philosophies

MDPI

Article

New Programming Styles Suggested by Human Languages

Baptiste Mélès

CNRS, Archives Henri-Poincaré—PReST (UMR 7117), Université de Lorraine, Université de Strasbourg, 91 avenue de la Libération, BP 454, F-54001 Nancy Cedex, France; baptiste.meles@univ-lorraine.fr

Abstract

Can human languages help us write programs in a different way than we usually do? To examine this question, we first define exactly what it means for a programming language to be "derived from" a human language. Next, we analyse cases in which translating a program from one human language to another does not significantly change the program's structure. Finally, we examine two game-changing cases: a programming language derived from Latin, in which syntax plays a limited role compared to morphology, and another derived from Classical Chinese, in which little linguistic recursion is available. These examples show that human languages, even ancient ones, are a reservoir for innovation in program writing. One can encourage programming language designers to dare learn foreign languages and not be ashamed of their own native language.

Keywords: programming language; natural language; morphology; syntax; recursion; Latin; Classical Chinese

1. Introduction

Even though programming languages are formal languages, they are mostly derived from human languages, typically English. The preponderance of English may raise educational issues [1–4]. Moreover, even many programming languages that are derived from other languages are just word-for-word translations of an English-based programming language. Might language switching result in more innovative programming styles? Or, in other words, can the diversity of human languages help us write programs in a different way?

To examine this question, we first precisely define what it means for a programming language to be "derived from" a human language. Then, we analyse cases in which translating a program from one human language to another does not fundamentally change its structure. Finally, we examine two game-changing cases: a programming language derived from Latin, in which syntax plays a limited role compared to morphology, and another derived from Classical Chinese, in which little linguistic recursion is available. These examples show that human languages, even ancient ones, are a reservoir for innovation in program writing.

2. What Programming Languages May Borrow from Human Languages

One might be tempted to argue that programming languages are independent of human languages. The argument is that they are formal languages, and that formal languages do not belong to any particular language. We will challenge this argument.

First, let us make some conceptual distinctions between kinds of languages:

Check for updates

Academic Editors: Raymond Turner and Henri Stephanou

Received: 29 September 2025
Revised: 11 March 2026
Accepted: 18 March 2026
Published: 7 April 2026

Copyright: © 2026 by the author. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

- By a *human language*, we mean a general-purpose communication system used by a human group.[1] Among these, we can distinguish between *natural languages*, which mainly evolved without planning (e.g., French or Italian), and *artificial languages*, which are mainly developed in a planned manner (e.g., Esperanto or Volapük);
- On the other hand, a *formal language* is a communication system defined by an alphabet and a set of formal rules, such as the language of first-order logic or arithmetic; among these, a *programming language* is a formal language designed to determine the behaviour of a machine.[2]

One might think that no formal language can be "derived from" a human language at all, since it can be constructed from any alphabet with arbitrary rules, generally motivated by scientific purposes. For instance, the language of arithmetic is not obviously derived from any particular human language. Similarly, the code in Figure 1 appears to have nothing in common with any human language, if we ignore the two comments at the end of lines 2 and 4 [9].

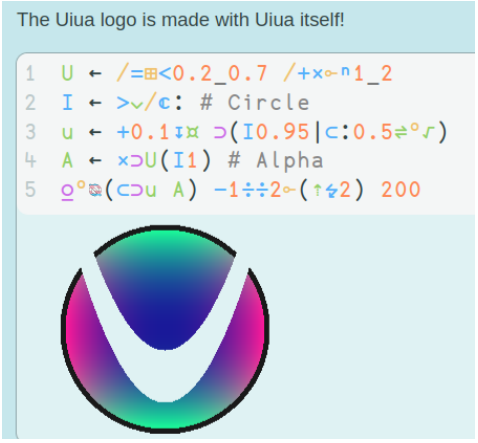


Figure 1. UIUA code drawing the UIUA programming language logo. Apart from the comments, preceded by hash marks, code written in this language clearly does not belong to any human language.

However, the fact that formal languages *need not* be derived from any particular human language does not imply that they *are not*. Nothing in the definition of a formal language prohibits it from borrowing "more or less" from a human language. In fact, English happens to predominate in programming languages, a predominance which is sufficient to pose educational problems [1–4]. By way of illustration, we encourage non-Irish-speaking readers to imagine how difficult the Java language would be for them if its keywords had been borrowed from Irish instead of English [4]. Even if the syntax were the same as it is today, the Irish lexicon would make it less easily readable at first sight:

```
poiblí aicme DiaDuitDomhan{
    poiblí statach folús príomh(Sreang[] args){
        // Priontáil "Dia duit, Domhan" go dtí an fuinneog teirminéal
        Córas.amach.priontáilln("Dia duit, Domhan");
    }
}
```

This example shows that our knowledge of the human language on which a programming language is based plays a role in the readability of the code.

What precisely does it mean to be "derived from" a human language? This expression has many meanings, since human languages can be studied from many different perspectives. To begin with *spoken human languages*, e.g., oral English, their various aspects are the subject of different branches of linguistics [10] (p. xvii), [11] (p. 149), [12] (pp. xiv–xv):

1. *Phonetics* breaks down the verbal chain into phones, or units of sound like [t] and [t’];
2. *Phonology* studies phonemes, the smallest phonetic units distinguished by a given language (e.g., English speakers do not consider the two T’s of the word *potato* as different, even though one is aspirated and the other is not);
3. *Lexicology* studies the lexicon, i.e., the set of words;
4. *Morphology* studies word variation and formation, e.g., inflection (declination, conjugation) and word composition;
5. *Syntax* studies the formation of sentences, mainly word order;
6. *Semantics* studies the relationship between linguistic expressions and their meaning;
7. *Pragmatics* studies how expressions are used in given social contexts.

The same seven dimensions can be found in *written human languages*, with the exception of the first two: phonetics is replaced by *graphetics* (1′), the study of shapes, i.e., the ways in which characters are represented, and phonology by *graphematics* (2′), the study of graphemes, i.e., the characters distinguished by a given written language. For instance, in written English, two tokens of the letter T having different shapes or sizes are by default considered as the same letter.

These seven dimensions of written human languages can, in turn, be used to describe *textual features of programming languages*[3]:

1. *Graphetics* determines the glyphs, i.e., the appearance that the characters of the language can take (e.g., highlighting in development environments);
2. *Graphematics* determines the alphabet, i.e., the set of characters allowed in and relevant to the language;
3. *Lexicology* determines the set of keywords;
4. *Morphology* governs the formation of lexemes, e.g., `rabbit.colour` in object-oriented programming;
5. *Syntax* governs the formation of expressions;
6. *Semantics* determines the interpretation of expressions;
7. *Pragmatics* describes programming practices, for example, the “best practices” associated with a given language.

These conceptual distinctions make it possible to specify some aspects by which most programming languages can be considered to be derived from the English language.

- At the *graphematic* level, a programming language may borrow its alphabet from a language. English is probably the only language whose characters, from “A” to “Z” and “a” to “z”, are admitted in almost all programming languages. Indeed, many programming languages do not accept characters such as the French “é”, the Greek “α”, the Ukrainian “ґ”, or the Chinese “思”. As a consequence, a French speaker writing a C program cannot name a variable `affiche` standing for the past participle “affiché” (“displayed”) without risking the reader confusing it with a function name based on the imperative verb “affiche” (“display”). Of course, programmers wanting to use non-English identifiers will not face this problem with all programming languages, e.g., Python accepts accentuated, Ukrainian, and Chinese characters. Our point is that—maybe only—English speakers will, so to say, never face this problem in any language;
- At the *lexical* level, most programming language keywords are borrowed from English: `if`, `for`, `break`, etc.;[4]
- At the *morphological* level, many programming languages share with English the absence of inflectional morphology: keywords generally do not change form according to their function in an expression. On the other hand, the compositional morphology found in object-oriented programming languages often adopts the Germanic

determiner–determined order (for example, `bus.station` follows the order of the English expression "bus station");

- At the *syntactic* level, many programming languages share with English the verb–complement structure—the name of the function being written before its arguments, for example, `print("Hello world\n")` meaning "print "Hello world""—or subject–verb–complement in object-oriented programming, for example, `girl.say("Hello world\n")` meaning "the girl says "Hello world"".

Such features may make programming languages more familiar to English speakers than others. They can perform "language transfer", i.e., transfer parts of their linguistic knowledge and habits from English to such programming languages, in the same way that native speakers of Italian are able to transfer to French some lexical, morphological and syntactical knowledge coming from their mother tongue. This problem is particularly sensitive for non-English-speaking children learning how to program. Professional programmers, given their age and level of education, have generally acquired familiarity with English.

One might wonder what justifies this predominance of English. Might there be reasons intrinsic to English? Peter Naur thinks that English has properties that are advantageous for programming. He takes up Otto Jespersen's thesis that languages evolve by becoming increasingly analytical, i.e., morphemes tend to be expressed in different words rather than in variations of word forms [14] (pp. 678–680), [15].[5] Analyticity is a desirable property in programming languages: different ideas should be expressed separately from one another. The high analyticity of English would, thus, predispose it to dominate the world of programming languages—but Chinese, a language completely devoid of declension and conjugation,[6] would have been an even better choice according to this criterion, as it would have avoided barbarisms such as `girl.say("Hello world\n")` or `girls.says("Hello world\n")`.

An extrinsic explanation for the predominance of English in programming languages is that the United States of America, where English is the primary language, is where most programming languages originate,[7] and that on a global scale, English is used as an international language, so much so that Python, created in the Netherlands, and Rocq (formerly Coq), created in France, rely on this human language to increase their chances of dissemination. Cultural reasons may, therefore, be sufficient to explain the predominance of English features and patterns in programming [18].

