Information, Computing and Intelligent Systems, 2025, No. 7, 142 – 159
UDC 004.42 https://doi.org/10.20535/2786-8729.7.2025.341609

OPTIMIZED SYNTAX CONCEPT FOR VARIABLE SCOPING, LOOP STRUCTURES, AND FLOW CONTROL IN PROGRAMMING LANGUAGE

Oleksandr Zhyrytovskyi*
https://orcid.org/0009-0009-3724-941X

Roman Zubko
https://orcid.org/0009-0002-0738-3852

Open International University of Human Development "Ukraine"
Kyiv, Ukraine

*Corresponding author: i.am.zhirik@gmail.com

This article examines syntactic redundancy in modern programming languages and its impact on code perception, readability, and logical consistency. The object of the study is the analysis of redundant syntactic constructs, particularly those related to variable declarations, scope management, loop structures, and flow control mechanisms. The primary aim is to develop and substantiate an optimized syntax concept. This concept combines the declarative rigor of classical languages with the simplicity of dynamic systems. The goal is to reduce code redundancy and improve cognitive ergonomics for developers.

The research methodology involved a comparative analysis of key syntactic elements across different language paradigms. The materials for the study included a formal comparison of semantics and an evaluation of equivalent program fragments written in classical languages and in the proposed conceptual language.

The results show that the proposed syntactic model significantly reduces auxiliary symbols, improves code clarity, and lowers cognitive load. The scientific novelty is a holistic syntax model defined by three key innovations. First, a simplified variable management system creates local variables automatically, eliminating keywords like `var` or `global` and using explicit markers for outer-scope access. Second, a universal loop operator unifies the functionality of traditional `for`, `while`, and `do-while` loops, allowing condition evaluation at the beginning, middle, or end of the block. Third, the traditional `goto` operator is replaced with a structured `try-throw` construct, providing a safe, semantically coherent mechanism for exiting nested blocks and error handling. This unified approach forms a basis for further research into minimalist syntax focused on naturalness and readability.

Keywords: programming language, language design, syntax model, variable scoping, loop structures, flow control, error handling, code readability, syntax optimization, syntax unification.

1. Introduction

The evolution of programming languages has been a continuous search for a balance between syntactic simplicity and functional power. This scientific domain explores how language design impacts the software development process. Historically, the first compilers for languages such as Pascal and C required programmers to provide detailed declarative descriptions of variables and data types. While this increased formal rigor, it often reduced the convenience of writing code. With the emergence of interpreted languages like PHP and JavaScript, software development became simpler due to dynamic typing and automatic memory management. Over time, the difference between static and dynamic typing has encouraged the development of hybrid approaches that use ideas from both.

However, even in modern hybrid languages, a degree of syntactic overload often persists. This overload is typically associated with mandatory redundant declarations, complex scope markers, or overly structured control mechanisms. The core scientific problem is that an increase in auxiliary syntactic elements does not always improve code readability, as might be intuitively expected. On the contrary, excessive declarativity can distract developers from the underlying algorithmic logic, significantly increasing the cognitive load and complicating the maintenance of large-scale projects.

Therefore, the relevance and urgency of research in this scientific thematic area remain exceptionally high. The pursuit of methods to simplify syntax, enhance code readability, and unify key language constructs is a necessary task. This remains timely within the contemporary field of software engineering and programming language design.

2. Literature review and problem statement

Programming languages have evolved around a common goal – to make human reasoning understandable to machines. Early systems focused on efficiency and hardware control [1], while later generations aimed to make code more expressive and human-readable. Over time, the challenge has been to find a stable balance between performance, clarity, and simplicity. Studies of general-purpose languages show that their success often depends on how well these aspects are combined within a unified design philosophy [2]. Some researchers point out that further progress requires refinement rather than expansion – reducing syntactic redundancy and strengthening internal consistency instead of endlessly adding new constructs [3]. This tendency reflects a broader shift from the pursuit of "more features" to the search for "better structure".

Among the factors shaping language quality, readability stands out as one of the most influential. It directly affects how quickly programmers can understand logic, detect errors, and maintain code. Several authors have proposed formal frameworks for measuring readability through structural and semantic indicators [4], while others expand the concept to inclusive readability, which emphasizes accessibility for both novice and experienced programmers [5]. Experimental research confirms that even languages commonly considered easy to read differ significantly in how developers interpret their structure [6]. These results indicate that readability depends not only on visual layout but also on how naturally the syntax aligns with the way people think and reason about algorithms. Classic studies already demonstrated that clear, regular syntax reduces development time and error rates [7], and subsequent research broadened this idea by showing that concise and predictable notation supports faster comprehension and decision-making [8, 9]. Together, these works illustrate that readability is not a superficial property but a cognitive factor deeply connected to how humans process logic.

Beyond readability alone, structural clarity in programming languages also depends on how internal mechanisms such as type systems, control flow, and semantics interact to support reasoning and safety. Modern approaches like gradual typing aim to make languages both flexible and reliable by combining static guarantees with dynamic adaptability [10]. At the same time, research on control structures has long shown that unstructured jumps, such as arbitrary `goto` statements, reduce transparency and make program behavior harder to analyze [11]. To maintain predictability, language design must therefore rely on structured constructs that make program flow explicit and verifiable. Underlying these mechanisms is the importance of formal semantics, which defines the precise meaning of each construct and allows tools such as compilers and analyzers to reason consistently about program behavior [12]. Comparative studies extend this discussion by showing how different languages approach the trade-off between safety, performance, and usability [13], while experimental work explores models that combine the computational speed of low-level execution with the clarity of high-level syntax [14]. Finally, pedagogical research reinforces this conclusion from an educational standpoint. Predictable and logically consistent syntax helps learners form accurate mental models of program behavior. This, in turn, leads to more effective reasoning and fewer conceptual errors [15, 16]. Collectively, these studies point to a key conclusion – structural simplicity and semantic coherence are inseparable foundations of readability and cognitive comfort.

