, 92]. У таких випадках транспілятор:

- аналізує нову конструкцію;
- перевіряє її контекстну коректність;
- переписує її в еквівалентну стандартну Go-форму;
- не додає додаткових структур даних чи механізмів виконання.

Перевага цього класу полягає у відсутності накладних витрат під час виконання. Обмеження полягає в тому, що такий спосіб придатний лише для конструкцій, чия семантика повністю редукується до вже наявної в Go.

### **2.6.2. Структурна трансформація**

Цей клас охоплює конструкції, для яких недостатньо просто переписати зовнішню форму запису. Потрібно додати допоміжні оголошення, синтетичні функції, нові проміжні сутності або розгорнути високорівневу конструкцію у більш громіздку, але семантично еквівалентну форму.

Прикладами є:

- значення параметрів за замовчуванням через допоміжні `{FUNCTION_NAME}__default`-функції [26];
- `match`-конструкції через виклики сгенерованих методів розбору варіантів [30];
- методи розширення через трансформацію до звичайних функцій і переписування місць виклику [24];
- короткі анонімні функції через повні літерали функцій із відновленими типами [91].

Такий клас перетворень зазвичай має нульові або дуже малі накладні витрати під час виконання, але вимагає складнішого семантичного аналізу і може робити згенерований код більшим.

### 2.6.3. Репрезентаційна трансформація

Найскладніший клас стосується конструкцій, які вимагають нової моделі даних, хоча й побудованої з наявних елементів Go. Прикладом є повнофункціональні `enum`-типи [31], які треба реалізувати так, щоб:

- значення могли представляти один із варіантів;
- були доступні операції розбору поточного варіанта;
- зберігалася типобезпечність;
- підтримувалася інтероперабельність із Go;
- не виникало конфліктів із роботою збирача сміття.

Саме тут найяскравіше проявляються обмеження хост-платформи. Якщо мовне розширення потребує нового представлення у рантаймі, автор транспілятора фактично виконує мікропроєктування рантайму всередині рамок Go. Це найбільш цікавий з наукового погляду клас перетворень, оскільки він дає змогу визначити, де саме проходить межа між «Go легко виражає цю функціональність» і «Go дозволяє це лише через складний компроміс».

## 2.7. Інтероперабельність як архітектурний принцип

Для більшості нових мов інтероперабельність розглядається як приємна, але необов'язкова властивість [10]. У цій роботі вона, навпаки, виступає одним із фундаментальних архітектурних принципів. Це зумовлено двома причинами.

По-перше, головна практична цінність використання Go як хост-платформи полягає саме в доступі до її екосистеми [80]. Якщо після введення нового мовного шару ця перевага зникає, значна частина доцільності підходу також зникає.

По-друге, інтероперабельність дисциплінує дизайн гостьової мови. Вона змушує автора не просто вигадувати цікаві конструкції, а шукати такі форми семантики, які можуть співіснувати з наявними типами, пакетами, правилами видимості й механізмами збірки.

У межах дисертації пропонується розрізнити односторонню та двосторонню інтероперабельність.

Одностороння інтероперабельність означає, що гостьова мова може використовувати Go-бібліотеки. Для багатьох проєктів цього вже достатньо, оскільки дозволяє писати прикладні програми на новій мові, не відмовляючись від зрілої бази пакетів.

Двостороння інтероперабельність означає додатково, що код, згенерований із гостьової мови, може споживатися звичайним Go-кодом без знання про внутрішні деталі гостьової мови. Це складніша, але набагато цінніша властивість. Вона дозволяє поступове впровадження нової мови в існуючі проєкти, поєднання файлів гостьової мови та Go, побудову бібліотек змішаного походження та еволюційне розгортання без «великих переписувань» [53].

З цього принципу випливають важливі практичні вимоги:

- генерований код має бути максимально близьким до звичайного Go;

- нові мовні конструкції не повинні вимагати модифікації інструментів на стороні споживача;
- типи, функції й методи повинні мати зрозумілі аналоги у Go;
- за можливості, синтетичні елементи повинні бути приховані від користувача гостьової мови, але лишатися коректними з погляду хост-компілятора.

Саме цей принцип згодом пояснює, чому в GoNext для багатьох можливостей було обрано, можливо, не найкоротші теоретично, але найпрагматичніші з погляду інтероперабельності схеми трансформації.

## **2.8. Принципи проєктування нових мов на базі Go**

На основі попереднього аналізу в роботі формулюється набір принципів, дотримання яких підвищує шанси на успішну реалізацію нової мови поверх Go.

### **Принцип 1. Мінімальна семантична відстань до Go там, де це не шкодить задачі**

Якщо мова прагне скористатися сильними сторонами Go, не слід без потреби розривати семантичну близькість до неї. Чим ближчими є базові моделі типів, пакетів, викликів і виконання, тим дешевше транспіляція й вища інтероперабельність.

### **Принцип 2. Нове мовне розширення повинно мати чітку схему транспіляції**

Кожна мовна конструкція повинна проєктуватися не лише «з погляду зовнішнього синтаксису», а й із погляду того, у що вона буде транспільована. Якщо конструкція виглядає красиво, але її схема транспіляції неочевидна або занадто дорога, реалізація такої конструкції на обраній платформі є недоцільною.

### **Принцип 3. Перевага локальних трансформацій над глобальними**

Під локальною трансформацією мається на увазі таке перетворення, для якого достатньо найближчого контексту: одного виразу, одного виклику, одного оголошення або, щонайбільше, одного файлу. Наприклад, якщо нова конструкція може бути зведена до звичайного Go-коду без зміни інших функцій, типів і модулів, це є локальна трансформація. Натомість глобальна трансформація вимагає перебудови ширшого фрагмента програми: узгодженої зміни багатьох оголошень, поширення нової семантики по ланцюжку викликів, аналізу кількох пакетів або генерації додаткової інфраструктури виконання.

Певага локальних трансформацій полягає в тому, що вони простіші в реалізації, легші для перевірки й налагодження та краще сумісні з інкрементальною компіляцією. Чим менший обсяг програми треба «переписувати» заради однієї мовної конструкції, тим передбачуванішою є транспіляція і тим менший ризик побічних ефектів в інших частинах системи.

### **Принцип 4. Максимальне використання офіційних інструментів Go**

Транспілятор повинен, за можливості, спиратися на `go/packages`, `go/types`, `go/ast`, `go/format` та інші бібліотеки екосистеми, а не дублювати їхню функціональність. Це не лише зменшує обсяг реалізації, а й підвищує майбутню стійкість системи.

### **Принцип 5. Поділ відповідальності між транспілятором і Go-компілятором**

Не всі помилки повинні діагностуватися самим транспілятором. Якщо деякі інваріанти природно перевіряються Go-компілятором після трансформації, їх слід делегувати йому. Транспілятор повинен виявляти насамперед ті помилки, без яких неможливо виконати коректне перетворення.

## **Принцип 6. Якість згенерованого коду є частиною дизайну мови**

Оскільки проміжним продуктом транспіляції є Go-код, його читабельність, структура і передбачуваність мають велике значення. Погано структурований згенерований код ускладнює налагодження, тестування, інтеграцію з наявними інструментами та довіру до мови в цілому.

## **Принцип 7. Інтероперабельність важливіша за «ідеальну» реалізацію окремої конструкції**

У багатьох випадках можливі кілька способів трансформації нової конструкції до Go. Найелегантніший з теоретичного погляду варіант не завжди є найкращим з практичної точки зору, особливо якщо він погіршує сумісність із пакетами, типами або правилами збирача сміття.

## **2.9. Висновки до розділу**

У другому розділі сформульовано теоретичні засади використання Go як хост-платформи для нових мов програмування. Запропоновано модель оцінювання доцільності використання Go за кількома критеріями, яка охоплює виконувальні властивості, виразні можливості хост-мови, інтероперабельність, наявність інструментів для транспіляції та економіку реалізації.

Показано, що Go є особливо доречною ціллю транспіляції для діалектів і надмножин самої мови, предметно-орієнтованих мов серверно-інфраструктурного профілю, нових виразніших Go-подібних мов, а також навчальних і дослідницьких мовних систем [43]. Водночас визначено класи мов, для яких Go менш придатний через радикально іншу семантику або вимогу до окремого спеціалізованого рантайму.

Розроблено узагальнену архітектуру транспілятора до Go, що включає парсинг, побудову внутрішнього представлення, зв'язування імен, частковий семантичний аналіз, нормалізацію мовних конструкцій, трансформацію у вузли `go/ast` та генерацію відформатованого Go-коду. Запропоновано

класифікацію перетворень на синтаксичні, структурні та репрезентаційні, що дозволяє систематизувати способи реалізації різних мовних розширень.

Обґрунтовано центральну роль інтеперабельності як архітектурного принципу та сформульовано набір практичних принципів проектування нових мов на базі Go. Отримані результати створюють методологічну основу для подальшого розгляду конкретної експериментальної реалізації — діалекту GoNext.

## РОЗДІЛ 3. ЕКСПЕРИМЕНТАЛЬНА РЕАЛІЗАЦІЯ ПІДХОДУ У ВИГЛЯДІ ДІАЛЕКТУ GONEXT

### 3.1. Призначення GoNext у межах дослідження

GoNext у цій дисертації розглядається не як самоціль і не як спроба «замінити Go». Натомість, його роль є методологічною. Це експериментальна надмножина Go, створена для перевірки гіпотези про те, що рантайм Go може слугувати ефективною платформою для побудови нових мовних конструкцій і навіть нових мов, якщо вони реалізуються шляхом транспіляції в коректний Go-код.

Такий вибір експериментального об'єкта був обумовлений кількома причинами.

По-перше, використання діалекту дозволяє краще контролювати умови дослідження. Якщо брати повністю нову мову з радикально іншим синтаксисом і семантикою, буде важко відокремити властивості самої мовної ідеї від властивостей хост-платформи. У випадку GoNext базова модель програмування лишається близькою до Go, тому значно легше простежити, які саме нові мовні розширення додаються, як вони транспілюються й які накладні витрати вони мають.

По-друге, діалект є зручним середовищем для перевірки сумісності з наявною екосистемою. Оскільки більшість коду лишається Go-подібною, можна прямо досліджувати, наскільки легко нові мовні розширення співіснують із пакетами стандартної бібліотеки та зовнішніми Go-модулями.

По-третє, GoNext демонструє важливий практичний сценарій: далеко не завжди потрібна цілком нова мова. Часто достатньо надбудови над уже знайомою мовою, яка усуває окремі точки тертя, робить моделювання даних виразнішим, а код компактнішим. Якщо така надбудова може бути створена без внесення змін в офіційний компілятор, зі збереженням сумісності та нульовими або малими накладними витратами під час виконання, це саме по собі є цінним результатом.

Отже, GoNext у роботі використовується як контрольований експериментальний майданчик, на якому досліджуються:

- принципова можливість реалізації нових мовних розширень поверх Go;
- архітектурні вимоги до транспілятора;
- типові класи перетворень;
- межі коректної й ефективної інтеграції нових конструкцій з хост-рантаймом;
- ступінь інтероперабельності зі звичайним Go-кодом.

### 3.2. Загальна архітектура GoNext-транспілятора

Узгоджено з моделлю, викладеною в попередньому розділі, транспілятор GoNext будується як багатостадійна система, у якій кожна стадія розв'язує окремий клас задач.

На вході транспілятор отримує набір вихідних файлів GoNext, які за своєю структурою максимально наближені до звичайних Go-файлів: мають оголошення пакета, імпорти, типи, функції, методи, вирази та оператори. Мовні розширення в GoNext не замінюють базову структуру Go-файлу, а лише точково додають нові конструкції поверх неї. Завдяки цьому GoNext лишається структурно близьким до Go, що спрощує як транспіляцію, так і подальшу інтеграцію зі звичними інструментами екосистеми.

Далі виконується синтаксичний аналіз з побудовою внутрішнього AST. На цьому етапі важливо не лише зафіксувати наявність нових синтаксичних форм, а й зберегти структуру областей видимості та типові для Go елементи. Оскільки значна частина коду GoNext є звичайним Go, внутрішнє представлення тяжіє до моделей `go/ast`, але доповнюється вузлами для `enum`-оголошень, `match`-конструкцій, модифікаторів розширення, іменованих аргументів, коротких функціональних літералів і параметрів зі значеннями за замовчуванням.

Після побудови AST виконується етап семантичної підготовки:

- формується інформація про локальні й імпортовані визначення;

- завантажуються сигнатури функцій, типів і методів із поточного пакета та імпортів;
- будується інформація, необхідна для контекстного трансформаційного перетворення;
- позначаються ті місця, де для трансформації потрібна уніфікація типів або відновлення типів, що були не потрібні в контексті GoNext, але потрібні для сгенерованого Go коду.

Наступний етап — нормалізація мовних конструкцій. На цьому етапі різні зовнішні форми запису зводяться до невеликої кількості канонічних внутрішніх представлень, з якими далі працює транспілятор. Наприклад, іменовані аргументи перетворюються на позиційні, для функцій з параметрами за замовчуванням додаються синтетичні `{FUNCTION_NAME}__default`-функції, а методи розширення замінюються на звичайні функції з додатковою ознакою, що вони беруть участь у механізмі резолюції методів.

Після цього відбувається безпосередня трансформація абстрактного синтаксичного дерева GoNext у вузли `go/ast`. На цій стадії всі специфічні для GoNext конструкції мають бути усунені або зведені до звичайних вузлів і комбінацій вузлів, що мають прямиий аналог у Go:

- `enum`-типи стають звичайними структурами або іншими допустимими представленнями;
- `match` стає викликом методу `Match` чи серією умовних конструкцій;
- методи розширення перетворюються на звичайні функції;
- іменовані аргументи переставляються в правильний порядок;
- короткі анонімні функції стають повними літералами функцій;
- параметри за замовчуванням реалізуються через допоміжні обгортки.

Нарешті, за допомогою генератора коду формується звичайний відформатований Go-код, який потім може бути безпосередньо скомпільований стандартними інструментами `go build` або `go test`. Таким

чином, GoNext не створює альтернативний режим виконання; він завершує свою роботу ще до запуску офіційного компілятора.

### 3.3. Принципи відбору мовних розширень для GoNext

Набір мовних розширень, реалізованих у GoNext, було сформовано не випадково, а відповідно до дослідницької мети роботи: перевірити, які класи мовних конструкцій доцільно реалізовувати поверх Go шляхом транспіляції без модифікації офіційного компілятора. Тому до GoNext включалися лише ті розширення, які одночасно мали практичну цінність для розробника і дозволяли перевірити окремий аспект придатності Go як хост-платформи.

Першим принципом відбору було покриття різних типів трансформацій. Частина цих розширень належить до синтаксичних перетворень, що повністю елімінуються під час транспіляції, як-от іменовані аргументи. Інші є структурними перетвореннями, що спираються на нормалізацію викликів, сигнатур і допоміжних конструкцій, як-от значення параметрів за замовчуванням, `match`, методи розширення та універсальний синтаксис виклику функцій. Окрему групу становлять репрезентаційні розширення, насамперед повнофункціональні `enum`-типи, для яких потрібно було узгодити мовну семантику з моделлю пам'яті та типів у Go.

Другим принципом була перевірка різних рівнів складності семантичного аналізу. Одні розширення можна реалізувати майже локально, тоді як інші потребують часткової типізації, завантаження сигнатур, уніфікації типів і врахування областей видимості. Саме тому до набору було включено короткі анонімні функції, іменовані аргументи та значення параметрів за замовчуванням: вони дозволяють перевірити, наскільки далеко може зайти транспілятор Go-подібної мови, не перетворюючись на окремий повноцінний компілятор із власною складною типовою системою.

Третім принципом була інтероперабельність. До GoNext відбиралися насамперед ті розширення, які зберігають сумісність із наявними Go-пакетами. З цієї причини особливу роль відіграють іменовані аргументи,

методи розширення та універсальний синтаксис виклику функцій, оскільки вони перевіряють, чи можна зробити API виразнішими, не руйнуючи двосторонню сумісність зі звичайним Go-кодом.

Четвертим принципом була навмисна перевірка меж самої платформи. До набору було включено конструкції, що виявляють фундаментальні обмеження Go: `enum`-типи показують вплив збирача сміття і узагальнень на представлення даних, короткі анонімні функції виявляють межі контекстного виведення типів, а універсальний синтаксис виклику функцій і методи розширення дозволяють дослідити неоднозначності резолюції функцій. Завдяки цьому GoNext виступає не просто набором зручних нововведень, а збалансованим експериментальним майданчиком, на якому перевіряються як сильні сторони мови Go, так і її межі як цілі транспіляції.

### 3.4. Повнофункціональні `enum`-типи як приклад репрезентаційної трансформації

Одним із найважливіших і найпоказовіших мовних розширень GoNext є повнофункціональні `enum`-типи [31]. У багатьох сучасних мовах, зокрема Rust, Haskell, Scala, Swift і F#, наявність типів-сум або алгебраїчних типів даних радикально впливає на спосіб моделювання даних і обробки станів. У стандартному Go подібна функціональність відсутня; найближчим аналогом є набір констант певного базового типу. Проте такий підхід не забезпечує ані вичерпності, ані власних полів у варіантів, ані інваріантів типу «змінна може містити лише один із заздалегідь визначених варіантів».

У GoNext запропоновано синтаксис, де `enum` є новим видом типового літерала. Наприклад, тип `Result[T, E]` може бути описаний як такий, що має варіанти `Ok(T)` і `Err(E)`, а тип `Option[T]` — варіанти `Some(T)` і `None`. Це принципово змінює підхід до моделювання помилок, відсутніх значень, станів обчислення й інших альтернативних сценаріїв.

```
type Result[T any, E error] enum {
    Ok(T)
    Err(E)
```

```

}

type Option[T any] enum {
    Some(T)
    None
}

type SomeEnum enum {
    Variant1(int)
    Variant2(string)
}

var resultValue Result[string, error] = Ok("success")
var optionalValue Option[string] = Some("value")
optionalValue = None()

```

Однак справжня дослідницька цінність enum-типів полягає не в синтаксисі, а в питанні їх представлення у рантаймі поверх Go. Саме тут виявляється, наскільки Go є придатною або непридатною платформою для реалізації складніших конструкцій.

### 3.4.1. Вимоги до представлення enum-типів

Щоб enum-тип був корисним, його реалізація має задовольняти кілька вимог:

- значення має однозначно вказувати, який варіант зараз зберігається;
- поля поточного варіанта мають бути доступні без втрати типової безпеки;
- має бути можливість створювати значення через конструктори варіантів;
- реалізація повинна по можливості підтримувати узагальнення;
- представлення не повинно конфліктувати з роботою збирача сміття Go;
- бажано зберігати прийнятний баланс між швидкістю, пам'яттю та інтероперабельністю.

### 3.4.2. Три схеми представлення enum-типів

У межах GoNext було розглянуто три основні схеми.

Перша схема — компактне представлення з тегом, який дозволяє визначити, який варіант зараз є активним, і блоком пам'яті, достатнім для того, щоб розмістити найбільший за розміром варіант. Саме так зазвичай реалізуються типи-суми у мовах із ручним управлінням пам'яттю [79] і це є найефективнішим варіантом, як з точки зору використання пам'яті, так і з точки зору швидкодії.

```
type SomeEnum struct {  
    tag uint8  
    data [N]byte  
}
```

Проте Go накладає на цю схему два суттєві обмеження.

Перше пов'язане зі збирачем сміття: якщо хоча б у одному варіанті є всередині хоча б один покажчик (тобто, наприклад він містить в собі рядок або інтерфейс) - при записі значення цього типу у масив байт - Go-рантайм більше не знатиме про те, що цей покажчик існує і може просто видалити об'єкт, на який покажчик посилається.

Друге пов'язане з узагальненнями: якщо хоча б один з варіантів enum має поле типу узагальненого параметру - неможливо на етапі компіляції знати розмір масиву `data`, а також, що цей узагальнений тип також може мати покажчики всередині.

Отже, ця схема виявляється небезпечною у загальному випадку та розглядається лише з теоретичної точки зору та як орієнтир з точки зору швидкодії.

Друга схема — структурне представлення, де enum містить окремий тег і окремі поля для кожного з можливих варіантів. Переваги такого підходу полягають у простоті, повній безпечності щодо збирача сміття та сумісності з узагальненнями. Недолік очевидний: пам'ять витрачається на всі варіанти одразу, а не лише на активний.

```

type SomeEnum struct {
    tag uint8
    variant1 someEnum_Variant1
    variant2 someEnum_Variant2
}

```

Третя схема — представлення через `interface{}`, що фактично зберігає тип змінної, що зараз записана у поле `inner` і посилання на інстанс цього типу. Перевага цього підходу в тому, що він добре узгоджується з моделлю Go, є безпечним і природно підтримує різні типи варіантів. Недолік полягає в додатковій непрямій адресації, потенційно більшій фрагментації пам'яті та певних рантаймових накладних витратах під час отримання значення активного варіанту.

```

type SomeEnum struct {
    inner interface{}
}

```

Науково важливим є те, що вибір між цими схемами не можна зробити лише на підставі «краса синтаксису». Він визначається саме властивостями хост-рантайму. Таким чином, `enum`-експеримент у `GoNext` прямо демонструє основну тезу дисертації: рантайм Go придатний для реалізації нових мовних конструкцій, але форма реалізації повинна бути узгоджена з його моделлю пам'яті та типів.

### 3.4.3. Інтерфейс доступу до варіантів

Окрім внутрішнього представлення, потрібно було вирішити, як гостьовий код буде працювати з `enum`-значеннями.

Було запропоновано два можливих варіанти API.

Перший надає доступ до варіантів через відповідні методи для кожного варіанту і є більш ідеоматичним з точки зору Go, але стає громіздким для `enum`-типів із багатьма варіантами.

```

func (self *SomeEnum) IsVariant1() bool
func (self *SomeEnum) Variant1() (bool, Variant1)
func (self *SomeEnum) IsVariant2() bool
func (self *SomeEnum) Variant2() (bool, Variant2)

```

Другий пропонує єдиний метод `Match`, який приймає функції для кожного з можливих варіантів `enum`-у і викликає тільки функцію, що відповідає поточному варіанту у даному інстансі `enum`-у.

```
func (self *SomeEnum) Match(variant1Func func(Variant1), variant2Func  
func(Variant2))
```

### 3.4.4. Значення `enum`-експерименту для оцінки `Go`

Реалізація `enum`-типів висвітлила кілька принципових речей.

1. `Go`-рантайм достатньо потужний, щоб підтримувати типи-суми поверх своєї моделі типів і пам'яті.
2. максимальна ефективність класичної реалізації `tagged unions` не завжди досяжна через властивості збирача сміття та узагальнень.
3. у `Go` важливішими за «ідеальне» компонування у рантаймі виявляються безпечність, сумісність і простота інтеграції з існуючим кодом.
4. навіть коли певне мовне розширення не має прямого аналога в хост-мові, воно все ж може бути реалізовано через поєднання синтетичних типів і методів без необхідності модифікувати сам компілятор `Go`.

### 3.5. Конструкція `match` як приклад структурної трансформації

Після введення `enum`-типів природно виникла потреба у зручному механізмі їх розбору. Саме тому наступним кроком стало впровадження `match`-синтаксису [30, 72]. Це добре ілюструє важливу властивість побудови мов: одні мовні конструкції майже завжди тягнуть за собою інші.

У `GoNext` `match` синтаксично наближений до `Go` `switch`, але семантично орієнтований на роботу з `enum`-варіантами. Такий вибір не випадковий. З одного боку, він робить нову конструкцію інтуїтивно знайомою `Go`-розробнику. З іншого боку, він дозволяє реалізувати розбір, подібний до зіставлення із взірцем, без введення занадто чужорідної синтаксичної форми.

Ключові можливості `match` у `GoNext` включають:

- вказування лише потрібних варіантів;
- обробку кількох варіантів однією гілкою;
- додаткові умови на значення всередині варіанта;
- перевірку вичерпності на етапі компіляції за відсутності `default`.

```

type SomeEnum enum {
    FirstVariant(int)
    SecondVariant(string)
    ThirdVariant
    FourthVariant(string)
}

var v SomeEnum
match v {
    case FirstVariant(_):
        // first
    case SecondVariant(value) if value != "":
        // non-empty second
    case SecondVariant(value) | FourthVariant(value):
        // second or fourth
    default:
        // all other variants, in this case third
}

```

З погляду трансформації ця конструкція не потребує окремого коду на етапі виконання програми. Вона трансформується у виклик раніше згенерованого для `enum` методу `Match`. Це є типовою структурною трансформацією: користувач бачить зручний синтаксис, але під ним лежить набір звичайних Go-викликів.

```

var v SomeEnum
v.Match(func(int) {
    // first
}, func(value string) {
    if value != "" {
        // non-empty second
    }
    return
}

```

```

    // second or fourth
}, func() {
    // all other variants, in this case third
}, func(string) {
    // second or fourth
})

```

Дослідницька цінність цього прикладу полягає в тому, що він показує можливості комбінування мовних розширень. Один раз спроектувавши стійкий базовий API для enum-значень, можна поверх нього розробляти більш виразні синтаксичні засоби для роботи з ними.

### 3.6. Значення параметрів за замовчуванням

Багато популярних мов підтримують значення параметрів за замовчуванням. У Go така мовна конструкція відсутня, і розробники зазвичай змушені використовувати шаблон «будівельник», передавати у функції додаткові структури даних із необов'язковими параметрами або оголошувати кілька функцій із різним набором параметрів та різними назвами замість однієї функції з параметрами за замовчуванням. Іноді це буває дуже незручно, особливо в API з великою кількістю необов'язкових параметрів.

