

# A survey of non-XID identifier usage in program text

To: Source code ad hoc working group

From: Robin Leroy

Date: 2022-06-07

---

## Summary

This is an informational document which may be used as material to guide proposals by the source code ad hoc working group for standard répertoire extensions for default identifiers.

We find, based on attested usage, that the space outside `XID_Continue` contains two main types of characters that have potential use in programming language identifiers: characters used in mathematical notation, and emoji.

This document is *not* a proposal; as such, it does not consider the potential issues with including these characters in identifiers (security, potential use as operators, etc.), nor does it discuss the needs that these characters fill. These issues will be discussed in the relevant proposals.

## Background

Unicode Standard Annex #31 defines two identifier syntaxes which generalize ASCII identifier syntaxes that use the répertoire `[A-Za-z0-9_]`.

Requirement UAX31-R1 *Default Identifiers*, used by many programming languages, defines an alphanumeric identifier syntax based on General Categories, extending the classic<sup>1</sup> ASCII Start-Continue pattern `[A-Za-z][A-Za-z0-9_]*`. It makes no attempt at seeking out other Unicode characters which may be useful in programming language identifier syntaxes. While it allows for the use of profiles to extend identifier syntaxes, very few language designers are willing to tailor Unicode defaults.

Conversely, requirement UAX31-R2 *Immutable Identifiers*, designed to address compatibility concerns, allows all but some selected syntax and spacing characters in identifiers. As a result immutable identifiers can contain many characters—such as punctuation—that look like they should be disallowed if one extrapolates the ASCII definition. Relatively few *programming* languages (as opposed to markup languages such as XML) have used this kind of definition.

Nevertheless, some major programming languages have adopted something similar to UAX31-R2; notably, this is the case of C, C++, and Swift. Other programming languages, such as Julia, have defined syntaxes based on additional general categories beyond those underlying `[:XID_Continue:]`. Looking at usage in these programming languages allows us to see which characters beyond the set `[:XID_Continue:]` are useful in programming language identifiers.

---

<sup>1</sup> On the history of that syntax leading up to UAX #31, see Appendix C.

## I. Mathematical notation

Note that the subscript and superscript *letters*, being (modifier) letters, are part of default identifiers already, so identifiers such as  $\Sigma_i b_i k_i$  (with a capital letter sigma, not a summation sign  $\Sigma$ ) that use these without non-XID digits or signs are outside the scope of this survey. Likewise, identifiers using the alphanumeric characters from the *Mathematical Alphanumeric Symbols* block, such as  $\mathbf{p}$ , are already default identifiers.

### I.1 Répertoire

As we were not able to comprehensively search modern Swift and C++ codebases (see Appendix A), we surveyed repositories written in Julia: in that language, the practice of non-ASCII mathematical notation is more established. It turns out that the non-XID characters in use are similar there, even though the set allowed is different.

#### I.1.1 Subscripts and superscripts

The subscript and superscript digits stand out: one finds Swift and C++ identifiers such as  $y^2$ ,  $\sigma^2$ ,  $t^2$ ,  $p_0$ ,  $y_1$ , but also  $\text{length}^2$ ,  $\text{sum}X^2$ , etc. One also finds identifiers that use the subscript and superscript signs, in identifiers such as  $\sigma_k \gamma_k^{-1} p_k^T p_k$  or  $\phi_{k+1}$ .

The same patterns of use are found in Julia: one commonly finds identifiers such as  $D^{-1}$ ,  $V_+$ ,  $x_0$ , etc. The superscript parentheses are also used, primarily to denote derivatives, in identifiers such as  $p^{(n)}$ .

#### I.1.2 Nabla and partial differential

Many mathematical symbols (and punctuation characters with mathematical use) are `Pattern_Syntax`, and thus are not allowed in identifiers by Swift and C++. Two of them however ( $\nabla$  and  $\partial$ ) have bold or italic variants in the *Mathematical Alphanumeric Symbols* block of the SMP, which is allowed in identifiers by these languages. These variants see use in Swift and C++ identifiers, both in the case of nabla (either used with single letters, e.g.,  $\nabla p$ , or with a word,  $\nabla \text{outputs}$ ) or in that of the partial differential sign (in identifiers such as  $\partial E \partial h$  or  $\partial \Omega$ ).

In Julia, many characters in the general category of mathematical symbols are allowed in identifiers, as well as most of those in the general category of other symbols. Despite this significantly larger repertoire, only a handful of mathematical symbols are in common use in identifiers; there too nabla and the partial differential (this time the non-bold ones) stand out.

#### I.1.3 Infinity

Another mathematical symbol commonly seen in Julia identifiers is the infinity sign, either `alone` or in identifiers such as  $V_\infty$  or  $c_\infty$ ; that one has no non-`Pattern_Syntax` equivalent.

