PROGRAMMING LANGUAGES TERM PROJECT

Team Members:

1. Min Myint Moh Soe (6530262)
2. Hpone Pyae Khine (6511146)
3. Ahkar Min Oo (6511121)

Table of Contents

Introduction

Nim, a statically typed, compiled programming language, emerged as a significant innovation in the programming landscape in the early 21st century. Conceived by German programmer Andreas Rumpf, Nim's development began in the early 2000s as a response to the growing need for a language that could bridge the gap between high-level expressiveness and low-level performance.

Origin and Development

- **Conceptualization (Early 2000s):**

  Andreas Rumpf envisioned a language combining Python's readability with C's performance. Initial design goals included static typing, efficient compilation, and meta-programming capabilities.

- **First Release as "Nimrod" (2008):**

  The language was initially released under the name "Nimrod." Early versions showcased the language's potential for systems programming and application development.

- **Evolution and Refinement (2008-2014):**

  Continuous improvements in compiler technology and language features. Growing community contributions and feedback shaped the language's development.

- **Rebranding as "Nim" (2014):**

  The language was renamed from "Nimrod" to "Nim" to reflect its maturation and broader appeal. This marked a significant milestone in the language's development and community growth.

- **Recent Developments (2014-Present):**

  Ongoing refinements in performance, syntax, and standard library. Increased adoption in various domains, including systems programming, web development, and scientific computing.

Nim's development has been characterized by a commitment to combining the best features of established languages. It draws inspiration from Python's readable syntax, Ada's strong typing system, and Modula's modularity, while aiming to achieve C-like performance. This unique blend positions Nim as a versatile tool in the modern programmer's toolkit, capable of addressing a wide range of programming challenges across different domains.

Strengths of Nim

Compilation and Performance

- **Efficient Compilation:** Nim’s compilation model is a key strength, enabling the language to generate highly optimized executables that can rival the speed and efficiency of those written

in C. By compiling to C, C++, or JavaScript, Nim harnesses the power of mature, optimized compilers, which not only enhances performance but also broadens the language's portability across different platforms.

- **Versatility in Deployment:** The ability to compile to multiple target languages offers significant deployment flexibility. For example, compiling to JavaScript allows developers to create web applications that run in the browser, while compiling to C or C++ facilitates the development of performance-critical native applications. This versatility makes Nim an attractive option for projects that need to target multiple environments with a single codebase.

Metaprogramming Capabilities

- **Powerful Macros:** Nim's macro system is a key feature, offering extensive metaprogramming capabilities that allow developers to extend and customize the language's syntax and semantics. By manipulating the abstract syntax tree (AST) at compile-time, these macros are particularly effective for creating domain-specific languages (DSLs) and optimizing complex programming tasks.
- **Enhanced Code Expressiveness:** Metaprogramming in Nim enables the creation of highly expressive and efficient code. Macros can automate repetitive tasks, enforce coding standards, and implement custom features specific to application needs. This flexibility is rare among programming languages and significantly contributes to Nim's distinct appeal.
- **Strengths:** Nim's metaprogramming capabilities are among its most powerful features, with a sophisticated macro system that supports extensive compile-time code generation and DSL creation. This enhances the language's expressiveness and enables performance optimizations by shifting computations to compile-time, thereby reducing runtime overhead.
- **Limitations:** However, the use of complex macros can make code challenging to debug and maintain. The advanced nature of Nim's metaprogramming features may present a steep learning curve for developers, particularly those new to the language. Overusing these features can lead to obfuscated code, complicating readability and maintenance.

Memory Management Flexibility

- Nim offers developers a wide range of memory management options, including manual management, reference counting, and tracing garbage collection. This flexibility is crucial for applications needing precise control over resource allocation, such as real-time systems or performance-critical software. By allowing developers to tailor memory management strategies to the specific requirements of their applications, Nim supports the creation of highly optimized software capable of operating efficiently under demanding conditions.
- **Strengths :** Nim's flexible memory management approach, combining garbage collection, manual management, and optional ownership, is particularly advantageous for systems programming, where detailed control over memory is essential. This versatility enables developers to optimize performance based on project-specific needs, making it well-suited for performance-critical applications.
- **Limitations :** However, the diversity of memory management options can result in inconsistent coding practices across or within projects. Manual memory management, while offering control, carries the risk of memory leaks and errors, requiring meticulous attention from developers. Additionally, the use of garbage collection may introduce performance overhead in real-time or performance-critical contexts.

Multi-paradigm Support

- **Versatile Programming Paradigms:** Nim's support for multiple programming paradigms—imperative, object-oriented, and functional—enables developers to choose the approach that best suits their needs. This versatility is particularly valuable for large, complex projects that may benefit from different paradigms for different components.
- **Robust and Maintainable Code:** The ability to mix and match paradigms within a single project allows developers to leverage the strengths of each, resulting in more robust and maintainable code. For instance, object-oriented programming can be used to model complex data structures, while functional programming techniques can efficiently manage concurrent or parallel operations. This flexibility makes Nim a powerful tool for balancing diverse requirements within a single application.

Concurrency Model

- **Efficient Concurrency**: Nim's concurrency model, based on the async-await syntax and lightweight coroutines, streamlines the development of concurrent and parallel programs. The async-await syntax enables developers to write asynchronous code that is clear and maintainable, while coroutines enhance efficient resource use.
- **Scalability for Modern Applications:** This model is ideal for applications that manage large numbers of simultaneous tasks, such as web servers or real-time data processing systems. Nim's efficient and user-friendly concurrency framework allows developers to build scalable, high-performance applications that meet the needs of modern software development.
- **Strengths:** Nim's concurrency model is versatile, supporting both thread-based and async/await paradigms, allowing developers to select the approach that best fits their application. The use of coroutines further boosts the language's capacity to handle concurrent operations efficiently, making it suitable for tasks that demand high performance and scalability.
- **Limitations:** However, Nim lacks built-in protections against data races, unlike languages like Rust, which can lead to potential safety issues in concurrent programming. Additionally, the challenge of mixing different concurrency models and the learning curve associated with coroutines and async programming may present difficulties for less experienced developers.

Weaknesses of Nim

Limited Ecosystem

- **Challenges in Resource Availability:** Despite its many strengths, Nim's relatively limited ecosystem poses significant challenges, particularly for larger projects. Compared to more established languages, Nim offers fewer libraries and tools, making it difficult for developers to find the necessary resources for their work. This scarcity often forces developers to implement certain functionalities from scratch or port libraries from other languages, which can increase both development time and effort, potentially slowing down project timelines.

Steep Learning Curve

- **Complexity for New Developers:** While Nim is designed with user-friendliness in mind, its unique combination of features and syntax can present a steep learning curve, especially for developers who are not already familiar with both high-level and systems programming concepts. The language's powerful macro system, though a significant strength, requires a deep understanding of its syntax and semantics, making it challenging to master. This complexity can act as a barrier to entry for new developers, limiting Nim's adoption in certain contexts and slowing the onboarding process.

