A Framework for the Localization of Programming Languages

Alaaeddin Swidan
alaaeddin.swidan@ou.nl
Open University of the Netherlands
Netherlands

Felienne Hermans
f.f.j.hermans@vu.nl
VU Amsterdam
Netherlands

Abstract

Most programming languages are only available in English, which means that speakers of other languages need to learn at least some English before they can learn to program. This creates well-documented barriers to entry into programming. While many educational programming languages are localized in some way (e.g. keywords), they often miss important other aspects (e.g. numerals or word order). This paper describes a framework of 12 aspects of programming languages that can be localized, helping tool designers localize their languages better and educators to make more informed decisions about introductory languages in non-English contexts.

CCS Concepts: • **Applied computing** → **Education**; • **Software and its engineering** → **Software creation and management**; **General programming languages**.

Keywords: programming languages, localization

ACM Reference Format:
Alaaeddin Swidan and Felienne Hermans. 2023. A Framework for the Localization of Programming Languages. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on SPLASH-E (SPLASH-E '23), October 25, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3622780.3623645

1 Introduction

According to language center Ethnologue, over a billion people in the world speak English, meaning about 6 billion do not.[1] Despite that, the world of software is largely an English centric world. Localization is a topic that has received considerable attention in the HCI community, which has consistently highlighted its importance [23, 33], however localization is expensive and hard to automate [21].

As such, most programming languages are only available in English. For speakers of other languages, especially for those belonging to small or marginalized linguistic groups, the only option is to program in English from their first line of code. In many cases, learning programming takes place before or during learning English. Prior work has shown that performance in English is correlated with programming ability. Furthermore, both students and teachers indicate that English is a barrier for Arabic speaking students [6, 15].

English is visible in various parts of the programming experience. Firstly, in nearly every widely used programming language, the names of symbols and keywords (if, else, while, for) are borrowed from English. Not only are the keywords are in English, the way keywords form statements often resembles pronounceable sentences in English; 'if x = 5' can be read as the sentence 'if x equals 5'. Finally, almost all programming languages only accept numerals 0 to 9 for calculations. [2]

The impact of the English-centricity goes beyond aspects of the programming language itself; compilers often make assumptions about user chosen parts of programs, such as identifier names, and assume these will always be in Latin letters (a-z). Some languages (like MATLAB) even limit the characters which may be used in comments, causing users to be unable express themselves in their native language in code comments, parts which are not even executable.

The goal of this paper is to describe a framework encapsulating the aspects of programming languages that can be translated to non-English, including decisions about keywords, numerals and non-letter characters. We demonstrate the applicability of our framework by applying it to evaluate 13 different programming languages, many of them educational languages, demonstrating that localization of these aspects in a programming language is indeed possible.

The use of this framework is two-fold. Firstly, it can be used both by language designers interested in assessing and improving the localization possibilities of their own languages. Secondly, this overview will support educators in selecting localized languages to teach with if they aim to also support non-English students.

[1] https://www.ethnologue.com/insights/ethnologue200/

[2] Formally these are called the Hindu-Arabic numerals, but for the sake of clarity we call these English numerals in the remainder of this paper.

SPLASH-E '23, October 25, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0390-4/23/10.
https://doi.org/10.1145/3622780.3623645

2 Background

Several non-English programming languages have been developed since the beginning of the 1960s to ease programming in languages other than English, often in an educational context (see further 4). In this section, we explore the difficulties that learners have when faced with non-English programming languages.

2.1 Error Messages

The aspect of programming languages most often localized are compiler error messages. Error messages are hard to understand for any novice, and them being (partly) in English adds an extra barrier for non-English novices. For example, Nnass et al. [27] compared the difficulties of Australian students to Libyan students and found that while only 2% of Australian students name error messages as a major problem in learning to program, 71% of (Arabic speaking) Libyan students named error messages. Reestman and Dorn [31] found a significant difference between native speakers and non-native speakers (Spanish, Chinese, Korean and Japanese) albeit with a small effect size.

In a survey among 150 students speaking isiZulu, isiXhosa, and Afrikaans in which the majority indicated that they believed that localized compiler messages would help them learn programming more effectively[26], and students with a good command of English indicates that they would value localized error messages as often as students with lower proficiency.

Different approaches for localizing error messages have been suggested over the years. For example, Roehner [32], suggest to automatically translate error messages, describing both approaches that use a (hand-made) dictionary and an approach using machine translation.

2.2 Teaching CS in non-English

Teachers need to consider the language of instructions when teaching a topic to a non-native English audience. Learners could face difficulties that originate from the need to develop English understanding at the same time that they are expected to learn about the topic's conceptual knowledge [15]. Additionally, a mismatch between the native language of the students and the native language of the teacher can hinder the relatedness of the topic and the classroom engagement [16, 20]. Pal and Iyer [28] investigate the effectiveness of using the native language versus English as the vocal instruction language in videos that contain English slides while teaching programming at the university level. They report that students who got instructions in the same native language that was used during their K-12 education (either English or Hindi) outperform the Hindi speaking and K-12 taught students who received instructions in English. In some other cases, the English language can become a barrier for reasons related to culture and values, for students and teachers. Gyabak and Godina [16] developed a course of digital story telling to improve the digital literacy of students in Bhutan. To their surprise, the English interface of the software environment was an obstacle to many students and teachers who had an 'embedded sense of historical and colonial perceptions about the English language'. Soosai Raj et al. [35] compared two versions of an operating systems course: one in Tamil *and* English and one in English only. They found no effect on learning outcomes but a significant effect on engagement in terms of more dialogue in the classroom where students could communicate in their native language.

Previous work has documented some of the issues that non-English learners have in using English programming languages. For example, Qian et al. [29] discuss the specific issues that Chinese learners face while using Python. For example, there are characters in Python deceivingly similar to their Chinese counterparts, but are not exactly the same. For example, compare Chinese brackets: （ and ） to Python's (and). This leads to confusing errors.

Quite some work has looked at the transition between block based languages and text-based languages, but implicitly assumes the transition happens between two English language systems, for example [5], while most block-based systems support some localization. Papers that did look at two different natural languages, like Espinal et al. [11] who studied switching from blocks in Spanish to Python, found that that gap is large. Students for example stated they did '…not understand the program because there are some words in English'.

Block-based languages, as explained above, are often better localized than their textual counterparts, and thus allow for experiments into the effect of localized programming languages on learning. Dasgupta and Hill [9] found that children that were programming in their own language learned programming concepts faster than their counterparts programming in English. The languages in this paper were all Latin languages written left to right (Italian, Portuguese, Brazilian Portuguese, German and Norwegian), so the difference might be even larger for non-Latin right to left languages. Other work on Spanish speakers using block-based languages showed that interaction with tooling is hampered when students native language is inconsistent with the language featured in the tool [12].

3 Localizable aspects of programming languages

Despite the existence of many localized programming languages, we are not aware of work that details the design considerations when building localized languages. Many localized programming languages are simply translations of existing English languages, which do not always take into

account the possibility of making changes away from English. This paper aims to present a comprehensive overview of 12 aspects that could be localized in a non-English programming language.