#### I.1.4 Prime

Julia also allows prime (`'`) and related characters (multiples `"`, `'''`, and `''''`, reversed `'`, `"`, and `'''`) in identifiers. The prime and double prime characters are in widespread use, in identifiers such as  $B'$ ,  $m'$ ,  $\gamma''$ , etc. We did not find the triple, quadruple, or reversed primes.

Note that `XID_Continue` contains the *modifier letters* prime and double prime; these can be found in mathematical use in many programming languages that do not allow their punctuation counterparts:  $x'$ ,  $f'$  (Swift),  $x''$  (JavaScript), etc.

### I.1.5 Sums and products

Julia allows the  $n$ -ary summation ( $\Sigma$ ) and product ( $\Pi$ ) characters in identifiers. These characters are occasionally used, *e.g.*, we find an identifier  $\Sigma PQ$  for a sum of the  $P \cdot Q$ . However, far more common is the use of the letter sigma ( $\Sigma$ ) for sums:  $\Sigma$ ,  $\Sigma M$ ,  $\Sigma \sigma^2$ , etc. The situation is similar with the letter pi ( $\Pi$ ) for products.

### I.1.6 Integrals

Julia allows the integral sign and variants thereof in identifiers. We were able to find a few occurrences of these signs; however, they were only used on their own, as the names of integration functions, rather than in variables representing an expression: *e.g.*,  $\int$  for integrate,  $\int$  for cum[ulative]sum; this usage is thus more akin to that of an operator.

### I.1.7 Other symbols (excluding emoji)

We were able to find references to one astronomical codebase in Julia which used the astronomical symbols; however their use appears to be anecdotal overall.

### I.1.8 Fractions

Fractions are allowed in identifiers by C++ and Swift (in the *Start* set), and Julia (in the *Continue* set). Their use was exceedingly rare in Julia, being essentially limited to various occurrences of the identifier  $\Sigma^{1/2}$  (for  $\Sigma^{1/2}$ , where  $\Sigma$  is a covariance matrix). In Swift and C++, they were sometimes used to represent the relevant quotients, *e.g.*,  $\frac{2}{3}$  for 2.0 / 3.0.

## I.2 Notability








Mathematical notation obviously does not occur in all codebases; one would not expect the identifier  $y^2$  in a networking stack. It is common in Julia, a language designed for scientific computing. However, as that notation is used in many technical fields, we find it in C++ and Swift in a broad variety of fields beyond the expected numerical analysis and physics, from [machine learning](#) to [cryptography](#), [forestry](#), or [UI toolkits](#).


Its use is not confined to obscure individual projects: we find it in repositories managed by major organizations, such as [Apple \(Swift\)](#), [TensorFlow \(Swift\)](#) or [Natural Resources Canada \(C++\)](#).

## II. Emoji

Many Swift codebases—and some C++ codebases—were found to contain emoji in identifiers.

Two main patterns arise:

1. logographic use, *e.g.*, a function  “print”, a function `disconnect`  “disconnect dæmon”, functions  and  “lock” and “unlock”, etc.
2. use as placeholder names, *e.g.*, `mock errors named` , , , etc.

Emoji usage appeared less frequently in Julia, but it is attested, *e.g.*, we found some  “turtle” graphics.

## II.1 Répertoire

We made no attempt at classifying the kinds of emoji used; for more on emoji see [UTS #51](#).

## II.2 Notability

While we found fewer major institutional users than for mathematical notation, we note that the mock errors mentioned above are in a repository managed by an international company.

## Appendix A. Methods

We document the two methods used to find non-XID identifiers; their limitations mean that, while we are able to document the nature of non-XID identifier usage, we cannot usefully report on its extent, besides the comments made in the *Notability* sections above.

### A.1 BigQuery

Google BigQuery was used to perform queries on the contents of the [github-repos](#) dataset. We looked for occurrences outside of comments and character or string literals of characters outside `XID_Continue` (the exact set is described in appendix B).

It was quickly found that, while large and up-to-date for the repositories it covers, the dataset is missing many repositories. For instance, the dataset contains none of [the Unicode Consortium's open-source repositories](#). Manually searching GitHub for specific C++ identifiers making use of mathematical notation (as was done in the rationale for [L2/22-087 Profile Changes in UAX #31 / UTS #39](#)) found five nontrivial repositories, only one of which was in the dataset (other C++ codebases were found in the dataset, as the BigQuery approach does not require guessing the exact identifiers). The set of repositories seems to be restricted to those that existed when the dataset was created, in 2016; this is a problem for assessing the extent of usage, as more modern codebases (created in the past five years) are excluded.

