Development of a Multilingual Programming Language Using Rust

JEMIMAH NATHANIEL
School of Computing, Joensuu campus
University of Eastern Finland
FINLAND

MOWETE FUMNANYA KAVAN
Department of Computer and Information Sciences
Covenant University, NIGERIA

Abstract: - The increasing globalization of technology necessitates the development of programming languages that accommodate diverse linguistic backgrounds. The entrenchment of the English Language as the lingua franca of computing has had dire consequences on determining the requirements for entry into the industry often barring qualified candidates because they do not possess adequate knowledge about the language—when there is no fundamental technical reason for doing so, as computers do not "care" about their language of instruction. To combat this, this paper designed and implemented a multilingual programming language using Rust called Lingo, it is an interpreter that translates source code for a multilingual programming language into executable programs which should be able to be natively executed on users' computers. Lingo aimed at enhancing accessibility for non-English speakers, the languages supported by this system include English, Francia's (French), Hausa, and Yoruba, with keywords, error messages, and functions available in each of the four of them. Lingo is inspired by Lisp, with modified syntax to facilitate multilingual interaction. The results of this study show that Lingo is a viable alternative to English-based programming languages and can be used to introduce programming to a wider audience, as well as provide localized error messages to help users understand and fix any issues with their code—is a step in the right direction to bridge this gap and make programming more accessible to a wider audience. The findings indicate that a multilingual approach can significantly improve the user experience and broaden participation in programming, ultimately contributing to a more diverse and innovative tech community.

Key-Words: - multilingual programming language, programming language, programming, bilingual, English-based programming languages.

Received: April 8, 2025. Revised: June 26, 2025. Accepted: June 30, 2025. Published: July 3, 2025.

1 Introduction

Digital computers form an integral part of our 21st-century lives as humans, coming a long way from being relegated to giant, ventilated rooms a` la the ENIAC [1], to being the primary means of communication for most humans on the planet—mobile phones [2]. Programming languages, therefore, have reached an unprecedented level of importance as they are the primary means of interfacing with computers—effectively "telling them what to do". As a result, they have adopted a break-neck pace of development, with new languages coming out every year with brand-new features like generics—which is when algorithms are not written with specific types but given a "to-be-specified-later" type—introduced by ML in 1973, borrow-checking—a technique used to manage memory and ensure safety without using a traditional garbage collector—demonstrated by [3] with Rust, and goroutines, a form of green threading (a thread scheduled by a program's runtime as opposed to the actual operating system) popularised by Go [4]. Due to this development pace, existing programming languages are varied in a wide range of attributes, with everything from the level of abstraction from the hardware to how much emphasis is placed on manual memory management, but one attribute they mostly share is the use of the English language vocabulary as a base—keywords are common concatenations of English words, error messages are in English and, more often than not, the language is designed to be written on a US-layout keyboard [5].

The current state of affairs is that English is entrenched as the lingua franca (of computing because of its origins in North American and European institutions, even though no technical limitations are keeping it so —it is all bits and bytes to the computer at the end [6].This heavy use of English leads to an uneven gradient of entry into the industry. For a new programmer to make progress, they have no choice but to learn English because even though programming languages are formal grammars disconnected from their origins [7], the majority of programming languages are based on English keywords. Apart from these, error messages and online discussions are predominantly in English [8],which provides an invaluable resource to developers everywhere [9], and would not help if they just managed to memorize the keywords and

kept working in their native language. Providing alternate means to program can help mitigate the effects of this and lower the barrier of entry, for example, by decoupling programming from textual representations and making it wholly visual-based [10] or translating language keywords and making localized versions of popular programming languages. Some existing languages have attempted to tackle this, but potential problems arise with this approach—these visual languages break down for large codebases as navigation becomes complex [11], and unofficial translators often have few resources to maintain the language.

This study hypothesizes that a well-designed programming language that caters to non-English languages is created—with localized keywords, error messages, and documentation—along with a forum that encourages questions to be asked and answered in the native language would benefit a vast number of people that otherwise would not be able to get into computing. In addition, it would encourage collaboration between people with whom the language barrier would otherwise separate. The tools developed as a result of this study have strict constraints to ensure their effectiveness in breaking down the language barrier in this industry. The language is targeted toward general-purpose computer programming and does not make any priorities towards producing the best performance. The primary audience for the language should be people with little to medium experience with programming who do not have a solid grasp of English, for example, secondary school and first-year university students.

2 Related Works

Digital computers are a marvel of modern technology whose continued presence continues to have a gigantic impact on our collective lives. We owe this thanks to the numerous layers of systems that work together beneath the surface to deliver seamless experiences. At a high level, we can subdivide these systems into two categories—hardware and software. Many parts come into play in the software aspect to cooperate and serve the user. The operating system manages resources, and applications carry out activities on behalf of the user. Some particular applications, called utilities, help manage other software, exist. Some utilities, such as programming tools and libraries, help create, test, and maintain applications the general public uses daily. The focus of this study is on the development of a multilingual programming language, which necessitates the review of some concepts relevant to the field and other attempted solutions to the problem.

