Mathematical notation profile for default identifiers

To: UTC
From: Robin Leroy, Source code ad hoc working group
Date: 2022-10-20

I. Summary

The mathematical notation profile for default identifiers consists in the addition of the set *Math_Start* to the set *Start*, and the set *Math_Continue* to the set *Continue*, in definition UAX31-D1. These sets are defined as follows, where the expressions in brackets are in UnicodeSet notation:

Math_Start ≔ [∂𝛛𝜕𝝏𝞉𝟃∇𝛁𝛻𝜵𝝯𝞩∞]

Math_Continue ≔ *Math_Start* ∪ [⁽₍⁾₎⁺₊⁼₌⁻₋⁰₀¹₁²₂³₃⁴₄⁵₅⁶₆⁷₇⁸₈⁹₉]

It is associated with a profile for UAX31-R3, which consists in removing the characters ∂, ∇, and ∞ from the set of characters with syntactic use (these are the characters in [:Pattern_Syntax:] \ *Math_Continue*).

Document L2/22-229 proposes adding the definition of that profile to Unicode Standard Annex #31. This document serves as a rationale for the set of characters added to *Start* and *Continue*.

II. Background

It is a long established practice in mathematical code to make use of identifiers "closely resembling the natural language of mathematics"[1]. Such identifiers include may represent single-letter variables (*e.g.*, `int i`, `COMPLEX*16 ALPHA`), subscripted variables (*e.g.,* `REAL X1, X2`), but also subexpressions, as in the following examples from classic scientific computing libraries:

(1)	`BR = B * R`	LAPACK (Fortran)
(2)	`REAL AINVNM`	LAPACK (Fortran)[2]
(3)	`SINPIY = ABS (SIN(PI*Y))`	FNLIB (Fortran)
(4)	`x2 = x * x;`	CEPHES (C)[3]
(5)	`ivln2 = 1.44269504088896338700e+00`	FDLIBM (C)[4]
(6)	`r2 = x0*x0 + y0*y0;` `r = sqrt(r2);`	IAU SOFA (C)[5]
(7)	`cost2 = np.cos(theta) ** 2`	astropy (Python)

[1] *Fortran Automatic Coding System for the IBM 704: Programmer's Reference Manual* (1956), p. 2.
[2] AINVNM stands for the norm of the inverse of A.
[3] From fresnlf.c; see GitHub mirror.
[4] The constant is the (rounded) inverse of the natural logarithm of 2.
[5] From tpxev.c. See tpxev.c in ERFA. The same code (except uppercase) is used in tpxev.for in SOFA.

Programming language identifiers do not[6] benefit from mathematical typesetting in display. The above usages are therefore examples of a fallback plain text representation of mathematical notation, as described in UTR #25; for instance, the identifiers in (6) above are a fallback for the following:

$$r^{2} = x_{0} * x_{0} + y_{0} * y_{0};$$
$$r = \mathtt{sqrt}(r^{2});$$

This fallback representation is even more restricted than ordinary plain text: as programming languages make use of mathematical syntax (such as the operator +) for executable operations, the expressions do not usually represent multiple terms; or if they do, they need to resort to a heavily degraded representation, such as `log1p(x)` for log(1+x) in IEEE 754.

Nevertheless, this fallback representation can extend beyond Basic Latin; most obviously, Greek letters can be used instead of their names[7]; indeed this usage is common in programming languages and code bases that allow for non-ASCII identifiers, even outside of specialized scientific computing codebases, including in standard libraries[8] and general-purpose frameworks[9].

The fallback representation of mathematical notation in identifiers can also benefit from some characters outside of [:XID_Continue:] (the set of characters allowed in UAX #31 default identifiers), either because they improve readability, or because they resolve ambiguities. This document proposes an extension that serves these aims, while staying clear of characters that could have syntactic use in a programming language, and taking confusability concerns into account.

Indeed, the identity of programming language identifiers should be visually ascertainable, so some lookalike mathematical symbols are not appropriate in identifiers, on account not just of security concerns, but also of usability concerns. For instance, getting a compilation error—or worse, an erroneous result—because one tries to refer to ∆x (increment x) as Δx (delta x) would be mystifying; at the same time, with any reasonable choice of notation, there is little use for these to be allowed as distinct identifiers: a capital delta is an acceptable fallback representation of the increment symbol.