У GoNext підтримка значень параметрів за замовчуванням була реалізована [26] так, щоб не змінювати модель виклику в самому Go. Головна складність полягала в тому, що значення за замовчуванням можуть залежати від попередніх аргументів функції, локального та глобального оточення, імпортів, констант і функцій. Тому варіант з простим підставленням значень за замовчуванням у місці виклику в загальному випадку не працював [65].

```

import "utils"
func compute(
    a int,
    b int = a + 1,
    c int = utils.Value()
) {}

```

Було запропоновано схему трансформації, що полягає у генерації двох функцій:

- основної функції без значень за замовчуванням, яка приймає всі параметри явно;
- допоміжної функції з умовною назвою `{FUNCTION_NAME}__default`, яка приймає параметри в обгортці на зразок `Optional` і відновлює пропущені значення всередині того ж оточення, де вони були визначені.

```
func compute(  
    a int,  
    b int,  
    c int,  
) {}  
  
func compute__default(  
    a int,  
    bOpt gn.Optional[int],  
    cOpt gn.Optional[int],  
) {  
    var b int  
    if bOpt.Exists {  
        b = bOpt.Value  
    } else {  
        b = a + 1  
    }  
  
    var c int  
    if cOpt.Exists {  
        c = cOpt.Value  
    } else {  
        c = utils.Value()  
    }  
  
    compute(a, b, c)  
}
```

Такий підхід має кілька переваг.

По-перше, він коректний щодо областей видимості. Значення за замовчуванням обчислюються там, де мають бути обчислені, а не в місці виклику.

По-друге, за наявності підтримки іменованих аргументів цей механізм дозволяє зручно працювати з функціями, що мають кілька необов'язкових параметрів: зокрема, можна явно вказати лише останній із них, не передаючи вручну всі попередні.

По-третє, рантаймові накладні витрати у більшості випадків мінімальні, а додатковий виклик часто може бути оптимізований хост-компілятором.

### 3.7. Іменовані аргументи

Іменовані аргументи є ще одним прикладом функціональності, широко поширеної в сучасних мовах, але відсутньої в Go. Особливо корисні вони в комбінації зі значеннями за замовчуванням, коли розробнику потрібно перевизначити лише кілька параметрів серед багатьох.

У проектуванні цього розширення в GoNext [27] було обрано підхід, близький до Python [34]: іменовані аргументи є суто call-site-конструкцією і не вимагають змін у сигнатурі самої функції. Такий вибір було зроблено з міркувань інтероперабельності. І завдяки цьому рішення іменовані аргументи можна застосовувати не лише для функцій GoNext, а й для імпортованих функцій звичайних Go-пакетів.

На рівні синтаксису іменовані аргумент має форму `name = value` у списку аргументів. Оскільки Go чітко розділяє вирази й оператори, таке розширення не конфліктує з наявними правилами граматики: у вихідному коді Go подібний запис у списку аргументів не є коректним.

```
func compute(a int, b int) {}
```

```
compute(10, b = 20)
```

```
compute(b = 20, a = 10)
```

Загалом алгоритм трансформації викликів функцій з іменованими аргументами (та значеннями за замовчуванням) можна подати у такому вигляді:

1. Визначити сигнатуру цільової функції.
2. Виконати відображення явно заданих позиційних аргументів на відповідні параметри.
3. Для параметрів, що залишилися незаповненими, здійснити зіставлення іменованих аргументів із параметрами за їхніми назвами.
4. Виконати перевірку коректності виклику, зокрема виявити повторне задання одного параметра позиційним та іменованим способом, наявність невідомих імен, а також порушення правил комбінування позиційних та іменованих аргументів.
5. У разі повного задання всіх параметрів згенерувати звичайний виклик функції.
6. Якщо частина незаповнених параметрів має значення за замовчуванням, використати допоміжну функцію `{FUNCTION_NAME}__default` замість цільової.

Реалізація цього мовного розширення, разом зі значеннями параметрів за замовчуванням, наочно демонструє необхідність семантичного аналізу та перевірки типів, зокрема сигнатур функцій, доступних у поточній області видимості. Причина полягає в тому, що транспілятор повинен спочатку виконати визначення цільової функції для кожного виклику, тобто встановити, яка саме функція буде викликана в кожній точці програми, щоб коректно з'ясувати назви її параметрів і те, які з них можуть бути пропущені. Цю інформацію неможливо отримати лише з абстрактного синтаксичного дерева.

Наприклад, у наведеному нижче випадку назва `hello` у функції `main` посилається на дві різні функції з різними сигнатурами:

```
func hello(a int, b int, c int) {} // 1
```

```
func main() {
    hello(1, 2, 3) // call 1
    hello := func(a int, b int) {} // 2
    hello(1, 2) // call 2
}
```

Значення параметрів за замовчуванням та іменовані аргументи в GoNext слід розглядати як єдиний клас мовних розширень. Їхня спільна реалізація спирається на один і той самий семантичний фундамент: потрібно не лише перебудувати список аргументів, а й коректно встановити цільову функцію, її сигнатуру, відповідність імен параметрам та спосіб заповнення пропущених значень. Саме цей приклад особливо чітко показує, що серйозні мовні розширення неможливо реалізувати як суто синтаксичні AST-перетворення; для них необхідний повноцінний або принаймні частковий семантичний аналіз програми.

### 3.8. Методи розширення та універсальний синтаксис виклику функцій

Одним із практично відчутних обмежень ергономіки Go є те, що на вбудованих колекційних типах `[]T` і `map[K]V` не можна безпосередньо визначати методи. Через це операції, які логічно сприймаються як дії над колекцією, наприклад `Filter`, `Map`, `Keys` або `GroupBy`, доводиться оформлювати як вільні функції, а не як ланцюжки викликів методів. Аналогічне обмеження стосується і інших структур даних, зокрема типів із зовнішніх пакетів, до яких також не можна додавати методи.

У GoNext було досліджено два суміжні підходи до розв'язання цієї проблеми [24, 92].

Перший — універсальний синтаксис виклику функцій [69], коли в разі відсутності методу на типі вираз `value.Method(args)` може інтерпретуватися як виклик вільної функції `Method(value, args)`.

Другий — методи розширення [20], коли лише спеціально позначені функції беруть участь у такому механізмі.

З погляду трансформації обидва підходи однакові. Різниця полягає в правилах резолюції вільних функцій.

Універсальний синтаксис виклику функцій є більш глобальним і менш контрольованим: фактично будь-яка функція з відповідною сигнатурою є кандидатом для такої трансформації. Це підвищує виразність, але створює ризик неоднозначностей і менш очевидної поведінки.

Методи розширення, навпаки, більш локальні й явні. З інженерного погляду цей підхід більше відповідає стилю Go, оскільки лише функції, спеціально позначені як `extension`, розглядаються як кандидати для такого переписування викликів.

У GoNext функція розширення оголошується як звичайна функція з ключовим словом `extension`, наприклад

```
extension func Filter[T any](collection []T, predicate func(T, int) bool) []T
```

Під час транспіляції в місці визначення такої функції слово `extension` просто прибирається, тому в згенерованому Go-кодi вона стає звичайною функцією `func Filter[T any](...) []T`. Додаткова трансформація відбувається в тих файлах, де пакет із цією функцією імпортовано і де вона викликається у формі методу: вираз `collection.Filter(...)` переписується в явний виклик `package.Filter(collection, ...)`.

Таке мовне розширення не вносить накладних витрат під час виконання програми. Також оскільки після транспіляції функції розширення стають звичайними Go-функціями, їх можна безпосередньо викликати і зі звичайного Go-коду.

### **3.9. Узагальнені методи як похідна обраної реалізації методів розширення**

Окремого розгляду заслуговує підтримка узагальнених методів. Стандартний Go навмисно не допускає узагальнених методів із новими тип-параметрами, які не входять до параметрів типу приймача [33, 78]. Причина полягає в конкретній проблемі реалізації викликів методів через інтерфейси.

Тривалий час автори Go виходили з того, що методи насамперед існують для реалізації інтерфейсів, а отже дозвіл на узагальнені методи для конкретних типів логічно потягнув би і узагальнені методи в інтерфейсах. Для такого механізму не було зрозумілого ефективного способу реалізації: оскільки в Go типи не декларують наперед, які інтерфейси вони реалізують, на етапі компіляції неможливо заздалегідь визначити, які саме інстанціювання такого методу, з потенційно нескінченної їх кількості, знадобляться під час виконання програми.

Водночас позиція авторів Go нині змінилася. Наразі пропонується дозволити узагальнені методи саме для конкретних типів, не поширюючи цей механізм на інтерфейси. Отже, автори Go планують додати цю функціональність у саму мову, але в обмеженій формі [37].

Проте методи розширення у GoNext [24] обходять цю проблему автоматично. Оскільки вони врешті-решт є звичайними вільними функціями, вони можуть мати довільний список типових параметрів і не залежать від механізму задоволення інтерфейсів. Завдяки цьому стає можливим, наприклад, виразити `Map` як «метод» над колекцією, де результат має інший елементний тип, ніж приймач.

```
extension func Map[T any, R any](array Array[T], fn func(item T, index int) R)
Array[R] {
    // ...
}
var a Array[int]
var b Array[string] = a.Map(func(item int, _ int) { return
strconv.Itoa(item) })
```

Це один із найцікавіших результатів GoNext. Він показує, що транспіляція може не лише імітувати відсутні мовні конструкції, а й відкривати нові комбінації конструкцій хост-мови, які в самій мові недосяжні з міркувань дизайну. Інакше кажучи, гостьова мова може використовувати Go не просто як обмеження, а як набір будівельних блоків, які комбінуються в новий спосіб.

### 3.10. Короткий синтаксис для анонімних функцій

Робота з функціями вищого порядку в Go часто є багатослівною. Повні літерали `func(...) ... { ... }` добре узгоджуються з простотою мови, але в місцях частих коротких викликів ускладнюють читання, особливо в ланцюжках викликів методів над колекціями, тестових сценаріях, роботі з обробниками подій і декларативних DSL.

Тому в GoNext було досліджено короткий синтаксис для анонімних функцій [91]. Його сутність полягає в тому, що типи параметрів не вказуються явно, а відновлюються з контексту очікуваного функціонального типу.

```
(argument1, argument2) => { body }
```

Із погляду транспіляції ця конструкція є нетривіальною. На відміну від більш простих синтаксичних розширень, короткий літерал не можна одразу перетворити на валідний Go-код, оскільки в ньому відсутні типи параметрів і, за потреби, тип результату:

```
func(argument1 type1, argument2 type2) resultType { body }
```

Отже, транспілятор мусить:

- побудувати тимчасове представлення короткого літерала з незаповненими типами;
- визначити очікуваний тип функції з зовнішнього контексту;
- уніфікувати параметри та результат із цим типом використовуючи правила уніфікації Go [74];
- лише після цього згенерувати повний Go-літерал функції.

Тобто, транспілятор не може обмежитися лише локальною AST-трансформацією: він має виконати повний семантичний аналіз оточуючого коду, майже той самий, що робить компілятор Go, щоб врахувати різні варіанти опису типу функції. Наприклад, у даному випадку при виклику функції `filter` короткий літерал потрібно уніфікувати з типом `FilterFunc`, який у загальному випадку може бути визначений навіть в іншому пакеті:

```
type FilterFunc func(int) bool
func filter(xs []int, f FilterFunc) {}
filter(array, (x) => { return x % 2 == 0 })
```

Особливо показовим є випадок узагальнених функцій вищого порядку. Якщо контекст задає тип на кшталт `func(A) B`, то тип параметра короткого літерала `A` ще можна відновити після інстанціювання, однак тип результату `B` у загальному випадку неможливо вивести з оточуючого контексту, лише з тіла самої анонімною функції:

```
func map[A any, B any](values []A, f func(A) B) []B {
    // ...
}
func main() {
    numbers := []int{1, 2, 3}
    squared := map(numbers, (n) => { return n * n })
}
```

Це істотно ускладнює транспіляцію, бо навіть сам Go не підтримує такого виведення типу результату для літералів функцій. Тому практичним висновком дослідження цього мовного розширення стало обмеження: короткий синтаксис доцільно підтримувати лише для непараметризованих контекстів, а в місцях, де очікуються узагальнені функції вищого порядку, його краще заборонити.

Повертаючись до другого принципу проектування нових мов на базі Go: не кожен синтаксичну ідею варто реалізовувати для всіх ситуацій, якщо ціна виходить за межі прагматичності.

### 3.11. Взаємодія реалізованих мовних розширень

Суттєвою частиною дослідження є не лише окрема реалізація кожного розширення, а й аналіз їх взаємодії. На практиці мовні конструкції не живуть ізольовано і саме їх комбінації визначають, чи є мова цілісною.

#### 3.11.1. Комбінація `enum` і `match`

Поєднання `enum`-типів і `match` є однією з найбільш природних і семантично цілісних комбінацій у GoNext. `enum`-типи дають змогу явно

моделювати альтернативні стани, варіанти результату, помилки або відсутність значення, тоді як `match` надає безпосередній і виразний механізм розбору таких значень. Наприклад, функція може повертати `Result` або `Option`, а код, що працює з цим результатом, може через `match` явно обробляти всі варіанти без переходу до менш структурованих умовних перевірок. Саме ця пара добре демонструє, як поява нової моделі даних природно потребує відповідного засобу її використання, а отже окремі мовні розширення можуть формувати цілісні та змістовно пов'язані комбінації.

### **3.11.2. Комбінація іменованих аргументів і значень параметрів за замовчуванням**

Іменовані аргументи і значення параметрів за замовчуванням доповнюють одне одного безпосередньо, оскільки розв'язують дві сторони тієї самої задачі: як зробити виклик функції з багатьма параметрами коротким, але водночас зрозумілим. Значення за замовчуванням дають змогу не передавати необов'язкові параметри взагалі, тоді як іменовані аргументи дозволяють явно вказати лише ті значення, які в конкретному виклику відхиляються від стандартних. У поєднанні це дає змогу будувати компактні виклики без втрати читабельності: програміст бачить не повний позиційний список аргументів, а лише семантично важливі відхилення від стандартної поведінки. Саме тому така комбінація є особливо доречною для API, конфігураційних сценаріїв і DSL-подібного коду, де значна частина параметрів має природні стандартні значення, і лише в окремих випадках доводиться їх замінити на інші.

### **3.11.3. Комбінація методів розширення або універсального синтаксису виклику функцій і коротких анонімних функцій**

Саме ця комбінація наближає GoNext до ланцюжкового стилю програмування над колекціями та потоками даних. Колекція може «мати» методи `Filter`, `Map`, `Reduce`, хоча в стандартному Go це були б вільні функції. Короткі анонімні функції зменшують багатослівність, а універсальний

синтаксис виклику функцій або синтаксис методів розширення роблять композицію візуально цілісною. У результаті з'являється стиль коду, який важко або незручно виразити в чистому Go, але який при цьому транспілюється до цілком звичайного Go-коду.

### **3.12. Інтероперабельність GoNext зі звичайним Go**

Оскільки інтероперабельність є одним із основних критеріїв роботи, експериментальна реалізація GoNext орієнтувалася на максимально можливу двосторонню сумісність.

#### **3.12.1. Використання Go-бібліотек із GoNext**

Для більшості реалізованих мовних розширень сумісність із наявними Go-пакетами досягається природно. Іменовані аргументи працюють із функціями, сигнатури яких відомі транспілятору. Методи розширення можуть бути використані поверх типів із зовнішніх пакетів або через спеціальні wrapper-пакети, або через спеціальний режим імпорту `import extension`, який трактує функції пакета як потенційні функції розширення.

`enum`-типи тут є винятком: стандартні Go-бібліотеки, звісно, не знають про них. Але це не проблема інтероперабельності як такої, це лише означає, що нові типи існують на рівні GoNext і мають бути представлені у Go через звичайні структури та методи.

#### **3.12.2. Використання згенерованого GoNext-коду зі звичайного Go**

Після трансформації більшість конструкцій GoNext зникає, залишаючи звичайні Go-функції, типи й методи. Це особливо добре видно на прикладі методів розширення: для Go-коду вони є просто функціями пакета. Значення параметрів за замовчуванням виражаються через допоміжні обгортки, але базова функція лишається звичайною. Іменовані аргументи взагалі повністю зникають.

Таким чином, Go-код може використовувати значну частину продуктів GoNext без знання про їхнє походження. Це критично важливий результат.

Він означає, що нова мова може впроваджуватися в існуючі проекти поступово.

### 3.13. Якість згенерованого коду та налагоджуваність

Для транспільованої мови важливе не лише те, що код компілюється, а і те, наскільки він придатний до аналізу людиною [61]. У випадку GoNext це особливо важливо, оскільки однією з головних обіцянок підходу є співіснування зі звичайною Go-екосистемою.

Згенерований код має бути:

- структурно передбачуваним;
- відформатованим стандартним `gofmt`-сумісним способом;
- мінімально засміченим технічними деталями;
- таким, що дозволяє читати стек викликів, повідомлення про помилки й результати тестів без повної втрати зв'язку з вихідною програмою.

У цьому сенсі використання стандартних пакетів `go/format` і `go/printer` є не просто технічною зручністю, а ключовою умовою прийнятної якості інструмента. Чим більше згенерований код схожий на те, що міг би написати Go-розробник вручну, тим вищою є довіра до транспілятора.

Окремим питанням є відображення позицій токенів між GoNext і згенерованим Go. Хоча в межах експериментального дослідження ця проблема не є центральною, вона визначає один із напрямів майбутнього розвитку. Для повноцінного IDE-досвіду, точного відображення діагностик і відлагодження на рівні вихідної мови бажано мати більш формалізовану систему відповідності позицій [67, 77]. Проте вже сам факт, що кінцевий код є звичайним Go, істотно полегшує роботу з наявними інструментами в порівнянні з мовами, що мають власну віртуальну машину або нестандартний формат артефактів.

### 3.14. Узагальнення результатів GoNext на ширший клас мов програмування

Хоча GoNext у межах цієї роботи побудовано як діалект Go, його значення не зводиться до демонстрації окремої надмножини однієї мови. Насправді GoNext виконує роль експериментального стенда, на якому перевіряються три важливі гіпотези: по-перше, що значна частина мовної виразності може бути додана поверх Go без модифікації офіційного компілятора; по-друге, що для цього достатньо поєднання розширеного синтаксичного аналізу, часткової типізації та контрольованих AST-трансформацій; по-третє, що практична цінність такого підходу визначається не «екзотичністю» нової конструкції, а тим, наскільки добре вона вкладається у модель виконання, типізацію, збирач сміття та екосистему Go.

Узагальнення результатів GoNext на інші мови програмування слід робити не на рівні буквального перенесення його синтаксису, а на рівні класів трансформацій. Реалізовані в GoNext значення параметрів за замовчуванням, іменовані аргументи, методи розширення, універсальний синтаксис виклику функцій і короткі анонімні функції показують, що великий клас ергономічних мовних розширень може бути реалізований як синтаксичний або структурний шар над уже наявною платформою. Це означає, що аналогічні розширення можуть бути додані не лише до Go-подібного діалекту, а і до ширшого кола прикладних мов.

Експеримент із `enum`-типами і `match` узагальнюється на інший клас задач: побудову мов, яким потрібні не лише синтаксичні покращення, а й нові моделі даних та нові способи їх опрацювання. Отриманий результат полягає в тому, що Go може бути цілком не лише для «косметичних» мовних надбудов, а й для мов, які вводять алгебраїчні типи даних, варіантні представлення, структуроване розгалуження за формою значення або спеціалізовані моделі опису станів і результатів обчислення. Проте це узагальнення має чітку межу: що далі нова модель даних відходить від

базових уявлень Go про значення, покажчики та роботу збирача сміття, то дорожчою й менш прозорою стає її реалізація.

Окремо важливим є й архітектурне узагальнення. Досвід створення GoNext показав, що практично корисний транспілятор до Go не повинен бути повноцінним автономним компілятором, який самостійно реалізує весь спектр обробки програми від лексичного аналізу до низькорівневої генерації коду, але й не може зводитися до суто текстового препроцесора. Для широкого класу нових мов оптимальним є проміжний варіант: власний фронтенд гостьової мови поєднується з повторним використанням інструментів Go на етапах побудови, аналізу, нормалізації та виведення цільового коду. Для діалектів Go і близьких до неї надмножин особливу цінність мають пакети `go/token`, `go/scanner`, `go/parser`, `go/ast`, `go/types`, `go/printer`, `go/format`, а також допоміжні засоби на кшталт `golang.org/x/tools/go/ast/astutil`, оскільки вони дозволяють працювати майже безпосередньо із синтаксисом, типами, трансформацією та форматуванням Go-коду. Для інших мов програмування, синтаксис яких від початку істотно відрізняється від Go, більш важливими стають інструменти пізніших фаз, насамперед `go/ast`, `go/types`, `go/printer`, `go/format`, `golang.org/x/tools/go/packages` та, за потреби, `golang.org/x/tools/go/ssa`, які використовуються вже не для розбору вихідної мови, а для побудови, перевірки, аналізу й генерації її цільового подання у Go. Це спостереження переноситься на інші сценарії: на нові статично типізовані мови, свідомо спроектовані «під Go», на предметно-орієнтовані мови для інфраструктурних або інтеграційних задач, а також на часткову транспіляцію вже наявних мов, якщо їхню семантику можна достатньо точно відобразити через звичайний Go-код.

Водночас експерименти з GoNext показали і межі такого узагальнення. Якщо мовна конструкція вимагає повноцінної окремої моделі виконання, агресивної динамічності, спеціалізованого керування пам'яттю, нетривіального JIT-підходу або незвичного виведення типів, яке не

узгоджується з правилами Go, то просте транспілювання у Go втрачає значну частину своєї привабливості. Показовим є приклад коротких анонімних функцій у параметризованих контекстах: навіть така на перший погляд локальна конструкція швидко наштовхується на межі, задані типізацією хост-мови. Отже, GoNext підтверджує не універсальність Go як цільової платформи, а її високу придатність для певного, досить широкого, але концептуально обмеженого класу мов програмування.

Таким чином, результати, отримані на прикладі GoNext, можна інтерпретувати як експериментальне підтвердження загальнішої тези дисертації: Go є вдалою платформою для тих нових мов програмування, які прагнуть поєднати власну мовну новизну з інтероперабельністю з екосистемою Go, передбачуваним і швидким виконанням коду, невисокою вартістю реалізації та доступом до зрілої інструментальної інфраструктури.

### 3.15. Висновки до розділу

У третьому розділі розроблено й проаналізовано експериментальну реалізацію запропонованого підходу у вигляді діалекту GoNext. Показано, що GoNext може слугувати не лише демонстраційним прототипом окремих мовних ідей, а повноцінним експериментальним стендом для перевірки того, які класи мовних розширень реально та доцільно реалізовувати поверх рантайму Go без модифікації офіційного компілятора.

У межах GoNext реалізовано кілька типів мовних розширень, що відповідають раніше запропонованій класифікації трансформацій. На прикладі `enum`-типів показано можливість репрезентаційної трансформації, коли нова мовна конструкція потребує власної схеми подання у Go. На прикладі `match` продемонстровано структурну трансформацію, яка перетворює більш виразний спосіб керування потоком виконання на еквівалентні базові конструкції Go. Значення параметрів за замовчуванням, іменовані аргументи, методи розширення, універсальний синтаксис виклику функцій і узагальнені методи, а також короткі анонімні функції підтвердили,

що значна частина ергономічних розширень може бути реалізована як синтаксичний або структурний шар із відносно невисокою складністю.

Експериментальна реалізація також показала, що для практично корисного транспілятора недостатньо лише локальних синтаксичних переписувань. Частина конструкцій, зокрема іменовані аргументи, значення параметрів за замовчуванням, методи розширення та короткі анонімні функції, потребує часткового семантичного аналізу, знання сигнатур функцій, областей видимості та типів оточуючого контексту. Це підтверджує коректність обраної архітектури, у якій власний фронтенд гостьової мови поєднується з використанням інструментів Go для аналізу, перевірки та генерації коду.

Показано, що окремі мовні розширення не існують ізольовано, а формують змістовні комбінації. Поєднання `enum` і `match` утворює цілісну модель роботи з варіантними значеннями. Комбінація іменованих аргументів зі значеннями параметрів за замовчуванням підвищує виразність API. Поєднання методів розширення або універсального синтаксису виклику функцій із короткими анонімними функціями робить можливим більш компактний стиль роботи з колекціями та потоками даних.

Окремим результатом розділу є підтвердження високої інтероперабельності GoNext зі звичайним Go. Більшість розширень після трансформації зникає, залишаючи звичайні Go-функції, типи та виклики, а отже згенерований код може використовуватися у наявних Go-проектах без спеціальної інфраструктури. Водночас продемонстровано важливість якості згенерованого коду: передбачувана структура, форматування стандартними засобами та відносна читабельність є необхідною умовою довіри до транспілятора і практичної придатності підходу.

Разом із тим експерименти з GoNext дозволили чітко окреслити межі застосовності запропонованої моделі. Якщо мовне розширення занадто далеко відходить від системи типів, моделі виконання або принципів роботи збирача сміття у Go, то ціна реалізації швидко зростає, а прозорість

трансформації зменшується. Отже, результати розділу підтверджують головну тезу роботи: Go є не універсальною ціллю для будь-якої нової мови, але є дуже сильною платформою для широкого класу мовних розширень, діалектів і нових мов, семантика яких може бути виражена через контрольовану транспіляцію у звичайний Go-код.

## **РОЗДІЛ 4. ОЦІНЮВАННЯ ДОЦІЛЬНОСТІ ПІДХОДУ ТА УЗАГАЛЬНЕННЯ НА ІНШІ МОВИ Й ПРЕДМЕТНІ ОБЛАСТІ**

### **4.1. Логіка оцінювання результатів**

Оцінювання в цьому розділі не зводиться до одного числового показника або окремої технічної характеристики. Доцільність використання Go як платформи для нової мови визначається балансом кількох груп критеріїв: виконувальних властивостей, виразності, інтегрованості, доступності інструментів для транспіляції і вартості реалізації.

Основою такого оцінювання є поєднання теоретичної моделі, сформульованої в розділі 2, та експериментальних результатів, отриманих на прикладі GoNext у розділі 3. Тому далі аналізується не абстрактна «зручність Go взагалі», а практична поведінка платформи під час реалізації конкретних мовних розширень різних типів.

Логіка висновків є такою: якщо мовна конструкція може бути трансформована у звичайний Go-код без окремого рантайму, зберігає сумісність з екосистемою Go і не потребує непропорційно складного транспілятора, то використання Go є доцільним. Якщо ж реалізація вимагає глибоких семантичних компромісів, громіздких схем представлення або окремої моделі виконання, доцільність підходу зменшується.