Thus, even though there is no conceptual requirement to do so, most programming languages are, in fact, derived from a particular language. Most of them borrow from English at least parts of their graphematics, lexicon, morphology and syntax. Now, is programming really different when it is derived from languages other than English?

3. Non-Idiomatic Variations

In a significant number of cases, programming languages derived from other languages show only marginal variations compared to their English counterparts. They are limited to the lexicon or a very superficial part of the syntax, as we will show with Rust and Scratch variants.[8]

3.1. Word-for-Word Translation in Rust Variants

In many cases, linguistic variation among programming languages is limited to the lexical aspect. There are, for example, a large number of local variants of Rust.[9] However, looking closer at one of these variants, for example, Rouille in French [22], one sees in Figure 2 that it is simply a lexical, word-for-word translation of the English original Rust [22] (SoftWare Hash Identifier (SWHID): swh:1:cnt:3cfb164c93daf32ddcab9bffcf609fc1fa493ef8;lines=31-59). The lexicon is French (`utilisons` instead of `use`, `comme` instead of `as`, etc.), but the syntax is rigorously identical to that of Rust.

Similarly, the Ukrainian variant іржа *Irzha* is derived from Ukrainian *without declension*. The Ukrainian cases—nominative, vocative, accusative, dative, genitive, instrumental—which could have provided a great richness of expression, have not been exploited. The code in Figure 3 [20] SWHID: swh:1:cnt:af9ecb2075ab11349d7e8d6b5f6ed04fdcc9380f;lines=4 contains the genitive plural form слов "of the words", which allows for the clever creation of genitive constructions; but since this form is hard-coded in line 4, the nominative or accusative singular form слово "word" would not behave in this language as another form of the same word, but as a totally distinct word.

```
rouille::rouille! {
    utilisons std::collections::Dictionnaire comme Dico;

    convention CléValeur {
        fonction écrire(&soi, clé: Chaîne, valeur: Chaîne);
        fonction lire(&soi, clé: Chaîne) -> PeutÊtre<&Chaîne>;
    }

    statique mutable DICTIONNAIRE: PeutÊtre<Dico<Chaîne, Chaîne>> = Rien;

    structure Concrète;

    réalisation CléValeur pour Concrète {
        fonction écrire(&soi, clé: Chaîne, valeur: Chaîne) {
            soit dico = dangereux {
                DICTIONNAIRE.prendre_ou_insérer_avec(Défaut::défaut)
            };
            dico.insérer(clé, valeur);
        }
        fonction lire(&soi, clé: Chaîne) -> Résultat<PeutÊtre<&Chaîne>, Chaîne> {
            si soit Quelque(dico) = dangereux { DICTIONNAIRE.en_réf() } {
```

Figure 2. Code excerpt written in Rouille, a French translation of the Rust programming language. The keywords are translated word for word from English to French, e.g., "`utilisons` [...] `comme`" stands for "`use` [...] `as`", "`convention`" for "`trait`", "`fonction`" for "`function`", etc.

```
1  irzha::irzha! {
2      зовнішній ящик irzha;
3
4      вжити std::collections::Словник як Слов;
5
6      конвенція КлючЗначення {
7          функція записати(&власний, ключ: Строка, значення: Строка);
8          функція отримати(&власний, ключ: Строка) -> Результат<Можливо<&Строка>, Строка>;
9      }
10
11     статична змінна СЛОВНИК: Можливо<Слов<Строка, Строка>> = Нічого;
12
13     структура Конкретний;
14
15     реалізація КлючЗначення для Конкретний {
16         функція записати(&власний, ключ: Строка, значення: Строка) {
17             нехай слов = небезпечно {
18                 СЛОВНИК.отримати_або_вставити_із(Default::default)
19             };
20             слов.вставити(ключ, значення);
21         }
22         функція отримати(&власний, ключ: Строка) -> Результат<Можливо<&Строка>, Строка> {
23             якщо нехай Дещо(слов) = небезпечно { СЛОВНИК.як_посилання() } {
24                 Файно(слов.отримати(&ключ))
25             } інакше {
26                 Хиба("перевірте словник".до())
27             }
28         }
```

Figure 3. Code written in Irzha, a Ukrainian translation of the Rust programming language. Since the genitive plural form слов "of the words" is hard-coded in line 4, the nominative or accusative singular form слово "word" would be a totally distinct word.

Rust-style multilingualism, therefore, does not change the way programs are written: it is simply a word-for-word translation. This low level of multilingualism is very widespread.[10]

3.2. Local Syntactic Variations in Scratch

Linguistic variations sometimes affect not only the alphabet and lexicon, but also syntax, albeit superficially. This is the case with multilingualism in the Scratch programming language.

As shown in Figure 4, the Scratch programming environment menu allows the programmer to switch an entire program[11]—except, of course, variable names, character strings, etc.—into another language: English, German, Greek, Ukrainian, Japanese, Chinese, etc. [31,32].

When the program switches from one language to another, the first obvious change is in the alphabet used for keywords. It is replaced by Chinese characters (如, 果, 那, 么...), Cyrillic characters (а, б, в, г, ґ...), Greek (α, β, γ, δ...), Japanese (押, あ, ア…), Latin (a, b, c, d…), etc.

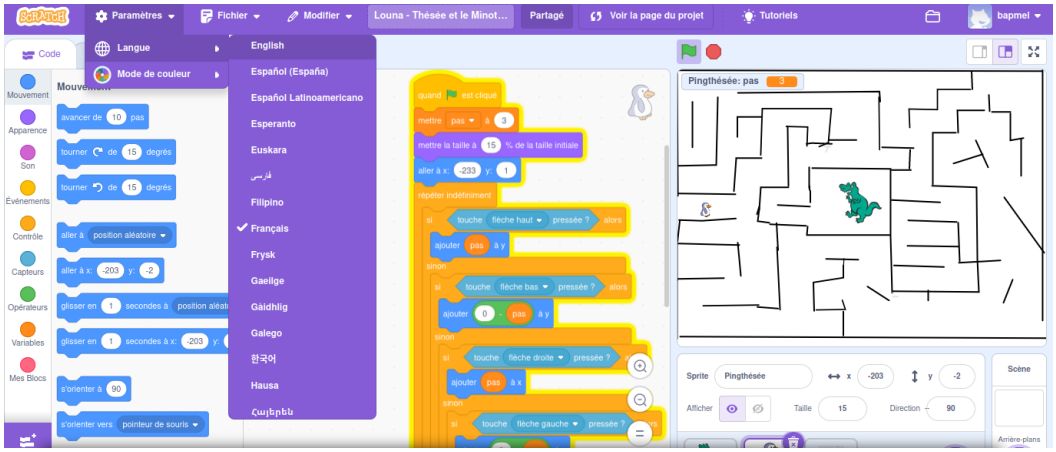


Figure 4. Language-switching menu in purple in the Scratch development environment. The programming palette, here with motion blocks, is on the left. The script, written in 2018 by Louna Mouzo (10 years old), describes the behaviour of the penguin sprite when the game starts. The stage is on the right. The purple language-switching menu is open: it allows the programmer to translate all keywords of the script to Spanish, Esperanto, French, etc.

The lexicon also varies. For example, the indefinite loop is expressed by the following phrases, where "B" indicates the position of the repeated instruction block (Table 1).

Table 1. Infinite loop in Scratch, in various languages. Keywords vary, but not the overall syntax: words meaning "forever" are followed by the block B.

de.	wiederhole fortlaufend	B
el.	για πάντα	B
en.	forever	B
es.	por siempre	B
fr.	répéter indéfiniment	B
it.	per sempre	B
ja.	すっと	B
pl.	zawsze	B
uk.	завжди	B
zhs.	重复执行	B

The syntax may also vary slightly from one language to another. For example, in the German version of the "if (Condition) then (Block) else (Block)" control structure, the keyword corresponding to "then", which exists in many other languages, is absent (Table 2).[12]

Table 2. "If" control structure in Scratch, in various languages. The overall syntax is the same in all these languages, even though the keyword "then" is translated by a mere concatenation in the German version.

de.	**falls**	**C**		**B**	**sonst**	**B**
el.	εάν	C	τότε	B	αλλιώς	B
en.	if	C	then	B	else	B
es.	si	C	entonces	B	si no	B
fr.	si	C	alors	B	sinon	B
it.	se	C	allora	B	altrimenti	B
ja.	もし	C	なら	B	でなければ	B
pl.	jeżeli	C	to	B	przeciwnym razie	B
uk.	якщо	C	то	B	інакше	B
zhs.	如果	C	那么	B	否则	B

We can also see in Table 3 that in variable assignment, Japanese uses postfix notation rather than prefix notation, contrary to other languages such as Polish.[13]

Table 3. Variable assignment in Scratch, in various languages. Unlike many other translations, e.g., to Polish, the Japanese version uses postfix keywords.

de.	setze	V	auf	X	
el.	όρισε	V	σε	X	
en.	set	V	to	X	
es.	dar a	V	el valor	X	
fr.	mettre	V	à	X	
it.	porta	V	a	X	
ja.		**V**	を	**X**	にする
pl.	**ustaw**	**V**	**na**	**X**	
uk.	надати	V	значення	X	
zhs.	将	V	设为	X	

Such local syntactic variations are frequent. For the event "Flag click" (noted "F"), Greek, Spanish, Italian and Polish use a prefix notation, Japanese uses a postfix notation, while English, German, French, Ukrainian and Simplified Chinese use a mixed notation (Table 4).