2.1 Review of Programming Languages

Programming languages are notation systems for writing computer programs [12]. They consist of all the rules that govern how they can be structured—their syntax, what they mean—their semantics, and what values and operations can be performed—their computational models. Programming languages are heavily influenced by the architecture of the computer they are designed to run on—[13] defines an architecture as a description of the structure of the separate components of a computer—and in turn, dictate how standard programs are built for systems of that architecture. This network of cause and effect leads to a broad spectrum and variety of languages, each with unique features, ease of use levels, tooling, benefits, and downsides. Low-level programming languages are languages that provide little-to-no abstraction—the generalization of specific, concrete implementation details that are not relevant [14]over a computer's instruction set. This leads to commands in the language being very similar in structure to the actual instructions executed by the processor. They are used when programs need to be run as fast as possible, with the downside being that it is hard to write and maintain [15]. Examples include assembly language, which is transformed directly into sequences of machine instructions and data according to [16], and languages with access to low-level functions like C [17].

High-level programming languages have a high level of abstraction from the computer's implementation details, often preferring natural language constructs to aid development and ease of use and automatically performing operations such as memory management. They provide a simpler frontend for writing programs – preferring to use constructs like variables and subroutines that hide the details of registers, opcodes, and memory allocation. These are commonly used when the ease of use of the program development process is more critical than any theoretical benefits that could be gotten from a more performant language. Some examples are the Lisp family of languages, the third-oldest high-level programming language still in common use as it was originally made in 1960 [18], C#, and Java, among others.

2.2 Review of the Language Barrier

Even though the most popular software applications have begun to move away from English-centrism and are becoming more accessible and internationalized [19], a startling majority of programming languages (and their associated

libraries) are not—they have stuck to their ASCII, English-based roots. Over a third of all programming languages have been developed in English-speaking countries [20]. Even languages not developed in one are still overwhelmingly English-based for example, Figure 1 shows Lua code that was made in Brazil (and named after the moon in Portuguese) but still uses English keywords.

```lua
local grid = {
  { 11, 12, 13 },
  { 21, 22, 23 },
  { 31, 32, 33 }
}

for y, row in pairs(grid) do
  for x, value in pairs(row) do
    print(x, y, value)
  end
end
```

Figure 1: Sample Lua code, with English keywords

Theoretically, there is no reason it should be so at a hardware level. According to [6], in an article aptly named Coding Is for Everyone—as Long as You Speak English, argues that computers have no technical limitations that make them any better at parsing English than any other language and could easily do it just as effectively if we used anything from emoji to Cyrillic. [6] also hypothesizes that the community around a language is the most important, practical aspect that keeps the status quo in place, and it rings true. Figure 2. shows a computer's steps when it builds an executable from a source program. There is no reason English has to be the language in the tokenization or parsing steps (also referred to as lexical and syntax analysis, respectively).

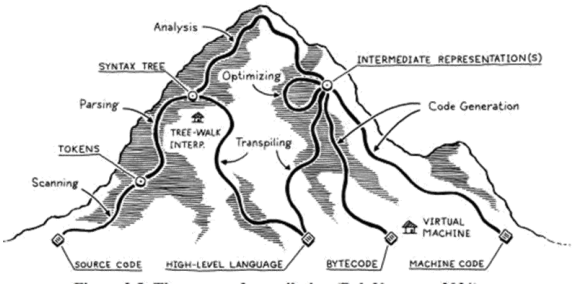

Figure 2: The stages of compilation [21]

Studies have shown a direct link between English proficiency and computer programming ability, up to the university level [22], which makes this entrenchment of English a major problem in this industry. Because learning programming languages primarily involves memorizing keywords, recognizing error messages and dealing with them appropriately, and building the skill of searching for documentation when needed, gatekeeping everyone unable or unschooling to learn the language from the industry actively hurts it.

2.3 Review of Existing Approaches

A common approach to this problem is to swap out the keywords of a PL for more localized alternatives, but this is not very effective. This is how quite a few esoteric languages —languages made for humorous purposes, like Pikachu, which only has three keywords: pi, pika, and Pikachu—are out there. A programming language is more than just keywords however, error messages also matter as a means for the user to be told what is wrong with any program being written. Error messages are so important that 72% of respondents used Rust at work because it was "enjoyable to use" [23], Figure 3 shows an illustration of errors in rust.

```
error[E0382]: borrow of moved value: `x`
--> src/main.rs:7:20
  |
5 |     let mut x = Food{ s: 5u32 };
  |         ----- move occurs because `x` has type `Food`, which
does not implement the `Copy` trait
6 |     let y = x;
  |             - value moved here
7 |     println!("{}", x.s);
  |                    ^^^ value borrowed here after move
  |
  = note: this error originates in the macro
`$crate::format_args_nl` which comes from the expansion of the
macro `println` (in Nightly builds, run with -Z macro-backtrace
for more info)

For more information about this error, try `rustc --explain
E0382`.
```