### **4.2. Оцінювання за виконувальними властивостями**

#### **4.2.1. Продуктивність виконання**

Критерій продуктивності в цій роботі полягає не в тому, чи можна уявити платформу, що в окремих сценаріях була б швидшою за Go, а в тому, чи забезпечує Go достатню швидкодію для класу мов, який розглядається: діалектів, надмножин і предметно-орієнтованих мов для серверних, прикладних та інфраструктурних задач. Отримані результати дають підстави для позитивного висновку: базова продуктивність Go-компілятора і

згенерованого ним коду є для цього класу застосувань цілком достатньою [39].

Експерименти з GoNext показали, що для значної частини мовних розширень транспіляція не чинить негативного впливу на продуктивність виконання. Іменовані аргументи повністю усуваються під час трансформації, методи розширення та універсальний синтаксис виклику функцій зводяться до звичайних викликів, а параметри за замовчуванням у гіршому випадку додають лише допоміжний виклик із простою умовною логікою. Отже, для досліджених розширень отриманий код зберігає передбачувані для Go характеристики виконання, і сама трансформація не робить підхід практично сумнівним.

Більш цікавими є запропоновані схеми реалізації повнофункціональних enum-типів. Дослідження показало, що tagged-union-подання, природне для низькорівневих мов програмування, для Go у більшості випадків неприйнятне через конфлікти зі збирачем сміття. Практичними натомість є представлення через структуру з полями для всіх варіантів або через `interface{}`. Обидва підходи вимагають компромісу між компактністю та простотою доступу, але залишаються достатньо ефективними для практичного використання. Тому обмеження стосується форми представлення, а не принципової неможливості ефективної реалізації. Показово, що в С 15 запропоновано додати аналогічну функціональність і в якості представлення було обрано еквівалент варіанту з `interface{}` з пункту 3.4.2 [55, 85].

Отже, жодне з мовних розширень, реалізованих і проаналізованих у межах цієї роботи, не виявилось критично проблемним для продуктивності. У дослідженому класі задач Go є ефективною цільовою платформою для реалізації нових мовних конструкцій.

#### **4.2.2. Керування пам'яттю і збирач сміття**

Однією з найвагоміших переваг Go як хост-платформи є той факт, що нова мова не повинна самотійно реалізовувати збирач сміття. Для багатьох

мов це колосальна інженерна перевага. Побудова власного якісного збирача сміття — дуже складне і дороге завдання [42]. Використовуючи Go як платформу, розробник гостьової мови отримує сучасний конкурентний збирач сміття автоматично [1, 75].

Водночас збирач сміття не є нейтральною деталлю реалізації, а накладає обмеження на схеми представлення даних. Експеримент з `enum` у GoNext показав, що класичне `compact tagged-union`-подання у Go загалом неприйнятне через ризик приховати від збирача сміття покажчики всередині значення. Тому доцільними виявляються лише ті трансформації, що узгоджені з моделлю пам'яті Go, навіть якщо вони трохи менш ефективні.

### **4.2.3. Конкурентність і системні властивості**

Ще одна сильна сторона Go як цільової платформи полягає в тому, що транспільований код може безпосередньо спиратися на горутіни, канали та дуже велику стандартну бібліотеку. Разом зі статичною збіркою, швидким запуском і кроскомпіляцією це робить Go особливо доречною платформою для діалектів і DSL серверного, інтеграційного та оркестраційного профілю.

Водночас ця перевага зберігається лише тоді, коли модель виконання гостьової мови сумісна з моделлю Go. Якщо мова вимагає власного планувальника [36], іншої семантики паралелізму або повного контролю над рантаймом, доцільність такого вибору зменшується. Отже, за критерієм конкурентності й системних властивостей Go є дуже вдалою, хоча й не універсальною, ціллю.

## **4.3. Оцінювання за виразними властивостями хост-мови**

### **4.3.1. Класи конструкцій, які Go дозволяє виразити природно**

Як було показано в розділі 2, виразність хост-мови в контексті цієї роботи означає не «красу» самої Go, а те, наскільки її семантика придатна для втілення конструкцій іншої мови. Експерименти з GoNext підтверджують, що істотна частина корисних мовних розширень належить до перших двох із

раніше виділених класів: або має майже пряме відображення в Go, або коректно виражається через комбінацію вже наявних засобів мови.

До першого класу належать насамперед ергономічні розширення, що не змінюють фундаментальної моделі виконання. Іменовані аргументи після встановлення цільової сигнатури зводяться до перестановки аргументів у позиційний порядок. Методи розширення та універсальний синтаксис виклику функцій трансформуються у звичайні виклики вільних функцій. У цих випадках Go виявляється достатньо виразною саме тому, що її базові конструкції вже охоплюють потрібну семантику, а гостьова мова додає передусім зручніший спосіб запису.

Це є важливим висновком для загальної оцінки платформи. Go не потребує надмірно багатой власної синтаксичної системи, щоб бути придатною основою для нової мови. Навпаки, її відносна простота в поєднанні з передбачуваними та простими правилами типізації створює стабільний набір примітивів, поверх яких можна будувати додатковий шар виразності. Для транспілятора це означає, що значна частина нових мовних конструкцій може бути реалізована як усунення синтаксичного цукру без окремого рантайму і без втрати простоти згенерованого коду.

#### **4.3.2. Конструкції, що потребують складеного відображення на засоби Go**

Другий клас виразних можливостей є для цієї дисертації ще важливішим, оскільки саме він показує справжню межу практичної придатності Go. До нього належать конструкції, які не мають прямого аналога в мові, але можуть бути коректно виражені через поєднання типів, функцій, інтерфейсів та інших конструкцій мови Go.

Найпоказовішим прикладом є повнофункціональні `enum`-типи. У самій Go немає алгебраїчних типів даних, проте експеримент показав, що вони можуть бути представлені через структури даних та згенеровані методи доступу. Аналогічно конструкція `match` не має прямого відповідника, але може бути зведена до раніше згенерованого API розбору варіантів. Значення

параметрів за замовчуванням реалізуються не через вбудовану підтримку мови, а через синтетичні функції-обгортки, які обчислюють пропущені аргументи в правильному семантичному оточенні. Узагальнені методи також не існують у стандартному Go, але стають можливими як наслідок реалізації методів розширення через узагальнені вільні функції.

Саме ці приклади дозволяють сформулювати центральний висновок: Go як хост-мова є достатньо виразною не тому, що вона безпосередньо містить усі бажані високорівневі конструкції, а тому, що її базові механізми добре комбінуються між собою. Інакше кажучи, придатність Go визначається не максимальною багатістю вбудованих абстракцій, а здатністю слугувати стабільною цільовою мовою, в якій можна реконструювати потрібну семантику через обмежений набір надійних будівельних блоків.

Додатково важливо, що така реконструкція часто зберігає читабельність і передбачуваність коду. Згенеровані допоміжні структури та функції можуть бути громіздкішими за вихідну конструкцію, однак вони лишаються зрозумілими для компілятора Go, засобів аналізу і людини-розробника. Це означає, що виразність хост-мови тут має оцінюватися разом із її здатністю породжувати керовані трансформації, а не лише з погляду лаконічності кінцевого представлення.

### 4.3.3. Межі виразності Go як цілі транспіляції

Водночас експерименти з GoNext показали, що виразність Go має чіткі межі. Вони проявляються там, де нова конструкція спирається на властивості, яких у Go немає не випадково, а через фундаментальні рішення дизайну мови та рантайму. Показовим прикладом є короткі анонімні функції в параметризованих контекстах вищого порядку. Для них уже недостатньо простої перебудови синтаксису: виникає потреба у виведенні типів, яке сам Go у таких випадках не підтримує. Отже, частину бажаної семантики доводиться або обмежувати, або взагалі не реалізовувати.

Іншу межу демонструють `enum`-типи. Хоча вони можуть бути реалізовані, їх не вдається відобразити в класичну компактну `tagged-union`-форму без

конфлікту зі збирачем сміття та узагальненнями. Отже, виразність Go як хост-мови є достатньою, але не необмеженою: вона дозволяє моделювати нові конструкції лише доти, доки їх можна узгодити з уже наявною моделлю типів, пам'яті та викликів.

Звідси випливає загальний висновок для оцінювання. За критерієм виразних властивостей Go є сильною платформою для еволюційних мовних розширень, надмножин, прикладних DSL і нових Go-споріднених мов, де основна новизна зосереджена у фронтенді та способі композиції конструкцій. Однак вона істотно менш придатна для мов, що вимагають радикально іншої типізації, нетривіального виведення типів, прихованих семантичних ефектів або окремої моделі виконання. Тому виразність Go слід оцінювати як високу в межах прагматично важливого класу задач, але не як універсальну.

## **4.4. Оцінювання за інтероперабельністю**

### **4.4.1. Використання екосистеми Go**

Інтероперабельність із екосистемою не виникає автоматично: вона є наслідком архітектурних рішень у гостьовій мові та транспіляторі. Якщо мова зберігає модель пакетів, типів і викликів, сумісну з Go, її користувачі можуть безпосередньо спиратися на стандартну бібліотеку та велику екосистему Go. Саме це є однією з головних практичних переваг Go як цілі транспіляції.

Для прикладних мов і діалектів така властивість має вирішальне значення. Вона дає змогу не будувати заново мережеві, файлові, криптографічні, конкурентні та інфраструктурні засоби, а повторно використати вже перевірені Go-бібліотеки. Водночас ця перевага зберігається лише тоді, коли транспільований код залишається достатньо «go-подібним»: експортує зрозумілий API, використовує звичні пакети й не вимагає непрозорого проміжного рантайму для базових операцій. У результаті основна складність переноситься з реалізації базової платформи на дизайн власних мовних абстракцій.

Отже, за критерієм доступу до екосистеми Go отримує позитивну оцінку: вона надає не лише рантайм виконання, а й великий готовий шар прикладної функціональності. Обмеження з'являються лише тоді, коли семантика нової мови настільки віддаляється від Go, що пряме використання її пакетів перестає бути природним.

#### 4.4.2. Інтеграція в існуючі проєкти, написані на Go

Наступний рівень інтеперабельності полягає не лише у виклику Go-бібліотек із гостьової мови, а й у можливості вбудовувати результати її транспіляції в уже наявні Go-проєкти. Для практичного впровадження це особливо важливо, оскільки нова мова тоді перестає бути ізольованим експериментом і може використовуватися як частина існуючої кодової бази.

Якщо функції, типи й модулі гостьової мови після транспіляції є семантично схожими з функціями, типами й модулями Go, то окремі її бібліотеки можуть підключатися та використовуватися як звичайні Go-пакети. Ключовою умовою тут є відсутність спеціальних вимог на межі між двома мовами: згенерований код має збиратися, тестуватися і підключатися стандартними засобами Go без окремої моделі запуску.

У прикладних проєктах це дає такі переваги:

- не потрібно переписувати всю кодову базу;
- нову мову можна впроваджувати поетапно, починаючи з окремих модулів;
- бібліотеки, створені цією мовою, можуть експортувати звичайні для Go функції та типи;
- існуюча інфраструктура збірки, тестування і розгортання зберігається.

Отже, саме можливість поступового впровадження є однією з головних практичних переваг Go як хост-платформи. Якщо ж семантика гостьової мови настільки відрізняється від Go, то ця перевага істотно слабшає.

## **4.5. Оцінювання за наявністю інструментів для транспіляції та вартістю реалізації**

### **4.5.1. Переваги офіційної екосистеми інструментів**

Із погляду побудови компіляторів Go має одну з найважливіших, хоча часто недооцінених переваг: вона надає офіційні бібліотеки для роботи з власним кодом. Це означає, що автор гостьової мови не лише використовує Go як ціль виконання, а й може будувати сам транспілятор значною мірою на її ж інструментах.

Практичні наслідки цього такі:

- зменшується кількість коду, який треба реалізовувати з нуля;
- знижується ризик помилок у синтаксичному й типовому аналізі;
- спрощується підтримка змін у специфікації Go;
- легше інтегрувати транспілятор у звичний робочий процес Go-проектів.

Порівняно з реалізацією синтаксичного, типового аналізу та генерації коду повністю з нуля, це різко зменшує вартість входу. Саме тому Go особливо привабливий для дослідницьких і експериментальних мов.

### **4.5.2. Порогова вартість реалізації нової мови**

Досвід GoNext дозволяє сформулювати важливий практичний висновок: побудова діалекту чи нової невеликої мови поверх Go є на порядок дешевшою, ніж реалізація повноцінної автономної мови з власним рантаймом. Це не означає, що транспілятор до Go є тривіальним. Але головні «дорогі» елементи мовної інженерії — кодогенерація машинних інструкцій, рантайм, збирач сміття, платформи збірки, стандартні бібліотеки — фактично вже наявні.

Отже, для багатьох класів задач, де мовна новизна концентрується на фронтенді, а не на моделі виконання, це суттєво підвищує доцільність вибору Go як хост-платформи.

## 4.6. Порівняння Go з альтернативними платформами

### 4.6.1. Порівняння з JVM

JVM має очевидну перевагу у вигляді формально визначеного байткоду, багаторічної історії підтримки великої кількості мов і потужної JIT-оптимізації. Показовим прикладом цього є мова Scala, яка поєднує функціональну та об'єктно-орієнтовану парадигми поверх JVM [58] та додає до алгебраїчні типи даних, зіставлення із взірцем та розвинену систему типів, зберігаючи при цьому повну інтероперабельність із Java-бібліотеками. Якщо нова мова потребує глибокої інтеграції з об'єктною моделлю JVM, динамічним завантаженням класів, рефлексією та великою вже існуючою екосистемою мовних рішень, JVM часто буде більш придатною платформою.

Однак у порівнянні з JVM Go виграє в інших аспектах:

- простішому ланцюжку збірки й розгортання;
- статичній компіляції у виконувани файли;
- швидкому старті;
- нижчому інфраструктурному порозі для серверних та інфраструктурних систем;
- ближчому інженерному профілі до хмарного й системного програмування.

Отже, JVM є кращою як загальна багатомовна віртуальна машина, тоді як Go краще підходить як практичний фундамент для транспільованих статично-типізованих системних, серверних та Go-споріднених мов.

### 4.6.2. Порівняння з LLVM

LLVM надає дуже потужний низькорівневий бекенд і, як правило, є кращим вибором тоді, коли нова мова потребує прямого контролю над представленням даних у пам'яті, генерацією інструкцій, специфічних оптимізацій, нестандартних ABI або повністю власної моделі виконання. Для

мов системного програмування «з нуля» LLVM часто є більш логічним вибором.

Порівняно з LLVM, Go доцільніше розглядати не як основу для низькорівневих мов, а як практичну хост-платформу для транспільованих мов, орієнтованих на швидку реалізацію, інтегрованість і повторне використання зрілої екосистеми. Go поступається LLVM у контролі над низькорівневими деталями виконання, але має перевагу там, де важливіші швидкість розроблення й доступ до готової прикладної інфраструктури.

#### **4.6.3. Порівняння з JavaScript-екосистемою транспіляції**

JavaScript-екосистема є наймасовішим середовищем транспіляції передусім тому, що браузер історично використовує саме JavaScript як універсальну мову виконання. Тому популярність JavaScript як цілі пояснюється не лише його властивостями, а й зовнішньою вимогою веб-платформи.

Go не має такої обов'язкової ролі, проте привабливий з інших причин: статична компіляція, просте розгортання, швидкий старт, конкурентний рантайм і зріла серверна екосистема. Тому JavaScript є природною ціллю для веб-орієнтованих мовних надбудов, тоді як Go доцільніший як ціль для серверних, інфраструктурних і прикладних мов, де важливі автономний виконуваний артефакт та інтеграція з Go-бібліотеками.

### **4.7. Go як цільова платформа для широкого класу мов програмування**

Підсумовуючи результати, отримані на прикладі GoNext, можна стверджувати, що Go є придатною цільовою платформою не лише для одного експериментального діалекту, а й для широкого кола нових мов. Насамперед це стосується тих мов, діалектів і мовних розширень, чиї конструкції можна коректно перетворити на звичайний Go-код без створення окремого рантайму, без запровадження несумісної системи типів і без втрати доступу до бібліотек та інструментів Go.

Передусім це стосується трьох груп систем. По-перше, діалектів, надмножин і еволюційних розширень самого Go, для яких транспіляція є природним способом реалізації. По-друге, предметно-орієнтованих мов для серверних, інтеграційних та інфраструктурних задач, де важливими є готовий рантайм Go, статична компіляція і доступ до зрілої екосистеми пакетів. По-третє, окремих нових мов, які від початку проєктуються з урахуванням обмежень і сильних сторін Go, тобто таких мов, чиї основні конструкції можна відносно просто й економно звести до звичайного Go-коду.

Водночас цей висновок не означає, що Go однаково добре підходить для будь-якої нової мови. Якщо семантика гостьової мови істотно відрізняється від Go, зростають складність транспілятора, потреба у власному рантайм-шарі та втрати інтероперабельності. Отже, Go слід вважати не універсальною платформою, а сильною і практично доцільною основою для широкого, але чітко окресленого класу мов програмування.

#### **4.8. Як писати парсери та компілятори нових мов на Go**

Окремим практичним результатом дисертації є висновок, що Go доцільно розглядати не лише як ціль виконання, а і як мову реалізації самих транспіляторів. Детальна архітектура такого інструментарію вже була обґрунтована в розділах 1.5 та 2.3-2.5; у межах цього розділу важливо підкреслити оцінний висновок: наявність офіційних пакетів для синтаксичного аналізу, роботи з AST, типами, пакетами і форматуванням істотно знижує поріг побудови нової мови поверх Go.

Для діалектів і надмножин Go це означає можливість спиратися на вже наявні синтаксичні та семантичні моделі, а для інших мов програмування - використовувати Go-інструменти на пізніших етапах побудови цільового коду, перевірки його коректності та інтеграції в типовий ланцюжок `go build` і `go test`. Практичний ефект полягає в тому, що розробник нової мови концентрується переважно на фронтенді й власних мовних трансформаціях, а не на створенні з нуля всього інструментального стеку.

Отже, придатність Go в межах цієї дисертації підтверджується у двох взаємопов'язаних ролях: як середовища виконання для транспільованого коду і як інженерної бази для побудови самих мовних транспіляторів.

## **4.9. Обмеження та ризики запропонованого підходу**

Жодна платформа не є універсальною, і доцільність використання Go як основи для нових мов повинна супроводжуватися чітким розумінням її меж.

### **4.9.1. Залежність від еволюції самої Go**

Оскільки гостьова мова транспілюється до Go, будь-які суттєві зміни в специфікації або внутрішній поведінці компілятора можуть впливати на транспілятор. На практиці ця проблема пом'якшується сильною політикою сумісності Go, але повністю не зникає.

### **4.9.2. Складність діагностики помилок**

Чим складніше перетворення, тим складніше пояснювати користувачеві походження помилок. Якщо діагностика і помилки формуються вже після генерації Go-коду, вони можуть бути віддалені від місця вихідного коду в гостьовій програмі. Це вимагає або власної системи повідомлень про помилки і додаткового етапу перевірки типів на стороні транспілятора, або побудови карт відповідності позицій (source maps).

### **4.9.3. Ризик надмірного ускладнення транспілятора**

Парадоксально, але саме доступність багатьох мовних розширень може спокусити автора мови додавати все нові й нові конструкції. Якщо не контролювати межі, транспілятор перетворюється на велику систему з власною майже самодостатньою системою типів, що частково нівелює переваги повторного використання хост-платформи. Звідси впливає принцип стриманості: реалізовувати варто лише ті мовні розширення, практична користь від яких перевищує вартість їх реалізації та подальшого супроводу.

#### **4.9.4. Межі семантичного розриву**

Існує поріг, після якого гостьова мова настільки віддаляється від Go, що вся перевага інтероперабельності починає зникати, а інфраструктура виконання стає надто великою. Тоді може бути доцільнішим або обрати іншу платформу, або перейти до більш автономної реалізації.

### **4.10. Практичні рекомендації щодо застосування Go як платформи для нових мов**

Практичний зміст отриманих результатів зводиться до кількох стислих рекомендацій. По-перше, Go доцільно обирати тоді, коли новизна майбутньої мови зосереджена переважно на рівні синтаксису, абстракцій і предметно-орієнтованого моделювання, а не на радикально іншій моделі виконання. По-друге, нові конструкції слід проєктувати відразу з урахуванням їхньої схеми трансформації у Go і майбутньої інтероперабельності зі звичайним Go-кодом. По-третє, інженерний вииграш підходу найбільший тоді, коли розробник мови максимально використовує офіційні пакети Go та свідомо уникає конструкцій, що потребують додаткового рантайм-шару або нетипової моделі пам'яті.

Наведені рекомендації далі конкретизуються у вигляді методики прийняття рішення про вибір Go як цільової платформи.

### **4.11. Типові сценарії практичного застосування підходу**

Щоб оцінка доцільності використання Go як платформи не залишалася суто абстрактною, доцільно коротко окреслити типові сценарії, у яких результати цієї роботи мають безпосередню практичну цінність.

Найприроднішим класом застосувань є предметно-орієнтовані мови для серверних, інтеграційних та інфраструктурних задач: DSL для workflow, оркестрації сервісів, політик, правил або конфігурацій із виконуваною семантикою. У таких сценаріях Go вже часто виступає базовою інженерною платформою, тому транспіляція до неї дозволяє поєднати декларативність

предметного опису з доступом до наявних бібліотек, засобів конкурентності та типового середовища розгортання.

Другий важливий сценарій становлять більш виразні надмножини або робочі діалекти Go, яким потрібно додати ергономічні можливості без відмови від екосистеми, інструментів і культурних практик самої мови. Саме тут результати GoNext мають найбільш пряме прикладне значення, оскільки показують, які класи розширень можна додавати поверх Go без модифікації офіційного компілятора.

Третій сценарій пов'язаний з освітніми і дослідницькими мовами, де ключовим є швидке прототипування мовних ідей. У такому випадку Go зменшує інфраструктурну вартість перевірки гіпотез, оскільки дозволяє зосередити основні зусилля на фронтенді та семантиці нової мови, спираючись на вже готовий рантайм та інструментальний стек.

## **4.12. Методика прийняття рішення про вибір Go як цільової платформи**

На основі проведеного дослідження можна запропонувати практичну методику, яка допомагає автору нової мови або DSL вирішити, чи варто обирати саме Go як платформу реалізації.

### **Крок 1. Визначити, де міститься головна новизна майбутньої мови**

Якщо мовна новизна полягає переважно у:

- новому синтаксисі;
- зручніших абстракціях поверх звичних типів;
- декларативному описі дій;
- комбінаторах, механізмах на кшталт методів розширення, нових формах виклику;
- новому предметно-орієнтованому шарі,

то Go є сильним кандидатом. Якщо ж нова мова потребує окремого рантайму, спеціального байткоду, нетипової моделі пам'яті або жорсткого

контролю над розміщенням даних у пам'яті, то слід розглядати інші платформи.

## **Крок 2. Оцінити необхідний рівень інтероперабельності**

Якщо нова мова повинна:

- використовувати вже наявні Go-бібліотеки;
- співіснувати зі звичайним Go-кодом;
- поступово впроваджуватися в існуючі сервіси;
- експортувати бібліотеки для Go,

то використання Go як цілі є природним рішенням. Якщо ж інтероперабельність не має значення, перевага Go зменшується, і слід більше уваги приділити іншим критеріям.

## **Крок 3. Класифікувати майбутні мовні конструкції за профілем трансформації**

Для кожної ключової конструкції слід заздалегідь визначити, чи є вона:

- синтаксичною;
- структурною;
- репрезентаційною.

Якщо більшість мовних конструкцій належать до перших двох класів, то реалізація поверх Go, як правило, буде відносно простою й ефективною. Якщо ж ядро мови складається з репрезентаційних конструкцій, потрібно окремо дослідити, чи дозволяє Go безпечне представлення відповідних даних без надмірних витрат пам'яті та деградації швидкодії.

## **Крок 4. Оцінити придатність стандартних інструментів Go до задачі**

Потрібно окремо відповісти на запитання:

- чи можна використати `go/packages` для завантаження зовнішніх сигнатур;
- чи достатньо можливостей `go/types` для потрібного рівня семантичного аналізу;
- чи зручно генерувати цільовий код через `go/ast` і `go/format`;

- чи збігатиметься структура проєкту нової мови з пакетною моделлю Go.

Якщо відповіді переважно позитивні, витрати на реалізацію транслятора суттєво зменшуються.

### **Крок 5. Перевірити відповідність предметної області сильним сторонам Go**

Go особливо добре проявляє себе в серверних, інфраструктурних, мережних, CLI та інтеграційних сценаріях. Якщо нова мова орієнтована саме на такі задачі, то платформа надає не лише рантайм, а й сумісний стек бібліотек. Якщо ж предметна область тяжіє до суто браузерного оточення, спеціалізованих задач чисельних обчислень або нестандартних моделей виконання, Go може бути менш природним вибором.

### **Крок 6. Провести експеримент із однією-двома найризикованішими мовними конструкціями**

До початку повної реалізації доцільно зробити невеликий перевірочний прототип для тих конструкцій, які є найскладнішими. У багатьох випадках саме один такий експеримент, подібний до дослідження enum-типів у GoNext, дозволяє швидко виявити, чи не приховує хост-платформа критичних обмежень.

### **Крок 7. Прийняти рішення не за одним показником, а за балансом**

Підсумкове рішення має прийматися не лише за критерієм пікової продуктивності або лише за критерієм зручності реалізації. Саме баланс між швидкістю реалізації, якістю виконання, інтероперабельністю, придатністю до підтримки та перспективами розгортання визначає, чи є Go доцільною платформою в конкретному випадку.

Запропонована методика є одним із практичних результатів дисертації. Вона дозволяє перетворити загальну тезу про придатність Go на конкретну схему інженерного прийняття рішень.

#### 4.13. Перспективні напрями подальших досліджень

Хоча в межах дисертації отримано достатньо результатів для обґрунтування придатності Go як платформи для нових мов, робота також відкриває кілька перспективних напрямів.