3.1 Alignment with Existing Language

The first aspect of a localized language is whether it is made as a translation of an existing language of which one localizes the keywords, or whether it will be a newly designed non-English language.

Some languages translate an existing language and translate the base language's keywords, or are heavily inspired by an existing language. قلب is an example of such a language; it is a translation of Lisp using Arabic keywords. Alf..Eih is another example, which is an Arabic translation of C++ [30], and Phoenix is a translation of C# [4].

Other languages, such as the Wenyan language [7],[3] or the Dolittle language [24] are designed from their inception to be non-English.

While translating is easier because fewer decisions will have to be made and existing tooling can be used, designing a PL for one natural language offers the possibility to use native constructs in programming. For instance in Arabic PL Ammoria, you can use different forms to indicate the number of times a loop iterates: singular (مرة), dual (مرتين), or plural (مرات).

In other cases, translation might not be feasible because the language differs too much from English. In Turkish, the keyword `while` is translated as *'iken'*, but placed after the condition rather than before it (`i==0 iken`), most likely requiring changes to existing tooling. In Standard Chinese the keyword `in`, translated as '在里面', is usually placed around a word, rather than before it: 'In the house' (house being '屋') would not be '在里面 屋' but '在 屋 子里面'. As such, a natural translation of a language construct like `for item in list` would be `for item 在 list 里面`.[4]

3.2 Keywords

Whether you are translating an existing programming language or creating one from scratch, decisions about the keywords will have to be made, and the influence of existing English programming languages cannot easily be ignored. Because non-English programming languages are often created by people with a good command of English themselves, it is hard to not take known word meanings into account.

For example, the keyword `print`, is used in several programming languages, but it also means printing on paper. This leads to interesting choices if your natural language has different words for showing something and for printing on paper. Dutch for example has the word 'tonen' meaning showing, but also the word 'print' that exclusively means printing on paper with a printer. What would be the better choice here? This is even more complex for languages that have a dedicated word for printing digitally or printing on the screen.

The problem lies in the fact that English keywords often have very specific meanings, for example a keyword like `echo`. The literal translations of echo are mostly associated with the natural phenomenon of sounds reflecting or coming back from a surface. In the programming context, however, the command is mostly associated with 'repeating' certain values passed to it. The decision to either be closer to the translation or select a word closer to the programming-context is not trivial.

3.3 Variable names

Traditionally, most programming languages only allow variable names consisting of Latin letters, combined with numbers and often special characters including the underscore. The assumptions about variable names are visible, but not explicitly named in our literature. The famous *Dragon* book on creating programming languages explains identifiers as shown in Figure 1. However, the fact that other languages use different numerals and letters remains unaddressed in the entire book [1].

Example 3.5: C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers. We shall conventionally use italics for the symbols defined in regular definitions.

$$
\begin{array}{rcl}
\mathit{letter\_} & \rightarrow & \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\
\mathit{digit} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\mathit{id} & \rightarrow & \mathit{letter\_} \; (\; \mathit{letter\_} \mid \mathit{digit} \;)^*
\end{array}
$$

Figure 1. Identifiers as explained in Compilers: Principles, Techniques, and Tools (2nd Edition), 2006

Newer works covering building programming languages, such as the (otherwise lovely) book by Nystrom, use a definition similar to the one in Figure 1

This problem is also visible in tutorials. When trying to find examples of a simple programming language with a Google search for 'simple lexer', all of the results on the first search page assume an alphabet of a-z and A-Z and digits 0-9, and do not explicitly mention the cultural bias therein.[5] [6] [7]

The problem of Latin variable names continues from resources about building language, to tools to *create* programming languages. Lark,[8] a widely used parser framework for Python, uses the following definition for variable names (simplified here for readability):

[3] https://wy-lang.org/

[4] Note that spaces are not common in Standard Chinese, we add them to increase readability for unfamiliar readers.

[5] https://balit.boxxen.org/lexing/matching_literals.html

[6] https://gist.github.com/arrieta/1a309138689e09375b90b3b1aa768e20

[7] https://beautifulracket.com/basic/the-lexer.html

[8] https://github.com/lark-parser/lark/blob/master/lark/grammars/common.lark

```
LETTER: "a".."z" | "A".."Z"
DIGIT : 0..9
CNAME : ("_"|LETTER) ("_"|LETTER|DIGIT)*
```

A programmer using Lark, not aware of localization issues, using the built-in variable (something simply accomplished by using `import common.CNAME`), might not even realize that they are creating variable names that only allow for Latin letters and numerals.

Recently some languages are starting to allow all Unicode characters. For example, both Python and Lua now support non-Latin letters.[9],[10]

While allowing non-Latin characters in variable names is a big step forwards to more inclusivity for non-English languages, it does not suffice entirely. For example, some languages have words containing characters that are generally not considered letters by English speakers and (thus) often by parsers. Ukrainian, for example, allows words to contain a single quote; 'ім'я' for example means *name* and is thus quite likely to occur in identifier names. Using quotation marks in identifiers is not allowed by the majority of textual programming languages (Lisps like Clojure, and functional languages like Haskell being the exceptions). Block-based languages like Scratch and Snap! do support variable names with all characters, and even allow names that consists entirely of numbers from all numeric systems (such as 1), a feature that is used in some programs [37].

3.4 Productions

In addition to the keywords themselves, the way they are used to form constructs too can be English-centric. Many constructs in English programming languages somewhat resemble sentences that can be pronounced as such, for example `for fruit in basket` or `if temperature == 5`. When implementing a non-English programming language, these productions could be made more natural sounding in the natural language, for example in German, one would not say 'if x is 5' but 'wenn x 5 ist' placing the verb at the end and in Korean one would not say 'turn left', but 'left turn'.

Other languages have words that will never be able to fit in the mold of English. For example, in Turkish, the keyword `while` is translated as *'iken'*, but placed before the condition, such that a natural way of phrasing would be `i==0 iken`. Or, in Standard Chinese the keyword `in`, translated as '在里面' comes around a word, rather than before it: 'In the house' (house being '屋') would not be '在里面 屋' but '在 屋 子里面'. As such, a natural translation of a language construct like `for item in list` would be `for item` 在 `list` 里面.[11]

[9] http://www.python.org/dev/peps/pep-3131
[10] https://www.lua.org/manual/5.3/manual.html#6.5
[11] Note that the spaces are used to easy readability for readers not familiar with Chinese, these are not commonly used in Standard Chinese sentences.

There are localized programming languages that use a non-English word order, for example Dolittle is a Japanese programming language which uses Japanese word order (Subject Object Verb).[12]. For example かめた！100　歩く。 meaning 'bite 100 forward' is used to move a turtle called 'bite' forward 100 pixels.