Compiler Stability Issues

- **Concerns with Reliability:** As a relatively young language, Nim's compiler has historically faced stability issues. Although there have been significant improvements over time, developers may still encounter bugs or unexpected behavior, particularly when utilizing advanced features. These stability issues can be particularly frustrating for developers working on large or complex projects where reliability is critical, potentially undermining confidence in the language for high-stakes applications.

Documentation Gaps

- **Inadequate Coverage of Advanced Topics:** Although the quality of Nim's documentation has improved, gaps remain, particularly in coverage of more advanced topics. This can make it challenging for developers to fully leverage all of Nim's capabilities without significant exploration or reliance on community support. These documentation gaps not only contribute to the learning curve but also make it difficult for new developers to get started with the language, potentially deterring adoption.

Limited Industrial Adoption

- **Concerns About Long-Term Viability:** Nim has not yet achieved widespread adoption in industry, which raises concerns about long-term support and the availability of experienced developers. This lack of adoption can make it difficult to find skilled Nim developers for large-scale projects, limiting the language's use in commercial applications. Additionally, uncertainty about the language's future direction may discourage developers and organizations from investing in Nim for critical projects.

Evolving Language Design

- **Impact on Code Stability:** Nim's evolving language design, while contributing to its continuous improvement, also introduces challenges related to code maintenance and stability. Frequent changes in the language's design and best practices can lead to code deprecation, requiring ongoing maintenance. This need for continual updates can be burdensome for developers working on long-term projects, potentially leading to issues with code stability and increased maintenance costs.

Functionality of Nim: Syntax, Runtime Behavior, and Underlying Principles

Nim is a programming language that distinguishes itself through a combination of clean syntax, robust runtime behavior, and powerful underlying principles. These characteristics make it a versatile tool for a wide range of programming tasks, from systems programming to application development. This section delves into how Nim works, with a focus on its syntax, runtime behavior, and the advanced features that contribute to its effectiveness.

Nim's Syntax and Core Language Features

Nim's syntax philosophy demonstrates a hybrid approach, combining elements from both Python-like and C-style languages. Key syntactic features include:

Indentation and Block Structure :
One of the foundational aspects of Nim's syntax is its use of indentation to define code blocks, similar to Python. This approach enhances readability by visually demarcating different sections of code. For example, a simple conditional structure in Nim may appear as follows:

```
if x > 0:
  echo "x is positive"
  if x > 10:
    echo "x is greater than 10"
else:
  echo "x is non-positive"
```

However, unlike Python, Nim provides the flexibility to use curly braces {} to explicitly define blocks. This feature can be particularly useful in scenarios where concise, one-liner code is desirable.

Variable and Constant Declarations :
Nim offers multiple mechanisms for declaring variables and constants, each catering to different needs within the language. The primary keywords involved in these declarations are var, let, and const:

var: Declares mutable variables, allowing their values to be changed after initialization.
let: Declares immutable variables, similar to the final keyword in Java or const in C++. Once initialized, the value cannot be altered.
const: Declares compile-time constants, which are values known and fixed during the compilation process.

```
Example:
var x = 5         # Mutable variable
let y = 10        # Immutable variable
const z = 15      # Compile-time constant
```

Nim also supports multiple variable declarations and tuple unpacking, which streamline the process of initializing several variables simultaneously:

```
var a, b, c: int  # Multiple variable declaration
let (d, e) = (1, 2)  # Tuple unpacking
```

This approach contrasts with Python's more simplistic variable declaration but aligns closer with Rust's let and mut keywords:

```rust
// rust example

let mut x = 5;  // Mutable variable
let y = 10;     // Immutable variable
const Z: i32 = 15;  // Compile-time constant
```

Go, on the other hand, uses a different approach:

```go
// go example

var x = 5       // Mutable variable
const y = 10    // Compile-time constant
```

Nim's approach offers fine-grained control over mutability and compile-time evaluation, potentially leading to more robust and performant code

Comments :
Nim provides support for both single-line and multi-line comments, facilitating code documentation and readability:

- Single-line comments: Begin with a # symbol.
- Multi-line comments: Enclosed within #[and]#.

Examples:

```nim
# This is a single-line comment

#[
  This is a
  multi-line comment
]#
```

Unified Type Hierarchy

Unlike many other languages, Nim features a unified type hierarchy where all types, including basic types like int and float, are subtypes of system.RootObj. This approach is unique compared to languages like Python (which has a similar concept but implemented differently) or Rust and Go (which don't have a unified type hierarchy).

This design choice in Nim allows for more flexible generic programming and type relationships, potentially offering advantages in certain programming scenarios

Primitive Types:
Nim supports a comprehensive set of primitive types, balancing low-level control with high-level programming convenience. These types include numeric types, boolean types, and character and string types.

Numeric Types:
Nim offers various numeric types, including platform-dependent and platform-independent integers, as well as floating-point numbers:

Examples:

```
var i: int = 42          # Platform-dependent sized integer
var i8: int8 = 127       # 8-bit signed integer
var u: uint = 42         # Unsigned integer
var f: float = 3.14      # 64-bit float by default
var f32: float32 = 3.14  # 32-bit float
```

Nim's automatic type inference often negates the need for explicit type annotations:

```
var x = 42    # Inferred as int
var y = 3.14  # Inferred as float
```

Boolean Type:
The boolean type in Nim is straightforward, supporting values of true and false:

```
var b1: bool = true
var b2 = false  # Type inference works here too
```

Character and String Types:
Nim differentiates between characters and strings, where characters are single-byte and strings are mutable sequences of characters:

```
var c: char = 'A'
var s: string = "Hello, Nim!"
```

This mutability feature of strings distinguishes Nim from many other languages where strings are typically immutable.

Constructed Types :
Nim provides several constructed types that allow developers to build more complex data structures, such as arrays, sequences, tuples, and objects.

Arrays :
Arrays in Nim are fixed-size sequences of elements of the same type:

```
var a: array[3, int] = [1, 2, 3]
var b = [1, 2, 3]  # Type inferred as array[3, int]
echo a[0]  # Accessing array elements (zero-based indexing)
```

Sequences :

Sequences in Nim are dynamic-size arrays, akin to vectors in C++ or lists in Python:

```
var s: seq[int] = @[1, 2, 3]
s.add(4)  # Append to sequence
echo s[^1]  # Access last element
```

Tuples :

Tuples in Nim represent fixed-size heterogeneous collections:

```
var t: tuple[name: string, age: int] = ("Alice", 30)
echo t.name  # Access by field name
echo t[1]    # Access by index
```

Objects :

Objects in Nim are user-defined types with named fields, forming the foundation for object-oriented programming within the language:

Example :

```
type
  Person = object
    name: string
    age: int

var p = Person(name: "Bob", age: 25)
echo p.name
```

Control Structures :

Nim provides several control structures for directing the flow of a program, including conditional statements and loops.