Table 4. Flag clicking event in Scratch, in various languages. All combinations are possible: some translations use a prefix notation, some others a postfix one, and the others a mixed one.

de.	wenn	F	angeklickt wird
el.	όταν γίνει κλικ σε	F	
en.	when	F	clicked
es.	al hacer clic en	F	
fr.	quand	F	est cliqué
it.	quando si clicca su	F	
ja.		F	が押されたとき
pl.	kiedy kliknięto	F	
uk.	коли	F	натиснуто
zhs.	当	F	被点击

However, all such variations remain marginal. In many cases, leaving English does not fundamentally change the way one programs: a change of alphabet, keywords, and local variations in syntax does not affect the overall syntax of the program.

Let us now examine two cases in which variation in human language makes possible more profound changes in program writing. The first is the Perligata programming language, a Perl module which, inspired by the inflectional morphology of Latin, offers

remarkable syntactic flexibility. The second example is the Wenyan programming language, which is inspired by Classical Chinese and uses a system of pronouns in order to limit the use of linguistic recursion.

4. Programming in Latin: Morphology at the Expense of Syntax

The Perligata programming language is described by its creator, Damian Conway, as "a programming language in which the syntax is (largely) non-positional" [33,34].

To enable programming in which word order is largely irrelevant, he introduced a feature that is extremely rare in programming: inflectional morphology, i.e., the variation in the form of words according to their function in an expression. Since the appearance of a word is sufficient to indicate its function in a given expression, it is not necessary for this function to be further indicated by its position in the expression.

This feature is expressly borrowed from Latin, a language with a rich inflectional morphology. It has a whole system of declensions: articles, pronouns, adjectives and nouns take different forms depending on their function in the sentence. A subject will be marked by the nominative *dominus*, a direct object by the accusative *dominum*, a noun complement by the genitive *domini*, an attributive complement by the dative *domino*, etc. Latin also has a system of conjugations: verbs change form according to tense, aspect and mood. This morphological richness allows for great syntactic flexibility:[14] the sentences *Mater panem dat*, *Mater dat panem*, *Panem mater dat*, etc., show only slight semantic or stylistic variations, such as the emphasis on a particular word. The predominance of inflectional morphology over word order was such that syntax itself, as part of grammar, only appeared late in Rome with Priscian (VIth century AD), long after morphological analyses such as Varro's (1st century BCE) [36].

The Perligata programming language is inspired by these characteristics, varying certain lexemes according to their function or context, which allows for considerable freedom in word order. The endings, borrowed from Latin, give the code a vaguely Latin appearance, admittedly closer to sword and sandal films than Virgil. See, for instance, the following implementation of the sieve of Eratosthenes, which prints out all prime numbers up to any given limit [34] SWHID: swh:1:cnt:53dad7ddaa7d941f0ffe70b244fd3ec89e246d38;lines=26-39:

```
#! /usr/bin/perl -w
use Lingua::Romana::Perligata;
adnota Illud Cribrum Eratothenis
maximum tum val inquementum tum biguttam tum stadium egresso scribe.
vestibulo perlegementum da meo maximo.
maximum tum novumversum egresso scribe.
da II tum maximum conscribementa meis listis.
dum damentum nexto listis decapitamentum fac sic
       lista sic hoc tum nextum recidementum cis vannementa da listis.
       next tum biguttam tum stadium tum nextum tum novumversum
                scribe egresso.
cis.
```

This code is the translation of the following Perl code, which asks the user for the limit (`$maxim`), makes a list of numbers from 2 to the limit, prints out every number on the list and removes its multipliers from the list:

```
print STDOUT 'maximum:';
my $maxim = <STDIN>;
my (@list) = (2..$maxim);
while ($next = shift @list)
```

```
{
    print STDOUT $next, "\n";
    @list = grep {$_ % $next} @list;
}
```

To achieve this result, the main Perl keywords obviously had to be translated into Latin. Numbers are called `nullum`, `unum`, `duo`, `tres`, etc., and can also be written `I`, `II`, `III`, etc. A number used as an index will take the cardinal form `nullimum` (a neologism), `primum`, `secundum`, ..., `XVimum`, etc. The expression `septimum noni tertii primi unimatrixorum`, therefore, translates the Perl expression `$unimatrix[1][3][9][7]`,[15] with the coefficients presented in reverse order. Floating point numbers are expressed by a cardinal numerator and an ordinal denominator; thus, `unum quartum` expresses the value 0.25, and `MMMCXLI Mimum` expresses the value 3.141. Table 5 shows how control structures are translated. The designers of the language chose to distinguish between the particles *dum* and *donec*, which are equivalent in Classical Latin, in order to arbitrarily specialise the former in the sense of "while" and the latter in the sense of "until".

Table 5. Control structures in Perligata. English keywords are translated in Latin. By convention, *en.* while is translated as *la.* dum, and *en.* until is translated as *la.* donec, even though both words have both meanings in Latin.

Perligata	Perl
`si ... fac`	`if ...`
`nisi ... fac`	`unless ...`
`dum ... fac`	`while ...`
`donec ... fac`	`until ...`
`per (quisque) ... in ... fac`	`for(each) ...`
`posterus`	`next`
`ultimus`	`last`
`reconatus`	`redo`
`confectus`	`continue`

The inflectional morphology of Latin makes it possible to distinguish between variable types and how they are used. Variable types are differentiated by number, case and declension (Table 6). A scalar variable is declined as a neuter singular; an array is declined as a neuter plural of the second declension; and a hash table is declined as a masculine singular of the fourth declension. The way the variable is used is denoted by its case: accusative to use its value, dative to follow a reference, and genitive to use it as an index. While the choice of number—singular for a scalar variable, which is a single value, plural for an array and a hash table, which can hold multiple values—is easily understood, as is the choice of neuter gender in the first two cases, the choice of the seemingly arbitrary declension is probably only justified by the desire to avoid any ambiguity: no two roles have the same form.[16]

This system of declensions allows the scope of variables to be declared with the singulars `meo` or `nostro` in a scalar context and the plurals `meis` or `nostris` in a list context:

```
meo varo haec da.              # my $var = @_;
meis varo haec da.             # my ($var) = @_;
meis varis haec da.            # my @var = @_;
nostro varo haec da.           # our $var = @_;
nostris varo haec da.          # our ($var) = @_;
nostris varis haec da.         # our @var = @_;
```

Demonstrative pronouns can be used to refer to variables, arrays, and indices in the default input (Table 7).

Table 6. Variable types in Perligata. A scalar variable is declined as a neuter singular; an array is declined as a neuter plural of the second declension; and a hash table is declined as a masculine singular of the fourth declension. Cases indicate how to use the variable: accusative to use its value, dative to follow a reference, genitive to use it as an index.

Perligata	Number, Case, Decl.	Perl	Role
`nextum`	Acc. sg. 2nd decl.	`$next`	scalar data
`nexta`	Acc. pl. 2nd decl.	`@next`	array data
`nextus`	Acc. pl. 4th decl.	`%next`	hash data
`nexto`	D. sg. 2nd decl.	`\$next`	scalar target
`nextis`	D. pl. 2nd decl.	`\@next`	array target
`nextibus`	D. pl. 4th decl.	`\%next`	hash target
`nexti`	G. sg. 2nd decl.	`[$next]`	indexed scalar
`nextorum`	G. pl. 2nd decl.	`$next[]`	indexed array
`nextuum`	G. pl. 4th decl.	`$next{}`	indexed hash

Table 7. In Perligata, Latin demonstrative pronouns refer to the default input.

Perligata	Number, Case, Decl.	Perl
`hoc`	Acc. sg. 2nd decl.	`$_`
`huic`	D. sg. 2nd decl.	`\$_`
`hujus`	G. sg. 2nd decl.	`[$_]`
`haec`	Acc. pl. 2nd decl.	`@_`
`his`	D. pl. 2nd decl.	`\@_`
`horum`	G. pl. 2nd decl.	`$_[]`

Latin-inspired verb inflections are used to define a function, its role, and its context (Table 8).

Table 8. Perligata function-related structures are inspired by Latin conjugation.

Perligata	Number, Mood, etc.	Perl	Role	Context
`countere`	inf.	`sub count`	definition	-
`counte`	imp. sg.	`count()`	call	void
`countementum`	Acc. sg. resultant	`count()`	call-data	scalar
`countementa`	Acc. pl. resultant	`count()`	call-data	list
`countemento`	D. sg. resultant	`count()`	call-target	scalar
`countementis`	D. pl. resultant	`count()`	call-target	list

This structure allows the form of the language's basic functions and operators to be varied in ways that would make any Latin scholar shudder (Table 9).

Thus, Perligata has a rich inflectional morphology, which is extremely rare, to our knowledge, in programming languages. This inflectional morphology allows, as in Latin, for great syntactic flexibility. To express the Perl instruction

```
$last = $next,
```

which assigns to the variable `$last` the value of the variable `$next`, one can write either:

```
lasto da nextum.
lasto nextum da.
nextum da lasto.
nextum lasto da.
```

https://doi.org/10.3390/philosophies11020055

```
da lasto nextum.
da nextum lasto.
```

Table 9. Built-in functions and operators in Perligata are translated to Latin and subjected to conjugation, at the cost of some barbarisms, e.g., `morimentum`.