Nevertheless, the dataset is useful for getting an overview of patterns of use, which is the aim of this document. Excluding the Swift repository itself (which has tests for Unicode identifier support), 111 distinct Swift repositories exceeding 1000 lines of code were found. About a tenth of those use them for mathematical notation; however, codebases using mathematical notation made up half of the top eight heaviest Swift users of non-XID identifiers.

The situation was messier for C++, as we found many encoding issues masquerading as non-XID identifiers. An attempt was made at excluding UTF-8 interpreted as Latin-1, but that left code written in other codepages interpreted as Latin-1. Sifting through the results nevertheless found usage that seemed consistent with the patterns found in Swift.

### A.2 Manual searches

Additional results were found by guessing plausible identifiers and using GitHub's search function. While this technique is not limited to repositories created prior to 2016, it has the downside of requiring a lucky guess of what identifiers might have been used; further, GitHub's search function is not exhaustive; attempts at recovering the results found using BigQuery occasionally failed even for large institutional repositories.

## Appendix B. Swift operators

The Swift programming language allows for user-defined operators in `Pattern_Syntax` (somewhat famously, this means that  $\oplus$  is an operator in Swift, whereas  $\otimes$  is an identifier).

This could be a concern for a principled extension of the identifier space: if  $\oplus$  sees widespread use as an operator, making it an identifier character poses compatibility issues, and retaining the different status of  $\otimes$  and  $\oplus$  is confusing. Perhaps more plausibly, if  $\nabla$  sees widespread use as an operator, making it an identifier poses compatibility issues, and retaining the different status of  $\nabla$  and  $\nabla$  is confusing.

However, the operators actually defined in Swift code overwhelmingly tend to be infix rather than prefix, with various symbols for binary operations and relations:  $\approx$  approximately equals,  $\pm$  plus or minus,  $\cap$  and  $\cup$  intersection and union, etc., which are not natural generalizations of characters found in identifiers.

While we found some usage of prefix operators, such as  $\sqrt{\quad}$  for the square root or  $\sum$  for summation over a container,  $\nabla$  and  $\partial$  were not among those.

## Appendix C. Historical note on Start-Continue identifier syntaxes

As best we can tell, Start-Continue identifiers originate<sup>2</sup> with the “IBM Mathematical Formula Translating System Fortran” (1956), wherein variables consisted of “1 to six alphabetic or numeric characters (not special characters) of which the first is alphabetic”. That syntax was meant to mimic mathematical notation: “the Fortran language closely [resembles] the ordinary language of mathematics”. Indeed, this resemblance went so far as to making the first letter of a variable determine its type, with `[I-N][A-Z0-9]*` being integers.<sup>3</sup>

While the practice of having types determined by the first letter promptly went out of fashion, later programming languages syntaxes generalized the Start-Continue syntax beyond the 47-character space of the IBM 704; in ASCII, this became the familiar `[A-Za-z][A-Za-z0-9_]`, with the low line typically used for word separation, as disregarding spaces had likewise fallen out of fashion.

These syntaxes were further generalized to 8-bit character sets; note for instance Ada 95, which extended the ASCII syntax<sup>4</sup> of Ada 83 to the Latin-1 Supplement, and used character names<sup>5</sup> to determine what a letter was within that character set.

This syntax was finally generalized<sup>6</sup> by Unicode in 1996, expressed using<sup>7</sup> the familiar set of General Categories since 2000, incorporating<sup>8</sup> provisions for backward compatibility since 2003, and ultimately moving to its current location<sup>9</sup> in 2005.

We cannot fail to note that the practice of mimicking “the ordinary language of mathematics” in programming language identifiers is alive and well sixty-five years later, with a character set three thousand times larger.

---

<sup>2</sup> Contrast FLOW-MATIC, the predecessor to COBOL, wherein names consist of “twelve or fewer non-space digits” (where “digit” means character). See *FLOW-MATIC Programming System*, p. 30.

<sup>3</sup> *Fortran Automatic Coding System for the IBM 704: Programmer’s Reference Manual*, pp. 2 & 10.

<sup>4</sup> *Ada 83 Language Reference Manual, Section 2.1 Character Set*.

<sup>5</sup> *Ada 95 Reference Manual, Section 2.1 Character Set*.

<sup>6</sup> *The Unicode Standard, Version 2.0*, p. 5-25.

<sup>7</sup> *The Unicode Standard, Version 3.0*, p. 135.

<sup>8</sup> *The Unicode Standard, Version 4.0*, p. 131.

<sup>9</sup> *Unicode Standard Annex #31: Identifier and Pattern Syntax, Version 4.1.0, Section 2 Default Identifier Syntax*.