The recommendations in this document are informed by real usage in programs: some programming languages (such as C and C++[10] 11, 14, 17, and 20, as well as Swift) define their identifiers based on requirement UAX31-R2 (immutable identifiers), which allows a much larger set of characters, while others (such as Julia) broadly extend their definition of identifiers to include characters with potential mathematical use. Document L2/22-102 surveys mathematical usage of non-XID identifier characters in these languages.

[6] Some programming environments specifically targeted to mathematical usage display their source code with advanced typesetting, such as displaying subexpressions used as an exponent as superscripts, or displaying fractions vertically; this is for instance the case of Wolfram Mathematica. Such languages are out of scope for the profile described in this document, which specifically targets the family of programming languages designed for plain text display and wherein the multiplication operation uses an explicit operator, such as *, rather than juxtaposition.
[7] Or other fallbacks for them, such as w for ω, or t for θ in (5) above.
[8] See `π : constant := Pi` in the standard library of Ada 2005 and later.
[9] See, *e.g.*, `sinα` in the Swift Foundation framework.
[10] C and C++ 11 through 20 as originally published. Upcoming standards (C23 and C++23) retroactively change the identifier definition to default identifiers, as a defect report to previous versions of the language. See Appendix C.

III. Rationale for inclusions

The following criteria in favour of inclusion in the répertoire were considered:

I1. Attested usage in identifiers, as documented in L2/22-102 *A survey of non-XID identifier usage in program text*.
I2. Lack of fallback representations, or ambiguity of such representations.

The following criteria in favour of exclusion from the répertoire were considered:

X1. A character with attested or potential syntactic usage should not be added to identifiers.
X2. A character that is confusable with ones that are already allowed in default identifiers, and are likely to be used instead, or a character that is confusable with characters that have syntactic use in mathematical code, should not be added to identifiers.

> Note: Criterion X2 isn't "there is a lookalike character which could be used for spoofing"; spoofing issues should be addressed with confusability checks rather than syntactic restrictions (this will be addressed in depth in upcoming documents from the Source Code Working Group). Instead the criterion is "there is a lookalike character which, in a pinch, people pick for the mathematical usage". Exclusions based on this criterion are based on attested usages of these confusables.

III.1. Superscripts and subscripts

The following characters are part of the mathematical répertoire extension for default identifiers, as an addition to the set *Continue*:

$$\left[{}^{(\,)\,+\,=\,-\,0\,1\,2\,3\,4\,5\,6\,7\,8\,9}_{(\,)\,+\,=\,-\,0\,1\,2\,3\,4\,5\,6\,7\,8\,9}\right]$$

These are the subscript and superscript digits, as well as the subscript and superscript characters with the property Math, in Unicode Version 14.0.0.

(I1) Their use is attested in identifiers; see L2/22-102, Section I.1.1.

(I2) They resolve ambiguities; for example:

— `x2` may be a fallback representation of either $\mathtt{x}^{2}$ or $\mathtt{x}_{2}$, including within the same codebase, as in the following examples from a robotics toolkit by MIT:
    — `const T x2 = x.squaredNorm();`
    — `const T x2 = c / (a * x1);`
— In example (7) from Section II, `cost2` (or, if we allow ourselves the Greek letters, `cosθ2`) could be misinterpreted as $\cos(\theta^2)$. In contrast, $\mathtt{cos}^{2}\theta$ would be unambiguous. Note however that an ASCII fallback `cos2θ` would be even worse than `cosθ2`, because it would be interpreted as $\cos(2\theta)$.
— In example (2) from Section II, the inverse is indicated by a suffix `INV`, whereas in example (5), a prefix `iv` fills this role. This can lead to ambiguity when interpreting an identifier such as `AINVB`. In contrast $\mathtt{A}^{-1}\mathtt{B}$ and $\mathtt{AB}^{-1}$ are unambiguous.
— Parentheses distinguish derivatives ($\mathtt{f}^{(3)}$ for the third derivative of `f`) from exponentiation ($\mathtt{f}^{3}$ for `f` cubed).

(X1) These characters are not used as syntax. Indeed, since exponentiation associates more strongly than multiplication, their use as operators would be misleading: `xy²` reads as `(x)(y²)`, but would parse as `(xy)²` if `²` were a postfix operator.

(X2) Most of these characters have no confusables. The exceptions are:

- — SUPERSCRIPT ZERO ⁰ is confusable with the following XID_Continue characters:
  - — MASCULINE ORDINAL INDICATOR º, and
  - — MODIFIER LETTER SMALL O ᵒ;
- — SUPERSCRIPT NINE ⁹ is confusable with the following XID_Continue character:
  - — MODIFIER LETTER US ꝰ.