Conditional Statements :

Nim uses if, elif, and else for conditional execution, allowing for straightforward branching logic:

Example :

```
let x = 10
if x < 0:
  echo "Negative"
elif x == 0:
  echo "Zero"
else:
  echo "Positive"
```

Nim also supports a case statement, which enables pattern matching:

Example :

```nim
let day = "Monday"
case day
of "Monday", "Tuesday": echo "Start of the week"
of "Friday": echo "End of the work week"
of "Saturday", "Sunday": echo "Weekend"
else: echo "Midweek"
```

Loops :

Nim offers for and while loops for iterative processes:

```nim
for i in 1..5:
  echo i

var j = 0
while j < 5:
  echo j
  inc j
```

Nim's for loops are versatile, capable of iterating over various types of collections:

```nim
for index, value in ["a", "b", "c"].pairs:
  echo index, ": ", value
```

Functions :

In Nim, functions are defined using the **proc** keyword, with support for default parameters, named arguments, and function overloading.

Basic Function Definition :A basic function in Nim is defined as follows:

```nim
proc greet(name: string): string =
  result = "Hello, " & name

echo greet("Nim")
```

Default Parameters and Named Arguments :

Nim supports default parameters and named arguments, providing flexibility in function calls:

```nim
proc greet(name: string, greeting: string = "Hello"): string =
  result = greeting & ", " & name

echo greet("Nim")
echo greet("Nim", greeting = "Hi")
```

Uniform Function Call Syntax (UFCS) :

Nim introduces a distinctive feature called Uniform Function Call Syntax (UFCS), which blurs the line between method calls and function calls. This syntax allows any function to be invoked using

method-call syntax, regardless of whether it was defined as a method of the object or as a separate function.

For example, in Nim, UFCS allows developers to write code like this:

```
// UFCS example

proc add(a, b: int): int = a + b
echo 5.add(3)  # Equivalent to add(5, 3)
```

This syntax is particularly powerful when chaining multiple operations:

```
import strutils
echo "Hello, World!".toUpperAscii().replace("WORLD", "Nim")
```

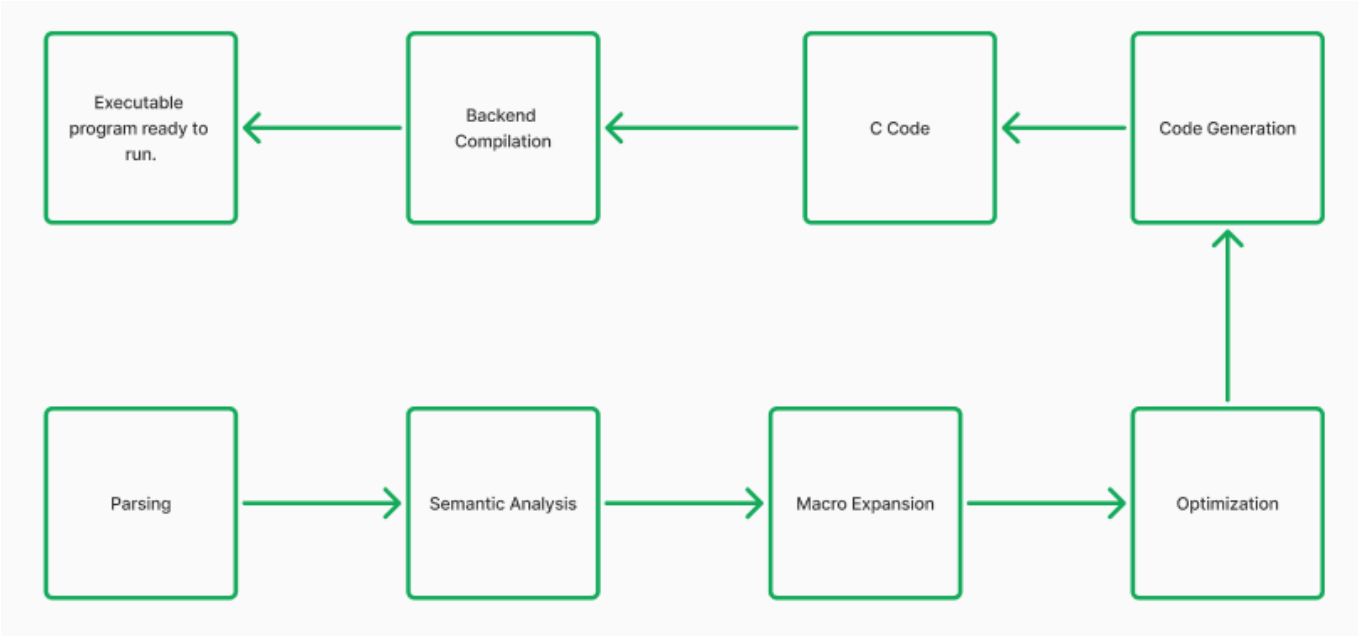
Other languages like Python does not have a direct equivalent to UFCS. Method chaining in Python is limited to methods defined within a class and Rust has a feature called "method syntax" which is similar to UFCS, but more limited. C# introduced a similar feature called "extension methods" in C# 3.0 and while powerful, C#'s extension methods require explicit declaration in static classes.

These core syntactic elements contribute to Nim's reputation for being both expressive and accessible, allowing developers to write concise, readable code.

Runtime Behavior and Compilation Process

Compilation Process

Nim's compilation process is a multi-stage, highly flexible operation designed to optimize performance while providing developers with a broad range of target outputs and customization options. This section outlines the various stages of Nim's compilation process and highlights advanced features that contribute to its efficiency and versatility.



1. **Parsing**

   The compilation process begins with parsing, where the Nim compiler analyzes the source code and converts it into an Abstract Syntax Tree (AST). This stage involves syntax analysis to ensure that the code conforms to the language's grammatical rules. Any syntax errors are detected and reported during this phase, allowing developers to correct issues early in the compilation process.

2. **Semantic Analysis**

   Following parsing, the AST undergoes semantic analysis, a crucial stage that involves type checking, symbol resolution, and the application of pragmas (compiler directives). During this phase, the compiler verifies that the code's semantics are correct, ensuring that variables, types, and functions are used consistently and according to the language's rules. This stage also includes the resolution of names and symbols, linking them to their definitions within the codebase.

3. **Macro Expansion**

   Nim's powerful macro system allows for extensive metaprogramming, and if the code contains macros, they are expanded at this stage. Macro expansion enables compile-time code generation and transformation, allowing developers to automate repetitive tasks and implement custom language constructs. This feature is one of Nim's most flexible tools, offering significant potential for optimizing and customizing the compilation process.

4. **Optimization**

   Once macro expansion is complete, the compiler performs various optimizations on the AST. These optimizations include dead code elimination, constant folding, and inlining, which collectively improve the efficiency and performance of the generated code. The optimization

phase is critical in reducing the overhead of the final executable, ensuring that the compiled code runs as efficiently as possible.