Operator/Function	Rendered as	Void Context	Scalar Data	List Data
`+`	"add"	`adde`	`addementum`	`addementa`
`=`	"give"	`da`	`damentum`	`damenta`
`.`	"conjoin"	`sere`	`serementum`	`serementa`
`..`	"enlist"	`conscribe`	`conscribementum`	`conscribementa`
`shift`	"behead"	`decapita`	`decapitamentum`	`decapitamenta`
`push`	"stack"	`cumula`	`cumulamentum`	`cumulamenta`
`pop`	"unstack"	`decumula`	`decumulamentum`	`decumulamenta`
`grep`	"winnow"	`vanne`	`vannementum`	`vannementa`
`print`	"write"	`scribe`	`scribementum`	`scribementa`
`write`	"write under"	`subscribe`	`subscribementum`	`subscribementa`
`die`	"die"	`mori`	`morimentum`	`morimenta`

This syntactic flexibility offers the programmer great stylistic freedom. As Guy Raynaud de Lage observes for ancient French, "the declination, in compensation of its servitudes, frees the language from other servitudes, gives it variety and expressiveness, or at least provides the speaker or writer with the possibility to be varied and expressive. [...] Style can therefore have more influence here than the syntax" [37] (pp. 15,148). For example, the use of anaphora makes it possible to show that different functions apply to the same variable, or vice versa, by noting the invariant at the beginning of the line and the variations at the end of the line [34] SWHID: swh:1:cnt:aab9c1be33f5b304096a51f7982a51a57922b831;lines=16-25:

```
valum da arcessemento varum.
valum da arcessementis varum.
valum da arcessementibus varum.

scribe egresso arcessementa arcessementum arcessementum arcessementum varum.
scribe egresso secundum varum arcessementorum.
scribe egresso secundum varum arcessementuum.
scribe egresso secundum varum arcessementi hashuum.
scribe egresso secundum varum arcessementi arrorum.
scribe egresso secundum varum intra Class arcessementi arrorum intra Class2.
```

This code is the translation of the following Perl code, which demonstrates various combinations of references. The invariant parts (`$var`, `print`, `STDOUT`, `${$var}[2]`) are more scattered in Perl than in Perligata.

```
${$val} = $var;
@{$val} = $var;
%{$val} = $var;

print (STDOUT @{${${${$var}}}});
print (STDOUT ${$var}[2]);
print (STDOUT ${$var}{2});
print (STDOUT $hash{${$var}}{2});
print (STDOUT $arr[${$var}][2]);
print (STDOUT $Class2::arr[${$Class::var}][2]);
```

The Perligata language offers great syntactic flexibility by allowing words to be written in an almost completely free order. Inspired by the characteristics of Latin, namely its rich morphology and syntactic flexibility, it challenges two common features that standard programming languages share with English: the absence of inflectional morphology and the rigidity of word order.

English, having little inflectional morphology, lends itself quite easily to formal grammars, systems that marginalise morphology by reducing it to an epiphenomenon and artificially integrating it into phonological or syntactic rewriting rules [38] (p. 22). For example, Chomsky speaks in 1957 of "morphophonemic rules" [39]. It is quite different from the Latin grammars of Antiquity, in which a precise treatment of syntax only appeared with Priscian, drawing inspiration from Apollonius Dyscolus' study of the Greek language, while the structures of declension, analysed since Aristotle in Greek, are synthesised for Latin by Varro.

At a time when the internationally dominant human language, as well as the most used programming languages, makes syntax understood as word order the essence of grammar, the Perligata language recalls the expressive potential that an inflectional morphology has to offer.

5. Programming in Classical Chinese: Pronouns at the Expense of Linguistic Recursion

The Wenyan programming language [40,41], for its part, dispenses with one of the most common mechanisms in programming languages, namely the expression of block nesting through linguistic recursion. In accordance with ordinary usage in Classical Chinese, it is replaced by a system of pronouns. This feature is all the more noteworthy as Marc Hauser, Noam Chomsky and Tecumseh Fitch consider unlimited linguistic recursion as a universal feature of human languages [42].

The Wenyan programming language, created by Huang Lingdong between December 2019 and July 2020, lets programmers write programs in a fragment of Classical Chinese (文言 *wényán*) that is admittedly stereotypical, but grammatically correct. Classical Chinese is a language derived from ancient Chinese (古文 *gǔwén*), in which many classics have been written, such as the mathematical treatise 九章算術 *Jǐuzhāngsuànshù* "*Nine Chapters on the Mathematical Art*" [43]. This language was used as a scholarly language until the 20th century, including in slogans and solemn formulas in the People's Republic of China.[17]

Two features of Classical Chinese make the understanding of a text heavily depend on context: high density and low use of linguistic recursion [46] (pp. 243–245 and 258–260).

The density of the language is primarily due to the fact that it is mainly composed of monosyllabic words [47] (pp. 18–19); it does not offer redundant words such as *zh.* 城市 *chéngshì* "city (litt. city wall/city)" and *zh.* 知道 *zhīdào* "to know (litt. know/think)", in which full-fledged Classical Chinese words are integrated as morphemes in Modern vernacular Chinese. This language also has great grammatical flexibility: the same character can function as a pronoun, an adjective, an adverb, or an active or a passive verb [45] (§ 63, 70), [46] (pp. 25–30), [47] (ch. II). As Classic Chinese has no inflectional morphology, the appearance of a word taken in isolation provides no clue as to its function; only the context allows it to be determined, starting with the word order—when it can be guessed, since the language is sometimes written without punctuation [45] (Appendix, §1), [46] (p. 244). From a pragmatic point of view, stylistic conciseness is particularly encouraged [45] (§ 64–66), [46] (pp. 243–245), which accentuates polysemy. Context, therefore, plays a huge role: it allows one to understand not only the semantics of the text, but even its syntax.

Another major feature of Classical Chinese is the limited use of linguistic recursion, i.e., the nesting of clauses within other clauses [46] (ch. V). Pronouns and other anaphoric devices are used to simulate recursion.

Although derived from an ancient language, the Wenyan programming language has everything one would expect from a modern programming language. It is free software [40], documented on a website [49], in a manual—in Classical Chinese [50,51]—and through many examples. It is enriched by libraries[18] and extensible by a package manager, `wyg` (文淵閣 *Wényuān Gé* "Belvedere of Literary Profundity") [52]. An online development environment allows one to write code, execute it and compile it in JavaScript, Python and Ruby [53]. The language is Turing-complete: anything that can be computed in any other programming language can also be computed in Wenyan. The syntax is formally defined in Backus–Naur form [41].

Accepting texts written in a fragment of Classical Chinese as programs, the Wenyan programming language differs from conventional programming languages in at least three obvious ways:

- At the *graphematic* level, its alphabet is that of traditional Chinese characters;[19]
- Its *lexicon* is strictly composed of Classical Chinese words. For instance, natural numbers are idiomatically written with Chinese integers 零, 一, 二, 三, 四, 五, 六, 七, 八, 九 and 十 from 0 to 10, 百 "100", 千 "1000", 萬 "10,000", ..., 極 "10^{48}"; floating numbers use traditional words such as 分 "10^{-1}", 釐 "10^{-2}", 毫 "10^{-3}", 絲 "10^{-4}", ..., 漠 "10^{-12}"; Booleans, a term cleverly translated as 爻 *yao* "*Yijing* hexagram line", are called 陽 *yang* and 陰 *yin*;
- At the *syntactic* level, it includes various Classical Chinese sentence patterns: the affirmative form "subject + verb + complement" (e.g., 吾有一數 *wu you yi shu* "I have one number"), the imperative form "verb + complement" (e.g., 名之 *min zhi* "name it"), the predicative form "subject + 者 + predicate + 也", the interrogative form "...乎" (e.g., 夫「甲」「乙」中無陰乎 "among A and B, is none false?") and "...幾何" (e.g., 所餘幾何 "how much remains?").

Since line breaks have no syntactic function in this language, Wenyan code can either appear as a block of text without indentation or punctuation, as below, or be displayed with punctuation, indentation, and syntax highlighting, as in Figure 5 [51] (SWHID: swh:1:cnt:a3a86b8532b2d46976fc90d35fd8f7a8e8b13fcc;lines=90-121, pp. 27–28).

```
夫「魯人耶」若其然者夫「德行科耶」若其然者夫「未仕耶」若其然者夫「蚤死耶」若其然者吾有一言曰「「賢哉回也人不堪其憂回也不改其樂」」書之若 非吾有一言曰「「孝哉閔子騫人不間於其父母昆弟之間」」書之也若非夫「病 厲耶」若其然者吾有一言曰「「亡之命也夫斯人也而有斯疾也」」書之若非吾 有一言曰「「雍也可使南面」」書之也云云若非夫「政事科耶」若其然者夫 「戰死耶」若其然者吾有一言曰「「若由也不得其死然」」書之若非吾有一言 曰「「求無乃爾是過與」」書之也若非吾有一言曰「「朽木不可雕也糞土之牆 不可杇也於予與何誅」」書之也云云若非夫「複姓耶」若其然者吾有一言曰「「賜也始可與言詩已矣告諸往而知來者」」書之若非夫「衛人耶」若其然者吾有一言曰「「起予者商也始可與言詩已矣」」書之若非吾有一言曰「「二三子偃之言是也」」書之也云云云云
```

Despite the obvious exoticism of the alphabet and lexicon, the syntax of the Wenyan language has surprisingly few truly idiomatic features. Indeed, let us compare, morpheme by morpheme, expressions in Wenyan with their translations into English and JavaScript, the programming language used as a means of comparison in the documentation [54]. We will see that the difference is often no less superficial than what we saw in Scratch.

Comments are, as is fairly common, framed by start and end lexemes (Table 10).



