# An Ambiguous Grammar for Mathematics

Richard Warfield

**Abstract**

Mathematical expressions as they appear in technical documents cannot be confined to a single simple grammatical class. We have developed a framework for more realistic understanding of such expressions than is possible from a traditional programming language parsing perspective. One application for this is for reading expressions from a technical document in preparation for storing them in a content-retrievable form or manipulating them further. In this case a relatively simple character-recognition and sorting technique can often convert 2-dimensional expressions into 1-dimensional strings. Applying the grammar and parser presented here can find one (or more, in cases of ambiguity) parses for such a string. Another application is the reading of ‘‘naive” input to a web page requesting mathematical input.

## Introduction

The grammar described in the following sections is intended for parsing a one-dimensional representation of mathematical equations as they would appear in publication (as opposed to the way they appear in computer programs). A typical grammar for parsing simple expressions as they appear in, for example, a C program is shown below[1].

$$
\begin{aligned}
&expr \rightarrow expr + term \mid expr - term \mid term \\
&term \rightarrow term * factor \mid term \,/\, factor \mid factor \\
&factor \rightarrow (\, expr\,) \mid -factor \mid \textbf{ident} \mid \textbf{ident}(\, expr\,)
\end{aligned}
$$

The language specified by this grammar has several properties not present in the language of "real" math that make it easy to parse with an LALR parser:

- It is context free. For example, because multiplication always requires * to be explicitly present, there is no ambiguity as to whether an expression such as A(X+Y) is a function application or a multiplication ( A*X + A*Y ).
- Any binary operation can be applied to either side of any term. This would not be true if function applications without parentheses and products without an asterisk were allowed in the language. For example, "sin x" cannot be multiplied on the right by the identifier "y", since "sin xy" clearly means sin(x*y) and not (sin(x)) * y.
- The location of all operands to these unary and binary operators is clear from the syntax. In contrast, the location of the variable of integration in an indefinite integral, for example, can only be found by digging through the operand looking for a differential.

Given that mathematical notation was (unfortunately) not designed with context-free parsers in mind, parsing this language presents some difficulties that require many complex grammar rules (relative to the short grammar presented above) to overcome, some difficulties that are impossible to overcome without semantic analysis (such as the function application versus multiplication ambiguity), and some which can be overcome easily by making certain assumptions about the language.

The following is a partial list of the structures and operations that are recognized by this parser and limitations and assumptions imposed on them:

| ***Expressions*** | ***Limitations*** |
|---|---|
| Variables and Function names | Only one-character identifiers are supported, except for a list of functions recognized by the parser (sin, cos, log, etc...) |
| Binary operators +, -, *, / | "/" operator requires three sets of parentheses in some expressions to produce a correct parse: one around the numerator, one around the denominator, and one around the entire fraction[1]. |
| Products with no operator<br>(such as ab = a*b) | |
| Exponentiation | |
| Function applications with explicit parentheses (such as f(x) or tan(x+y) ) | Whenever an expression such as f(x) or a(b+c) is encountered, an ambiguous parse is produced. Without semantic information it is impossible to distinguish between a function application of this form and the product of a variable and expression in parentheses |
| Function applications with no parentheses (such as sin xy) | Operand must consist of either a product of non-parenthesised variables and numbers, or some other sort of single factor. |

1 The justification for this seemingly severe limitation is that this information is present in fractions as they appear on a two-dimensional page, since the fraction bar delimits the numerator, the denominator, and the fraction itself from an enclosing expression. In general, I have assumed that no additional ambiguity is introduced in the process of converting the two-dimensional expression into a one-dimensional expression.

| ***Expressions*** | ***Limitations*** |
|---|---|
| Definite and indefinite integrals | The differential must appear in one of two places: either on the far right of the integrand (such as in $\int ax+bx^2 dx$ ) or on the far right of the numerator of the rightmost factor (such as in $\int \sin x \frac{xdx}{(x+1)}$ ) |
| Summations | Associativity of summation with the * operator is ambiguous: $\sum_{i=0}^{n} x*y$ will produce two parses, one with y inside the sum and one with y outside |
| Relational operators $=, \le, \ge, <, >$ | |

## Overview of the Parser and Grammar

This grammar was developed using the context-free parser from chapter 19 of *Paradigms of Artificial Intelligence Programming* by Peter Norvig[2] together with a scanner written by Richard Fateman. The parser is written in Lisp and accepts a list of grammar rules as input. It is capable of producing multiple parses in the case of ambiguous grammars. There is no distinguished start symbol in the grammars used by this parser; any reduction that uses up all of the input results in a valid parse (duplicate parses are removed, however).