In addition to different word orders, different modifications also exist. Some languages use a so-called *separable verb*, where a verb can be split and placed in different places in a sentence, for example in Dutch 'afdrukken' means printing, but when used in a sentence with an object, it becomes 'druk ... af' *around* the object. As such a translation of 'print hello word' would be 'druk hello world af' which sounds natural, but would be hard to put into existing programming language syntax, especially when `print()` is a function. Changing the syntax to a more common `druk_af('hello world')` is often possible, but removes the correct word order. Splitting the function name around arguments (something like `druk('hello world')_af` is more natural. This construction, also called a mixfix operation [8]) is not usually supported in traditional programming languages (Smalltalk and ObjectiveC are a notable exceptions, in addition to proof assistants Coq and Isabelle).

Hindu-Arabic	0	1	2	3	4	5	6	7	8	9	10
Arabic	٠	١	٢	٣	٤	٥	٦	٧	٨	٩	١٠
Hindi	०	१	२	३	४	५	६	७	८	९	१०
Bengali	০	১	২	৩	৪	৫	৬	৭	৮	৯	১০
Thai	๐	๑	๒	๓	๔	๕	๖	๗	๘	๙	๑๐
Chinese (Mandarin)	〇	一	二	三	四	五	六	七	八	九	十

Figure 2. A list of known numeral systems [22]

3.5 Numerals and numbers

Another aspect of consideration for language builders are about numerals. There are several different numeral systems that are used in different language families. Figure 3.4 shows an overview of numeral systems. While Hindu-Arabic numerals (0 to 9) are the most used around the globe, other numeral systems are commonly used, especially in K-12 teaching.

However, non-English numerals are rarely supported in programming. Even programming languages that allow for localization, such as Scratch and Snap!, do not also allow for localization of numerals. In Scratch, non-English numerals can be entered in numeric slots, but the block will not be executed because the numeral is not recognized, without showing an error message. In Snap!, non-English numerals cannot be entered in numeric slots.

Learners will also see that when working with computers they have to detach from part of their native language and

[12] https://dolittle.eplang.jp/

use something foreign, opposite to other topics, which could put a gap between them and programming. In her ethnoprogramming model, Laiti stresses that such cultural elements are important to the learner especially for non-English contexts [20].

Even using English numerals (0 to 9) there can be differences impacting programming language design. For example, many non-English, European languages use commas as decimal separators and periods as thousand separator: 2,041.35 in English is 2.041,35 in French or Italian. ometimes also the apostrophe is used as thousand separator (2'041) or a high point (2˙041). The places of separators can also differ; in India and some of its surrounding countries 10 million is written as 1,00,00,000 and not as 1.000.000 as in other English speaking countries. Creating a language that feels natural for many European users might impact how numbers are inputted or shown. Localizing numbers is possible: Excel allows commas as decimal separators for numbers, however this is not a feature of any text based programming languages known to the authors, although OCaml allows the placement of underscore within an integer literal for readability, e.g. 1_00_00_00 is valid.

3.6 Characters 'without meaning'

Some languages contain characters that do not change the meaning of a word, but are written to express something else. For example, Arabic has a tatweel character (see Figure 3). The tatweel, is the unicode character name (U+640 Arabic Tatweel) that originates from the Kashida. A Kashida is a curvilinear elongation of certain Arabic letters depending on the chosen style of writing, and is used for improving the text in aspects that are not limited to justification, aesthetics, and emphasis [2, 18, 25].

Spaces too can have different implications in different languages. In many Western languages, spaces are used to separate words and as such have meaning. But many character based languages like Chinese or Japanese do not commonly use spaces between words, and neither do alphabet based Tai-Kadai languages spoken in Thailand and Laos. It will be hard for language and tool designers to building programming languages for such languages since tooling for creating programming languages usually assumes that words and keywords will be separated by spaces.

The Wenyan language, trying to be close to Classical Chinese, mandates the 者 character at the end of an if statement. The production in the Wenyan specification is as follows:[13]

```
if_statement : IF if_expression '者'
statement+ (ELSE statement+)? FOR_IF_END ;
```

Making a character without direct meaning mandatory is an interesting decision which deviates from what is common in programming language design, where typically keywords are used that have explicit meaning.

[13] https://wy-lang.org/spec.html

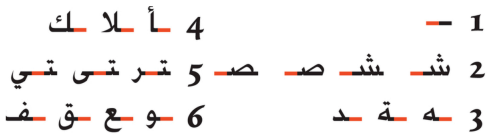

Figure 3. Examples of the tatweel character (in red) as used in Arabic modern typography to prolong words. Multiple tatweel characters can be attached to make a longer tatweel. [34]

3.7 Diacritics

Many languages make use of diacritics to modify letters, such as the accent accute (on the a: á), accent grave (à) and accent circumflex (â) in French. These are sometimes used to indicate prosody, in Dutch for example an accent is used to indicate stress on a syllable. In other languages accents can also change the meaning of a (written) word. In French, 'là' means 'there' while 'la' means 'the' but both words are pronounced the same.

When designing a programming language for a language with accents, such as French, an interesting open question is how to handle diacritics in various cases. One of such cases is keywords, if a keyword is translated with a word that has accents, how to thread the equivalent word without diacritics? For example, in French, 'repeat' is translated as 'répète', which is used in some programming languages for repetition (for example Quorum [36]). If `répète 3 fois` is the correct code, do we also allow `repete 3 fois` or will that lead to an error message? If we do not see repete as an alternative to répète, is it still allowed as a variable name? These question are especially interesting for teaching languages of which the users might not yet know the correct use of diacritics on all words, such as educational languages.

If we allow all characters in variable names including letters with diacritics, another problem arises, similar to case sensitivity, which we could call *diacritic sensitivity*: do words with and without diacritics (such as `quantité` and `quantite`) refer to the same variable, or to different ones? Most languages that support non-Latin variable names currently are *diacritic sensitive*, i.e. if in one program, a name with and without accents is used (`là` and `la`) these point to different objects, although we suspect this is a matter of tooling than a deliberate choice, since most underlying frameworks do not capture the fact that that certain letters are interchangeable.

In addition to accents, a similar question can be posed for letters that are modifications of letters in different ways than with accents, such as c-cedilla (ç) in Albanian and Turkish.

3.8 Alternative keywords

Traditionally, programming languages use exactly one keyword for a concept. However, this limitation makes it hard to fully support a broad range of natural languages. One

example where a user might need the support of multiple keywords for one concept is in the case of gendered words. Arabic for example has gendered nouns; a noun can be either masculine or feminine. This will have consequences on the forms of verbs and demonstrative pronouns, among other things. In programming, identifier names are often nouns and as such they will be gendered in some languages. This makes the translation of programming keywords that manipulate variables dependant on their gender as well. In Section 3.1, we discussed how the word 'is' could have multiple ways of translating it in Arabic depending on the context. One possibility when translating 'is' as a demonstrative pronoun can have it as هو if it follows a masculine variable name but as هي when the variable name is feminine. Similarly, when translate the word equals could have either the word يساوي or تساوي depending on the variable's gender.