Figure 5. Wenyan code published in the old-fashioned Wenyan programming language manual [51] (pp. 27–28). The code is written with syntax highlighting and indentation.

Table 10. Comments in Wenyan, with English and JavaScript translations. Like JavaScript, Wenyan frames the comment between start and end lexemes.

wy.	批曰。「「	利見大人	」」。
en.	comment: ’	…	’.
js.	/*	…	*/

Binary Boolean tests—conjunction and disjunction—are not written in infix notation as in JavaScript, but in prefix, infix and suffix notation, with the operands separated by quotation marks (Tables 11 and 12).

Table 11. Conjunction in Wenyan, with English and JavaScript translations. Wenyan uses a prefix, infix and postfix notation.

wy.	夫「	甲	」「	乙	」中無陰乎。
en.	that ’	a	’ ’	b	’ among-them-none-false?
js.		a	&&	b	

Table 12. Disjunction in Wenyan, with English and JavaScript translations. Wenyan uses a prefix, infix and postfix notation.

wy.	夫「	甲	」「	乙	」中有陽乎。
en.	that ’	a	’ ’	b	’ among-them-any-true?
js.		a	||	b	

https://doi.org/10.3390/philosophies11020055

Negation is denoted by a prefix, namely the verb 變 *biàn* "transform", used in the 易經 *Yìjīng* "*Book of Changes*" to change hexagram lines [43] (p. 901). This is parallel to JavaScript (Table 13).

Table 13. Negation in Wenyan, with English and JavaScript translations. They all use an infix notation.

wy.	變	「甲」。
en.	change	'a'.
js.	!	a

Arithmetical tests are translated in the usual way, in parallel with JavaScript (Table 14).

Table 14. Arithmetical tests in Wenyan, with English and JavaScript translations. This is a word-for-word translation.

wy.	*en.*	*js.*
小於	smaller-than	<
不小於	not-smaller-than	>=
大於	greater-than	>
不大於	not-greater-than	<=
等於	equal-with	==
不等於	not-equal-with	!=

Arithmetical operations are translated using standard mathematical verbs (加 *jiā* "add", 減 *jiǎn* "subtract", 乘 *chéng* "multiply", 除 *chú* "divide"), with the preposition 以 *yǐ* "by means of" separating the two operands (Table 15, 16, 17 and 18) [43] (pp. 909, 911, 936–938).

Table 15. Addition in Wenyan, with English and JavaScript translations. Wenyan uses a both prefix and infix notation.

wy.	加	一	於	二。
en.	add	1	using	2.
js.		1	+	2

Table 16. Subtraction in Wenyan, with English and JavaScript translations. Wenyan uses a both prefix and infix notation.

wy.	減	七百五十六	以	四百三十三。
en.	reduce	756	using	433.
js.		756	-	433

Table 17. Multiplication in Wenyan, with English and JavaScript translations. Wenyan uses a both prefix and infix notation.

wy.	乘	二十三	以	五十七。
en.	multiply	23	by	57.
js.		23	*	57

Table 18. Division in Wenyan, with English and JavaScript translations. Wenyan uses a both prefix and infix notation.

wy.	除	十	以	三。
en.	divide	10	by	3.
js.		10	/	3

https://doi.org/10.3390/philosophies11020055

Euclidean division is marked by a question concerning the remainder (餘 *yú* [43] (p. 1027)) after an ordinary division (Table 19).

Table 19. Euclidean divison in Wenyan, with English and JavaScript translations. Wenyan uses a prefix, infix and postfix notation.

wy.	除	十	以	三。	所餘幾何。
en.	divide	10	by	3.	what-remains-how-much?
js.		10	%	3	

The control structure "if… then.… else" with 若 *ruò* "if" [43, p. 974] is parallel to that of JavaScript (Table 20).

Table 20. "If" control structure in Wenyan, with English and JavaScript translations. The Wenyan syntax is parallel to that of JavaScript.

wy.	若	三不大於五	者。	乃得	「「天」」。
en.	if	$3 \leq 5$	,	then get	"heaven".
js.	if	$(3 \leq 5)$	{	return	"heaven";}
wy.	若非。	乃得	「「土」」	也。	
en.	if-not	then get	"earth"	so-it-is.	
js.	else	{	return "earth";	}	

The structures "for all" and "while" are parallel to JavaScript as well (Table 21).

Table 21. "For" and "while" loops in Wenyan, with English and JavaScript translations. The Wenyan syntax is parallel to that of JavaScript.

wy.	為是	百	遍。	...	云云。
en.	do-this	100	times.	…	and-so-on.
js.	for	(var i = 1; i < 100; i++)	{	…	}
wy.	恆為是。	...	乃止。	...	云云。
en.	constantly-do-this	…	then-stop	…	and-so-on.
js.	while (true) {	…	break;	…	}

Function (術 *shù* "procedure" [43] (pp. 986–987)) definition is no more specific to Wenyan (Table 22).

Table 22. Function definition in Wenyan, with English and JavaScript translations. The Wenyan syntax is parallel to that of JavaScript.

wy.	吾有一術。名之曰	「方田」。	
en.	I-have-a-procédure-name-it-as	'square field'.	
js.	function	square_field	
wy.	是術曰。	...	
en.	this-procedure-says	…	
js.	{	…	
wy.	是謂	「方田」	之術也。
en.	call-this	'square field'	's-procedure-so-it-is.
js.			}

The definition of a function with two arguments is also parallel to JavaScript (Table 23).

Table 23. Function of two arguments in Wenyan, with English and JavaScript translations. The Wenyan syntax is parallel to that of JavaScript.

wy.	吾有一術。名之曰	「方田」。			
en.	I-have-a-procedure-name-it-as	'square field'.			
js.	function	square_field			
wy.	欲行是術。	必先得	二	數。	
en.	wanting-to-apply-this-procedure,	first-need-to-get	2	numbers.	
js.	(			var	
wy.	曰	「廣」。	曰	「從」。	乃行是術曰。
en.	say	'width'.	say	'length'.	now-apply-this-procedure-saying
js.		width	,	length	) {
wy.	...	是謂	「方田」	之術也。	
en.	...	this-is-called	'square field'	's-procedure-so-it-is.	
js.	...			}	

A function call is simply prefixed with a verb, namely 施 *shī* "apply" (Table 24) [43] (p. 983).

Table 24. Function call in Wenyan, with English and JavaScript translations. Wenyan uses a both prefix and infix notation.

wy.	施	「階乘」	於	五。
en.	apply	'step multiplication'	concerning	5.
js.		step_multiplication	(	5);

Thus, despite the apparent exoticism associated with the use of Chinese characters and words, many syntactic features of the Wenyan language are fairly standard in programming, as shown by their literal translation into JavaScript.

Some other features, however, are more idiomatic. One such feature is the extensive use of anonymous variables, which, lacking a name, can be used with the pronouns 之 *zhī* 'this'[20] and 其 *qí* 'that'. For example, the pronoun 其 *qí* "that" [43] (p. 969) takes as its input the output of the previous operation,[21] which allows us to write the operation

$$(1 + 2) \times 3$$

as shows Table 25.

Table 25. Use of the pronoun 其 *qí* "that" in Wenyan, with English translation. The demonstrative pronoun takes as input the output of the previous operation.

wy.	加	一	以	二。	乘	其	以	三。
en.	add	1	using	2.	multiply	that	using	3.

The pronouns 之 *zhī* 'this' and 其 *qí* 'that' allow these values to be used without needing to determine the names of the variables—or sometimes even their value, which defaults to 0 for a number, the empty string for a character string, and true for a Boolean.

吾有一數。加其以二。書之。
吾有一言。曰『商』。書之。
吾有一爻。變其。書之。

```
I-have 1 number. add it using 2. write it.
I-have 1 word. Say ‘‘quotient’’. Write it.
I-have 1 boolean. Change it. Write it.
```

之-formulae are frequent in Chinese mathematical treatises, e.g., 以三百七十五乘之，即欽數 "Multiplying *this* by 375 gives the quantity of *mu*" (a measuring unit) in the *Nine Chapters* [43] (pp. 154–155). Far from being an isolated case, anonymous variables denoted by 之 *zhī* are, in fact, the general case: named variables are themselves only a special case of anonymous variables, to which the naming operation (名 *ming* "name" [43] (pp. 964–965)) is applied after the fact (Table 26)!

Table 26. Use of the pronoun 之 *zhī* "this" in Wenyan, with English translation. The demonstrative pronoun refers to the output of the previous operation.

wy.	加	一	以	二。	名	之	曰	「從」。
en.	add	1	using	2.	name	THIS	say	3.

The structure of operations composed by 之 *zhī* "this" has an obvious limitation: it only allows left-associated formulae, i.e., formulas that do not contain any opening parentheses anywhere other than at the very beginning. This means that formulas of the form

$$(a + b) + c,$$

$$((a + b) \times c) + d,$$

$$((a + b) \times c$$

can be expressed, but not formulas of the form

$$a + (b + c),$$

$$d + ((a + b) \times c),$$

$$(a \times c) + (b \times c).$$

This limitation is common to the standard use of the abacus and basic electronic calculators having no key for parentheses [55].

This computational limitation is overcome by a second idiomatic feature of the Wenyan programming language, namely using the demonstrative adjective from Classical Chinese 夫 *fu* 'that . . . ' (Table 27).