Figure 3: An example of a comprehensive rust error - E0382

Unfortunately, most languages also lack the extensibility to change the error messages. Also, libraries and other resources (such as documentation) fall back to the original language, English, making it challenging to discover resources or any issues a user might have. Developer tooling is also a big deal as this is also how users primarily interact with any programming language—without a good way to utilize any features fully, users should turn away to more accessible tools. For example, the lack of high adoption of package managers by most C++ developers [24] is a pain point in the ecosystem that Rust's Cargo conveniently solves—along with taking care of dependency and project management. Similarly, any multilingual programming language should have to ensure users are not inundated with all the possibilities and give up.

2.3.1 Visual Programming Languages

These programming languages allow programs to be specified using graphical elements instead of traditional text. These are commonly used as introductory material to novice programmers as they reduce the potential of ill-formed programs resulting from syntactic errors [25]. A disadvantage, however, is that managing projects once they become too large becomes impractical with a visual PL instead of a textual one. Some popular ones are:
Blockly: Blockly is an open-source Google for creating browser-based visual programming languages. It was created in 2011 by Neil Fraser, Quynh Neutron, Ellen Spertus, and Mark Friedman. It forms a basis for many educational tools such as Scratch and Code.org The reference implementation presents interlocking blocks that can generate code in Javascript, Dart, Python, and more. It has also been localized into over 100 languages [26] and supports RTL (right-to-left languages, such as Arabic and Hebrew).
Scratch: Scratch was initially created by Mitchel Resnick and Yasmin Kafaiin in 2003 (Maloney et al., 2004). Its purpose is to be an educational tool for 8-16-year-olds [27] It is available in more than 70 languages in most countries of the world [27]. The user uses various blocks which can be combined in the stage area to create various special effects (eg music, animations) and control a sprite–a two-dimensional fixed-size object composited with a background in a scene [28].

2.3.2 Non-English-based Programming Languages

These programming languages do not use syntactic elements inspired by the English language as shown in shown in Table 1. They achieve this by deriving the keywords from other languages like Russian and Latin or by doing away with keywords entirely, for example, APL: uses special graphical symbols to perform actions on multidimensional arrays [29]. Piet: uses bitmaps that look like abstract art of the ones that use traditional textual keywords [30], just not derived from English, some examples should be considered here:
Hapy: This is a simple language comprised of Hausa keywords that transpiles to Python [31], with a few syntactic changes, such as the inclusion of braces for scopes, as opposed to Python's whitespace, and semicolon line-endings. It lacks localized errors in Hausa and does not provide facilities for code to be shared and reused across projects.
Hedy: Hedy is an educational open-source Python subset. It is multilingual, supporting over 40 languages for the actual language and the UI of the editor application It also allows teachers to fully customize the students' experience [32]. A considerable emphasis is placed on ensuring the learning experience is tailored to newcomers to the language—the tutorial is subdivided into "levels," which lower the language barrier by starting with an elementary language and adding concepts and syntax as the learner progresses. This technique is referred to as "Gradual Programming" and is further expanded upon by [33] and [34]. Levels are also taken advantage of to produce gradual error messages [34] as code written in a lower level has simpler grammar and more limited options—which can be taken advantage of to produce more precise error messages. While admittedly very good, Hedy is fundamentally limited because it transpires to regular Python and is a subset of it. Even when written offline, there is no option to not write Hedy or to somehow interoperate with other people's code written in other PLs. Additionally, libraries you've written cannot be shared with others because Hedy aims to be a learning tool first and general-purpose language second.
Babylscript: This is a multilingual version of Javascript built by [35] that has multiple tokenizers, allowing objects and variables to have different names in different languages. This functionality is exposed for all the standard library methods, allowing people with different languages to get up and running quickly. Babylscript has the unique feature of allowing programmers to mix code from different vocabularies in a single project—enhancing collaboration. It also allows developers to extend their code to hook into the multilingual API exposed by the language and specify alternate names in other vocabularies. A significant downside is that it is modeled after Javascript, which, even though it is the most used language in the world, lacks the safety of static typing, leading to errors by beginners who have no idea what the language is doing [36]. Despite the multilingual benefits, this makes it a poor choice to recommend to someone completely new to programming. International Scheme: As a Lisp, Scheme is a very flexible and expressive language—allowing Scheme code to modify syntax directly. This has given rise to translations of schemes distributed as libraries, one of which is the International Scheme [37] - an open-source collection of translated keywords. Existing error

messages cannot be translated using the library, which is a big drawback. Apart from the ergonomics of the programming language itself, certain specific tools give languages an edge over others when it comes to overall developer happiness and experience. Some important ones should be evaluated to measure their impact.