Some languages allow for multiple different keywords for similar concepts. For example, Arabic Lisp [10] translates the keyword NIL to two different keywords in Arabic depending on the context: فراغ if it refers to an empty list, or خطأ if it refers to `false`. Another example in Arabic is the translation 'to'. Depending on whether you write the standard or Egyptian Arabic, the word 'to' may be written as إلى or إلي, with the only difference being the two dots under the last letter. The dots are a special type of diacritic, it is not simply a mark on around the letter, but part of the letter. To accommodate for the two variants of Arabic, Ebda3 allows both إلى إلي.[14]

3.9 Localized punctuation characters

As mentioned in the introduction, many cultures use punctuation characters that are different from the ones commonly used in English. The list is very long, but a few notable examples include:

Symbols for quotation While single and double quotation marks ('...' and "...") are common in English, other cultures also use guillemets «...» (French), reversed guillemets (Danish) »..«, or corner brackets 『 ... 』 (Chinese).

Parentheses Chinese uses different but very similar symbols. In right to left languages the parentheses change places; coming from the right, the first bracket we encounter is).[15]

Question marks and exclamation marks Spanish uses an inverted question mark and exclamation mark at the beginning of a sentence, Greek uses ;.

Commas Arabic uses a comma that points rightwards and upwards: ،.

For any of the above characters and numerous other, choices will have to be made whether or not to allow them in code. Can ، be used as a list separator? Do we allow guillemets to start and close string values?

An interesting and related question is whether we should allow multiple symbols. Should we allow both English brackets and Chinese brackets for function calls so Chinese speakers can use theirs without issues? In a case where a question mark is used, such as in Rust when opening a file: `let f = File::open("data.txt")?;` should it start with an inverted question mark so it reads more natural for Spanish speakers? Traditionally, multiple symbols for similar roles in programming have not been common, but it could be possible to allow for different symbols such as Chinese brackets, Arabic commas or optional inverted question marks.

3.10 Right to left support

There are multiple issues when it comes to supporting writing and editing text in right to left (RTL) languages. The first issue is the alignment: RTL text should be aligned correctly to the right side of the screen towards the left. Secondly, there is the direction of letters and words in a sentence. Some editors fail to render RTL text correctly, flipping the order: the first letter in a RTL word becomes the last. Thirdly, ligatures: in some RTL languages (like Arabic), letters have different forms depending on their position in a word to allow for connecting letters together. Many editors fail to respect this, and split the letters, rendering the text very hard or impossible to read.

In the case of Arabic, two websites[16] collect images of situations when the typing of Arabic words or sentences is wrong.

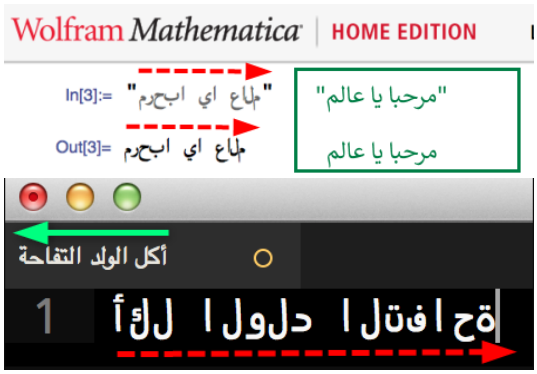

Figure 4. Right to left support when typing Arabic. Top (Mathematica): direction flipped for both sentence and words, and letters disconnected. Correct representation is shown in the green box. Bottom (Sublime): from Sublime text, the window's title is correctly represented, but the text in the editing area has direction flipped for the sentence and words, and letters disconnected.

[14] http://ebda3lang.blogspot.com/

[15] This problem is solved in many cases by fonts that show a closing bracket but save an opening bracket to not confuse parsers

[16] https://notarabic.com/,https://isthisarabic.com/

3.11 Multi-lingual programming

Another aspect to take into account in designing a truly inclusive programming language experience is the use of multiple languages intermixed. Many people around the world are native speakers of not one but two or even three languages. The BBC estimates that between 60 and 75% of people are bilingual.[17] For bilingual people, being constrained to one of their native languages while programming is limiting, as they often switch languages when speaking or writing. Vogel et al. [40] argues that translanguaging pedagogies, i.e. ways of teaching in which multiple natural languages are used, can be beneficial in learning programming.

Another reason to make programming languages multilingual is not for the bilingual speaker, but for a person learning English and programming at the same time. This is named by Guo [15] as a core difficulty in learning to program for non-native speakers. When a programming language allows for the native language of the learner but also English, they can use their own native languages initially, and later on mix in more English keywords as they learn them, easing a transition to a fully English language later on.

There are programming languages supporting *multiple* natural languages. For instance, ALGOL 68 supported several natural languages other than English. There is also Rapira [3] built with Russian, English, and Moldovan variants. Rapture[18], a modern implementation of Rapira, supports Russian and English keywords that can be used together in one file. Finally, BabylScript[19] supports 12 different languages, which can also be mixed within a file (although not within statements).

3.12 Error messages

Programming language design includes the design of error messages, which are also shown in English for most English language programming languages. However, as we have outlined in Section 2, error messages in English can be a substantial barrier for non-English users. Error messages, in principle, can be localized with relative ease, compared to keywords or productions, however, error messages are not often localized for professional languages. Educational languages do sometimes support localized error messages, for example in Hedy and Scratch.

4 Application of the framework to existing languages

In the above, we described aspects of programming languages that language designers could take into account when localizing a language, and that people, for example educators, can use to decide whether a localized language fits their teaching needs.

In this section, we demonstrate the applicability of our framework to a set of 12 programming languages.

Table 1 presents an overview of programming languages. Since there is a large set of very diverse localized programming languages to present, we aimed to find at least one programming language for each of the aspects of the framework (the rows of Table 1) and then scored these languages on all other aspects.

The legend of the table is as follows. The first row (Alignment) indicates whether the language is a direct translation of an existing language (T) or is a newly designed language (N). The other rows follow this scheme:

- ● indicates that the aspect is fully supported in the programming language. For example, Scratch fully supports non-Latin variable names.
- ◑ indicates that the aspect is partly supported in the programming language. For example, ebda3 has a ◑ for diacritics because it supports some, but not all diacritics in Arabic. Aspects that are partly supported are explained in more depth in the accompanying text.
- ○ indicates that the aspect is not supported in the programming language. For example Hedy does not support productions differing from the original (English) ordering.
- \- indicates that the aspect is not applicable. For example, Snap! does not use punctuation characters such as brackets for functions, or commas for list creation and therefore these they cannot be localized.

In the following text, we discuss the languages in the order of the table. For all languages, we discuss features that are partly supported by the languages and explain why they are only partly present, and we discuss selected features that are of special interest.

4.1 Scratch

Scratch does not generally support customizeable production orders, the order of keywords is as is defined in the language specification. Only when a user creates their own custom blocks, they can choose to add labels in between arguments, allowing for the creation of a block that expresses, for example, `print argument_1 uit` in Dutch. As explained earlier Scratch does not allow for the use of non-English numerals. Scratch largely supports right to left programming: the UI switches to right to left, also inverting the blocks. However, misplaces punctuation, see Figure 5.