Перший напрям — розробка узагальненої системи підсвітки синтаксису та серверів, імплементуючих language server protocol (LSP) [60] та debug adapter protocol (DAP) [59] для Go-подібних транспільованих мов. Це покращило б діагностику, налагодження й інтеграцію з IDE.

Другий напрям - дослідження можливості додавання концепції mutable та immutable змінних та спрощеної версії borrow checker [64] у мову Go, у тому числі і через транспіляцію. Основна мета полягає у тому, щоб зрозуміти, чи можна взагалі реалізувати аналог borrow checker-а для рантайма зі збирачем сміття, і якщо так - чи буде результуюча мова настільки ж складною, як Rust, чи завдяки наявності збирача, можна спростити реалізацію цієї функціональності і зробити її більш зручною для користувачів [15].

Третій напрям - адаптація системи безпечної конкурентності у Go. Для забезпечення відсутності стану гонитви у Rust додали Send і Sync трейти [19]. Swift для схожих задач має протокол Sendable [70]. Go немає механізмів перевірки відсутності станів гонитви на етапі компіляції програми, тож дослідження того, чи можна реалізувати такий механізм є перспективною задачею.

#### 4.14. Висновки до розділу

У четвертому розділі виконано комплексне оцінювання доцільності використання Go як платформи для нових мов програмування. Встановлено, що в багатьох практично орієнтованих сценаріях Go забезпечує дуже вдалий баланс між виконуваними властивостями, інтероперабельністю, доступністю інструментів для транспіляції і низькою вартістю реалізації транспілятора.

Показано, що найбільший вигрaш підхід дає для діалектів і надмножин Go, предметно-орієнтованих мов серверно-інфраструктурного профілю, а також для нових мов, свідомо спроектованих з орієнтацією на трансформацію у Go. Для таких систем переваги Go полягають у наявності зрілого рантайму, сучасного збирача сміття, засобів конкурентності, статичної компіляції, кросплатформності, великої бібліотечної екосистеми та потужних бібліотек для аналізу і генерації коду.

Водночас окреслено обмеження підходу: Go менш придатний для мов з радикально іншою моделлю виконання, ручною схемою керування пам'яттю або потребою в окремому байткодi чи спеціалізованому JIT. Також транспіляція в Go зазвичай не є доцільною для динамічно типізованих мов програмування. Усе це дозволяє зробити не абстрактний, а предметно-орієнтований висновок про межі застосовності запропонованого методу.

Сформульовано практичні рекомендації щодо проектування нових мов і транспіляторів на Go та окреслено перспективні напрями подальших досліджень. У сукупності це дає підстави стверджувати, що рантайм Go є не універсальною, але дуже вдалою та прагматично виправданою платформою для широкого класу нових мов програмування.

## ВИСНОВКИ

У дисертаційній роботі розв'язано актуальну науково-технічну задачу обґрунтування зручності та доцільності використання рантайму Go як платформи для побудови нових мов програмування. На відміну від традиційних підходів, де Go розглядається лише як окрема мова програмування для системної й серверної розробки, у роботі її проаналізовано як хост-платформу особливого типу: не як класичну багатомовну віртуальну машину і не як низькорівневий компіляторний фреймворк, а як практичну ціль для транспіляції мов, діалектів та предметно-орієнтованих надбудов.

Основні результати дисертації полягають у такому.

1. На основі аналізу сучасної літератури з LLVM, JVM, Truffle, транспіляції та теоретичних робіт про Go показано, що повторне використання наявних платформ є домінантним підходом у сучасному мовотворенні, а транспіляція виступає повноцінним методом реалізації мов.
2. Сформульовано систему критеріїв оцінювання придатності Go як хост-платформи, яка охоплює виконувальні властивості, виразні можливості хост-мови, інтегрованість, наявність інструментів для транспіляції та вартість реалізації.
3. Доведено, що Go має унікально вдале поєднання практичних властивостей, корисних для побудови нових мов: швидкий компілятор, статичну збірку, кроскомпіляцію, конкурентний і ефективний рантайм, сучасний збирач сміття, розвинену стандартну бібліотеку, велику екосистему пакетів та офіційні бібліотеки для синтаксичного аналізу, семантичного аналізу і генерації коду.
4. Розроблено узагальнену архітектуру транспілятора до Go, яка включає етапи синтаксичного аналізу, побудови внутрішнього AST, часткової типізації, нормалізації конструкцій, трансформації у вузли `go/ast` та

генерації звичайного Go-коду, придатного до компіляції офіційними інструментами.

5. Запропоновано класифікацію перетворень мовних розширень на синтаксичні, структурні та репрезентаційні. Показано, що саме ця класифікація дозволяє передбачати складність реалізації та характер накладних витрат нових мовних розширень поверх Go.
6. Створено експериментальний діалект GoNext, який використано як доказ можливості еволюційного розширення Go без модифікації її компілятора. У межах GoNext реалізовано повнофункціональні enum-типи, `match`, значення параметрів за замовчуванням, іменовані аргументи, методи розширення, узагальнені методи, універсальний синтаксис виклику функцій і короткий синтаксис для анонімних функцій.
7. На прикладі повнофункціональних enum-типів виявлено ключову межу використання Go як платформи: складні нові моделі даних можуть бути реалізовані, але повинні проєктуватися з урахуванням роботи збирача сміття, представлення покажчиків, узагальнень і семантики типів Go.
8. Показано, що значна частина корисних мовних розширень може бути реалізована поверх Go з нульовими або мінімальними накладними витратами під час виконання, якщо їх семантика виражається через локальні або структурні перетворення.
9. Доведено, що високий рівень інтероперабельності з екосистемою Go є не побічною перевагою, а центральним аргументом на користь використання Go як хост-платформи. Саме можливість природно використовувати наявні бібліотеки й поступово впроваджувати нову мову в існуючі проєкти робить цей підхід практично цінним.
10. Обґрунтовано, що GoNext є лише одним із прикладів. Отримані результати узагальнюються на ширший клас систем: транспіляцію інших мов у Go, створення нових мов, спроектованих спеціально під

Go-рантайм, а також побудову предметно-орієнтованих мов для серверних, інфраструктурних, інтеграційних і конфігураційних задач.

Проведене дослідження дає підстави стверджувати, що рантайм Go є доцільною платформою для побудови нових мов програмування тоді, коли:

- нова мова не потребує повністю автономної моделі виконання;
- важлива швидка реалізація та низька вартість побудови транспілятора;
- критичною є інтеграція з уже наявними Go-бібліотеками;
- бажано отримувати статично скомпільовані та кросплатформні артефакти;
- мовні новації стосуються насамперед виразності фронтенду, а не повної заміни хост-рантайму.

Таким чином, основну гіпотезу дисертації підтверджено. Go і її рантайм не є універсально найкращою платформою для будь-якої нової мови, однак для широкого і практично значущого класу мов програмування вони забезпечують винятково вдале поєднання простоти реалізації, якості виконання, доступності інструментів для транспіляції та бібліотечної інтероперабельності. Саме це робить Go важливим і перспективним фундаментом для подальших досліджень і практичної побудови нових мов програмування.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. A Guide to the Go Garbage Collector. URL: <https://go.dev/doc/gc-guide> (дата звернення: 15.01.2026).
2. Aho A. V., Lam M. S., Sethi R., Ullman J. D. Compilers: Principles, Techniques, and Tools. 2nd ed. Pearson. 2006.
3. Akre P. D., Pacharaney U. A Comprehensive Review of Mojo: A High-Performance Programming Language. In: 2025 6th International Conference on Mobile Computing and Sustainable Informatics (ICMCSI). 2025. DOI: 10.1109/ICMCSI64620.2025.10883176.
4. Bastidas Fuertes A., Pérez M., Meza Hormaza J. Transpilers: A Systematic Mapping Review of Their Usage in Research and Industry. Applied Sciences. 2023. DOI: 10.3390/app13063667.
5. Bastidas Fuertes A., Pérez M., Meza J. Transpiler-Based Architecture Design Model for Back-End Layers in Software Development. Applied Sciences. 2023. DOI: 10.3390/app132011371.
6. Bezanson J., Edelman A., Karpinski S., Shah V. B. Julia: A Fresh Approach to Numerical Computing. SIAM Review. 2017. Vol. 59, No. 1. P. 65–98. DOI: 10.1137/141000671.
7. Borgo Programming Language. URL: <https://borgo-lang.github.io/> (дата звернення: 15.01.2026).
8. Broggi D., Liu Y. On the Interoperability of Programming Languages via Translation. CSCE. 2023. DOI: 10.1109/CSCE60160.2023.00413.
9. CUE. URL: <https://cuelang.org/> (дата звернення: 15.01.2026).
10. Cherny-Shahar T., Yehudai A. Multi-Lingual Development & Programming Languages Interoperability: An Empirical Study. arXiv. 2024. DOI: 10.48550/arXiv.2411.08388.
11. Chisnall D. Modern Intermediate Representations. LLVM IR and Transform Pipeline. Modern Processor Architectures. JIT Compilation. URL:

- <https://llvm.org/devmtg/2017-06/1-Davis-Chisnall-LLVM-2017.pdf> (дата звернення: 15.01.2026).
12. Cox R. Backward Compatibility, Go 1.21, and Go 2. The Go Blog. 2023. URL: <https://go.dev/blog/compat> (дата звернення: 15.01.2026).
  13. Cox R., Griesemer R., Pike R., Taylor I. L., Thompson K. The Go Programming Language and Environment. Communications of the ACM. 2022. DOI: 10.1145/3488716.
  14. Cpp2 and cppfront — an experimental ‘C++ syntax 2’ and its first compiler. URL: <https://hsutter.github.io/cppfront/> (дата звернення: 15.01.2026).
  15. Crichton W. The Usability of Ownership. arXiv. 2021. DOI: 10.48550/arXiv.2011.06171.
  16. Cross-compiling made easy with Golang | Opensource.com. URL: <https://opensource.com/article/21/1/go-cross-compiling> (дата звернення: 15.01.2026).
  17. Dilley N., Lange J. An Empirical Study of Messaging Passing Concurrency in Go Projects. SANER. 2019. DOI: 10.1109/SANER.2019.8668036.
  18. Dubochet G., Odersky M. Compiling Structural Types on the JVM: A Comparison of Reflective and Generative Techniques from Scala’s Perspective. Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS ’09). 2009. DOI: 10.1145/1565824.1565829.
  19. Extensible Concurrency with Send and Sync. URL: <https://doc.rust-lang.org/book/ch16-04-extensible-concurrency-sync-and-send.html> (дата звернення: 15.01.2026).
  20. Extension members - C | Microsoft Learn. URL: <https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods> (дата звернення: 15.01.2026).

21. Farshidi S., Deldar M., Jansen S. A decision model for programming language ecosystem selection: Seven industry case studies. *Information and Software Technology*. 2021. DOI: 10.1016/j.infsof.2021.106640.
22. Felleisen M. On the Expressive Power of Programming Languages. *Science of Computer Programming*. 1991. DOI: 10.1016/0167-6423(91)90036-W.
23. Feng Q., Ji H., Ma X., Liang P. Cross-Language Dependencies: An Empirical Study of Kotlin-Java. *ESEM*. 2024. DOI: 10.1145/3674805.3686680.
24. Forkert P. P., Ivanchenko M. G. Implementing extension methods and generic methods in Go programming language dialect. *Системні технології*, 2026. Т.163. С. 111-121. DOI: <https://doi.org/10.34185/1562-9945-2-163-2026-10>.
25. Forkert P. P., Ivanchenko M. G. A Generalized Transpiler Architecture For Languages Targeting Go. *Стан, досягнення і перспективи інформаційних систем і технологій: тези доповідей XXVI всеукраїнської науково-технічної конференції молодих вчених, аспірантів та студентів*. Одеса, 16-17 квітня 2026. С. 143-145.
26. Forkert P. P., Ivanchenko M. G. Implementing Default Parameter Values In Go Programming Language Dialect. *Математичне та програмне забезпечення інтелектуальних систем (МПЗІС-2025): тези доповідей XXIII міжнародної науково-практичної конференції*. Дніпро, 19-21 листопада 2025. С. 8-9.
27. Forkert P. P., Ivanchenko M. G. Implementing Named Arguments In Go Programming Language Dialect. *Актуальні проблеми автоматизації та інформаційних технологій*, 2025. Т.29. С. 3-11. DOI: <https://dx.doi.org/10.15421/432501>.
28. Forkert P. P., Sydorova M. G. Advantages Of Golang As A Foundation For New Programming Languages. *Математичне та програмне забезпечення інтелектуальних систем (МПЗІС-2023): тези доповідей XXI*

- міжнародної науково-практичної конференції. Дніпро, 22-24 листопада 2023. С. 7-8.
29. Forkert P. P., Sydorova M. G. Challenges Of Using Golang As A Foundation For New Programming Languages. Сучасні інформаційні системи та технології: тези доповідей VI Всеукраїнської науково-практичної інтернет-конференція студентів, аспірантів та молодих вчених. Хмельницький, 30 листопада 2023, С. 55-56.
30. Forkert P. P., Sydorova M. G. Improving Enums In Go Programming Language Dialect. Інженерія програмного забезпечення і передові інформаційні технології (SOFT TECH-2024): тези доповідей VI Міжнародної науково-практичної конференції молодих вчених та студентів. Київ, 21-23 травня 2024. С. 148-150.
31. Forkert P. P., Sydorova M. G. Integrating Full-Featured Enums Into Go Programming Language. Актуальні проблеми автоматизації та інформаційних технологій, 2023. Т.27. С. 3-16. DOI: <http://dx.doi.org/10.15421/432301>.
32. Freitas A., Baptista T., Henriques P. R. Bridging Language Barriers: A Comparative Review and Empirical Evaluation of Source-To-Source Transpilers. SLATE 2025. DOI: 10.4230/OASISs.SLATE.2025.11.
33. Frequently Asked Questions (FAQ) - The Go Programming Language. URL: [https://go.dev/doc/faq#generic\\_methods](https://go.dev/doc/faq#generic_methods) (дата звернення: 15.01.2026).
34. Glossary — Python 3.10.20 documentation. URL: <https://docs.python.org/3.10/glossary.html#term-argument> (дата звернення: 15.01.2026).
35. Go 1 and the Future of Go Programs. URL: <https://go.dev/doc/go1compat> (дата звернення: 15.01.2026).
36. Go runtime HACKING guide. URL: <https://go.dev/src/runtime/HACKING> (дата звернення: 15.01.2026).

37. Griesemer R. spec: generic methods for Go. GitHub issue #77273. 2026. URL: <https://github.com/golang/go/issues/77273> (дата звернення: 15.01.2026).
38. Griesemer R., Hu R., Kokke W., Lange J., Taylor I. L., Toninho B., Wadler P., Yoshida N. Featherweight Go. Proceedings of the ACM on Programming Languages. 2020. DOI: 10.1145/3428217.
39. Howard J. Analyzing Go Build Times. 2024. URL: <https://blog.howardjohn.info/posts/go-build-times/> (дата звернення: 15.01.2026).
40. Hu R., Lange J., Toninho B., Wadler P., Griesemer R., Randall K. Welterweight Go: Boxing, Structural Subtyping, and Generics. Proceedings of the ACM on Programming Languages. 2026. DOI: 10.1145/3776721.
41. Jakimoski K., Chavkovski B. Analysis of the New Generation Source-to-Source Compilers Using the Google Web Toolkit. IJIEEB. 2022. Vol. 14, No. 5. P. 32–41. DOI: 10.5815/ijieeb.2022.05.04.
42. Jones R., Hosking A., Moss J. E. B., Blackburn S. M. Concurrent garbage collection. In: The Garbage Collection Handbook. 2nd ed. 2023. DOI: 10.1201/9781003276142-15.
43. Kölling M. Principles of Educational Programming Language Design. Informatics in Education. 2024. DOI: 10.15388/infedu.2024.29.
44. Křikava F., Miller H., Vitek J. Scala implicits are everywhere: a large-scale study of the use of Scala implicits in the wild. Proceedings of the ACM on Programming Languages. 2019. DOI: 10.1145/3360589.
45. Lattner C., Adve V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. CGO. 2004. DOI: 10.1109/CGO.2004.1281665.
46. Lattner C., Adve V. The LLVM Compiler Framework and Infrastructure Tutorial. LCPC. 2005. DOI: 10.1007/11532378\_2.

47. Lattner C., Amini M., Bondhugula U., Cohen A., Davis A., Pienaar J., Riddle R., Shpeisman T., Vasilache N., Zinenko O. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. CGO. 2021. DOI: 10.1109/CGO51591.2021.9370308.
48. Leduc M., Jouneaux G., Degueule T., Le Guernic G., Barais O., Combemale B. Automatic Generation of Truffle-based Interpreters for Domain-Specific Languages. The Journal of Object Technology. 2020. DOI: 10.5381/jot.2020.19.2.a1.
49. Lee Y., Gopinathan K., Yang Z., Flatt M., Sergey I. DSLs in Racket: You Want It How, Now? SLE. 2024. DOI: 10.1145/3687997.3695645.
50. Li H. Kernel-FFI: Transparent Foreign Function Interfaces for Interactive Notebooks. arXiv. 2025. DOI: 10.48550/arXiv.2507.23205.
51. Li W. H., White D. R., Singer J. JVM-hosted languages. PPPJ. 2013. DOI: 10.1145/2500828.2500838.
52. Marrão B., Leal J. P., Queirós R. Osiris: A Multi-Language Transpiler for Educational Purposes. ICPEC 2025. DOI: 10.4230/OASlcs.ICPEC.2025.17.
53. Mateus B. G., Martinez M., Kolski C. Learning migration models for supporting incremental language migrations of software applications. Information and Software Technology. 2023. DOI: 10.1016/j.infsof.2022.107082.
54. Mernik M., Heering J., Sloane A. M. When and How to Develop Domain-Specific Languages. ACM Computing Surveys. 2005. Vol. 37, No. 4. P. 316–344. DOI: 10.1145/1118890.1118892.
55. NDepend Blog. C 15 Unions. URL: <https://blog.ndepend.com/csharp-unions/> (дата звернення: 15.01.2026).
56. Nicolini T., Hora A., Figueiredo E. On the Usage of New JavaScript Features Through Transpilers: The Babel Case. IEEE Software. 2024. DOI: 10.1109/MS.2023.3243858.

57. Nim Programming Language. URL: <https://nim-lang.org/> (дата звернення: 15.01.2026).
58. Odersky M., Rompf T. Unifying Functional and Object-Oriented Programming with Scala. *Communications of the ACM*. 2014. Vol. 57, No. 4. P. 76–86. DOI: 10.1145/2591013.
59. Official page for Debug Adapter Protocol. URL: <https://microsoft.github.io/debug-adapter-protocol/> (дата звернення: 15.01.2026).
60. Official page for Language Server Protocol. URL: <https://microsoft.github.io/language-server-protocol/> (дата звернення: 15.01.2026).
61. Oliveira D., Santos R., de Oliveira B., Monperrus M., Castor F., Madeiral F. Understanding Code Understandability Improvements in Code Reviews. *IEEE Transactions on Software Engineering*. 2025. DOI: 10.1109/TSE.2024.3453783.
62. Oliveira V., Teixeira L., Ebert F. On the Adoption of Kotlin on Android Development: A Triangulation Study. *Proceedings of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2020. P. 206–216. DOI: 10.1109/SANER48275.2020.9054859.
63. One VM to rule them all. *Onward!* 2013. DOI: 10.1145/2509578.2509581.
64. References and Borrowing - The Rust Programming Language. URL: <https://doc.rust-lang.org/stable/book/ch04-02-references-and-borrowing.html> (дата звернення: 15.01.2026).
65. Rytz L., Odersky M. Named and default arguments for polymorphic object-oriented languages: a discussion on the design implemented in the Scala language. *SAC '10*. 2010. DOI: 10.1145/1774088.1774523.

- 66.Scarsbrook J. D., Utting M., Ko R. K. L. TypeScript's Evolution: An Analysis of Feature Adoption Over Time. MSR. 2023. DOI: 10.1109/msr59073.2023.00027.
- 67.Source Map Format Specification. URL: <https://tc39.es/ecma426/> (дата звернення: 15.01.2026).
- 68.Starlark Language. URL: <https://starlark-lang.org/> (дата звернення: 15.01.2026).
- 69.Sutter H. Unified function call syntax (UFCS). ISO/IEC JTC1/SC22/WG21 paper P3021R0. 2023. URL: <https://open-std.org/JTC1/SC22/WG21/docs/papers/2023/p3021r0.pdf> (дата звернення: 15.01.2026).
- 70.Swift Sendable. URL: <https://developer.apple.com/documentation/Swift/Sendable> (дата звернення: 15.01.2026).
- 71.Syme D. The Early History of F#. Proceedings of the ACM on Programming Languages. 2020. DOI: 10.1145/3386325.
- 72.Taft S. T. Rigorous pattern matching as a language feature. International Journal on Software Tools for Technology Transfer. 2025. DOI: 10.1007/s10009-025-00788-z.
- 73.The Dhall configuration language. URL: <https://dhall-lang.org/> (дата звернення: 15.01.2026).
- 74.The Go Programming Language Specification. URL: [https://go.dev/ref/spec#Type\\_unification\\_rules](https://go.dev/ref/spec#Type_unification_rules) (дата звернення: 15.01.2026).
- 75.The Green Tea Garbage Collector. The Go Blog. URL: <https://go.dev/blog/greenteagc> (дата звернення: 15.01.2026).
- 76.The Oden Programming Language. URL: <https://oden-lang.github.io/> (дата звернення: 15.01.2026).

77. Titzer B. L., Gilbert E., Teo B. W. J. Flexible Non-intrusive Dynamic Instrumentation for WebAssembly. arXiv. 2024. DOI: 10.48550/arXiv.2403.07973.
78. Type Parameters Proposal. URL: <https://go.golang.org/proposal/+HEAD/design/43651-type-parameters.md#No-parameterized-methods> (дата звернення: 15.01.2026).
79. Type layout - The Rust Reference. URL: <https://doc.rust-lang.org/reference/type-layout.html#primitive-representation-of-enums-with-fields> (дата звернення: 15.01.2026).
80. Ufliand E. The Go Ecosystem in 2025: Key Trends in Frameworks, Tools, and Developer Practices. The GoLand Blog. 2025. URL: <https://blog.jetbrains.com/go/2025/11/10/go-language-trends-ecosystem-2025/> (дата звернення: 15.01.2026).
81. Wimmer C., Würthinger T. Truffle: a self-optimizing runtime system. SPLASH. 2012. DOI: 10.1145/2384716.2384723.
82. Yang H., Lian W., Wang S., Cai H. Demystifying Issues, Challenges, and Solutions for Multilingual Software Development. ICSE. 2023. DOI: 10.1109/ICSE48619.2023.00157.
83. altJS compile-to-JavaScript language list. URL: <https://altjs.org/> (дата звернення: 15.01.2026).
84. d5/tengo. The Tengo Language. URL: <https://github.com/d5/tengo> (дата звернення: 15.01.2026).
85. dotnet/csharplang. Unions proposal. URL: <https://github.com/dotnet/csharplang/blob/main/proposals/unions.md> (дата звернення: 15.01.2026).
86. go/ directory - go - Go Packages. URL: <https://pkg.go.dev/go> (дата звернення: 15.01.2026).
87. google/go-jsonnet. JSONNet implementation in Go. URL: <https://github.com/google/go-jsonnet> (дата звернення: 15.01.2026).

- 88.google/grumpy. Python to Go transcompiler and runtime. URL: <https://github.com/google/grumpy> (дата звернення: 15.01.2026).
- 89.tools module - golang.org/x/tools - Go Packages. URL: <https://pkg.go.dev/golang.org/x/tools> (дата звернення: 15.01.2026).
- 90.Šipek M., Mihaljević B., Radovan A. Exploring Aspects of Polyglot High-Performance Virtual Machine GraalVM. Proceedings of the 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). 2019. P. 1671–1676. DOI: 10.23919/MIPRO.2019.8756917.
- 91.Форкерт П. П., Іванченко М. Г. Короткий синтаксис для анонімних функцій в діалекті мови програмування Go. Інформаційні Технології: Моделі, Алгоритми, Системи (ITMAS – 2025): тези доповідей VI Міжнародної науково-практичної інтернет конференції. Миколаїв, 16-17 листопада 2025. С. 531-533.
- 92.Форкерт П. П., Іванченко М. Г. Універсальний синтаксис виклику функцій в діалекті мови програмування Go. Інформаційні технології в освіті та науці: збірник наукових праць IV Міжнародної науково-практичної конференції. Запоріжжя, 20 травня 2025. С. 561-565.

## ДОДАТКИ

### Додаток А

#### **Статті у наукових фахових виданнях України:**

1. Forkert P. P., Sydorova M. G. Integrating full-featured enums into Go programming language. *Актуальні проблеми автоматизації та інформаційних технологій*, т. 27, Дніпро, 2023, с. 3-16. DOI: <http://dx.doi.org/10.15421/432301> (особистий внесок Форкерта П.П.: провів аналіз підходів до реалізації мов програмування та можливостей використання транспіляції в Go, сформулював вимоги до повнофункціональних епит-типів у діалекті GoNext, запропонував синтаксис їх оголошення, дослідив та порівняв кілька варіантів представлення епит-типів у Go-кодi, зокрема з урахуванням узагальнень, вказівників, роботи збирача сміття та накладних витрат, розробив підхід до генерації конструкторів, Match-методів і допоміжних методів доступу, а також проаналізував сумісність запропонованого рішення з екосистемою Go; Іванченко М.Г.: постановка завдання, узагальнення отриманих результатів).
2. Forkert P. P., Ivanchenko M. G. Implementing named arguments in go programming language dialect. *Актуальні проблеми автоматизації та інформаційних технологій*, т. 29, Дніпро, 2025, с. 3-11. DOI: <https://dx.doi.org/10.15421/432501>. (особистий внесок Форкерта П.П.: провів порівняльний аналіз підходів до реалізації іменованих аргументів у сучасних мовах програмування; обґрунтував вибір call-site-підходу, за якого іменовані аргументи не потребують змін у визначеннях функцій, запропонував синтаксис іменованих аргументів для GoNext, дослідив його сумісність із граматикою Go та розробив алгоритм транспіляції, що передбачає визначення сигнатури функції, перевірку помилкових і дубльованих імен параметрів, окремо проаналізував інтеграцію іменованих аргументів із механізмом значень параметрів за

замовчуванням; Іванченко М.Г.: постановка мети дослідження, контроль та узагальнення отриманих результатів).