Table 1: Comparison of the non-English-based programming languages

S/N	Name	Interface	Localisation	Libraries?
1.	Blockly	Graphical, website	Keywords	No
2.	Hedy	Text, CLI, website	Keywords, Errors	No
3.	Scratch	Graphical, website	Keywords	Yes
4.	Hapy	Text, CLI, website	Keywords	No
5.	International Scheme	Text, CLI	Keywords	Yes
6.	Babylscript	Text, CLI	Keywords, Functions	Yes
7.	**Proposed Solution**	**Text, CLI**	**Keywords, Functions, Errors**	**Yes**

2.3.3 Q&A Software

Q&A software facilitates discussions that aim to answer user-provided questions around a specific topic or inter-est. They commonly occur on or are an integral aspect of an internet forum—a discussion website where people communicate in written messages. They typically focus on a field or industry so that specialists have a place to meet and build knowledge in that domain. Some of the most popular Q&A sites are the Stack Exchange family of sites, the biggest of which are (Stack Exchange, 2024): Stack Overflow (https://stackoverflow.com/) which focuses on "professional and enthusiast programmers" (23m users). Super User (https://superuser.com/), targeted toward "computer enthusiasts and power users" (1.6m users). Ask Ubuntu (https://askubuntu.com/), for "Ubuntu users and developers" (1.5m users). These Q&A sites have a large amount of programming-related content flowing through them daily—more than 2400 questions are asked on Stack Overflow alone daily, which makes it an invaluable resource whenever developers get stuck with a problem relating to a piece of code. Any new programming language would benefit from being featured on a good quality Q&A site, but adapting them to a multilingual audience would be an arduous task as they have mostly been designed English-first—so far, since 2013, only three additional Stack Overflow sites have been made, for Spanish (https://es.stackoverflow.com/), Portuguese (https://pt.stackoverflow.com)

2.3.4 Build Automation Utilities

As a codebase increases in complexity, manually invoking command-line tools to build and test binaries can become complicated, with many command-line options. Make, one of the earliest build automation tools was inspired by a developer spending an entire morning looking for the source of a bug he had already fixed but forgot to recompile due to the sheer size of the project [38]. As computing has become even more complex, tools like CMake, Cargo, and Meson have stepped in to fill the gap but complaints have arisen at how complex they can be to set up initially and efforts are being made to simplify the process [39].

3 Methodology

The proposed system is an interpreter that translates source code for a multilingual programming language called "Lingo" into executable programs that should be able to be natively executed on users' computers. The languages supported by this system include English, Fran¸cais (French), Igb`o, Hausa, and Yor`ub´a, with keywords, error messages, and functions available in each of the four of them. Lingo is inspired by Lisp, with modified syntax to facilitate multilingual interaction. A quick overview follows:

(i) All statements are delimited by parentheses:

```
; these output some text in English and French
(print "Hello world!")
(imprimer "Bonjour le monde!")
```

(ii) Keywords have equivalents in multiple languages:

```
; these output some text in English and French
(print "Hello world!")
(imprimer "Bonjour le monde!")
```

(iii) Variables can be declared with the use of the let keyword and associated types and values:

```
; these all make a `fruit` with value "thing"
(let str <- fruit)
(let fruit "apple")
(let str <- fruit "apple")
```

(iv) Functions can be created with the return type, arguments, and body:

```
; a function returning "Hello World"
(str <- () "Hello World")

; returns "Hello " + `name`
(str <- (name : str) (concat "Hello " name))

; returns `a` + `b`
(num <- (a b : num num) (+ a b))
```

(v) Adding double semicolons on top of a variable or function declaration allows aliases to be specified in other languages:

```
;; @fr : salut
(let str <- hello () "Hello World")
; now this variable can be called `salut` and `hello`
```

3.1 Data Collection

The system's dataset comprises mappings of localized keywords and diagnostic messages to their internal representations. These keywords and messages are sourced from various sources, including but not limited to, native speakers, Google and Microsoft translation services, and localized media. These mappings are stored as TOML files in source form, which are then integrated into the interpreter itself.

4 Discussion

The proposed system is divided into several modules, each responsible for a specific task. The modules and their interfaces are detailed in the sections. Figure 4 shows the proposed system called Lingo, the architecture comprises:

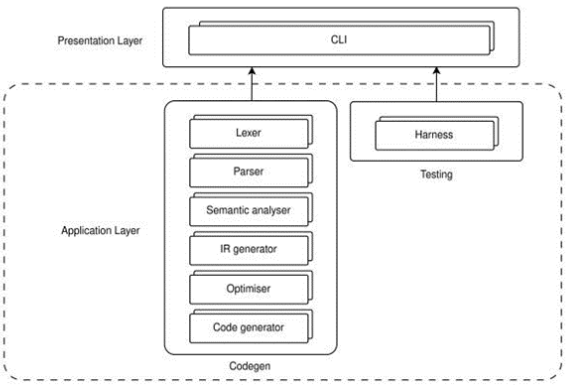

Figure 4: System Architecture Diagram for the Lingo

Command-line interface

The command-line interface is the primary means by which the user interacts with the system. It allows the user to input source code, interpret it, and run the resulting executable. The interface also displays messages meant for the user's consumption, such as localized error reports and progress information.