4.2 Snap!

Like Scratch, Snap! allows the creation of custom blocks with labels in between arguments of the block, so that it is possible to create blocks that support separable verbs in a

[17] https://www.bbc.com/future/article/20160811-the-amazing-benefits-of-being-bilingual
[18] https://github.com/mattmikolay/rapture
[19] http://www.babylscript.com/

Table 1. A application of the framework to a set of programming languages

Aspect/Prog Language	Scratch	Snap!	Dolittle	Ebda3	قلب	Wenyan	Excel	PSeInt	Rapture	Hedy	Hindi	Linotte
Language	Multi	Multi	Japanese	Arabic	Arabic	Chinese	Multi	Spanish	Russian	Multi	Hindi	French
Alignment	N	N	N	N	T	N	T	N	T	N	T	N
Non-English keywords	●	●	●	●	●	●	●	●	●	●	●	●
Non-Latin variable names	●	●	●	●	●	●	●	○	●	●	●	●
Non-English productions		◑	●	-	-	●	○	●	-	-	-	●
Non-English numerals	-	-	-	-	●	●	●	-	-	●	-	-
Characters without meaning	-	-	-	-	●	●	-	-	-	◑	-	-
Diacritics	●	●	○	◑	●	○	-	●	○	●	●	●
Alternative keywords	-	-	-	◑	-	●	-	●	◑	●	-	-
Localized punctuation	○	○	◑	◑	◑	●	●	-	-	●	-	-
Right to left support	◑	-	○	●	●	○	●	-	-	●	○	◑
Multi-lingual programming	-	-	-	-	-	-	-	-	●	◑	-	-
Error messages	○	○	●	◑	◑	-	●	●	-	●	●	●

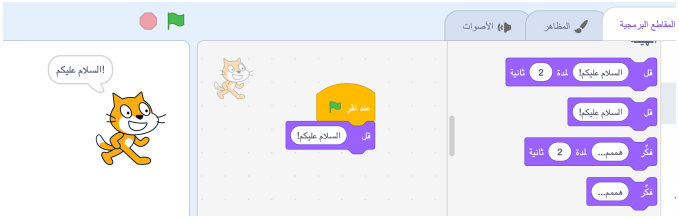

Figure 5. Scratch rendering the exclamation mark in two ways, correct in the palette and in the programming field, but incorrect in the execution.

natural way. This customization however is not available for built-in blocks.

4.3 Dolittle

Dolittle is a Japanese programming language which has a lot of good support for an authentic Japanese programming experience, such as a Japanese word order in productions. Interestingly enough, they do still use some English concepts in the language and the examples, such as English numerals. While their code uses the Japanese versions of characters, such as Japanese round brackets, they also use exclamation marks. Classroom experiments by the authors of Dolittle show that the use of exclamation mark is confusing to students, since it is not commonly used in Japanese [24].

4.4 Ebda3

Ebda3 is an Arabic programming language and is inspired in its style by multiple other programming languages such as C++ and Python. The only type of diacritics the language supports is the *hamza* (ء) which is a sign in the Arabic script that represents the glottal stop. This sign can be a standalone letter, but it can be added on top or below the basic three vowels in Arabic. Some of Ebda3 keywords include the *hamza* sign with the vowels, for example أما and أكتب. Ebda3 is diacritic sensitive, the equivalent اما and اكتب, without the *hamza*, will produce syntax errors. However, there is only one special case where Ebda3 allows multiple versions of a keyword. This happens when writing some letters with dots, a special type of Arabic diacritics, and because of historical differences between standard and Egyptian Arabic on how to write those letters. An example is the keyword إلى which is the translation of 'to', as mentioned before in Section 3.8.

In the same way, Ebda3 partially supports Arabic specific punctuation. For example, the Arabic comma, that we mention in Section 3.9, is supported as a list separator. However, the decimal comma, which is similar to the ones used in Dutch ',' is not supported. Only the dot is accepted as a decimal separator. Moreover, the tatweel character (see Section 3.6) is not allowed in keywords. Finally, although Ebda3 is a programming language targeting Arabic speakers, the production of the sentences is affected by the English-based programming languages.

4.5 قلب

قلب [20] supports Arabic symbols such as the Arabic question mark and comma. It is also the only language we encountered apart from Hedy that accepts Arabic numerals, however, it only accepts these numerals and not the English ones. We think this is because the language is made as part of an art project to show the beauty of the Arabic language in a digital context, hence the focus on Arabic.

Similar to Ebda3, قلب accepts keywords with the *hamza* such as إذا, also being diacritic sensitive in this case. To the best of our knowledge, قلب is the only Arabic language that allows the tatweel character (see Section 3.6) in keywords, and so does Hedy when used in Arabic. The error messages

[20] Transliterated as 'Qalb', pronounced as 'Elb' and meaning heart.

are mostly Arabic, but sometimes they include English syntactic terminology such as `null`. Right to left support is supported as expected. The localized punctuation is partially supported though. The language supports the Arabic question mark in multiple keywords such as عدم؟ and ؟أصغر. The Arabic comma is also supported but as a decimal separator, which is not the standard way of writing decimal numbers in Arabic, which typically uses the Latin comma in numbers (٣,١٤ is 3,14 for pi rounded to 2 decimals).

Because it is a translation of lisp, the productions sometimes sound a bit counter-intuitive when compared to native Arabic examples. For example, for some multi-word keywords the creator replaced the space between the words with a hyphen: ؟أكبر-أو-يساوي (meaning 'greater or equal?'). This would not be usually used in this way, and breaks the flow of the ligatures in the words.

4.6 Wenyan

Wenyan is by far the most 'localized' language considered in this paper [7]. It is designed to support a fully Chinese programming experience and succeeds at that well, allowing for Chinese order of keywords in productions, working with Chinese numerals, and the addition of characters that add meaning but not semantics like 者. It also allows for alternatives, for example both 吾有 and 今有 can be used to declare a variable, differing in their meaning only slightly. Given that, is interesting that Wenyan does not have Chinese error messages, but English ones. As such, it would be very hard to actually use by people without an understanding of English, especially learners.

4.7 Excel

Excel is probably the most widely used programming language that supports extensive localization. Non-English numerals can be used and work out of the box, if you sum two numbers using Arabic numerals, the result will also be presented in Arabic numerals, and one worksheet might even mix English numerals with other systems. Localized versions also support different decimal separators, although these, for obvious reasons, cannot be mixed. The biggest missing feature might be the lack of multilingual support; one cannot mix one of the dozens of languages it supports with each other, or with English.

4.8 PSeInt

PSeInt supports diacritics, although not explicitly mentioned in their documentation. The language is diacritic insensitive in that sense: the keyword `Segun`, for example, can be written as `Según`.

PSeInt provides an 'flexible syntax' option. When that option is enabled, the syntax can be closer to Spanish sentence production, and different keywords should be used for different grammatical rules, as such:

```
Algoritmo example
    var_a  Es ENTERO
    var_x, var_y  Son ENTEROS
FinAlgoritmo
```

In the previous example, PSeInt allows the use of different verbs Es and Son, equivalent to Is and Are in English, to reflect the number of variables being defined. The same applies for the integr type used, in singular and plural forms, ENTERO and ENTEROS. The language is sensitive to the native grammatical rules, thus a syntax error will be shown when using the Es instead of Son, and vice versa.