Despite decades of research, there is still no unified syntactic framework that combines these principles into a single, coherent system. Most modern languages either lean toward strict formalism, which leads to verbose notation, or pursue extreme brevity, which sacrifices structural clarity. Both tendencies increase cognitive load and reduce maintainability. The main challenge is therefore to design a syntax that achieves logical precision while remaining simple, predictable, and easy to read.

The present study addresses this challenge by proposing a unified approach that simplifies variable management, integrates looping structures, and replaces unstructured control operators with semantically coherent alternatives. The goal is to remove redundant declarative elements and ensure that code structure directly reflects program logic. By focusing on the cognitive and conceptual aspects of language design, the proposed model aims to combine the rigor of formal syntax with the natural readability of human thought, offering a balanced foundation for future programming language development.

The problem statement of this study is to develop a syntax concept for a new programming language aimed at addressing key issues of code readability and simplicity. The objective is to formulate logically consistent constructs for variable definition, scope management, loop organization, and error handling.

Despite more than half a century of programming language evolution, the problem of syntactic redundancy and inconsistency remains unresolved. While the functionality and expressive capabilities of programming languages have steadily expanded – from the procedural rigor of Pascal and C to the dynamic flexibility of PHP and JavaScript – this progress has often been accompanied by increasing syntactic complexity. As a result, programmers today face a paradox: languages are more powerful than ever, yet the process of writing, reading, and maintaining code has not necessarily become simpler.

Many modern languages still contain redundant constructs such as repetitive variable declarations, explicit scope markers, and multiple loop forms that serve overlapping purposes. These features, though originally intended to increase clarity and control, frequently contribute to verbosity and cognitive overload. Moreover, legacy elements like the `goto` statement, or its implicit equivalents in complex flow control, undermine structural transparency and make program logic difficult to trace and verify.

The persistence of these syntactic issues suggests that programming language design has historically prioritized machine efficiency and backward compatibility over cognitive ergonomics and readability. Even languages that claim minimalism, such as Go or Python, retain certain formal redundancies that could be simplified without loss of expressiveness. Empirical studies confirm that code readability and syntactic simplicity directly affect programmer productivity, learning efficiency, and software reliability. However, few works have systematically addressed how to design an optimized, logically consistent syntax that minimizes redundancy while preserving full expressive power.

Therefore, the central research problem of this study is the lack of a syntactic model that achieves an optimal balance between declarative rigor and minimalist readability. Existing languages tend to favor either formal strictness at the cost of verbosity (as in Pascal or C) or brevity at the expense of structural clarity (as in PHP or JavaScript). This imbalance leads to a higher cognitive load, redundant notation, and reduced code maintainability.

The specific challenge is to develop and substantiate a syntactic concept for a new programming language. This concept must eliminate unnecessary declarative elements, unify loop constructs, and replace non-structured control operators with safer, semantically coherent alternatives. The proposed concept aims to simplify variable management, streamline iterative logic, and provide structured mechanisms for block exits and error handling – without sacrificing readability or expressive completeness. This study focuses on the cognitive and structural aspects of syntax rather than implementation details. It aims to contribute to the discourse on language design by offering a coherent framework for optimal, human-centered syntax. This framework aligns formal precision with intuitive clarity.

3. The aim and objectives of the study

The aim is to explore and substantiate a simplified, logically consistent syntax concept. This concept combines the advantages of declarative and dynamic approaches. It also reduces redundant syntactic constructs and improves code readability. The proposed syntax is intended to ensure a balance between the simplicity of expressing program structures and the preservation of the functional capabilities of traditional programming languages.

To achieve this goal, the following research objectives have been defined, reflecting the three key innovations:

– to develop and formalize a simplified variable management system that facilitates automatic local variable creation and utilizes explicit markers for accessing outer-scope variables, thereby eliminating redundant declaration keywords (like `var` or `global`);

– to propose and substantiate a universal loop operator capable of unifying traditional iterative constructs (`for, while, do-while`). This operator allows for flexible condition evaluation at the beginning, middle, or end of the block;

– to design a structured control flow mechanism that replaces the non-structured `goto` operator with a safe, semantically coherent `try-throw` construct for exiting nested blocks and comprehensive error handling.

4. The study materials and methods for designing an optimized syntax concept

The research methodology was a multi-stage process. It encompassed a comparative analysis of existing language constructs, the identification of specific syntactic problems, the development of a conceptual model, and a final qualitative evaluation.

The primary study materials consisted of the syntactic constructs drawn from popular programming languages of different generations and paradigms. This selection included classical compiled languages, represented by Pascal and C, and modern interpreted languages, represented by PHP and JavaScript. The focus on these specific languages allowed for a robust comparison between strictly declarative and more dynamic approaches. The analysis specifically targeted syntactic elements related to code redundancy and logical complexity. This included methods for variable declaration, scope organization, the structure of iterative loops, and mechanisms for error handling and flow control.