Lexer

Lexer is responsible for tokenizing the raw source code to make subsequent processing easier. It reads the source code character by character and converts it into meaningful tokens that can be understood by the parser. logos was used to define the lexer for the proposed system which results in extremely fast lexing due to its various performance improvements over na¨ıve lexing such as combining all token definitions into a single deterministic state machine and unwinding all loops as shown in Figure 5. This lexer utilizes various dialect definition tool files to allow languages to be added at ease to the interpreter.

```
#[derive(Logos, Debug, PartialEq, Clone)]
pub enum Token {
    // whitespace
    #[regex(r"\s+", |_| Skip)]
    Whitespace,

    #[regex(r"\n", newline_callback, priority = 3)]
    Newline,

    // punctuation
    #[token("(")]
    OpenParen,
    #[token(")")]
    CloseParen,
    #[token("<-")]
    LeftArrow,
    #[token(":")]
    Colon,

    // keywords
    #[token("let")]
    #[token("ce")]
    Let,

    #[token("if")]
    #[token("si")]
    If,

    ...
}
```

Figure 5: A snippet of the lexer definition for the proposed system:

Parser

The parser is responsible for constructing an abstract syntax tree (AST) from the tokens produced by the lexer. It checks the tokens for conformance to the rules of the language and constructs a tree that represents the structure of the source code. Figure 6 shows the parser for the proposed system was implemented using the Chomsky parser generator for Rust, which allows for the definition of grammar in a concise and readable manner.

```
impl SExpr {
  pub fn is_compat(&self, r#type: &str) -> bool {
      match self {
          SExpr::Rational(_) => r#type == "num",
          SExpr::Literal(_) => r#type == "str",
          SExpr::Float(_) => r#type == "dec",
          _ => false,
      }
  }
}
```

Figure 6: parser definition for the proposed system

Semantic analyzer

Figure 7 shows a snippet of the semantic analyzer which is responsible for checking the AST for rule violations by analyzing the source code in a context-sensitive manner. It finds logical errors, such as the use of an invalid type/value combination, and annotates the parse tree with this information. The semantic analyzer for the proposed system was implemented as a series of functions that traverse the AST and perform the necessary checks.

```
use chumsky::{lexer, parser, Grammar};

#[derive(Debug, PartialEq)]
enum SExpr {
  Atom(String),
  List(Vec<SExpr>),
}

fn parser<'a>(def: Dialect) -> impl Parser<Token<'a>,
SExpr<'a>, Error = Simple<Token<'a>>> {
  recursive(|sexpr| {
      // usual "value" types
      let atom = select! {
          Token::Real(s) =>
              SExpr::Float(s.parse().unwrap()),
          Token::Rational(s) => {
              let frac = s
                  .split_once('/')
                  .map_or_else(
                      || (s.parse().unwrap(), 1),
                      |(num, dec)| (num.parse().unwrap(),
                      dec.parse().unwrap())
                  );

              SExpr::Rational(frac.0, frac.1)
          },
          Token::Literal(s) =>
              SExpr::Literal(s.trim_matches('\"')),
          Token::Ident(s) =>
              SExpr::Var(TranslatedName {
                name: s,
                dialect: def
              }, None)
          ...
      };
  }
  ...
```

Figure 7: Semantic Analyser Snippet

Intermediate Representation Generator

The intermediate representation (IR) generator is responsible for transforming the annotated parse tree (APT) into an IR, a platform-agnostic language-independent instruction set. This IR allows for highly efficient optimizations with a variety of transformations, to make the best use of hardware. The intermediate representation generator for the proposed system was implemented as a series of functions that traverse the APT and generate the corresponding IR bytecode as a shows Figure 8.

```
pub enum InterpreterError {
  UndefinedVariable(Dialect),
  CannotPrint(Dialect),
  TryingToRedefine(Dialect),
  NonExistentType(Dialect),
  IncompatibleType(Dialect),
}

fn generate_ir(ast: &Ast) -> Result<IRModule, InterpreterError>
{
  let mut module = IRModule::new("main");

  for stmt in ast {
    match stmt {
      Stmt::Let(name, expr) => {
        let value = generate_expr(expr)?;
        module.add_global(name, value);
      },
      Stmt::If(cond, then, else) => {
        let cond = generate_expr(cond)?;
        let then = generate_block(then)?;
        let else = generate_block(else)?;
        module.add_if(cond, then, else);
      },
      ...
    }
  }

  Ok(module)
}
```

Figure 8: Intermediate representation generator for the proposed system

Optimizer: The optimizer performs operations on the IR to increase the performance and quality of the generated code. It analyses data flow in the program and utilizes various heuristic techniques that have been solved for NP-Hard optimization problems.