The confusables with superscript zero are usually visually distinct; they are no more likely to be mistakenly entered instead of superscript zero than o is likely to be used instead of 0. The confusable with superscript nine, while it is visually very similar, is an obscure medievalist[11] character. As such, it is not likely to be inadvertently entered, and since its name does not contain the word "nine", users looking for the superscript nine by name are unlikely to come across it; contrast the primes mentioned in Appendix A. In practice we were not able to find that character in source code in any of the languages considered in L2/22-102.

III.2. Differential operators

The following characters are part of the mathematical répertoire extension for default identifiers, as an addition to the sets *Start* and *Continue*:

[∂𝛛𝜕𝝏𝞉∇𝛁𝛻𝜵𝝯]

These are the partial differential symbol, the nabla symbol, and their mathematical style variants.

(I1) Their use is attested in identifiers; see L2/22-102, Section I.1.2.

(I2) The use of the partial differential symbol may resolve ambiguities with `d` (total differential, or infinitesimal difference—often used for small differences in identifiers). It is also the symbol for the boundary of a set, a sense not conveyed by `d`. While the notation `grad` can be a substitute for `∇` (and `Δ` for `∇²`), it is not common in all mathematical traditions.

(X1) Where languages allow them to be defined as operators, these characters are not attested in operators; see L2/22-102, Appendix B. Note that their use in mathematical notation is as prefix operators, so that, if allowed as function identifiers, they could be used as function names; in a language that uses parentheses for function calls, `∇f` would then be an identifier, and `∇(f)` the computation of the gradient of `f`. This is similar to the situation for the trigonometric functions, which are often written without parentheses in mathematical typesetting.

(X2) Confusability (excluding between the style variants) is only recorded with unrelated Warang Citi and Mende Kikakui characters, which are unlikely to be used inadvertently in mathematics.

Note that all mathematical style variants are included here, since all style variants of the mathematical alphabets and digits are allowed in default identifiers. See *Semantic Distinctions* in Unicode Technical Report #25, pp. 6 sq., on the use of style variants in plain text fallback representations of mathematical expressions.

[11] See proposal L2/06-027.

III.3. Infinity

The character U+221E INFINITY (∞) is part of the mathematical répertoire extension for default identifiers, as an addition to the sets *Start* and *Continue*.

(I1) Its usage is attested in identifiers; see L2/22-102, Section I.1.3.

(I2) Fallbacks generally consist in spelling out "infinity" or an abbreviation thereof (frequently "inf"). This can be ambiguous with the infimum (for which the standard notation is inf), or with other abbreviations (for instance, LAPACK uses `INF` for an `INFO` parameter and `POSINF and NEGINF` for ±∞). Note that infinity is used as part of identifiers other than ∞, such as L∞ or m∞, wherein an alphabetic fallback would lead to further ambiguities (`minf` could easily be misread as the minimum of f, rather than m-infinity).

(X1) The infinity symbol is not an operator. A programming language could use it in its syntax as a literal for infinity in an arbitrary type; however, this does not preclude making it an identifier character: the identifier ∞ would then merely be a reserved word in the language. Alternatively, it could be a library constant; cf. `Ada.Numerics.π`.

(X2) Its confusables are `oo` and the Latin and Cyrillic letters ꝏ and ꚙ. The ASCII sequence `oo`, while attested as a fallback for infinity, looks clearly distinct, especially in fixed-width fonts, and the double letters are unrelated enough that they are unlikely to be picked inadvertently. In practice we were not able to find either double-o ligature in source code in any of the languages considered in L2/22-102.

Appendix A. Other characters considered

The classification in the MathClassEx file associated with UTR #25 (Revision 15) was used together with the survey L2/22-102 to review additional candidates.

The characters in classes Binary, Closing, Fence, Opening, Punctuation, Relation, Space, and Vary, as well as the characters in [:ASCII:] ∩ [:Pattern_Syntax:], were discarded based on criterion X1 (attested or potential syntactic use). Compatibility variants (class X) and glyph parts (class G) were also excluded. This leaves the following characters non-XID_Continue characters, besides the ones discussed above:

¢£¤¥¦§¨¬¯°´¶¼½¾˘˙˚˜϶؆؇؈…′″‴‵‶‷※‼⁑⁗⃝⃞⃟ ℧℩⅀⅁⅂⅃⅄⅋∀∁∃∄∅∆∎∏∐∑√∛∜∟∠∡∢∫∬∭∮∯∰∱
∲∳∼⊤⊾⊿⋀⋁⋂⋃⌀⌂⌐⌑⌙⌶⌬⎴⎵⎶⏜⏝⏞⏟⏠⏡⏢⏣⏤⏥⏦⏧Ⓢ■□▪▫▭▮▯▰▱◆◇◈◉◎●◐◑◒◓◖
◗◢◣◤◥◧◨◩◪◯☉☌☽☾☿♀♁♂♃♄♆♇♈︎♉︎♠♡♢♣♤♥♦♧♩♭♮♯⚀⚁⚂⚃⚄⚅⚆⚇⚈⚉⚪⚫⚬⚲✓✗✠✪✶
⟀⟁⟌⟐⟕⟖⟗⟘⟙⦁⦛⦜⦝⦞⦟⦠⦡⦢⦣⦤⦥⦦⦧⦨⦩⦪⦫⦬⦭⦮⦯⦰⦱⦲⦳⦴⦵⦶⦻⦼⦽⦾⦿⧂⧃⧉⧊⧋⧌⧍
⧜⧝⧞⧠⧧⧨⧩⧪⧬⧭⧮⧯⧰⧱⧲⧳⧸⧹⨀⨁⨂⨃⨄⨅⨆⨇⨈⨉⨊⨋⨌⨍⨎⨏⨐⨑⨒⨓⨔⨕⨖⨗⨘⨙⨚⨛⨜⨝⨞⨟⨠⨡⫡⫱⫼⫿⬒
⬓⬔⬕⬖⬗⬘⬙⬛⬜⬝⬞⬟⬠⬡⬢⬣⬤⬥⬦⬧⬨⬩⬪⬫⬬⬭⬮⬯⭐︎⭑⭒⭓⭔⯂⯃⯄⯅⯆⯇⯈⯊⯋ •▪■⬩⬥ ⬩⬥

Notes: VARIATION SELECTOR-15 was inserted after the characters ♈, ♉, and ⭐ to avoid emoji presentation (♈, ♉, ⭐). VS-15, like all other variation selectors, is allowed in default identifiers. The tofu at the end of this list consists of the following characters:

- ARABIC MATHEMATICAL OPERATOR MEEM WITH HAH WITH TATWEEL
- ARABIC MATHEMATICAL OPERATOR HAH WITH DAL
- BLACK MEDIUM SMALL DIAMOND
- BLACK MEDIUM SMALL LOZENGE

Many of these characters are allowed in Julia identifiers, however most are not used in practice. In this section, we consider those that are attested, as well as some that are common enough in general mathematical notation that their exclusion requires an explanation.

A.1 Unary operators (Class U)

A.1. Increment

As discussed above, the increment symbol fails to meet the criteria for inclusion, and is excluded per criterion X2 (confusability). In Julia which allows both U+2206 INCREMENT (∆) and U+0394 GREEK CAPITAL LETTER DELTA (Δ) in identifiers, the latter is significantly more common in identifiers representing an increment. Indeed increments and Laplacians are both commonly read “delta”; see also the notes for U+2206 in MathClassEx.

A.2. Quantifiers

The universal and existential quantifiers ∀ and ∃ fail to meet the criteria for inclusion, and are excluded per criterion X1 (potential syntactic use). They are not attested in Julia identifiers; conversely it is plausible that they could be given syntactic use in a programming language, as several programming languages have quantification as operations on sequences, *e.g.*, Ada 2012 `for some` and `for all`, Python `any` and `all`.

A.2 Large operators (Class L)

A.2.1 Radicals

Excluded per criterion X1 (syntactic use). The character U+221A SQUARE ROOT (√) is a predefined operator in Julia (likewise ∛). It is sometimes used as an operator in Swift (see L2/22-102, Appendix B).

A.2.2 Summation and product

Excluded per criteria X1 (syntactic use) and X2 (confusability). In Julia which allows both U+2211 N-ARY SUMMATION (∑) and U+03A3 GREEK CAPITAL LETTER SIGMA (Σ) in identifiers, the latter is significantly more common in identifiers representing a sum; see L2/22-102, Section I.1.5. Its usage as an operator is attested in Swift (see L2/22-102, Appendix B). The situation is similar for the product operator ∏ and the letter Π.

A.2.3 Integrals

The integral sign fails to meet the criteria for inclusion, and is excluded per criterion X2 (confusability). The numerous integral signs are not attested in Julia identifiers, except in situations where they effectively serve as integration operators; see L2/22-102, Section I.1.6. The XID_Continue LATIN LETTER ESH (ʃ) is attested in identifiers representing integrals, *e.g.*, ʃT.

