A Context-Free Parser of Math Expressions

Florian Rabe[0000−0003−3040−3655]

University Erlangen-Nuremberg

Abstract. The interpretation of mathematical expressions is inherently context-sensitive as it depends on (at least) local notations and typing information. Nonetheless, it is desirable to have parsers that can handle as much as possible in a single context-free pass.
We present a set of rules for context-free parsing of expressions in a way that covers many use cases correctly and makes it easy to elaborate context-sensitively in a subsequent type-checking phase. While far from sufficient for processing all expressions found in informal mathematical texts, it is a reasonable trade-off for implementations of formal languages.
We have implemented our parser as a part of the UniFormal language.

1 Introduction and Related Work

Parsing symbolic mathematical expressions is notoriously difficult due to the heavy use of notational conventions and context-sensitive disambiguation. It is even more difficult in text-based languages that are processed by formal systems like proof assistants: These are restricted to one-dimensional notations like $x\hat{}2$, and they usually do not use the font for disambiguation like treating $\mathbf{v}$ and v as different identifiers.

A typical system processes formal expressions as follows: (i) **lexing**: separate identifiers (like x and 2) from operators like $\hat{}$ (ii) **notations**: classify operators as, e.g., infix, prefix, postfix (iii) **precedences**: disambiguate where multiple notations meet as in $x * 2 + y$ (iv) **type-checking**: elaborate every operator application into a function, e.g., treat $x\hat{}2$ as matrix multiplication based on the type of x. Critically, all four steps can *use* context information, but also contribute to *building* the context in the first place. For example, in $\forall L : \texttt{Semilattice}, x.\ x \circ x = x$, the scope and type of the bound variable x becomes clear only during type-checking, and the elaboration of the operator $\circ$ requires a lookup in the theory `Semilattice`, at which point x can be inferred to be an element of L. A common trade-off is to use a simple context-free phase that is so fast that it can, e.g., be called on every keystroke in the editor, followed by a complex context-sensitive phase that might even incorporate undecidable search such as theorem proving.

We present a notation-based parser that we implemented as a part of the UniFormal system [9].[1] It uses a context-free parsing phase that already fixes the *shape* of the AST, and a context-sensitive type-checking phase that elaborates the parsed notations into their meanings. While only a few details of our

[1] Available at https://github.com/UniFormal/UPL/.

implementation are novel[2], it is interesting as an experiment to see how much can be pushed into the context-free phase while still (i) supporting a wide variety of notations and (ii) keeping the type-checker simple.

Our motivation to emphasize the context-free phase is that in large formalizations the context may be spread over hundreds of files, and many operations can (or must) be done without loading them all. This includes, e.g., syntax highlighting, auto-formating, AST display and navigation, and auto-completion using identifiers and notations from the current file. Context-freeness also helps casual readers of formalizations, who are often overwhelmed by the need to peruse large contexts to parse single expressions.

Moreover, while LLMs are increasingly good at partially automating the *writing* of mathematical expressions, e.g., when auto-formalizing a mathematical text [13], it remains important that these expressions be easily human-*readable*. In fact, it is becoming *increasingly* important because (i) humans already cannot evaluate large formalizations for adequacy as fast as LLMs can generate them, and (ii) contrary to common implementations of formal systems, LLM's often work with the current file(s) without knowing the logical context and might not even correctly construct the context when they try.

2 Lexing via Unicode Categories

The Unicode standard [3] defines a range of properties for each character. The ones most relevant here are sketched in the table below.

property	values (simplified)
plane	e.g., basic (16 bit), supplementary (beyond 16 bit)
block	e.g., Basic Latin, mathematical symbols, arrows
category	e.g., letter, symbol, opening bracket, closing bracket
mirrored	mirror image character (if any)
script	e.g., Latin, Greek, mathematical notation

Our intuition is that identifiers are declared and occur in internal syntax, and operators are introduced in notations and are elaborated into identifiers, e.g., $x + y$ contains $+$ as an infix operator and is elaborated into $plus(x, y)$. We see identifiers as essentially alphanumeric and operators as symbolic. But some corner cases must be considered: Some alphanumeric operators are disambiguated by spatial or font dimension that are difficult to use in text formats (such as the symbol T used as an operator for matrix transposition, but then written as x^{T}) or have evolved into separate characters that can be accommodated easily (such as the Σ U+2211 for summation vs. U+03A3 for the Greek letter). Moreover, we need to support identifiers like `commutative_`$+$ and x'.