5. Code Generation

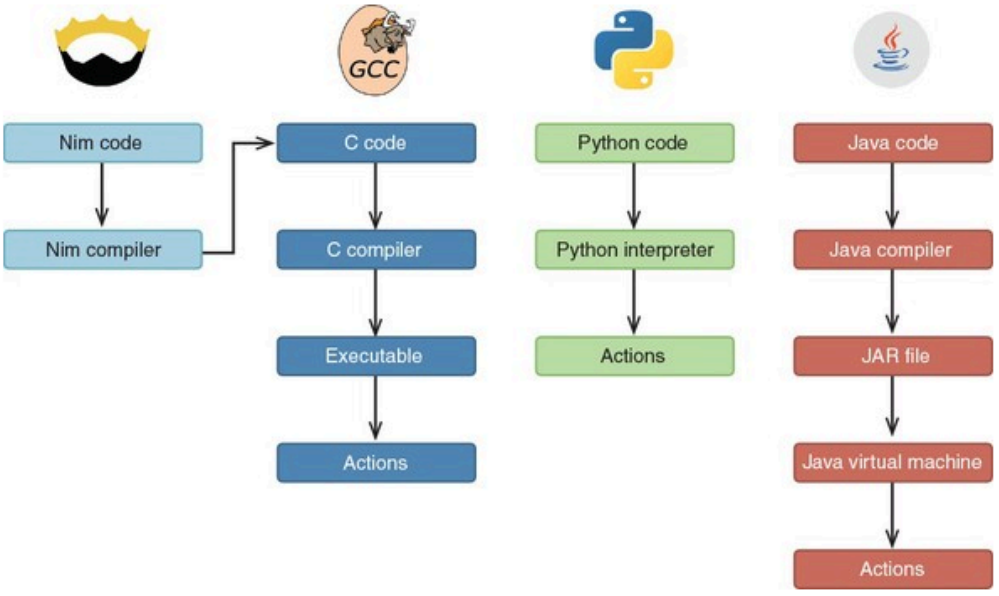
After optimization, the compiler generates an intermediate representation, typically in the form of C code. This step, known as transpilation, allows Nim to leverage the extensive optimization capabilities of existing C compilers. The generated C code serves as a bridge between Nim's high-level constructs and the low-level operations performed by the machine code.

6. Backend Compilation

The final stage of the compilation process involves backend compilation, where the generated C code is compiled into machine code using a C compiler such as GCC or Clang. This stage can be customized to target different architectures or to apply specific compiler optimizations, making Nim highly adaptable to various deployment environments. Nim also supports alternative backends, allowing for compilation to C++, Objective-C, and JavaScript, further expanding its deployment possibilities.

Nim's compilation process differs from other programming languages as shown in the below figure (Picheta, 2017)

Figure 1.3. How the Nim compilation process compares to other programming languages



"Source: Picheta, D. (2017). Nim in Action. Manning Publications. Retrieved from https://livebook.manning.com/book/nim-in-action/chapter-1/90"

Strengths:

Nim's compilation strategy, which targets C as an intermediate language, offers several significant advantages. By leveraging existing C compiler optimizations, Nim can produce highly optimized code that is efficient and fast. Furthermore, this strategy facilitates easy integration with C libraries and ensures good cross-platform support, making Nim a practical choice for a wide range of applications.

Limitations:

However, the reliance on C as an intermediate compilation step can impact compilation speed, potentially slowing down the development cycle. Debugging can also be more challenging due to the presence of intermediate C code, which may obscure the original Nim code's logic. Additionally, while this compilation strategy offers many benefits, it may limit optimizations specific to Nim's higher-level constructs, preventing the language from fully exploiting its unique features in certain scenarios.

Advanced Compilation Features

Nim's compilation process includes several advanced features that enhance its performance and flexibility, enabling developers to fine-tune the behavior of their applications.

Whole Program Optimization

Nim supports whole program optimization, which analyzes and optimizes the entire codebase, including across module boundaries. This feature can lead to significant performance improvements by allowing the compiler to apply global optimizations that are not possible when compiling modules in isolation.

Compile-Time Function Execution (CTFE)

Nim allows certain functions to be executed at compile-time through Compile-Time Function Execution (CTFE). This feature enables complex computations and optimizations to be performed during compilation, reducing runtime overhead and improving the efficiency of the generated code. CTFE is particularly useful for scenarios where calculations can be determined in advance, allowing the compiler to generate optimized constants and code paths.

Custom Backends

While C is the default backend for Nim, the language also supports compilation to C++, Objective-C, and JavaScript. This flexibility allows developers to target a wide range of platforms and environments, making Nim suitable for a variety of applications, from low-level systems programming to web development.

Runtime Behavior

Nim's runtime behavior is characterized by a combination of efficient memory management, advanced concurrency models, and customizable runtime features. These aspects of Nim's runtime environment contribute to its versatility and suitability for different types of applications.

Memory Management

- **a) Garbage Collection**

  Nim uses a deferred reference counting garbage collector with cycle detection by default. This garbage collection strategy provides efficient memory management for most use cases, balancing performance with ease of use. The garbage collector automatically handles memory allocation and deallocation, reducing the risk of memory leaks and other memory-related errors.

- **b) Manual Memory Management**

  For performance-critical sections of code, Nim allows developers to manage memory manually. This can be done using reference counting (`ref`) for controlled memory allocation and deallocation or unsafe pointer operations (`ptr`) for direct memory manipulation. This level of control is essential for low-level systems programming, where precise memory management is crucial.

- **c) Custom Allocators**

  Nim supports the use of custom allocators, giving developers fine-grained control over memory allocation strategies. This feature is particularly useful in scenarios where the default garbage collection behavior does not meet the performance requirements of the application, such as in real-time systems or memory-constrained environments.

Exception Handling

Nim employs a zero-cost exception handling model when exceptions are not raised, similar to C++. This model ensures that exception handling does not incur runtime costs unless an exception is actually thrown, making it an efficient choice for applications where exceptions are rare. This approach balances the need for robust error handling with the performance demands of high-performance applications.

Concurrency Model

- **a) Threading**

  Nim supports OS-level threads, making it suitable for CPU-bound tasks that require parallel execution. This feature allows developers to take full advantage of multi-core processors, enabling efficient concurrent processing and improving the performance of compute-intensive applications.

- **b) Async I/O**

  For I/O-bound operations, Nim provides a coroutine-based async/await system. This model allows for efficient handling of concurrent I/O operations without blocking the main execution thread. The async/await system in Nim is similar to those found in languages like Python and JavaScript, offering a familiar and powerful tool for managing asynchronous operations.

Advanced Runtime Features

Nim's runtime environment includes several advanced features that enhance its flexibility and adaptability, particularly in scenarios that require specialized performance or safety considerations.

Soft Real-Time Capabilities

Nim's customizable garbage collector and support for manual memory management make it suitable for soft real-time systems, where predictable performance is crucial. By allowing developers to fine-tune memory management and control garbage collection behavior, Nim can be used in applications that require consistent and reliable performance within specified time constraints.

Foreign Function Interface (FFI)

Nim provides a powerful Foreign Function Interface (FFI), enabling seamless integration with C libraries. This capability allows Nim programs to efficiently leverage existing C ecosystems, making it easier to reuse and integrate with existing software components. The FFI in Nim is straightforward to use, providing a smooth interface between Nim code and external C functions.