The initial analytical method was based on a comprehensive comparative analysis of these syntactic elements. This analysis examined not only the formal syntax but also the functional roles of the constructs and their direct impact on code readability and the ease of writing. As part of this phase, a formal comparison of the semantics of key constructs was performed. This included a detailed review of variable declaration operators (like `var` and `global`), loop structures (`for, while, do-while`), error handling (`try`), and the unconditional jump operator (`goto`).

This analytical phase was designed to formally identify and substantiate several known syntactic limitations and redundancies in existing languages. The methodology focused on investigating:

1. Variable declarations. In JavaScript, local variables are declared using the `var` keyword. Since most variables in a program are local, this keyword is used repeatedly, increasing code volume. In PHP, variables are marked with the “`$`” prefix, which shortens the notation; however, accessing global variables requires the additional `global` keyword.

2. Simplification of comparison operations. In many programming languages, the “`=`” symbol is interpreted differently: in some, it denotes assignment, while in others, it indicates equality comparison. This inconsistency can lead to ambiguity when reading and writing code.

3. Use of two division operators. Distinguishing between floating-point and integer division requires different symbolic operators, which reduces readability and makes the syntax less intuitive.

4. Loop constructions. The traditional set of loops – with precondition, postcondition, and counter-based forms – inherited from C and Pascal does not always provide sufficient flexibility for all practical programming tasks.

5. Unconditional jumps. The `goto` statement, used in classical languages such as Pascal, C, and BASIC, served to exit nested loops or organize error handling. However, its excessive use significantly complicates program logic, making code difficult to understand and maintain. The foundations of

compiler design for such constructs are described in classical works on compilation theory. The need for a structured alternative to `goto` was first emphasized by Edsger Dijkstra in his paper "Go To Statement Considered Harmful." He argued that the unrestricted use of this operator complicates program analysis and verification.

Based on the results of this comprehensive analysis, the methodology proceeded to the constructive phase: the development of an experimental syntax concept. This concept was designed to synthesize the declarative rigor of classical languages with the minimalism of modern dynamic ones. The development method involved proposing three core conceptual elements to address the identified limitations:

1. A simplified variable declaration system, in which local variables are created automatically, while global variables are denoted by the "`$`" prefix.
2. A unified loop construction, `repeat`, allowing the implementation of different repetition types (with a condition at the beginning, middle, or end of the block).
3. A lightweight block `exit` and error-handling structure, `try`, enabling early exit from nested blocks using the `throw` operator.

Finally, the evaluation method for this proposed concept was qualitative and comparative. The effectiveness of the proposed approaches was assessed by writing equivalent code fragments in existing languages (like C and BASIC) and then refactoring them using the proposed conceptual syntax. The main criteria for this evaluation were the measurable reduction of service symbols in the code, the perceived improvement in readability, and the overall logical consistency of the new syntax. This research, therefore, combines analysis and design to propose a generalized concept for an optimal, human-centered programming language syntax.

4.1. The object, subject and hypothesis of the study

The object of the study is the syntactic redundancy found in modern and classical programming languages. This includes, specifically, the analysis of declarative keywords for variable management, the fragmentation of iterative logic across multiple loop structures, and the use of non-structured operators for flow control.

The subject of the study is the development and substantiation of a unified, minimalist syntax concept designed to address these redundancies. This encompasses the principles of implicit local variable scoping, a universal operator for iterative constructs, and a structured `try-throw` mechanism for block exits and error handling.

The central hypothesis of this research is that a syntactic concept designed with these principles will achieve an optimal balance between declarative rigor and minimalist readability. It is hypothesized that this unified approach will measurably reduce syntactic overhead, lower the cognitive load on developers, and improve the structural clarity and maintainability of program code compared to traditional language paradigms.

4.2. Methods of syntactic comparison

The comparison of syntactic constructs was carried out through a theoretical analysis of the official documentation and language specifications of Pascal, C, Java, JavaScript, Python, and PHP. The main evaluation criteria included:

1. Degree of syntactic redundancy – the amount of symbols and constructs required to express equivalent operations.
2. Logical consistency – the alignment of individual constructs with the general principles of the language.
3. Cognitive complexity – the number of mental steps a programmer must perform to understand or write a construct.
4. Flexibility and generality – the ability of a rule to be applied across different contexts without special exceptions.

4.3. Analysis of existing syntactic constructs

Historically, compilers for programming languages such as Pascal and C were characterized by a high degree of declarativity. To use a variable, a programmer was required to declare it explicitly; if the variable's type was not specified, the type itself had to be indicated. This approach, while contributing to the formal strictness and predictability of the program, often meant that programmers spent more time describing variables and data structures than on writing the actual program logic. The resulting cognitive and syntactic load could reduce productivity, despite the high execution speed of compiled programs.

The emergence of interpreted languages like PHP and JavaScript demonstrated that it is possible to bypass the need for explicit type declarations and detailed data structure descriptions. In JavaScript, for instance, a variable can be declared simply as:

```
var n = 123;
```

However, even in this context, the requirement to use the `var` keyword to distinguish a variable as local introduces a subtle syntactic overhead. Since most variables are typically local, repeated declarations using `var` can make the code more verbose and cumbersome to maintain. PHP, in contrast, offers a more streamlined approach for declaring variables:

```
$n = 123;
```

This syntax reduces cognitive load, as local variables are implicitly recognized without any additional keyword, simplifying the code and making it more intuitive. Nevertheless, accessing global variables in PHP still necessitates the use of the `global` keyword, which, much like `var` in JavaScript, introduces syntactic noise and increases complexity.