Spanish-specific symbols are not allowed, such as the comma for decimals or the special characters ¿ and ¡ indicating the beginning of a question or an exclamatory sentence respectively. PSeInt is designed for Spanish speakers in mind, that is why we do not find support to non-Latin and Right-to-left text and numerals in the language.

4.9 Rapture

Rapture is a modern implementation of an old multi-lingual language, Rapira, that was Russian, Moldovan, and English. Rapture as a modern implementation however supports Russian and English only. There is one special case in where Rapture translates multiple English keywords into one Russian keyword: the English version uses the terminating lexeme `fi` for the `if` statement, and similarly `esac` to end the `case` statement. However, the Russian version has the '`все`' as the terminating lexeme for both the `if` and `case` statements. Moreover, Rapture is one of only two languages in Table 1 that support multi-lingual programming. One can use keywords in English and Russian and freely mix them.

4.10 Hedy

Hedy [13, 17] is a gradual language made for education, using language levels leading up to Python. Hedy was partly multi-lingual from its inception in 2020 and has since added a number of features to support localization. Hedy supports some characters without direct meaning, for example it can handle the tatweel within Arabic keywords. Hedy also support alternative keywords in several situations, for example both the male and female versions of `is` in Arabic. Hedy keywords are also not diacritic sensitive, in French one may use `repete` or `repétè` for a loop. Localized non-letter characters can also be processed, such as the Arabic comma.

Multi-lingual programming is supported, but only English can be mixed with other languages. Mixing two non-English languages is not supported.

4.11 Hindi

Hindi supports non-English numerals partially, including Hindi numerals (see Figure 3.4). When using them in programs, no syntax error is shown and thus the program passes

the build phase. However, a run-time error causes the program to crash. The language does support diacritics in keywords such as पंक्ति, Hindi is diacritic sensitive in this case.

4.12 Linotte

Linotte allows for diacritics in keywords, such as 'carré', 'déplace', and 'espèces'. However it is diacritic sensitive, which means 'carre' without the accent, for instance, is not accepted and a syntax error is shown. Linotte does not support French specific symbols, for example the comma's in decimal numbers and the guillemets, the special quotation marks common in French text « and ». Linotte presents itself as a language that is close to the natural French, and that it is easy to start learning programming for children with the language. For that, we see they allow the programmer to be closer to the sentence production in French by using articles. The language support the following articles that can be used when defining a variable of a built-in or custom datatype : 'le', 'la', 'l' ', 'les', 'du', 'un', 'une ", 'mon', 'ma', 'ton', 'ta', and 'tes'. One can use them in this way:

`prénom est un texte` (meaning: name is a text)
`prénom prend "Clément"`(meaning: name takes "Clément")

The language follows native language rules and grammar when using the articles. As a result, a random article cannot replace the 'un' in the example above. The articles' use is optional, though. One can write the variable definition above as `prénom est texte`. Although the programming language is french they have a partial support for other native languages. Variable names can be non-Latin, also string variables can contain text in non-Latin languages: the editor and the language show them normally as outputs.

4.13 Summary

In summary, we see that while many languages cater to some aspects of localization, they often do not do well on all aspects. Especially notable are numerals, which are rarely supported, even in non-English languages, while, technically, this would be relatively easy to address; certainly easier than localized keywords in a textual language. Multilingual programming is also rarely considered, even languages which allow for programming in multiple languages, such as Scratch, Snap! and Excel, do not allow for the mixing of languages. While this choice is understandable, since allowing different languages would require more effort, and count lead to confusing situations, the absence of multi-lingual support means that these languages do not cater for various scenarios including teaching and multi-lingual business contexts.

5 Discussion

In this section we discuss a variety of factors take can be taken in to consideration when localizing languages. It also discusses threads to validity.

5.1 Arguments against localization of programming languages

Exploring the technical challenges of localizing programming languages is always done in context of the question whether it is 'needed' or 'good' to be inclusive of non-English programmers by supporting them to program in their own native language. Since we have been working on this topic, we have received consistently received push back on the idea (including in the reviews for this paper), so here we collect and partly rebut some arguments.

5.1.1 "Just" Learn English. The often heard reason against lozalication is that if you want to program, you should just learn English.[21] This is certainly possible: the millions of people speaking non-English native languages that are programmers (including both authors of this paper) are exemplars of that. However, there are audiences for whom this is less feasible, most notably learners, but also casual or end-user programmers. They might not have the time or energy to learn programming and also English. Also, people don't know what they don't know. Trying to use your own numerals, quotes or comma's and subsequently being confronted with error messages might lead people to conclude programming is hard, or will not serve their needs, and they might drop out before realizing they had to use other characters.

5.1.2 The community will be split. This is an often heard argument, which is also noted by Atwood: *'If everybody blogs and develops in English ... you will have better chances of finding an answer to your problem.'*. Interestingly enough, on our personal experience, this question is exclusively asked by people with a great command of English, because they have grown accustomed to be able to access the repository of information on programming with ease. People who struggle to read English do not worry about a future world in which a lot of content is not readable to them, since that is already the case for them.

5.1.3 Machine translation will fix this. A final argument that we often here is that translating programming languages is not needed, since machine translations are, or will soon be, good enough to solve the issue. Sadly however, it is a known fact that machine translation does not work so well for languages with smaller populations and smaller online repositories (often because of lower internet connectivity among their speakers)[14, 19]. Also, since machine translation tools are built both by the community also

[21] https://blog.codinghorror.com/the-ugly-american-programmer/

building today's programming languages, and by using today's programming languages, it is very likely that they will have issues with non-English letters, numbers and characters similar to those described in this paper. [22]

5.2 The importance of multi-lingual programming

One argument that is often named by programmers as a argument against localization of programming languages, is that it is confusing and potentially error-prone to go back and forth between languages. This highlights why one of our aspects, namely multi-lingual programming, is more important than one might assume. If you know keywords in multiple languages, the easiest default is not being forced to go back and forth, and being allowed to use keywords in multiple languages in one formula or program. Especially for learners, gradually introducing more English keywords is likely going to be less painful than a full switch at a certain point.

5.3 Transfer from non-English

A relatively recent topic in programming education is research into how to transfer from one *programming* language to another [38, 39]. However, we do not yet have literature on how to best transfer from a localized programming language into an English one. In transfer research, two different strategies are often described: *hugging* which is decreasing the distance between the two domains, and *bridging* which is explicitly encouraging noticing the similarities between concepts. It is not clear what hugging and bridging would look like in the transfer into an English programming language. One can imagine a hugging strategy in which a learner write a program in their native language first and then translates in into English, or a bridging strategy in which a problem is solved in more abstract notation such as a flow chart and then into the new, English, programming language, but different ways are imaginable too, and which is more effective is a topic for future work. It is also not at all clear at what moment in time a potential switch should take place, after a learned concept, or after mastering a larger set of concepts.