3. Forkert P. P., Ivanchenko M. G. Implementing extension methods and generic methods in Go programming language dialect. *Системні технології*, т. 163, Дніпро, 2026, с. 111-121. DOI: <https://doi.org/10.34185/1562-9945-2-163-2026-10>. (особистий внесок Форкерта П.П.: провів аналіз реалізацій методів розширення в сучасних мовах програмування та порівняв їх з універсальним синтаксисом виклику функцій, сформулював вимоги до реалізації методів розширення в GoNext з урахуванням мінімальних накладних витрат і повної сумісності зі стандартним Go, запропонував синтаксис оголошення методів розширення через модифікатор *extension*, механізм *import extension* для використання функцій наявних Go-бібліотек як методів розширення, а також алгоритм переписування викликів виду *value.Method(args)* у звичайні виклики функцій, дослідив застосування цього підходу для підтримки узагальнених методів, які відсутні у стандартному Go, та проаналізував інтероперабельність запропонованого рішення з існуючим кодом на Go; Іванченко М.Г.: постановка задачі, аналіз результатів).

**Наукові праці, які засвідчують апробацію матеріалів дисертації:**

1. Forkert P. P., Sydorova M. G., Honcharova Yu. S. MODERN ARCHITECTURE OF DYNAMICALLY TYPED PROGRAMMING LANGUAGE VMS. *Тези доповідей IV Всеукраїнської науково-практичної конференції молодих науковців та здобувачів вищої освіти «Сучасні науково-технічні дослідження у контексті мовного простору»*, Дніпро, 11 травня 2023, с. 188-190. URL: <https://www.confcontact.com/2023-suchasni-ntd/3-Forkert-Sydorova-Honcharova.pdf> (особистий внесок Форкерта П.П.: провів аналіз

*архітектур сучасних віртуальних машин динамічно типізованих мов програмування, зокрема V8, SpiderMonkey та JavaScriptCore, узагальнив спільну для них багаторівневу схему виконання з інтерпретатором і JIT-компіляторами; Іванченко М.Г.: постановка завдання, узагальнення результатів; Гончарова Ю.С.: аналіз формулювань).*

2. Forkert P. P., Sydorova M. G. ADVANTAGES OF GOLANG AS A FOUNDATION FOR NEW PROGRAMMING LANGUAGES. *Тези доповідей XXI міжнародної науково-практичної конференції «МАТЕМАТИЧНЕ ТА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ІНТЕЛЕКТУАЛЬНИХ СИСТЕМ (МПЗІС-2023)»,* Дніпро, 22-24 листопада 2023, с. 7-8. URL: <https://mpzis.dnu.dp.ua/wp-content/uploads/2023/11/mpzis-2023.pdf> *(особистий внесок Форкерта П.П.: проаналізував переваги Go як основи для побудови нових мов програмування, зокрема продуктивність рантайму, підтримку конкурентності, збирач сміття, кроскомпіляцію та екосистему бібліотек, обґрунтував транспіляцію як основний спосіб використання Go-рантайму гостьовими мовами; Іванченко М.Г.: постановка завдання, узагальнення результатів).*
3. Forkert P. P., Sydorova M. G. CHALLENGES OF USING GOLANG AS A FOUNDATION FOR NEW PROGRAMMING LANGUAGES. *Тези доповідей VI Всеукраїнської науково-практичної інтернет-конференція студентів, аспірантів та молодих вчених «Сучасні інформаційні системи та технології»,* Хмельницький, 30 листопада 2023, с. 55-56. URL: <https://kntu.net.ua/ukr/content/download/110490/623634/file/CICT2023.pdf> *(особистий внесок Форкерта П.П.: систематизував обмеження використання Go як хост-платформи, зокрема відсутність стабільного API рантайму та готових інструментів транспіляції,*

- обмеженість узагальнень, і запропонував шляхи їх подолання; Іванченко М.Г.: постановка завдання, узагальнення результатів).*
4. Forkert P. P., Sydorova M. G. IMPROVING ENUMS IN GO PROGRAMMING LANGUAGE DIALECT. *Тези доповідей VI Міжнародної науково-практичної конференції молодих вчених та студентів «ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ І ПЕРЕДОВІ ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ (SOFT TECH-2024)», Київ, 21-23 травня 2024, с. 148-150. URL: <https://drive.google.com/file/d/18bZ9QBure7U08rbqiHmxWglTrK1C9D4L/view> (особистий внесок Форкерта П.П.: запропонував для GoNext синтаксис зіставляння із взірцем match, адаптований до стилю оператора switch у Go, із захисними умовами та перевіркою вичерпності, а також схему його транспіляції у виклики згенерованих Match-методів enum-типів; Іванченко М.Г.: постановка завдання, аналіз результатів).*
  5. Форкерт П. П., Іванченко М. Г. УНІВЕРСАЛЬНИЙ СИНТАКСИС ВИКЛИКУ ФУНКЦІЙ В ДІАЛЕКТІ МОВИ ПРОГРАМУВАННЯ GO. *Збірник наукових праць IV Міжнародної науково-практичної конференції «ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ В ОСВІТІ ТА НАУЦІ», Запоріжжя, 20 травня 2025, с. 561-565 (особистий внесок Форкерта П.П.: проаналізував реалізації універсального синтаксису виклику функцій у сучасних мовах програмування, запропонував варіант його інтеграції в GoNext зі збереженням зворотної сумісності з наявним Go-кодом та правила розв'язання неоднозначностей під час пошуку функцій; Іванченко М.Г.: постановка завдання, узагальнення результатів).*
  6. Форкерт П. П., Іванченко М. Г. КОРОТКИЙ СИНТАКСИС ДЛЯ АНОНІМНИХ ФУНКЦІЙ В ДІАЛЕКТІ МОВИ ПРОГРАМУВАННЯ GO. *Тези доповідей VI Міжнародної науково-практичної інтернет конференції «ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ: МОДЕЛІ, АЛГОРИТМИ,*

- СИСТЕМИ (ITMAS – 2025)», Миколаїв, 16-17 листопада 2025, с. 531-533. URL: <https://itconf.nuos.edu.ua/2025/publications/short-syntax-for-anonymous-functions-in-go-programming-language-dialect/> (особистий внесок Форкерта П.П.: запропонував розширення граматики GoNext короткими літералами анонімних функцій та підхід до їх транпіляції з відновленням пропущених типів шляхом уніфікації з очікуваним типом функцій; Іванченко М.Г.: постановка завдання, контроль результатів).*
7. Forkert P. P., Ivanchenko M. G. IMPLEMENTING DEFAULT PARAMETER VALUES IN GO PROGRAMMING LANGUAGE DIALECT. *Тези доповідей XXIII міжнародної науково-практичної конференції «МАТЕМАТИЧНЕ ТА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ІНТЕЛЕКТУАЛЬНИХ СИСТЕМ (МПЗІС-2025)», Дніпро, 19-21 листопада 2025, с. 8-9. URL: <https://mpzis.dnu.dp.ua/wp-content/uploads/2025/11/%D0%9C%D0%9F%D0%97%D0%86%D0%A1-2025.pdf> (особистий внесок Форкерта П.П.: запропонував синтаксис значень параметрів за замовчуванням для GoNext та алгоритм їх транпіляції через генерацію допоміжних функцій з опціональними параметрами, що коректно обробляє значення, залежні від області видимості означення функції; Іванченко М.Г.: постановка завдання, аналіз результатів).*
8. Forkert P. P., Ivanchenko M. G. A GENERALIZED TRANSPILER ARCHITECTURE FOR LANGUAGES TARGETING GO. *Тези доповідей XXVI всеукраїнської науково-технічної конференції молодих вчених, аспірантів та студентів «СТАН, ДОСЯГНЕННЯ І ПЕРСПЕКТИВИ ІНФОРМАЦІЙНИХ СИСТЕМ І ТЕХНОЛОГІЙ», Одеса, 16-17 квітня 2026, с. 143-145. URL: [https://drive.google.com/file/d/1QEcwdAJ9J4nzSCRZZ5X\\_eW5JFQE7a5vh/view](https://drive.google.com/file/d/1QEcwdAJ9J4nzSCRZZ5X_eW5JFQE7a5vh/view) (особистий внесок Форкерта П.П.: описав узагальнену багатоступеневу архітектуру транпілятора для мов, що компілюються*

*в Go, виокремив нормалізацію конструкцій як ключовий етап, що забезпечує модульність; Іванченко М.Г.: постановка завдання, узагальнення результатів).*

## Код транпілятору GoNext в Go

```
// cmd/gonext/main.go
package main

import (
    "fmt"
    "os"

    "gonext/internal/transpiler"
)

func main() {
    if len(os.Args) < 3 {
        die("usage: gonext <transpile|run> <file.gn>")
    }

    switch os.Args[1] {
    case "transpile":
        out, err := transpiler.TranspileFile(os.Args[2])
        if err != nil {
            die(err.Error())
        }
        fmt.Print(out)
    case "run":
        if err := transpiler.RunFile(os.Args[2]); err != nil {
            die(err.Error())
        }
    default:
        die("unknown command: " + os.Args[1])
    }
}

func die(message string) {
    fmt.Fprintln(os.Stderr, message)
    os.Exit(1)
}
```

```

}

// internal/transpiler/analyze.go
package transpiler

import (
    "fmt"
    "go/build"
    "go/token"

    "gonext/internal/frontend/ast"
    "gonext/internal/frontend/parser"
    "gonext/internal/frontend/srcimporter"
    "gonext/internal/frontend/types"
)

const preludeBody = `
func __gnx_named[T any](name string, value T) T {
    return value
}

func __gnx_match[T any](value T) T {
    return value
}

func __gnx_guard[T any](pattern T, cond bool) T {
    return pattern
}
`

const gnImportPath = "gonext/gn"

const gnSource = `package gn

// Optional carries an explicitly provided value of an argument whose
// parameter declares a default value. A zero Optional means "use the
// default".
type Optional[T any] struct {

```

```

    Exists bool
    Value T
}

func Some[T any](value T) Optional[T] {
    return Optional[T]{Exists: true, Value: value}
}

func None[T any]() Optional[T] {
    return Optional[T]{}
}
`

type analyzer struct {
    fset    *token.FileSet
    imp     types.Importer
    prelude *ast.File
}

type gnImporter struct {
    fset    *token.FileSet
    base    types.Importer
    gnPkg   *types.Package
}

func (g *gnImporter) Import(path string) (*types.Package, error) {
    if path != gnImportPath {
        return g.base.Import(path)
    }
    if g.gnPkg == nil {
        file, err := parser.ParseFile(g.fset, "gn.go", gnSource, 0)
        if err != nil {
            return nil, err
        }
        conf := types.Config{Importer: g.base}
        pkg, err := conf.Check(gnImportPath, g.fset, []*ast.File{file}, nil)
        if err != nil {
            return nil, err
        }
    }
}

```

```

    }
    g.gnPkg = pkg
}
return g.gnPkg, nil
}

func newAnalyzer(fset *token.FileSet, pkgName string) (*analyzer, error) {
    prelude, err := parser.ParseFile(fset, "gnx_prelude.go", "package
"+pkgName+"\n"+preludeBody, 0)
    if err != nil {
        return nil, fmt.Errorf("internal error: prelude does not parse: %w",
err)
    }
    base := srcimporter.New(&build.Default, fset, map[string]*types.Package{})
    return &analyzer{
        fset:    fset,
        imp:    &gnImporter{fset: fset, base: base},
        prelude: prelude,
    }, nil
}

func (a *analyzer) check(file *ast.File) (*types.Package, *types.Info) {
    info := &types.Info{
        Types:    map[ast.Expr]types.TypeAndValue{},
        Defs:     map[*ast.Ident]types.Object{},
        Uses:     map[*ast.Ident]types.Object{},
        Selections: map[*ast.SelectorExpr]*types.Selection{},
    }
    conf := types.Config{
        Error:    func(error) {},
        Importer: a.imp,
    }
    pkg, _ := conf.Check(file.Name.Name, a.fset, []*ast.File{file, a.prelude},
info)
    return pkg, info
}

func validType(t types.Type) bool {

```

```

    if t == nil {
        return false
    }
    basic, ok := t.(*types.Basic)
    return !ok || basic.Kind() != types.Invalid
}

func concreteType(t types.Type) types.Type {
    return types.Default(t)
}

func typeToExpr(t types.Type, pkg *types.Package, file *ast.File) (ast.Expr,
error) {
    s := types.TypeString(t, fileQualifier(pkg, file))
    expr, err := parser.ParseExpr(s)
    if err != nil {
        return nil, fmt.Errorf("cannot express type %s in the generated
code: %w", s, err)
    }
    clearPositions(expr)
    return expr, nil
}

func fileQualifier(pkg *types.Package, file *ast.File) types.Qualifier {
    aliases := map[string]string{}
    for _, spec := range file.Imports {
        p, err := importPathOf(spec)
        if err != nil {
            continue
        }
        alias := ""
        if spec.Name != nil {
            alias = spec.Name.Name
        }
        aliases[p] = alias
    }
    return func(other *types.Package) string {
        if other == pkg {

```

```

        return ""
    }
    if alias, ok := aliases[other.Path()]; ok && alias != "" {
        return alias
    }
    return other.Name()
}
}

func importPathOf(spec *ast.ImportSpec) (string, error) {
    return unquote(spec.Path.Value)
}

func unquote(s string) (string, error) {
    if len(s) >= 2 && s[0] == '"' && s[len(s)-1] == '"' {
        return s[1 : len(s)-1], nil
    }
    return "", fmt.Errorf("invalid import path %s", s)
}

// internal/transpiler/defaults.go
package transpiler

import (
    "fmt"
    "go/token"
    "gonext/internal/frontend/ast"
    "gonext/internal/frontend/parser"
    "reflect"
    "strings"
)

const defaultSuffix = "__default"

func gnSomeExpr(value ast.Expr) ast.Expr {
    return &ast.CallExpr{
        Fun: &ast.SelectorExpr{X: ast.NewIdent("gn"), Sel:
ast.NewIdent("Some")},

```

```

    Args: []ast.Expr{value},
  }
}

func gnNoneExpr(typeExpr ast.Expr) ast.Expr {
  return &ast.CallExpr{
    Fun: &ast.IndexExpr{
      X: &ast.SelectorExpr{X: ast.NewIdent("gn"), Sel:
ast.NewIdent("None")},
      Index: typeExpr,
    },
  }
}

func parseTypeSrc(src string) (ast.Expr, error) {
  expr, err := parser.ParseExpr(src)
  if err != nil {
    return nil, err
  }
  clearPositions(expr)
  return expr, nil
}

func clearPositions(root ast.Node) {
  ast.Inspect(root, func(n ast.Node) bool {
    if n == nil {
      return false
    }
    v := reflect.ValueOf(n).Elem()
    for _, f := range v.Fields() {
      f := f
      if f.Type() == posType {
        f.SetInt(0)
      }
    }
    return true
  })
}

```

```

var posType = reflect.TypeOf[token.Pos]()

func appendDefaultWrappers(st *state) error {
    tokenFile := st.fset.File(st.file.Pos())
    for _, name := range st.norm.defOrder {
        def := st.norm.defaults[name]
        decl := findFuncDecl(st.file, name)
        if decl == nil {
            return fmt.Errorf("internal error: declaration of %s not found",
name)
        }
        resultsSrc := ""
        if decl.Type.Results != nil && len(decl.Type.Results.List) > 0 {
            start := tokenFile.Offset(decl.Type.Results.Pos())
            end := tokenFile.Offset(decl.Type.Results.End())
            resultsSrc = " " + st.norm.src[start:end]
        }

        var b strings.Builder
        var params, callArgs []string
        for i, p := range def.params {
            if i < def.required {
                params = append(params, p.name+" "+p.typeSrc)
            } else {
                params = append(params, p.name+"Opt
gn.Optional["+p.typeSrc+"]")
            }
            callArgs = append(callArgs, p.name)
        }
        fmt.Fprintf(&b, "func %s%s(%s)%s {\n", name, defaultSuffix,
strings.Join(params, ", "), resultsSrc)
        for i := def.required; i < len(def.params); i++ {
            p := def.params[i]
            fmt.Fprintf(&b, "\tvar %s %s\n", p.name, p.typeSrc)
            fmt.Fprintf(&b, "\tif %sOpt.Exists {\n\t\t%s = %sOpt.Value\n\t}
else {\n\t\t%s = %s\n\t}\n",
                p.name, p.name, p.name, p.name, p.defaultSrc)

```

```

    }
    b.WriteString("\t")
    if resultsSrc != "" {
        b.WriteString("return ")
    }
    fmt.Fprintf(&b, "%s(%s)\n}\n", name, strings.Join(callArgs, ", "))

    wrapper, err := parseSingleDecl(st.fset, b.String())
    if err != nil {
        return fmt.Errorf("function %s: cannot generate the default-
values helper: %w", name, err)
    }
    st.file.Decls = append(st.file.Decls, wrapper)
}
addImport(st.file, gnImportPath)
st.usesGn = true
return nil
}

func findFuncDecl(file *ast.File, name string) *ast.FuncDecl {
    for _, decl := range file.Decls {
        if fn, ok := decl.(*ast.FuncDecl); ok && fn.Recv == nil &&
fn.Name.Name == name {
            return fn
        }
    }
    return nil
}

func parseSingleDecl(fset *token.FileSet, src string) (ast.Decl, error) {
    file, err := parser.ParseFile(fset, "generated.go", "package p\n"+src, 0)
    if err != nil {
        return nil, err
    }
    if len(file.Decls) != 1 {
        return nil, fmt.Errorf("expected exactly one declaration")
    }
    return file.Decls[0], nil
}

```

```

}

func addImport(file *ast.File, path string) {
    spec := &ast.ImportSpec{Path: &ast.BasicLit{Kind: token.STRING, Value:
`"` + path + `"}}
    for _, decl := range file.Decls {
        if gen, ok := decl.(*ast.GenDecl); ok && gen.Tok == token.IMPORT {
            gen.Specs = append(gen.Specs, spec)
            if !gen.Lparen.IsValid() && len(gen.Specs) > 1 {
                gen.Lparen = gen.Pos()
                gen.Rparen = gen.End()
            }
            file.Imports = append(file.Imports, spec)
            return
        }
    }
    gen := &ast.GenDecl{Tok: token.IMPORT, Specs: []ast.Spec{spec}}
    file.Decls = append([]ast.Decl{gen}, file.Decls...)
    file.Imports = append(file.Imports, spec)
}

// internal/transpiler/enums.go
package transpiler

import (
    "fmt"
    "strings"
    "unicode"
    "unicode/utf8"
)

type enumDef struct {
    name          string
    typeParams    string
    paramNames    []string
    variants      []enumVariant
}

```

```

type enumVariant struct {
    name    string
    fields []string
}

func (d *enumDef) typeArgs() string {
    if len(d.paramNames) == 0 {
        return ""
    }
    return "[" + strings.Join(d.paramNames, ", ") + "]"
}

func (d *enumDef) variantStruct(variant string) string {
    return lowerFirst(d.name) + "_" + variant
}

func (d *enumDef) variant(name string) *enumVariant {
    for i := range d.variants {
        if d.variants[i].name == name {
            return &d.variants[i]
        }
    }
    return nil
}

func (d *enumDef) hasFieldVariants() bool {
    for _, v := range d.variants {
        if len(v.fields) > 0 {
            return true
        }
    }
    return false
}

func callbackName(variant string) string {
    return lowerFirst(variant) + "Func"
}

```

```

func lowerFirst(s string) string {
    r, size := utf8.DecodeRuneInString(s)
    return string(unicode.ToLower(r)) + s[size:]
}

func (d *enumDef) generate() string {
    var b strings.Builder
    tp, ta := d.typeParams, d.typeArgs()

    fmt.Fprintf(&b, "type %s%s struct {\n\tinner any\n}\n\n", d.name, tp)

    for _, v := range d.variants {
        vt := d.variantStruct(v.name)
        fmt.Fprintf(&b, "type %s%s struct {\n", vt, tp)
        for fi, ft := range v.fields {
            fmt.Fprintf(&b, "\t%f%d %s\n", fi, ft)
        }
        fmt.Fprintf(&b, "}\n\n")

        var params, inits []string
        for fi, ft := range v.fields {
            params = append(params, fmt.Sprintf("v%d %s", fi, ft))
            inits = append(inits, fmt.Sprintf("f%d: v%d", fi, fi))
        }
        fmt.Fprintf(&b, "func %s%s(%s) %s%s\n\n",
            v.name, tp, strings.Join(params, ", "), d.name, ta,
            d.name, ta, vt, ta, strings.Join(inits, ", "))
    }

    var callbacks []string
    for _, v := range d.variants {
        callbacks = append(callbacks, fmt.Sprintf("%s func(%s)",
            callbackName(v.name), strings.Join(v.fields, ", ")))
    }
    fmt.Fprintf(&b, "func (self %s%s) Match(%s) {\n", d.name, ta,
        strings.Join(callbacks, ", "))
    if d.hasFieldVariants() {

```

```

        b.WriteString("\tswitch v := self.inner.(type) {\n")
    } else {
        b.WriteString("\tswitch self.inner.(type) {\n")
    }
    for _, v := range d.variants {
        fmt.Fprintf(&b, "\tcase %s%:\n\t\t%s(", d.variantStruct(v.name), ta,
callbackName(v.name))
        var args []string
        for fi := range v.fields {
            args = append(args, fmt.Sprintf("v.f%d", fi))
        }
        b.WriteString(strings.Join(args, ", "))
        b.WriteString("\n")
    }
    b.WriteString("\t}\n}\n\n")

    for _, v := range d.variants {
        vt := d.variantStruct(v.name)
        fmt.Fprintf(&b, "func (self %s%) Is%s() bool {\n\t_, ok :=
self.inner.(%s%)\n\treturn ok\n}\n\n",
            d.name, ta, v.name, vt, ta)
        if len(v.fields) == 0 {
            fmt.Fprintf(&b, "func (self %s%) %s() bool {\n\treturn
self.Is%s()\n}\n\n",
                d.name, ta, v.name, v.name)
            continue
        }
        results := []string{"ok bool"}
        rets := []string{"true"}
        for fi, ft := range v.fields {
            results = append(results, fmt.Sprintf("f%d %s", fi, ft))
            rets = append(rets, fmt.Sprintf("v.f%d", fi))
        }
        fmt.Fprintf(&b, "func (self %s%) %s() (%s) {\n\tv, isSet :=
self.inner.(%s%)\n\tif !isSet {\n\t\treturn\n\t}\n\treturn %s\n}\n\n",
            d.name, ta, v.name, strings.Join(results, ", "), vt, ta,
strings.Join(rets, ", "))
    }
}

```

```

    return b.String()
}

// internal/transpiler/normalize.go
package transpiler

import (
    "fmt"
    "go/scanner"
    "go/token"
    "gonext/internal/frontend/ast"
    "gonext/internal/frontend/parser"
    "path"
    "sort"
    "strconv"
    "strings"
)

const markerPrefix = "__gnx_"

const (
    markerNamed = markerPrefix + "named"
    markerMatch = markerPrefix + "match"
    markerGuard = markerPrefix + "guard"
)

type normalized struct {
    src          string
    enums        map[string]*enumDef
    extensions   map[string]bool
    extImports   map[string]bool
    defaults     map[string]*defaultsDef
    defOrder     []string
}

type defaultsDef struct {
    name string

```

```

required int
params []defaultParam
}

type defaultParam struct {
    name      string
    typeSrc   string
    defaultSrc string
}

func normalize(src string) (*normalized, error) {
    n := &normalized{
        enums:      map[string]*enumDef{},
        extensions:  map[string]bool{},
        extImports:  map[string]bool{},
        defaults:   map[string]*defaultsDef{},
    }
    steps := []func(string) (string, error){
        n.rewriteEnums,
        n.rewriteExtensions,
        n.rewriteDefaultParams,
        n.rewriteMatches,
        n.rewriteNamedArgs,
    }
    var err error
    for _, step := range steps {
        if src, err = step(src); err != nil {
            return nil, err
        }
    }
    n.src = src
    return n, nil
}

type tok struct {
    off int
    end int
    tk  token.Token
}

```

```

    lit string
}

func scanAll(src string) ([]tok, error) {
    fset := token.NewFileSet()
    file := fset.AddFile("input.gn", fset.Base(), len(src))
    var errs scanner.ErrorList
    var s scanner.Scanner
    s.Init(file, []byte(src), func(pos token.Position, msg string) {
        errs.Add(pos, msg)
    }, scanner.ScanComments)

    var out []tok
    for {
        pos, tk, lit := s.Scan()
        if tk == token.EOF {
            break
        }
        if tk == token.COMMENT {
            continue
        }
        off := file.Offset(pos)
        out = append(out, tok{off: off, end: off + tokenLength(tk, lit), tk:
tk, lit: lit})
    }
    if len(errs) > 0 {
        return nil, errs.Err()
    }
    return out, nil
}

func tokenLength(tk token.Token, lit string) int {
    switch {
    case tk == token.SEMICOLON && lit == "\n":
        return 0
    case lit != "":
        return len(lit)
    default:

```

```

        return len(tk.String())
    }
}

type edit struct {
    start, end int
    text      string
}

func applyEdits(src string, edits []edit) string {
    sort.Slice(edits, func(i, j int) bool { return edits[i].start <
edits[j].start })
    var out strings.Builder
    last := 0
    for _, e := range edits {
        out.WriteString(src[last:e.start])
        out.WriteString(e.text)
        last = e.end
    }
    out.WriteString(src[last:])
    return out.String()
}

func matchForward(toks []tok, i int, open, close token.Token) (int, error) {
    depth := 0
    for j := i; j < len(toks); j++ {
        switch toks[j].tk {
        case open:
            depth++
        case close:
            depth--
            if depth == 0 {
                return j, nil
            }
        }
    }
    return 0, fmt.Errorf("unbalanced %q starting at offset %d", open.String(),
toks[i].off)
}