We make heavy use of the *category* property, and the following table summarizes its values, grouped according to how we will use them:

[2] For example, mainstream proof assistants like Mizar [12], Isabelle [6], or Rocq [10] have used similar designs for decades.

categories	size	remarks
... Letter (4x)	$\approx 131k$	letters and related
... Symbol (4x)	$\approx 8k$	includes mathematical symbols
... Number (3x)	$\approx 2k$	digits and related
Open/Close Punctuation	75/73	brackets, mostly in pairs
Connector Punctuation	10	underscore and related
... Punctuation (4x)	640	all other punctuation symbols
... Separator (3x)	19	whitespace
... Mark (3x)	$\approx 3k$	diacritics, merge with preceding character

Concretely, we define: A **name** is a string of Letter, Number, or Mark characters; if it starts with a Letter with an assigned script, all other Letters must be from the same script (see below for a caveat). A **connector** is a Connector Punctuation character. A **symbol** is a string of Symbol or Punctuation (except Connector Punctuation) characters subject to the following restriction: if and only if the first character is from the Basic Latin block and not one of the 6 brackets in that block (i.e., ()[]{}), more such characters may follow; otherwise, only Modifier Symbols (which include many sub/superscripts and accents) may follow. This restriction is a trade-off to allow for (i) using ASCII art multi-character symbols like (a) ! = or <=> (which are common in text-based formal math), (b) modified symbols like $\exists^!$ (consisting of the Symbol $\exists$ and the Modifier Symbol U+A71D ˡ), but also (ii) multiple different symbols to occur in a row without whitespace separators as in $x \cup \emptyset$ or $3 - (-5)$.

Finally, an **identifier** is a connector-separated sequence of names and symbols, starting with a name that starts with a Letter. An **operator** is an according sequence that starts with a symbol. For example, `commutative_`+ and x' (with Modifier Letter U+02B9 as the prime) are identifiers. Σ (U+2211) is an operator, and Σ (U+03A3) is a letter. The character U+2254 := is a symbol and so is the sequence := of a Punctuation and a Symbol character from the Basic Latin block. Specific to UniFormal is the additional restriction that the symbols ().,:; are reserved as special delimiters.

Our motivation for restricting letters in a name to a fixed script is to allow lexing, e.g., λx or $\sin\alpha$ as two separate identifiers without requiring whitespace. Unfortunately, the *script* property is not determined for the mathematically relevant block Mathematical Alphanumeric Symbols starting at U+1D400, which provides 13 copies of the Latin and 5 of the Greek letters for mathematical font changes as used in $\mathbb{N}$ and $\mathbf{v}$. Therefore, we have implemented an ad hoc variant of the script property that assigns these letters a script such as Latin-bold.

We single out certain operators as **brackets**: a operator is an opening bracket if it starts with any of the 63 characters of category Open Punctuation that have a vertical mirror image of category Close Punctuation.[3] We define the closing bracket operator of an opening bracket by reverting the string and mirroring all

[3] Open Punctuation includes 75 characters, of which only single and double low quotation mark have no mirror images and 10 others such as U+FE35 are 90 degrees-rotated brackets and therefore have a horizontal mirror image. It includes, e.g., U+2329 ⟨, but not Basic Latin <.

brackets. [4] For example, ⌜ and [: (using Modifier Symbol U+A789 :) are opening brackets with closing brackets ⌝ and :].

To avoid confusion, we do not allow operators that contain both Open and Close Punctuation. Neither do we allow ASCII art brackets such as Mizar's (# [12] so that, e.g., (− is always two separate operators. We find that acceptable because 123 Modifier Symbols are available to form multi-character brackets.

3 Forming and Elaborating ASTs via Notation Styles

Context-free parsing using notations is impossible: even if we correctly lex all operators, any one of them could be infix, prefix, etc. For example, only after inferring the types of x and y and collecting all notations from the context, can we tell if $x => y$ is an implication or a greater-or-equal, or if it is $x = (> y)$ with a prefix $>$. Similarly, the $+$ in $x + y = z$ could be a Boolean operator, in which case only type-checking can tell if it means $x + (y = z)$ or $(x + y) = z$.

Our goal here is not to faithfully parse *all* reasonable mathematical expressions. Instead, we tried to *minimize* the number of whitespace or brackets that must be inserted during formalization. For example, requiring $x = \_ > y$ or $x = (> y)$ is acceptable in the rare case where $>$ is a unary prefix.