The following sections give a rough description of the grammar. Details specific to the parser and scanner being used are omitted. The grammar is structured loosely the way the computer language expression grammar shown above is. At the top level is the nonterminal *S*, which encapsulates the relational operators:

$S \to S = expr$
$S \to S < expr$
$S \to S > expr$
$S \to S \le expr$
$S \to S \ge expr$
$S \to expr$

The *expr* nonterminal appears in its usual form:

$expr \to expr + term$
$expr \to expr - term$
$expr \to term$

When we come to the *term* nonterminal, the grammar departs rather dramatically from the usual expression grammar. As discussed above, certain terms cannot be multiplied on the right by some kinds of factors unless an asterisk or parentheses are used. In other words, if $\alpha$ is the term in question then $\alpha\beta$ is not allowed for some factors $\beta$. *term* is divided into three nonterminals based on what $\beta$ are allowed. One nonterminal, *lara_term* can be multiplied on the left by anything and on the right by anything. This nonterminal includes simple numbers and variables, function applications with parentheses, all parentheseized subexpressions, and products of these entities. The second term type, *larf_term*, can be multiplied on the left by anything on the right by (non-variable, non-number) factors. It includes function applications without parentheses, such as *sin* x. The third type, *larn_term*, can be multipled on the left by anything and on the right by nothing. It includes summations and integrations.

The basic *term* nonterminal appears only to attach the other term types into *expr*:

$$term \rightarrow term * any\_term$$
$$term \rightarrow term / any\_term$$
$$term \rightarrow any\_term$$

$$any\_term \rightarrow lara\_term$$
$$any\_term \rightarrow larf\_term$$
$$any\_term \rightarrow larn\_term$$
$$any\_term \rightarrow -any\_term$$

### Function Applications without Parentheses : *larf_term*

This section describes the nonterminal *larf_term* and related nonterminals. *larf_term* represents structures which may appear in a product only if the factor immediately to the right is not a number or variable. Function applications such as $\sin x$ and $\coth 2nx$ fall into this category because an expression like the latter should be interpreted as $\coth(2nx)$ and not as $\coth(2n)*x$ or $\coth(2)*nx$ . However, an expression such as $\sin x(a+b)$ is likely to mean $(\sin x)(a+b)$ . Other structures may fall into this category as well and the grammar could easily be extended to include them.

$$larf\_term \rightarrow larf\_term\ larf\_factor$$
$$larf\_term \rightarrow lara\_term\ larf\_factor$$
$$larf\_term \rightarrow larf\_factor$$

$$larf\_factor \rightarrow funapp\_noparens$$

$$funapp\_noparens \rightarrow fun\ \ funarg\_noparens$$
$$funapp\_noparens \rightarrow fun\ \wedge\ left\_delimited\_item\ \ funarg\_noparens$$

The first grammar rule allows products of *larf_factor*s (for example, sin *x* cos *x*). The second rule says that when a *lara_term* is multiplied on the right by a *larf_term*, the result is a *larf_term* because it still cannot be multiplied on the right by variables or numbers. The last grammar rule allows for expressions like $\sin^2 x$ . *funarg_noparens* rewrites as:

*funarg_noparens* → *var_prod*
*funarg_noparens* → *lara_factor*
*funarg_noparens* → *left_delimited_item*
*funarg_noparens* → − *funarg_noparens*

*var_prod* → *var_prod var_power*
*var_prod* → *var_power*

*var_power* → **var** ^ *left_delimited_item*
*var_power* → **number** ^ *left_delimited_item*
*var_power* → **var**
*var_power* → **number**