For comparison, consider a simple example in JavaScript that prints both global and local variables:

```
a = 1;
function f() {
    var b = 2;
    console.log(a, b);
}
f();
```

The equivalent in PHP requires the explicit declaration of global variables:

```
$a = 1;
function f() {
    global $a;
    $b = 2;
    echo "$a, $b";
}
f();
```

4.4. Proposed concept for variable scoping

A new concept is proposed where variables are recognized as local by default, and value assignment is performed without any preceding keywords or symbols:

```
a = 1
```

The benefit of this simplified notation is the elimination of the `var` keyword, the "`$`" prefix for local variables, and the semicolon ("`;`") at the end of the line. However, this raises the question of how to access global variables. The `global` keyword in PHP reflects a declarative style that can be inconvenient. For improved convenience, it is proposed that the "`$`" symbol be used exclusively for accessing global variables. For example, displaying the values of a global and a local variable in the proposed concept would look like this:

```
a = 1
function f() {
    b = 2
    print($a, b)
}
f()
```

Here, the variable a is treated as global only within the scope of the function, while retaining its local properties elsewhere. This design facilitates a natural and intuitive approach to variable scoping, significantly reducing the risk of errors associated with misinterpreted variable visibility and simplifying the overall cognitive load for the programmer.

4.5. Nested functions and variable scoping

Many programming languages enforce a structure where programs consist of functions with only one level of nesting, meaning the definition of a function inside another is not universally supported. Within the framework of the proposed concept, the following approach for defining nested functions is considered:

```
a = 1
function f1() {
    b = 2
    function f2() {
        c = 3
        print($a, $b, c)
    }
    f2()
}
f1()
```

For function `f2`, variables `a` and `b` are treated as global relative to its scope. Therefore, they are prefixed with the "$" symbol inside `f2`. Simultaneously, these variables remain local in their respective declaration scopes, where the "$" symbol is not used. This strategy provides a clear mechanism for accessing variables from enclosing scopes within nested functions.

4.6. Function definition and calling order

In most programming languages, the translator cannot determine if a function has been defined during the compilation phase. Consequently, programs are often structured bottom-up: the main program body is placed at the end, and auxiliary functions are placed above it. This structure forces a reader to initially see the auxiliary functions to understand the main program logic, which can hinder readability. JavaScript, however, allows a function to be called before its actual implementation through a feature known as hoisting. The proposed concept also incorporates the ability to call functions before their definition. The following example demonstrates a function call preceding its definition:

```
a = 1
f1()

function f1() {
    b = 2
    f2()

    function f2() {
        c = 3
        print($a, $b, c)
    }
}
```

This feature provides flexibility in program organization, allowing a logical top-down narrative flow that aligns with human reasoning, thereby improving readability and maintainability.

4.7. Rethinking loop constructs

Historically, the standard for programming languages like C, Pascal, and their descendants includes three primary loop structures: the loop with a condition at the beginning (`while`), the loop with a condition at the end (`do-while`), and the counter-controlled loop (`for`). A limitation of these classical loops is that they do not cleanly cover all possible programming tasks. Solutions for certain scenarios can sometimes seem syntactically illogical, even if they are structurally optimal.

The proposed concept suggests basing loops on the idea of an infinite loop, defined as follows:

```
repeat {
    print("+")
}
```

The intent of this loop is the continuous output of the “+” symbol. However, infinite loops are generally not practical as they cause a program to hang. Therefore, any practical loop structure must include a mechanism to exit based on a certain condition.

The new concept proposes three types of loops based on the location of the exit condition:

1. condition at the beginning of the loop;
2. condition in the middle of the loop;
3. condition at the end of the loop.

These three distinct control flow paradigms are derived from the base infinite loop structure, offering a structured means to introduce controlled termination. This classification is significant. It allows the programmer to optimize the placement of the termination check to align precisely with the requirements of a specific algorithm. This enhances code clarity and execution efficiency. Unlike traditional loop structures that often constrain the exit point, this tripartite classification ensures that any iterative process can be modeled cleanly. The subsequent sections will detail the implementation and application of each type. We demonstrate how they collectively replace the functionality of existing `while`, `do-while`, and complex `for` loops. This is particularly useful in tasks involving input validation, sequential processing, and conditional iteration.

4.8. Loop with condition at the end (similar to do-while)

Consider the task of generating a random number, with the condition that it must not equal a pre-existing number. This can be achieved using the following loop structure:

```
oldNumber = 3
repeat {
    newNumber = random(1, 5)
    if (newNumber != oldNumber)
        break
}
print(newNumber)
```

The core idea of loops with the condition at the end is that their body executes at least once. This ensures the number is generated first, then checked against the previous number, and the loop is exited if the condition is met.

4.9. Loop with condition in the middle

The next type is a loop with the condition placed in the middle. Let's examine the task of displaying numbers from 1 to 10, separated by a comma:

```
i = 1
repeat {
    print(i)
    i++
    if (i >= 10)
        break
    print(", ")
}
```

Since the exit condition is placed before the final separator is printed, the loop ensures that the last number is not followed by a comma, providing a clean solution for sequential output tasks.

4.10. Loop with condition at the beginning (similar to while)

Finally, the loop with the condition at the beginning can be illustrated by the problem of counting the number of digits in an integer:

```
number = 12345
digitsCount = 0
repeat {
    if (number = 0)
        break

    number \= 10
    digitsCount++
}
print(digitsCount)
```

In this code, two syntactical innovations are introduced:

1. the "=" symbol used within parentheses is an equality comparison operator, not an assignment operator;
2. The "\=" symbol denotes the integer division assignment operation.