Code generator: code generator finally translates the improved IR to the target language the machine code of the processor—along with ensuring it is in a format that can easily be executed by the user.

Error Reporter

The error reporting module is responsible for providing informative and actionable error messages to the user. It analyses the source code and the annotations in the APT to detect any possible errors and generates messages that help the user understand and fix the issues. in Figure 9 shows error reporting module for the proposed system was implemented as a series of functions that traverse the APT and generate the corresponding localized error messages.

```
fn report_errors(ast: &Ast) -> Vec<InterpreterError> {
  let mut errors = vec![];

  for stmt in ast {
    match stmt {
      Stmt::Let(name, expr) => {
        if !is_valid_name(name) {
          errors.push(Error::InvalidName(name.clone()));
        }
        if !is_valid_expr(expr) {
          errors.push(Error::InvalidExpr(expr.clone()));
        }
      },
      ...
    }
  }

  errors
}
```

Figure 9: Error Reporting Snippet

4.2 Evaluation of Result

Performance: Lingo was found to have comparable performance to the Hedy,Hapy, and International Scheme as seen in Table 2. Babylscript was consistently 4-10 times faster than equivalent-looking Lingo code as it utilizes advanced performance features of various browser engines such as Speculation [40]. This is not to say that language usage was sluggish, typical interactions at the REPL took less than 2 seconds, but running code in a tight loop took a big hit performance-wise. This gap can potentially be shortened by the integration of ahead-of-time and just-in-time compilation techniques into the interpreter, as well as the use of more advanced optimization algorithms.

Beginner Friendliness: Lingo was found to be more beginner-friendly than Hapy and International Scheme. The syntax of Lingo was designed to be easy to learn and understand, with a user-friendly command-line interface and informative error messages to help users understand and fix issues with their code. Various evaluators commented on how the existence of localized errors helped them grasp and fix problems faster than when they had to read everything in English. Babylscript was also beginner-friendly, but its reliance on preexisting knowledge of Javascript syntax made it less accessible to novice programmers. Hedy excelled in this regard, as it was designed to be an educational tool for beginners. Its documentation and tutorials were considerably more fleshed out than Lingo's which, being still in development, was more of just a reference.

Extensibility: Lingo allows for the addition of new languages through the use of dialect definition TOML files. This makes it easy to extend the system to support additional programming languages. Hedy, in contrast, specifies that translations should be contributed to through their Weblate instance. The use of a website for translations as opposed to a more programmatic approach has pros and cons, the significant ones being the website is more accessible to possible volunteers while giving up a more programmatic approach such as allowing user-defined translations in files. If possible, future work on Lingo could consider integrating a similar system to allow for more crowdsourced translations while still keeping the advantages of the TOML file approach. Babylscript also supports multiple languages, but it requires users to manually specify the translations for each keyword and function, which can be time-consuming and error-prone.

Lingo was evaluated against existing systems to determine its effectiveness and efficiency in solving the issue of the language barrier. The systems compared were Hedy, Hapy, Babylscript, and International Scheme due to their similar text-based natures. In addition to this, several English-speaking bilingual university students were informally interviewed after they used the system for a twenty-minute session. The testing methods and criteria are detailed below: Performance: Equivalent programs to generate the stopping time of the Collatz sequence for the first 1000 natural numbers were made and timed. The results are shown below in Table 2.

Table 2: Collatz Sequence Number Generation Benchmark

Language	Time taken (ms)- Averaged over 10 runs
Babylscript	35
International Scheme	169
Lingo (proposed system)	**181**
Hapy	190
Hedy	202

Beginner-friendliness: The 7 interviewees were walked through the language and were shown its basic capabilities in both their native languages and English. They were then asked to rate the learning experience in both languages and whether or not having the keywords and errors localized helped during the native language round. The results are shown in Table 3.

Table 3: Interview Results

Question	YES	NO
Was the quality of the English error reporting good?	7	0
Was the quality of the alternate language error reporting good?	5	2
Was the language shown to be capable and ready for release?	2	5
Would you add a new language to the interpreter?	2	5
Was the language shown to be capable and ready for release?	1	6

Extensibility: The difficulty of adding a new language's keywords to these languages was evaluated by inspecting their repositories and websites. Hedy was unable to be evaluated with thiscriteria because it does not support language extensions. The results are shown in Table 4.