Runtime Type Information (RTTI)

Nim supports Runtime Type Information (RTTI), enabling dynamic type checks and reflective programming techniques. This feature allows developers to inspect types at runtime, perform type-based operations, and implement flexible and dynamic behavior in their programs. RTTI is particularly useful in scenarios where the exact types of objects may not be known until runtime.

Hot Code Reloading

While not a built-in feature, Nim's metaprogramming capabilities and FFI support enable the implementation of hot code reloading techniques for certain types of applications. Hot code reloading allows updates to be made to a running system without stopping or restarting it, which is valuable in scenarios requiring high availability and minimal downtime.

Nim's Type System

Nim's type system is a well-balanced integration of theoretical principles and practical design, aimed at achieving strong type safety while maintaining flexibility and ease of use. It features static typing with type inference, allowing developers to write concise and readable code without compromising the benefits of a statically typed language. This analysis delves into Nim's type system through the lens of language theory, highlighting its key features, theoretical foundations, and implications in the broader context of programming language design.

Strengths: Nim combines static typing with type inference, enhancing code safety and readability. The language accommodates both high-level abstractions and low-level control, making it suitable for a wide range of programming tasks. Advanced features, such as sum types and an effect system, offer a robust framework for expressive programming, enabling the development of complex and nuanced applications.

Limitations: However, these advanced features can contribute to a steep learning curve for new users. While type inference improves code simplicity, it can also reduce explicitness, particularly in larger

codebases, making maintenance and readability more challenging. Additionally, Nim does not support higher-kinded types, unlike languages such as Haskell or Scala, which could limit its expressiveness for developers familiar with such capabilities.

- **Static Typing with Type Inference**

  Nim employs a static type system enhanced by type inference, which combines the rigor of static typing with the convenience of dynamic languages. The system is inspired by the Hindley-Milner type system, known for its efficient type inference without requiring explicit type annotations in most cases. This allows Nim to infer types based on context, leading to concise and expressive code.

  Example:

```
let x = 5  # Type inferred as int
let y = 3.14  # Type inferred as float
let z = "Hello"  # Type inferred as string
```

  While similar to languages like Haskell and OCaml in its use of type inference, Nim's approach is more pragmatic, allowing for explicit type annotations when needed. This contrasts with languages like Java or C++, where type annotations are generally required.

- **Nominal Type System**

  Nim primarily utilizes a nominal type system, where two types are considered equal only if they share the same name. This contrasts with structural type systems, like those in OCaml or Go, where compatibility is determined by the structure of the types rather than their names. Nim's nominal approach aligns with traditional object-oriented languages, reinforcing the language's commitment to clear and explicit type relationships.

  Example:

```
type
  Animal = object
    name: string
  Dog = object
    name: string

proc makeSound(a: Animal) = echo "Generic animal sound"

let dog = Dog(name: "Fido")
# makeSound(dog)  # This would result in a type error
```

  This nominal typing is similar to Java or C#, providing clear type distinctions. It differs from structural typing in languages like TypeScript or Go, where the above example would be valid if the structures match.

- **Subtyping and Object-Oriented Features**

Nim supports subtyping through its object-oriented features, using single inheritance for objects and enabling interface-like behavior through concepts. This design provides a balance between flexibility and simplicity, allowing developers to leverage polymorphism while maintaining a clear and predictable type hierarchy.

Example :

```
type
  Animal = ref object of RootObj
    name: string
  Dog = ref object of Animal

method makeSound(a: Animal) = echo "Generic animal sound"
method makeSound(d: Dog) = echo "Woof!"

let animal: Animal = Dog(name: "Fido")
animal.makeSound()  # Outputs: Woof!
```

Nim's approach to subtyping is more similar to languages like Java or C# than to languages with multiple inheritance like C++. However, Nim's use of the `ref` keyword for reference types provides more explicit control over object allocation, similar to Rust's approach.

- **Generics**

  Nim's implementation of generics is both powerful and versatile, supporting type parameters and value parameters. This allows developers to create highly reusable and type-safe code, with the ability to define generic functions and data structures that work across various types.

  Example :

```
proc swapValues[T](a, b: var T) =
  let temp = a
  a = b
  b = temp

var x = 5
var y = 10
swapValues(x, y)
echo x, y  # Outputs: 10 5
```

  Nim's generics are more flexible than those in Java or C#, allowing for value parameters similar to C++'s template value parameters. This flexibility is closer to C++'s templates or Rust's generics, but with a simpler syntax.

Advanced Type System Features

Nim’s type system includes several advanced features that further enhance its flexibility and expressiveness, drawing on concepts from both practical and theoretical perspectives.

- **Union Types**

  Nim supports union types, allowing variables to hold values of multiple types. This feature is implemented using tagged unions, also known as sum types in type theory. Union types enable more expressive type definitions, facilitating the creation of data structures that can accommodate various forms of data.

  Example :

```nim
type
  MyUnion = object
    case kind: bool
    of true: intVal: int
    of false: floatVal: float

var x: MyUnion
x = MyUnion(kind: true, intVal: 42)
```

  This implementation of union types is similar to Rust's enums or Haskell's sum types, providing type-safe alternatives. It differs from C's unions, which are untagged and potentially unsafe.

- **Intersection Types**

  While Nim does not explicitly support intersection types, its concept system allows for functionality similar to intersection types, enabling ad-hoc polymorphism. Concepts in Nim provide a mechanism to enforce that a type must satisfy multiple constraints, effectively simulating intersection types.

  Example :

```nim
type
  Readable = concept
    proc read(self: Self): string
  Writable = concept
    proc write(self: Self, data: string)

proc process[T: Readable and Writable](x: T) =
  let data = x.read()
  x.write("Processed: " & data)
```

  This approach is similar to TypeScript's intersection types or Scala's traits, allowing for flexible composition of behaviors. It differs from languages like Java, which require explicit interface implementations.

- **Dependent Types**

  Although Nim does not fully support dependent types, it offers features like compile-time function execution (CTFE) and macros, which can emulate certain aspects of dependent

typing. These features allow developers to perform complex computations and enforce constraints at compile-time, enhancing type safety and program correctness.

Example :

```
import macros

macro createArrayType(n: static[int]): untyped =
  result = newTree(nnkBracketExpr, ident("array"), newLit(n), ident("int"))

type
  MyArray = createArrayType(5)

var x: MyArray  # Equivalent to: var x: array[5, int]
```

While not as powerful as the dependent types in languages like Idris or Agda, Nim's approach provides a pragmatic middle ground. It offers more compile-time guarantees than languages like C++ or Java, while remaining more accessible than fully dependent-typed languages.

- **Effect System**

Nim’s effect system extends its type system to include information about computational effects, such as side effects. This system is grounded in effect typing theory and allows developers to track and reason about side effects in their code. By explicitly declaring or inferring effects, Nim enhances the safety and predictability of function behavior.