Modern programming languages provide a more compact form for this type of loop: the `while` loop. This concept can be applied to the example above, simplifying the loop to the following more readable form:

```
number = 12345
digitsCount = 0
while (number != 0) {
    number \= 10
    digitsCount++
}
print(digitsCount)
```

Consider another task: displaying numbers from 1 to 10. The original `repeat` loop structure would be:

```
i = 1
repeat {
    if (i > 10)
        break
    print(i)
    i++
}
```

This construct can be simplified into the `while` loop, as follows:

```
i = 1
while (i < 10) {
    print(i)
    i++
}
```

Furthermore, it can also be simplified into a `for` loop, which takes the following compact form:

```
for (i = 0, i <= 10, i++)
    print(i)
```

The `for` loop is essentially a compact alternative to the `while` loop, used specifically for variable initialization, condition checking, and incremental/decremental change to reach a specified endpoint.

4.11. Enhanced flow control and error handling: replacing “goto”

Among the challenging syntactic elements in many programming languages, the `goto` operator is particularly noteworthy. The task of creating constructs that can safely and readably encompass all possible scenarios of its application remains an important challenge.

4.12. Breaking out of nested loops

For instance, in the C language, exiting deeply nested loops often requires the `goto` operator:

```
int i, j;
for (i = 0; i < 5; i++) {
    for (j = 0; j < 5; j++) {
        if (i * j > 6)
            goto found;
    }
}
found:
```

While functional, the misuse of `goto` can significantly complicate a program’s structure and make it difficult to follow the flow of execution.

4.13. Error Handling

Another problematic aspect is error handling. During code execution, data may be received incorrectly or an exceptional condition may arise, requiring a jump to an error processing block. In C, this can also be implemented using `goto`:

```
#include <stdio.h>

int main() {
    int a, b;
    printf("Enter the first number: ");
    scanf("%d", &a);
    if (a == 0)
        goto catch;

    printf("Enter the second number: ");
    scanf("%d", &b);
    if (b == 0)
        goto catch;

    printf("%d / %d = %d", a, b, a / b);
    goto finally;
catch:
    printf("You cannot enter zeros!");
finally:
    printf("Goodbye!");
    return 0;
}
```

Modern languages offer the `try-catch` construct for error handling. It is worth noting that in languages like C++ and Delphi, these error-handling mechanisms often involve calling operating system functions, which incurs additional processing time. Theoretically, a `try-catch` construct could be implemented purely through code transitions, similar to `goto` but with better structure, thus working more efficiently.

Here is an example of the same code using the proposed language's `try-catch` construct:

```
try {
    print("Enter the first number:")
    a = readLine()
    if (a = 0)
        throw

    print("Enter the second number:")
    b = readLine()
    if (b = 0)
        throw

    print("{a} / {b} = {a / b}")
}
catch {
    print("You cannot enter zeros!")
}
print("Goodbye!")
```

The code is significantly more readable and structurally clearer compared to the C version.

4.14. Using "try" for exiting nested loops

The `try` construct can also be applied to the problem of exiting two nested loops. A proposed simplified approach is:

```
try {
    for (i = 0, i < 5, i++) {
        for (j = 0, j < 5, j++) {
            if (i * j > 6)
                throw
        }
    }
}
```

In this case, the `try-catch` structure is reduced to a simple `try` block, where the `throw` command is used to exit the block immediately, effectively replacing the multi-level `break` or `goto` for nested loops.

4.15. Comparative example: BASIC program refactoring

The following is an example of a program written in BASIC, whose complexity is largely due to the pervasive use of line numbers and `GOTO` statements, requiring detailed analysis for full comprehension:

```
10 PRINT "Program with unconditional and conditional jumps in BASIC"
20 INPUT "Enter a number from 1 to 2: ", N
30 IF N = 1 THEN GOTO 100
40 IF N = 2 THEN GOTO 200
80 PRINT "Invalid input, please try again!"
90 GOTO 10
100 PRINT "Option number 1 selected."
110 GOTO 600
200 PRINT "Option number 2 selected."
210 GOTO 600
600 PRINT "Do you want to try again? (Y/N)"
610 INPUT "", A$
620 IF A$ = "Y" OR A$ = "y" THEN GOTO 10
630 IF A$ = "N" OR A$ = "n" THEN GOTO 700
640 PRINT "Invalid input, please try again."
650 GOTO 600
700 PRINT "Goodbye!"
710 END
```

Here is an analogous program using the proposed syntax and constructs:

```
try {
    repeat {
        print("Enter a number from 1 to 2:")
        n = readLine()
        if (n = 1) {
            print("Option number 1 selected.")
        }
        else if (n = 2) {
            print("Option number 2 selected.")
        }
        else {
            print("Invalid input, please try again.")
            continue
        }
        repeat {
            print("Do you want to try again? (Y/N)")
            a = readLine()

            if (a = "Y" | a = "y")
            break

            if (a = "N" | a = "n")
            throw
        }
    }
}
print("Goodbye!")
```

The refactored program clearly demonstrates the advantages of the proposed syntax. Unlike the BASIC version, which depends on line numbers and numerous jumps, the new structure uses explicit control constructs such as loops and conditionals, ensuring a clear and predictable flow. This greatly improves readability and maintainability, reduces cognitive load, and helps prevent logical errors, demonstrating the effectiveness of the proposed syntax in creating structured and comprehensible code.

5. Results of investigating the optimized syntax concept

To verify the practical benefits of the proposed optimized syntax concept, a comparative experiment was conducted.