*var_prod* represents a product of numbers and variables possibly raised to powers (for example, $7nx^2 y$ ). *left_delimited_item* represents essentially anything that can act as a factor:

*left_delimited_item* → *delimited_nonvar_factor*
*left_delimited_item* → *larf_factor*
*left_delimited_item* → **var**
*left_delimited_item* → **number**
*left_delimited_item* → − *left_delimited_item*

*delimited_nonvar_factor* includes all non-variable, non-number factors that are both left-delimited and right-delimited. By delimited I mean that the boundary of the factor is marked by some terminal or otherwise doesn't depend on the surrounding expression. For example, anything surrounded by parentheses is both left and right-delimited, as are function applications with parentheses. Function applications without parentheses are not right-delimited because adding additional characters to the right may (or may not) alter the original factor (e.g., adding "y" to the right of "sin *x*" changes the expression to "sin *xy*", but adding "cos *y*" does not change the argument to sin).

*delimited_nonvar_factor* → ( *expr* )
*delimited_nonvar_factor* → *funapp_parens*

*funapp_parens* → *fun* ( *exprlist* )

**Integrations and Summations** : *larn_term*

*larn_term* represents mathematical structures which cannot be multiplied on the right without some form of explicit separation of factors (parentheses or an asterisk). Integrations and summations fall into this category: $\sum xy$ of course means $\sum (xy)$ rather than $(\sum x)\, y$ , and an expression such as $\int x\, dx \frac{x}{1-x}$ is presumed to mean $\int \frac{x^2}{1-x} dx$ rather than $(\int x\, dx)(\frac{x}{1-x})$ .

The structure of *larn_term* is similar to that of *larf_term*:

*larn_term* → *larf_term* *larn_factor*
*larn_term* → *lara_term* *larn_factor*
*larn_term* → *larn_factor*

*larn_factor* → *summation*
*larn_factor* → *integration*

The first two rules say that when either a *larf_term* or *lara_term* is multiplied on the right by a *larn_term*, the result is a *larn_term* since the rightmost factor is an integration or summation. The rule for summation is:

*summation* → *sigma* *any_term*

where *simga* is a terminal or sequence of nonterminals representing $\sum$ and possibly the bounds of the summation. Note that this rule does not allow * to appear in the summand unless there are parentheses around the expression. An alternative rule would use *term* instead of *any_term*. This rule would result in an ambiguous parse when * is encountered in the top-level of the summand.

Rules for correctly parsing integration are more difficult to formulate. Such a set of rules would have to extract the variable of integration by finding a differential (such as dx), and ensure that the integrand contains exactly one such differential. Such a set of rules would necessarily be long and complicated, since it would have to keep track of whether or not the variable of integration has been while the integrand is built. An alternative solution is to assume that the differential will always appear in one of two places: either to the far right of the integrand (as in $\int x + x^2 dx$ ) or, if the integrand consists of a single term, at the right of the numerator of the rightmost fraction (as in $\int \sin x \cos x \frac{x^2 dx}{x-1}$ ). I represent this rightmost factor containing the differential by the nonterminal *diffntl_fraction.* Rules to parse integrals, given these assumptions, are given below:

$integration \rightarrow integral\ expr\ differential$
$integration \rightarrow integral\ integrand$

$integrand \rightarrow integrand\ /\ any\_term$
$integrand \rightarrow term\ differential$
$integrand \rightarrow differential$
$integrand \rightarrow term\ diffntl\_factor$
$integrand \rightarrow diffntl\_factor$

$diffntl\_factor \rightarrow (\ integrand\ )$

where *integral* is some terminal or sequence of terminals used to represent $\int$ and possibly the limits of integration. These rules work by starting with the differential and building the integrand from right to left, and from top to bottom (as in a fraction).