```

```

}

func matchBackward(toks []tok, i int, open, close token.Token) (int, error) {
    depth := 0
    for j := i; j >= 0; j-- {
        switch toks[j].tk {
            case close:
                depth++
            case open:
                depth--
                if depth == 0 {
                    return j, nil
                }
        }
    }
    return 0, fmt.Errorf("unbalanced %q ending at offset %d", close.String(),
toks[i].off)
}

type depthTracker struct{ paren, brack, brace int }

func (d *depthTracker) update(tk token.Token) {
    switch tk {
        case token.LPAREN:
            d.paren++
        case token.RPAREN:
            d.paren--
        case token.LBRACK:
            d.brack++
        case token.RBRACK:
            d.brack--
        case token.LBRACE:
            d.brace++
        case token.RBRACE:
            d.brace--
    }
}
}

```

```
func (d *depthTracker) zero() bool { return d.paren == 0 && d.brack == 0 &&
d.brace == 0 }
```

```
func splitTokens(toks []tok, sep token.Token) [][]int {
    var parts [][]int
    var d depthTracker
    start := 0
    for i, t := range toks {
        if t.tk == sep && d.zero() {
            parts = append(parts, [2]int{start, i})
            start = i + 1
            continue
        }
        d.update(t.tk)
    }
    parts = append(parts, [2]int{start, len(toks)})
    return parts
}
```

```
func isDeclBoundary(toks []tok, i int) bool {
    if i == 0 {
        return true
    }
    switch toks[i-1].tk {
    case token.SEMICOLON, token.RBRACE, token.LBRACE, token.COLON:
        return true
    }
    return false
}
```

```
func (n *normalized) rewriteEnums(src string) (string, error) {
    toks, err := scanAll(src)
    if err != nil {
        return "", err
    }
    var edits []edit
    for i := 0; i < len(toks); i++ {
        if toks[i].tk != token.TYPE {
```

```

        continue
    }
    j := i + 1
    if j >= len(toks) || toks[j].tk != token.IDENT {
        continue
    }
    name := toks[j].lit
    k := j + 1
    typeParams := ""
    if k < len(toks) && toks[k].tk == token.LBRACK {
        closeIdx, err := matchForward(toks, k, token.LBRACK, token.RBRACK)
        if err != nil {
            return "", err
        }
        typeParams = src[toks[k].off:toks[closeIdx].end]
        k = closeIdx + 1
    }
    if k >= len(toks) || toks[k].tk != token.IDENT || toks[k].lit !=
"enum" {
        continue
    }
    braceIdx := k + 1
    if braceIdx >= len(toks) || toks[braceIdx].tk != token.LBRACE {
        return "", fmt.Errorf("enum %s: missing body", name)
    }
    closeIdx, err := matchForward(toks, braceIdx, token.LBRACE,
token.RBRACE)
    if err != nil {
        return "", fmt.Errorf("enum %s: %w", name, err)
    }
    def, err := parseEnumDef(src, toks[braceIdx+1:closeIdx], name,
typeParams)
    if err != nil {
        return "", err
    }
    n.enums[name] = def
    edits = append(edits, edit{toks[i].off, toks[closeIdx].end,
def.generate()})

```

```

        i = closeIdx
    }
    return applyEdits(src, edits), nil
}

func parseEnumDef(src string, body []tok, name, typeParams string) (*enumDef,
error) {
    paramNames, err := typeParamNames(typeParams)
    if err != nil {
        return nil, fmt.Errorf("enum %s: %w", name, err)
    }
    def := &enumDef{name: name, typeParams: typeParams, paramNames:
paramNames}

    for i := 0; i < len(body); i++ {
        if body[i].tk == token.SEMICOLON {
            continue
        }
        if body[i].tk != token.IDENT {
            return nil, fmt.Errorf("enum %s: unexpected %q in body", name,
body[i].tk)
        }
        variant := enumVariant{name: body[i].lit}
        if i+1 < len(body) && body[i+1].tk == token.LPAREN {
            closeIdx, err := matchForward(body, i+1, token.LPAREN,
token.RPAREN)
            if err != nil {
                return nil, fmt.Errorf("enum %s: %w", name, err)
            }
            fields := body[i+2 : closeIdx]
            for _, part := range splitTokens(fields, token.COMMA) {
                if part[0] == part[1] {
                    continue
                }
                fieldSrc :=
strings.TrimSpace(src[fields[part[0]].off:fields[part[1]-1].end])
                if _, err := parser.ParseExpr(fieldSrc); err != nil {
                    return nil, fmt.Errorf("enum %s, variant %s: invalid

```

```

field type %q", name, variant.name, fieldSrc)
    }
    variant.fields = append(variant.fields, fieldSrc)
}
i = closeIdx
}
def.variants = append(def.variants, variant)
}
if len(def.variants) == 0 {
    return nil, fmt.Errorf("enum %s: must declare at least one variant",
name)
}
return def, nil
}

func typeParamNames(typeParams string) ([]string, error) {
    if typeParams == "" {
        return nil, nil
    }
    file, err := parser.ParseFile(token.NewFileSet(), "probe.go", "package
p\ntype probe"+typeParams+" int", 0)
    if err != nil {
        return nil, fmt.Errorf("invalid type parameters %q", typeParams)
    }
    gen, ok := file.Decls[0].(*ast.GenDecl)
    if !ok || len(gen.Specs) == 0 {
        return nil, fmt.Errorf("invalid type parameters %q", typeParams)
    }
    spec, ok := gen.Specs[0].(*ast.TypeSpec)
    if !ok || spec.TypeParams == nil {
        return nil, fmt.Errorf("invalid type parameters %q", typeParams)
    }
    var names []string
    for _, field := range spec.TypeParams.List {
        for _, name := range field.Names {
            names = append(names, name.Name)
        }
    }
}

```

```

    return names, nil
}

func (n *normalized) rewriteExtensions(src string) (string, error) {
    toks, err := scanAll(src)
    if err != nil {
        return "", err
    }
    var edits []edit
    for i := range toks {
        t := toks[i]
        if t.tk == token.IDENT && t.lit == "extension" && i+1 < len(toks) &&
toks[i+1].tk == token.FUNC && isDeclBoundary(toks, i) {
            j := i + 2
            if j >= len(toks) || toks[j].tk != token.IDENT {
                return "", fmt.Errorf("extension functions must be plain
named functions (offset %d)", t.off)
            }
            n.extensions[toks[j].lit] = true
            edits = append(edits, edit{t.off, toks[i+1].off, ""})
            continue
        }
        if t.tk == token.IMPORT && i+1 < len(toks) && toks[i+1].tk ==
token.IDENT && toks[i+1].lit == "extension" {
            edits = append(edits, edit{toks[i+1].off, toks[i+2].off, ""})
            j := i + 2
            switch toks[j].tk {
            case token.STRING:
                alias, err := importAlias("", toks[j].lit)
                if err != nil {
                    return "", err
                }
                n.extImports[alias] = true
            case token.IDENT:
                if j+1 >= len(toks) || toks[j+1].tk != token.STRING {
                    return "", fmt.Errorf("invalid import extension
declaration at offset %d", t.off)
                }
            }
        }
    }
}

```

```

        n.extImports[toks[j].lit] = true
    case token.LPAREN:
        closeIdx, err := matchForward(toks, j, token.LPAREN,
token.RPAREN)
        if err != nil {
            return "", err
        }
        for k := j + 1; k < closeIdx; k++ {
            switch toks[k].tk {
            case token.SEMICOLON:
            case token.IDENT:
                if k+1 < closeIdx && toks[k+1].tk == token.STRING {
                    n.extImports[toks[k].lit] = true
                    k++
                }
            case token.STRING:
                alias, err := importAlias("", toks[k].lit)
                if err != nil {
                    return "", err
                }
                n.extImports[alias] = true
            default:
                return "", fmt.Errorf("unsupported import extension
form at offset %d", toks[k].off)
            }
        }
    default:
        return "", fmt.Errorf("invalid import extension declaration
at offset %d", t.off)
    }
}
return applyEdits(src, edits), nil
}

func importAlias(alias, quotedPath string) (string, error) {
    if alias != "" {
        return alias, nil
    }
}

```

```

}
p, err := strconv.Unquote(quotedPath)
if err != nil {
    return "", fmt.Errorf("invalid import path %s", quotedPath)
}
return path.Base(p), nil
}

func (n *normalized) rewriteDefaultParams(src string) (string, error) {
    toks, err := scanAll(src)
    if err != nil {
        return "", err
    }
    var edits []edit
    for i := 0; i < len(toks); i++ {
        if toks[i].tk != token.FUNC {
            continue
        }
        topLevel := isDeclBoundary(toks, i)
        j := i + 1
        hasRecv := false
        if j < len(toks) && toks[j].tk == token.LPAREN {
            closeIdx, err := matchForward(toks, j, token.LPAREN, token.RPAREN)
            if err != nil {
                return "", err
            }
            hasRecv = true
            j = closeIdx + 1
        }
        if j >= len(toks) || toks[j].tk != token.IDENT {
            continue
        }
        name := toks[j].lit
        j++
        hasTypeParams := false
        if j < len(toks) && toks[j].tk == token.LBRACK {
            closeIdx, err := matchForward(toks, j, token.LBRACK, token.RBRACK)
            if err != nil {

```

```

        return "", err
    }
    hasTypeParams = true
    j = closeIdx + 1
}
if j >= len(toks) || toks[j].tk != token.LPAREN {
    continue
}
closeIdx, err := matchForward(toks, j, token.LPAREN, token.RPAREN)
if err != nil {
    return "", err
}
params := toks[j+1 : closeIdx]
if !hasAssign(params) {
    i = closeIdx
    continue
}
switch {
case !topLevel:
    return "", fmt.Errorf("default parameter values are only
supported on top-level functions: %s", name)
case hasRecv:
    return "", fmt.Errorf("methods cannot have default parameter
values: %s", name)
case hasTypeParams:
    return "", fmt.Errorf("generic functions cannot have default
parameter values: %s", name)
}
def, stripped, err := parseDefaultParams(src, params, name)
if err != nil {
    return "", err
}
n.defaults[name] = def
n.defOrder = append(n.defOrder, name)
edits = append(edits, edit{toks[j].end, toks[closeIdx].off, stripped})
i = closeIdx
}
return applyEdits(src, edits), nil

```

```

}

func hasAssign(toks []tok) bool {
    var d depthTracker
    for _, t := range toks {
        if t.tk == token.ASSIGN && d.zero() {
            return true
        }
        d.update(t.tk)
    }
    return false
}

func parseDefaultParams(src string, params []tok, fnName string)
(*defaultsDef, string, error) {
    def := &defaultsDef{name: fnName, required: -1}
    var plain []string
    for _, part := range splitTokens(params, token.COMMA) {
        toks := params[part[0]:part[1]]
        if len(toks) == 0 {
            continue
        }
        if len(toks) < 2 || toks[0].tk != token.IDENT {
            return nil, "", fmt.Errorf("function %s: every parameter must be
declared as \"name type\" when default values are used", fnName)
        }
        p := defaultParam{name: toks[0].lit}
        rest := toks[1:]
        assignIdx := -1
        var d depthTracker
        for k, t := range rest {
            if t.tk == token.ASSIGN && d.zero() {
                assignIdx = k
                break
            }
            d.update(t.tk)
        }
        if assignIdx < 0 {

```

```

        p.typeSrc = strings.TrimSpace(src[rest[0].off:rest[len(rest)-
1].end])
    } else {
        if assignIdx == 0 || assignIdx == len(rest)-1 {
            return nil, "", fmt.Errorf("function %s: invalid default
value for parameter %s", fnName, p.name)
        }
        p.typeSrc = strings.TrimSpace(src[rest[0].off:rest[assignIdx-
1].end])
        p.defaultSrc =
strings.TrimSpace(src[rest[assignIdx+1].off:rest[len(rest)-1].end])
    }
    if p.defaultSrc == "" && def.required >= 0 {
        return nil, "", fmt.Errorf("function %s: required parameter %s
follows a parameter with a default value", fnName, p.name)
    }
    if p.defaultSrc != "" && def.required < 0 {
        def.required = len(def.params)
    }
    def.params = append(def.params, p)
    plain = append(plain, p.name+" "+p.typeSrc)
}
if def.required < 0 {
    def.required = len(def.params)
}
return def, strings.Join(plain, ", "), nil
}

func (n *normalized) rewriteMatches(src string) (string, error) {
    toks, err := scanAll(src)
    if err != nil {
        return "", err
    }
    var edits []edit

    for i := range toks {
        t := toks[i]
        if t.tk != token.IDENT || t.lit != "match" || !isDeclBoundary(toks, i)

```

```

{
    continue
}
braceIdx := -1
var d depthTracker
scanExpr:
for j := i + 1; j < len(toks); j++ {
    switch {
    case toks[j].tk == token.LBRACE && d.zero():
        braceIdx = j
        break scanExpr
    case toks[j].tk == token.SEMICOLON && d.zero():
        break scanExpr
    }
    d.update(toks[j].tk)
}
if braceIdx < 0 || braceIdx == i+1 {
    continue
}
first := braceIdx + 1
for first < len(toks) && toks[first].tk == token.SEMICOLON {
    first++
}
if first >= len(toks) || (toks[first].tk != token.CASE &&
toks[first].tk != token.DEFAULT) {
    continue
}
edits = append(edits,
    edit{t.off, t.end, "switch " + markerMatch + "("},
    edit{toks[braceIdx].off, toks[braceIdx].off, ") "})
}

for i := range toks {
    if toks[i].tk != token.CASE {
        continue
    }
}
colonIdx, ifIdx := -1, -1
var d depthTracker

```

```

scanHeader:
    for j := i + 1; j < len(toks); j++ {
        switch {
        case toks[j].tk == token.COLON && d.zero():
            colonIdx = j
            break scanHeader
        case toks[j].tk == token.IF && d.zero():
            ifIdx = j
        case (toks[j].tk == token.SEMICOLON || toks[j].tk == token.LBRACE)
&& d.zero():
            break scanHeader
        }
        d.update(toks[j].tk)
    }
    if colonIdx < 0 || ifIdx < 0 {
        continue
    }
    patterns := strings.TrimSpace(src[toks[i].end:toks[ifIdx].off])
    cond := strings.TrimSpace(src[toks[ifIdx].end:toks[colonIdx].off])
    if patterns == "" || cond == "" {
        return "", fmt.Errorf("invalid match guard at offset %d",
toks[i].off)
    }
    edits = append(edits, edit{toks[i].end, toks[colonIdx].off,
        fmt.Sprintf(" %s(%s, %s)", markerGuard, patterns, cond)})
}

return applyEdits(src, edits), nil
}

func (n *normalized) rewriteNamedArgs(src string) (string, error) {
    toks, err := scanAll(src)
    if err != nil {
        return "", err
    }
    var edits []edit
    var stack []token.Token
    for i, t := range toks {

```

```

switch t.tk {
case token.LPAREN, token.LBRACK, token.LBRACE:
    stack = append(stack, t.tk)
    continue
case token.RPAREN, token.RBRACK, token.RBRACE:
    if len(stack) > 0 {
        stack = stack[:len(stack)-1]
    }
    continue
case token.ASSIGN:
default:
    continue
}
if len(stack) == 0 || stack[len(stack)-1] != token.LPAREN {
    continue
}
if i < 2 || toks[i-1].tk != token.IDENT {
    continue
}
switch toks[i-2].tk {
case token.LPAREN, token.COMMA:
default:
    continue
}
if i+1 >= len(toks) {
    continue
}
endIdx := -1
var d depthTracker
scanValue:
for j := i + 1; j < len(toks); j++ {
    switch {
    case (toks[j].tk == token.COMMA || toks[j].tk == token.RPAREN) &&
d.zero():
        endIdx = j - 1
        break scanValue
    }
    d.update(toks[j].tk)
}

```

```

    }
    if endIdx < i+1 {
        return "", fmt.Errorf("invalid named argument %q at offset %d",
toks[i-1].lit, toks[i-1].off)
    }
    edits = append(edits,
        edit{toks[i-1].off, toks[i+1].off, markerNamed + "(" +
strconv.Quote(toks[i-1].lit) + ", ","},
        edit{toks[endIdx].end, toks[endIdx].end, ")"})
    }
    return applyEdits(src, edits), nil
}

// internal/transpiler/rewrite.go
package transpiler

import (
    "fmt"
    "go/token"
    "gonext/internal/frontend/ast"
    "gonext/internal/frontend/types"
    "strings"
)

type rewriter struct {
    st      *state
    pkg     *types.Package
    info    *types.Info
    changed bool
    errs    []error
    pending []string
    results []*types.Tuple
}

func (r *rewriter) errorf(pos token.Pos, format string, args ...any) {
    prefix := ""
    if pos.IsValid() {
        prefix = r.st.fset.Position(pos).String() + ": "
    }

```

```

    }
    r.errs = append(r.errs, fmt.Errorf("%s%s", prefix, fmt.Sprintf(format,
args...)))
}

func (r *rewriter) defer_(pos token.Pos, format string, args ...any) {
    prefix := ""
    if pos.IsValid() {
        prefix = r.st.fset.Position(pos).String() + ": "
    }
    r.pending = append(r.pending, fmt.Sprintf("%s%s", prefix,
fmt.Sprintf(format, args...)))
}

func (r *rewriter) run() {
    for _, decl := range r.st.file.Decls {
        switch d := decl.(type) {
        case *ast.FuncDecl:
            if d.Body == nil {
                continue
            }
            r.pushResults(r.funcDeclResults(d))
            r.walkStmtList(d.Body.List)
            r.popResults()
        case *ast.GenDecl:
            for _, spec := range d.Specs {
                if vs, ok := spec.(*ast.ValueSpec); ok {
                    r.handleValueSpec(vs)
                }
            }
        }
    }
}

func (r *rewriter) funcDeclResults(d *ast.FuncDecl) *types.Tuple {
    if obj, ok := r.info.Defs[d.Name].(*types.Func); ok {
        return obj.Type().(*types.Signature).Results()
    }
}

```

```

    return nil
}

func (r *rewriter) pushResults(t *types.Tuple) { r.results = append(r.results,
t) }
func (r *rewriter) popResults() { r.results =
r.results[:len(r.results)-1] }

func (r *rewriter) currentResults() *types.Tuple {
    if len(r.results) == 0 {
        return nil
    }
    return r.results[len(r.results)-1]
}

func (r *rewriter) walkStmtList(list []ast.Stmt) {
    for i, s := range list {
        list[i] = r.walkStmt(s)
    }
}

func (r *rewriter) walkStmt(s ast.Stmt) ast.Stmt {
    switch v := s.(type) {
    case nil:
        return s
    case *ast.BlockStmt:
        r.walkStmtList(v.List)
    case *ast.ExprStmt:
        v.X = r.walkExpr(v.X)
    case *ast.AssignStmt:
        if v.Tok == token.ASSIGN {
            for i := range v.Rhs {
                if i < len(v.Lhs) {
                    if target := r.info.TypeOf(v.Lhs[i]); validType(target) {
                        r.instantiateCtor(target, v.Rhs[i])
                    }
                }
            }
        }
    }
}

```

```

}
for i := range v.Lhs {
    v.Lhs[i] = r.walkExpr(v.Lhs[i])
}
for i := range v.Rhs {
    v.Rhs[i] = r.walkExpr(v.Rhs[i])
}
case *ast.DeclStmt:
    if gen, ok := v.Decl.(*ast.GenDecl); ok {
        for _, spec := range gen.Specs {
            if vs, ok := spec.(*ast.ValueSpec); ok {
                r.handleValueSpec(vs)
            }
        }
    }
case *ast.ReturnStmt:
    if results := r.currentResults(); results != nil && results.Len() ==
len(v.Results) {
        for i := range v.Results {
            r.instantiateCtor(results.At(i).Type(), v.Results[i])
        }
    }
    for i := range v.Results {
        v.Results[i] = r.walkExpr(v.Results[i])
    }
case *ast.IfStmt:
    v.Init = r.walkStmt(v.Init)
    v.Cond = r.walkExpr(v.Cond)
    r.walkStmtList(v.Body.List)
    v.Else = r.walkStmt(v.Else)
case *ast.ForStmt:
    v.Init = r.walkStmt(v.Init)
    v.Cond = r.walkExpr(v.Cond)
    v.Post = r.walkStmt(v.Post)
    r.walkStmtList(v.Body.List)
case *ast.RangeStmt:
    v.X = r.walkExpr(v.X)
    r.walkStmtList(v.Body.List)

```

```

case *ast.SwitchStmt:
    if isMatchMarker(v.Tag) {
        if stmt, ok := r.buildMatch(v); ok {
            return r.walkStmt(stmt)
        }
        return v
    }
    v.Init = r.walkStmt(v.Init)
    v.Tag = r.walkExpr(v.Tag)
    for _, clause := range v.Body.List {
        if cc, ok := clause.(*ast.CaseClause); ok {
            for i := range cc.List {
                cc.List[i] = r.walkExpr(cc.List[i])
            }
            r.walkStmtList(cc.Body)
        }
    }
case *ast.TypeSwitchStmt:
    v.Init = r.walkStmt(v.Init)
    v.Assign = r.walkStmt(v.Assign)
    for _, clause := range v.Body.List {
        if cc, ok := clause.(*ast.CaseClause); ok {
            r.walkStmtList(cc.Body)
        }
    }
case *ast.SelectStmt:
    for _, clause := range v.Body.List {
        if cc, ok := clause.(*ast.CommClause); ok {
            cc.Comm = r.walkStmt(cc.Comm)
            r.walkStmtList(cc.Body)
        }
    }
case *ast.LabeledStmt:
    v.Stmt = r.walkStmt(v.Stmt)
case *ast.GoStmt:
    v.Call = r.walkExpr(v.Call).(*ast.CallExpr)
case *ast.DeferStmt:
    v.Call = r.walkExpr(v.Call).(*ast.CallExpr)

```

```

case *ast.SendStmt:
    v.Chan = r.walkExpr(v.Chan)
    v.Value = r.walkExpr(v.Value)
case *ast.IncDecStmt:
    v.X = r.walkExpr(v.X)
}
return s
}

func (r *rewriter) handleValueSpec(vs *ast.ValueSpec) {
    for i := range vs.Values {
        var target types.Type
        if vs.Type != nil {
            target = r.info.TypeOf(vs.Type)
        } else if i < len(vs.Names) {
            target = r.info.TypeOf(vs.Names[i])
        }
        if validType(target) {
            r.instantiateCtor(target, vs.Values[i])
        }
        vs.Values[i] = r.walkExpr(vs.Values[i])
    }
}

func (r *rewriter) walkExpr(e ast.Expr) ast.Expr {
    switch v := e.(type) {
    case nil:
        return e
    case *ast.CallExpr:
        switch fun := v.Fun.(type) {
        case *ast.SelectorExpr:
            fun.X = r.walkExpr(fun.X)
        case *ast.Ident:
        default:
            v.Fun = r.walkExpr(v.Fun)
        }
        for i := range v.Args {
            v.Args[i] = r.walkExpr(v.Args[i])
        }
    }
}

```

```

    }
    return r.rewriteCall(v)
case *ast.FuncLit:
    var results *types.Tuple
    if sig, ok := r.info.TypeOf(v).(*types.Signature); ok {
        results = sig.Results()
    }
    r.pushResults(results)
    r.walkStmtList(v.Body.List)
    r.popResults()
case *ast.ShortFuncLit:
    var results *types.Tuple
    if sig, ok := r.info.TypeOf(v).(*types.Signature); ok {
        results = sig.Results()
    }
    r.pushResults(results)
    r.walkStmtList(v.Body.List)
    r.popResults()
    return r.convertShortFuncLit(v)
case *ast.ParenExpr:
    v.X = r.walkExpr(v.X)
case *ast.SelectorExpr:
    v.X = r.walkExpr(v.X)
case *ast.StarExpr:
    v.X = r.walkExpr(v.X)
case *ast.UnaryExpr:
    v.X = r.walkExpr(v.X)
case *ast.BinaryExpr:
    v.X = r.walkExpr(v.X)
    v.Y = r.walkExpr(v.Y)
case *ast.KeyValueExpr:
    v.Key = r.walkExpr(v.Key)
    v.Value = r.walkExpr(v.Value)
case *ast.CompositeLit:
    for i := range v.Elts {
        v.Elts[i] = r.walkExpr(v.Elts[i])
    }
case *ast.IndexExpr:

```

```

    v.X = r.walkExpr(v.X)
    v.Index = r.walkExpr(v.Index)
    case *ast.IndexListExpr:
        v.X = r.walkExpr(v.X)
        for i := range v.Indices {
            v.Indices[i] = r.walkExpr(v.Indices[i])
        }
    case *ast.SliceExpr:
        v.X = r.walkExpr(v.X)
        v.Low = r.walkExpr(v.Low)
        v.High = r.walkExpr(v.High)
        v.Max = r.walkExpr(v.Max)
    case *ast.TypeAssertExpr:
        v.X = r.walkExpr(v.X)
    }
    return e
}

func isMarkerCall(e ast.Expr, name string) (*ast.CallExpr, bool) {
    call, ok := e.(*ast.CallExpr)
    if !ok {
        return nil, false
    }
    id, ok := call.Fun.(*ast.Ident)
    if !ok || id.Name != name {
        return nil, false
    }
    return call, true
}

func isMatchMarker(tag ast.Expr) bool {
    _, ok := isMarkerCall(tag, markerMatch)
    return ok
}

func (r *rewriter) rewriteCall(call *ast.CallExpr) ast.Expr {
    if id, ok := call.Fun.(*ast.Ident); ok && strings.HasPrefix(id.Name,
markerPrefix) {