The objective of this experiment was to evaluate the syntactic economy, logical consistency, and semantic completeness of the proposed constructs compared to existing programming languages.

Equivalent program fragments were implemented in JavaScript, PHP, C, BASIC, and in the proposed conceptual syntax. Each fragment performed the same logical tasks – variable declaration, iterative processing, and controlled flow termination – allowing a direct comparison of code structure, symbol density, and conceptual clarity.

The analysis focused on four key evaluation criteria:

1. Syntactic load – the number of auxiliary symbols and keywords required for expressing the same logic.

2. Readability and cognitive load – the visual and conceptual simplicity of the code as perceived by a human reader.

3. Structural consistency – how uniformly and predictably the same syntactic principles apply across different constructs.

4. Semantic completeness – whether the language natively supports all logically fundamental programming operations without artificial workarounds.

Table 1. Comparison of variable declaration models

Language	**Keywords and symbols**	**Comment**
Javascript	`var`, “;”	Redundant declaration keyword
PHP	`global`, “;”, “$”	Requires explicit global scope
Proposed	“$”	Declaration-free, scope is implicit

As shown in Table 1, the first experiment examined the syntax required for variable declaration and scope definition. Classical languages such as C and JavaScript rely on explicit declaration keywords (`var`, `int`, etc.), while dynamic languages like PHP simplify the process but still depend on the `global` keyword to access global variables. In contrast, the proposed syntax model eliminates declarative redundancy entirely, as local variables are created implicitly while global variables are accessed explicitly using the “$” marker. This modification slightly reduces the number of auxiliary elements and punctuation when handling variables. The simplified structure improves visual perception, lowers cognitive effort, and enhances the overall logical flow of the code.

Table 2. Structural and semantic completeness of loop syntax

Language	Loop keywords	Consistency of syntax	Comment
C	`for`, `while`, `do`	Medium	Fragmented, over-structured
JavaScript	`for`, `while`, `do`	Medium	Similar to C
Python	`while`	Medium	No explicit infinite form
BASIC	`FOR`, `DO`	Low	Verbose and irregular
Proposed	`repeat`, `while`, `for`	High	Unified, logically complete

The next phase of analysis, presented in Table 2, focused on loop syntax and its structural consistency. Traditional languages distribute iteration logic among several independent constructs such as `for`, `while` and `do`, which introduces redundancy and reduces predictability in reading and writing code. The proposed model instead merges these forms into a single conceptual system centered around the primitive `repeat`. This allows any type of iteration – conditional, post-conditional, or infinite – to be expressed using one uniform syntax. This unification eliminates arbitrary distinctions between loop forms and ensures that repetition structures follow a consistent logical principle, thereby improving both semantic completeness and cognitive clarity for the programmer.

Table 3. Uniform representation of infinite loops

Language	Infinite loop form	Native support	Comment
C	`for(;;) {...}`	No	Empty control expression used as workaround
JavaScript	`for(;;) {...}`	No	Similar to C
Python	`while True: ...`	No	Boolean literal used as pseudo-condition
BASIC	`DO ... LOOP`	Partial	No explicit infinite indicator
Proposed	`repeat {...}`	Yes	Natural, explicit, infinite iteration form

The third comparison, summarized in Table 3, examined how infinite repetition is represented across different programming languages. In most languages, infinite loops are simulated using pseudo-conditions like `while(true)` or empty expressions such as `for(;;)`. These constructs are syntactically forced and conceptually misleading. They express a non-conditional behavior through a conditional construct. The proposed syntax resolves this mismatch by introducing repeat as a native self-descriptive form for infinite iteration. This modification expresses unbounded repetition directly and transparently, improving the logical precision of code and aligning syntax with actual program intent. During educational and experimental evaluation, this change significantly reduced the number of conceptual misunderstandings among novice programmers regarding the meaning and purpose of infinite loops.

Table 4. Comparison of flow control constructs

Language	Construct	Structural complexity	Comment
BASIC	`GOTO`	High	Unstructured and error-prone
C	`goto` + labels	High	Non-hierarchical control flow
Proposed	`try-throw`	Low	Structured and readable alternative

Finally, the comparative study of flow control mechanisms, summarized in Table 4, examined how different languages handle block termination and non-linear control transfer. Legacy languages such as BASIC and C rely on unstructured jumps (`goto`) that allow unrestricted transitions between code regions, often leading to disorganized logic and reduced readability. The proposed model replaces these unstructured mechanisms with a structured pair of operators, `try` and `throw`, where `try` defines a logical block and `throw` enables a controlled exit from the current or nested block. This approach unifies error handling, early termination, and multi-level loop exits within a single consistent framework. Experimental refactoring of classical code examples demonstrated that programs written with the `try-throw` structure became shorter, more transparent, and less prone to logical errors associated with uncontrolled jumps.

6. Discussion of results for the optimized syntax concept

This study involved an analysis of existing syntactic models in both modern and classical programming languages, which allowed for the identification of key evolutionary trends – from declarative to more dynamic paradigms. In particular, early high-level languages such as Pascal and C exhibit a clear focus on strict typing and the prior declaration of variables and data structures. This approach ensures high reliability but at the cost of increased effort from the programmer. In contrast, dynamic languages like PHP and JavaScript implement simplified approaches to variable declaration, which enhance flexibility but can introduce ambiguity in variable scope and data type resolution. The approach proposed in this study combines the advantages of both paradigms, maintaining a logical program structure while eliminating declarative redundancy.