We decided not to support general mixfix notations that use multiple operators such as $[x]$ or $G \vdash x \rightsquigarrow y$. From supporting them in MMT [8], we learned that they increase parsing complexity a lot because the visibility of notations affects the shape of the AST. For the same reason, we rejected more sophisticated approaches that parse ambiguous expressions into all possible parse trees and then choose the one that type-checks. While that would improve coverage, we are here interested in determining the entire shape of the AST already in the context-free phase.

Therefore, we fix a set of **notation styles**. Writing $*$ as a placeholder for a non-bracket operator, and $\triangleright$ and $\triangleleft$ for a bracket pair, these styles are: (i) **prefix** $*e$ (ii) **postfix** $e*$ (iii) **infix** $e_1 * e_n$ (iv) **circumfix** $\triangleright e \triangleleft$ (v) **applyfix** $e \triangleright e \triangleleft$ (vi) **bindfix** $*\Gamma.e$ where Γ declares some variable bindings. (Below we introduce some variants for flexible arity such as for associative infix operators.)

Restricting the notations to such a fixed set and hard-coding those in the parser greatly reduces the parsing complexity while still covering many common mixfix notations. And it is always possible to add new styles if necessary.

Because brackets are already lexically distinguished, the only remaining parsing conflicts, which we discuss below, are: (i) **pre-conflict**: Is $*$ at the beginning of an expression prefix or binding? (ii) **post-conflict**: Is $*$ after an expression postfix or infix? (iii) **in-conflicts**: How to bracket $e * f \circ g$ and $e * f \circ$ and $*e \circ f$?

Elaboration during Type-Checking It is routine to parse unresolved identifiers into placeholders in the AST, and to resolve them into qualified identifiers during type-checking. We piggy-back on this mechanism by also parsing all operators

[4] The mirrored bracket character is the immediately following Unicode character except for ([{⌜⌞, where it is a few codepoints away.

into placeholders and resolve them into identifiers during type-checking. The placeholder consists of the operator string and the fixity: for example, $x + y$ is parsed as the function application $\texttt{infix\_+}(x, y)$, and the type-checker must elaborate $\texttt{infix\_+}$ into an identifier with a matching notation.

In our implementation, to each identifier declaration, a notation may be attached, as in $\texttt{plus} : (\mathbb{N}, \mathbb{N}) \to \mathbb{N}$ # $\texttt{infix}+$. To elaborate $\texttt{infix\_+}(x, y)$, the type-checker (i) infers the types of x and y, and (ii) collects all identifiers in the context with an $\texttt{infix}+$ notation. If that information does not allow uniquely disambiguating, an error is reported. The type of the thus-selected identifier must match the number and types of the arguments as described in the following table:

notation	identifier f with type	concrete syntax	elaboration
$\texttt{prefix}\circ$	$a \to b$	$\circ e$	$f(e)$
$\texttt{postfix}\circ$	$a \to b$	$e\circ$	$f(e)$
$\texttt{infix}\circ$	$(a, b) \to c$	$d \circ e$	$f(d, e)$
$\texttt{infixleft}\circ$	$(a, b) \to a$	$e_1 \circ \ldots \circ e_n$	$f(\ldots f(e_1, e_2), \ldots, e_n)$
$\texttt{infixright}\circ$	$(a, b) \to b$	$e_1 \circ \ldots \circ e_n$	$f(e_1, \ldots, f(e_{n-1}, e_n)\ldots)$
$\texttt{infixassoc}\circ$	$\texttt{list}\, a \to a$	$e_1 \circ \ldots \circ e_n$	$f([e_1, \ldots, e_n])$
$\texttt{circumfix}\triangleright$	$(a_1, \ldots, a_n) \to b$	$\triangleright e_1, \ldots, e_n \triangleleft$	$f(e_1, \ldots, e_n)$
$\texttt{circumfixflex}\triangleright$	$\texttt{list}\, a \to b$	$\triangleright e_1, \ldots, e_n \triangleleft$	$f([e_1, \ldots, e_n])$
$\texttt{applyfix}\triangleright$	$(c, a_1, \ldots, a_n) \to b$	$e \triangleright e_1, \ldots, e_n \triangleleft$	$f(e, e_1, \ldots, e_n)$
$\texttt{applyfixflex}\triangleright$	$(c, \texttt{list}\, a) \to b$	$e \triangleright e_1, \ldots, e_n \triangleleft$	$f(e, [e_1, \ldots, e_n])$
$\texttt{bindfix}\circ$	$(a \to b) \to c$	$\circ x : A.e$	$f(\lambda x : A.e)$
$\texttt{bindfixassoc}\circ$	$(a \to b) \to c$	$\circ x_1 : A_1, \ldots, x_n : A_n.e$	$f(\lambda x_1 : A_1.f(\lambda x_2 : A_2.\ \ldots e)\ldots)$