Table 27. Use of the demonstrative adjective 夫 *fu* "that" in Wenyan, with English and JavaScript translations. It allows the construction of arbitrarily complex formulas.

wy.	夫	「甲」。	夫	「乙」。	夫	「丙」。
en.	that	'a'.	that	'b'.	that	'c'.
js.						
wy.	取	二	以施	「丁」。		
en.	take	2	apply	'd'.		
js.				d(b,c)		
wy.	取	二	以施	「戊」。	名之曰	「己」。
en.	take	2	apply	'e'.	call-it	'f'.
js.				e(a,d(b,c))		var f = e(a,d(b,c))

Technically, the values of a parenthesised expression are stored on a stack, which is fed (operation `push`) with the demonstrative adjective 夫 *fu* and accessed (operation `pop`) with the operation 取 *gu* 'take' (Table 28).

Table 28. Push and pop operations made by the 夫 *fu* "that..." and 取 *gu* 'take' operators in Wenyan.

1 夫	2 夫	3 夫	4 取二	5 取二	6 取一
a	b a	c b a	d(b,c) ~~c~~ ~~b~~ a	e(a,d(b,c)) ~~d(b,c)~~ ~~a~~	f = e(a,d(b,c)) ~~e(a,d(b,c))~~

This notation is equivalent to a form of reverse Polish notation ($abcd^2e^2f^1$), a practice attested in Chinese mathematical texts [56]. This original structure, stylistically awkward but idiomatic, provides the construction of arbitrarily complex formulas. This feature of the Wenyan language, rare in programming languages, allows for the best possible adherence to the grammatical constraints and stylistic conventions of Classical Chinese.

Thus, the Wenyan language attests in programming something that, according to Daniel Everett [57] (ch. 15), the Pirahã language reveals about human languages. Contrary to what Marc Hauser, Noam Chomsky and Tecumseh Fitch argue [42], linguistic recursion is not a universal. A system of cross-references, as in pirahã the phrase *Xaisigíai* "they are the same", allows linking the two occurrences of *nails* in the sentences "Hey Paitá, bring back some nails. Dan bought those very nails. They are the same". Recursion may be constructed at the speech level, not at the sentence level. Pirahã and Wenyan show that one can both talk and program without linguistic recursion.

The Wenyan programming language is, thus, inspired by Classical Chinese, borrowing its *graphematics*, its *lexicon*—nouns, prepositions, pronouns and demonstrative adjectives, verbs from mathematical language—its *morphology*—absence of inflectional morphology, presence of compositional morphology—and its *syntax* by adopting various common sentence patterns, but also part of its *pragmatics* by replacing recursion with the use of pronouns. A code in Wenyan is, therefore, not some superficial Rust-like or Scratch-like paraphrase of a code written in some other language. The human language on which it is based truly requires the reformulation of program writing in an idiomatic manner.

6. Conclusions

While some programming languages seem largely independent of any human language, such as APL and Uiua, many of them share the characteristics of a language, usually English: the alphabet, the lexicon, the morphology (absence of inflectional morphology, compositional morphology adopting the Germanic order), and the syntax. The rare linguistic variations are generally limited to the alphabet and lexicon, without requiring a more profound rethinking of the idea of morphology or syntax.

By contrast, we have seen that these variations can be more daring. Perligata uses a Latin-like morphology, while Wenyan uses the syntax of Classical Chinese. Such inventions may at first sight seem like just a schoolboy joke. In some ways, this is obviously the case. But humour sometimes hides deep ideas. The use of a Latin-like morphology almost entirely abolishes the received idea that syntax means, or is based on, word order. The use of Classical Chinese pronouns shows that it is possible to program with little linguistic recursion, even though Chomsky thought that it was essential to any human language. These languages are proofs of concept that refute two hasty assumptions about languages in general, which have a counterpart in the design of programming languages.

Esoteric programming languages in general show that one can program differently than usual, which is a sufficient reason to take them seriously [58]. For example, in Whitespace, one can write programs composed only of whitespace characters (space, tab, and newline), while all other characters are ignored [59] SWHID:

swh:1:cnt:78e253d57a29defc8f391752059d18aa049416cd;lines=2-110. But in most cases, no one would seriously consider programming like this, as it is doubtful that anyone has ever thought so. By contrast, programming languages derived from human languages offer the certainty that they are viable ways, if not to think, then at least to communicate one's thoughts.

Thus, these two examples make it possible to respond positively to the two questions posed by this article: 1. language switching may result in innovative programming styles, 2. the diversity of human languages helps us write programs in a different way.

7. Discussion

Innovation can obviously go further than these two examples and challenge other linguistic beliefs.

Such languages, attested by contemporary experience or by history, can lead to the questioning of usually unquestioned habits or hasty beliefs about languages in general and programming languages in particular.

Some languages can, thus, challenge the very idea that grammar is based on the concatenation operation, understood as the juxtaposition of elements of the "alphabet" in a linear order. This belief is so strong that it is hard-coded in Chomsky's grammars, in which sentences are defined as *sequences* of phonemes or words [39] (pp. 18–19). It is also hard-coded in the Backus–Naur form, whose metalinguistic formulae describe syntax as based on "*sequences* of symbols" and "*juxtaposition* of the sequences" [60] (p. 4, emphasis added). However, some human languages do not assume message linearity. In sign language, five types of information are given simultaneously: the configuration of hands, their orientation, their location, their movement, and the expression of the face. The interrogation mark is expressed by the face and accompanies the entirety of the sentence [61] (pp. 11–12, 110–111). This characteristic goes against the presupposition of Chomsky's grammars and Backus-Naur form, according to which sentences obey the uni-dimensional order of concatenation. Sign language shows that one symbol can be expressed at the same time as several others, and not necessarily before or after any other. Parallel programming languages could mimic such forms of linguistic parallelism.

A very natural question will arise: what other innovations can one expect from human languages? The answer is very simple: by definition, innovation cannot be anticipated. One can just encourage programming language designers to dare to learn foreign languages and not be ashamed of their own native language. The Helsinki Initiative on Multilingualism may thusly benefit programming language design [62].

At a time when programming is threatened with standardisation—through the hegemonic use of Python [63] and the increasing delegation of programming to generative artificial intelligence tools—human languages are a reservoir of living creativity that deserves to be exploited more, be it in general-purpose or in domain-specific programming languages.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article.

Acknowledgments: The author thanks Renaud Alexandre, Andrew Arana, Coraline Jortay and Kai Wehmeier for their precious remarks.

Conflicts of Interest: The authors declare no conflicts of interest.

Notes

1. Many Roman languages have a useful distinction between human languages and languages in general: *es.* idioma/lenguaje, *fr.* langue/langage, *it.* lingua/linguaggio. This distinction is used, for instance, in [5] (Introduction, chap. 3).
2. The very fact that what we call "programming languages" are languages is not self-evident: as Adamczy observes, "Calling something a language does not make it so" [6] (p. 124). This metaphor has a history [7]. Nevertheless, this metaphor is relevant, not only for the reason that programming languages do share some key features of human languages [6,8], but also because they trivially satisfy the definition of what is called "formal languages". As a consequence, if one considers that formal languages *are* languages, which we think reasonable, then programming languages also are.
3. We will limit ourselves here to textual features of programming languages, leaving aside other features such as graphical ones. For instance, in Scratch, text is written in coloured blocks nested inside each other. We will consider here the textual grammar of the written parts, not the graphical grammar of the coloured blocks.
4. By contrast, the identifiers created by the user do not have to adhere to English. For instance, in the French version of Kernighan and Ritchie's C book, English keywords coexist with French identifiers: `while ((l = lireligne(ligne, MAXLIGNE)) > 0)` [13] (p. 28).
5. This thesis, which is true for English, in which Jespersen was a specialist, does not hold true for other languages. Hagège describes a "cycle de l'analytique au synthétique": "La forme du futur, par exemple, analytique au stade prélatin, synthétique en latin : *cantabo*, se disloque en roman : *cantäre avyo* ; ressoudée en français : *chanterai*, décumulée en pidgin, elle se resynthétise en créole baïtien : *m'ap-chante*, je vais chanter" [16] (p. 123).
6. Contemporary English has retained some marks of conjugation, for example, the S of the third-person singular in the present indicative, and the declension of personal and relative pronouns (*I*, *me*, *mine*; *who*, *whom*, *whose*).
7. As of 25 November 2025, 3957 of the 6147 programming languages (64 %) classified by countries on the HOPL website originate in the US [17].
8. There is a detailed analysis of equivalence relations between interpreted languages in [19]. Applying it systematically to programming languages would be beneficial.
9. For instance: іржа in Ukrainian [20], Rost in German [21], Rouille in French [22], Ruggine in Italian [23], Rustico in Spanish [24], Rdza in Polish [25], *Sabi* in Japanese [26], Σκουριασμένο in Greek [27], 锈 *Xiu* in simplified Chinese [28], etc.
10. For instance, the Hedy project teaches programming in the user's mother tongue, with instructions such as *de.* **DRUCKE** HALLO WELT! corresponding to *en.* **PRINT** HELLO WORLD! or *fr.* **AFFICHE** ANIMAUX **AU HASARD** corresponding to *en.* **PRINT** ANIMALS **AT RANDOM**. Such instructions are just word-for-word translations. Moreover, the last step of the course explicitly says: "Congratulations! You have reached the last level of Hedy! The code you have created here can be copied to real Python environments [...]! Note, however, that Python can only read English commands, so if you have been using other languages, you will need to switch to English now." [29].
11. We are talking here about the code, as it is usually visible and manipulable by the programmer. Scratch is one of the cases in which the programmer's ordinary interaction with the code is not textual [30]: the default programming environment is a visual editor, whose code is displayed in the human language chosen by the user. It is possible to view the underlying text code in JSON format by saving and then uncompressing the project's `.sb3` file. Regardless of the language in which the code is displayed, this underlying code is invariably derived from English.
12. Scratch translators could have used *de.* dann.
13. One may say as a joke, comparing two lines of the table, that Japanese uses an "inverse Polish notation".
14. A rich morphology allows for flexibility but does not enforce it. For instance, "German is a language that has both rich casual morphology and strict fixity phenomena in word order" [35] (p. 886): the verb is in second position in a main clause and last in a subordinate clause.
15. In genitive constructions, word order matters, in order to avoid ambiguity.
16. Such ambiguities happened in Latin. For instance, *-a* was both the termination of some feminine nominative singular words of the first declension and that of neuter plural nominative or accusative. The confusion which this ambiguity engendered is the reason why some doublets exist in Roman languages, e.g., *fr.* grain m. sg. < *la.* granum N. n. sg., whose plural form *la.* grana N. n. pl., interpreted as f. sg. N., gave *fr.* graine f. sg.
17. For example, 百花齐放、百家争鸣 *bǎihuāqífàng bǎijiāzhēngmíng* "Let a hundred flowers bloom, let a hundred schools of thought contend", 批林批孔 *pīlín pīkǒng* "Criticise Lin Biao, criticise Confucius"... Viviane Alleton elegantly suggests translating *wenyan* as "noble style", as opposed to 白话 *baihua* "vernacular style" [44], [45–48] (p. 6).
18. The names of the libraries allude to titles of Chinese classics: 算經 *Suànjīng* "Math", 易經 *Zhōuyì* "Random", 列經 *Lièjīng* "Array", 曆法 *Lìfǎ* "Calendar", 位 經 *Wèijīng* "Bit", etc.
19. At least this is the case for the basic lexemes of the language. Nothing prevents the user from defining lexemes that deviate from this alphabet, but this practice is obviously discouraged.
20. This must not be confused with its use as a Saxon genitive when using arrays (列 *lie* 'row') and hash tables (物 *wu* 'thing') [43] (pp. 954 and 1009), whose attributes are designated according to the usual compositional morphology of Chinese, with the preposition 之 *zhī* having the same function as a Saxon genitive, for example, in the expression 物 之「「廣」」 "thing's "width"", which in a language such as Java would be written `this.width`.
21. This process is reminiscent of the variables `$_` (or `$ARG`) and `@_` (or `@ARG`) in Perl, which Larry Wall explicitly compared to the pronouns "this" and "them".