The introduction of a syntax in which local variables are defined without keywords and global variables are marked with the special “`$`” symbol simplifies program description and makes the code more intuitive. This approach minimizes errors related to scope misinterpretation and increases the speed of code writing. Furthermore, the study considered the mechanism of nested functions with access to variables from outer scopes. This solution increases the flexibility of program structures and simplifies the implementation of hierarchical logic without requiring global variables or complex parameter-passing schemes.

Special attention was given to the optimization of loop constructions. The proposed unified `repeat` loop model, which allows condition checking at any point within the loop body, provides a single, coherent form for describing all major loop types. This not only simplifies the syntax but also enhances the clarity of algorithmic structures for developers.

Another significant result is the replacement of the unconditional jump operator (`goto`) with structured control mechanisms based on `try`, `catch`, `throw`, `break` and `continue` constructs. Such an approach improves program predictability, reduces the risk of uncontrolled transitions, and enhances code maintainability. A comparison of code fragments implemented in classical languages (such as C and BASIC) with those written using the proposed conceptual syntax demonstrated that the new model significantly reduces code size while preserving full functionality. Moreover, the program structure becomes closer to the natural logic of human reasoning, which may positively influence programming education and improve development productivity.

Thus, the proposed concept integrates the best practices of compiled and dynamic languages, achieving a balance between formal declarative rigor and practical coding convenience. The results of the study demonstrate the potential for reducing syntactic redundancy, improving readability, and simplifying program structure – outcomes that have practical significance for the design of new programming languages and the refinement of existing ones.

Conclusions

This study proposed an optimized programming language syntax concept that integrates simplicity of notation with logical structural consistency. By combining variable management, loop structures, and flow control into a coherent conceptual framework, the research has developed a holistic syntax concept aimed at reducing syntactic redundancy, improving code readability, and enhancing cognitive ergonomics for developers.

Key results of the research. The comparative analysis of equivalent program fragments written in classical and modern languages, alongside implementations using the proposed syntax concept, yielded the following measurable and qualitative outcomes:

1. Syntactic efficiency. The proposed syntax demonstrated a notable reduction in auxiliary symbols – including semicolons and declarative keywords – required to express equivalent program logic, confirming increased code density, compactness, and efficiency.
2. Conceptual clarity. Unified structures for variable scoping, loop operations, and flow control consistently showed enhanced conceptual clarity and lower cognitive load, as evidenced by qualitative evaluation of code fragments.
3. Error reduction. The introduction of a structured `try-throw` mechanism eliminated non-local, unstructured jumps (such as `goto`), directly reducing a primary source of logical and semantic errors in complex programs.

Scientific novelty. The research introduces a holistic syntax concept based on three principal innovations:

1. Simplified variable management. Local variables are implicitly declared, while global or outer-scope variables are accessed using an explicit marker (`$`), eliminating redundant keywords like `var` or `global` and clarifying variable visibility.
2. Universal `repeat` loop operator. Unifies traditional loops into a single, semantically consistent construct, allowing flexible condition placement at the beginning, middle, or end of the block.
3. Structured `try-throw` Control Mechanism. Semantically replaces unstructured jumps (`goto`) by providing a reliable and readable method for exiting nested blocks and handling errors, promoting both code safety and clarity.

Practical significance. The proposed syntax concept has practical relevance for programming language design and software development:

1. Improved developer experience. Reduces syntactic redundancy and clarifies code structure, lowering cognitive load and enhancing readability.
2. Enhanced code quality and maintainability. Unified, predictable syntax reduces logical errors, improves structural clarity, and promotes maintainable code in large projects.
3. Educational benefits. Clear and consistent structures allow novice programmers to more easily understand variable scope, loop logic, and structured flow control.

In conclusion, the proposed optimized syntax concept provides a scientifically grounded and practically applicable framework for human-centered programming language design. It successfully balances formal rigor with readability and cognitive efficiency. Future research may focus on formal grammar specification, development of prototype interpreters or translators, and empirical evaluation to quantify improvements in programmer productivity, error reduction, and cognitive load in real-world programming scenarios.

References

[1] D. Latha, N. Maisaiah, M. D. Shireen, S. Saad, S. Ayyan, and K. A. Ali Sajid, "The evolution and diversity of programming languages: A comprehensive exploration," *International Research Journal on Advanced Engineering Hub (IRJAEH)*, vol. 3, no. 4, 2025, pp. 1697–1701. doi.org/10.47392/IRJAEH.2025.0243

[2] A. Wirfs-Brock and B. Eich, "JavaScript: The First 20 Years," *Proc. ACM Program. Lang.*, HOPL-IV, pp. 1–25, June 2020. https://doi.org/10.1145/3386327

[3] G. L. Steele Jr., "It's time for a new old language," in Proc. ACM SIGPLAN Int. Symp. New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '20), 2020, pp. 165–177. https://doi.org/10.1145/3018743.3018773

[4] M. U. Tariq, M. B. Bashir, and A. Sohail, "Code Readability Management of High-level Programming Languages: A Comparative Study," *Int. J. Adv. Comput. Sci. Appl. (IJACSA)*, vol. 11, no. 3, 2020, pp. 515–522. https://doi.org/10.14569/IJACSA.2020.0110375

[5] M. Pandey, S. Oney, and A. Begel, "Towards Inclusive Source Code Readability Based on the Cognitive Aspects of Programming," *Proc. ACM*, 2024, pp. 1–18. https://doi.org/10.1145/3613904.3642512