Example :

```
proc pureFunction(x: int): int {.noSideEffect.} =
  result = x * 2

proc impureFunction(x: int): int {.raises: [IOError].} =
  result = readLine(stdin).parseInt() + x
```

Nim's effect system is more explicit than those in languages like Java or C++, which do not track effects at the type level. It's less comprehensive than the effect systems in languages like Koka or Eff, but provides a practical balance between expressiveness and simplicity.

- **Type Classes and Ad-Hoc Polymorphism**

Nim’s concepts are similar to type classes in Haskell, providing a form of bounded parametric polymorphism. Concepts allow developers to define interfaces that types must satisfy, enabling flexible and reusable code while maintaining type safety.

Example :

```nim
type
  Numeric = concept x
    x + x is type(x)
    x - x is type(x)
    x * x is type(x)
    x / x is float

proc sum[T: Numeric](values: openArray[T]): T =
  for value in values:
    result += value

echo sum([1, 2, 3, 4, 5])  # Works with int
echo sum([1.1, 2.2, 3.3, 4.4, 5.5])  # Works with float
```

Nim's approach to ad-hoc polymorphism is more flexible than Java's interfaces or C++'s virtual functions. It's similar to Haskell's type classes or Rust's traits, allowing for retroactive implementation of interfaces.

References and Pointers in Nim

Nim's approach to references and pointers is designed to balance safety, performance, and ease of use. This section explores how Nim handles references and pointers, comparing it with other languages in the systems programming family, such as C, C++, and Rust.

References in Nim

In Nim, references are created using the `ref` keyword. Unlike C++ references, Nim's references are more akin to safe pointers:

```nim
type
  Node = ref object
    data: int
    next: Node

var n = Node(data: 42, next: nil)
```

Compared to other languages:

- **C++:** References in C++ are aliases and must be initialized when declared. They cannot be reassigned.
- **Rust**: References in Rust are similar to C++ but with explicit lifetime annotations.
- **Nim**: References in Nim are more flexible than C++ references and safer than C pointers.

Pointers in Nim

Nim also supports low-level pointers using the `ptr` keyword, which are similar to C pointers:

```nim
var x: int = 42
var p: ptr int = addr(x)
echo p[]  # Dereference the pointer
```

Comparison with other languages:

- **C**: Pointers in C are low-level and prone to errors like buffer overflows.
- **C++**: C++ pointers are similar to C but with added features like smart pointers.
- **Rust**: Rust uses references for safe memory access and raw pointers for unsafe operations.
- **Nim**: Nim's `ptr` is similar to C pointers but with added safety features.

Memory Safety

Nim provides several mechanisms for memory safety:

- **Garbage Collection**: By default, Nim uses garbage collection for `ref` types.
- **Manual Memory Management**: For `ptr` types, manual management is possible.
- **Ownership Model**: Nim has an optional ownership model inspired by Rust.

```nim
proc takeOwnership(x: owned Node) =
  echo "Node data: ", x.data

var node = Node(data: 10)
takeOwnership(move(node))
# node is no longer accessible here
```

Comparison:

- **C/C++**: Manual memory management with potential for leaks and dangling pointers.
- **Rust**: Ownership system with borrow checker for compile-time memory safety.
- **Nim**: Flexible approach combining garbage collection, manual management, and optional ownership.

Null Safety

Nim addresses null pointer dereferences through its type system:

```
type
  SafeString = string | nil

proc process(s: SafeString) =
  if s != nil:
    echo s
  else:
    echo "String is nil"
```

Comparison:

- **C/C++**: No built-in null safety, relying on programmer discipline.
- **Rust**: Uses `Option<T>` for nullable types.
- **Nim**: Provides `Option[T]` and allows union types with `nil`.

Some Trade-offs in Nim's Design Decisions

This section examines the critical trade-offs in Nim's key design decisions, focusing on how these choices balance factors such as safety, performance, ease of use, and expressiveness.

1. Memory Management: Safety, Performance, and Flexibility

Nim's memory management approach offers a spectrum from garbage collection to manual management, attempting to cater to diverse use cases and programmer preferences.

1.1 Safety vs. Control

- Garbage collection: Offers safety and ease of use at the potential cost of performance and predictability.
- Manual management: Provides maximum control and optimization potential but increases the risk of memory-related errors.

This flexible approach raises questions about language coherence and potential fragmentation in coding practices.

1.2 Optimization vs. Consistency

The ability to choose different memory management strategies within a single project allows for performance optimization but may lead to:

- Inconsistencies across projects or within codebases
- Increased maintenance costs
- Higher likelihood of errors
- Steeper learning curve for new developers

1.3 Formal Guarantees vs. Expressiveness

Nim's approach bridges multiple paradigms, raising questions about type safety and formal verification:

- Garbage collection provides stronger memory safety guarantees
- Manual management allows more expressive power in resource control

This design opens avenues for research into formal methods for reasoning about safety and correctness in mixed memory management contexts.

2. Metaprogramming: Power vs. Simplicity and Maintainability

Nim's macro system offers powerful metaprogramming capabilities, representing a significant trade-off between expressive power and language simplicity.

2.1 Extensibility vs. Predictability

- Macros enable creation of domain-specific abstractions and reduce boilerplate
- Increased complexity can make code harder to understand and reason about
- Positions Nim closer to Lisp in metaprogramming power, distinct from more restrictive languages like Java or Go

2.2 Short-term Productivity vs. Long-term Maintenance

- Domain-specific abstractions can lead to more concise, readable code for experts
- Increased risk of creating opaque, hard-to-maintain code
- Potential barriers for new team members
- Complications in debugging and refactoring processes

2.3 Compile-time Expressiveness vs. Static Analyzability

- Powerful compile-time computations enabled
- Static analysis becomes more challenging due to differences between source code and executed code
- Raises fundamental questions about abstraction and compile-time vs. runtime computation boundaries

Comparing Nim with Prominent Programming Languages Across Paradigms

- Nim is a versatile, multi-paradigm programming language that effectively combines features from imperative, object-oriented, and functional programming paradigms. This flexibility allows it to compete with more established languages across various technical aspects, including syntax, memory management, metaprogramming, and performance.

Feature	Nim	C	Python	Java	C++	Haskell	OCaml

Paradigm	Multi-paradigm	Imperative	Multi-paradigm	Object-oriented	Multi-paradigm	Functional	Multi-paradigm
Type System	Static, inferred	Static	Dynamic	Static	Static	Static, inferred	Static, inferred |
Memory - management	GC, manual options	Manual	GC	GC	Manual, RAII	GC	GC
Concurrency	Threads, async/await	Threads (libraries)	GIL, async	Threads	Threads, std::async	STM, par	Lwt, Async
Metaprogramming	Powerful macros	Preprocessor macros	Decorators, metaclass	Reflection	Templates	Template Haskell	PPX
Compilation	Compiled (via C)	Compiled	Interpreted	Compiled to bytecode	Compiled	Compiled	Compiled
Performance	High	High	Moderate	Moderate to High	High	Moderate to High	High

Imperative Paradigm Comparison

Nim vs. C

- **Syntax:** Nim's syntax is concise and inspired by Python, using indentation to define blocks, unlike C's reliance on curly braces and semicolons. This makes Nim more readable and accessible to developers familiar with high-level languages.
- **Memory Management:** Nim's automatic garbage collection simplifies development compared to C's manual memory management, reducing risks of memory leaks and errors, thereby streamlining the development process.
- **Metaprogramming:** Nim's macro system, operating on the Abstract Syntax Tree (AST), offers far greater flexibility than C's basic text substitution macros, enabling sophisticated compile-time code manipulation.
- **Performance:** Both Nim and C achieve high-performance levels, with Nim compiling to C and leveraging mature optimization techniques from C compilers like GCC and Clang, maintaining performance while improving code readability.