A.3 Class N

A.3.3 Fractions

Where they are allowed, the fractions are rarely attested in identifiers. See L2/22-102, section I.1.8. Vulgar fractions (as opposed to vertical ones) are not commonly used in mathematical notation, so they are poorly suited to improving the readability of identifiers representing mathematical expressions.

A.3.2 Primes

Excluded per criterion X2 (confusability). Primes (and double primes, though not triple and higher) are very common in Julia identifiers. However, in programming languages that do not allow them in identifiers, but instead use default identifiers, the use of the modifier letters prime and double prime, which *are* in XID_Continue, is well attested. See L2/22-102, section I.1.4. This is likely because neither the Other Punctuation nor the Modifier Letter are entered via the keyboard; instead users look them up by name, and find the letters as well as the punctuation characters.

A.3.1 Empty set

We found the character U+2205 EMPTY SET (∅) used in one Julia library to represent the empty set; example (i) below. However, it is often confusable with U+00D8 LATIN CAPITAL LETTER O WITH STROKE (Ø), which is attested as a fallback, as in examples (ii) and (iii).

(i) `const ∅ = emptyinterval(Float64)` (Julia)
(ii) `Ø: Set[str] = set()` (Python)
(iii) `let Ø = NSSet()` (Swift)

Further, contrary to the conceptually similar infinity, this symbol is not productive; its only attested usage in identifiers is to represent the empty set, whereas infinity is used in identifiers representing the uniform norm, the value of variables at infinity, etc.

A.3.4 Astronomical symbols

The astronomical symbols fail to satisfy either criterion for inclusion, and are excluded per criterion X2 (confusability). The astronomical symbols are unattested in Julia identifiers, wherein they are allowed. It is unlikely that they would be necessary; indeed, the use of the astronomical symbols for planets, while still common in some publications (especially in the case of 🜨, Earth), is not recommended by the International Astronomical Union, which instead recommends alphabetic abbreviations (Me, V, E, EM, Ma, J, S, U, N, P). See the IAU style manual, Section 5.25. While the IAU does use the symbol ☉, an alphabetic fallback such as "Sun" do not appear to be a problem in practical usage. At the same time, the symbols ☉ (Sun) and 🜨 (Earth) are confusable with the operators ⊙ (circled dot: Hadamard product, symmetric product...) and ⊕ (circled plus: direct sum, XOR...), which have potential syntactic use; indeed in T$_{E}$X these astronomical symbols are generally represented by the operators.

Appendix B. Why a profile?

We are not proposing to add these characters to XID_Continue (and thus to default identifiers with no profile).

One reason for this is technical: the introduction of the characters ${}^{()}{}_{()}{}^{+}{}_{+}{}^{=}{}_{=}{}^{-}{}_{-}$ violates identifier closure under Normalization Form KC; while this is not a problem for most implementations, as normalization is generally performed after lexing, it is a guarantee offered by the default.

Another is compatibility with stable properties: an implementation that conforms to UAX31-R1 and UAX31-R3 and uses this profile removes some characters from the set defined by Pattern_Syntax; the Pattern_Syntax property itself is immutable, so this must be done as part of a profile.

Finally, the use case of identifiers in programming languages used in scientific computing is more restricted than the scope of default identifiers, which is intended to serve as a basis for domain names, usernames, identifiers in markup languages, and other identifier systems where mathematical notation is not used.

Appendix C. Compatibility considerations

Some programming languages (notably, C, C++, and Swift) have adopted identifier definitions based on requirement UAX31-R2, immutable identifiers, but are considering a switch to a definition based on requirement UAX31-R1, default identifiers. This is in part because it is impractical to mitigate spoofing issues in the space of immutable identifiers, which includes unassigned code points. Such a migration would however break backward compatibility, leading to migration issues and divergence between implementations (some implementations do not implement the more restrictive definition). See SG16 issue #79. UAX #31 provides a mechanism to deal with backward compatibility, namely profiles: implementations could add characters commonly used in identifiers as part of a profile in order to avoid breaking real use cases, while still drastically restricting the identifier space.

However, implementers and standardizers are reluctant to define profiles without standard guidance from Unicode. Defining the mathematical notation profile in UAX #31 allows these implementers to refer to a standard, while both retaining backward compatibility in practical use cases and restricting the identifier space to a more manageable set.

Acknowledgements

We thank Murray Sargent for reviewing this document and suggesting improvements.