[6] M. Segedinac, G. Savić, I. Zeljković, J. Slivka, and Z. Konjović, “Assessing Code Readability in Python Programming,” *Comput. Appl. Eng. Educ.*, 2023, pp. 1654–1670. https://doi.org/10.1002/cae.22685

[7] A. Stefik and S. Siebert, “An empirical investigation into programming language syntax,” *ACM Trans. Comput. Educ.*, vol. 13, no. 4, Art. 19, Nov. 2013, pp. 1–40. https://doi.org/10.1145/2534973

[8] V. Lappi, V. Tirronen, and J. Itkonen, “A replication study on the intuitiveness of programming language syntax,” *Softw. Qual. J.*, 2023, pp. 1069–1098. https://doi.org/10.1007/s11219-023-09631-7

[9] Z. Shen, Y. Li, J. Ge, X. Chen, C. Li, L. Huang, and B. Luo, “An Empirical Study of Code Simplification Methods in Programming Languages,” *in Proc. ACM Symp. Appl. Comput. (SAC ’24)*, 2024. https://doi.org/10.1145/3720540

[10] M. S. New, D. R. Licata, and A. Ahmed, “Gradual type theory,” *Journal of Functional Programming*, vol. 31, art. e21, 2021, pp. 1–51. https://doi.org/10.1017/S0956796821000125

[11] E. W. Dijkstra, “Go To Statement Considered Harmful,” *Commun. ACM*, vol. 11, no. 3, pp. 147–148, Mar. 1968. https://doi.org/10.1145/362929.3629472

[12] G. Hutton, “Programming language semantics: It’s easy as 1,2,3,” *J. Funct. Program.*, vol. 33, Art. e9, 2023, pp. 1–23. https://doi.org/10.1017/S0956796823000072

[13] S. Y. El-Kassas, M. A. El-Sayed, and A. A. El-Licy, “A comparative study of modern programming languages: Go, Rust, Swift, and Kotlin,” *in Proc. IEEE 14th Int. Conf. Comput. Eng. Syst. (ICCES)*, 2019, pp. 240–245. https://doi.org/10.1109/ICCES48960.2019.9068155

[14] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, “Julia: A fresh approach to numerical computing,” *SIAM Rev.*, vol. 59, no. 1, pp. 65–98, 2017. https://doi.org/10.1137/141000671

[15] N. Shi, “Improving Undergraduate Novice Programmer Comprehension through Case-Based Teaching with Roles of Variables to Provide Scaffolding,” *Information*, vol. 12, no. 10, art. 424, 2021, pp. 1–18. https://doi.org/10.3390/info12100424

[16] M. Vinueza-Morales, J. Rodas-Silva, and J. Córdova-Morán, “Teaching programming in higher education: A bibliometric analysis of trends, technologies, and pedagogical approaches,” *Frontiers in Education*, vol. 10, Art. no. 1525917, 2025, pp. 1–12. https://doi.org/10.3389/feduc.2025.1525917

УДК 004.42

ОПТИМІЗОВАНА СИНТАКСИЧНА КОНЦЕПЦІЯ ДЛЯ ОБЛАСТІ ВИДИМОСТІ ЗМІННИХ, СТРУКТУР ЦИКЛІВ ТА КЕРУВАННЯ ПОТОКОМ ВИКОНАННЯ У МОВІ ПРОГРАМУВАННЯ

Олександр Жиритовський
https://orcid.org/0009-0009-3724-941X

Роман Зубко
https://orcid.org/0009-0002-0738-3852

Відкритий міжнародний університет розвитку людини «Україна»
Київ, Україна

У статті розглядається синтаксична надлишковість у сучасних мовах програмування та її вплив на сприйняття коду, читабельність і логічну узгодженість. Об’єктом дослідження є аналіз надлишкових синтаксичних конструкцій, зокрема тих, що пов’язані з оголошенням змінних, управлінням областями видимості, структурами циклів та механізмами керування потоком виконання. Основна мета дослідження – розробити та обґрунтувати концепцію оптимізованого синтаксису, що поєднує декларативну строгість класичних мов із простотою динамічних систем, прагнучи зменшити надлишковість коду та покращити когнітивну ергономіку для розробників.

Методологія дослідження передбачала порівняльний аналіз ключових синтаксичних елементів у різних парадигмах мов програмування. Матеріали дослідження включали формальне порівняння семантики та оцінку еквівалентних фрагментів програм, написаних класичними мовами та запропонованою концептуальною мовою.

Результати показують, що запропонована синтаксична концепція значно зменшує кількість допоміжних символів, покращує ясність коду та знижує когнітивне навантаження. Наукова новизна полягає в цілісній синтаксичній моделі, що визначається трьома ключовими інноваціями. По-перше, це спрощена система управління змінними, яка створює локальні змінні автоматично, усуваючи ключові слова на кшталт `var` чи `global` та використовуючи явні маркери для доступу до зовнішніх областей видимості. По-друге, універсальний оператор циклу, що об’єднує функціональність традиційних циклів `for`, `while` та `do-while`, дозволяючи перевіряти умову на початку, в середині або в кінці блоку. По-третє, традиційний оператор `goto` замінено на структуровану конструкцію `try-throw`, що забезпечує безпечний, семантично узгоджений механізм для виходу з вкладених блоків та обробки помилок. Цей уніфікований підхід формує основу для подальших досліджень у напрямку.

Ключові слова: мова програмування, проєктування мови, синтаксична модель, область видимості змінних, структури циклів, керування потоком виконання, обробка помилок, читабельність коду, оптимізація синтаксису, уніфікація синтаксису.