## Everything Else: *lara_term*

*lara_term* represents any term which can be multiplied by any factor on either side. Most single-factor terms fall into this category, including variables, numbers, parenthesized subexpressions, and function applications with parentheses. It also includes these items when raised to exponents, and it includes function applications without parentheses that have been multiplied on the right by anything other than another function application without parentheses:

$lara\_term \rightarrow lara\_term\ lara\_factor$
$lara\_term \rightarrow lara\_factor$
$lara\_term \rightarrow larf\_term\ delimited\_nonvar\_factor$

$lara\_factor \rightarrow lara\_factor\ \wedge\ left\_delimited\_item$
$lara\_factor \rightarrow delimited\_factor$

$delimited\_factor \rightarrow delimited\_nonvar\_factor$
$delimited\_factor \rightarrow var$
$delimited\_factor \rightarrow number$

## Conclusion

I have presented a grammar that covers the most common structures used with the operators supported. It could be expanded to cover many other structures. However, I find that the way different types of mathematical expressions interact with each other tends to be somewhat unpredictable. As a result adding support for a new structure tends to require rethinking of the way others are implemented, despite efforts to keep the grammar modular.

A semantic processing phase following the parsing phase would allow some of the difficult syntactic checks to be offloaded from the parser. It could potentially also allow some of the context-sensitive areas of the language to be processed. For example, the search for the variable integration in an integrand could be perfomed by allowing any expression to be in an integrand then throwing away parses that have integrands without a unique differential in a legitimate location.

That is only one of the possible directions this method of processing formulas on a page could take. A variety of techniques could doubtless improve the balance between efficiency, accuracy, and flexibility of the procedure.

## Appendix A:The Complete Grammar

The following is a complete copy of the full grammar in one piece. Semantic rules for constructing the output are not included. Nonterminal symbols are shown in italics. Terminal symbols, some of which are described below, are shown in bold:

- **var** represents single-character identifiers. It could also be implented as a non-terminal representing other sorts of variables as well (such as subscripted variables or special symbols).
- **integral** represents an integration sign: $\int$ . This could also be implented as a nonterminal incorporating more information about the integraion (limits, for example).
- **sigma** represents $\sum$ . It could also be implemented as a nonterminal incorporating information about the bounds of the summation.

$S \to S = expr$
$S \to S < expr$
$S \to S > expr$
$S \to S \le expr$
$S \to S \ge expr$
$S \to expr$

$expr \to expr + term$
$expr \to expr - term$
$expr \to term$

$term \to term * any\_term$
$term \to term \,/\, any\_term$
$term \to any\_term$

$any\_term \to lara\_term$
$any\_term \to larf\_term$
$any\_term \to larn\_term$
$any\_term \to -any\_term$

$lara\_term \rightarrow lara\_term\ \ lara\_factor$
$lara\_term \rightarrow lara\_factor$
$lara\_term \rightarrow larf\_term\ \ delimited\_nonvar\_factor$

$lara\_factor \rightarrow lara\_factor\ \wedge\ left\_delimited\_item$
$lara\_factor \rightarrow delimited\_factor$

$delimited\_factor \rightarrow delimited\_nonvar\_factor$
$delimited\_factor \rightarrow$ **var**
$delimited\_factor \rightarrow$ **number**

$delimited\_nonvar\_factor \rightarrow (\ expr\ )$
$delimited\_nonvar\_factor \rightarrow funapp\_parens$
$delimited\_nonvar\_factor \rightarrow$ ’’sqrt’’ $(\ expr\ )$

$left\_delimited\_item \rightarrow delimited\_nonvar\_factor$
$left\_delimited\_item \rightarrow larf\_factor$
$left\_delimited\_item \rightarrow$ **var**
$left\_delimited\_item \rightarrow$ **number**
$left\_delimited\_item \rightarrow -\ left\_delimited\_item$

$larf\_term \rightarrow larf\_term\ larf\_factor$
$larf\_term \rightarrow lara\_term\ larf\_factor$
$larf\_term \rightarrow larf\_factor$