Table 4: Extensibility of Evaluated Languages

Language	Steps Needed	Difficulty
Hedy	Add a translation on Weblate	Easy
Lingo	*Add a* `dialect.toml` *file to nterpreter*	*Medium*
International Scheme	Add `translation.scm` to repository	Medium
Babylscript	Create language to-keniser and add to repository	Hard
Hapy	N/A	N/A

5 Conclusion

The implementation of the proposed interpreter along with the design of the Lingo programming language shows that this proof-of-concept is viable and further work should be done in this area of study. By applying efficient compiler design techniques, the interpreter can efficiently and accurately execute source code and give informative localized user feedback. Based on the results of the evaluation, Lingo meets the requirements set forth at the beginning of the study. It provides a beginner-friendly environment, supports extensibility, excels in localization, and promotes collaboration among users, making it a great solution for overcoming the language barrier in programming. However, the addition of some quality-of-life features like an easier-to-use translation interface, better guides, and text-editor integration would further improve the language and make it even more welcoming to newer users.

Some recommendations for the future of the study and further work on programming language design in the area of localization and accessibility. They include: Increased focus on localization efforts: The study only focused on four languages for the initial implementation. Future work should expand this to include more languages, especially those with non-Latin scripts, as well as provide a web interface to reduce the effort needed by potential volunteers. Better localized documentation and tutorials: The study did not cover the localization of extensive documentation for the language—just settling with a reference in the various languages offered. Future work should look into this to ensure that users can learn programming concepts with the use of Lingo in their native dialect, no matter their skill level. As the world gets more digital, the language barrier only serves to reduce the number of potential computer programmers who could make a difference in the industry. This means that the industry is missing out on a lot of potential talent that could be contributing to the field. The proposed system—one that can take source code written in a multilingual programming language and execute it on a user's computer, as well as provide localized error messages to help users understand and fix any issues with their code—is a step in the right direction to bridge this gap and make programming more accessible to a wider audience.

The repository containing the code for the implemented language can be found at https://codeberg.org/fumnanya/lingo.

References:

[1] Weik, M. H. (1955, December). A Survey of Domestic Electronic Digital Computing Systems. https://ed‐thelen.org/comp‐hist/BRL‐e‐h.html#ENIAC

[2] Gunter, B. (2019). The Prevalence of Mobile Phones in Children's Lives. Children and Mobile Phones: Adoption, Use, Impact, And Control, 25–34. https://doi.org/10.1108/978‐1‐78973‐035‐720191006

[3] Turon, A. (2015). Fearless Concurrency with Rust. The Rust Programming Language Blog.https://blog.rust‐lang.org/2015/04/10/Fearless‐Concurrency.html

[4] Donovan, A. A., & Kernighan, B. W. (2016). The Go Programming Language (pp. 280–281). Addison‐Wesley. https://doi.org/10.5555/2851099

[5] Sommerfeld, H. (2021). Use a US Keyboard for Programming. Henrik Sommerfeld's Blog.https://www.henriksommerfeld.se/use‐a‐us‐keyboard‐for‐programming/

[6] McCulloch, G. (2019). Coding Is for Everyone—as Long as You Speak English. https://www.wired.com/story/coding‐is‐for‐everyoneas‐long‐as‐you‐speak‐english/

[7] Goguen, J. A. (1975). Semantics of computation. Category Theory Applied to Computation and Control. https://doi.org/10.1007/3‐540‐07142‐3_75

[8] Stack Overflow. (2023). Key territories: Stack Overflow DeveloperSurvey 2023. https://web.archive.org/web/2/https://survey.st

ackoverflow.co/2023/#developer‐ profile‐ key ‐ territories

[9] Abdalkareem, R., Shihab, E., & Rilling, J. (2017). What Do Developers Use the Crowd For? A Study Using Stack Overflow (Vol. 34, pp. 53–60). https://doi.org/10.1109/MS.2017.31

[10] Price, T. W., & Barnes, T. (2015). Comparing Textual and Block Interfaces in a Novice Programming Environment. Association for Computing Machinery. https://doi.org/10.1145/2787622.2787712

[11] Rymer, J. R., & Richardson, C. (2015). Low‐ Code Platforms Deliver Customer‐ Facing Apps Fast, But Will They Scale Up? https://web.archive.org/web/20240404121723/https://www.forrester.com/report/

[12] Aaby, A. A. (2004). Introduction to Programming Languages. https://web.archive.org/web/20121108043216/http://www.emu.edu.tr/aelci/Courses/D‐ 318/D‐ 318‐ Files/ plbook/intro.htm

[13] Dragoni, N. (2014). Introduction to P2P Computing (p. 2–3). http://www2.imm.dtu.dk/courses/02220/2017/L6/P2P.pdf

[14] Colburn, T., & Shute, G. (2007). Abstraction in Computer Science. Minds and Machines, 17(2), 169–184. https://doi.org/10.1007/s11023‐ 007‐ 9061‐ 7

[15] O'Regan, G. (2008). Computer Programming Languages. In A Brief History of Computing (pp. 73–102). Springer. https://doi.org/10.1007/978‐ 1‐ 84800‐ 084‐ 1

[16] Saxon, J. A., & Plette, W. S. (1962). Programming the IBM 1401, a self‐ instructional programmed manual. Prentice‐ Hall. https://babel.hathitrust.org/cgi/pt?id=mdp.