https://doi.org/10.3390/philosophies11020055

References

1. Eastman, C.M. A Comment on English Neologisms and Programming Language Keywords. *Commun. ACM* **1982**, *25*, 938–940. [CrossRef]
2. Veerasamy, A.K.; Shillabeer, A. Teaching English Based Programming Courses to English Learners/Non-Native Speakers of English. Available online: https://www.researchgate.net/profile/Ashok-Kumar-Veerasamy/publication/304024941_Teaching_English_based_programming_courses_to_English_learnersnon-native_speakers_of_English/links/5763e37e08aeb4b997fc87a9/Teaching-English-based-programming-courses-to-English-learners-non-native-speakers-of-English.pdf (accessed on 17 March 2026).
3. Guo, P.J. 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, Montréal, QC, Canada, 21–26 April 2018; ACM: New York, NY, USA, 2018; pp. 1–14. [CrossRef]
4. Becker, B.A. Parlez-Vous Java? Bonjour La Monde != Hello World: Barriers to Programming Language Acquisition for Non-Native English Speakers. In Proceedings of the PPIG, Newcastle University, Newcastle upon Tyne, UK, 28–30 August 2019.
5. Saussure, F. *Cours de Linguistique Générale*; Payot: Paris, France, 1974.
6. Adamczyk, P. On the Language Metaphor. In Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software; ACM: Portland, Oregon, USA, 2011; pp. 121–128. Available online: https://dl.acm.org/doi/10.1145/2089131.2089133 (accessed on 17 March 2026).
7. Nofre, D.; Priestley, M.; Alberts, G. When Technology Became Language: The Origins of the Linguistic Conception of Computer Programming, 1950-1960. *Technol. Cult.* **2014**, *55*, 40–75. Available online: https://dare.uva.nl/search?identifier=aa322682-e9ba-4fe6-8277-3956d45ee463 (accessed on 17 March 2026). [CrossRef] [PubMed]
8. Mélès, B. Approche Philologique des Langages de Programmation. *Rev. Sci. Technol. L'Information Tech. Sci. Informatiques* **2016**, *35/2*, 237–254. Available online: https://hal.archives-ouvertes.fr/hal-01413615 (accessed on 17 March 2026). [CrossRef]
9. Language Tour—Uiua Docs. Available online: https://www.uiua.org/tour (accessed on 17 March 2026).
10. Brinton, L.J.; Brinton, D.M. *The Linguistic Structure of Modern English*; John Benjamins Publishing Company: Amsterdam, The Netherlands, 2010.
11. Landragin, F. *Comment Parler à Un Alien ? Langage et Linguistique Dans la Science-Fiction*; Le Bélial': Moret-Loing-et-Orvanne, France, 2018.
12. Fromkin, V.; Rodman, R.; Hyams, N.M. *An Introduction to Language*, 11th ed.; Wadsworth/Cengage Learning: Boston, MA, USA, 2019.
13. Kernighan, B.W.; Ritchie, D.M. *Le Langage C. Norme ANSI*; Dunod: Paris, France, 2007.
14. Naur, P. Programming Languages, Natural Languages, and Mathematics. *Commun. ACM* **1975**, *18*, 676–683. [CrossRef]
15. Jespersen, O. *Language: Its Nature, Development and Origin*; G. Allen and Unwin: London, UK, 1922.
16. Hagège, C. *La Structure des Langues*; Presses Universitaires de France: Paris, France, 2020.
17. Online Historical Encyclopaedia of Programming Languages. 2025. Available online: https://hopl.info/countriesreportgr.prx (accessed on 17 March 2026).
18. Arawjo, I. To Write Code: The Cultural Fabrication of Programming Notation and Practice. In Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems, Honolulu, HI, USA, 25–30 April 2020; ACM: New York, NY, USA, 2020; pp. 1–15. [CrossRef]
19. Wehmeier, K.F. On Equivalence Relations Between Interpreted Languages, with an Application to Modal and First-Order Language. *Erkenntnis* **2023**, *88/1*, 193–213. Available online: https://link.springer.com/10.1007/s10670-020-00346-w (accessed on 17 March 2026). [CrossRef]
20. Stasiuk, I. Іржа (Irzha). Lic.: WTFPL. SWHID. 2021. Available online: https://archive.softwareheritage.org/swh:1:dir:e3d255c42da6091f79c5e82238579eac9ae2f446;origin=https://github.com/brokeyourbike/irzha;visit=swh:1:snp:e024bbfacb7522ebf50e1557bcd05e195c0f7d0c;anchor=swh:1:rev:fd32b598ceabb0f399f51861a3cc80068124a111 (accessed on 17 March 2026).
21. Lohr, M. Rost. Lic.: WTFPL. SWHID. 2021. Available online: https://archive.softwareheritage.org/swh:1:dir:4f1fcf0eb8acc0838125565c9c8a73827e11999d;origin=https://github.com/michidk/rost;visit=swh:1:snp:3fca968bb0e607cea755280f78b62f61d0c4dbc0;anchor=swh:1:rev:daa1a891e9bc9420967ed84e30190657a791d371 (accessed on 17 March 2026).
22. Bouvier, B. Rouille. Lic.: WTFPL. SWHID. 2021. Available online: https://archive.softwareheritage.org/swh:1:dir:65f2ee226d089122269173631922829d197d14dc;origin=https://github.com/bnjbvr/rouille (accessed on 17 March 2026).
23. DamianX. Ruggine. Lic.: WTFPL. SWHID. 2021. Available online: https://archive.softwareheritage.org/swh:1:dir:532acfbb38f260c2ddb9fbf5f0e1e1ef1164cc22;origin=https://github.com/DamianX/ruggine;visit=swh:1:snp:1ee39c058f7f3555f9d91904b3f66cdcad7cabdb;anchor=swh:1:rev:51026d7b6a0a1cac80c5e58cfd9e04bef719ec76 (accessed on 17 March 2026).
24. Bobadilla, E. Rustico. Lic.: WTFPL. SWHID. 2022. Available online: https://archive.softwareheritage.org/swh:1:dir:73ebba1dece6308b82a4e48adee8be031421bf4f;origin=https://github.com/UltiRequiem/rustico;visit=swh:1:snp:

c1feda93e459abae9cf945bf04b515a30b773132;anchor=swh:1:rev:c14999c9cffb5ef745b3938e502933ae5b716bc8 (accessed on 17 March 2026).
25. Niki. Rdza. Lic.: WTFPL. SWHID. 2021. Available online: https://archive.softwareheritage.org/swh:1:dir:af28bf732bc3fd63d0be208b5fc90cf265458604;origin=https://github.com/phaux/rdza;visit=swh:1:snp:063fdb3c48615a91ae12fd5a1848d678c5f672c5;anchor=swh:1:rev:f48141b908ea8be552b2aa1e0d01ec9bf046c721 (accessed on 17 March 2026).
26. Toyoda, Y. (Sabi). Lic.: WTFPL. SWHID. 2021. Available online: https://archive.softwareheritage.org/swh:1:dir:14e5e8a04e531fcfb27ad0b3d36f205cb09694d6;origin=https://github.com/yuk1ty/sabi;visit=swh:1:snp:0f5fa11fd90a53c7f0d7b20b517fbc59f7cdcce3;anchor=swh:1:rev:2c73dc4a3bd168c5fe2557ba56b5de6af8b11d72 (accessed on 17 March 2026).
27. devlocalhost. Σκουριασμένο (Skouriasmeno). Lic.: WTFPL. SWHID. 2021. Available online: https://archive.softwareheritage.org/swh:1:dir:ed493d1eecfb71da785b5572688b76cd08dc5beb;origin=https://github.com/devlocalhost/skouriasmeno;visit=swh:1:snp:075dc62d0e465f64e9a0438c66e922eb7ec008c9;anchor=swh:1:rev:c8d1c5bb3d4b4c44f47eb7a00d30004b53436d38 (accessed on 17 March 2026).
28. Wu, G. 锈 (Xiu). Lic.: WTFPL. SWHID. 2021. Available online: https://archive.softwareheritage.org/swh:1:dir:95bb3d3775f9ef7b658c24e64102e5fac9b3e5d5;origin=https://github.com/lucifer1004/xiu;visit=swh:1:snp:8b019339486069044b15df389e70e675f0534e67;anchor=swh:1:rev:39ba949f55e2666e125238368318804b410f35c6 (accessed on 17 March 2026).
29. Hedy. 2025. Available online: https://www.hedy.org/ (accessed on 17 March 2026).
30. Gobert, C. *Projecting Computer Languages for a Protean Interaction*; Université Paris-Saclay: Paris, France, 2024.
31. The Scratch Editor Translation Project on Transifex. Available online: https://explore.transifex.com/llk/scratch-editor/ (accessed on 17 March 2026).
32. Mouzo, L. Thésée et le Minotaure. Lic.: Creative Commons Share Alike. 2018. Available online: https://scratch.mit.edu/projects/238150785 (accessed on 17 March 2026).
33. Conway, D. Lingua::Romana::Perligata—Perl for the XXIimum Century. Available online: https://web.eecs.umich.edu/~imarkov/Perligata.html (accessed on 17 March 2026).
34. Conway, D. Perligata. Lic.: Perl Artistic License. SWHID. 2001. Available online: https://archive.softwareheritage.org/swh:1:dir:a98d05f094bc7e225ce1fe7df7cd92467c3fa9fe;origin=https://github.com/gitpan/Lingua-Romana-Perligata;visit=swh:1:snp:189084d86077838080f17d506072374eed50fd8f;anchor=swh:1:rev:686e3f3786ee81aac3ca9350a62d8f627d863688 (accessed on 17 March 2026).
35. Kerleroux, F.; Marandin, J.M. Ordre des mots. In *Vocabulaire Européen des Philosophies: Dictionnaire des Intraduisibles*; Cassin, B., Ed.; Seuil: Paris, France, 2004; pp. 881–892.
36. Baratin, M. *La Naissance de la Syntaxe à Rome*; Les Éditions de Minuit: Paris, France, 1989.
37. Raynaud de Lage, G. *Introduction à l'Ancien Français*; Société d'Édition d'Enseignement Supérieur: Paris, France, 1964.
38. Fradin, B. Syntaxe et morphologie. *Hist. Épistémol. Lang.* **1999**, *21*, 7–43. [CrossRef]
39. Chomsky, N. *Syntactic Structures*; Mouton & Co.: The Hague, The Netherlands, 1957.
40. Huang, L. 文言 (Wenyan). Lic.: MIT. SWHID. 2020. Available online: https://archive.softwareheritage.org/swh:1:dir:5dabbe640a8ebd2ae56de923bcec198b474a2c90;origin=https://github.com/wenyan-lang/wenyan;visit=swh:1:snp:10c5cb6f205a33e9dd5182b9eeb35b40509f8f33;anchor=swh:1:rev:97f0a4b8c5a815467c5c2cac08215d722efde208 (accessed on 17 March 2026).
41. Wenyan Language Specification. Available online: https://wy-lang.org/spec (accessed on 17 March 2026).
42. Hauser, M.D.; Chomsky, N.; Fitch, W.T. The Faculty of Language: What Is It, Who Has It, and How Did It Evolve? *Science* **2002**, *298*, 1569–1579. [CrossRef] [PubMed]
43. Chemla, K.; Guo, S. *Les Neuf Chapitres. Le Classique Mathématique de la Chine Ancienne et ses Commentaires*; Dunod: Paris, France, 2005.
44. Alleton, V. Le nombre de syllabes d'un mot est-il pertinent en chinois contemporain? *Cah. Linguist. Asie Orient.* **1994**, *1*, 5–11. https://www.persee.fr/doc/clao_0153-3320_1994_num_23_1_1764. [CrossRef]
45. Abel-Rémusat, J.P. *Elémens de la Grammaire Chinoise: Ou Principes Généraux du Kou-Wen ou Style Antique, et du Kouan-Hoa, c'est-à-dire, de la Langue Commune Généralement Usitée dans l'Empire Chinois*; Maisonneuve: Paris, France, 1857.
46. Fan, K. *Les Modèles de Phrases dans la Langue Chinoise Classique*; You Feng: Paris, France, 1990.
47. Fan, K. *Le Mot Vide dans la Langue Chinoise Classique*; You Feng: Paris, France, 1991.
48. Lewis, M.E. Chinese Philosophical Texts. Available online: https://web.archive.org/web/20250803173616/https:/chinesetexts.stanford.edu/ (accessed on 17 March 2026).
49. 文言/wenyan-lang. 編程語言 Programming Language for the ancient Chinese. Available online: https://wy-lang.org/ (accessed on 17 March 2026).
50. wenyan-book / 文言陰符. Available online: https://book.wy-lang.org/ (accessed on 17 March 2026).

51. Huang, L. 文言陰符 (Wenyan Yinfu). Lic.: Creative Commons Attribution-NonCommercial 4.0 International. 2020. SWHID. Available online: https://archive.softwareheritage.org/swh:1:dir:788b0dbb09f0e87144d7cde6152d57792851c010;origin=https://github.com/wenyan-lang/book;visit=swh:1:snp:ccda29ec0615a7f69b6e91636a4fa9052bffc19d;anchor=swh:1:rev:d73bb7b6f3aeb3ce13591fa120362e4065234d9f (accessed on 17 March 2026).
52. Fu, A. 文淵閣 (Wényuān Gé). Lic.: MIT. SWHID. 2020. Available online: https://archive.softwareheritage.org/swh:1:dir:4d06c427acfd978c4ca9e983d2567c6af7720bcf;origin=https://github.com/wenyan-lang/wyg;visit=swh:1:snp:83a3c3a23b910d03e4fd5a15a666420f4b3301ac;anchor=swh:1:rev:5af82aa3ebf73be044656c1cb2a01daea54e6564 (accessed on 17 March 2026).
53. 曼德博集—文言 Wenyan Online IDE. Available online: https://ide.wy-lang.org/ (accessed on 17 March 2026).
54. Standard Library Cheatsheet. Available online: https://github.com/wenyan-lang/wenyan/wiki/Standard-Library-Cheatsheet (accessed on 17 March 2026).
55. Mélès, B. Computing Tools and Representations of Arithmetic. In *Cultures of Computation and Quantification in the Ancient World: Numbers, Measurements, and Operations in Documents from Mesopotamia, China and South Asia*; Chemla, K., Keller, A., Proust, C., Eds.; Springer International Publishing: Berlin/Heidelberg, Germany, 2022; pp. 401–431. [CrossRef]
56. Chemla, K. Artificial Languages in the Mathematics of Ancient China. *J. Indian Philos.* **2006**, *34*, 31–56. [CrossRef]
57. Everett, D.L. *Don't Sleep, There Are Snakes: Life and Language in the Amazonian Jungle*; Pantheon Books: New York, NY, USA, 2008.
58. Singer, J.; Draper, S. Let's Take Esoteric Programming Languages Seriously. In Proceedings of the 2025 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Singapore, Singapore, 12–18 October 2025; ACM: New York, NY, USA 2025; pp. 213–226. [CrossRef]
59. Brady, E. WS-idr. Lic.: Unknown. 2016. SWHID. Available online: https://archive.softwareheritage.org/swh:1:dir:83053006ce724ff3a423f3357df77ec19f5658b8;origin=https://github.com/edwinb/WS-idr;visit=swh:1:snp:20f609a1e30f03574f8492d858b172fe9172daff;anchor=swh:1:rev:d900bd406b8ba8e34855a91df8e3a3ecd78d3a2a (accessed on 17 March 2026).
60. Backus, J.W.; Bauer, F.L.; Green, J.; Katz, C.; McCarthy, J.; Perlis, A.J.; Rutishauser, H.; Samelson, K.; Vauquois, B.; Wegstein, J.H.; et al. Revised Report on the Algorithmic Language ALGOL 60. *Commun. ACM* **1963** , *6*, 1–17. [CrossRef]
61. Moody, B.; Vourc'h, A.; Girod, M.; Hof, D.; Benelhocine, R.; Dumartin, S.; Dufour, A.C.; Cardot, V.; Labbé, H. *La Langue des Signes. Tome 1, Histoire et Grammaire*; Éditions IVT: Paris, France, 1998.
62. Federation of Finnish Learned Societies; The Committee for Public Information; The Finnish Association for Scholarly Publishing; Universities Norway; European Network for Research Evaluation in the Social Sciences and the Humanities. *Helsinki Initiative on Multilingualism in Scholarly Communication*; Federation of Finnish Learned Societies: Helsinki, Finland, 2019. [CrossRef]
63. TIOBE Index. 2025. Available online: https://www.tiobe.com/tiobe-index/ (accessed on 17 March 2026).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.