$larf\_factor \rightarrow funapp\_noparens$

$funapp\_parens \rightarrow fun\ (\ exprlist\ )$

$exprlist \rightarrow exprlist\ ,\ expr$
$exprlist \rightarrow expr$

$funapp\_noparens \rightarrow fun\ \ funarg\_noparens$
$funapp\_noparens \rightarrow fun\ \wedge\ left\_delimited\_item\ \ funarg\_noparens$

$$
\begin{aligned}
&\mathit{funarg\_noparens} \rightarrow \mathit{var\_prod} \\
&\mathit{funarg\_noparens} \rightarrow \mathit{lara\_factor} \\
&\mathit{funarg\_noparens} \rightarrow \mathit{left\_delimited\_item} \\
&\mathit{funarg\_noparens} \rightarrow - \; \mathit{funarg\_noparens}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{var\_prod} \rightarrow \mathit{var\_prod} \; \mathit{var\_power} \\
&\mathit{var\_prod} \rightarrow \mathit{var\_power}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{var\_power} \rightarrow \mathbf{var} \; \wedge \; \mathit{left\_delimited\_item} \\
&\mathit{var\_power} \rightarrow \mathbf{number} \; \wedge \; \mathit{left\_delimited\_item} \\
&\mathit{var\_power} \rightarrow \mathbf{var} \\
&\mathit{var\_power} \rightarrow \mathbf{number}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{larn\_term} \rightarrow \mathit{larf\_term} \;\; \mathit{larn\_factor} \\
&\mathit{larn\_term} \rightarrow \mathit{lara\_term} \;\; \mathit{larn\_factor} \\
&\mathit{larn\_term} \rightarrow \mathit{larn\_factor}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{larn\_factor} \rightarrow \mathit{summation} \\
&\mathit{larn\_factor} \rightarrow \mathit{integration}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{integration} \rightarrow \mathit{integral} \;\; \mathit{expr} \;\; \mathit{differential} \\
&\mathit{integration} \rightarrow \mathit{integral} \;\; \mathit{integrand}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{integrand} \rightarrow \mathit{integrand} \;\; / \;\; \mathit{any\_term} \\
&\mathit{integrand} \rightarrow \mathit{term} \;\; \mathbf{differential} \\
&\mathit{integrand} \rightarrow \mathbf{differential} \\
&\mathit{integrand} \rightarrow \mathit{term} \;\; \mathit{diffntl\_factor} \\
&\mathit{integrand} \rightarrow \mathit{diffntl\_factor}
\end{aligned}
$$

$$
\mathit{diffntl\_factor} \rightarrow ( \; \mathit{integrand} \; )
$$

$$
\mathit{summation} \rightarrow \mathbf{sigma} \;\; \mathit{any\_term}
$$

## Appendix B: Test Cases and Sample Output

The following is a group of test inputs designed to demonstrate the capabilities and limitations of the grammar. I have arbitrarily chosen the following mappings from two-dimensional mathematics to the one-dimensional string of characters that is the input to the parser:

$\int x\,dx \;\rightarrow\;$ ’’[:integral:]xdx’’

$\int_a^b xdx \;\rightarrow\;$ ’’[:integral a, b:]xdx’’

$\sum_a^b x \;\rightarrow\;$ ’’[:sum a, b:]x’’

The output is shown in LISPstyle prefix notation. Indefinite integrals are shown as:

(INTEGRAL <integrand> <variable>)

Definite integrals are shown as:

(INTEGRAL <integrand> <variable> <lower bound> <upper bound>)

Summations are shown as:

(SUM <expression> <initialization statement> <upper bound>)

Here are the test cases. Note that, where possible, the output has been simplified and redundant parses removed. I have used a scanner and simplifier written by Richard J. Fateman together with Norvig’s parser:

- "xy" => (* x y)
- "(a+b)(c+d)" => (* (+ c d) (+ a b)
- "abc/xyz" => (/ (* a b c) (* x y z))
- "f(x)" => (f x) AND (* f x)
- "c(a+b)" => (c (+ a b)) AND (* c (+ a b))
- "xsinx" => (* x (sin x))
- "tanxy" => (tan (* x y))

```
• "xsinxtanxy"                    => (* x (sin x) (tan (* x y)))

• "xsinxycos-xztan(a+bx)" => (*  (cos (* -1 x z))
                                  x
                                  (tan (+ a (* b x)))
                                  (sin (* x y)))

• "sinxycosxy/sin5xycos5xy"
                           => (/  (* (cos (* x y))
                                     (sin (* x y)))
                                  (* (sin (* 5 x y))
                                     (cos (* 5 x y))))

• "sin^2x + cos^2x = 1"   => (= (+ (EXPT (sin x) 2) (EXPT (cos x) 2)) 1)

• "sin^3xyz"              => (EXPT (sin (* x y z)) 3)
```