5.4 Ergonomic issues

In this paper we have described aspects of languages that could be localized. However, the design of languages usable for non-English speakers also has ergonomic factors.

Firstly, there is the assumption which characters will be easily available on keyboard. Not all keyboards, for example, have direct access to the back tick `, Italian keyboard typically lack it, requiring you to using ALT and two numeric keys, making it a less than ideal choice for programming language syntax.

Furthermore it is very common for European keyboard setting to use a 'dead key' mechanism to be more easily able to create accented letters (diacritics), such as á or ë. A dead key mechanism means that when pressing a dead letter key (often the back tick, or single or double quote) the character does not appear immediately. Instead the next character is awaited, if that is a letter on which the diacritic could be placed, it is placed on it. If the letter does not have the option to be modified (k, for example) both the symbol and letter appear after the second key press. As such, typing code such as `print('allo')` without paying attention will result in `print(állo')` which does not compile. Choosing a single quote as string delimiter thus has an ergonomic impact for programming in certain default language settings. Of course, these keyboard settings can be changed, however that too requires both considerable computer skills and a single use computer which not everyone has access to.

5.5 Threats to validity

This paper has chosen to extract the aspects of the framework from existing programming languages, and as such does not describe aspects that no languages support yet. Furthermore, some programming languages could be missed since non-English languages often also have non-English documentation limiting the accessibility. This paper also limits its view to aspects of programming languages, and does not incorporate editors in which these languages are written.

6 Concluding remarks

The goal of this paper is to introduce a framework encompassing aspects of programming languages that can be localized to non-English. This framework can help language designers to make informed decisions about localization, and help decision makers better choose languages for various non-English contexts. Our framework includes the localization of keywords, numerals, programming constructs and support for multi-lingual programming.

Future work can be imagined in various directions. Firstly, the set of programming languages that our framework is applied to could be extended, studying multiple programming languages in the same natural language or implementing the same aspects, giving an even more in-depth view of the field. Furthermore, our framework can be enriched to include best practices on how to support the localized aspects, both in terms of programming language design and in terms of implementation. We could also explore best practices in teaching with localized programming languages, especially in the interesting case of mixed-language classrooms.

[22] Note that both these papers were pre-prints at the time of this paper's submission.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[2] Andreu Balius. 2013. *Arabic type from a multicultural perspective: Multi-script Latin-Arabic type design.* Ph.D. University of Southampton. https://eprints.soton.ac.uk/355433/1/Final%2520PhD%2520thesis_Andreu%2520Balius.pdf

[3] L.S. Baraz, E.V. Borovikov, N.G. Glagoleva, P.A. Zemtsov, E.V. Nalimov, and V.A. Tsikoza. 1987. Rapira Programming Language. (1987). http://ershov.iis.nsk.su/ru/node/772586

[4] Youssef Bassil. 2019. Phoenix - The Arabic Object-Oriented Programming Language. 67, 2 (2019), 7–11. https://doi.org/10.14445/22312803/IJCTT-V67I2P102

[5] David Bau, D. Anthony Bau, Mathew Dawson, and C. Sydney Pickens. 2015. Pencil Code: Block Code for a Text World. In *Proceedings of the 14th International Conference on Interaction Design and Children* (Boston, Massachusetts) *(IDC '15)*. Association for Computing Machinery, New York, NY, USA, 445–448. https://doi.org/10.1145/2771839.2771875

[6] Mrwan Ben Idris and Hany Ammar. 2018. The Correlation between Arabic Student's English Proficiency and Their Computer Programming Ability at the University Level. 9 (2018), 01–10. https://doi.org/10.5121/ijmpict.2018.9101

[7] Charles Q. Choi. 2020. World's First Classical Chinese Programming Language. *IEEE Spectrum* (2020). https://spectrum.ieee.org/classical-chinese

[8] Nils Danielsson and Ulf Norell. 2008. Parsing Mixfix Operators, Vol. 5836. 80–99. https://doi.org/10.1007/978-3-642-24452-0_5

[9] Sayamindu Dasgupta and Benjamin Mako Hill. 2017. Learning to Code in Localized Programming Languages. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale* (2017-04) *(L@S '17)*. Association for Computing Machinery, 33–39. https://doi.org/10.1145/3051457.3051464

[10] Hanan Elazhary. 2009. Arabic Lisp. In *Proceedings of the 21st International Conference on Software Engineering & Knowledge Engineering (SEKE'2009)* (Boston, Massachusetts, USA, 2009-07-01). Knowledge Systems Institute Graduate School, 382–385.

[11] Alejandro Espinal, Camilo Vieira, and Valeria Guerrero-Bequis. [n. d.]. Student ability and difficulties with transfer from a block-based programming language into other programming languages: a case study in Colombia. 0, 0 ([n. d.]), 1–33. https://doi.org/10.1080/08993408.2022.2079867

[12] Pedro Guillermo Feijóo-García, Keith McNamara, and Jacob Stuart. 2020. The Effects of Native Language on Block-Based Programming Introduction: A Work in Progress with Hispanic Population. In *2020 Research on Equity and Sustained Participation in Engineering, Computing, and Technology (RESPECT)*, Vol. 1. 1–2. https://doi.org/10.1109/RESPECT49803.2020.9272513

[13] Marleen Gilsing and Felienne Hermans. 2021. Gradual Programming in Hedy: A First User Study. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2021-10). 1–9. https://doi.org/10.1109/vl/hcc51201.2021.9576236

[14] Nuno M Guerreiro, Duarte Alves, Jonas Waldendorf, Barry Haddow, Alexandra Birch, Pierre Colombo, and André FT Martins. 2023. Hallucinations in large multilingual translation models. *arXiv preprint arXiv:2303.16104* (2023).

[15] Philip J. Guo. 2018. Non-Native English Speakers Learning Computer Programming: Barriers, Desires, and Design Opportunities. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (2018) *(CHI '18)*. Association for Computing Machinery, 1–14. https://doi.org/10.1145/3173574.3173970 event-place: New York, NY, USA.

[16] Khendum Gyabak and Heriberto Godina. 2011. Digital storytelling in Bhutan: A qualitative examination of new media tools used to bridge the digital divide in a rural community school. *Computers & Education* 57, 4 (Dec. 2011), 2236–2243. https://doi.org/10.1016/j.compedu.2011.06.009

[17] Felienne Hermans. 2020. Hedy: A Gradual Language for Programming Education. In *Proceedings of the 2020 ACM Conference on International Computing Education Research* (2020) *(ICER '20)*. Association for Computing Machinery, 259–270. https://doi.org/10.1145/3372782.3406262 event-place: New York, NY, USA.

[18] Mohamed Hssini and Azzeddine Lazrek. 2011. Design of Arabic Diacritical Marks. *IJCSI International Journal of Computer Science Issues* 8, 3 (May 2011). https://ijcsi.org/papers/IJCSI-8-3-2-262-271.pdf

[19] Viet Dac Lai, Nghia Trung Ngo, Amir Pouran Ben Veyseh, Hieu Man, Franck Dernoncourt, Trung Bui, and Thien Huu Nguyen. 2023. Chatgpt beyond english: Towards a comprehensive evaluation of large language models in multilingual learning. *arXiv preprint arXiv:2304.05613* (2023).