Nim vs. Python

- **Execution:** As a compiled language, Nim typically outperforms Python, which is interpreted, making Nim more suitable for performance-critical applications.
- **Typing:** Nim's static typing with type inference ensures type safety at compile time, reducing runtime errors, whereas Python's dynamic typing offers flexibility but may introduce type-related issues during execution.
- **Concurrency:** Nim supports both threading and asynchronous programming with built-in tools, offering robust concurrency options. Python's Global Interpreter Lock (GIL) limits true parallelism, particularly in CPU-bound tasks.

Object-Oriented Paradigm Comparison

Nim vs. Java

- **Class Definition**: Nim uses the type keyword to define objects, offering a flexible approach compared to Java's class keyword.
- **Inheritance**: Nim supports single inheritance with the ref object of syntax, similar to Java's extends, emphasizing simplicity and flexibility.
- **Interfaces**: Although Nim lacks direct equivalents to Java interfaces, it uses concepts and type classes to achieve similar functionality, offering a more flexible way to enforce behaviors across different types.
- **Runtime**: Nim compiles to native code, which eliminates the need for a virtual machine like Java's JVM, offering performance and portability advantages.

Nim vs. C++

- **Memory Management**: Nim's garbage collection contrasts with C++'s manual memory management and RAII (Resource Acquisition Is Initialization). While C++ offers detailed control, Nim simplifies development and reduces memory-related errors.
- **Templates**: Nim's generics are more akin to D's templates, offering a simpler, more powerful approach compared to C++.
- **Operator Overloading:** Both languages support operator overloading, but Nim's syntax is generally more concise and easier to read.
- **Multiple Inheritance:** Nim only supports single inheritance, simplifying the inheritance model but potentially limiting some design possibilities compared to C++.

Functional Paradigm Comparison

Nim vs. Haskell

- **Purity**: Haskell enforces pure functions by default, aiding in code reasoning. Nim allows purity but doesn't enforce it, offering more flexibility with the trade-off of potentially introducing side effects.
- **Lazy Evaluation:** Haskell's lazy evaluation means expressions are only evaluated when needed, whereas Nim uses strict evaluation, though lazy sequences are possible when necessary.
- **Pattern Matching:** Haskell's pattern matching is more expressive and powerful compared to Nim's case statements, simplifying complex code patterns.
- **Type System:** Haskell's advanced type system includes higher-kinded types, making it more suitable for abstract and complex type manipulations than Nim.

Nim vs. OCaml

- **Type Inference:** Both Nim and OCaml support type inference, but OCaml's is generally more sophisticated, providing stronger guarantees and catching more errors at compile time.
- **Module System:** OCaml's module system, featuring functors for modular and reusable code, is more powerful than Nim's module system.

- **Functional Data Structures:** OCaml emphasizes immutable data structures essential for functional programming. Nim can support functional programming but doesn't emphasize immutability to the same extent.
- **Metaprogramming:** Nim's macro system offers more extensive metaprogramming capabilities than OCaml's PPX system, enabling complex compile-time code generation and manipulation to enhance flexibility and reduce runtime overhead.

Comparing Nim With Even More Languages from Different Paradigms

Feature	Nim	Rust	Go	Erlang
Paradigm	Multi-paradigm	Multi-paradigm	Concurrent	Functional
Type System	Static, inferred	Static, inferred	Static	Dynamic
Memory-Management	GC, manual options	Ownership model	GC	GC
Concurrency	Threads, async/await	Fearless concurrency	Goroutines, channels	Actor model
Metaprogramming	Powerful macros	Procedural macros	Limited	Parse transforms
Safety Features	Optional GC	Borrow checker	Built-in concurrency	Supervisor trees
Compilation	Compiled (via C)	Compiled	Compiled	Compiled to bytecode
Performance	High	High	High	Moderate to High

Comparison with Rust

- **Memory Safety:** Rust enforces memory safety through its ownership system and borrow checker, ensuring safe memory access at compile time without needing a garbage collector. Nim, by contrast, uses garbage collection by default but allows manual memory management. While Nim offers simpler memory management options, it does not provide the same compile-time safety guarantees as Rust.
- **Concurrency Model**: Rust's concurrency model is tightly integrated with its ownership system, preventing data races at compile time. Nim supports both threading and asynchronous programming using async/await. However, Nim's approach relies more on developer discipline, lacking the strict compile-time checks found in Rust.
- **Metaprogramming:** Nim's macro system is more powerful and flexible than Rust's procedural macros, enabling complex compile-time code manipulation and custom syntax extensions. Rust's macros are more restricted, prioritizing safety and hygiene but offering less flexibility.

- **Performance:** Both Nim and Rust achieve performance levels comparable to C, thanks to low-level system access and optimizations. However, Rust's zero-cost abstractions and stricter memory safety can lead to more stable performance in complex concurrent systems.
- **Learning Curve:** Nim is generally easier to learn due to its Python-like syntax and more straightforward memory management model. Rust's steep learning curve is primarily due to its ownership system and rigorous compile-time checks, requiring a deeper understanding of memory management.

Comparison with Go

- **Type System:** Nim's type system is more advanced than Go's, featuring generics, operator overloading, and support for multiple paradigms. Go, designed for simplicity, initially had a very basic type system, with generics only introduced in Go 1.18, reflecting its focus on reducing complexity and enhancing ease of use.
- **Concurrency Model:** Go's concurrency model, based on goroutines and channels, provides a lightweight and scalable approach to concurrent programming. Nim offers both OS threads and async/await, which add flexibility but also complexity in managing different concurrency models.
- **Compilation:** Nim compiles to C, which is then compiled to machine code, leveraging the mature optimization techniques of C compilers and enabling cross-platform development. Go compiles directly to machine code, often resulting in faster compilation times and a simplified build process.
- **Memory Management:** Both Nim and Go use garbage collection, but Nim offers more control, allowing developers to fine-tune memory management. Go's garbage collector is optimized for low-latency applications, focusing on minimizing pause times, albeit with less developer control.
- **Metaprogramming:** Nim's macro system allows for extensive metaprogramming, enabling custom syntax creation, task automation, and complex compile-time code generation. Go intentionally limits metaprogramming to maintain simplicity and readability, aligning with its design philosophy of clarity and maintainability.