• "[:integral:]xdx"  (meaning $\int xdx$ )

```
                           => ((INTEGRATE x x))
```

```
• "[:integral:]x"         => NIL
                             (error, because there is no differential)
```

• "[:integral a,b:]x^2y^2tanxdx"  (meaning $\int_a^b x^2 y^2 \tan x\, dx$ )

```
                           => (INTEGRATE (* (tan x)
                                            (EXPT x 2)
                                            (EXPT y 2))
                                         x a b)
```

• "[:integral a,b:][:integral c,d:]xydxdy" (meaning $\int_a^b \int_c^d xy\, dxdy$ )

```
                          => (INTEGRATE (INTEGRATE (* x y) x c d)
                                        y
                                        a
                                        b)
```

• "[:integral:]lnx(xdx/(1+x))" (meaning $\int \ln x \frac{xdx}{1+x}$ )

```
                          => (INTEGRATE (/ (* x (ln x)) (+ 1 x))
                                        x)

• [:integral:]dxsinx      => NIL
                            (an error because dx is not on the far right
                             of the integrand)
```

- "[:integral:]((1+x+px^2+p^2q^-2cos^2x)/(1-pq))(dx/x)"

  (meaning $\int \frac{1+x+px^2+p^2q^{-2}\cos^2 x}{1-pq}\frac{dx}{x}$

```
                     =>
(INTEGRATE (/ (+ 1 x (* p (EXPT x 2))
                  (* (EXPT (cos x) 2) (EXPT q -2) (EXPT p 2)))
               (* x (+ 1 (* -1 p q))))
            x)
```

- "[:integral:]sintcosx+sinxcostdx"

```
=> (INTEGRATE (+ (* (sin x) (cos t))
                 (* (sin t) (cos x)))
              x)
```

- "[:integral:]((xydx)/z)"

```
=> (INTEGRATE (/ (* x y) z) x)
```

- "[:integral:]xydx/z" => (/ (INTEGRATE (* x y) x) z) AND (INTEGRATE (/ (* x y) z) x)
  (division really requires parentheses to work predictably, especially where integrals are involved)

- "[:integral:]sinxy(sinxcosxdx)/x"

```
=>
(/ (INTEGRATE (* (|sin| |x|) (|sin| (* |x| |y|)) (|cos| |x|)) |x|) |x|)
AND (INTEGRATE (/ (* (|sin| |x|) (|sin| (* |x| |y|)) (|cos| |x|)) |x|) |x|)
```

- "n[:sum i=1, n:]3i^2" (meaning $n\sum_{i=1}^{n} 3i^2$ }

```
=>   ((* n (SUM (* 3 (EXPT i 2)) (= i 1) n)))
```

- "[:sum i=0,10:][:sum j=0,10:]ij" (meaning $\sum_{i=0}^{10}\sum_{j=0}^{10} ij$ )

```
=>(SUM (SUM (* i j) (= j 0) 10) (= i 0) 10)
```

- "[:sum i=0, n:]i+1" => (+ 1 (SUM i (= i 0) n))
  (note that summation takes precedence over **+** and **-**)

- "[:sum i=0, n:]3i^3+[:sum j=0, m:]2j^2"

```
=>     (+ (SUM (* 2 (EXPT |j| 2)) (= |j| 0) |m|)
          (SUM (* 3 (EXPT |i| 3)) (= |i| 0) |n|)
```

1 Aho, Alfred, Ravi Sethi, and Jefferey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Massachusetts, 1986
2 Norvig, Peter, *Artificial Intelligence Programming: Case Studies in Common Lisp*, Morgan Kaufmann Publishers, San Mateo, California, 1992