[20] Outi Laiti. 2016. The Ethnoprogramming Model. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '16)*. Association for Computing Machinery, New York, NY, USA, 150–154. https://doi.org/10.1145/2999541.2999545

[21] Luis A. Leiva and Vicent Alabau. 2015. Automatic Internationalization for Just In Time Localization of Web-Based User Interfaces. 22, 3 (2015). https://doi.org/10.1145/2701422

[22] William Judson LeVeque and David Eugene Smith. 2022. *Numerals and numeral systems.* Encyclopedia Britannica.

[23] Aaron Marcus, Nuray Aykin, Apala Lahiri Chavan, Donald L. Day, Emilie West Gould, Pia Honold, and Masaaki Kurosu. 2000. Cross-Cultural User-Interface Design: What? So What? Now What?. In *CHI '00 Extended Abstracts on Human Factors in Computing Systems* (2000) *(CHI EA '00)*. Association for Computing Machinery, 299. https://doi.org/10.1145/633292.633468 event-place: New York, NY, USA.

[24] Hu. Ming and Emi Keiji. 2017. Educational Report of Programming Language Dolittle for Foreign Students. *Journal of the Native American and Indigenous Studies Association* 11 (Jan. 2017), 30–33.

[25] Titus Nemeth. 2019. *On Arabic justification, part 1 – a brief history.* https://research.reading.ac.uk/typoarabic/on-arabic-justification-part-1/

[26] Momed A. A. Neves and Seraphin Desire Eyono Obono. 2013. On the perceived usefulness of the localization of compilers in African indigenous languages. (Feb. 2013). https://doi.org/10.7763/IJIET.2013.V3.243 Accepted: 2014-06-24T10:46:24Z Publisher: IJIET.

[27] Ibrahim Nnass, Michael A. Cowling, and Roger Hadgraft. 2022. Identifying the Difficulties of Learning Programming for Non-English Speakers at CQUniversity and Sebha University. *Journal of Pure & Applied Sciences* 21, 4 (Oct. 2022), 290–295. https://doi.org/10.51984/jopas.v21i4.2258 Number: 4.

[28] Yogendra Pal and Sridhar Iyer. 2015. Effect of Medium of Instruction on Programming Ability Acquired through Screencast. In *2015 International Conference on Learning and Teaching in Computing and Engineering*. 17–21. https://doi.org/10.1109/LaTiCE.2015.38

[29] Yizhou Qian, Peilin Yan, and Mingke Zhou. 2019. Using Data to Understand Difficulties of Learning to Program: A Study with Chinese Middle School Students. In *Proceedings of the ACM Conference on Global Computing Education* (2019) *(CompEd '19)*. Association for Computing Machinery, 185–191. https://doi.org/10.1145/3300115.3309521 event-place: New York, NY, USA.

[30] Hussam Hatem Abdul Razaq, Ayedh Shahadha Gaser, Mazin Abed Mohammed, Esam Taha Yassen, Salama A. Mostafad, Subhi R. M. Zeebaree, Dheyaa Ahmed Ibrahim, Mohd Khanapi Abd Ghania, Rabah N. Farhan, Hussam Hatem Abdul Razaq, Ayedh Shahadha Gaser, Mazin Abed Mohammed, Esam Taha Yassen, Salama A. Mostafad, Subhi R. M. Zeebaree, Dheyaa Ahmed Ibrahim, Mohd

Khanapi Abd Ghania, and Rabah N. Farhan. 2019. Designing and Implementing an Arabic Programming Language for Teaching Pupils. 54, 3 (2019). https://doi.org/10.35741/issn.0258-2724.54.3.11

[31] Kyle Reestman and Brian Dorn. 2019. Native Language's Effect on Java Compiler Errors. In *Proceedings of the 2019 ACM Conference on International Computing Education Research (ICER '19)*. Association for Computing Machinery, New York, NY, USA, 249–257. https://doi.org/10.1145/3291279.3339423

[32] Bertrand Roehner. 2015. Translation into any natural language of the error messages generated by any computer program. https://doi.org/10.48550/arXiv.1508.04936 arXiv:1508.04936 [cs].

[33] Patricia Russo and Stephen Boor. 1993. How Fluent is Your Interface? Designing for International Users. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems* (1993) *(CHI '93)*. Association for Computing Machinery, 342–347. https://doi.org/10.1145/169059.169274 event-place: New York, NY, USA.

[34] Huda Smitshuijzen AbiFarès. [n. d.]. *The Big Kashida Secret*. https://www.khtt.net/en/page/1821/the-big-kashida-secret

[35] Adalbert Gerald Soosai Raj, Eda Zhang, Saswati Mukherjee, Jim Williams, Richard Halverson, and Jignesh M. Patel. 2019. Effect of Native Language on Student Learning and Classroom Interaction in an Operating Systems Course. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '19)*. Association for Computing Machinery, New York, NY, USA, 499–505. https://doi.org/10.1145/3304221.3319787

[36] Andreas Stefik and Richard Ladner. 2017. The Quorum Programming Language (Abstract Only). In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (2017) *(SIGCSE '17)*. ACM, 641–641. https://doi.org/10.1145/3017680.3022377 event-place: New York, NY, USA.

[37] Alaaeddin Swidan, Alexander Serebrenik, and Felienne Hermans. 2017. How do scratch programmers name variables and procedures?. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)* (2017). IEEE, 51–60. https://doi.org/10.1109/scam.2017.12

[38] Ethel Tshukudu and Quintin Cutts. 2020. Semantic Transfer in Programming Languages: Exploratory Study of Relative Novices. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (2020-06) *(ITiCSE '20)*. Association for Computing Machinery, 307–313. https://doi.org/10.1145/3341525.3387406 event-place: Trondheim, Norway.

[39] Ethel Tshukudu, Quintin Cutts, Olivier Goletti, Alaaeddin Swidan, and Felienne Hermans. 2021. Teachers' Views and Experiences on Teaching Second and Subsequent Programming Languages. https://eprints.gla.ac.uk/250525/ Conference Name: 17th ACM Conference on International Computing Education Research (ICER 2021) ISBN: 9781450383264 Meeting Name: 17th ACM Conference on International Computing Education Research (ICER 2021) Pages: 294-305 Publisher: ACM.

[40] Sara Vogel, Christopher Hoadley, Ana Rebeca Castillo, and Laura Ascenzi-Moreno. 2020. Languages, literacies and literate programming: can we use the latest theories on how bilingual people learn to help us teach computational literacies? *Computer Science Education* 30, 4 (Oct. 2020), 420–443. https://doi.org/10.1080/08993408.2020.1751525 Publisher: Routledge _eprint: https://doi.org/10.1080/08993408.2020.1751525.

Received 2023-07-27; accepted 2023-08-24