Comparison with Erlang

- **Concurrency Model:** Erlang's concurrency model is based on the actor model, utilizing lightweight processes and message passing, making it ideal for building reliable, scalable distributed systems. Nim supports threads and async/await but lacks built-in actor model support, requiring additional libraries or custom implementations for similar functionality.
- **Fault Tolerance:** Erlang is renowned for its fault tolerance, encapsulated in its "let it crash" philosophy and supervisor trees that automatically recover from failures. Nim does not natively support such fault tolerance mechanisms, requiring manual strategies for error handling and recovery.
- **Distribution:** Erlang includes built-in support for distributed computing, simplifying the management of systems across multiple nodes. Nim can support distributed systems, but achieving the same level of functionality requires additional libraries and frameworks, increasing complexity.

- **Type System:** Nim's static typing with type inference ensures compile-time type safety, reducing runtime errors. In contrast, Erlang is dynamically typed with optional type specifications via tools like Dialyzer, emphasizing flexibility and runtime adaptability over strict compile-time guarantees.
- **Pattern Matching:** Pattern matching is central to Erlang's design, providing powerful tools for working with complex data structures. Nim supports pattern matching through case statements, but its capabilities are less extensive compared to Erlang, where pattern matching is more integral to the language's functionality.
- **Hot Code Reloading:** Erlang supports hot code reloading, allowing updates to be made to a running system without downtime—a critical feature for high-availability systems. Nim does not natively support hot code reloading, which can be a limitation in scenarios requiring uninterrupted service.

Performance Benchmarks of Nim with Various Programming Languages

To provide a quantitative comparison of Nim's performance against other programming languages, we can examine the results of **the Longest Path Finding Benchmark as reported by Felsing (2015)**. This benchmark provides insights into execution time, memory usage, compilation time, and code compactness across various languages.

Lang	Time [ms]	Memory [KB]	Compile Time [ms]	Compressed Code [B]
Nim	1400	1460	893	486
C++	1478	2717	774	728
D	1518	2388	1614	669
Rust	1623	2632	6735	934
Java	1874	24428	812	778
OCaml	2384	4496	125	782
Go	3116	1664	596	618
Haskell	3329	5268	3002	1091
LuaJit	3857	2368	-	519
Lisp	8219	15876	1043	1007
Racket	8503	130284	24793	741

Benchmark conducted on Linux x86-64, Intel Core2Quad Q9300 @2.5GHz, as of 2014-12-20 from Felsing, D. (2015, January 1). What is special about Nim?. HookRace Blog. https://hookrace.net/blog/what-is-special-about-nim/

- **Execution Time:** Nim demonstrates the fastest execution time (1400 ms) among all tested languages, slightly outperforming even C++ (1478 ms).
- **Memory Usage:** Nim shows efficient memory utilization (1460 KB), second only to Go (1664 KB) in this benchmark.
- **Compile Time:** While not the fastest to compile, Nim's compile time (893 ms) is competitive, especially when compared to languages like Rust (6735 ms) or Haskell (3002 ms).
- **Code Compactness:** Nim achieves the smallest compressed code size (486 bytes), indicating that it allows for concise expression of the algorithm.

These results suggest that Nim offers a compelling combination of performance, memory efficiency, and code compactness. It's particularly noteworthy that Nim outperforms C++ in this specific benchmark, both in execution time and code size, while maintaining comparable compile times.

Conclusion

Nim, as a multi-paradigm programming language, offers a unique blend of features that position it at an intriguing intersection between systems and application programming. Nim's compilation model, which involves generating C code as an intermediate step, allows it to leverage the optimizations of existing C compilers while maintaining a more approachable and readable syntax. This, coupled with Nim's powerful metaprogramming capabilities and flexible memory management, enables developers to write efficient and maintainable code. However, it is important to acknowledge that Nim's ecosystem and level of industry adoption are still developing, which could present challenges for large-scale or highly specialized projects.

In comparison with languages across various paradigms, Nim demonstrates remarkable versatility. It offers performance on par with systems languages like C and Rust, syntactic clarity akin to Python, object-oriented capabilities similar to Java and C++, and functional programming features that, while not as comprehensive as those in Haskell or OCaml, still provide significant flexibility. This adaptability makes Nim a valuable tool for developers working across diverse domains and paradigms.

However, Nim's versatility also means that it may not always be the optimal choice for tasks requiring highly specialized solutions where domain-specific languages excel. Furthermore, its smaller community and less mature ecosystem, compared to more established languages, could be a consideration for those developing large-scale industrial applications.

In conclusion, Nim represents an innovative approach to language design, effectively bridging the gap between low-level efficiency and high-level expressiveness. As the language continues to evolve and its community expands, Nim has the potential to become a significant player in the programming language ecosystem, especially for projects that require a balance of performance, readability, and cross-paradigm flexibility. The future of Nim will likely depend on its ability to carve out niches

where its unique combination of features offers tangible advantages over more established alternatives.

References

1. Rumpf, A. (2008). *Nimrod Programming Language*. Retrieved from nim-lang.org
2. Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review, 59*(1), 65-98.
3. Nystrom, R. (2021). *Crafting Interpreters*. Genever Benning.
4. Meyer, B. (1992). *Eiffel: The Language*. Prentice Hall.
5. Picheta, D. (2017). *Nim in Action*. Manning Publications.
6. Salzman, E. J. (2019). *Nim for Python Programmers*. Apress.
7. Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley Professional.
8. Stroustrup, B. (2013). *The C++ Programming Language* (4th ed.). Addison-Wesley.
9. Klabnik, S., & Nichols, C. (2018). *The Rust Programming Language*. No Starch Press.
10. Donovan, A. A., & Kernighan, B. W. (2015). *The Go Programming Language*. Addison-Wesley Professional.
11. Cesarini, F., & Thompson, S. (2009). *Erlang Programming*. O'Reilly Media.
12. Fasolin, A. (2021). Nim vs. Crystal vs. Go - A detailed comparison. *Dev.to*. Retrieved from https://dev.to/andrefasolin/nim-vs-crystal-vs-go-a-detailed-comparison-5bm1
13. Nim. (n.d.). *Nim Programming Language*. Retrieved from https://nim-lang.org
14. LogRocket. (2023, June 8). Nim vs. Python: Which should you choose? *LogRocket Blog*. Retrieved from https://blog.logrocket.com/nim-vs-python-which-should-you-choose/
15. LogRocket. (2023, June 8). Comparing Rust and Nim: Which should you use? *LogRocket Blog*. Retrieved from https://blog.logrocket.com/comparing-rust-nim/
16. Modrzyk, N. (2018). Week 2: Nim. In *Building Telegram Bots* (pp. 23-45). Apress. https://doi.org/10.1007/978-1-4842-4197-4_2
17. Nim. (n.d.). *Nim Features*. Retrieved from https://nim-lang.org/features.html
18. Nim Documentation. (n.d.). Retrieved from https://nim-lang.org/documentation.html
19. Picheta, D. (2017). *Nim in Action*. Manning Publications. Retrieved from https://livebook.manning.com/book/nim-in-action/chapter-1/90
20. Felsing, D. (2015, January 1). What is special about Nim?. *HookRace Blog*. Retrieved from https://hookrace.net/blog/what-is-special-about-nim/