```

```

    return call
}

sig, def := r.calleeSignature(call)

if sel, ok := call.Fun.(*ast.SelectorExpr); ok && sig == nil {
    if extSig, handled := r.rewriteExtensionCall(call, sel); handled {
        sig = extSig
    }
}

needsRemap := hasNamedArgs(call.Args) || (def != nil && len(call.Args) <
len(def.params) && !hasEllipsis(call))
if needsRemap {
    if sig == nil {
        r.defer_(call.Pos(), "cannot resolve the signature of the called
function yet")
        return call
    }
    if !r.remapArgs(call, sig, def) {
        return call
    }
}

return call
}

func hasEllipsis(call *ast.CallExpr) bool { return call.Ellipsis.IsValid() }

func hasNamedArgs(args []ast.Expr) bool {
    for _, a := range args {
        if _, ok := isMarkerCall(a, markerNamed); ok {
            return true
        }
    }
    return false
}
}

```

```

func (r *rewriter) calleeSignature(call *ast.CallExpr) (*types.Signature,
*defaultsDef) {
    switch fun := call.Fun.(type) {
    case *ast.Ident:
        obj := r.info.Uses[fun]
        if obj == nil {
            obj = r.pkg.Scope().Lookup(fun.Name)
        }
        switch o := obj.(type) {
        case *types.Func:
            sig := o.Type().(*types.Signature)
            var def *defaultsDef
            if o.Parent() == r.pkg.Scope() {
                def = r.st.norm.defaults[fun.Name]
            }
            return sig, def
        case *types.Var:
            if sig, ok := o.Type().Underlying().(*types.Signature); ok {
                return sig, nil
            }
        }
    case *ast.SelectorExpr:
        if sel, ok := r.info.Selections[fun]; ok {
            if sig, ok := sel.Type().(*types.Signature); ok {
                return sig, nil
            }
            return nil, nil
        }
        if obj, ok := r.info.Uses[fun.Sel].(*types.Func); ok {
            return obj.Type().(*types.Signature), nil
        }
        if x, ok := fun.X.(*ast.Ident); ok {
            if imported := r.importedPackage(x.Name); imported != nil {
                if fn, ok :=
imported.Scope().Lookup(fun.Sel.Name).(*types.Func); ok {
                    return fn.Type().(*types.Signature), nil
                }
            }
        }
    }
}

```

```

    }
}
if t := r.info.TypeOf(call.Fun); validType(t) {
    if sig, ok := t.Underlying().(*types.Signature); ok {
        return sig, nil
    }
}
return nil, nil
}

func (r *rewriter) importedPackage(alias string) *types.Package {
    for _, spec := range r.st.file.Imports {
        p, err := importPathOf(spec)
        if err != nil {
            continue
        }
        var imported *types.Package
        for _, candidate := range r.pkg.Imports() {
            if candidate.Path() == p {
                imported = candidate
                break
            }
        }
        if imported == nil {
            continue
        }
        effective := imported.Name()
        if spec.Name != nil {
            effective = spec.Name.Name
        }
        if effective == alias {
            return imported
        }
    }
    return nil
}

```

```

func (r *rewriter) remapArgs(call *ast.CallExpr, sig *types.Signature, def

```

```

*defaultsDef) bool {
    if sig.Variadic() {
        r.errorf(call.Pos(), "named arguments and default parameter values
are not supported for variadic functions")
        return false
    }
    total := sig.Params().Len()
    if def != nil && len(def.params) != total {
        r.errorf(call.Pos(), "internal error: default parameter descriptor
does not match the signature of %s", def.name)
        return false
    }

    slots := make([]ast.Expr, total)
    positional := 0
    seenNamed := false
    for _, arg := range call.Args {
        marker, named := isMarkerCall(arg, markerNamed)
        if !named {
            if seenNamed {
                r.errorf(arg.Pos(), "positional arguments must come before
named arguments")
                return false
            }
            if positional >= total {
                r.errorf(arg.Pos(), "too many arguments in call")
                return false
            }
            slots[positional] = arg
            positional++
            continue
        }
        seenNamed = true
        name, err := unquote(marker.Args[0].(*ast.BasicLit).Value)
        if err != nil {
            r.errorf(arg.Pos(), "invalid named argument")
            return false
        }
    }
}

```

```

    idx := paramIndex(sig, name)
    if idx < 0 {
        r.errorf(arg.Pos(), "unknown named argument %q", name)
        return false
    }
    if slots[idx] != nil {
        r.errorf(arg.Pos(), "argument %q is specified more than once",
name)
        return false
    }
    slots[idx] = marker.Args[1]
}

var missing []int
for i, slot := range slots {
    if slot == nil {
        missing = append(missing, i)
    }
}
if len(missing) == 0 {
    call.Args = slots
    r.changed = true
    return true
}

if def == nil {
    r.errorf(call.Pos(), "missing argument %q in call",
sig.Params().At(missing[0]).Name())
    return false
}
for _, i := range missing {
    if i < def.required {
        r.errorf(call.Pos(), "missing required argument %q in call of %s",
def.params[i].name, def.name)
        return false
    }
}
}

```

```

args := append([]ast.Expr{}, slots[:def.required]...)
for i := def.required; i < total; i++ {
    if slots[i] == nil {
        typeExpr, err := parseTypeSrc(def.params[i].typeSrc)
        if err != nil {
            r.errorf(call.Pos(), "invalid parameter type %q in %s",
def.params[i].typeSrc, def.name)
            return false
        }
        args = append(args, gnNoneExpr(typeExpr))
    } else {
        args = append(args, gnSomeExpr(slots[i]))
    }
}
call.Fun = ast.NewIdent(def.name + defaultSuffix)
call.Args = args
r.changed = true
return true
}

func paramIndex(sig *types.Signature, name string) int {
    for i := 0; i < sig.Params().Len(); i++ {
        if sig.Params().At(i).Name() == name {
            return i
        }
    }
    return -1
}

type extensionCandidate struct {
    fn    *types.Func
    alias string
}

func (r *rewriter) rewriteExtensionCall(call *ast.CallExpr, sel
*ast.SelectorExpr) (*types.Signature, bool) {
    if x, ok := sel.X.(*ast.Ident); ok {
        if _, isPkg := r.info.Uses[x].(*types.PkgName); isPkg {

```

```

        return nil, false
    }
}
if !r.isExtensionName(sel.Sel.Name) {
    return nil, false
}
if _, isSelection := r.info.Selections[sel]; isSelection {
    return nil, false
}
recvType := r.info.TypeOf(sel.X)
if !validType(recvType) {
    r.defer_(call.Pos(), "receiver type of .%(...) is not known yet",
sel.Sel.Name)
    return nil, false
}
recvType = concreteType(recvType)

var matched *extensionCandidate
for _, cand := range r.extensionCandidates(sel.Sel.Name) {
    sig := cand.fn.Type().(*types.Signature)
    if sig.Params().Len() == 0 {
        continue
    }
    ok := false
    if sig.TypeParams() != nil {
        ok = unify(sig.Params().At(0).Type(), recvType, bindings{})
    } else {
        ok = types.AssignableTo(recvType, sig.Params().At(0).Type())
    }
    if !ok {
        continue
    }
    if matched != nil {
        r.errorf(call.Pos(), "ambiguous extension method %s for receiver
type %s", sel.Sel.Name, recvType)
        return nil, false
    }
    matched = &cand
}

```

```

}
if matched == nil {
    r.errorf(call.Pos(), "no extension function %s matches receiver
type %s", sel.Sel.Name, recvType)
    return nil, false
}

if matched.alias == "" {
    call.Fun = ast.NewIdent(matched.fn.Name())
} else {
    call.Fun = &ast.SelectorExpr{X: ast.NewIdent(matched.alias), Sel:
ast.NewIdent(matched.fn.Name())}
}
call.Args = append([]ast.Expr{sel.X}, call.Args...)
r.changed = true
return matched.fn.Type().(*types.Signature), true
}

func (r *rewriter) isExtensionName(name string) bool {
    if r.st.norm.extensions[name] {
        return true
    }
    for alias := range r.st.norm.extImports {
        if imported := r.importedPackage(alias); imported != nil {
            if _, ok := imported.Scope().Lookup(name).(*types.Func); ok {
                return true
            }
        }
    }
    return false
}

func (r *rewriter) extensionCandidates(name string) []extensionCandidate {
    var out []extensionCandidate
    if r.st.norm.extensions[name] {
        if fn, ok := r.pkg.Scope().Lookup(name).(*types.Func); ok {
            out = append(out, extensionCandidate{fn: fn})
        }
    }
}

```

```

}
for alias := range r.st.norm.extImports {
    imported := r.importedPackage(alias)
    if imported == nil {
        continue
    }
    if fn, ok := imported.Scope().Lookup(name).(*types.Func); ok &&
fn.Exported() {
        out = append(out, extensionCandidate{fn: fn, alias: alias})
    }
}
return out
}

func (r *rewriter) convertShortFuncLit(lit *ast.ShortFuncLit) ast.Expr {
    sig, ok := r.info.TypeOf(lit).(*types.Signature)
    if !ok {
        r.defer_(lit.Pos(), "the expected type of the short anonymous
function is not known yet")
        return lit
    }

    params := &ast.FieldList{}
    for i, name := range lit.Params {
        typeExpr, err := typeToExpr(sig.Params().At(i).Type(), r.pkg,
r.st.file)
        if err != nil {
            r.errorf(lit.Pos(), "%v", err)
            return lit
        }
        params.List = append(params.List, &ast.Field{
            Names: []*ast.Ident{ast.NewIdent(name.Name)},
            Type: typeExpr,
        })
    }
    var results *ast.FieldList
    if sig.Results().Len() > 0 {
        results = &ast.FieldList{}

```

```

    for i := 0; i < sig.Results().Len(); i++ {
        typeExpr, err := typeToExpr(sig.Results().At(i).Type(), r.pkg,
r.st.file)
        if err != nil {
            r.errorf(lit.Pos(), "%v", err)
            return lit
        }
        results.List = append(results.List, &ast.Field{Type: typeExpr})
    }
}

r.changed = true
return &ast.FuncLit{
    Type: &ast.FuncType{Params: params, Results: results},
    Body: lit.Body,
}
}

func (r *rewriter) instantiateCtor(target types.Type, e ast.Expr) {
    call, ok := e.(*ast.CallExpr)
    if !ok {
        return
    }
    fun, ok := call.Fun.(*ast.Ident)
    if !ok {
        return
    }
    def := r.st.variantToEnum[fun.Name]
    if def == nil || len(def.paramNames) == 0 {
        return
    }
    if obj := r.info.Uses[fun]; obj != nil {
        if _, isFunc := obj.(*types.Func); !isFunc {
            return
        }
    }
    named, ok := types.Unalias(target).(*types.Named)
    if !ok || named.Obj().Name() != def.name || named.Obj().Pkg() != r.pkg {

```

```

    return
}
args := named.TypeArgs()
if args.Len() != len(def.paramNames) {
    return
}
indices := make([]ast.Expr, args.Len())
for i := range indices {
    expr, err := typeToExpr(args.At(i), r.pkg, r.st.file)
    if err != nil {
        return
    }
    indices[i] = expr
}
if len(indices) == 1 {
    call.Fun = &ast.IndexExpr{X: fun, Index: indices[0]}
} else {
    call.Fun = &ast.IndexListExpr{X: fun, Indices: indices}
}
r.changed = true
}

type matchClause struct {
    bindings []string
    guard    ast.Expr
    body     []ast.Stmt
}

func (r *rewriter) buildMatch(sw *ast.SwitchStmt) (ast.Stmt, bool) {
    tag := sw.Tag.(*ast.CallExpr)
    operand := tag.Args[0]
    opType := r.info.TypeOf(operand)
    if !validType(opType) {
        r.defer_(sw.Pos(), "the type of the match operand is not known yet")
        return nil, false
    }
    opType = concreteType(opType)
    lookupType := opType

```

```

if ptr, ok := lookupType.Underlying().(*types.Pointer); ok {
    lookupType = ptr.Elem()
}
named, ok := types.Unalias(lookupType).(*types.Named)
if !ok {
    r.errorf(sw.Pos(), "match requires an enum value, got %s", opType)
    return nil, false
}
def := r.st.enums[named.Obj().Name()]
if def == nil || named.Obj().Pkg() != r.pkg {
    r.errorf(sw.Pos(), "match requires an enum value, got %s", opType)
    return nil, false
}

matchObj, _, _ := types.LookupFieldOrMethod(opType, true, r.pkg, "Match")
matchFn, ok := matchObj.(*types.Func)
if !ok {
    r.errorf(sw.Pos(), "internal error: enum %s has no generated Match
method", def.name)
    return nil, false
}
matchSig := matchFn.Type().(*types.Signature)

perVariant := map[string][]*matchClause{}
var defaultBody []ast.Stmt
hasDefault := false
for _, stmt := range sw.Body.List {
    cc, ok := stmt.(*ast.CaseClause)
    if !ok {
        r.errorf(stmt.Pos(), "unexpected statement inside match")
        return nil, false
    }
    if cc.List == nil {
        hasDefault = true
        defaultBody = cc.Body
        continue
    }
    for _, expr := range cc.List {

```

```

guard, patterns, err := parseMatchPatterns(expr)
if err != nil {
    r.errorf(expr.Pos(), "%v", err)
    return nil, false
}
for _, p := range patterns {
    variant := def.variant(p.variant)
    if variant == nil {
        r.errorf(expr.Pos(), "enum %s has no variant %s",
def.name, p.variant)
        return nil, false
    }
    if len(p.bindings) != len(variant.fields) {
        r.errorf(expr.Pos(), "variant %s of enum %s has %d fields,
but the pattern binds %d",
            p.variant, def.name, len(variant.fields),
len(p.bindings))
        return nil, false
    }
    perVariant[p.variant] = append(perVariant[p.variant],
&matchClause{
        bindings: p.bindings,
        guard:    guard,
        body:     cc.Body,
    })
}
}

var callbacks []ast.Expr
for i, variant := range def.variants {
    clauses := perVariant[variant.name]
    if len(closures) == 0 && !hasDefault {
        r.errorf(sw.Pos(), "non-exhaustive match: variant %s of enum %s
is not handled and there is no default case",
            variant.name, def.name)
        return nil, false
    }
}

```

```

        cbSig, ok :=
matchSig.Params().At(i).Type().Underlying().(*types.Signature)
    if !ok {
        r.errorf(sw.Pos(), "internal error: unexpected Match signature
for enum %s", def.name)
        return nil, false
    }
    callback, err := r.buildMatchCallback(cbSig, clauses, defaultBody)
    if err != nil {
        r.errorf(sw.Pos(), "%v", err)
        return nil, false
    }
    callbacks = append(callbacks, callback)
}

r.changed = true
return &ast.ExprStmt{X: &ast.CallExpr{
    Fun: &ast.SelectorExpr{X: operand, Sel: ast.NewIdent("Match")},
    Args: callbacks,
}}, true
}

type matchPattern struct {
    variant string
    bindings []string
}

func parseMatchPatterns(expr ast.Expr) (ast.Expr, []matchPattern, error) {
    var guard ast.Expr
    if marker, ok := isMarkerCall(expr, markerGuard); ok {
        guard = marker.Args[1]
        expr = marker.Args[0]
    }
    var patterns []matchPattern
    var flatten func(e ast.Expr) error
    flatten = func(e ast.Expr) error {
        switch v := e.(type) {
        case *ast.BinaryExpr:

```

```

    if v.Op != token.OR {
        return fmt.Errorf("invalid match pattern")
    }
    if err := flatten(v.X); err != nil {
        return err
    }
    return flatten(v.Y)
case *ast.Ident:
    patterns = append(patterns, matchPattern{variant: v.Name})
    return nil
case *ast.CallExpr:
    id, ok := v.Fun.(*ast.Ident)
    if !ok {
        return fmt.Errorf("invalid match pattern")
    }
    p := matchPattern{variant: id.Name}
    for _, arg := range v.Args {
        binding, ok := arg.(*ast.Ident)
        if !ok {
            return fmt.Errorf("match patterns may only bind plain
identifiers")
        }
        p.bindings = append(p.bindings, binding.Name)
    }
    patterns = append(patterns, p)
    return nil
case *ast.ParenExpr:
    return flatten(v.X)
default:
    return fmt.Errorf("invalid match pattern")
}
}
if err := flatten(expr); err != nil {
    return nil, nil, err
}
return guard, patterns, nil
}

```

```

func (r *rewriter) buildMatchCallback(cbSig *types.Signature, clauses
[]*matchClause, defaultBody []ast.Stmt) (ast.Expr, error) {
    fieldCount := cbSig.Params().Len()

    paramNames := make([]string, fieldCount)
    for i := range paramNames {
        paramNames[i] = "_"
    }
    for _, cl := range clauses {
        for i, binding := range cl.bindings {
            if paramNames[i] == "_" && binding != "_" {
                paramNames[i] = binding
            }
        }
    }

    params := &ast.FieldList{}
    for i := range fieldCount {
        typeExpr, err := typeToExpr(cbSig.Params().At(i).Type(), r.pkg,
r.st.file)
        if err != nil {
            return nil, err
        }
        params.List = append(params.List, &ast.Field{
            Names: []*ast.Ident{ast.NewIdent(paramNames[i])},
            Type: typeExpr,
        })
    }

    var stmts []ast.Stmt
    terminal := false
    for idx, cl := range clauses {
        var rebinds []ast.Stmt
        for i, binding := range cl.bindings {
            if binding != "_" && binding != paramNames[i] {
                rebinds = append(rebinds, &ast.AssignStmt{
                    Lhs: []ast.Expr{ast.NewIdent(binding)},
                    Tok: token.DEFINE,
                })
            }
        }
        stmts = append(stmts, cl.Body(stmts, rebinds, terminal))
    }
    return defaultBody[0], nil
}

```

```

        Rhs: []ast.Expr{ast.NewIdent(paramNames[i])},
    })
}
}
if cl.guard == nil {
    stmts = append(stmts, rebinds...)
    stmts = append(stmts, cl.body...)
    terminal = true
    break
}
followed := idx < len(clauses)-1 || len(defaultBody) > 0
guardedBody := append([]ast.Stmt{}, cl.body...)
if followed {
    guardedBody = append(guardedBody, &ast.ReturnStmt{})
}
guarded := []ast.Stmt{&ast.IfStmt{Cond: cl.guard, Body:
&ast.BlockStmt{List: guardedBody}}}
if len(rebinds) > 0 {
    guarded = []ast.Stmt{&ast.BlockStmt{List: append(rebinds,
guarded...)}}
}
stmts = append(stmts, guarded...)
}
if !terminal {
    stmts = append(stmts, defaultBody...)
}

return &ast.FuncLit{
    Type: &ast.FuncType{Params: params},
    Body: &ast.BlockStmt{List: stmts},
}, nil
}

// internal/transpiler/transpiler.go
package transpiler

import (
    "bytes"

```

```

"errors"
"fmt"
"go/token"
"os"
"os/exec"
"path/filepath"
"strings"

"gonext/internal/frontend/ast"
"gonext/internal/frontend/parser"
"gonext/internal/frontend/printer"
)

const maxPasses = 16

type state struct {
    fset      *token.FileSet
    file      *ast.File
    norm      *normalized
    enums     map[string]*enumDef
    variantToEnum map[string]*enumDef
    usesGn    bool
}

type output struct {
    code string
    usesGn bool
}

func TranspileFile(path string) (string, error) {
    out, err := transpileFile(path)
    if err != nil {
        return "", err
    }
    return out.code, nil
}

func TranspileSource(src string) (string, error) {

```

```

out, err := transpile(src, "input.gn")
if err != nil {
    return "", err
}
return out.code, nil
}

func RunFile(path string) error {
    out, err := transpileFile(path)
    if err != nil {
        return err
    }

    tmpDir, err := os.MkdirTemp("", "gonext-*")
    if err != nil {
        return err
    }
    defer os.RemoveAll(tmpDir)

    files := map[string]string{
        "go.mod": "module gonext\n\ngo 1.21\n",
        "main.go": out.code,
    }

    if out.usesGn {
        files[filepath.Join("gn", "gn.go")] = gnSource
    }

    for name, content := range files {
        target := filepath.Join(tmpDir, name)
        if err := os.MkdirAll(filepath.Dir(target), 0o755); err != nil {
            return err
        }
        if err := os.WriteFile(target, []byte(content), 0o644); err != nil {
            return err
        }
    }

    cmd := exec.Command("go", "run", ".")
    cmd.Dir = tmpDir

```

```

cmd.Stdout = os.Stdout
cmd.Stderr = os.Stderr
return cmd.Run()
}

func transpileFile(path string) (*output, error) {
src, err := os.ReadFile(path)
if err != nil {
return nil, err
}
return transpile(string(src), filepath.Base(path))
}

func transpile(src, filename string) (*output, error) {
norm, err := normalize(src)
if err != nil {
return nil, err
}

fset := token.NewFileSet()
file, err := parser.ParseFile(fset, filename, norm.src,
parser.ParseComments|parser.SkipObjectResolution)
if err != nil {
return nil, fmt.Errorf("after normalization the source does not parse
as Go: %w", err)
}

st := &state{
fset:      fset,
file:      file,
norm:      norm,
enums:     norm.enums,
variantToEnum: map[string]*enumDef{},
}

for _, def := range norm.enums {
for _, v := range def.variants {
st.variantToEnum[v.name] = def
}
}
}

```

```

}

if len(norm.defaults) > 0 {
    if err := appendDefaultWrappers(st); err != nil {
        return nil, err
    }
}

analyzer, err := newAnalyzer(fset, file.Name.Name)
if err != nil {
    return nil, err
}

for range maxPasses {
    pkg, info := analyzer.check(file)
    r := &rewriter{st: st, pkg: pkg, info: info}
    r.run()
    if len(r.errs) > 0 {
        return nil, errors.Join(r.errs...)
    }
    if !hasMarkers(file) {
        break
    }
    if !r.changed {
        msg := "could not transform the remaining GoNext constructs"
        if len(r.pending) > 0 {
            msg += ":\n " + strings.Join(r.pending, "\n ")
        }
        return nil, errors.New(msg)
    }
}

if hasMarkers(file) {
    return nil, errors.New("internal error: GoNext constructs remained
after the transformation passes")
}

var buf bytes.Buffer
cfg := printer.Config{Mode: printer.UseSpaces | printer.TabIndent,

```

```

Tabwidth: 8}
    if err := cfg.Fprint(&buf, fset, file); err != nil {
        return nil, fmt.Errorf("cannot format the generated code: %w", err)
    }
    return &output{code: buf.String(), usesGn: st.usesGn}, nil
}

func hasMarkers(file *ast.File) bool {
    found := false
    ast.Inspect(file, func(n ast.Node) bool {
        switch v := n.(type) {
        case *ast.Ident:
            if strings.HasPrefix(v.Name, markerPrefix) {
                found = true
            }
        case *ast.ShortFuncLit:
            found = true
        }
        return !found
    })
    return found
}

// internal/transpiler/unify.go
package transpiler

import (
    "gonext/internal/frontend/types"
)

type bindings map[*types.TypeParam]types.Type

func unify(pattern, actual types.Type, bind bindings) bool {
    pattern = types.Unalias(pattern)
    actual = types.Unalias(actual)

    if tp, ok := pattern.(*types.TypeParam); ok {
        if bound, ok := bind[tp]; ok {

```

```

        return types.Identical(bound, actual)
    }
    bind[tp] = actual
    return true
}

switch p := pattern.(type) {
case *types.Slice:
    a, ok := actual.(*types.Slice)
    return ok && unify(p.Elem(), a.Elem(), bind)
case *types.Array:
    a, ok := actual.(*types.Array)
    return ok && p.Len() == a.Len() && unify(p.Elem(), a.Elem(), bind)
case *types.Pointer:
    a, ok := actual.(*types.Pointer)
    return ok && unify(p.Elem(), a.Elem(), bind)
case *types.Map:
    a, ok := actual.(*types.Map)
    return ok && unify(p.Key(), a.Key(), bind) && unify(p.Elem(),
a.Elem(), bind)
case *types.Chan:
    a, ok := actual.(*types.Chan)
    return ok && p.Dir() == a.Dir() && unify(p.Elem(), a.Elem(), bind)
case *types.Signature:
    a, ok := actual.(*types.Signature)
    if !ok || p.Variadic() != a.Variadic() {
        return false
    }
    return unifyTuples(p.Params(), a.Params(), bind) &&
unifyTuples(p.Results(), a.Results(), bind)
case *types.Named:
    a, ok := actual.(*types.Named)
    if !ok || p.Obj() != a.Obj() {
        return types.Identical(pattern, actual)
    }
    pargs, aargs := p.TypeArgs(), a.TypeArgs()
    if pargs.Len() != aargs.Len() {
        return false
    }

```

```

    }
    for i := 0; i < pargs.Len(); i++ {
        if !unify(pargs.At(i), aargs.At(i), bind) {
            return false
        }
    }
    return true
default:
    return types.Identical(pattern, actual)
}
}

func unifyTuples(pattern, actual *types.Tuple, bind bindings) bool {
    if pattern.Len() != actual.Len() {
        return false
    }
    for i := 0; i < pattern.Len(); i++ {
        if !unify(pattern.At(i).Type(), actual.At(i).Type(), bind) {
            return false
        }
    }
    return true
}

```

### Зміни у коді пакетів go/\*

```

// ===== ast/ast.go =====
@@ -326,6 +326,16 @@
    Body *BlockStmt // function body
}

+ // A ShortFuncLit node represents a GoNext short function literal
+ // "(a, b) => { ... }". Parameter and result types are not part of the
+ // syntax; they are recovered from the context by the type checker.
+ ShortFuncLit struct {
+     Lparen token.Pos // position of "("
+     Params []*Ident   // parameter names; or nil
+     Arrow  token.Pos // position of "=>"
+     Body   *BlockStmt // function body