Here we assume that the language supports lists $[e_1, \ldots, e_n]$ to handle flexible arities and λ-abstraction for higher-order abstract syntax. Note that we support associative binder notations that allow elaborate, e.g., $\forall x, y.P$ as $\forall x.\forall y.P$ Because not all binders are associative (e.g., unique-existential is not), we use two separate notations for binders.

In a few cases, it is necessary to use unapplied operators as in "Consider a semigroup $(G, \circ)$" or $x.\texttt{fold}(+)$ for a list x. Our parser treats an operator as unapplied if it occurs at the beginning of an expression and is followed directly by a comma or a closing bracket. We elaborate this into any identifier with a notation for that operator that matches the expected type.

Pre-Conflict This conflict must already be resolved just to decide whether the subsequent characters should be parsed as a context. Unicode categories do not help here, and neither does common practice in formal math languages: because the most common binders are not part of ASCII, practitioners have developed many different workarounds for writing binders in text files. Examples include % for λ in Isabelle [6], $![\Gamma] : e$ for $\forall$ in TPTP [11], $(\Gamma) \Rightarrow e$ for λ in Scala [5], $[\Gamma]e$ for λ in Twelf [7], or $\text{fun}\,\Gamma \Rightarrow e$ for λ in Rocq [10]). Unicode descriptions

only help a little bit: the word "N-ARY" is used for 17 characters such as $\bigwedge$, integral for 25 such as $\int$, and quantifier for $\forall$, $\exists$, and $\nexists$[5].

But in general there cannot be a single satisfactory rule that recognizes binder operators purely lexically. We settled for a quick lookahead that checks if the text following an operator can be interpreted as a binding. That is straightforward because it must be an identifier followed by : (type variable), . (untyped variable), or , (sequence of variables), all of which are reserved in UniFormal.

Post-Conflict A post-conflict operator must be postfix if a closing bracket or a comma follows, and infix if an opening bracket or binder follows. These special rules already cover many important cases such as the $\wedge$ in $A \wedge \forall x.P$.

Many operators are almost exclusively used as infix, such as $\wedge$. Therefore, one could create a dictionary that classifies every operator as either postfix or infix. But such a dictionary is expensive to represent and use (especially for casual readers, who have not memorized it), and there are still many characters where an operator is used for multiple notations such as $*$.

After much back and forth, we eventually abandoned dictionary-based approaches and settled on the following rule: If the number of spaces around the operator are the same on the left and the right, it is infix. If it is smaller on the left than on the right, it is postfix. If is greater on the left, it is neither and the expression ends before the operator.

For example, this parses $x!y$ or $x␣!␣y$ as infix, $x!␣y$ as postfix $x!$ followed by unparsed y, and $x␣!y$ as x followed by unparsed $!y$. Because we only count space and not newline characters, this also provides an awkward but easy way to parse multi-line expressions: $x!$ or $x␣!␣$ at the end of the line expect another argument on the next line, but $x!␣$ does not.

Making whitespace relevant for parsing is increasingly popular as modern IDEs are auto-formating consistently anyway. Our rule was motivated by the spacing rules in employed by MathML's operator dictionary [1, Appendix B].[6] It governs the presentation of MathML expressions by defining the left and right padding for each operator. And this is indeed symmetric for infix and asymmetric for other operators.

In-Conflicts Similar to post-conflicts, these may require a fixed dictionary that subclassifies the infix operators by precedence level. For example, MathML uses 3 levels, roughly for multiplication-like, addition-like, and other operators, which are then presented with different amounts of symmetric padding.

While 3 levels are reasonable for presentation, it is too little for parsing: Already the simple expression $a \wedge b = c + d * e$, which humans can parse easily, requires 4 levels. It even makes sense to add more levels for very tight-binding infixes like $\hat{}$ and very loose-bindings ones like long arrows.

Ultimately, we abandoned dictionary-based approaches here as well. Instead, we define the precedence level of an operator as its length plus the number

[5] Interestingly, Unicode does not have a negated universal quantifier.