[17] Kernighan, B. W., & Ritchie, D. M. (1978). Preface to the first edition. In The C Programming Language (2nd ed., p. 8–9). Prentice‐ Hall. https://books.google.com/books?id=FGkPBQAAQBAJ

[18] Perlis, A. J. (1996). Foreword. In H. Abelson, G. J. Sussman, & J. Sussman, Structure and Interpretation of Computer Programs (2nd ed.). https://web.archive.org/web/20010727170154/http://mitpress.mit.edu/sicp/full‐ text/book/book‐ Z‐ H‐ 5.html

[19] GNOME (2023). GNOME Languages.https://web.archive.org/web/20230829184045/ https://l10n.gnome.org/languages

[20] Axtens, B. M. (2024). HOPL. http://hopl.info/

[21] Bob Nystrom (2021). A map of the territory. https://github.com/munificent/craftinginterpreters/blob/cf221a107ac185c1d6ebd02324179934714c2538/site/image/a‐ map ‐ of‐ the‐ territory/mountain.png

[22] Benidris, M., & Ammar, H. (2018). The Correlation between Arabic Students' English Proficiency and Their Computer Programming Ability at the University Level. International Journal of Managing Public Sector Information and Communication Technologies, 9. https://doi.org/10.5121/ijmpict.2018.9101

[23] Rust Survey Team. (2023). State of Rust Annual Survey 2023. https://blog.rust‐ lang.org/2024/02/19/2023‐ Rust‐ Annual‐ Survey‐ 2023‐ results.html

[24] Miranda, A., & Pimentel, J. (2018). On the use of package managers by the C++open‐ source community. https://doi.org/10.1145/3167132.3167290

[25] Kuhail, M. A., Farooq, S., Hammad, R., & Bahja, M. (2021). Characterizing Visual Programming Approaches for End‐ User Developers: A Systematic Review. IEEE Access,9, 14181–14202. https://doi.org/10.1109/ACCESS.2021.3051043

[26] Translatewiki. (2024). Localisation statistics for the Blockly core module. https://translatewiki.net/wiki/Special:MessageGroupStats/out‐ blockly‐ core#sortable:3=desc

[27] Scratch Team. (2024). Scratch Statistics. https://web.archive.org/web/20240229194506/https://scratch.mit.edu/statistics/

[28] Hague, J. (2013). Why Do Dedicated Game Consoles Exist? Programming in the Twenty-First Century. https://prog21.dadgum.com/181.html

[29] McIntyre, D. B. (1991). Language as an intellectual tool: From hieroglyphics to APL. IBMSystems Journal, 30(4), 554–581.

[30] Morgan‐ Mar, D. (2008). Piet. https://dangermouse.net/esoteric/piet.html \

[31] Segun‐ Lean, E., & Wuta, S. (2021b). Hapy: Hausa Programming Language [Computersoftware]. https://github.com/hapy‐ lang/hapy

[32] Hedy Team. (2021). Hedy: Textual programming for the classroom. https://www.hedy.org/

[33] Hermans, F. (2020). Hedy: A Gradual Language for Programming Education. ICER '20: International Computing Education Research Conference. https://doi.org/10.1145/3372782.3406262
[34] Gilsing, M., & Hermans, F. (2021). Gradual Programming in Hedy: A First User Study. 2021 IEEE Symposium on Visual Languages and Human‐Centric Computing (VL/HCC). https://doi.org/10.1109/VL/HCC51201.2021.9576236
[35] Iu, M.‐Y. C. (2012). Babylscript: Multilingual Javascript. Companion to the 26th Annual ACM SIGPLAN Conference on Object‐Oriented Programming, Systems, Languages, And Applications, OOPSLA 2011, 197–198. https://doi.org/10.1145/2048147.2048204
[36] Adriano Torres, Caio Oliveira, Márcio Okimoto, Diego Marcílio, Pedro Queiroga, Fernando Castor, Rodrigo Bonifácio, Edna Dias Canedo, Márcio Ribeiro, & Eduardo Monteiro. (2023). An Investigation of confusing code patterns in JavaScript. Journal of Systems and Software, 9.
[37] Isakovic, A. (2020). International Scheme [Computer software]. https://github.com/metaphorm/international‐scheme/
[38] Raymond, E. S. (2003). The Art of Unix Programming. http://www.catb.org/esr/writings/taoup/html/
[39] Bartlett, R. (2022). A new CMake Scripting Language?. https://www.osti.gov/servlets/purl/2003320
[40] Pizlo, F. (2020). Speculation in JavaScriptCore. https://webkit.org/blog/10308/speculation‐in‐javascriptcore/