```

```

+   }
+
    // A CompositeLit node represents a composite literal.
    CompositeLit struct {
        Type      Expr      // literal type; or nil
@@ -496,7 +506,8 @@
    func (x *Ident) Pos() token.Pos    { return x.NamePos }
    func (x *Ellipsis) Pos() token.Pos { return x.Ellipsis }
    func (x *BasicLit) Pos() token.Pos { return x.ValuePos }
-   func (x *FuncLit) Pos() token.Pos { return x.Type.Pos() }
+   func (x *FuncLit) Pos() token.Pos    { return x.Type.Pos() }
+   func (x *ShortFuncLit) Pos() token.Pos { return x.Lparen }
    func (x *CompositeLit) Pos() token.Pos {
        if x.Type != nil {
            return x.Type.Pos()
@@ -544,6 +555,7 @@
            return x.ValueEnd
        }
    func (x *FuncLit) End() token.Pos    { return x.Body.End() }
+   func (x *ShortFuncLit) End() token.Pos { return x.Body.End() }
    func (x *CompositeLit) End() token.Pos { return x.Rbrace + 1 }
    func (x *ParenExpr) End() token.Pos  { return x.Rparen + 1 }
    func (x *SelectorExpr) End() token.Pos { return x.Sel.End() }
@@ -575,6 +587,7 @@
    func (*Ellipsis) exprNode()    {}
    func (*BasicLit) exprNode()    {}
    func (*FuncLit) exprNode()     {}
+   func (*ShortFuncLit) exprNode() {}
    func (*CompositeLit) exprNode() {}
    func (*ParenExpr) exprNode()   {}
    func (*SelectorExpr) exprNode() {}

// ===== ast/walk.go =====
@@ -77,6 +77,10 @@
        Walk(v, n.Type)
        Walk(v, n.Body)

+   case *ShortFuncLit:

```

```

+     walkList(v, n.Params)
+     Walk(v, n.Body)
+
+     case *CompositeLit:
+         if n.Type != nil {
+             Walk(v, n.Type)
+
+ // ===== parser/parser.go =====
@@ -28,7 +28,6 @@
+     "fmt"
+     "gonext/internal/frontend/ast"
+     "go/build/constraint"
-     "gonext/internal/frontend/scannerhooks"
+     "go/scanner"
+     "go/token"
+     "strings"
@@ -74,6 +73,11 @@
+     // nestLev is used to track and limit the recursion depth
+     // during parsing.
+     nestLev int
+
+ // ahead holds raw tokens buffered by peekRaw and replayed by next0.
+ // It gives the parser the multi-token lookahead needed to recognize
+ // GoNext short function literals "(a, b) => { ... }".
+     ahead []aheadToken
+ }
+
+ func (p *parser) init(file *token.File, src []byte, mode Mode) {
@@ -154,7 +158,7 @@
+     }
+
+     for {
-         p.pos, p.tok, p.lit = p.scanner.Scan()
+         p.pos, p.tok, p.lit = p.rawScan()
+         if p.tok == token.COMMENT {
+             if p.top && strings.HasPrefix(p.lit, "//go:build") {
+                 if x, err := constraint.Parse(p.lit); err == nil {
@@ -166,7 +170,11 @@

```

```

    }
} else {
    if p.tok == token.STRING {
-       p.stringEnd = scannerhooks.StringEnd(&p.scanner)
+       // The stock parser obtains the exact end position from the
+       // scanner through an internal backdoor. The approximation
+       // below is exact except for raw strings containing carriage
+       // returns, which the scanner strips from the literal.
+       p.stringEnd = p.pos + token.Pos(len(p.lit))
    }

    // Found a non-comment; top of file is over.
@@ -176,6 +184,107 @@
    }
}

+type aheadToken struct {
+    pos token.Pos
+    tok token.Token
+    lit string
+}
+
+// rawScan returns the next raw token, replaying tokens buffered by peekRaw.
+func (p *parser) rawScan() (token.Pos, token.Token, string) {
+    if len(p.ahead) > 0 {
+        t := p.ahead[0]
+        p.ahead = p.ahead[1:]
+        return t.pos, t.tok, t.lit
+    }
+    return p.scanner.Scan()
+}
+
+// peekRaw returns the i-th raw token (0-based) after the current one,
+// buffering the scanned tokens so that next0 replays them later.
+func (p *parser) peekRaw(i int) (token.Pos, token.Token, string) {
+    for len(p.ahead) <= i {
+        pos, tok, lit := p.scanner.Scan()
+        p.ahead = append(p.ahead, aheadToken{pos, tok, lit})

```

```

+     if tok == token.EOF {
+         break
+     }
+ }
+ if i >= len(p.ahead) {
+     i = len(p.ahead) - 1
+ }
+ t := p.ahead[i]
+ return t.pos, t.tok, t.lit
+}
+
+// shortFuncLitAhead reports whether the tokens starting at the current "("
+// form the head of a short function literal: "(" [ident {"," ident}] ")"
+// "=>"
+// with "=>" written without an intervening space.
+func (p *parser) shortFuncLitAhead() bool {
+    if p.tok != token.LPAREN {
+        return false
+    }
+    i := 0
+    next := func() (token.Token, token.Pos) {
+        for {
+            pos, tok, _ := p.peekRaw(i)
+            i++
+            if tok != token.COMMENT {
+                return tok, pos
+            }
+        }
+    }
+    tok, _ := next()
+    if tok != token.RPAREN {
+        for {
+            if tok != token.IDENT {
+                return false
+            }
+            tok, _ = next()
+            if tok == token.RPAREN {
+                break
+            }
+        }
+    }
+}

```

```

+         }
+         if tok != token.COMMA {
+             return false
+         }
+         tok, _ = next()
+     }
+ }
+ tok, assignPos := next()
+ if tok != token.ASSIGN {
+     return false
+ }
+ tok, gtrPos := next()
+ return tok == token.GTR && gtrPos == assignPos+1
+}
+
+// parseShortFuncLit parses a short function literal "(a, b) => { ... }".
+// The caller must have verified the head with shortFuncLitAhead.
+func (p *parser) parseShortFuncLit() ast.Expr {
+    if p.trace {
+        defer un(trace(p, "ShortFuncLit"))
+    }
+
+    lparen := p.pos
+    p.next() // consume "("
+    var params []*ast.Ident
+    for p.tok != token.RPAREN && p.tok != token.EOF {
+        params = append(params, p.parseIdent())
+        if p.tok != token.COMMA {
+            break
+        }
+        p.next()
+    }
+    p.expect(token.RPAREN)
+    arrow := p.pos
+    p.expect(token.ASSIGN)
+    p.expect(token.GTR)
+    p.exprLev++
+    body := p.parseBody()

```

```

+   p.exprLev--
+   return &ast.ShortFuncLit{Lparen: lparen, Params: params, Arrow: arrow,
Body: body}
+}
+
// lineFor returns the line of pos, ignoring line directive adjustments.
func (p *parser) lineFor(pos token.Pos) int {
    return p.file.PositionFor(pos, false).Line
@@ -1489,6 +1598,9 @@
    return x

    case token.LPAREN:
+   if p.shortFuncLitAhead() {
+       return p.parseShortFuncLit()
+   }
    lparen := p.pos
    p.next()
    p.exprLev++

// ===== parser/resolver.go =====
@@ -264,6 +264,12 @@
    r.walkFuncType(n.Type)
    r.walkBody(n.Body)

+   case *ast.ShortFuncLit:
+       r.openScope(n.Pos())
+       defer r.closeScope()
+       r.declare(n, nil, r.topScope, ast.Var, n.Params...)
+       r.walkBody(n.Body)
+
    case *ast.SelectorExpr:
        ast.Walk(r, n.X)
        // Note: don't try to resolve n.Sel, as we don't support qualified

// ===== printer/nodes.go =====
@@ -878,6 +878,20 @@
    p.signature(x.Type)
    p.funcBody(p.distanceFrom(x.Type.Pos(), startCol), blank, x.Body)

```

```

+ case *ast.ShortFuncLit:
+     p.setPos(x.Lparen)
+     p.print(token.LPAREN)
+     for i, param := range x.Params {
+         if i > 0 {
+             p.print(token.COMMA, blank)
+         }
+         p.expr(param)
+     }
+     p.print(token.RPAREN, blank)
+     p.setPos(x.Arrow)
+     p.print(token.ASSIGN, token.GTR)
+     p.funcBody(p.distanceFrom(x.Lparen, p.out.Column), blank, x.Body)
+
+ case *ast.ParenExpr:
+     if _, hasParens := x.X.(*ast.ParenExpr); hasParens {
+         // don't print parentheses around an already parenthesized
expression
// ===== srcimporter/srcimporter.go =====
@@ -19,7 +19,6 @@
    "path/filepath"
    "strings"
    "sync"
- _ "unsafe" // for go:linkname
)

// An Importer provides the context for importing packages from source code.
@@ -265,5 +264,6 @@
    return filepath.Join(elem...)
}

-//go:linkname setUsesCgo go/types.srcimporter_setUsesCgo
-func setUsesCgo(conf *types.Config)
+func setUsesCgo(conf *types.Config) {
+     types.SetUsesCgo_ForSrcImporter(conf)
+}

```

```

// ===== types/api.go =====
@@ -199,9 +199,11 @@
}

// Linkname for use from srcimporter.
-//go:linkname srcimporter_setUsesCgo

-func srcimporter_setUsesCgo(conf *Config) {
+// SetUsesCgo_ForSrcImporter replaces the go:linkname hand-off between the
+// stock srcimporter and go/types: inside the fork a plain exported call is
+// enough.
+func SetUsesCgo_ForSrcImporter(conf *Config) {
    conf.go115UsesCgo = true
}

// ===== types/call.go =====
@@ -389,7 +389,14 @@
    // single value (possibly a partially instantiated function), or a
multi-valued expression
    e := elist[0]
    var x operand
-   if ix := unpackIndexedExpr(e); ix != nil && check.indexExpr(&x, ix) {
+   if lit, ok := e.(*ast.ShortFuncLit); ok {
+       // GoNext: short function literals are resolved later, against
the
+       // expected parameter type (see Checker.resolveShortFuncArgs).
+       x.mode = value
+       x.expr = lit
+       x.typ = Typ[Invalid]
+       resList = []*operand{&x}
+   } else if ix := unpackIndexedExpr(e); ix != nil && check.indexExpr(&x,
ix) {
        // x is a generic function.
        targs := check.funcInst(nil, x.Pos(), &x, ix, infer)
        if targs != nil {
@@ -424,7 +431,13 @@

```

```

    targList = make([][]Type, n)
    for i, e := range elist {
        var x operand
-       if ix := unpackIndexedExpr(e); ix != nil && check.indexExpr(&x,
ix) {
+       if lit, ok := e.(*ast.ShortFuncLit); ok {
+           // GoNext: short function literals are resolved later,
against
+           // the expected parameter type (see
Checker.resolveShortFuncArgs).
+           x.mode = value
+           x.expr = lit
+           x.typ = Typ[Invalid]
+       } else if ix := unpackIndexedExpr(e); ix != nil &&
check.indexExpr(&x, ix) {
            // x is a generic function.
            targs := check.funcInst(nil, x.Pos(), &x, ix, infer)
            if targs != nil {
@@ -607,6 +620,13 @@
                // tparams holds the type parameters of the callee and generic function
arguments, if any:
                // the first n type parameters belong to the callee, followed by mi type
parameters for each
                // of the generic function arguments, where mi =
args[i].typ.(*Signature).TypeParams().Len().
+
+ // GoNext: resolve short function literal arguments against the expected
+ // parameter types before running the inference, so that the resolved
+ // literals participate in it like ordinary typed function arguments.
+ if !check.resolveShortFuncArgs(call, sigParams, tparams, targs, args) {
+     return
+ }

                // infer missing type arguments of callee and function arguments
                if len(tparams) > 0 {

// ===== types/expr.go =====
@@ -1072,6 +1072,12 @@

```

```

        goto Error
    }

+   case *ast.ShortFuncLit:
+       check.shortFuncLit(x, e, T)
+       if x.mode == invalid {
+           goto Error
+       }
+
    case *ast.CompositeLit:
        check.compositeLit(x, e, hint)
        if x.mode == invalid {

```

```

// types/shortfunclit.go
// Copyright 2026 GoNext. All rights reserved.
// This file is a GoNext extension of the vendored go/types and is not part
of
// the upstream sources.

// Type checking of GoNext short function literals "(a, b) => { ... }".
//
// A short literal carries no parameter or result types of its own: they are
// recovered from the expected function type of the context, following the
// contextual type recovery scheme of the GoNext transpiler. Two kinds of
// context are supported:
//
// - assignment-like contexts (variable initialization, assignment, return
//   statements) provide the expected type through the *target mechanism
the
//   checker already uses for those positions (see exprInternal);
//
// - call arguments receive their expected type from the (possibly generic)
//   signature of the callee in Checker.arguments. For generic callees the
//   type parameters are first partially inferred from the remaining
//   arguments; a result type that is still an unresolved type parameter is
//   recovered from the literal's body, after which the literal
participates

```

```

//      in the regular inference like any explicitly typed function argument.

package types

import (
    "gonext/internal/frontend/ast"
    . "gonext/internal/frontend/typeserrors"
)

// shortFuncLit type-checks a short function literal in an assignment-like
// context whose expected type is provided by T.
func (check *Checker) shortFuncLit(x *operand, e *ast.ShortFuncLit, T *target)
{
    if T == nil || T.sig == nil {
        check.errorf(e, CannotInferTypeArgs, "cannot infer the parameter
types of the short function literal: the context does not provide a function
type")
        x.mode = invalid
        x.typ = Typ[Invalid]
        return
    }
    sig := check.shortFuncSig(e, T.sig)
    if sig == nil {
        x.mode = invalid
        x.typ = Typ[Invalid]
        return
    }
    x.mode = value
    x.typ = sig
}

// shortFuncSig builds the signature of a short function literal from the
// expected signature, declares the parameters in a new scope, recovers the
// result type from the body when the expected result is an unresolved type
// parameter, and schedules the body for the usual deferred checking.
func (check *Checker) shortFuncSig(e *ast.ShortFuncLit, expected *Signature)
*Signature {
    if expected.Variadic() {

```

```

    check.errorf(e, CannotInferTypeArgs, "short function literals cannot
implement the variadic function type %s", expected)
    return nil
}
if expected.Params().Len() != len(e.Params) {
    check.errorf(e, WrongArgCount, "short function literal has %d
parameters, but the expected type %s has %d", len(e.Params), expected,
expected.Params().Len())
    return nil
}

scope := NewScope(check.scope, e.Pos(), e.End(), "function")
scopePos := e.Body.Pos()
params := make([]*Var, len(e.Params))
for i, name := range e.Params {
    ptype := expected.Params().At(i).Type()
    if containsTypeParam(ptype) {
        check.errorf(name, CannotInferTypeArgs, "cannot recover the type
of parameter %s: the expected function type %s is generic", name.Name,
expected)
        return nil
    }
    par := NewParam(name.Pos(), check.pkg, name.Name, ptype)
    check.declare(scope, name, par, scopePos)
    params[i] = par
}

sig := NewSignatureType(nil, nil, nil, NewTuple(params...), nil, false)
sig.scope = scope

if n := expected.Results().Len(); n > 0 {
    vars := make([]*Var, n)
    for i := range n {
        rtype := expected.Results().At(i).Type()
        if containsTypeParam(rtype) {
            if n != 1 {
                check.errorf(e, CannotInferTypeArgs, "cannot recover the
result types of the short function literal from the generic type %s",

```

```

expected)
        return nil
    }
    rtype = check.inferShortFuncResult(sig, e.Body)
    if rtype == nil {
        check.errorf(e, CannotInferTypeArgs, "cannot recover the
result type of the short function literal from its context or body")
        return nil
    }
}
vars[i] = NewVar(nopos, check.pkg, "", rtype)
}
sig.results = NewTuple(vars...)
}

// Schedule the body like an ordinary function literal (see funcLit).
if !check.conf.IgnoreFuncBodies && e.Body != nil {
    decl := check.decl
    iota := check.iota
    check.later(func() {
        check.funcBody(decl, "<short function literal>", sig, e.Body,
iota)
    }).describef(e, "short func literal")
}
return sig
}

// inferShortFuncResult recovers the result type of a short function literal
// from its body: the parameters are already declared in sig.scope, and the
// type of the first well-typed single-value return expression decides.
func (check *Checker) inferShortFuncResult(sig *Signature, body
*ast.BlockStmt) Type {
    defer func(env environment) {
        check.environment = env
    }(check.environment)
    check.environment = environment{
        decl:    check.decl,
        scope:   sig.scope,

```

```

    version: check.version,
    iota:    check.iota,
    sig:     sig,
}

var result Type
ast.Inspect(body, func(n ast.Node) bool {
    switch ret := n.(type) {
    case *ast.FuncLit, *ast.ShortFuncLit:
        return false // returns of nested literals do not count
    case *ast.ReturnStmt:
        if result == nil && len(ret.Results) == 1 {
            var x operand
            check.expr(nil, &x, ret.Results[0])
            if x.mode != invalid && isValid(x.typ) {
                result = Default(x.typ)
            }
        }
        return false
    }
    return result == nil
})
return result
}

// shortFuncArg returns the short function literal of a call argument operand
// that has not been resolved yet (see genericExprList), or nil.
func shortFuncArg(arg *operand) *ast.ShortFuncLit {
    if lit, ok := arg.expr.(*ast.ShortFuncLit); ok && !isValid(arg.typ) {
        return lit
    }
    return nil
}

// resolveShortFuncArgs resolves short function literals among call arguments
// once the expected parameter types are known. For generic callees the type
// parameters are first partially inferred from the other arguments; literals
// resolved here then take part in the regular inference as fully typed

```

```

// function arguments. It reports whether checking of the call may proceed.
func (check *Checker) resolveShortFuncArgs(call *ast.CallExpr, sigParams
*Tuple, tparams []*TypeParam, targs []Type, args []*operand) bool {
    found := false
    for _, arg := range args {
        if shortFuncArg(arg) != nil {
            found = true
            break
        }
    }
    if !found {
        return true
    }

    // Partial inference from the already typed arguments, so that the
    // expected function types become as concrete as possible.
    var smap substMap
    if len(tparams) > 0 {
        u := newUnifier(tparams, targs, check.allowVersion(go1_21))
        for i, arg := range args {
            if i >= sigParams.Len() {
                break
            }
            if shortFuncArg(arg) != nil || arg.mode == invalid
|| !isValid(arg.typ) {
                continue
            }
            u.unify(sigParams.vars[i].typ, Default(arg.typ), assign)
        }
        smap = make(substMap)
        for _, tpar := range tparams {
            if t := u.at(tpar); t != nil {
                smap[tpar] = t
            }
        }
    }

    for i, arg := range args {

```

```

    lit := shortFuncArg(arg)
    if lit == nil || i >= sigParams.Len() {
        continue
    }
    ptype := sigParams.vars[i].typ
    if len(smap) > 0 {
        ptype = check.subst(call.Pos(), ptype, smap, nil, check.context())
    }
    expected, _ := ptype.Underlying().(*Signature)
    if expected == nil {
        check.errorf(lit, IncompatibleAssign, "cannot use a short
function literal as a value of type %s", ptype)
        arg.mode = invalid
        return false
    }
    sig := check.shortFuncSig(lit, expected)
    if sig == nil {
        arg.mode = invalid
        return false
    }
    arg.mode = value
    arg.typ = sig
    check.record(arg)
}
return true
}

// containsTypeParam reports whether t contains unresolved type parameters.
func containsTypeParam(t Type) bool {
    switch v := Unalias(t).(type) {
    case *TypeParam:
        return true
    case *Slice:
        return containsTypeParam(v.Elem())
    case *Array:
        return containsTypeParam(v.Elem())
    case *Pointer:
        return containsTypeParam(v.Elem())

```

```

case *Map:
    return containsTypeParam(v.Key()) || containsTypeParam(v.Elem())
case *Chan:
    return containsTypeParam(v.Elem())
case *Signature:
    return tupleContainsTypeParam(v.Params()) ||
tupleContainsTypeParam(v.Results())
case *Named:
    args := v.TypeArgs()
    for i := 0; i < args.Len(); i++ {
        if containsTypeParam(args.At(i)) {
            return true
        }
    }
    return false
default:
    return false
}
}

func tupleContainsTypeParam(tuple *Tuple) bool {
    for i := 0; i < tuple.Len(); i++ {
        if containsTypeParam(tuple.At(i).Type()) {
            return true
        }
    }
    return false
}
}

```

## Акт впровадження результатів роботи

ПОГОДЖЕНО

Проректор з наукової роботи  
Дніпровського національного університету  
імені Олеся ГончараОлег МАРЕНКОВ  
«18» Травня 2026 р.

ЗАТВЕРДЖЕНО

В.о. першого проректора  
Дніпровського національного університету  
імені Олеся ГончараВалентина СІЛІЧ-БАЛГАБАЄВА  
«18» Травня 2026 р.

АКТ

впровадження результатів роботи **Форкерта Павла Павловича**, поданої на здобуття наукового ступеня доктора філософії, на тему  
**«Дослідження використання рантайму Go як платформи для побудови нових мов програмування»**  
 в освітній процес Дніпровського національного університету імені Олеся Гончара

1. Вчена рада факультету прикладної математики та інформаційних технологій у складі 17 осіб заслухала повідомлення аспіранта кафедри інженерії програмного забезпечення та інформаційних технологій Форкерта Павла Павловича про результати наукового дослідження та їхнє використання в освітньому процесі галузі інформаційних технологій.

2. Стисла характеристика дослідження:

Павло Форкерт досліджує проблему використання рантайму Go як практичної платформи для побудови нових мов програмування, діалектів та мовних надбудов шляхом транспіляції у стандартний Go-код. У роботі проаналізовано сучасні підходи та платформи для створення нових мов програмування, зокрема компіляцію безпосередньо в машинний код, використання віртуальних машин, проміжних представлень, транспіляції, а також платформ JVM, LLVM, GraalVM/Truffle, JavaScript-екосистеми та .NET CLR. У цьому контексті обґрунтовано доцільність розгляду Go не як класичної багатомовної віртуальної машини чи низькорівневого компіляторного фреймворку, а як стабільної хост-платформи, що поєднує ефективний компілятор, конкурентний рантайм, сучасний збирач сміття, кроскомпіляцію, розвинену стандартну бібліотеку та інструменти для аналізу й генерації коду.

У межах дослідження розроблено узагальнену архітектуру транспілятора до Go та створено експериментальний діалект GoNext, який демонструє можливість додавання нових мовних конструкцій без модифікації офіційного компілятора Go. Реалізовано й проаналізовано механізми транспіляції для повнофункціональних enum-типів, зіставлення із взірцем, іменованих аргументів, параметрів за замовчуванням, методів розширення, узагальнених методів, коротких анонімних функцій та універсального синтаксису виклику функцій. Отримані результати мають практичне значення для створення нових мов програмування, транспіляторів чи мовних надбудов, що використовують екосистему Go та зберігають сумісність із наявними Go-бібліотеками.

3. Використання в освітньому процесі:

Результати дисертаційних досліджень впроваджено в освітній процес кафедри інженерії програмного забезпечення та інформаційних технологій факультету прикладної математики та інформаційних технологій ДНУ під час викладання освітньої компоненти «Мовні технології» для здобувачів першого (бакалаврського) рівня вищої освіти освітньої програми «Інженерія програмного забезпечення» спеціальності 121 Інженерія програмного забезпечення.

4. Відомості про впроваджені об'єкти інтелектуальної власності:

1. Р. Р. Forkert, М. G. Sydorova. Інтеграція повнофункціональних перерахувань у мову програмування Go // Актуальні проблеми автоматизації та інформаційних технологій. – Дніпро: ДНУ, 2023. – Т.27. – С. 3 – 16. DOI: <http://dx.doi.org/10.15421/432301> [Фахове видання України категорії Б]

(особистий внесок: провів аналіз підходів до реалізації мов програмування та можливостей використання транспіляції в Go; сформулював вимоги до повнофункціональних enum-типів у діалекті GoNext; запропонував синтаксис їх оголошення. Дослідив та порівняв кілька варіантів представлення enum-типів у Go-кодi, зокрема з урахуванням узагальнень, вказівників, роботи збирача сміття та накладних витрат. Розробив підхід до генерації конструкторів, Match-методів і допоміжних методів доступу, а також проаналізував сумісність запропонованого рішення з екосистемою Go.)

2. Р. Р. Forkert, М. G. Ivanchenko. Implementing named arguments in go programming language dialect // Актуальні проблеми автоматизації та інформаційних технологій. – Дніпро: ДНУ, 2025. – Т.29. – С. 3 – 11. DOI: <http://dx.doi.org/10.15421/432501> [Фахове видання України категорії Б]

(особистий внесок: провів порівняльний аналіз підходів до реалізації іменованих аргументів у сучасних мовах програмування; обґрунтував вибір call-site-підходу, за якого іменовані аргументи не потребують змін у визначеннях функцій. Запропонував синтаксис іменованих аргументів для GoNext, дослідив його сумісність із граматикою Go та розробив алгоритм транспіляції, що передбачає визначення сигнатури функції, перевірку помилкових і дубльованих імен параметрів. Окремо проаналізував інтеграцію іменованих аргументів із механізмом значень параметрів за замовчуванням.)

3. Форкерт П. П., Іванченко М. Г. Реалізація методів розширення та узагальнених методів у діалекті мови програмування Go // Системні технології. – Дніпро, 2026. – Т.2 (163). – С. 111 – 121. DOI: <https://doi.org/10.3418508/1562-9945-2-163-2026-10> [Фахове видання України категорії Б]

(особистий внесок: провів аналіз реалізацій методів розширення в сучасних мовах програмування та порівняв їх з універсальним синтаксисом виклику функцій. Сформулював вимоги до реалізації методів розширення в GoNext з урахуванням мінімальних накладних витрат і повної сумісності зі стандартним Go. Запропонував синтаксис оголошення методів розширення через модифікатор extension, механізм import extension для використання функцій наявних Go-бібліотек як методів розширення, а також алгоритм переписування викликів виду value.Method(args) у звичайні виклики функцій. Дослідив застосування цього підходу для підтримки узагальнених методів, які відсутні у стандартному Go, та проаналізував інтероперабельність запропонованого рішення з існуючим кодом на Go.)

5. Пропозиції ради:

Запропоновано впровадити результати дисертаційної роботи Форкєрта Павла Павловича на тему «Дослідження використання рантайму Go як платформи для побудови нових мов програмування» в освітній процес Дніпровського національного університету імені Олеся Гончара.

Голова вченої ради  
факультету прикладної математики  
та інформаційних технологій

Секретарка



Олена КІСЕЛЬОВА

Наталія ЛИСИЦЯ