[6] This sorts Unicode symbols into 13 categories: 3 for infixes, 2 for pre/postfix, 2 for open/close brackets, 2 for binders, 1 for accents, and 3 for various special cases.

of surrounding spaces.[7] Then longer operators appear higher in the AST. Thus, $x+y == z$ and $x \Longrightarrow y \wedge z$ can be written without spaces. Our approach has two huge advantages: Any operator can be used with any precedence. And humans can see the correct parse immediately, without having to remember an operator dictionary. In fact, the author found that he anyway often adds spaces around looser-binding operators to enhance readability even when the parser does not require it. This can become cumbersome as in $a\,\,\,\wedge\,\,\,b\,\,\,=\,\,\,c\,+\,d*e$, but one pair of brackets suffices to write it as $a\,\wedge\,b = (c\,+\,d*e)$.

4 Conclusion

We presented an experiment with a simple parser design for formal expression that focuses on resolving the entire AST shape context-freely. A key idea is to leverage the Unicode standard as much as possible. While Unicode characters used to be cumbersome to type, modern editors are good at making it seamless (e.g., by auto-completing LaTeX macros into Unicode characters), and for generated content, Unicode is no obstacle anyway. This worked well for lexing identifiers and operators and for parsing notations that use brackets (at the cost of excluding a few common notations like $|x|$, which use non-bracket characters as brackets). It worked less well for lexing binders and parsing other notations.

A novel and possibly controversial idea is to use horizontal space around operators for the context-free resolution of fixity and precedence. Such a convention was supported by an informal evaluation, in which the author sampled formulas from Isabelle's library [4] and Lean's Mathlib library [2]. In Isabelle, there is already sporadic use of a whitespace convention: in multiple places low-binding infixes are type-set as, e.g., $\,\,\wedge\,\,$ to separate them from tighter-binding infix operators in the arguments. In Mathlib, on the other hand, the author was stumped multiple times by infix chains of the form $x\,*\,y\,\circ\,z$ where the correct parse was quite difficult to recognize.

Current work focuses on evaluating the rules in larger case studies and increasing the coverage. The current implementation already experiments with additional heuristics for situations where multiple operators occur in a row without any whitespace. For example, a prefix-over-postfix heuristic seems to work well, e.g., to parse $F \wedge \neg \Box \forall x.P$. A particular difficulty here are nullfix symbols like $\bot$ or $\emptyset$: for example, it is difficult to choose between nullfix-infix as in $\varnothing \cup x$ and prefix-prefix as in $\neg\neg x$.

References

1. R. Ausbrooks, S. Buswell, D. Carlisle, S. Dalmas, S. Devitt, A. Diaz, M. Froumentin, R. Hunter, P. Ion, M. Kohlhase, R. Miner, N. Poppelier, B. Smith, N. Soif-

[7] We count characters like $\Longrightarrow$ as having length 2. Indeed, most monospace fonts render these as taking up 2 columns. Unfortunately, this set is not standardized, and the behavior is editor and font-specific. But we include in this set at least the 23 long arrows and tacks starting at U+27F5.

fer, R. Sutor, and S. Watt. Mathematical Markup Language (MathML) Version 2.0 (second edition), 2003. See http://www.w3.org/TR/MathML2.
2. Lean Community. mathlib4, 2026. https://github.com/leanprover-community/mathlib4.
3. The Unicode Consortium. The unicode standard, version 17.0.0, 2025. https://www.unicode.org/versions/Unicode17.0.0/.
4. Isabelle library, 2008. http://isabelle.in.tum.de/dist/library/index.html.
5. M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. artima, 2007.
6. L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.
7. F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Automated Deduction*, pages 202–206, 1999.
8. F. Rabe. The MMT API: A Generic MKM System. In J. Carette, D. Aspinall, C. Lange, P. Sojka, and W. Windsteiger, editors, *Intelligent Computer Mathematics*, pages 339–343. Springer, 2013.
9. F. Rabe. Global, Regional, and Local Contexts. In P. Koepke and V. de Paiva, editors, *Intelligent Computer Mathematics*. Springer, 2025.
10. Rocq Consortium. The Rocq Prover: Reference Manual. Technical report, INRIA, 2025.
11. G. Sutcliffe. The Logic Languages of the TPTP World. *Logic Journal of the IGPL*, 2022.
12. A. Trybulec and H. Blair. Computer Assisted Reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28. Morgan Kaufmann, 1985.
13. J. Urban. 130k lines of formal topology in two weeks: Simple and cheap autoformalization for everyone?, 2026.